Red-black trees

CS 146 - Spring 2017

Today

- Recurrence equations wrap up
- Red-black trees

What’s the time complexity?

```
void foo(int n) {
    if (n <= 1) return;
    for (int i = 0; i < n*n; i++)
        print “woof”;
    foo(n/3);
    foo(n/3);
    foo(n/3);
    foo(n/3);
}
```

Solving a recurrence

- total time =
  - sum from 1 to the number of tree levels of…
  - where at level i,
    - #nodes at this level, times,
    - the time taken to complete function call with given input size at this level

Recurrence equation

- an equation where the **unknown is a function** T(n)
- as in algebra, there are multiple approaches to solve:
  - guess and check
  - **solve step by step** ← **tree method**
  - plug numbers into a formula ← master theorem

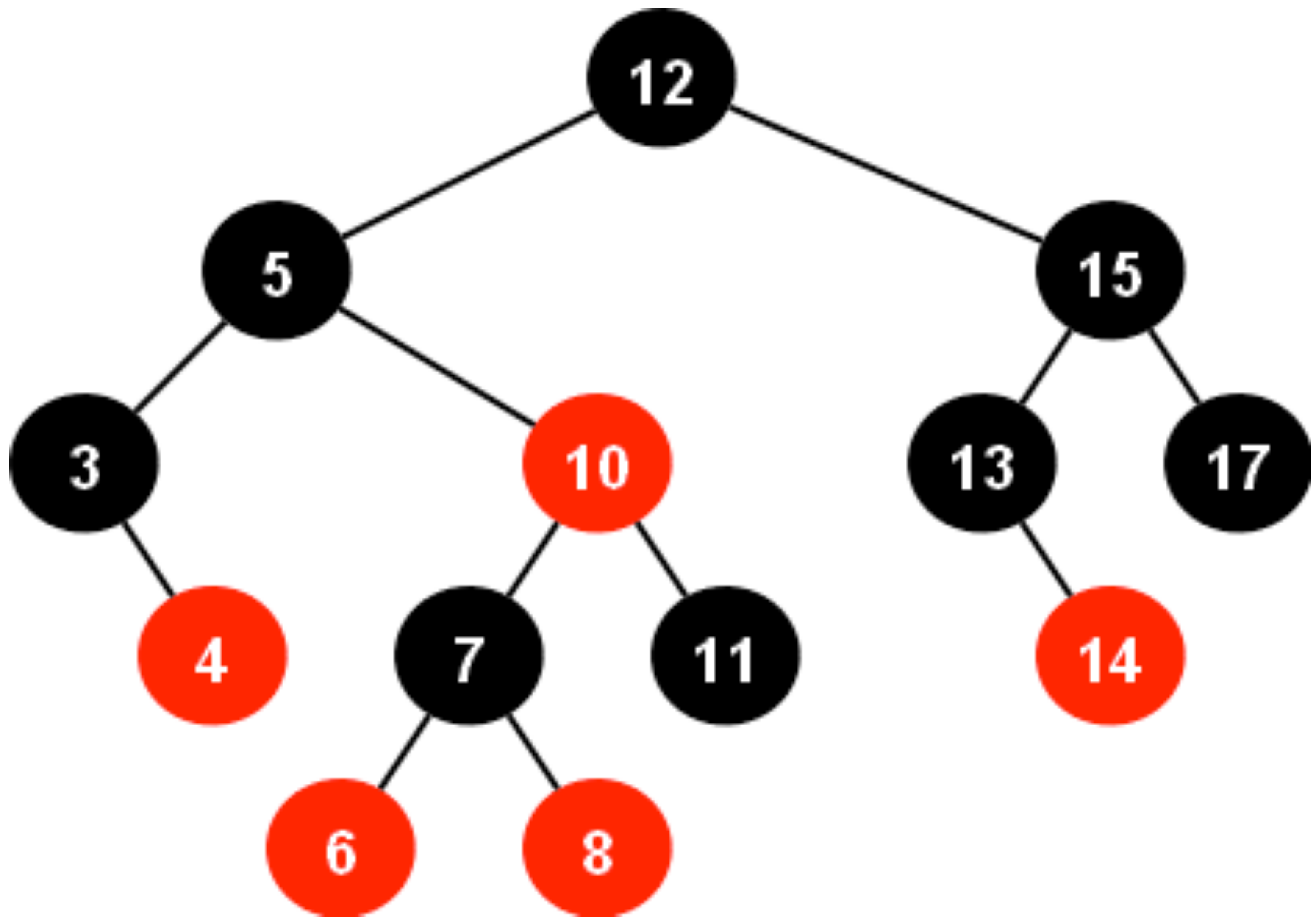
know this!

Red-black tree properties

0. binary search tree property
1. all nodes are red or black
2. root is black
3. leaves (nil) are black
4. if node red, both children are black
5. black height property: all paths from root to leaf have same number of black nodes

black height
= height of tree with red nodes removed
= (#black nodes per path) - 1

Is this a red-black tree?

