Binary Trees

Lecture #3 of Algorithms & Data Structures (Module 7)

Rom Langerak

E-mail: `r.langerak@utwente.nl`

February 11, 2019

© GfH&JPK&RL

Plan for today

- Binary trees, an important class of objects
  - what are they, and what makes them important
  - what do typical methods on trees look like
  - how do we count steps in these recursive methods
- Binary Search Trees
  - what is a binary search tree?
  - how to find, insert and delete elements of a binary search tree?
  - rotations for balancing trees

Binary trees – 1

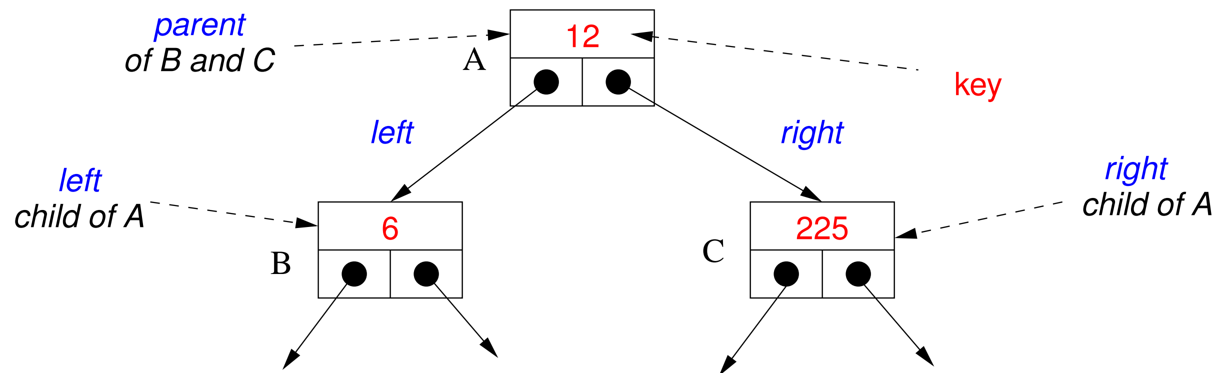
Storage structures: arrays, lists, **(binary) trees**, ...

Typical properties

- An array allows random access, a list and a tree have a single starting point.
- A tree has a good ratio between the number of elements and the average distance to the starting point, a list has not.

Binary trees – 2

A binary tree: each data object has two pointers (left and right) to successive objects, this yields a storage structure like:



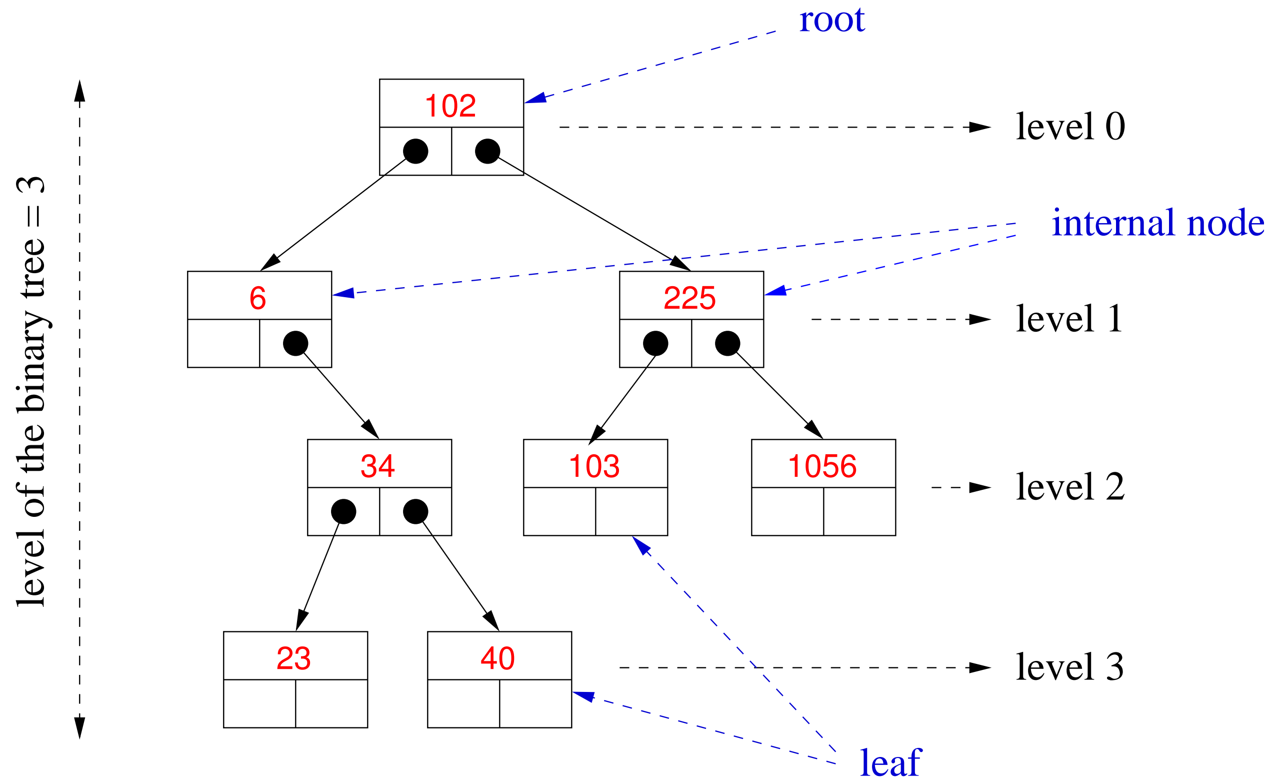
Binary trees – 2

A binary tree is a set of nodes that is empty *or* else satisfies

- there is a single distinguished node called root
- remaining nodes are divided into a left sub-tree and a right sub-tree; we also speak of children instead of sub-trees (and use parent for the node itself).
- the left and right sub-tree of a leaf are empty
- the in-degree of a node is 1 (except the root); its out-degree is maximally 2

Binary trees – 4

- The level (or depth) of a node is its distance to the root
- The level of a tree is the maximum level of its leafs



The benefits of binary trees – 1

Suppose you want to hold 30 data objects:

- Level 0 (root) holds 1 data object
- Level 1 holds 2 data objects total: 3
- Level 2 holds 4 data objects total: 7
- Level 3 holds 8 data objects total: 15
- Level 4 holds 16 data objects total: 31
- a data object can be traced in 5 steps (instead of 31) if stored handy

