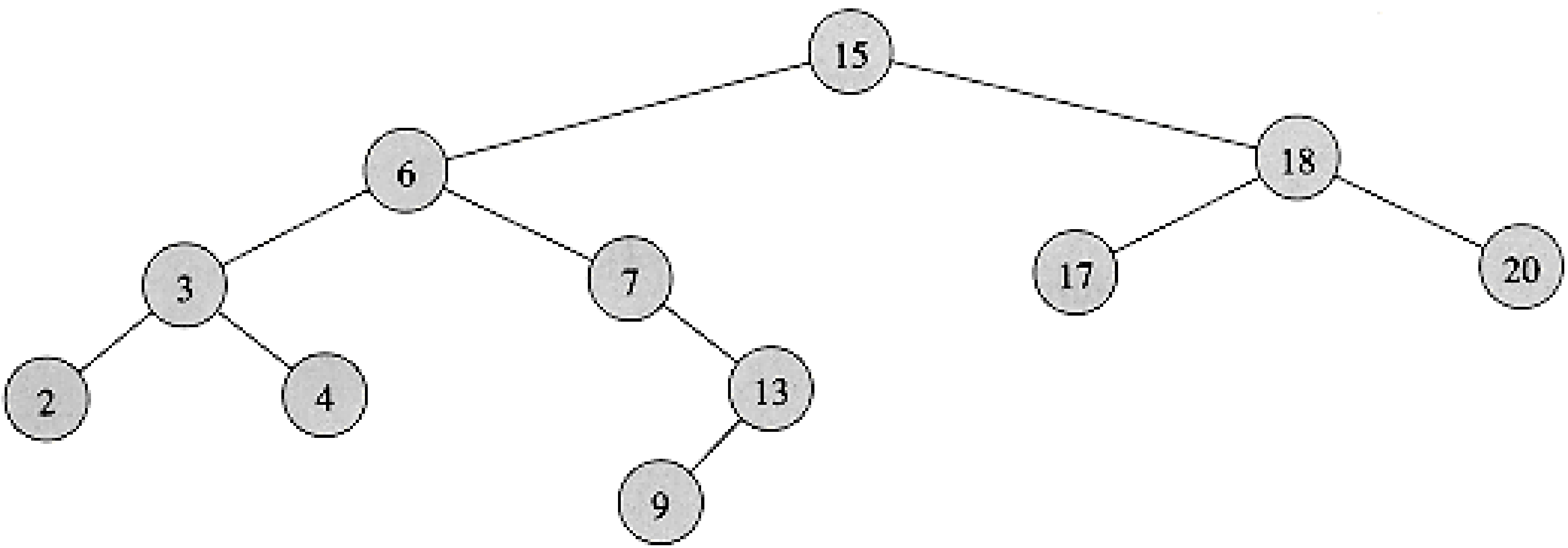
Binary search tree



Binary search tree

A sorted container

Other terms: Sorted binary tree

By default (for us)

- Elements are less than comparable

Other choices:

- Frequent: elements are identified by a key. Keys are less than comparable
- ….

    -> see sorted map, priority queue

Binary search tree (BST)

BST - a binary tree that has the **BST Property**

BST Property

For each node **x** :

- if **y** is a node in the left subtree of **x** , then info(**y**) <= info(**x**)
- if **z** is a node in the right subtree of **x** , then info(**x**) <= info(**z**)

Property:

Inorder traversal → ascending order of elements

- can be used to implement a sorting algorithm. insert all the values we wish to sort into a new BST

  traverse it in order

BST – definitions **(equivalent)**

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key(y) <= key(x). If y is a node in right subtree of x, then key(x) <= key(y)

Cormen

A binary tree where every node's left subtree has keys less than the node's key, and every right subtree has keys greater than the node's key.