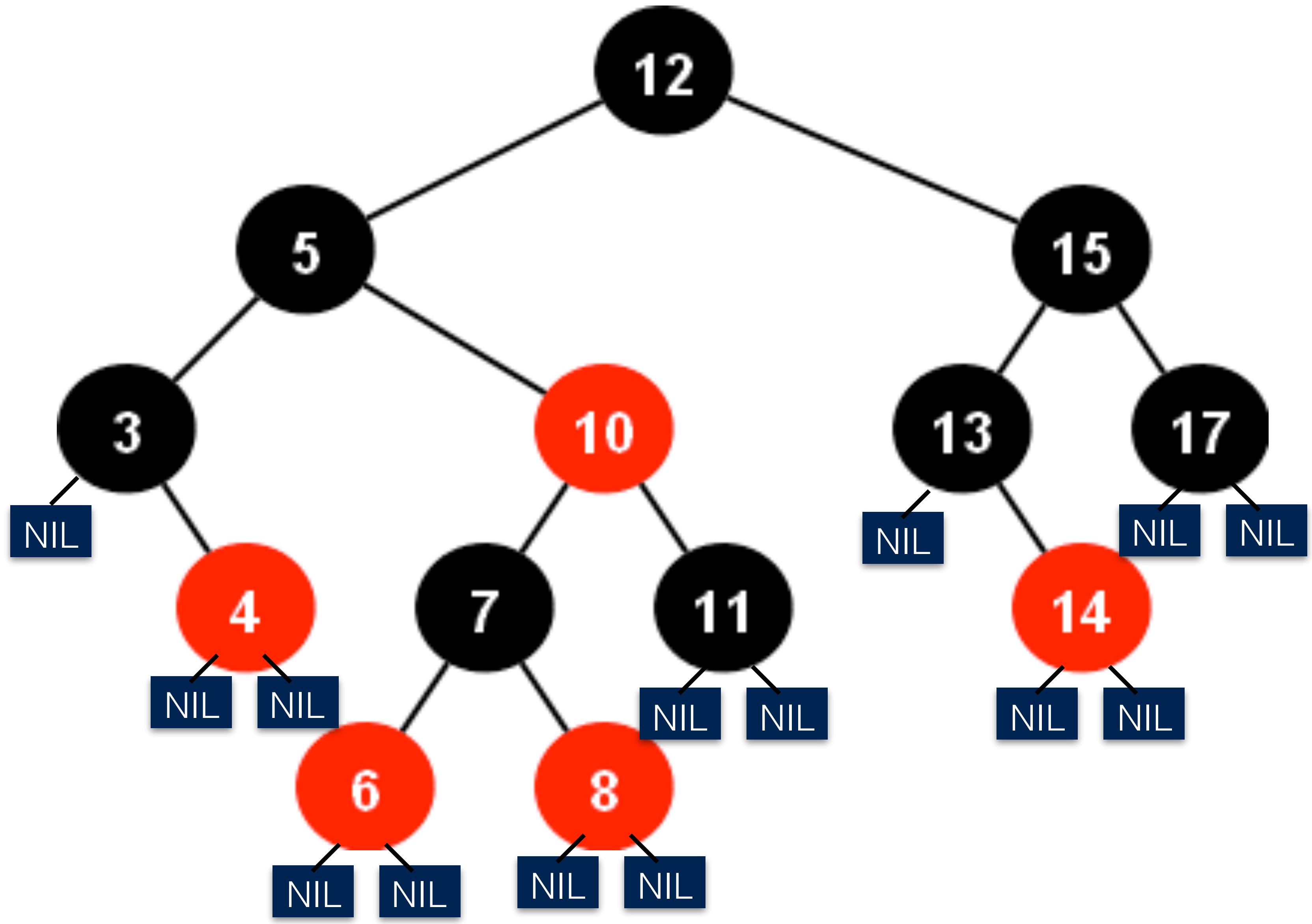


what's the height?

what's the black height?

http://cs.lmu.edu/~ray/notes/redblacktrees/

Is this a red-black tree?



what's the height?

what's the black height?

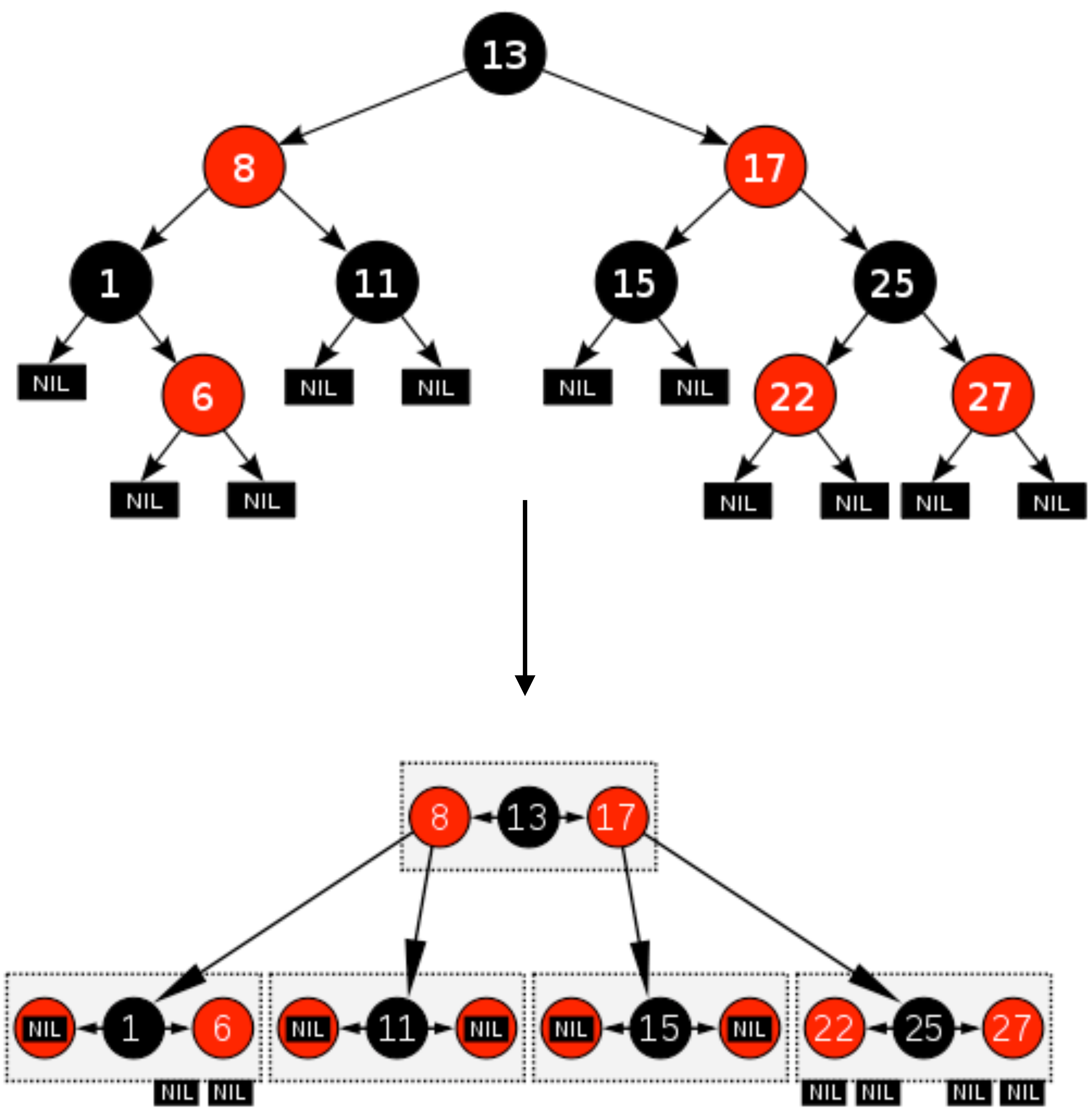
http://cs.lmu.edu/~ray/notes/redblacktrees/

Red-black tree properties

0. binary search tree property
1. all nodes are red or black
2. root is black
3. leaves(nil) are black
4. if node red, both children are black
5. black height property: all paths from root to leaf have same number of black nodes

