Lecture
11_8.6

Trees & Traversals

Alessandro SAVINO
Politecnico di Torino (Italy)
alessandro.savino@polito.it
www.testgroup.polito.it

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Goal

- This lecture aims at presenting the *Tree* container, the related operations, and the visiting techniques.

Prerequisites

- **Lectures:**
  - **11_7.x Pointers & Dynamic Memory**

Further readings

- **Students interested in a deeper look at the covered topics can refer, for instance, to the books listed at the end of the lecture.**

Outline

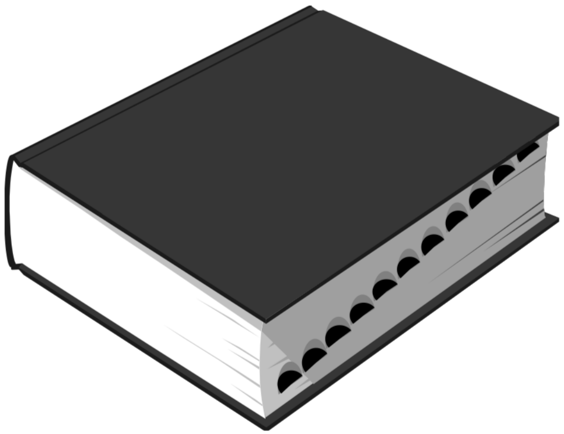
- **Trees introduction**
- **Binary search trees**
- **Traversing algorithms**
- **Searching a BST**
- **Insert and Delete in a BST**
- **Tree balancing**

Outline

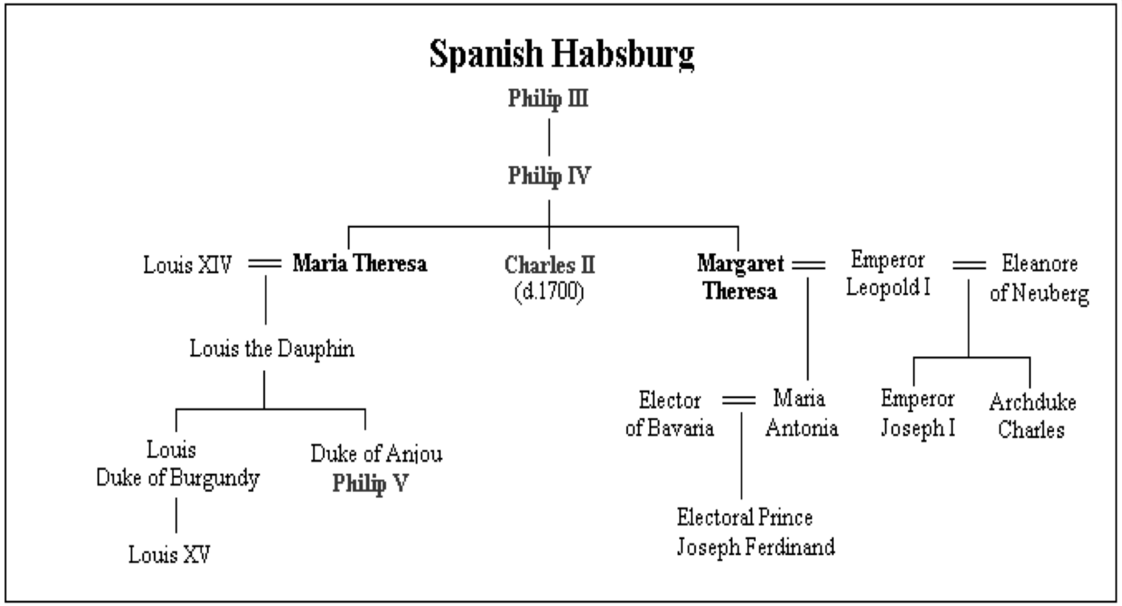
- Trees introduction
- Binary search trees
- Traversing algorithms
- Searching a BST
- Insert and Delete in a BST
- Tree balancing

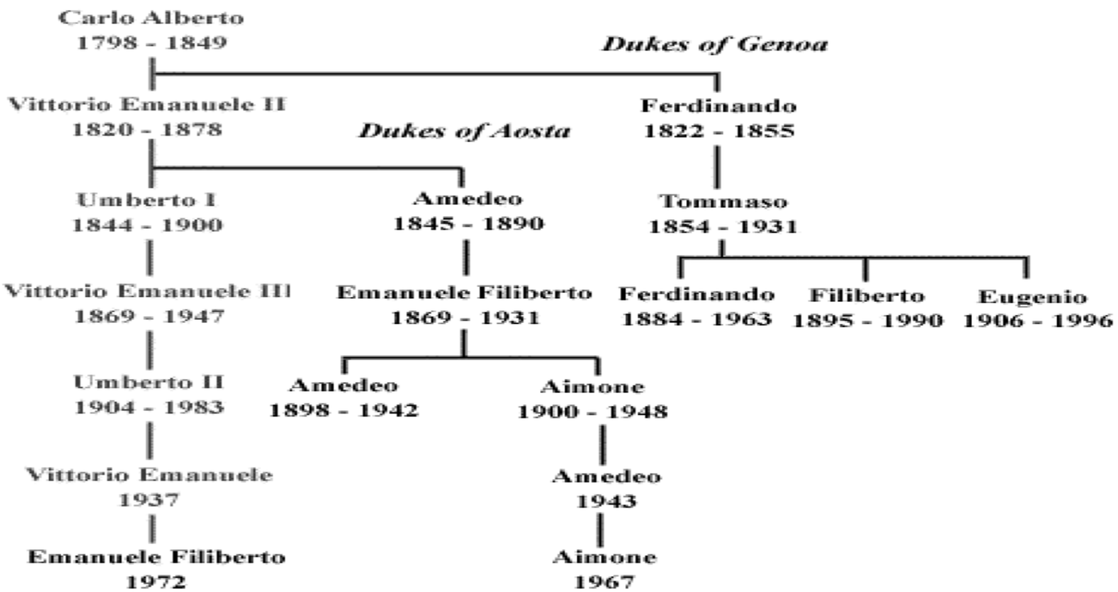
Tree

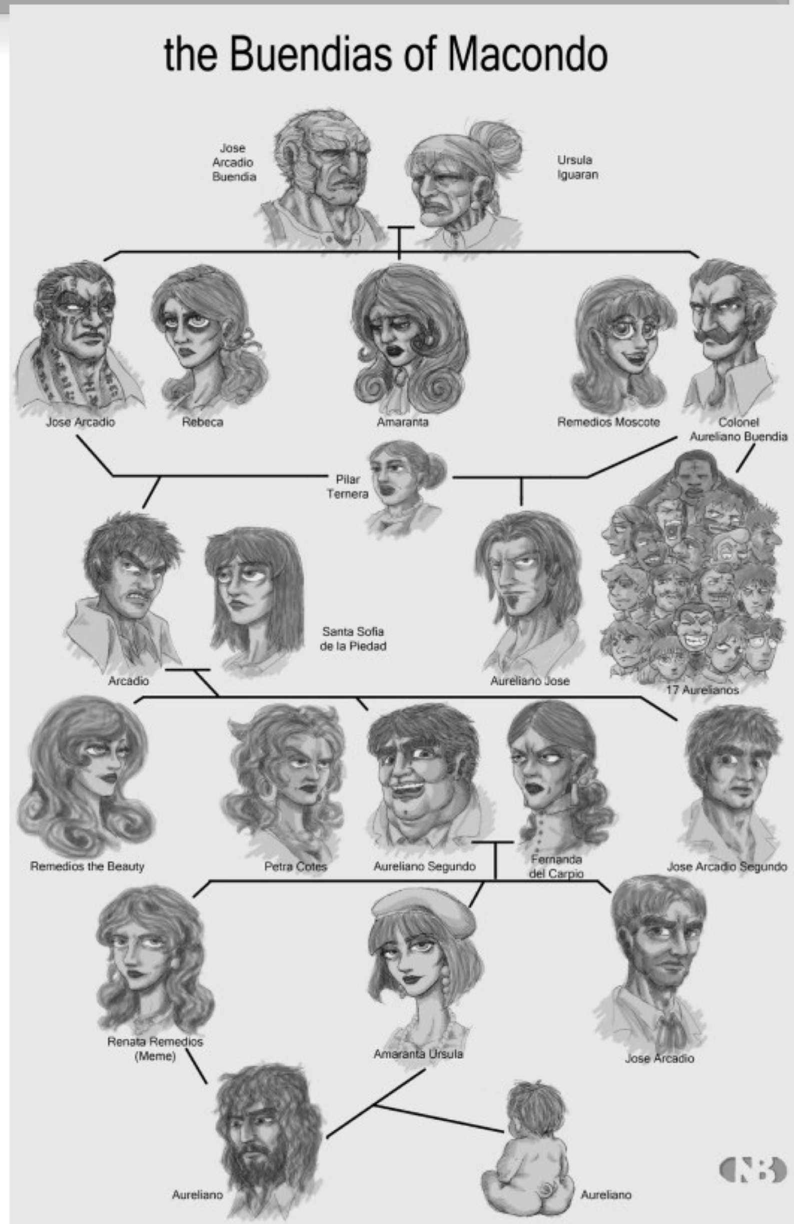
A tree is an ADT that represents a hierarchical structure. Containing a **root** ***node and a set of linked nodes known as*** **children.** ***There exists only*** **one path** ***from the root to any node.***



Family trees







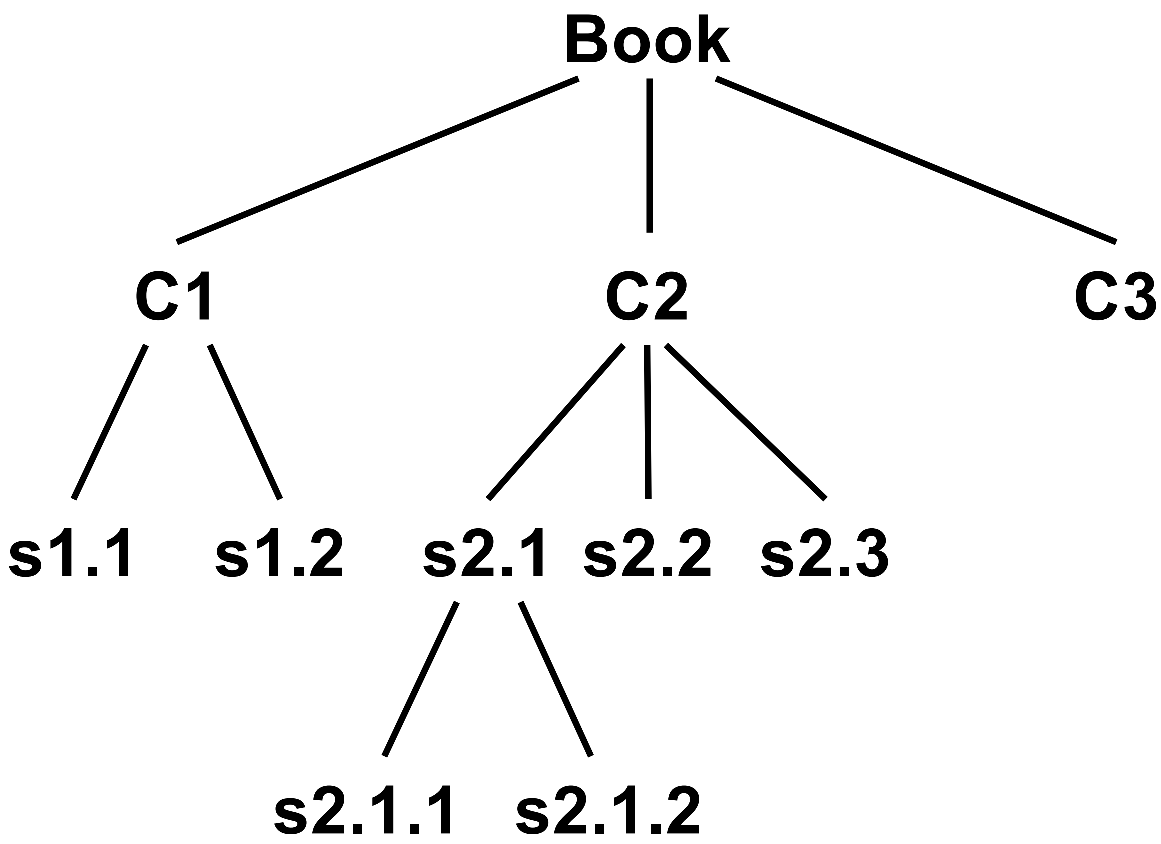
One Hundred Years of Solitude by GGM

Indexes

- Book
  - C1
    - s1.1
    - s1.2
  - C2
    - s2.1
      - s2.1.1
      - s2.1.2
    - s2.2
    - s2.3
  - C3

Indexes

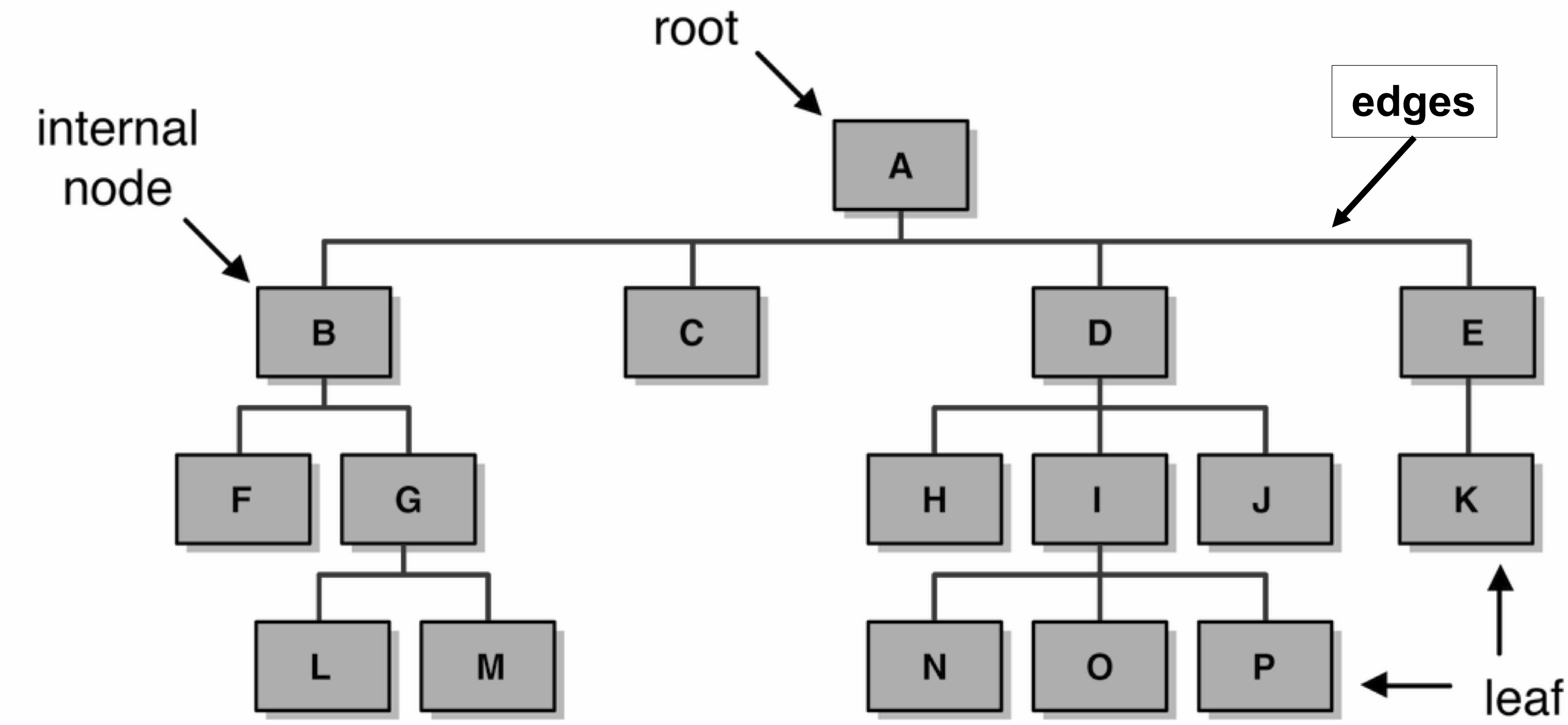
- Book
  - C1
    - s1.1
    - s1.2
  - C2
    - s2.1
      - s2.1.1
      - s2.1.2
    - s2.2
    - s2.3
  - C3