xlinux.nist.gov/dads/

Binary search tree

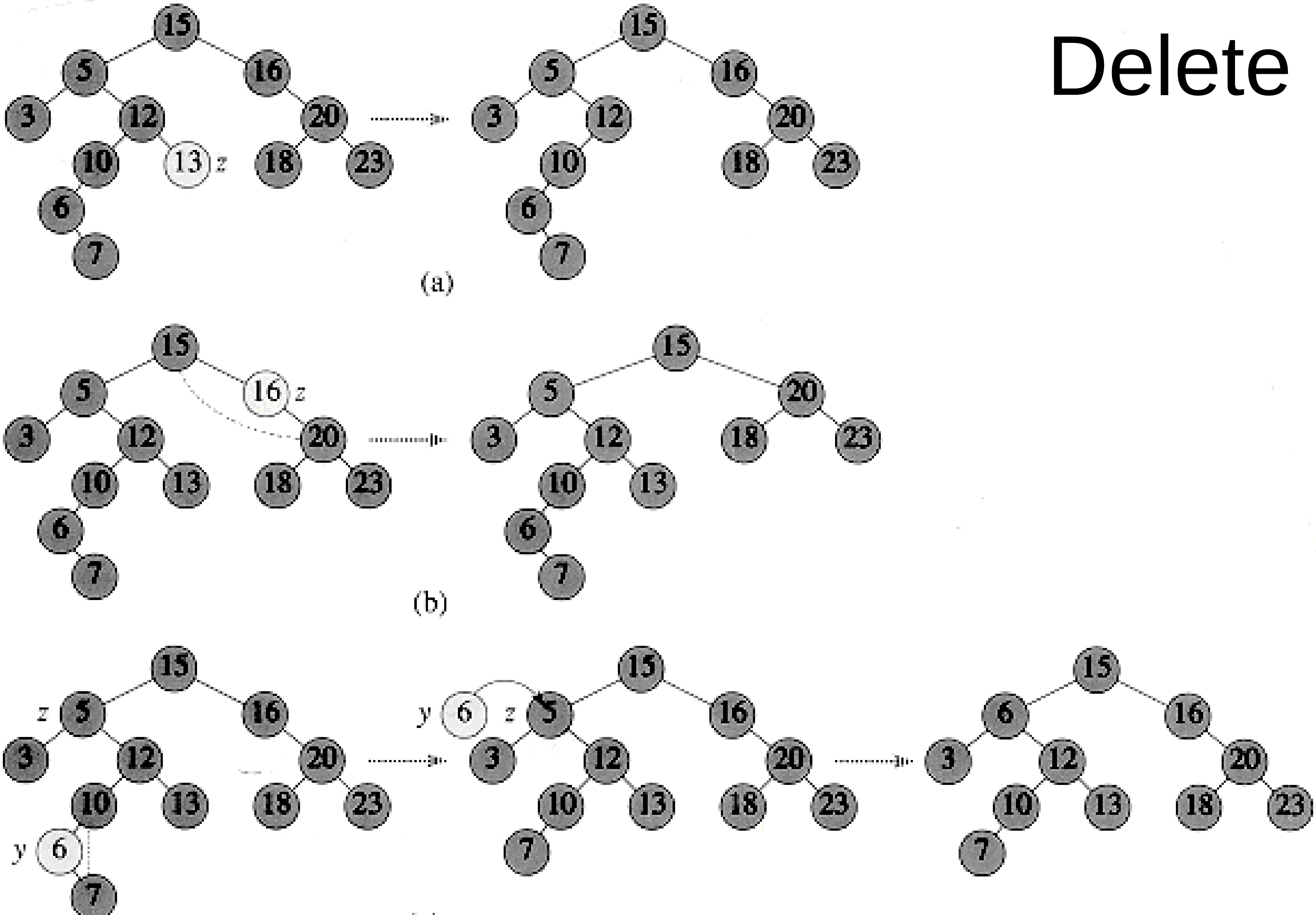
BST

	Average	Worst case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

.

```
TreeNode: record
   info: TComparable
   left:       ^TreeNode
   right:      ^TreeNode
   parent:  ^TreeNode
end
```

Delete



(a)

(b)

(c)

Delete

Delete a leaf

Deleting a node with one child:

delete it and replace it with its child.

Deleting a node with two children:

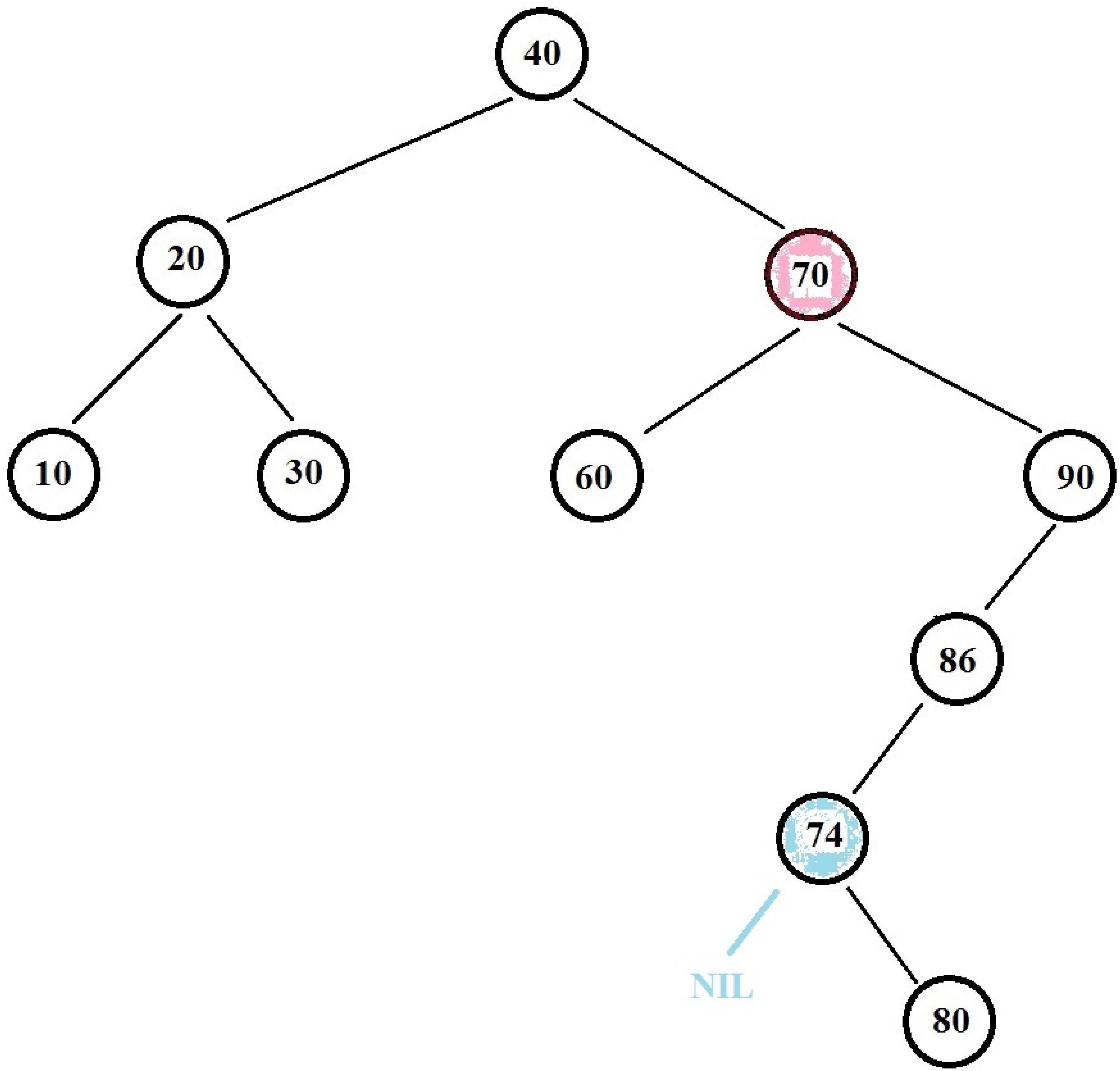
replace value with (either)

– its in-order successor

or

– its in-order predecessor

and then delete the succ. or pred.


40
20
70
10
30
60
90
86
74
NIL
80

Subalg. delete(T, z)

@ collect information about nodes involved

  z – node to be (logically) deleted

  get y - the node to be really deleted (z or its successor)

  get x – the child of y (NIL if no children)

  get q – the parent of y (NIL if no parent)

@ copy information from y to z

@ remake link over the node y to delete

  from child to parent parent(x) <- q (if child exists)

  from parent to child

    if parent of y does not exist: update root node value

    else link from q to x

@delete y

(Nearly) Balanced BST tree

(nearly) balanced: no leaf is much farther away from the root than any other leaf.

Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

Self-balancing binary search tree :

- a binary search tree
- & keep it balanced

Popular self-balancing BSTree

- red-black tree
- AVL tree

no leaf is more than a certain amount farther from the root than any other

(Height-)Balanced tree

Height-balanced tree :

A tree whose subtrees differ in height by no more than one and the subtrees are height-balanced, too.

An empty tree is height-balanced.