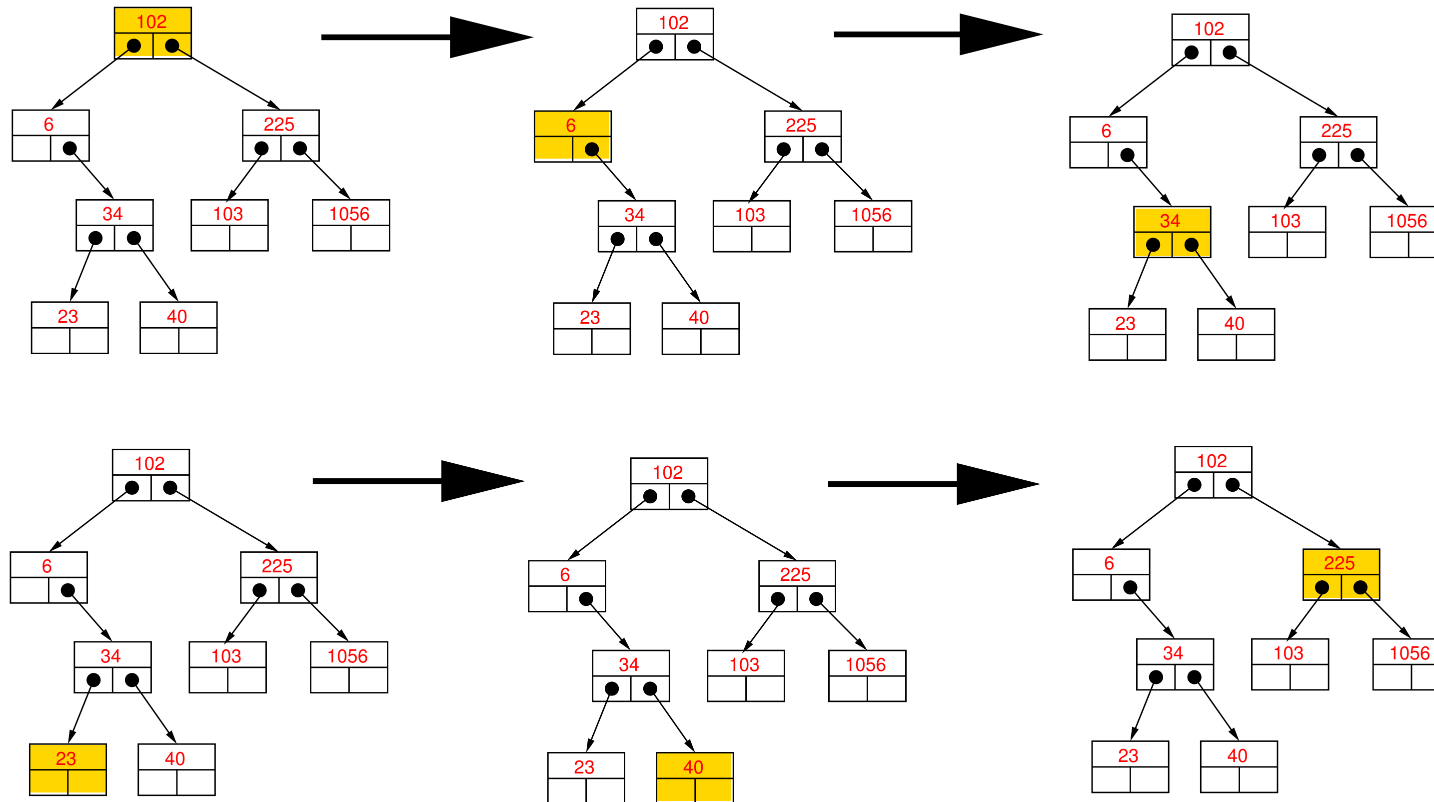
The benefits of binary trees – 2

Some facts about binary trees:

- there are at most 2^d nodes at level d
- a binary tree of level h contains at most $2^{h+1} - 1$ nodes
- a binary tree with n nodes has level at least $\lceil \log(n+1) \rceil - 1$