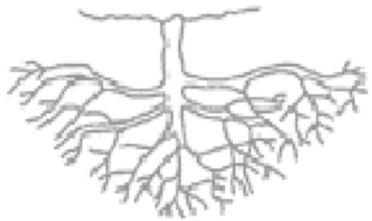


Formal definition

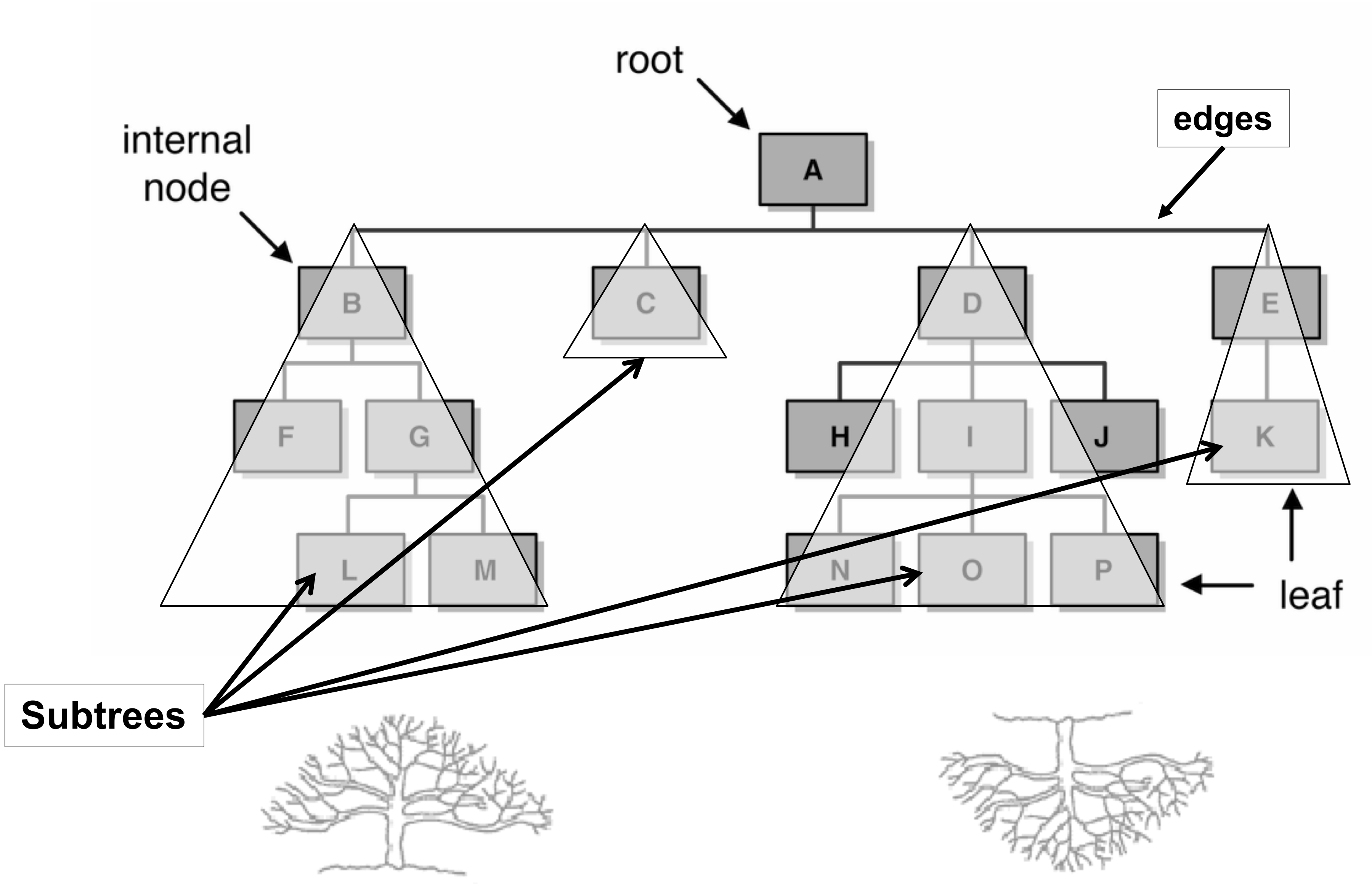
- A *tree* is an *acyclic* data structure composed by *nodes* and *edges* accessed beginning at a *root node*
  - Each node is either a *leaf* or an *internal node*
  - An *internal node* has 1 or more *children*, nodes that can be reached directly from that internal node.
  - The internal node is said to be the *parent* of its child nodes

Tree diagram





Tree diagram

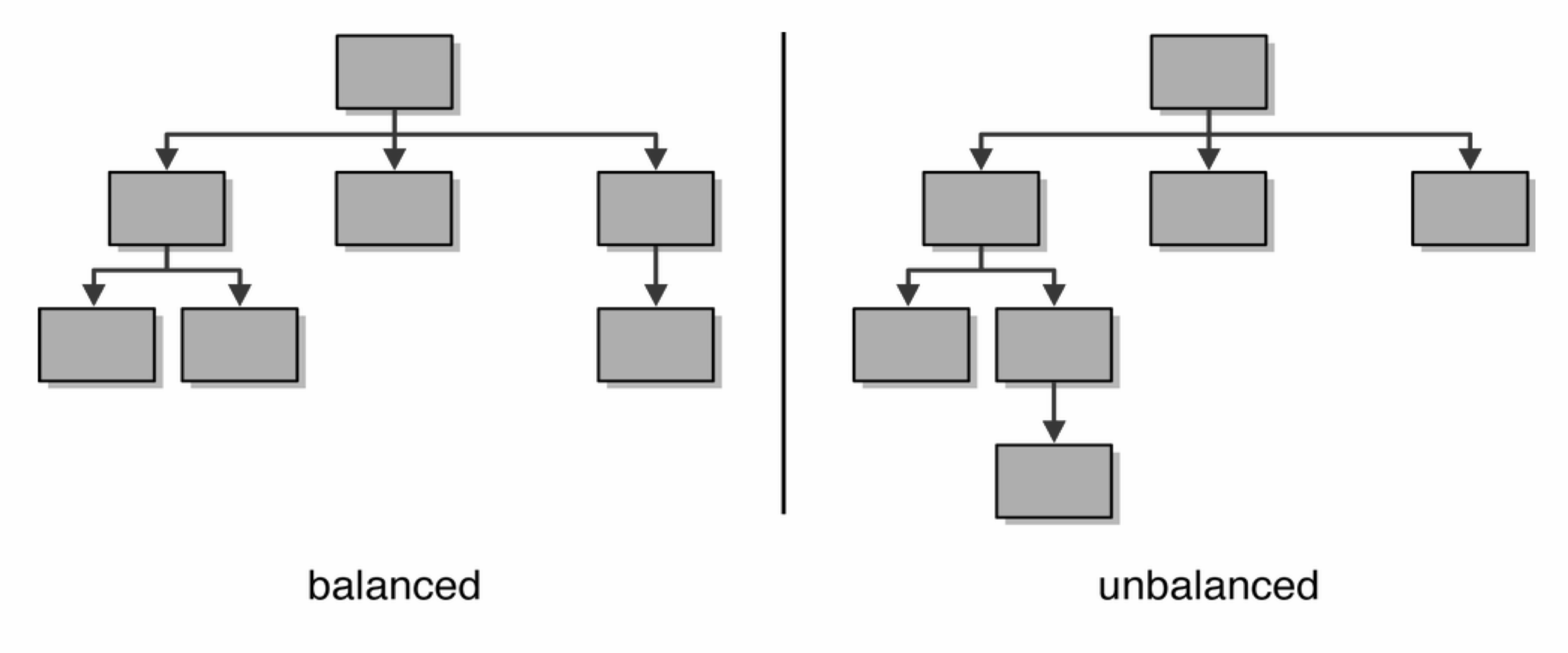


Tree terminology

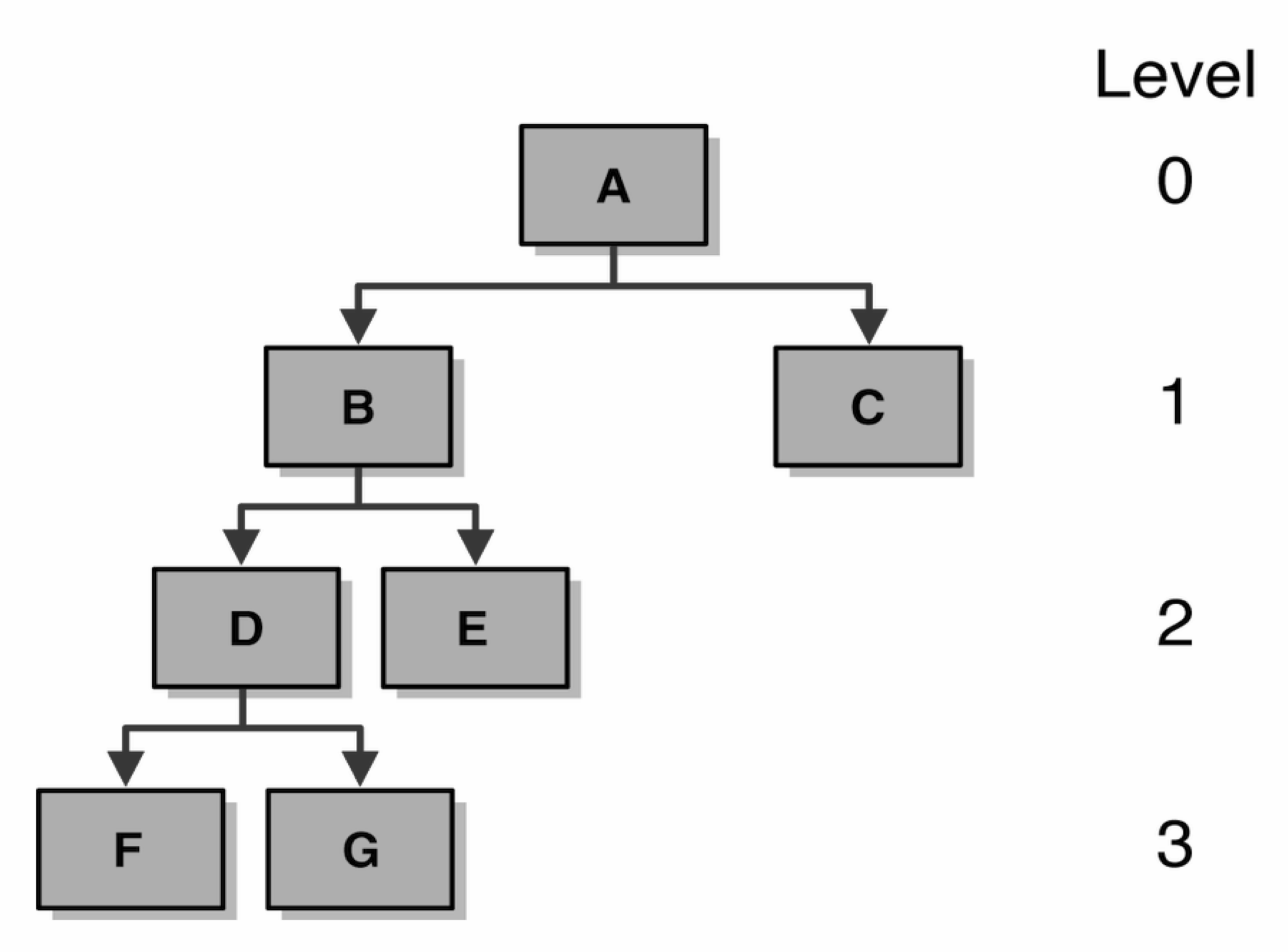
- *Leaf*: node with no children
- *Siblings*: two or more nodes with the same parent.
- *Path*: a sequence of nodes n_1, n_2, … , n_k such that n_i is the parent of n_{i+1} for $1 \le i < k$
  - the length of a path is the number of edges in the path, or 1 less than the number of nodes in it
- *Depth or level*: length of the path from root to the current node (depth of root = 0)
- *Height*: length of the longest path from root to any leaf
- *Degree*: number of subtrees of a node.

Balanced trees

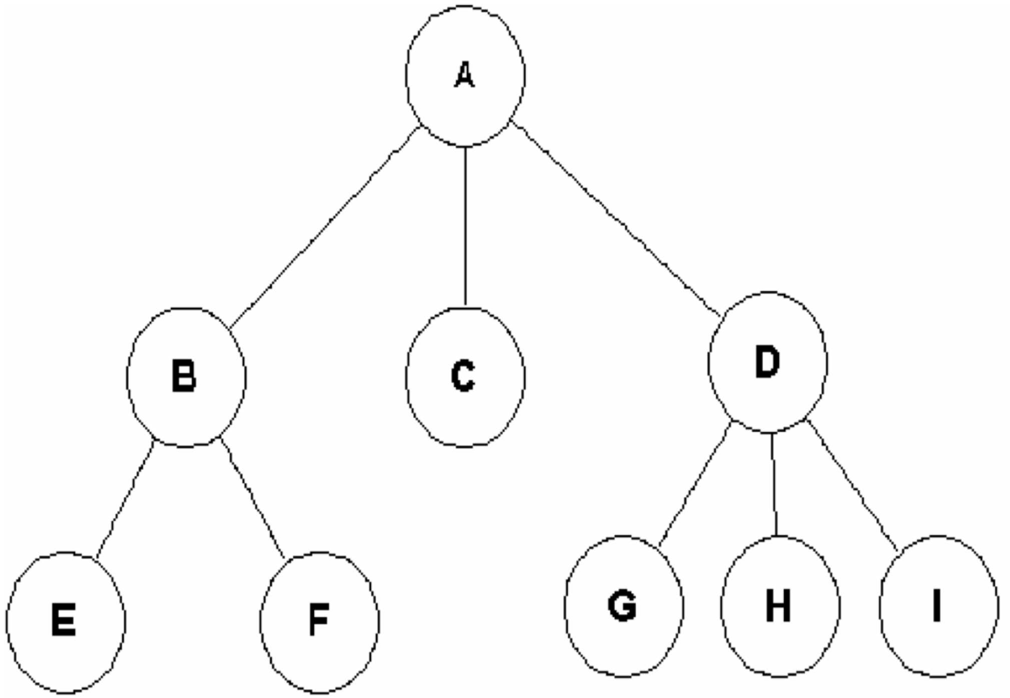
- A *balanced* tree is one where no node has two subtrees that differ in height by more than 1
  - visually, balanced trees look wider and flatter



Tree path length and depth



Tree example



root	A
leaves	E F C G H I
height	2
level of root	0
level of node with F	2
nodes at level 1	3
parent of G, H and I	D
descendants of B	E F