How tall can a RB tree with 5 internal nodes be?

a red-black tree with n internal nodes has height at most 2 log(n + 1) - proof:

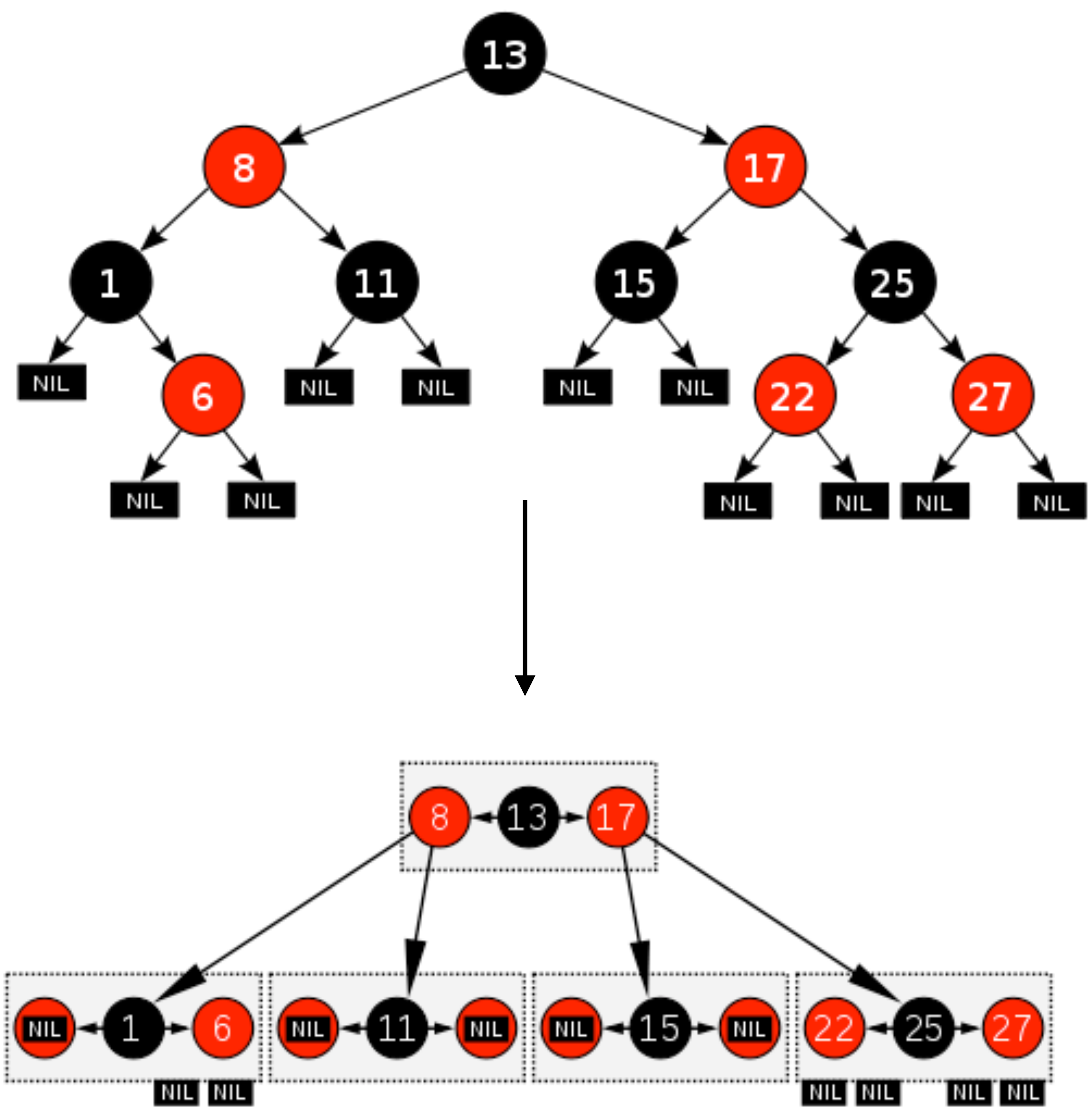


convert RB tree
to a 2-3-4 tree

by moving red nodes
into their parent black
node

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

a red-black tree with n internal nodes has height at most 2 log(n + 1) - proof:



notice:

black height of RB tree

=

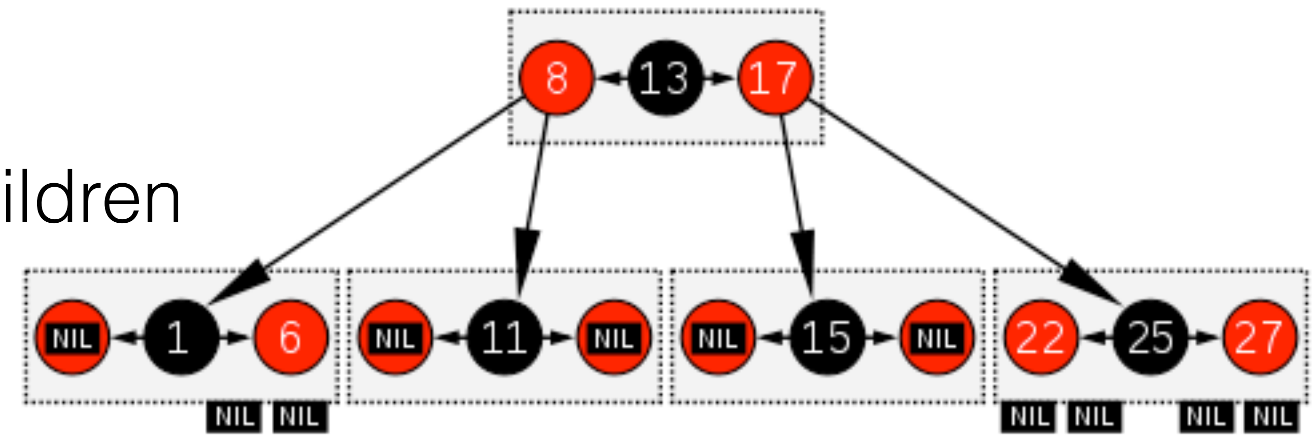
height of 2-3-4 tree

=

h' (let's call this value h')