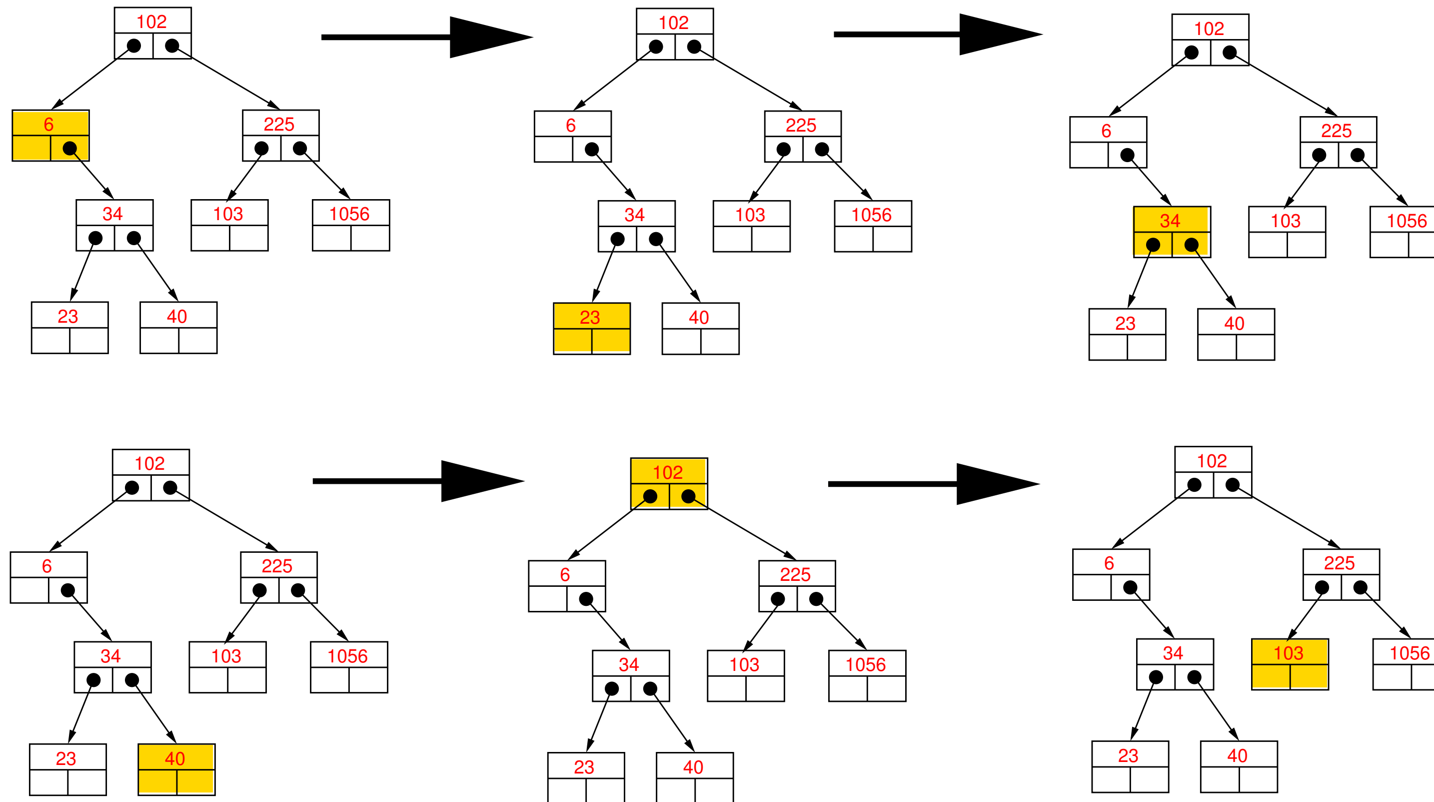
© GfH&JPK&RL

Preorder traversal of a binary tree



parent, left subtree, right subtree

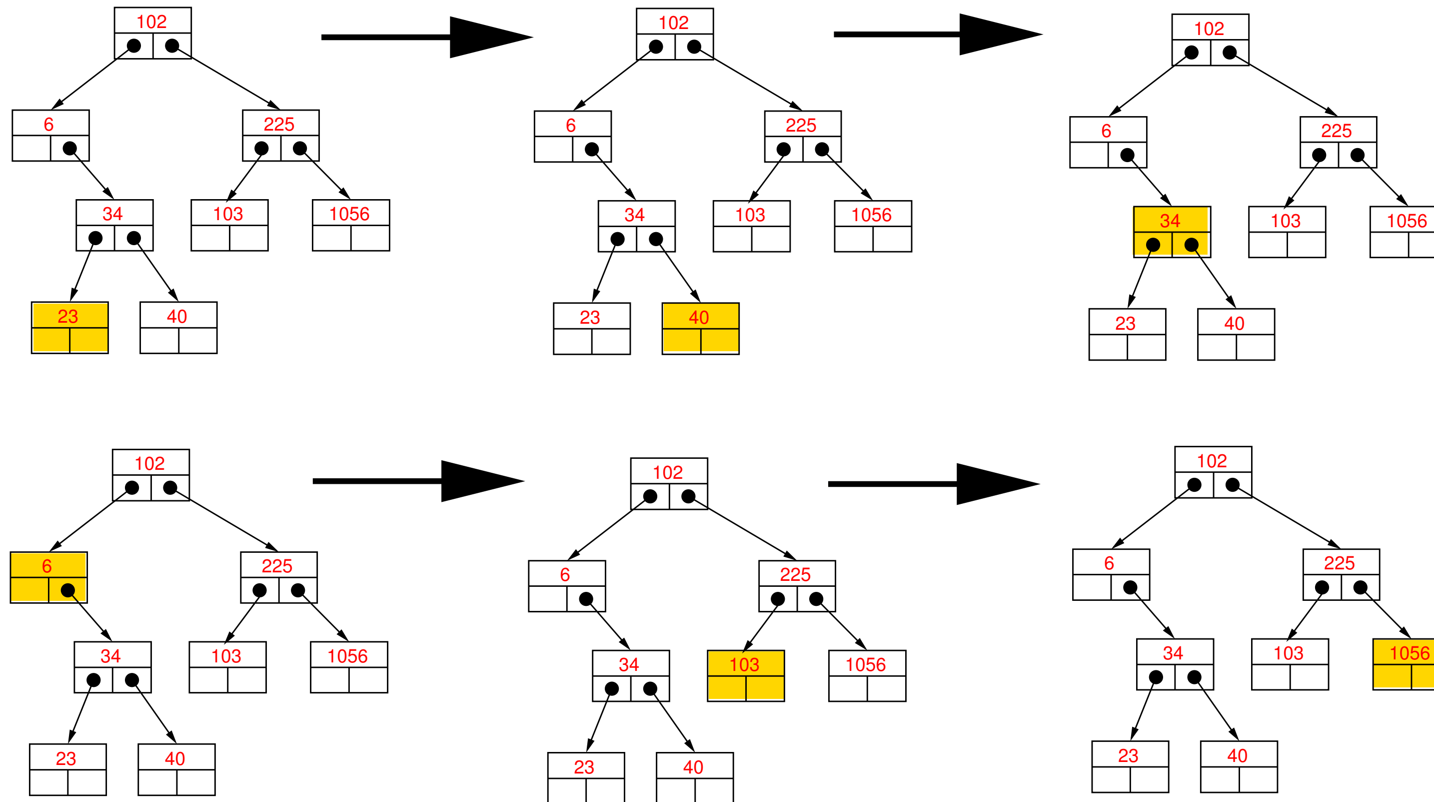
Inorder traversal of a binary tree



left subtree, parent, right subtree

© GfH&JPK&RL

Postorder traversal of a binary tree



left subtree, right subtree, parent

An algorithm – preparation

We have binary trees with integer keys that are sorted *preorder*.

We have **node** objects with the following features:

- *key*, an integer key value,
- *left*, points to the left child,
- *right*, points to the right child

We want to know if the tree has two nodes with keys that are immediate successors (n and $n+1$).

We design two methods: *his*, for “has immediate successors”, and *fism*, for “found immediate successors or maximum”. *fism* is an auxiliary for *his*

© GfH&JPK&RL

An algorithm – 1

```
def his(root):
    if root==null:
        return False
    else:
        last,found=fism(root.key,root.left)
        if found:
            return True
        else:
            last,found=fism(last,root.right)
            return found
```

© GfH&JPK&RL

An algorithm – explanation

fism is supposed to produce a pair of values. The boolean signals that two immediate successors have been found. If the boolean is **false** , the integer is the maximum element found so far (otherwise it is the first of the pair of immediate successors).

With every invocation of *fism* it takes two parameters, a tree and an integer. The integer is the last value found in the previous sub-tree or node (last and previous in the sense of the preorder traversal). The tree is the next sub-tree in the preorder traversal.

When executing *fism* you will first check if its integer parameter is the immediate predecessor of the value in the node. If so you are ready. Otherwise you will invoke *fism* recursively for the left- and right sub-trees.

An algorithm – 2

```
def fism(last,N):
    if N==null:
        return last,False
    else:
        if N.key==last+1:
            return last,True
        else:
            last,found=fism(N.key,N.left)
            if found:
                return last,True
            else:
                return fism(last,N.right)
```

© GfH&JPK&RL

An alternative his

Convince yourself that the following *his* also works:

```
def his(root):
    last,found=fism(root.key, root)
    return found
```

An algorithm – counting steps

As a step we shall count every comparison between integers.

There is a simple formula for the worst case number of steps $T_{fism}(N)$:

$$T_{fism}(N) = \begin{cases} 0, \text{if } N = \textbf{null} \\ 1 + T_{fism}(N.left) + T_{fism}(N.right), \text{otherwise} \end{cases}$$

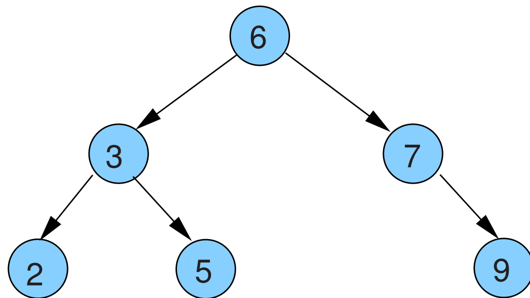
But without much mathematical analysis we can also see that in the worst case every node in the tree (except the root) will lead to exactly one comparison, so the number of steps equals (in the sense of Θ) the number of nodes in the tree.

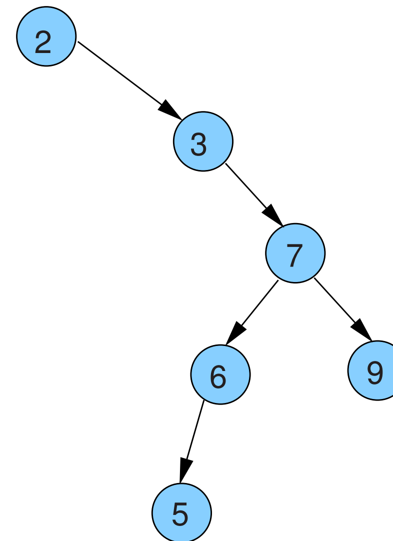
Binary search tree

A binary search tree (BST) is

- a binary tree with keys in nodes, i.e. such that:
- the key at a node is greater than all keys in its left subtree
- the key at a node is less or equal than all keys in its right subtree