Trees representing arithmetic expressions

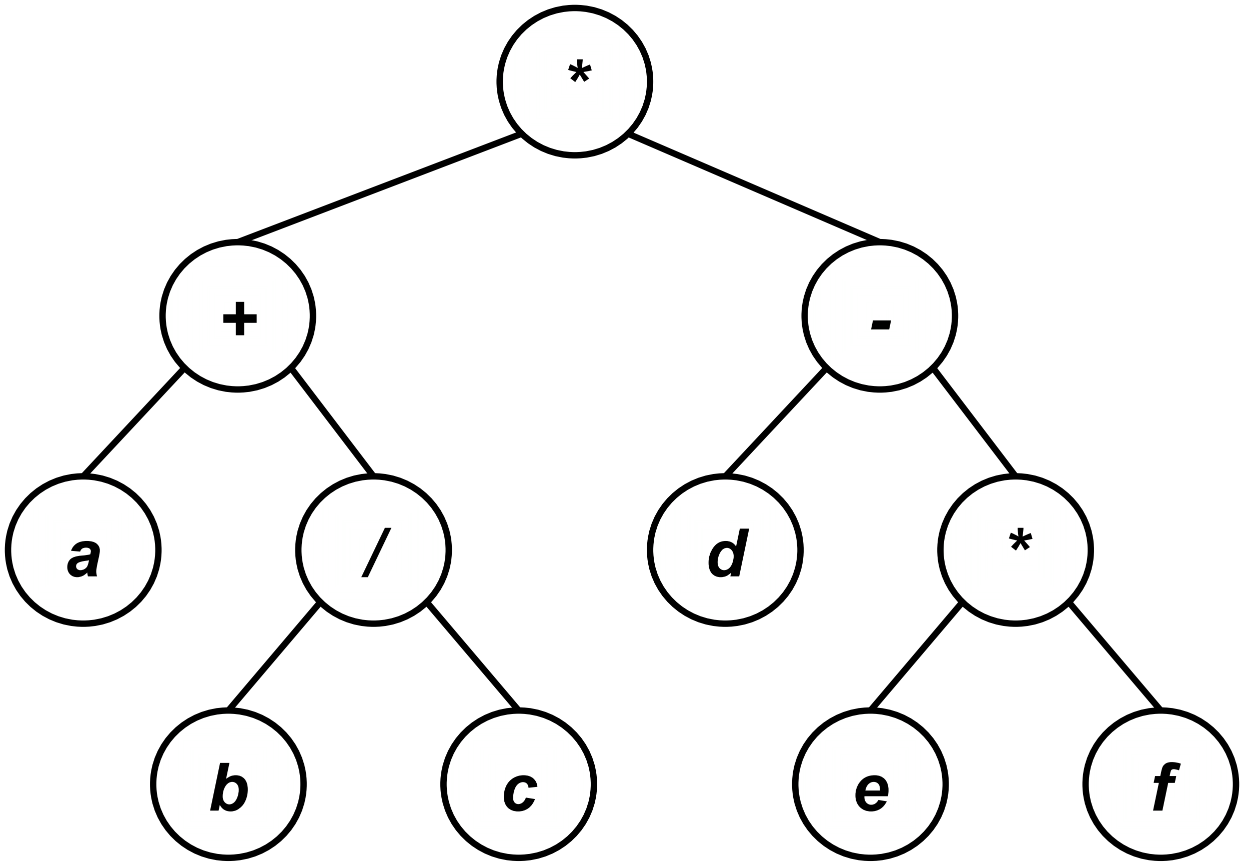
A tree representing an arithmetic expression follows these rules:

- ***Leaves*: Operands (constants or variables)**
- ***Non-leaves* nodes: operators.**

Trees representing arithmetic expressions

Example:

(a+b/c)*(d-e*f)



Outline

- Trees introduction
- **Binary search trees**
- Traversing algorithms
- Searching a BST
- Insert and Delete in a BST
- Tree balancing

Binary Search Trees - BST

Binary Search Trees (BST) are containers efficiently supporting the following operations:

Search, Minimum, Maximum, Predecessor, Successor, Insert, Delete.

They are a good solution to implement dictionaries or priority queues.

Goals

BSTs are defined in such a way that the complexity of each operation is proportional to the *height h* of the tree.

For a complete and balanced tree with *n nodes*, the complexity is *Θ(log n)* in the worst case.

For a fully unbalanced tree, the worst case is O(n).

In average we expect Θ(log n).

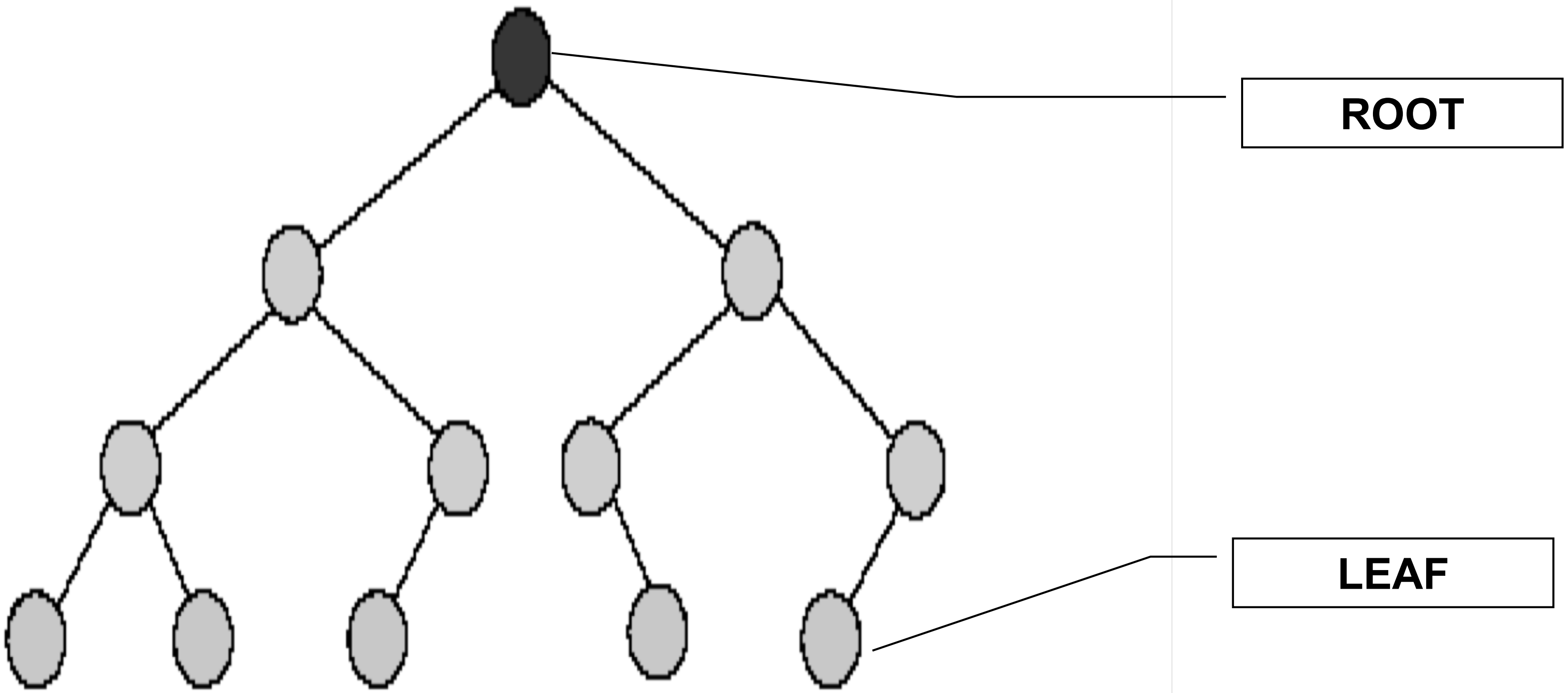
Definitions

Binary Search Tree:

- *Tree*: hierarchical structure with ONE root and ONLY ONE path from the root to any node
- *Binary*: each node has at most two children (left and right) and (except for the root) exactly one father (p)
- *Search*: the nodes have a key, used as a sorting criteria

BST

A 2nd degree tree, may be a BST

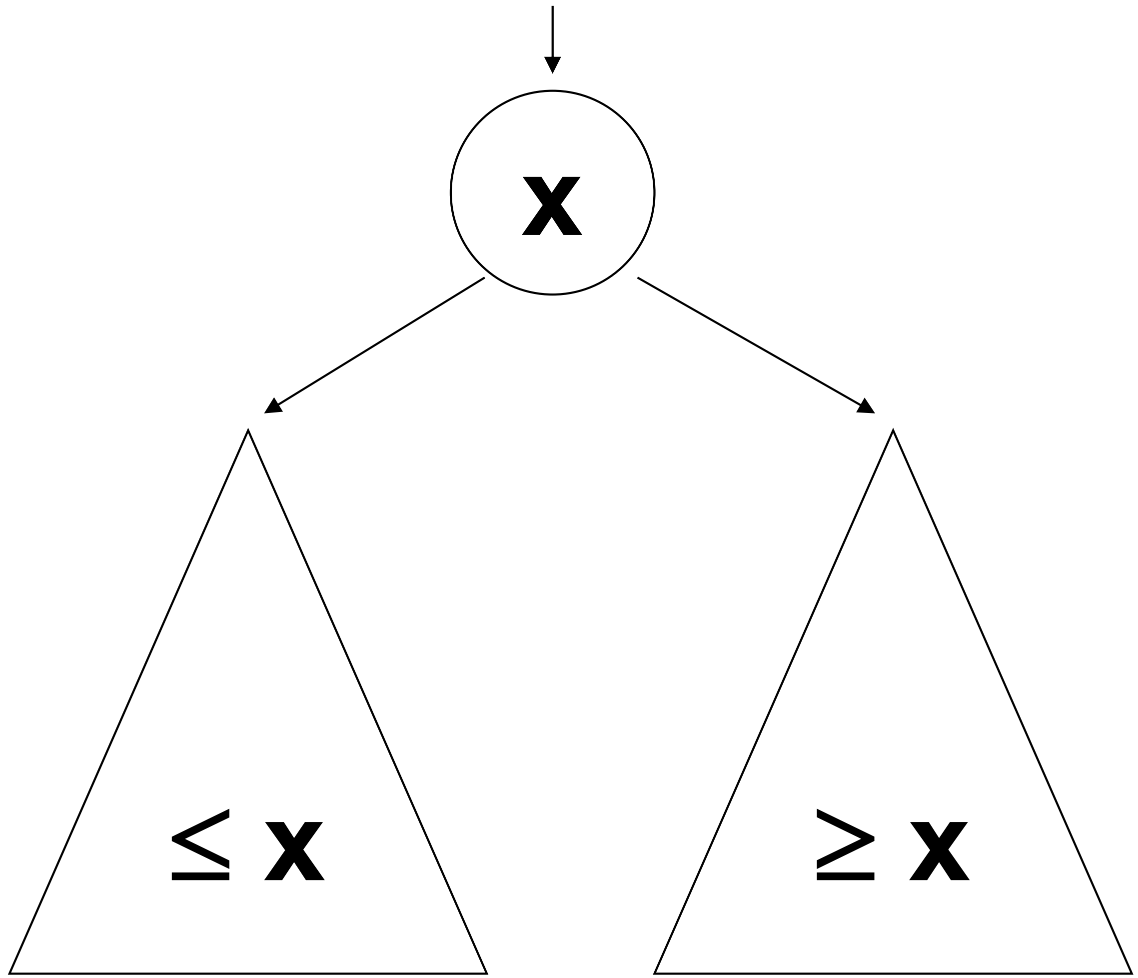


Sorting rule (I)

- For each node x:
  - For all nodes in the left tree:
    key[y] ≤ key[x]
  - For all nodes in the right tree:
    key[y] ≥ key[x]

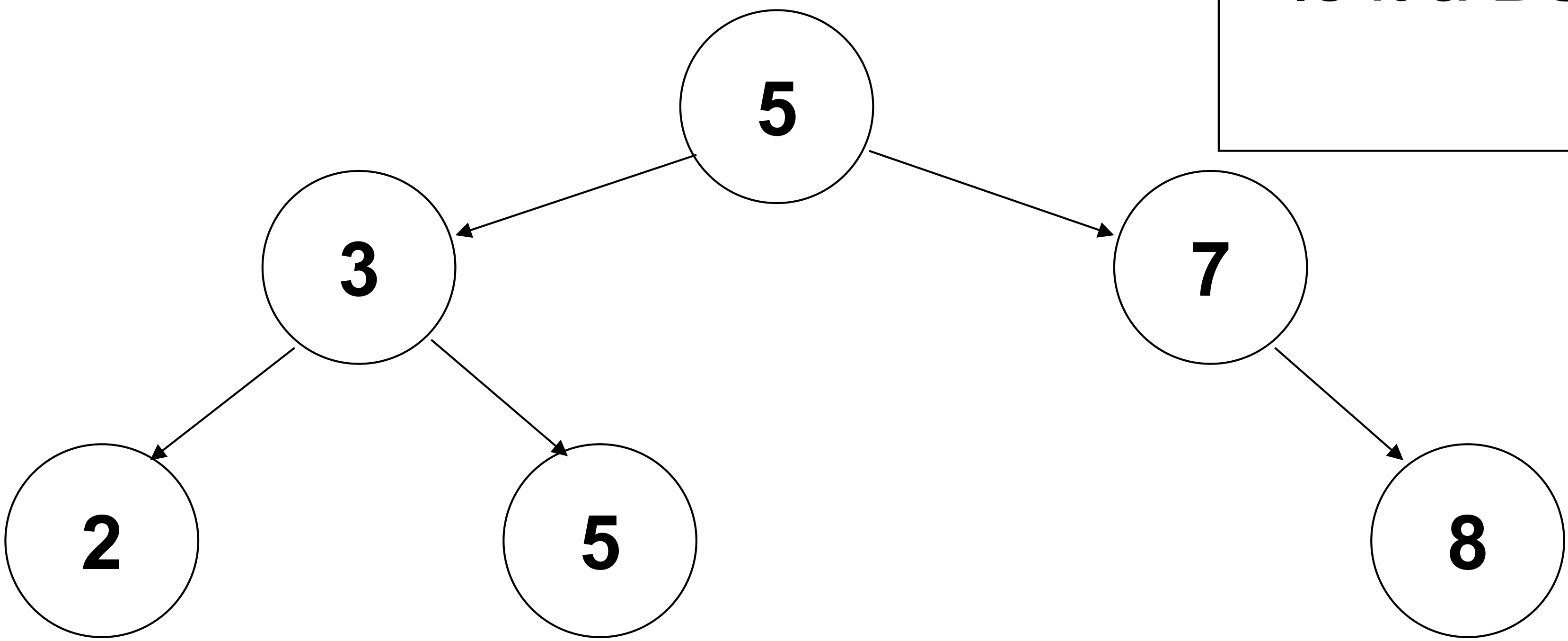
The definition is recursive along the tree!