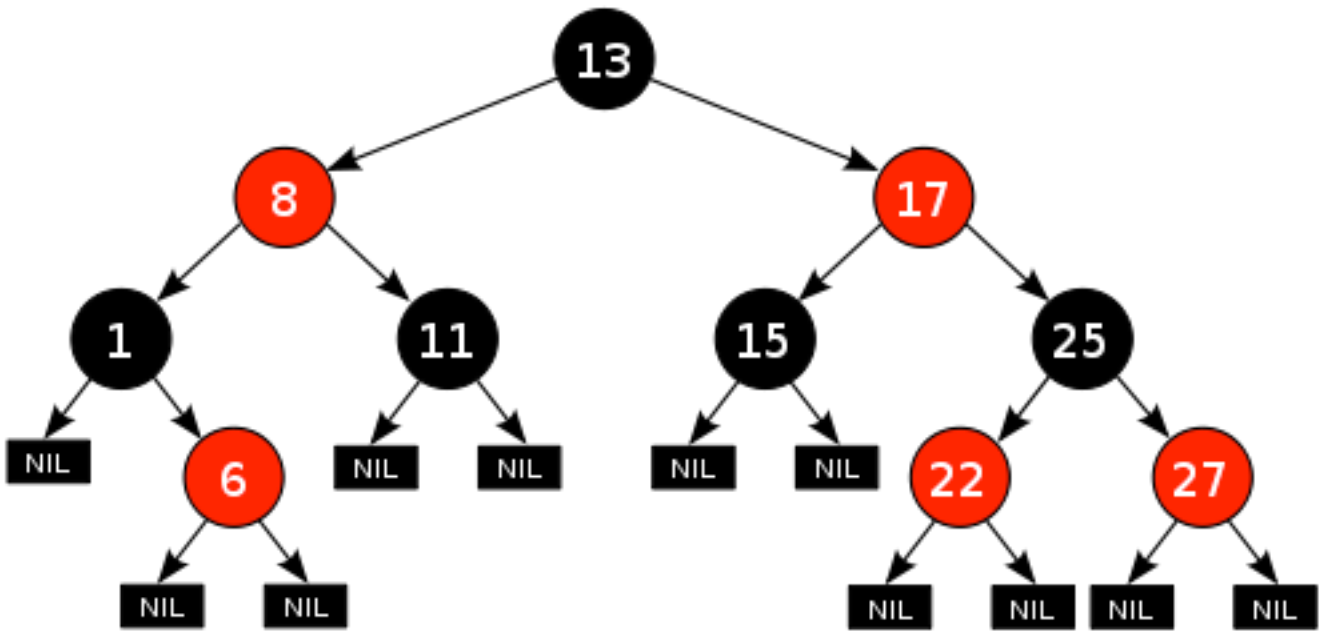
https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

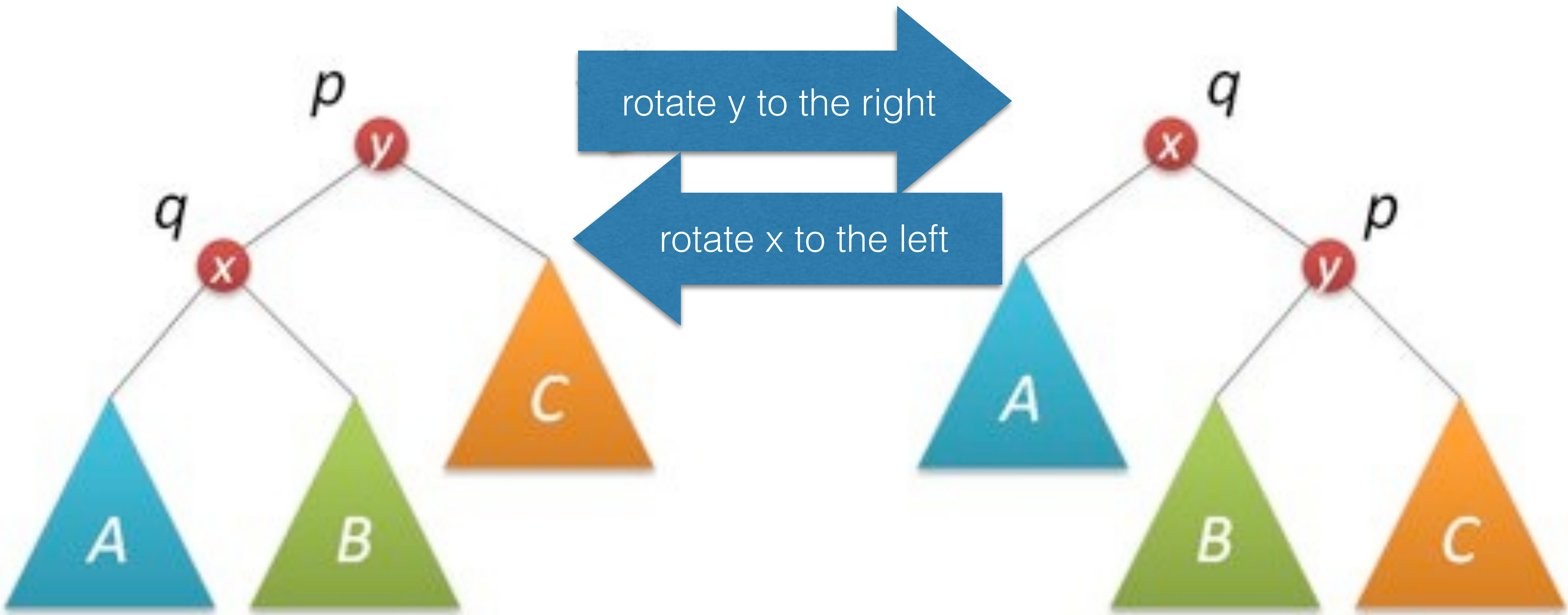
a red-black tree with n internal nodes has height at most 2 log(n + 1) - proof:



- every internal node has 2–4 children
- every leaf has same height, the black height
- number of leaves = n + 1 in 2–3–4 tree
- 2^{h'} <= number of leaves
- 2^{h'} <= n+1
- (1/2)h <= h' (at most half are red)

corollary:
all queries are 2 log(n+1)
or O(log n) time



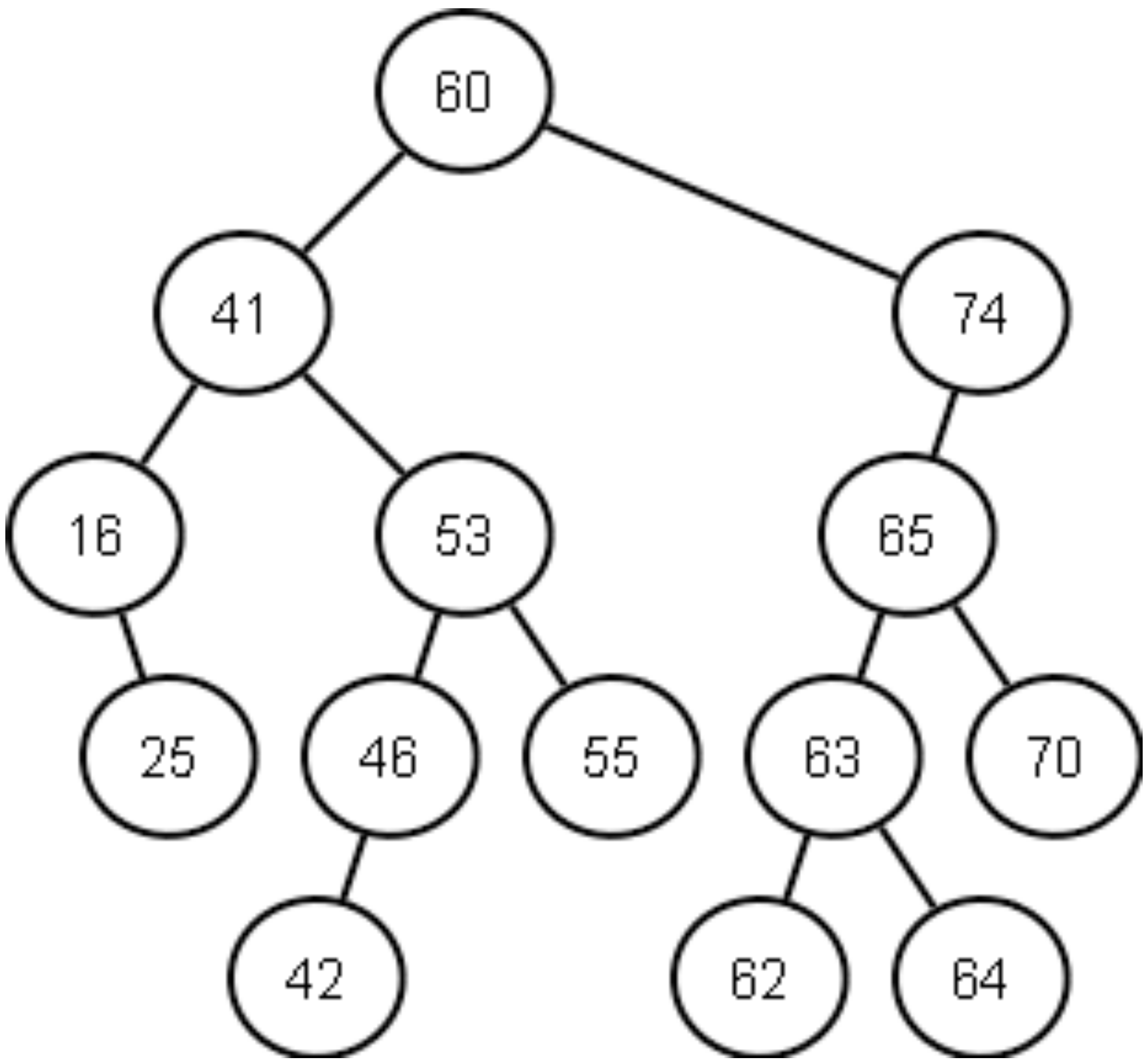


$A < x < B < y < C$

BST property is maintained

https://kukuruku.co/post/avl-trees/

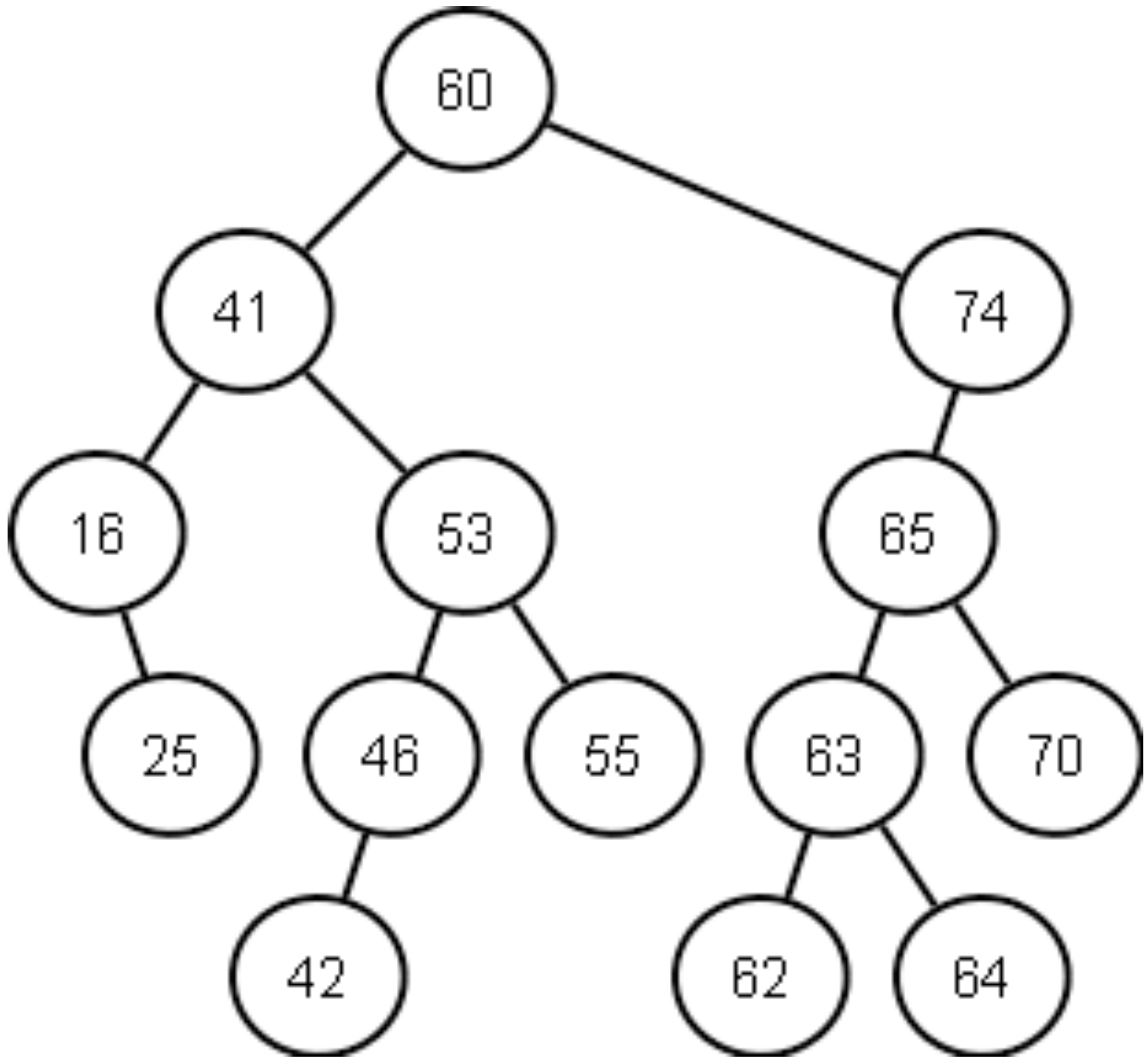
Example rotation



- what is the result of rotating 41 to the left?
- what is the result of rotating 41 to the right?