`http://xlinux.nist.gov/dads/`

Height-balanced tree

- AVL tree

BST in Java.util and C++ STL

Java.util

- TreeMap

  a Red-Black tree based implementation

- TreeSet

  implementation based on a TreeMap

C++ STL

- map, multimap

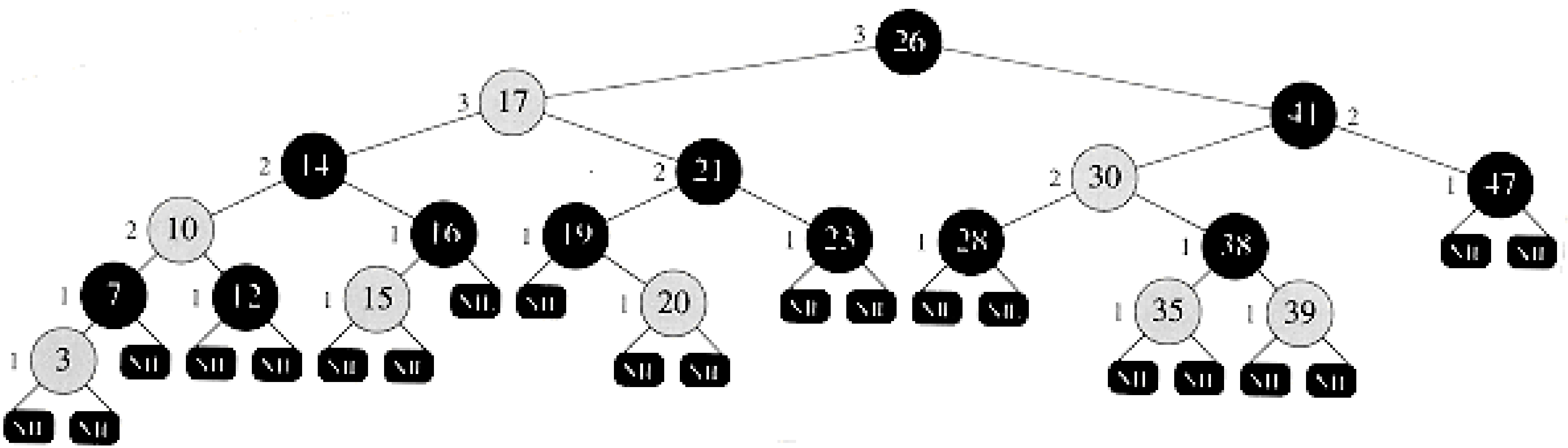
  are typically implemented as binary search trees

  `www.cplusplus.com`

  maps are usually implemented as red-black trees

  `en.cppreference.com/w/cpp/container/map`

Red-black tree



Red-Black tree

A red-black tree is a binary search tree which satisfies:

1. Every node is either red or black.
2. The root is black.
3. Every leaf NIL is considered black.
4. A red node have two black children.
5. For each node:

   all paths from the node to descendant leaves contain the same number of black nodes.

This rule is sometimes omitted, since the root can always be changed from red to black

Red-Black tree

- one extra information per node:

  its ***color***, which can be either RED or BLACK.

- **black-height** of a node x: bh(x)

  the number of black nodes on any path from x to a leaf node
- **black-height of a red-black tree**: the black-height of its root.

Red-Black tree

Lemma

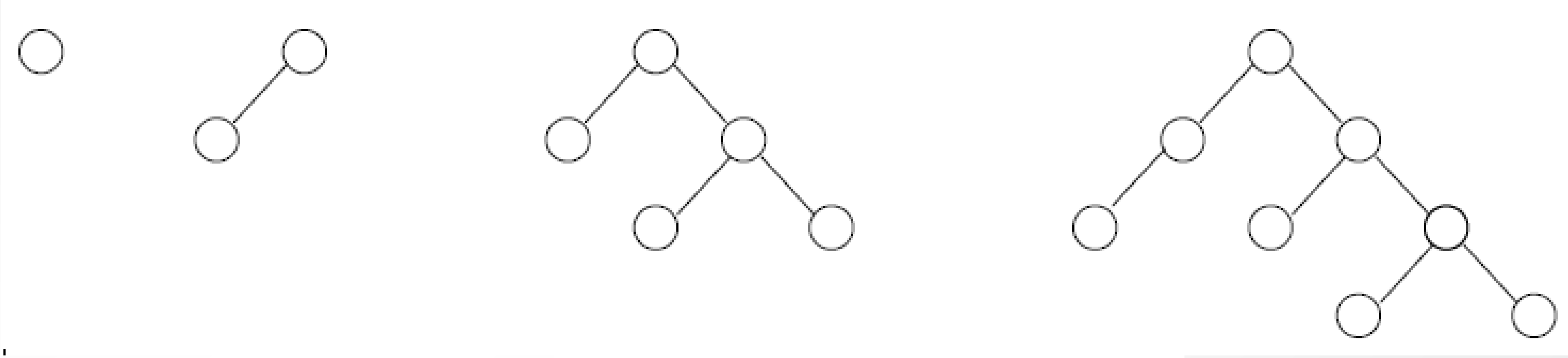
A red-black tree with n internal nodes has height at most $2*\log_2(n + 1)$.