An inorder traversal of a binary search tree yields a sorted list of keys



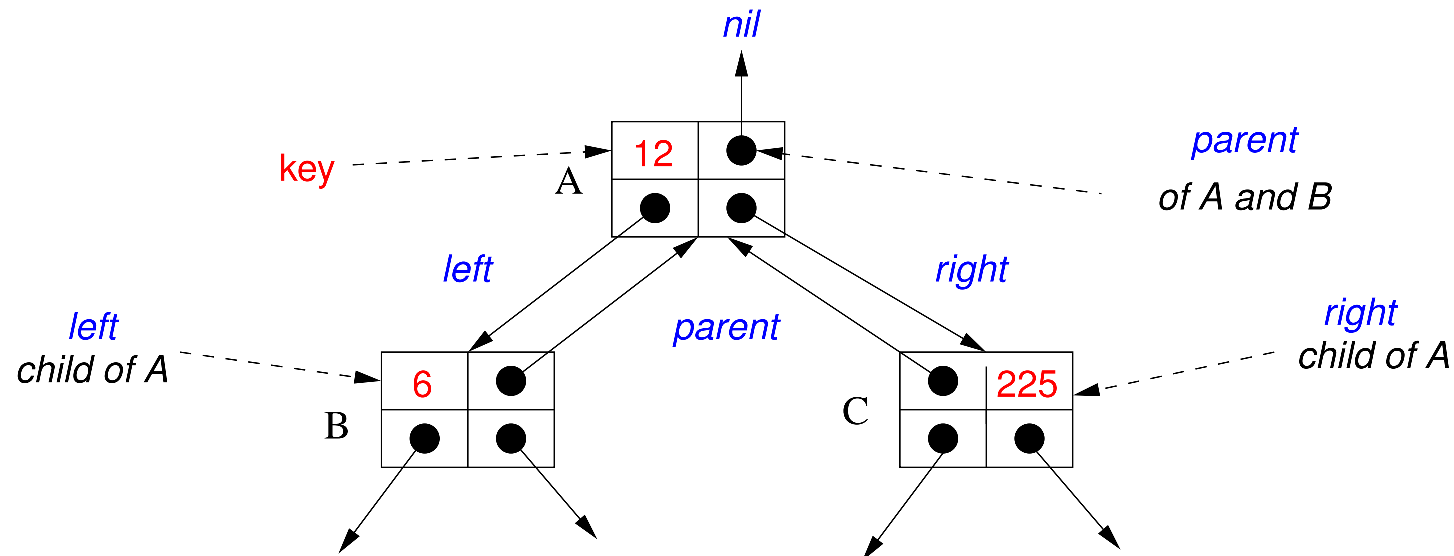


two binary search trees
containing 2, 3, 5, 6, 7, 9

Binary search tree

A node in a binary search tree has (at least) four features:

- a key – the “value” of the node
- a (pointer to a) left and right subtree (possibly empty)
- a (pointer to its) parent (which is empty for the root)



Searching for key k in a BST – Strategy

Exploit the binary search tree property:

- the key at a node is greater than all keys in its left subtree
- the key at a node is less or equal than all keys in its right subtree

Thus,

- if key of current node equals k, we are done
- If $k <$ key of current node, we recursively search the left subtree
- If $k >$ key of current node, we recursively search the right subtree

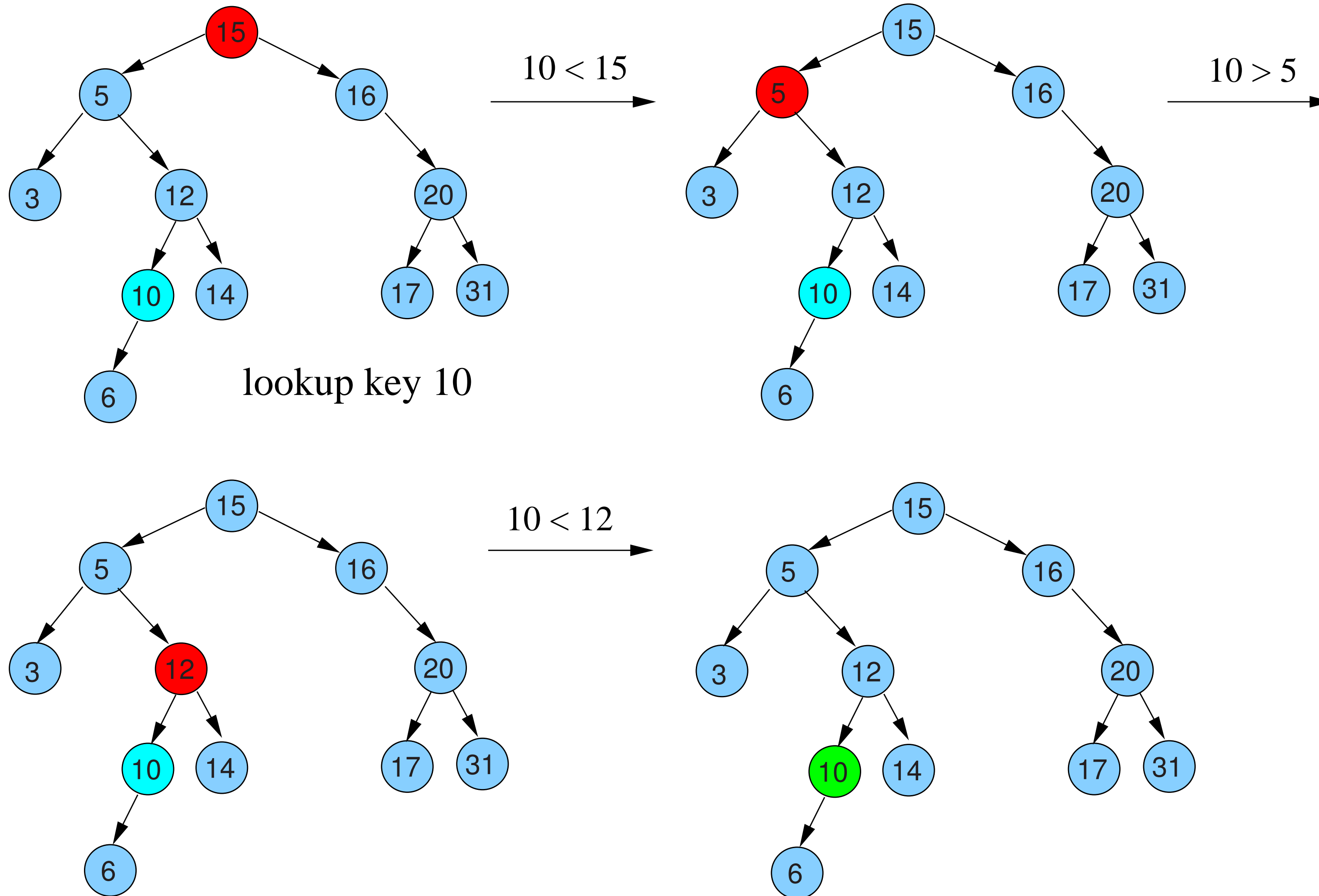
Searching for key k in a BST – Algorithm

```
def bstSearch(root,k):
    if root==null or root.key==k:
        return root
    else:
        if k<root.key:
            return bstSearch(root.left,k)
        else:
            return bstSearch(root.right,k)
```

The worst-case complexity is linear in the level d of the tree: $\Theta(d)$

- for a chain-like tree structure with n nodes this amounts to $\Theta(n)$
- if the BST is as balanced as possible this amounts to $\Theta(\log n)$

Note that such search procedure does not work for heaps; Why?


15
5
16
3
12
20
10
14
17
31
6
lookup key 10
10 < 15
10 > 5
10 < 12

Insertion of key k into a BST – Strategy

Search for key k in the tree T, exactly like we did before.

However, we do not stop if we find a node that has key k; in such a case we try to insert k in the right subtree of that node.