https://dzone.com/articles/java-how-create-binary-search

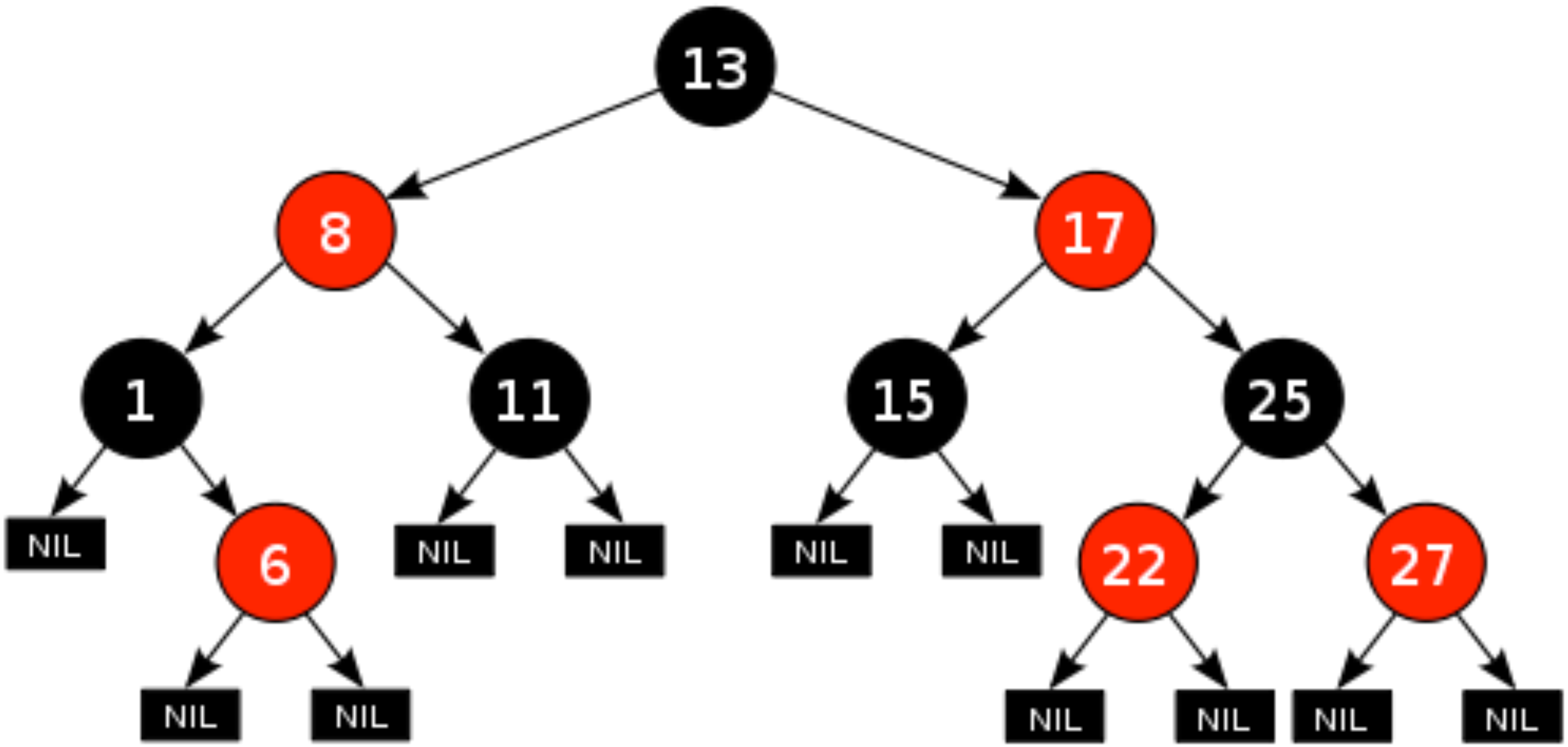
Example rotation



- what's a node that should be rotated to improve balance?

https://dzone.com/articles/java-how-create-binary-search

How to insert into a RB-tree?



insert 12, what color would you assign it?

insert 5, what color would you assign it?

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

RB-tree insertion: the idea

- do a regular BST insertion
- color the new node red (to preserve black height)

- potential violation: parent might be red!
- idea: move violation up the tree by recoloring and rotating until the violation is gone