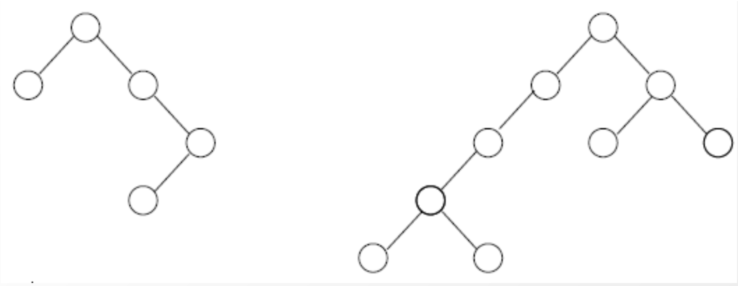
AVL tree

An AVL tree

is a binary search tree which satisfies:

the heights of the two subtrees of any node differ by at most one





?

AVL tree

Suppose we have $\boldsymbol{n}$ nodes in an AVL tree of height $\boldsymbol{h}$.

$$\boldsymbol{h} \sim < 1.44 * \log_2 \boldsymbol{n}$$

Balanced trees

Operations

1. Search is *O(log n)* since the trees are always balanced.

2. Insertion and deletions are also *O(log n)*

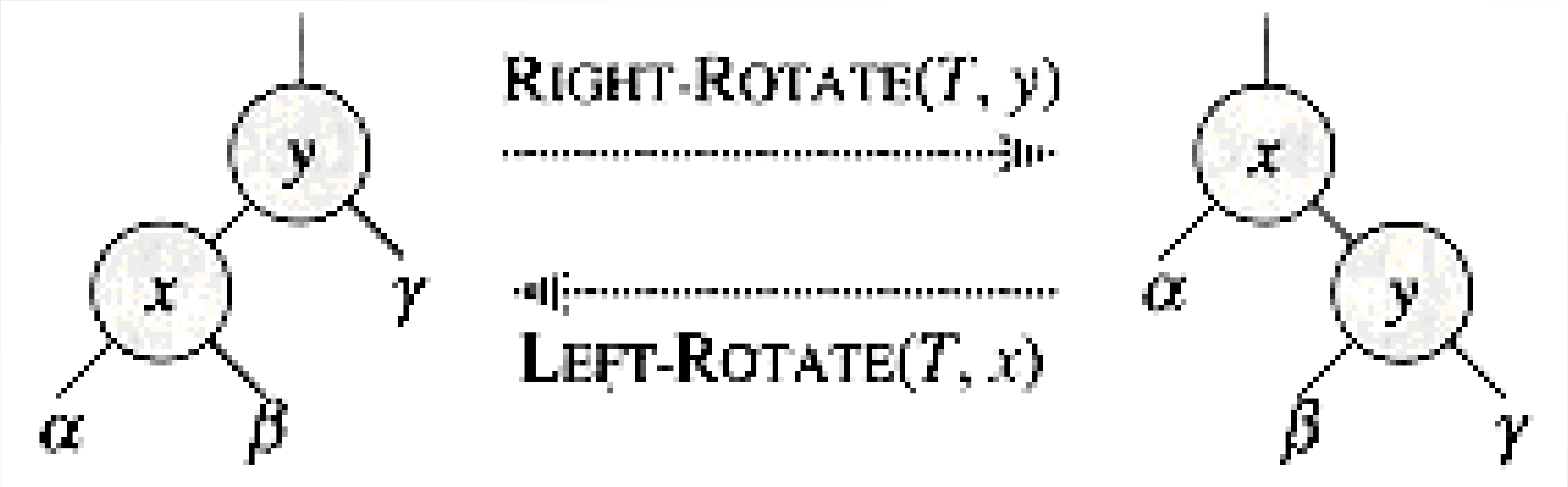
3. *Balancing* adds *a constant factor* to the speed of insertion/deletion.

- Difficult to program; more space for balance factor.
- Asymptotically faster but rebalancing costs time.

Remark:

Use left-rotate / right-rotate for rebalance

Rotation



DS

TreeNode:

info: TComparable

left: ^TreeNode

right: ^TreeNode

parent:^TreeNode

end