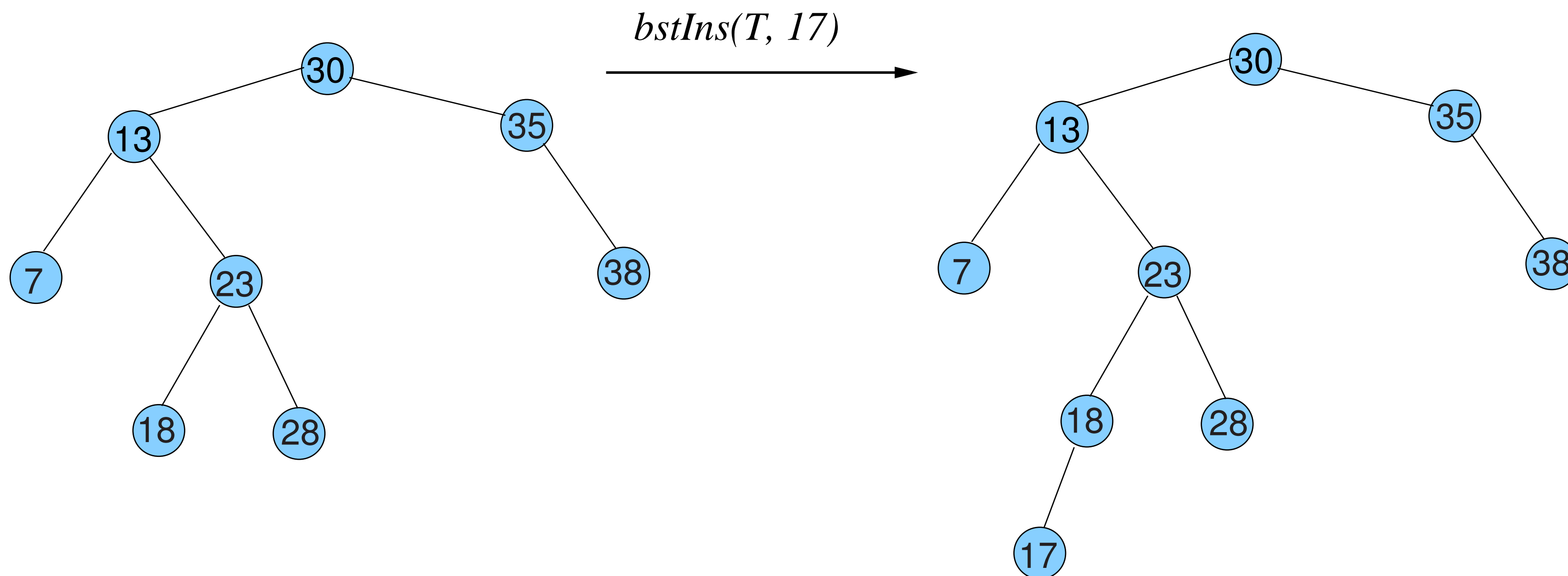
So the search will end at a leaf.

Now replace the empty tree enccountered by a node with key k

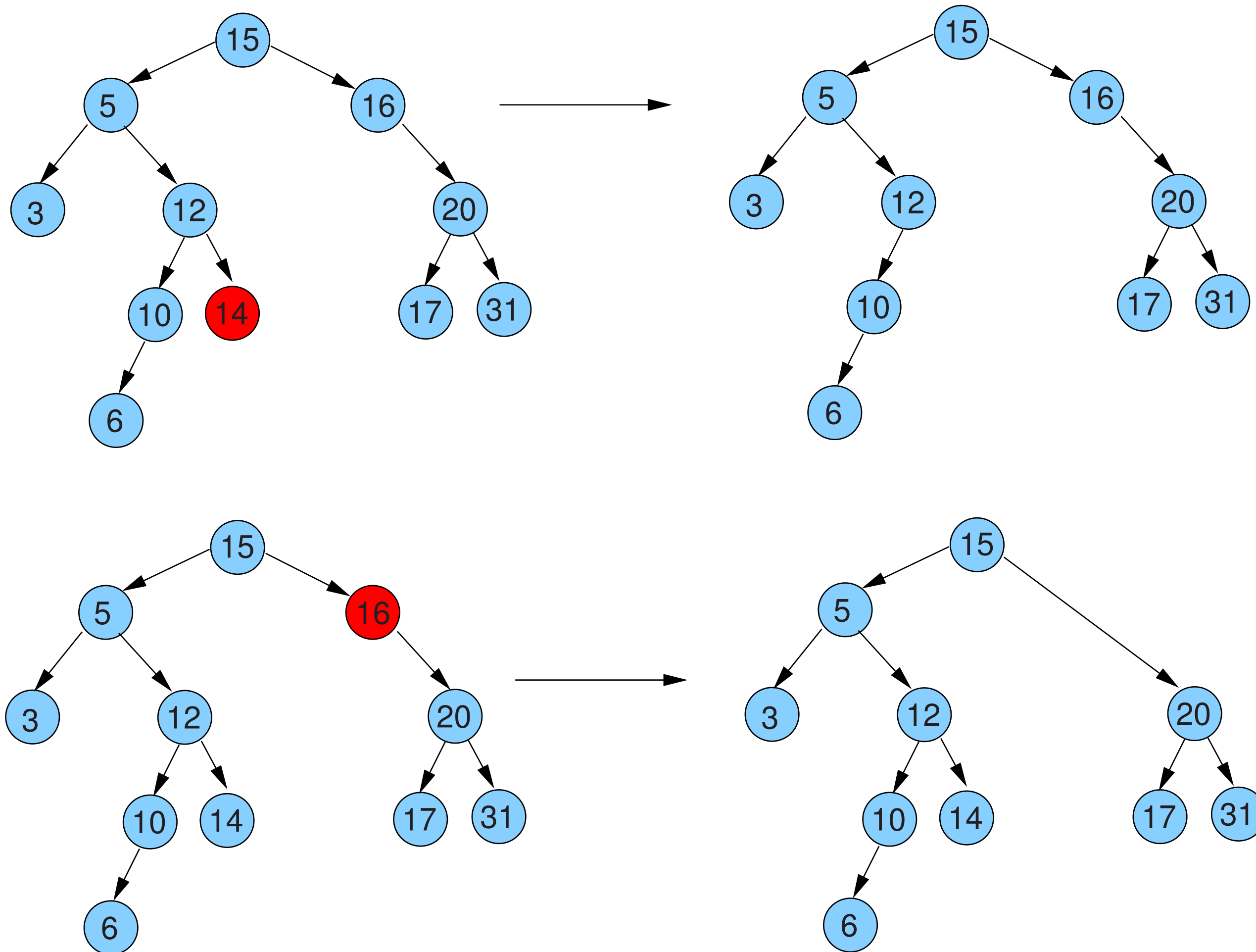
Insertion of node with key k into a BST – Algorithm

```
def bstIns(T,z):
    x,y=T.root,null
    while x!=null:      #search leaf to insert z
        y=x             #keep track of parent of x
        if z.key<x.key:
            x=x.left    #go left
        else:           #go right
            x=x.right
    z.parent=y          #start inserting z
    if y==null:
        T.root=z        #z is new root
    else:
        if z.key<y.key:  #put z in right place
            y.left=z
        else:
            y.right=z
```