Sorting rule (II)



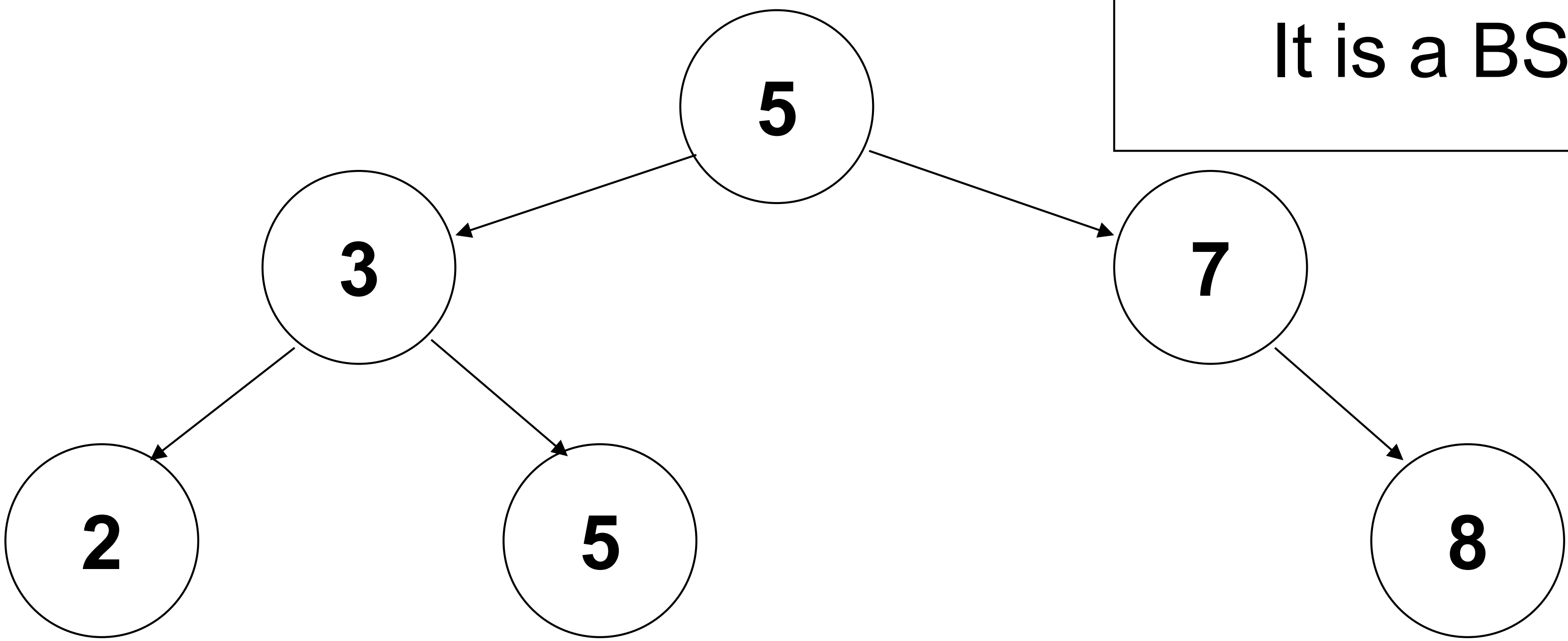
Example I

Is it a BST ?



Example I

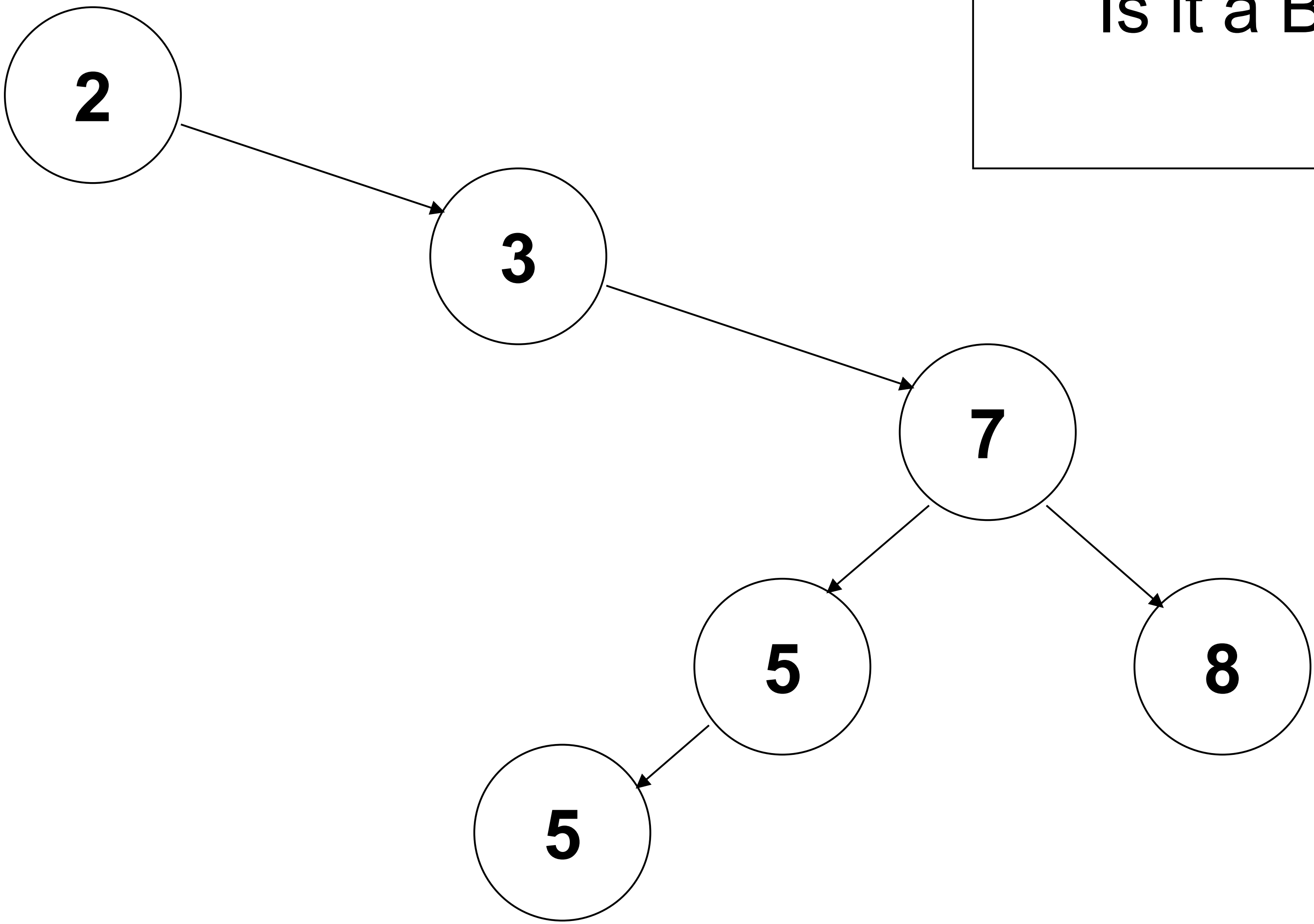
The sorting rule is true for each node:
It is a BST



 Rel. 10/06/2019

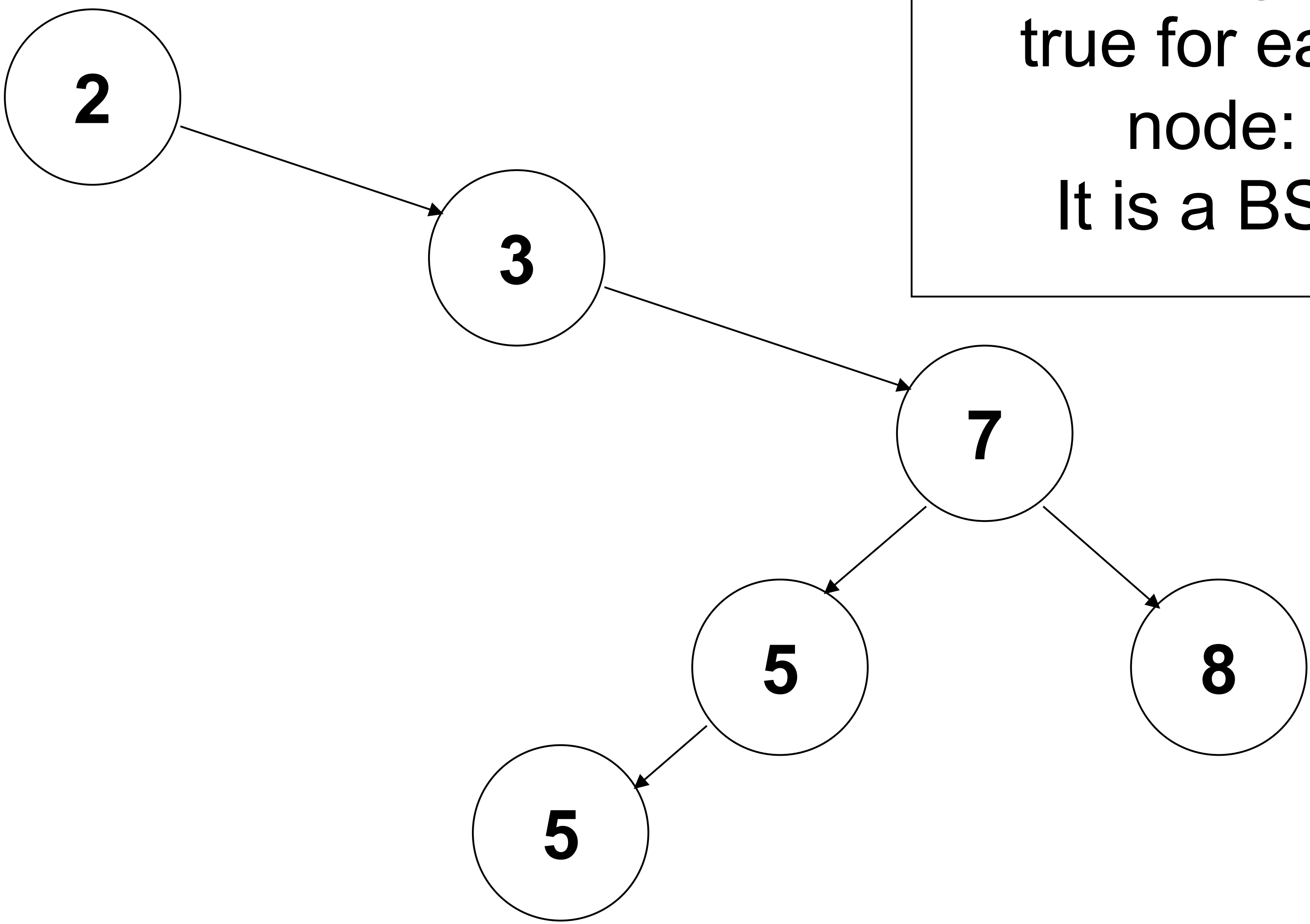
Example II

Is it a BST?



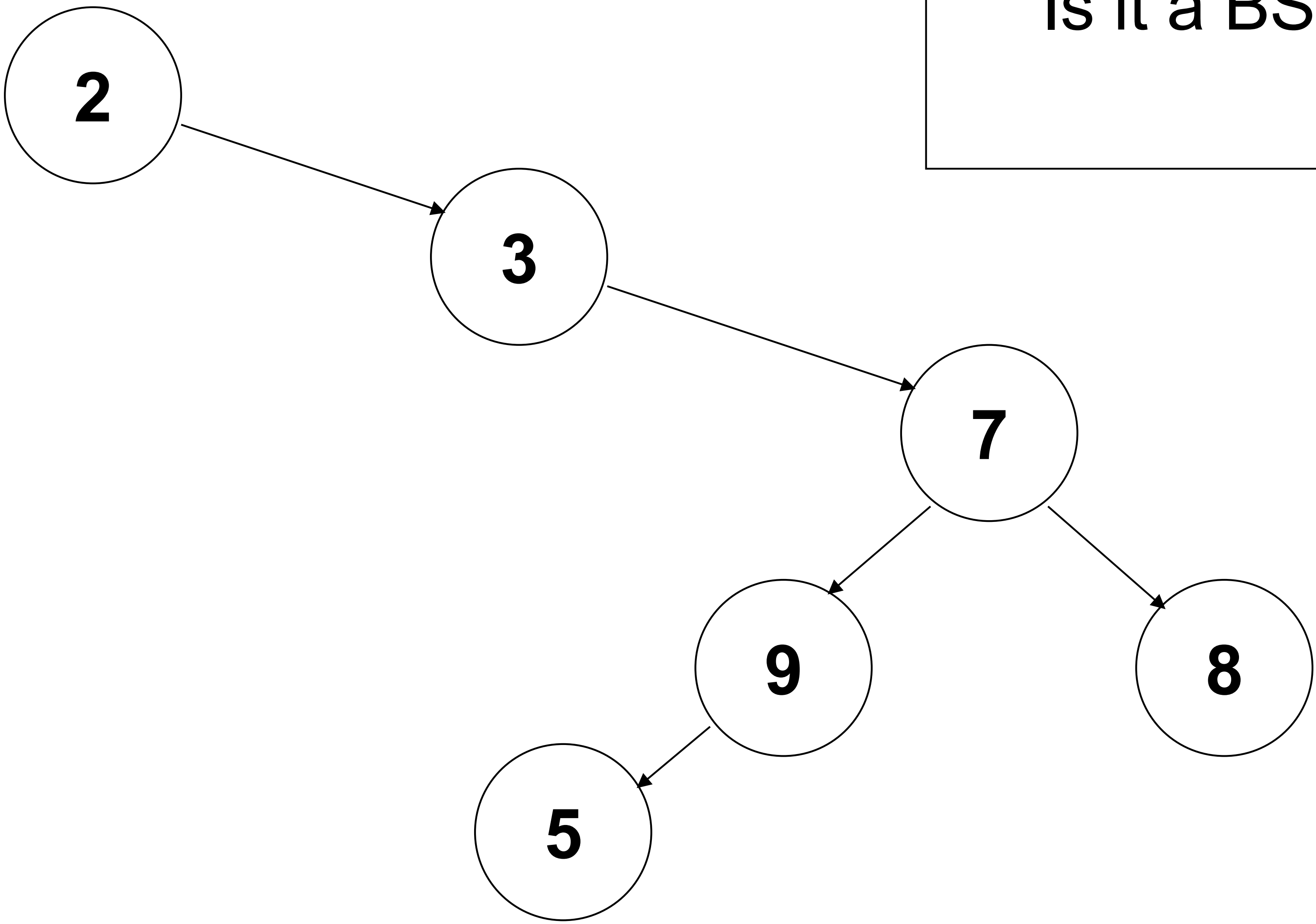
Example II

The sorting rule is true for each node:
It is a BST

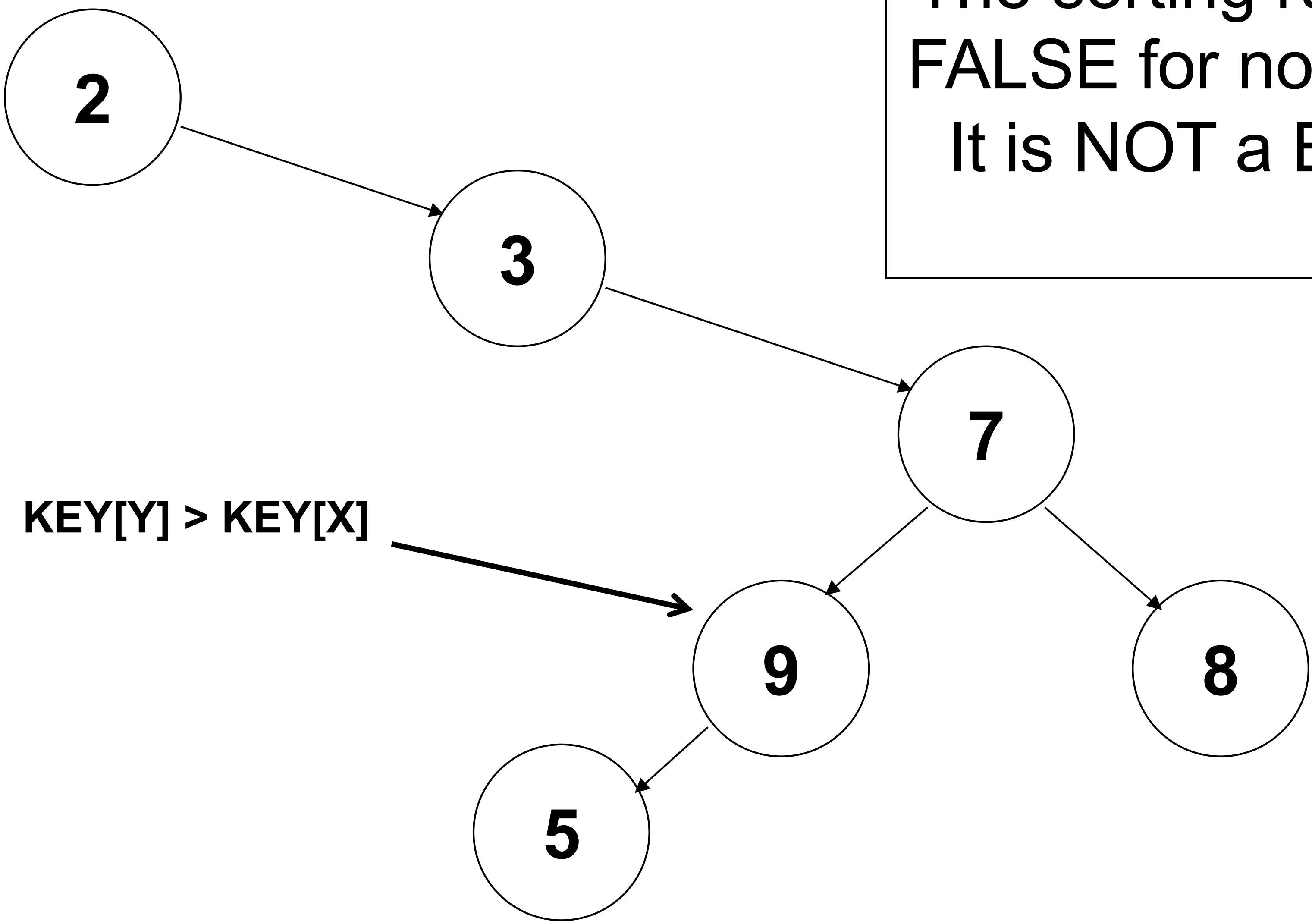


Example III

Is it a BST ?