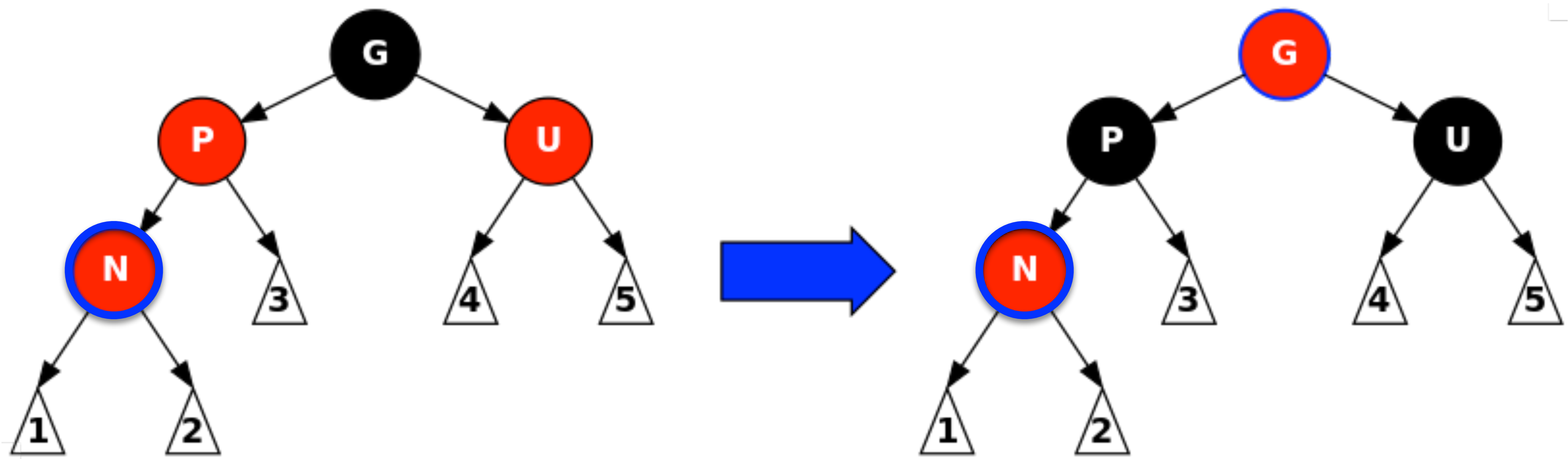
Insertion fix: case 1

current node's parent is red and a left child

case A

current node's **uncle is red**

case A1



-> recolor parent, uncle and grandparent
fix grandpa

Note: case numbers match textbook, not Wikipedia

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

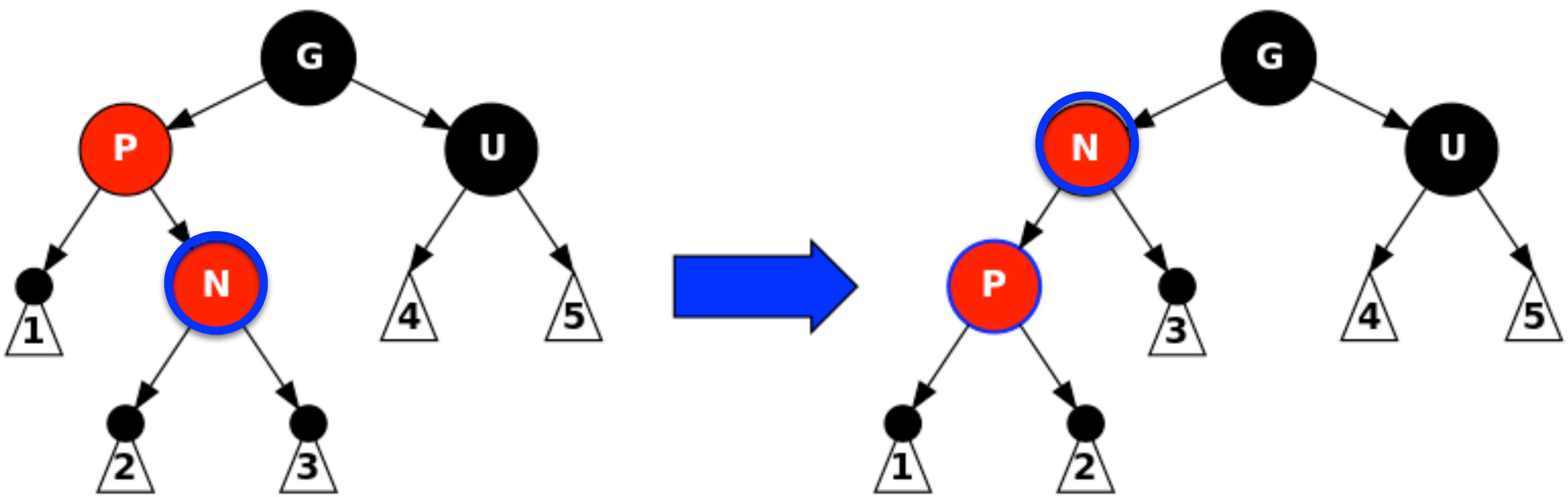
Insertion fix: case 2

current node's parent is red and a left child

case A

current node's **uncle is black**
and **current node is a right child**

case A2



-> rotate parent left

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

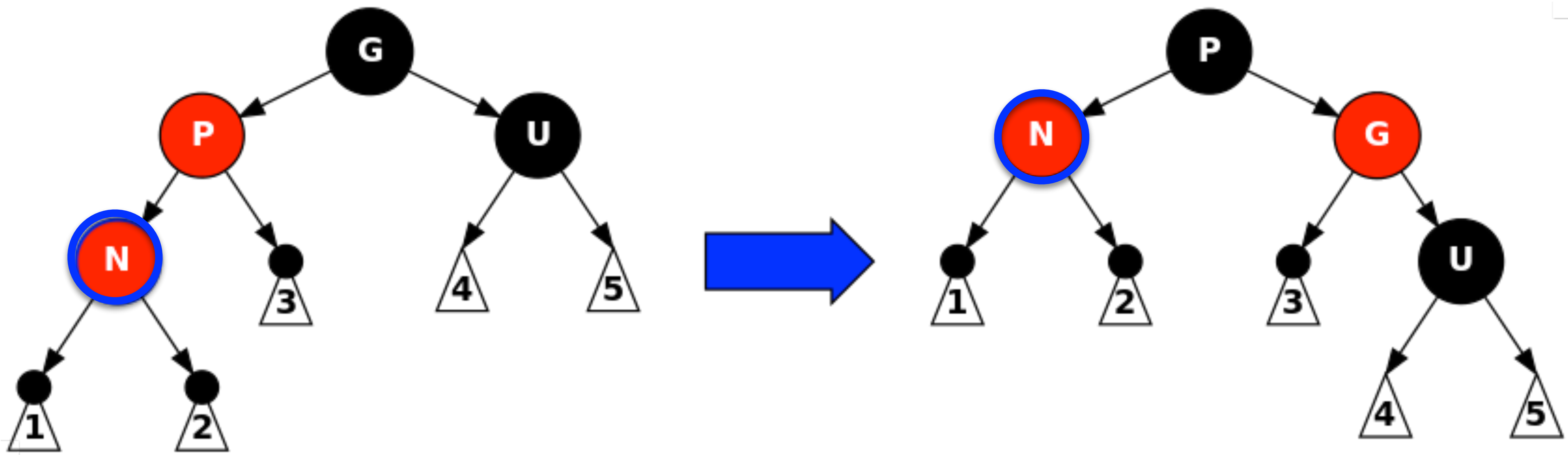
Insertion fix: case 3

current node's parent is red and a left child

case A

current node's **uncle is black**
and **current node is a left child**

case A3



-> recolor parent and grandpa
and rotate grandpa right

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

RB-insert(x)

- BST-insert(x)
- color x red
- while x’s parent is red (violation!)
  - if x’s parent is left child: (case A)
    - if x’s uncle is red: (case 1)
    - else
      - if x is right child: (case 2)
        - turn x into left child (do case 2)
      - (case 3)
  - else (case B)
    - same as A but reverse “left” and “right”
- color root black

RB-insert(x)

- BST-insert(x)
- color x red
- while x's parent is red (violation!)
  - if x's parent is left child: (case A)
    - if x's uncle is red: (case 1)
    - else
      - if x is right child: (case 2)
        - turn x into left child (do case 2)
      - (case 3)
  - else (case B)
    - same as A but reverse "left" and "right"
- color root black

alg correctness

define loop invariant:
BST property (0), 1,3 and
black-height property (5) met

by induction, loop invariant
holds at every beginning of an
iteration of the while loop

also, at end of the loop,
property 4 are met

at end, property 2 is met

at end, all properties are met!