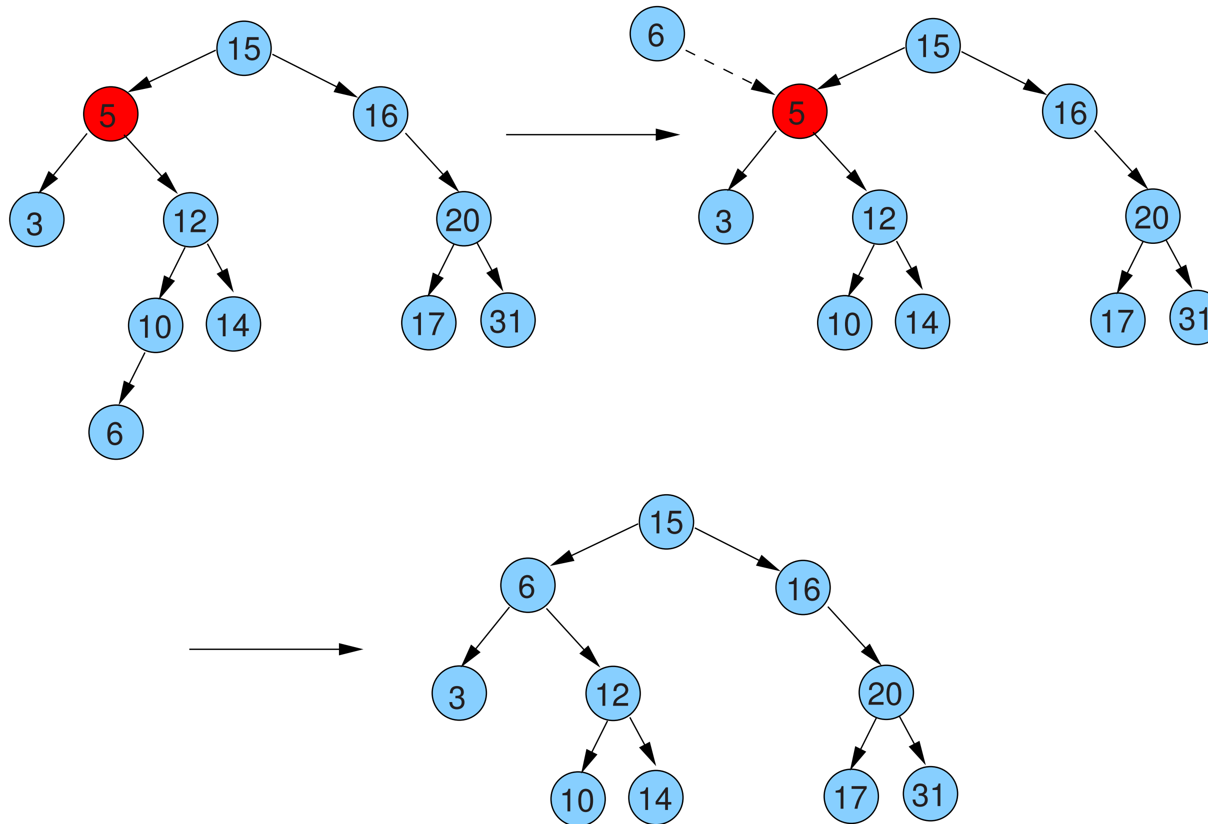
Insertion of key k into a BST – Example



Deletion: the two simple cases



Deletion: the more involved case



© GfH&JPK&RL

Deletion from a BST – Strategy

Deleting node z from a BST is performed as follows:

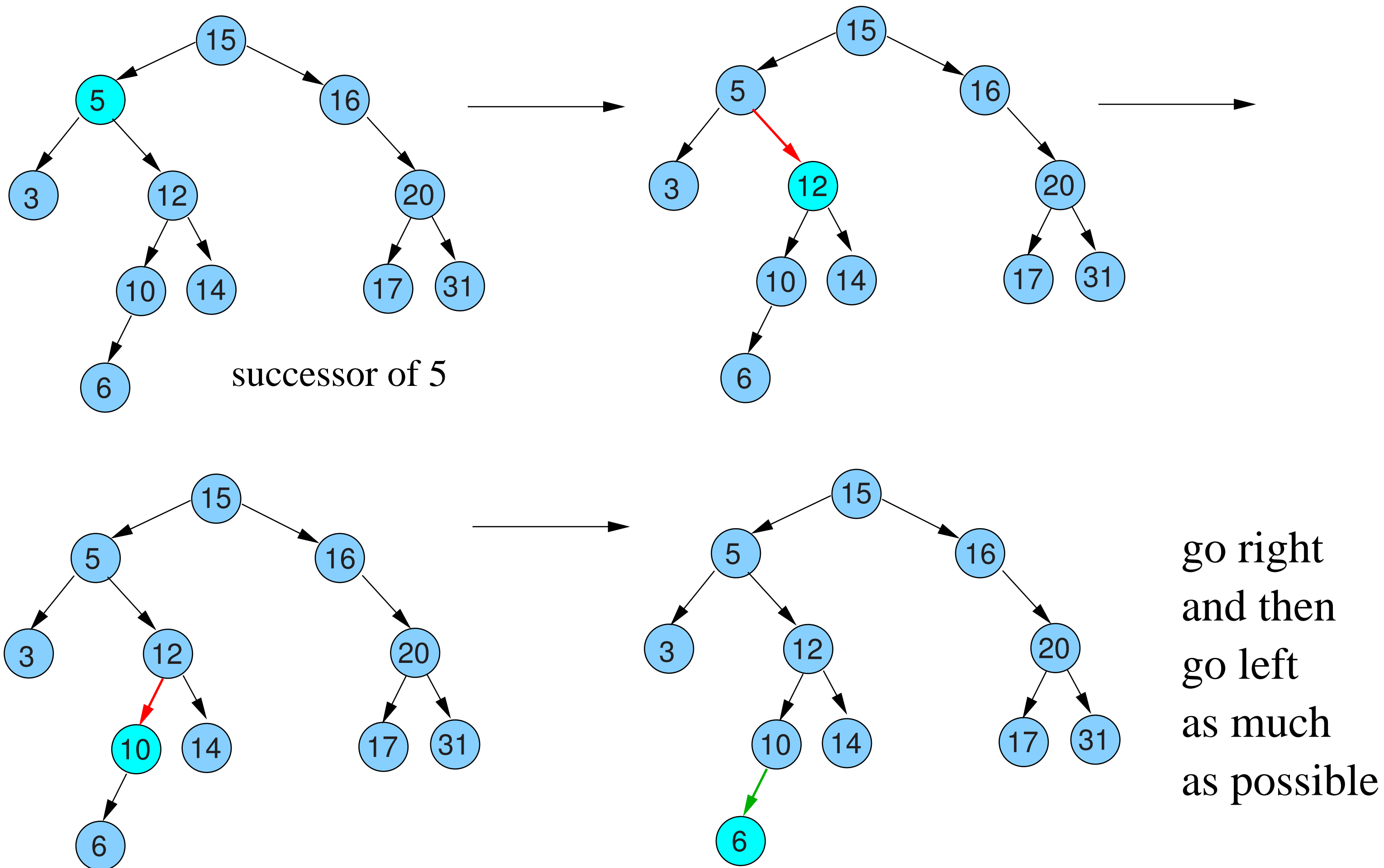
- If z has no children, we modify the parent of z by replacing z by **null**
- If z has one child, we “splice out” z by linking its parent and its child
- If z has two children then:
  - we determine its successor
  - eliminate that
  - replace z's key with that of its successor

First consider how to determine the successor of a node in a BST

Finding the successor of a key – Algorithm

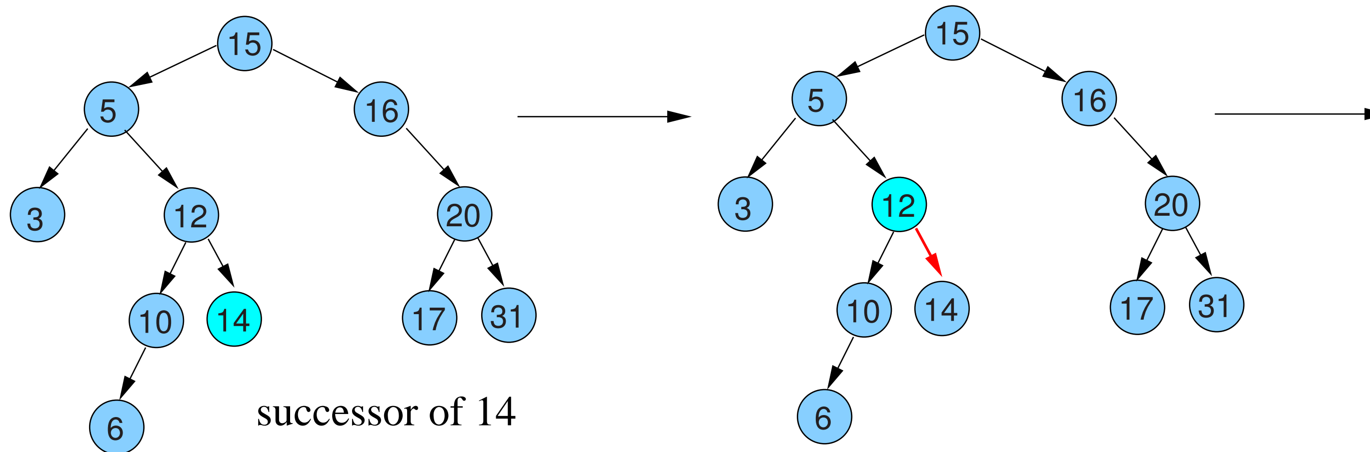
```
def bstSucc(x):
    if x.right!=null:
        x=x.right                  #go right
        while x.left!=null:        #go as left as possible
            x=x.left
        return x     #return leftmost node in right subtree
    else:
        y=x.parent                         #go up
        while y!=null and y.right==x: #while x left child
            x=y                            #go up again
            y=y.parent
        return y          #this case not used for deletion!
```