Example III



The sorting rule is FALSE for node 7:
It is NOT a BST

BST and Reality

- C++ **maps** are internally represented as binary search trees.
  - While the standard does not require this, it is implicit in the performance requirements for the data type.
- **Key** data type requires a total ordering.
  - Common examples include numbers ordered by size, strings ordered lexically, year/month/date triples ordered chronologically.
- One node is the *root node* of the tree
  - For each node, node.left.key < node.key < node.right.key

BST and Reality

```
std::map<std::string, int>::iterator it =
                    myMap.find ( "Key" );
```

Since std::string is the key data type, the structure must ensure an internal organization to make the *find* function feasible in a reasonable time.

Outline

- Trees introduction
- Binary search trees
- Traversing algorithms
- Searching a BST
- Insert and Delete in a BST
- Tree balancing

Traversing

It is possible to define three different BST traversing:

– Preorder:

  first the node, then its children

– Inorder:

  first the left child, then the node, and finally the right child.

– Postorder:

  first the two children, then the node.

Preorder

```
Preorder-Tree-Walk(x)
    if x ≠ NULL
        then print key[x]
        Preorder-Tree-Walk(left[x])
        Preorder-Tree-Walk(right[x])
```

Inorder

```
Inorder-Tree-Walk(x)
    if x ≠ NULL
        then Inorder-Tree-Walk(left[x])
        print key[x]
        Inorder-Tree-Walk(right[x])
```

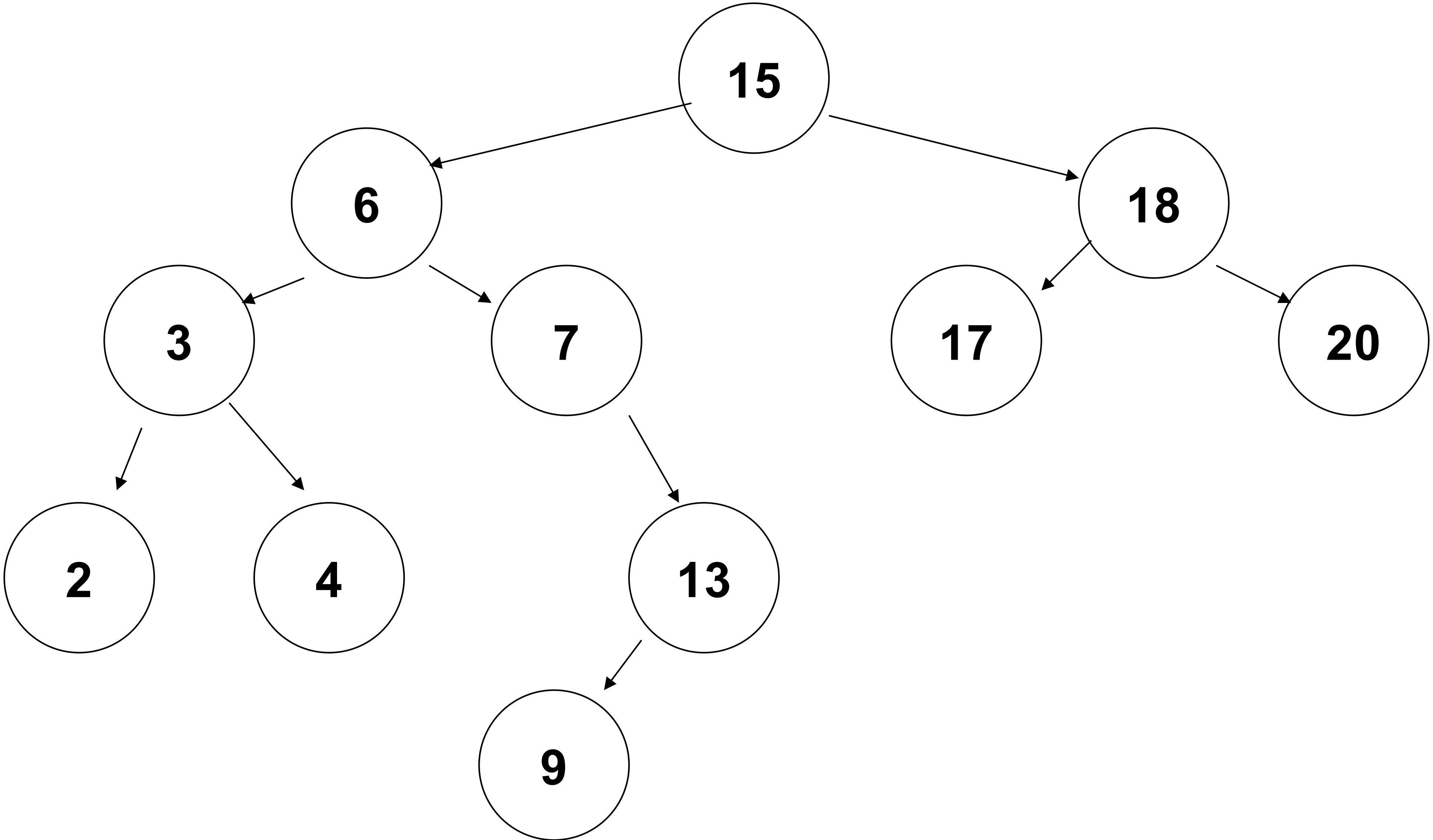
Postorder

```
Postorder-Tree-Walk(x)
    if x ≠ NULL
        then Postorder-Tree-Walk(left[x])
        Postorder-Tree-Walk(right[x])
        print key[x]
```

Notes

- **The Inorder traversing visits all the elements in ascending order (of the key field).**
- **All traversals have complexity equal to $\Theta(n)$, since each node is considered exactly once.**

Example



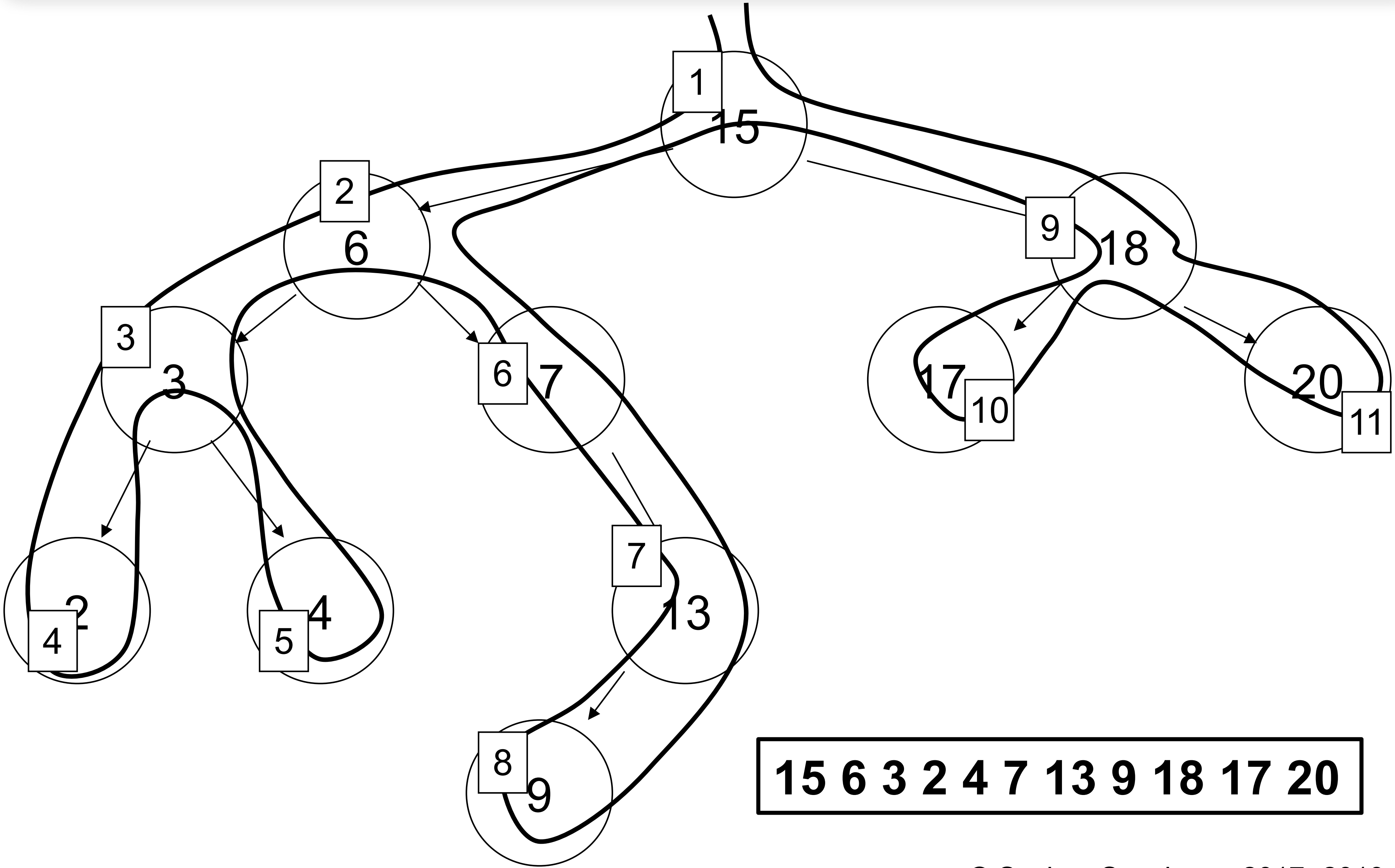
Exercise

Show the three possible traversals for the BST in the previous slide.

Hints for Pre-order

1. **Draw a line along the tree**
2. **Walk through it counterclockwise**
3. **Visit a node the 1st time you reach it**

Solution (Preorder)

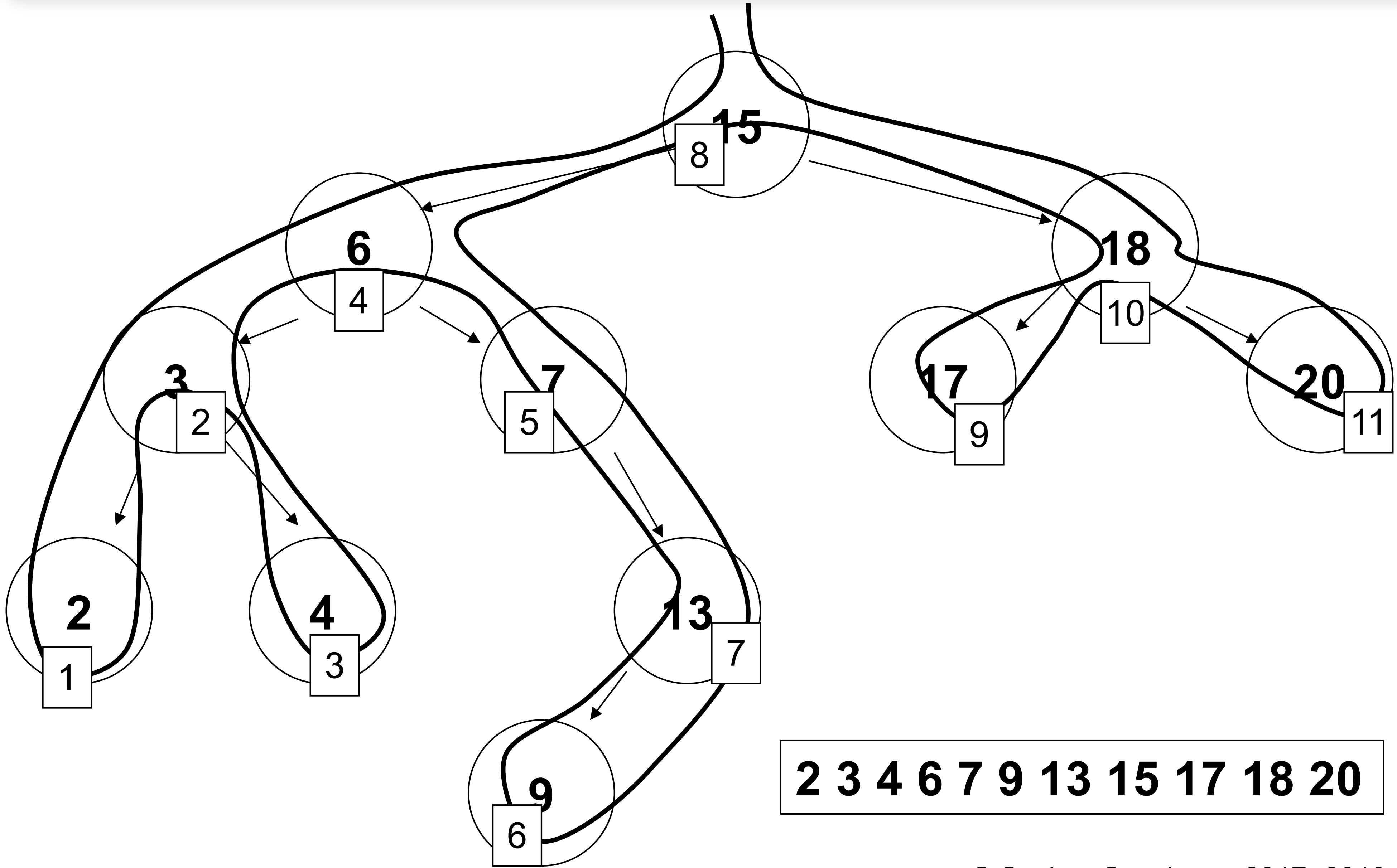


15 6 3 2 4 7 13 9 18 17 20

Hints for In-order

1. **Draw a line along the tree**
2. **Walk through it counterclockwise**
3. **Visit a node the 2^{nd} time you reach it**

Solution (Inorder)