RB-insert(x)

- BST-insert(x)
- color x red
- while x's parent is red (violation!)
  - if x's parent is left child: (case A)
    - if x's uncle is red: (case 1)
    - else
      - if x is right child: (case 2)
        - turn x into left child (do case 2)
      - (case 3)
  - else (case B)
    - same as A but reverse "left" and "right"
- color root black

up to 2 log n

propagate violation up the tree, up to 2 log n times

at most once
(reduces to case 3)

at most once (terminal case)

total: at most O(log n)

Why we like RB-trees

AVL tree	**RB tree**
log n height (perfectly balanced)	2 log n height (roughly balanced)
after insertion rebalancing via **≤ log n rotations**	after insertion rebalancing via **≤ 2 rotations** and **≤ 2 log n recoloring**
easier to implement (correctly)	harder to implement (correctly)

Big picture: Designing a data structure

- **define an invariant** of the data structure
  - typically a desirable property
  - the desirable property may ensure fast lookup
  - example: BST property -> ensures that lookup only explore nodes on one root-to-leaf path rather than the whole tree
  - example: RB tree properties -> ensures that the tree is roughly balanced, so together with BST property, ensures O(log n) search time.

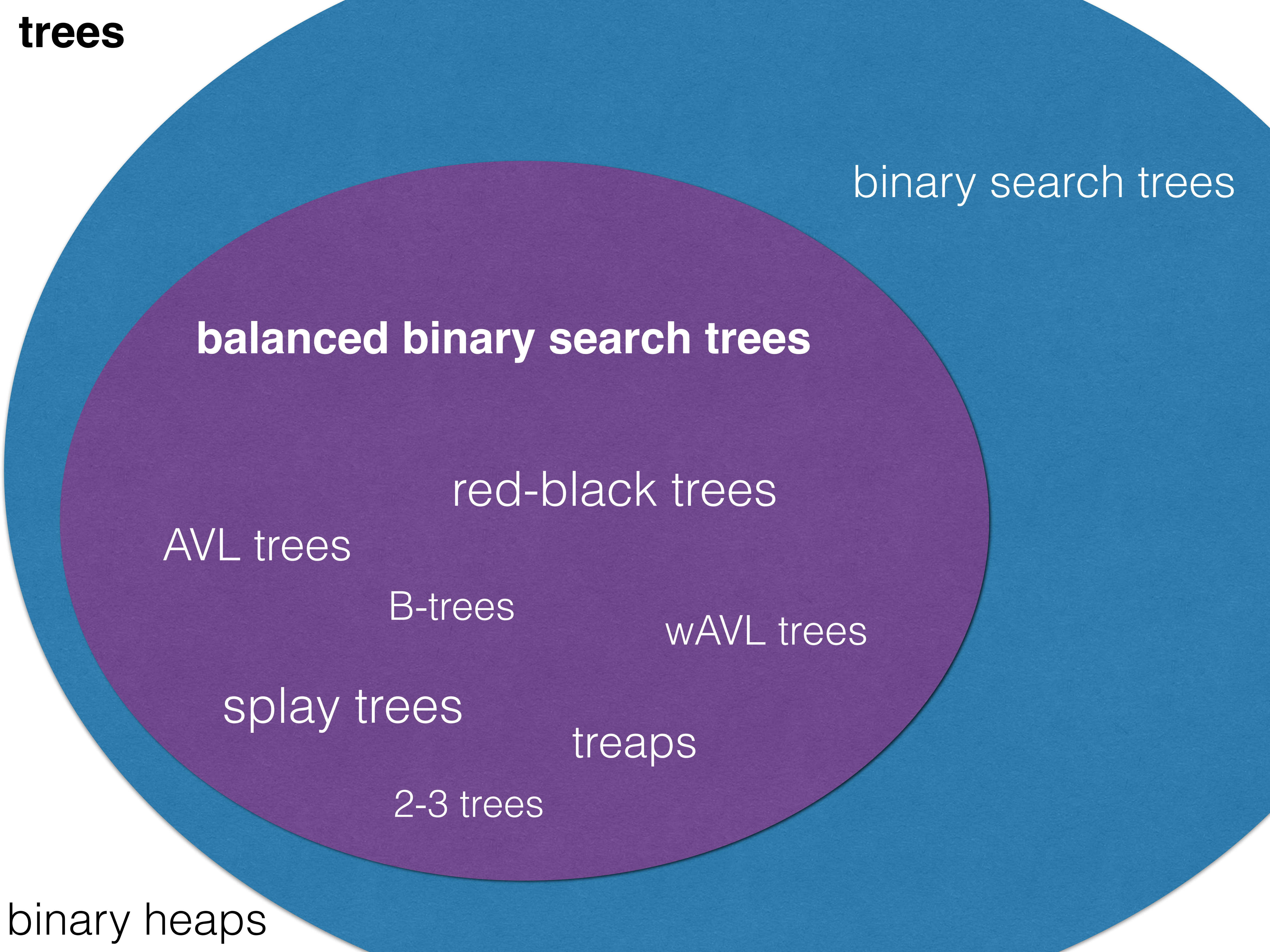
Big picture: Designing a data structure

- an operation on the data structure can potentially invalidate the invariant
  - example: insertion, deletion
- **find an (efficient) algorithm which will perform the operation and still maintain that invariant**
  - example: BST insertion chooses insert location so that BST property is maintained
  - example: after insertion into red-black tree, a series of carefully chosen rotations re-establish RB properties

Recap: Designing a data structure

- to ensure **efficiency**: define an invariant of the data structure

- to ensure the **invariant**: define data structure operations so they always maintain or re-establish the invariant


trees
binary search trees
balanced binary search trees
red-black trees
AVL trees
B-trees
wAVL trees
splay trees
treaps
2-3 trees
binary heaps