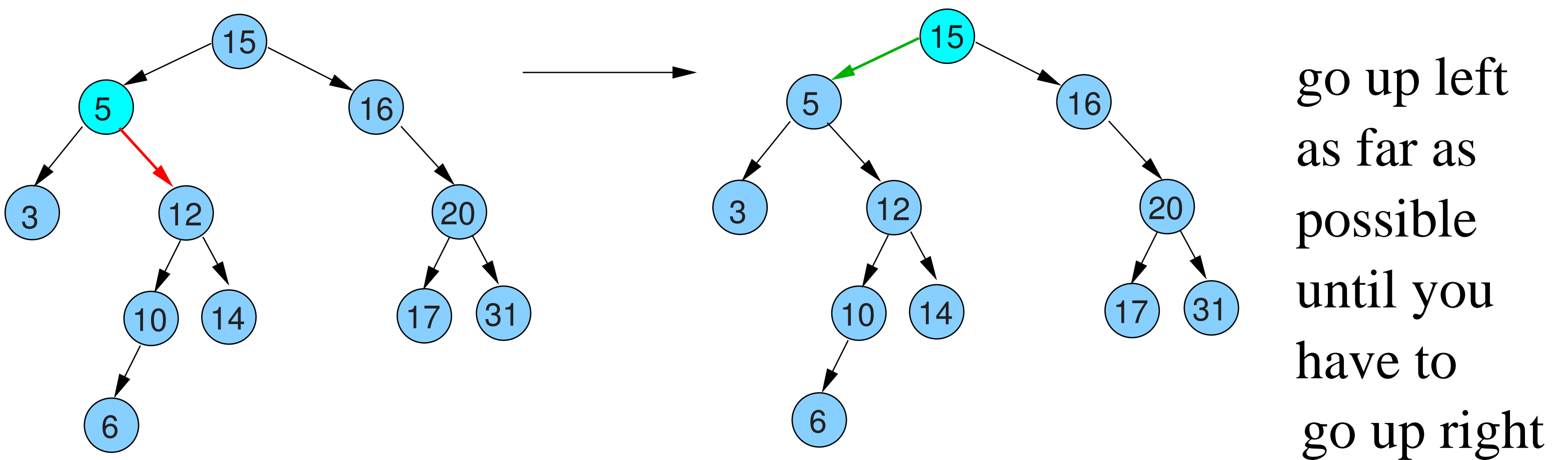
Finding the successor of a key – Case 1



go right
and then
go left
as much
as possible

Finding the successor of a key – Case 2





Deletion from a BST – Algorithm

```
def bstDel(T,z):
    y= (z if (z.left==null or z.right==null) else bstSucc(z) )
            #y is node to be eliminated
    x= (y.left if y.left!=null else y.right)
            #x is non-null child of y (if any)
    if y.parent==null:
        T.root=x
    else:                          #replace y by x
        if y==(y.parent).left:
            (y.parent).left=x
        else:
            (y.parent).right=x
    z.key=y.key          #logically delete z, i.e. it becomes y
    return y
```

Complexity of BST operations

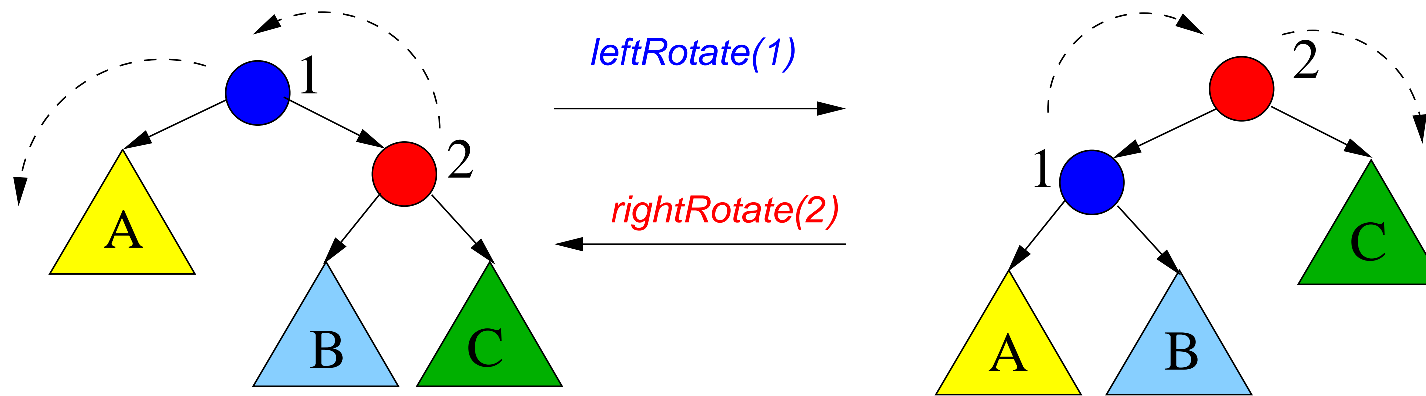
Operation	*Time*
bstSearch	$\Theta(d)$
bstSucc	$\Theta(d)$
bstMax	$\Theta(d)$
bstIns	$\Theta(d)$
bstDel	$\Theta(d)$

all operations are linear in the depth d of the bs-tree

this is $\log n$ if the tree is not too much “unbalanced”