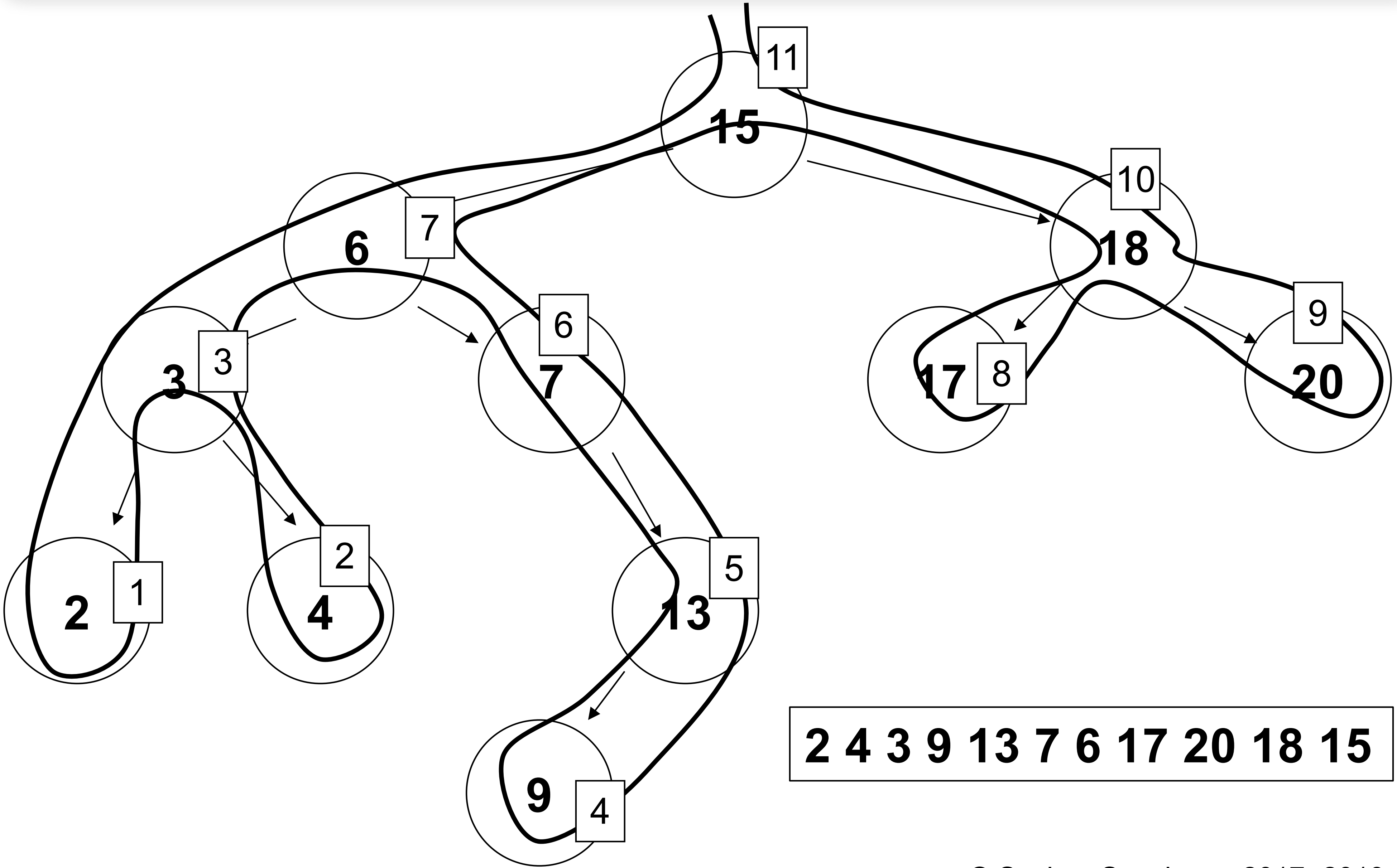
2 3 4 6 7 9 13 15 17 18 20

 Rel. 10/06/2019

Hints for Post-order

1. Draw a line along the tree
2. Walk through it counterclockwise
3. Visit a node the last time you reach it

Solution (Postorder)



2 4 3 9 13 7 6 17 20 18 15

Outline

- Trees introduction
- Binary search trees
- Traversing algorithms
- Searching a BST
- Insert and Delete in a BST
- Tree balancing

Searching a BST

BSTs are particularly optimized for search operations:

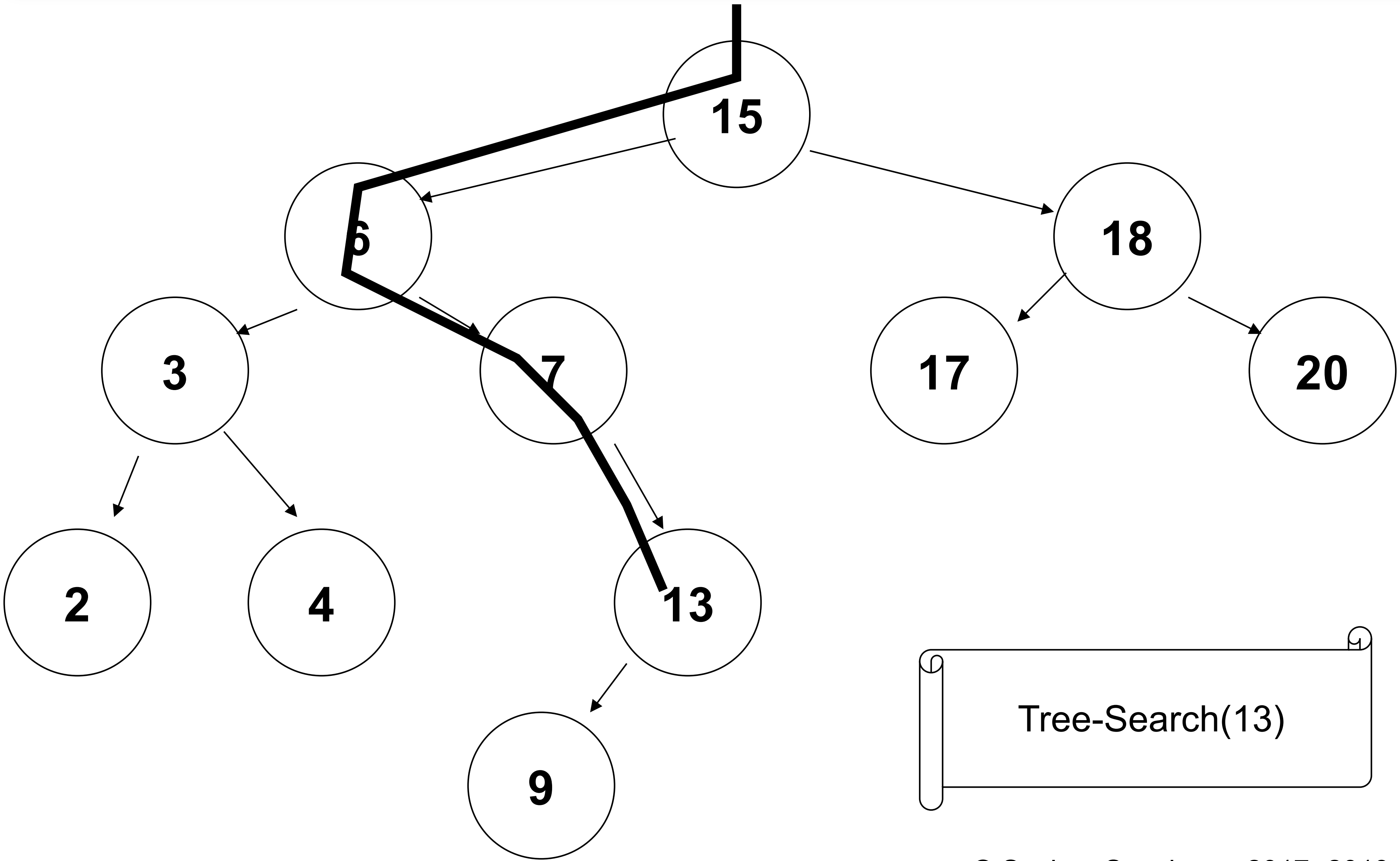
- Search
- Minimum/Maximum
- Predecessor/Successor.

Their complexity is O(h), where h is the height of the tree.

Tree-Search

```
Tree-Search(x, k)
    if x = NULL or k = key[x]
        then return x
    if k < key[x]
        then return Tree-Search(left[x], k)
        else return Tree-Search(right[x], k)
```

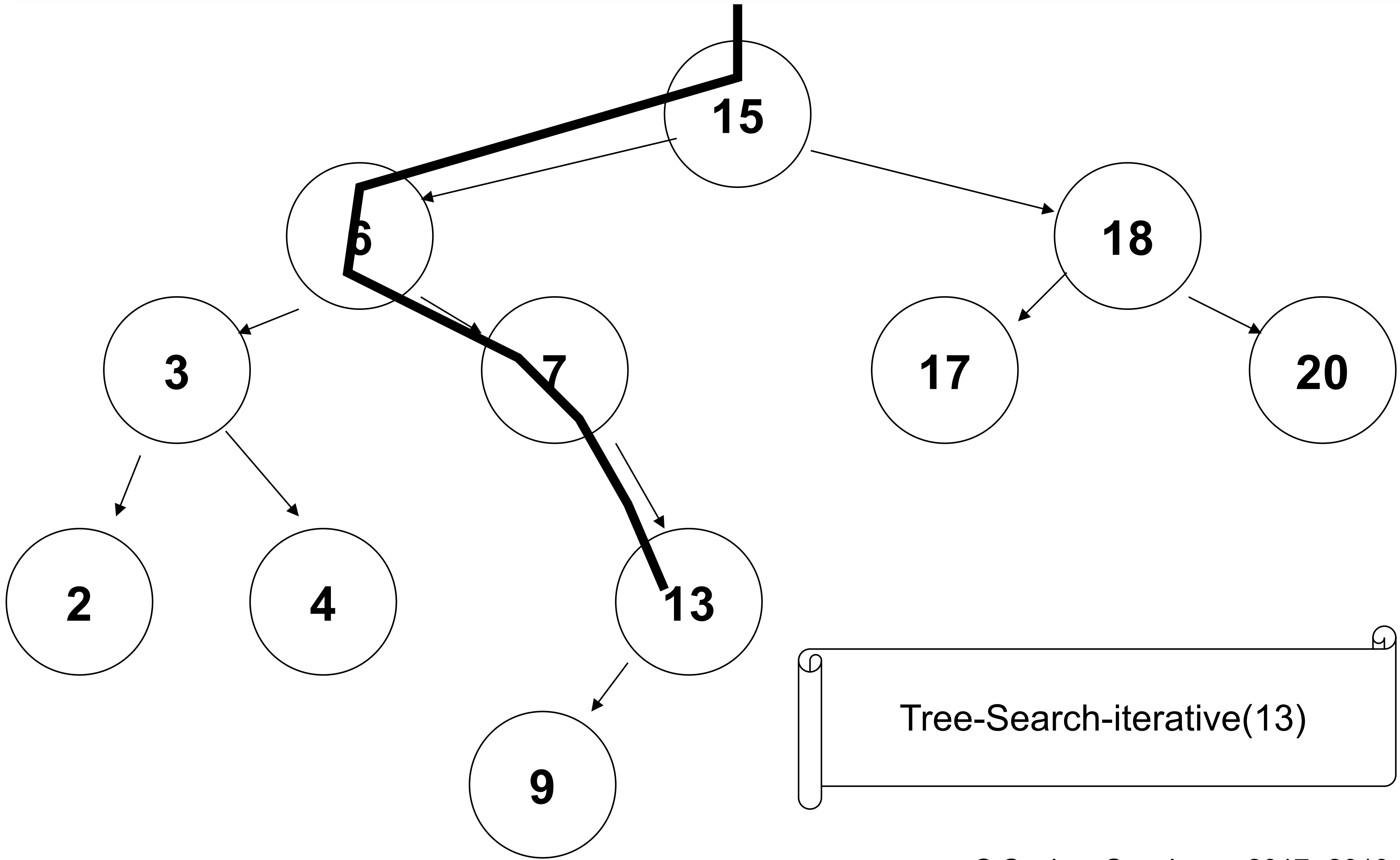
Example



Tree-Search (iterative)

```
Tree-Search-iterative(x, k)
    while x ≠ NULL and k ≠ key[x]
        do if k < key[x]
            then x ← left[x]
            else x ← right[x]
    return x
```

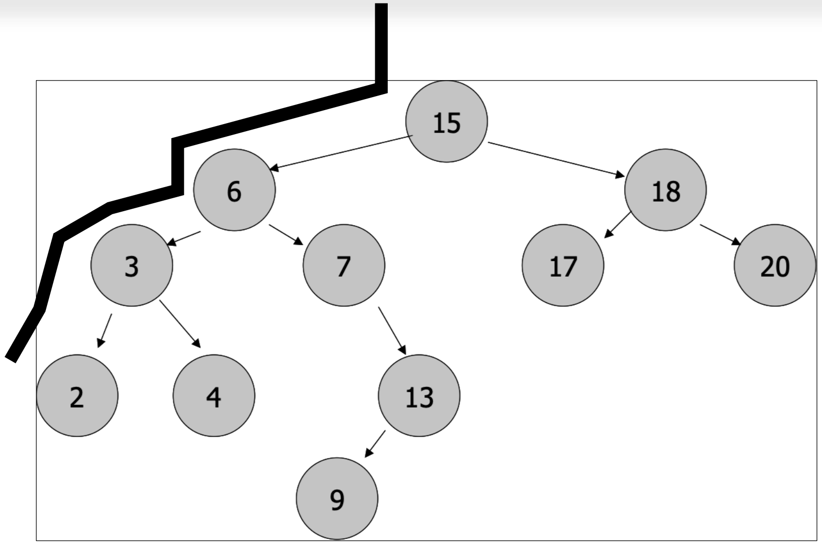
Example



Min and Max (iterative)

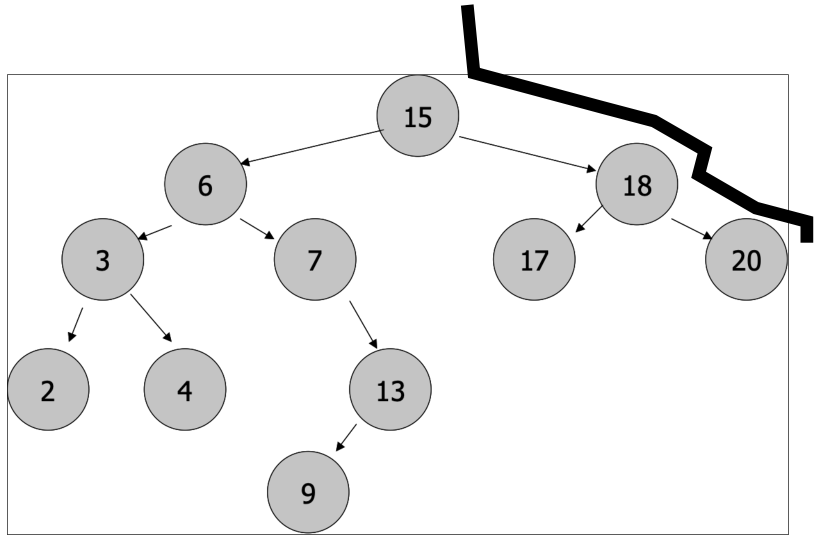
Tree-Minimum(x)

```
while left[x] ≠ NULL
        do x ← left[x]
return x
```



Tree-Maximum(x)

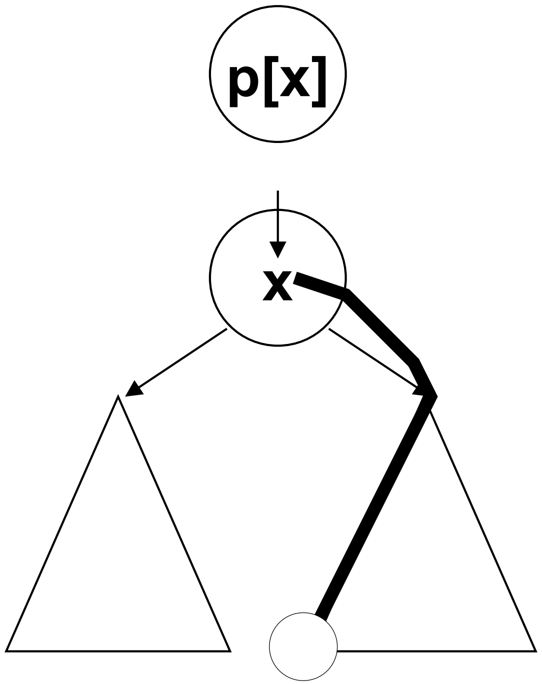
```
while right[x] ≠ NULL
        do x ← right[x]
return x
```

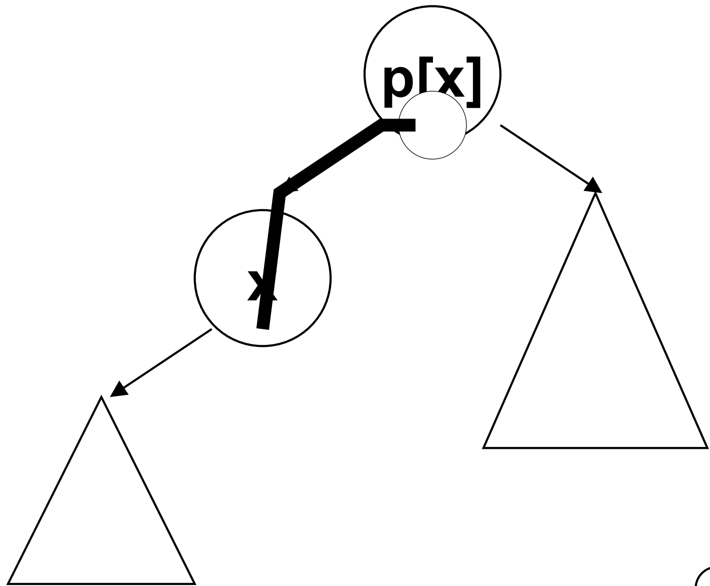


Successor

- **Given a node x, find the next element. There are 2 possible situations**

The minimum of the right subtree



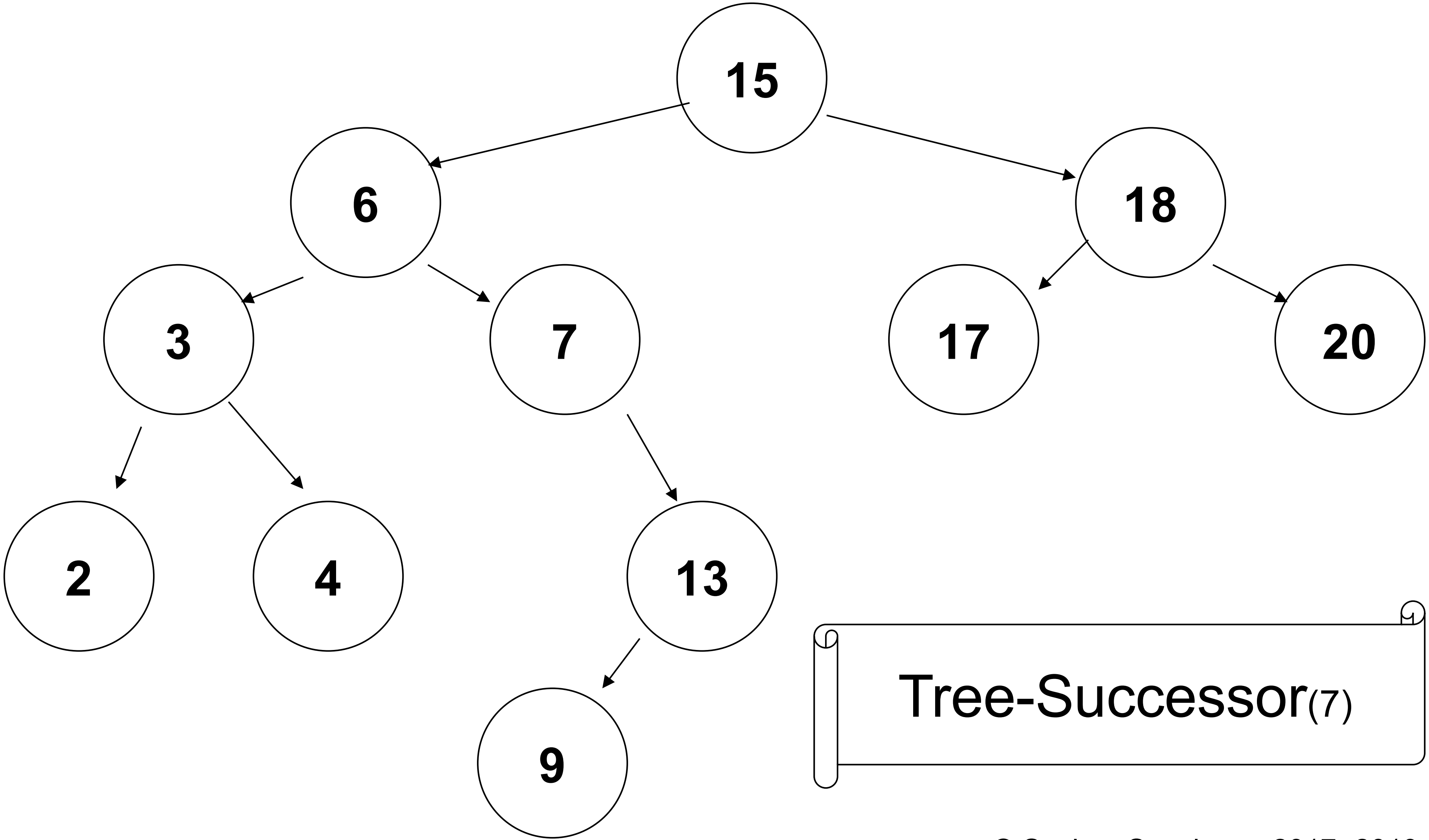


The first father for which x is in the left subtree

Successor

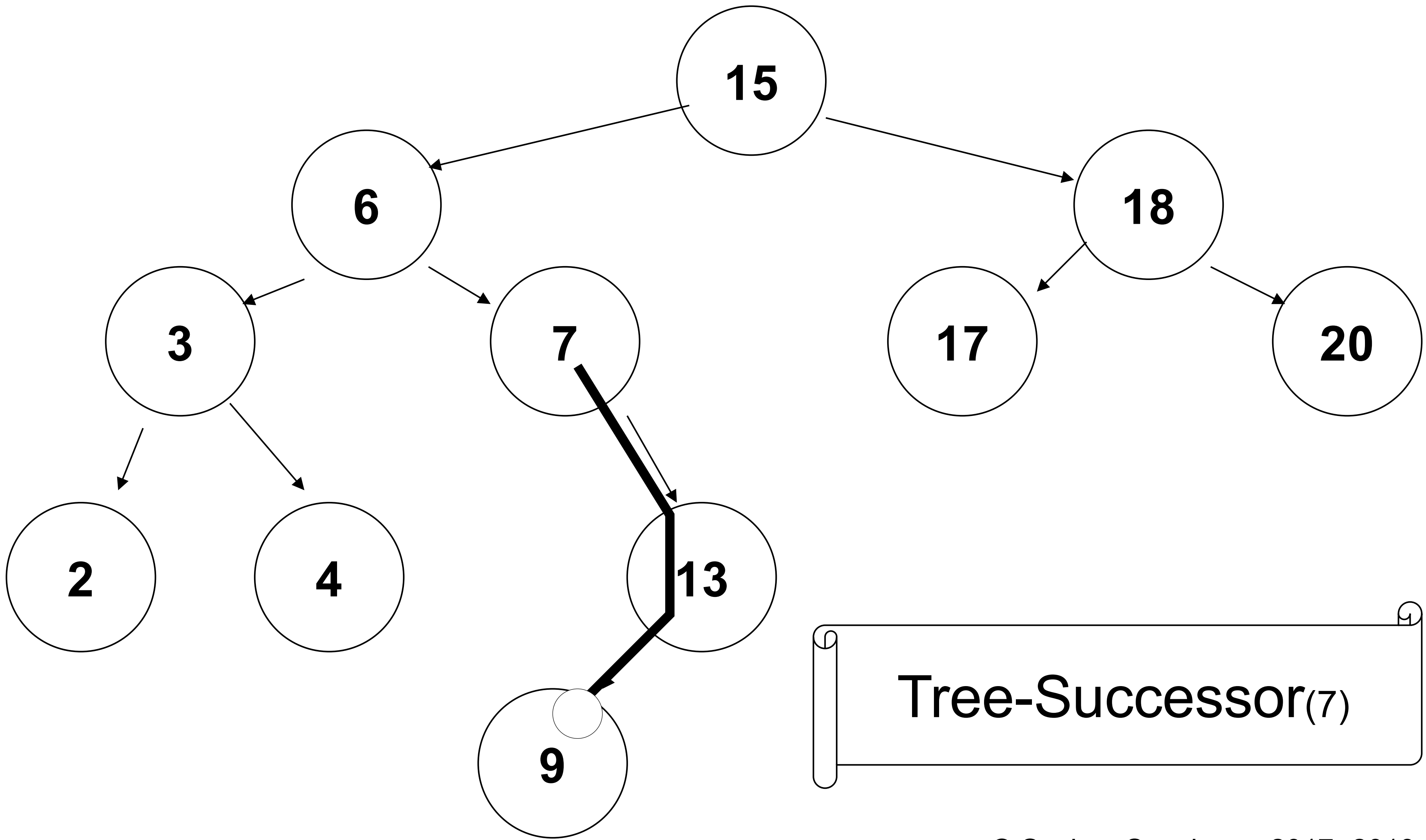
```
Tree-Successor(x)
    if right[x] ≠ NULL
        then return Tree-Minimum(right[x])
    y ← p[x]
    while y ≠ NULL and x = right[y]
        do x ← y
        y ← p[y]
    return y
```

Example

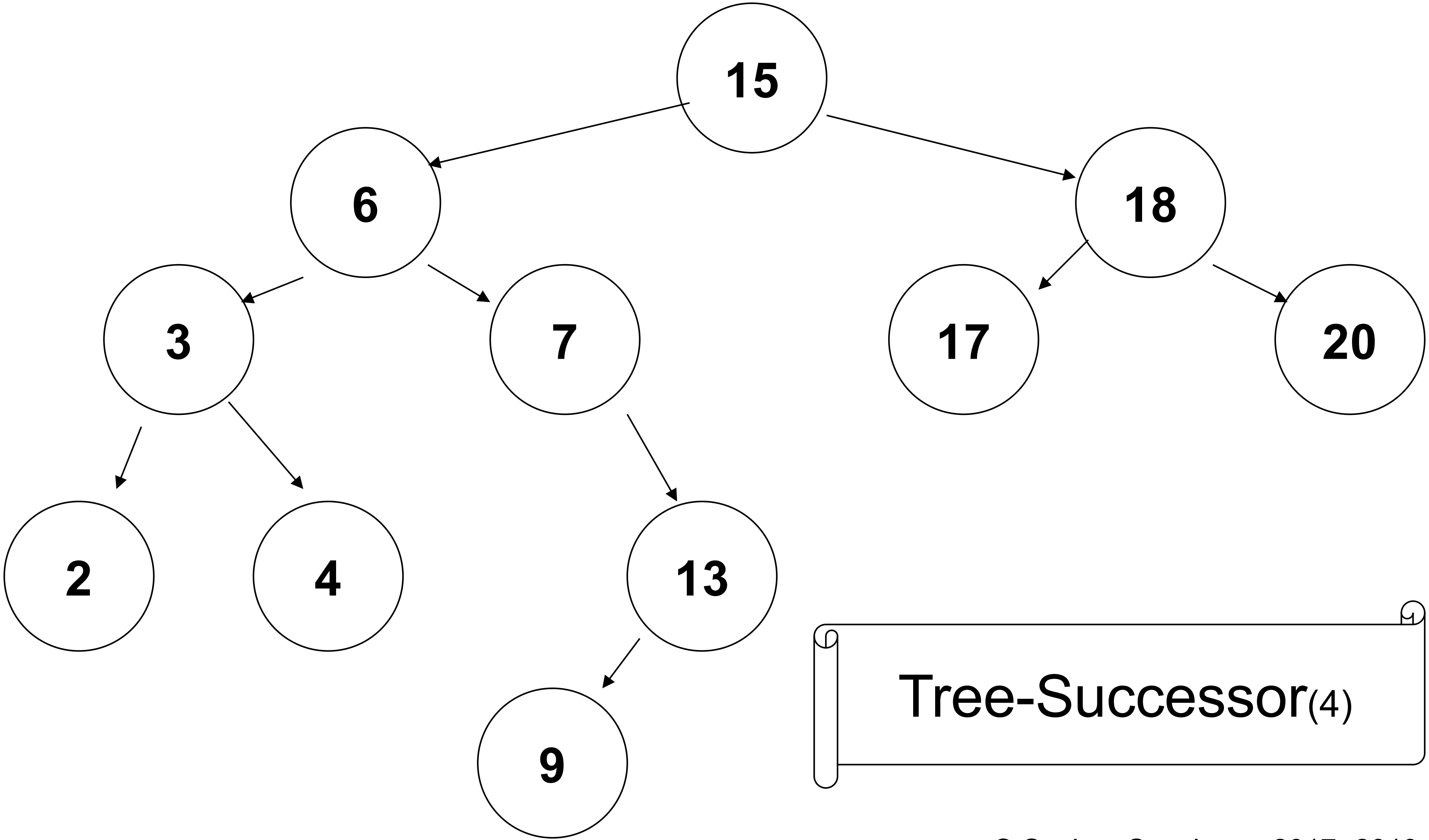


Tree-Successor(7)

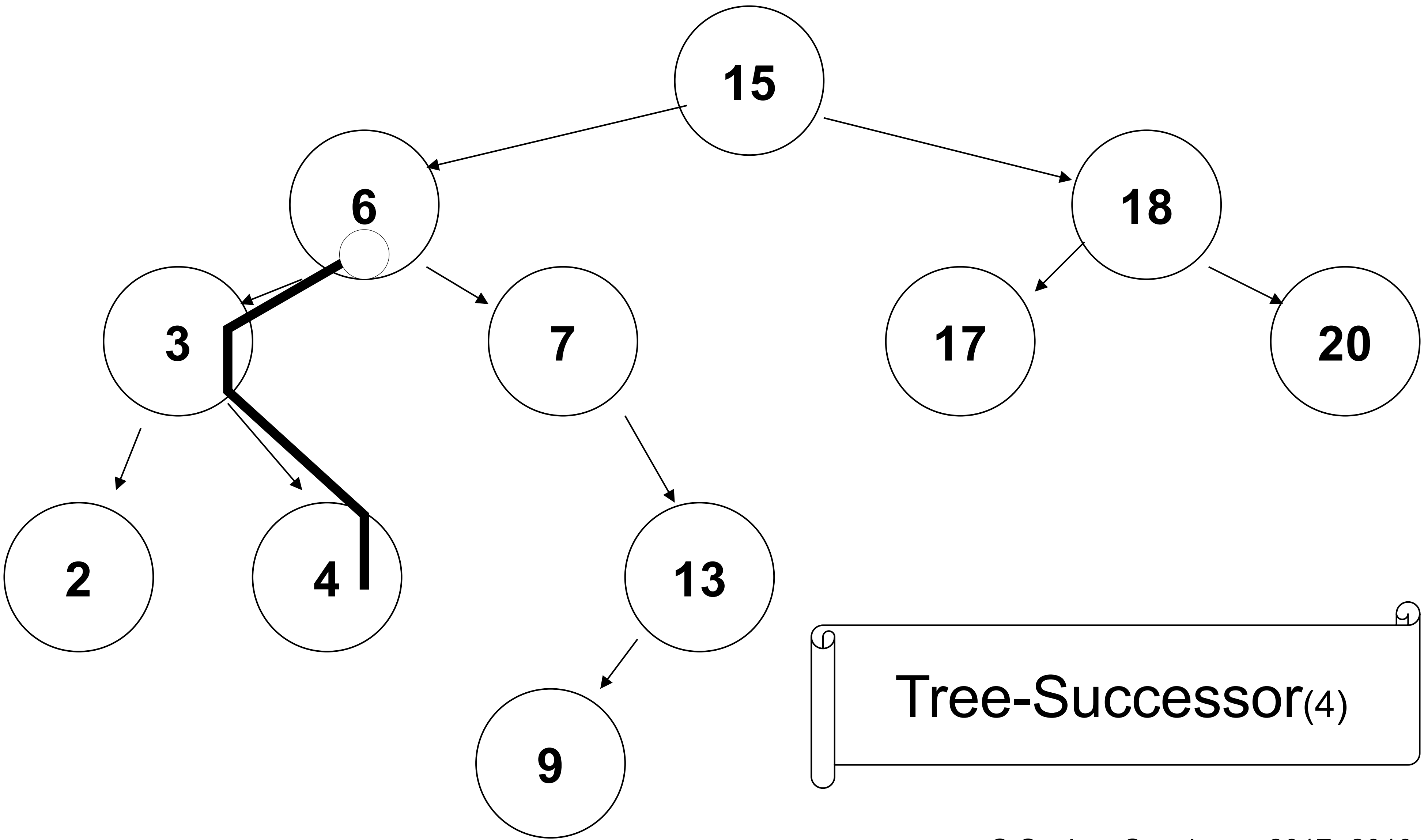
Example



Example



Example



Predecessor

```
Tree-Predecessor(x)
    if left[x] ≠ NULL
        then return Tree-Maximum(left[x])
    y ← p[x]
    while y ≠ NULL and x = left[y]
        do x ← y
        y ← p[y]
    return y
```


Complexity

- The complexity for all search operations is O(h).