balancing of binary trees amounts to using rotations

Rotations - Concept

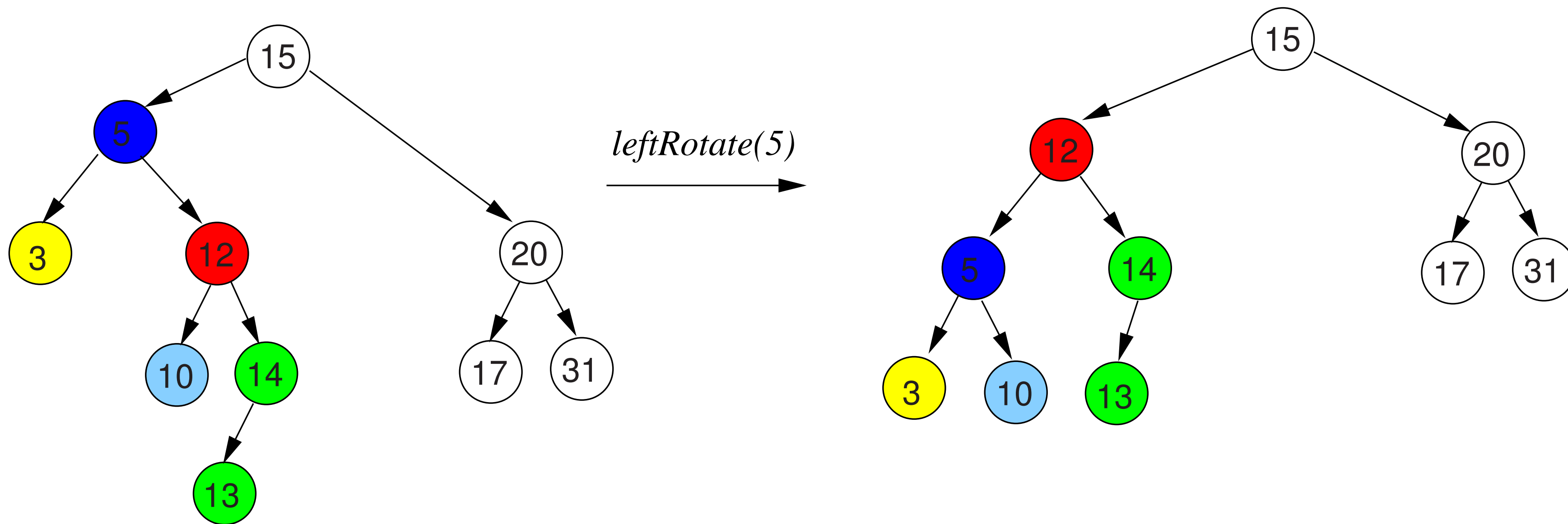


if original tree is a bst, then the rotated tree is a bst

moreover, the inorder traversal of original and rotated tree are identical!

© GfH&JPK&RL

Rotate-left – Example



Rotate-left – Algorithm

```
def leftRotate(T,x):
    y=x.right
    x.right=y.left
    (y.left).parent=x
    y.parent=x.parent
    if x.parent==null: T.root=y
    else:
        if x===(x.parent).left: (x.parent).left=y
        else: (x.parent).right=y
    y.left=x
    x.parent=y
```

this algorithm takes $\Theta(1)$ time in worst case

rightRotate is similar (symmetric) and is omitted here

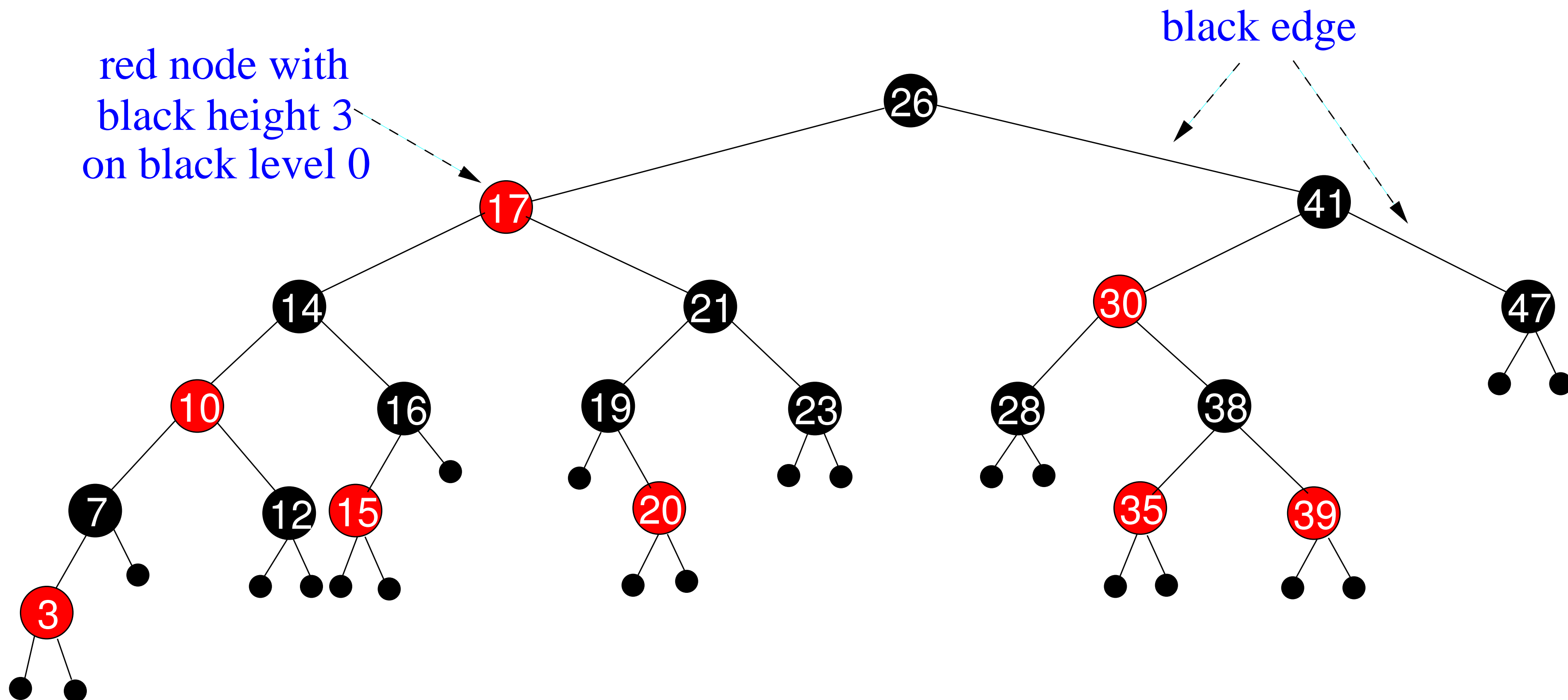
Red-black trees

A red-black tree (RBT) is a *binary search tree* that fulfills:

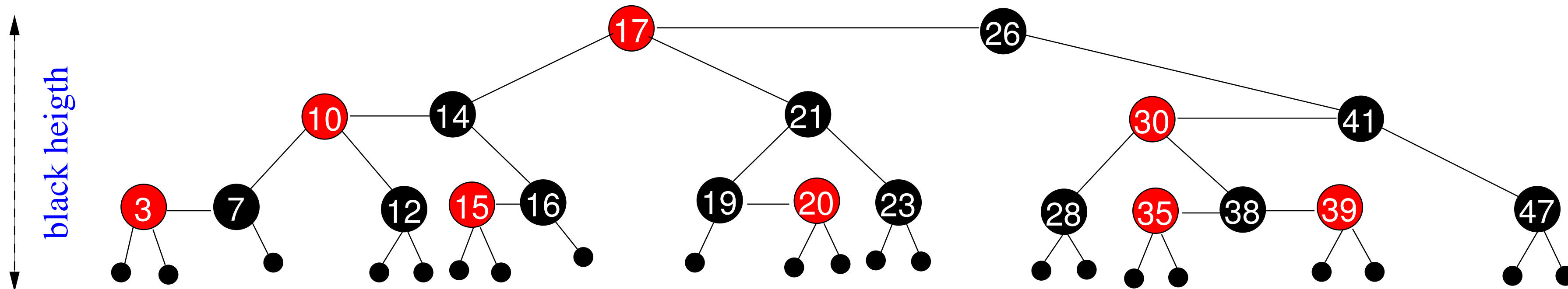
1. every node is either red or black
2. the root is black; final nil-nodes are also black
3. a red node has no red children
4. An edge to a black node is called a black edge
5. The black level of a node N is #black edges you pass, walking from the root to N
6. the black height of every node is well-defined: all paths from a node to a nil-node have the same black length

It is convenient to draw red nodes on the same level as their parents

Red-black trees – picture 1



Red-black trees – picture 2



Insertion and deletion require complicated ”repair”: operations (using color flipping and rotations), in order to keep the tree balanced.