 Rel. 10/06/2019

Outline

- Trees introduction
- Binary search trees
- Traversing algorithms
- Searching a BST
- Insert and Delete in a BST
- Tree balancing

Insert and Delete

- **The issue with these operations is to maintain the sorting criteria while adding or deleting nodes.**

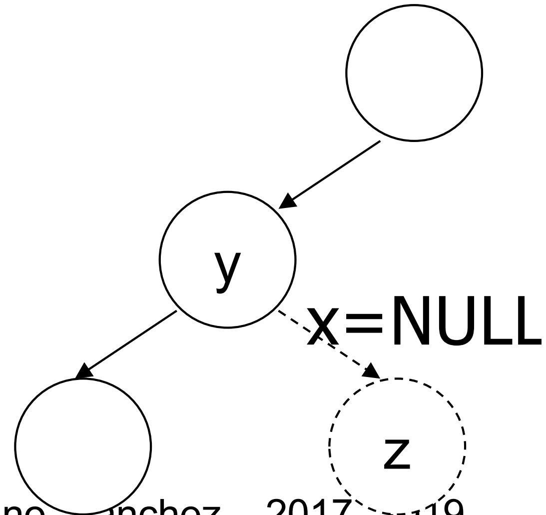
Insert

- **Insert node z with key v:**
  - **Create a new node z with**
    - **left[z] = right[z] = NULL**
  - **The correct insert location is fond by simulating a search for key[z]**
  - **Left and right pointers are then updated accordingly**
- **The new node is always inserted as a leaf.**

Tree-Insert (I)

```
Tree-Insert(T, z)
    y ← NULL
    x ← root[T]
    while x ≠ NULL
        do y ← x
        if key[z]<key[x]
            then x ← left[x]
            else x ← right[x]
```

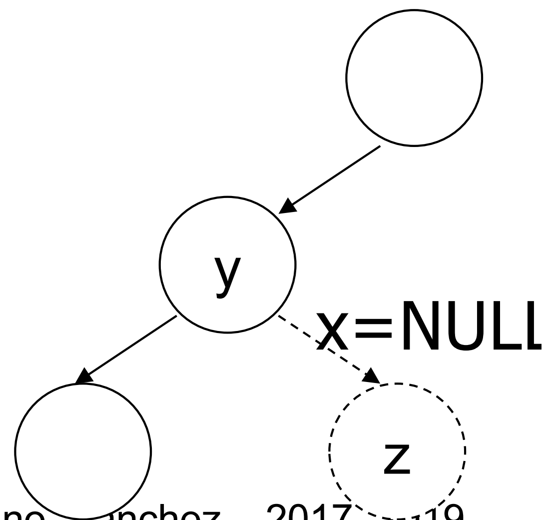
Search key[z] in the tree



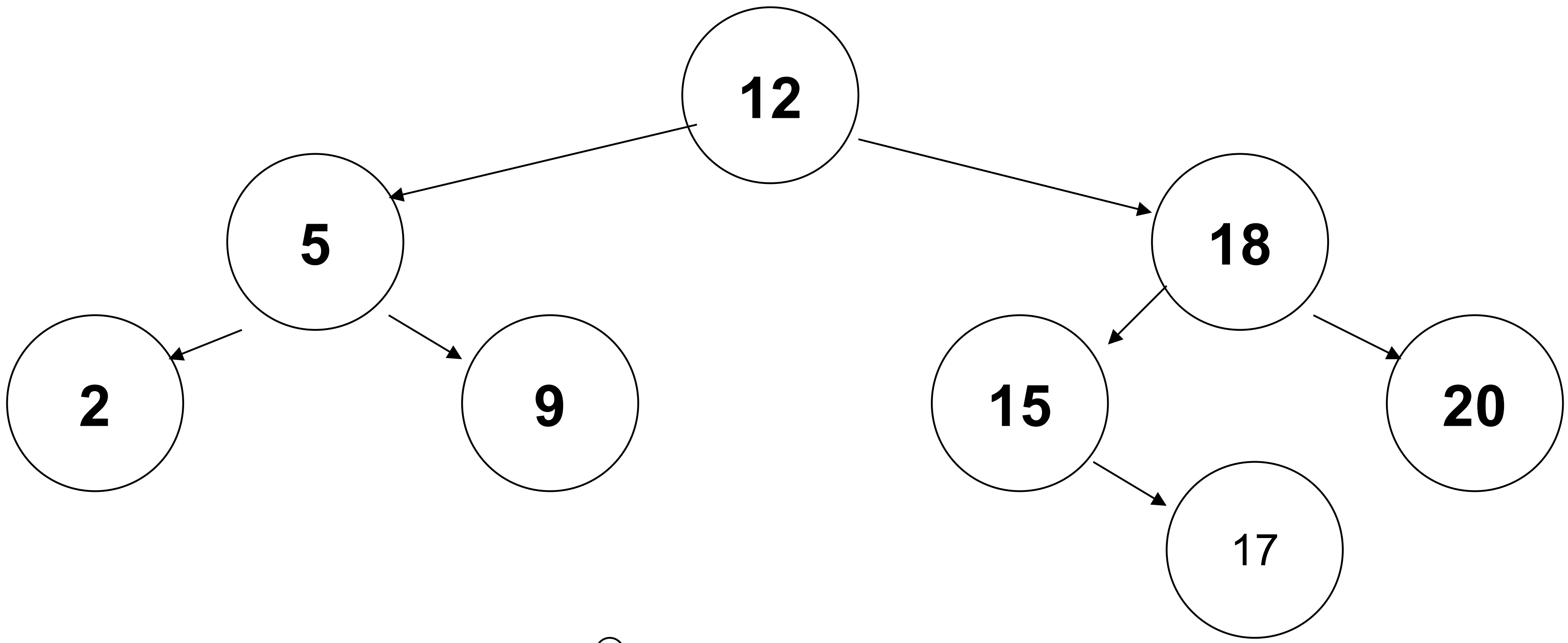
Tree-Insert (II)

```
p[z] ← y
if y = NULL
        then root[T] ← z
else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
```

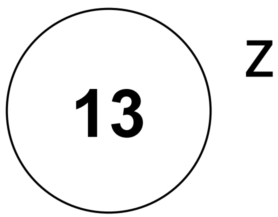
Insert z as child of y



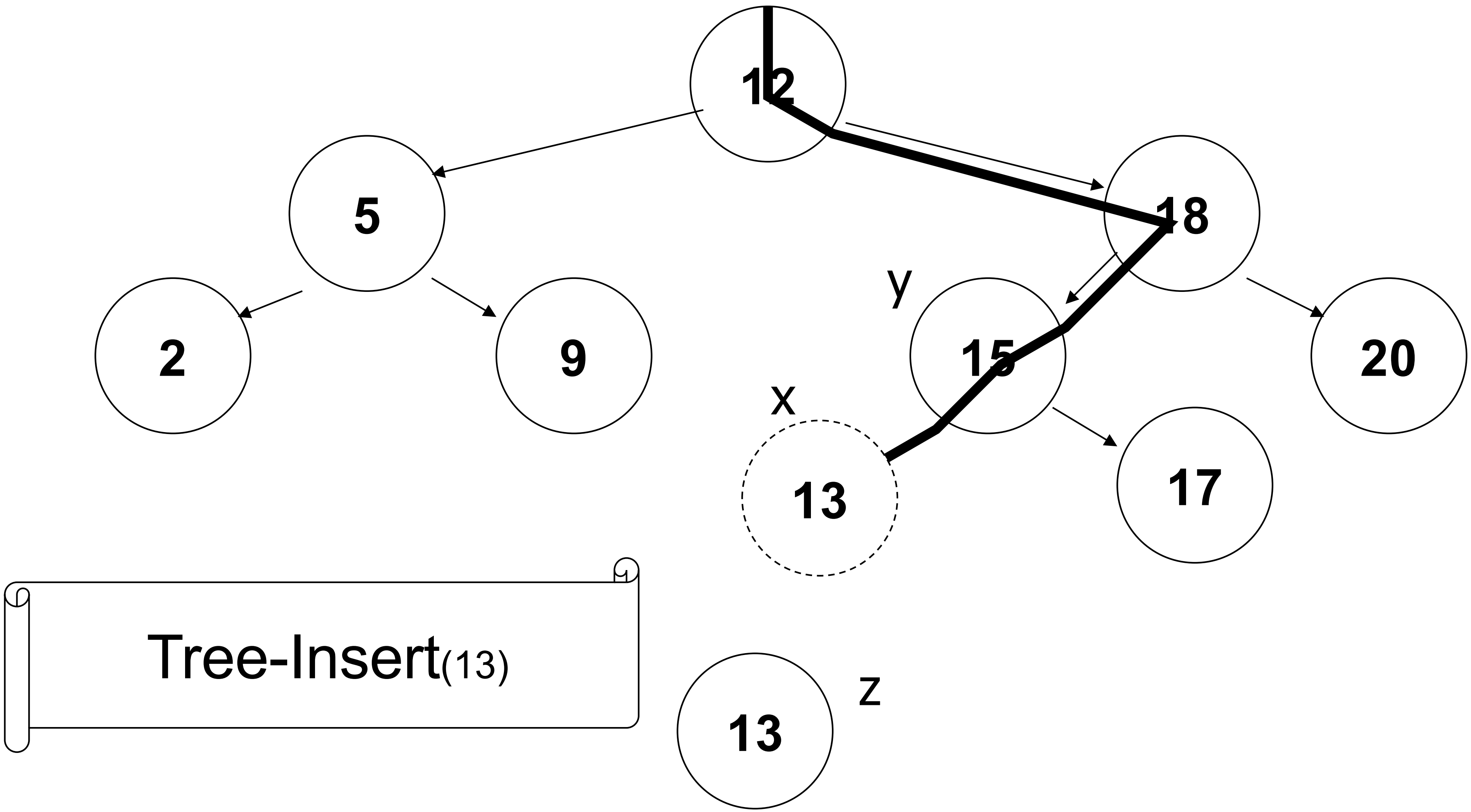
Example



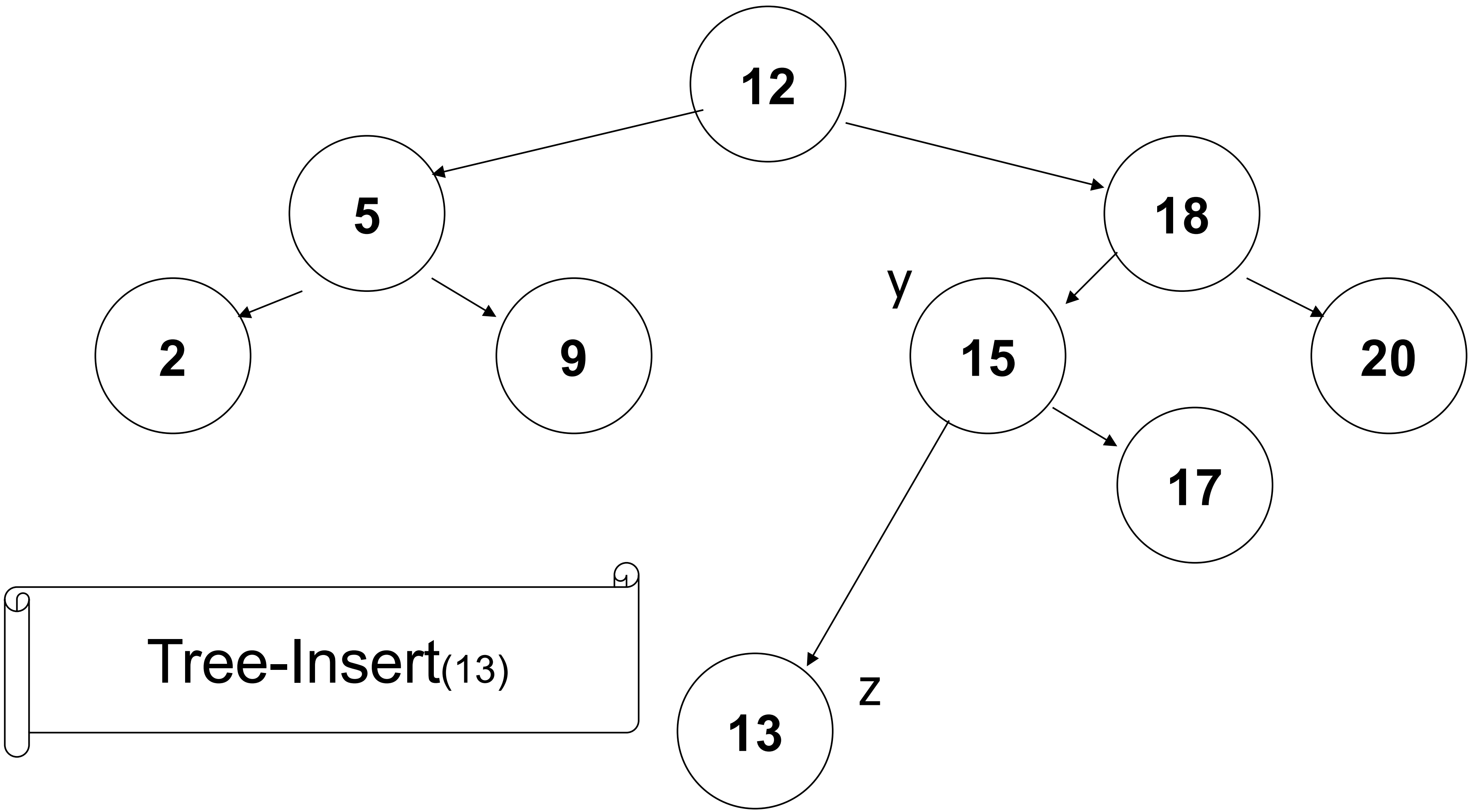
Tree-Insert(13)



Example



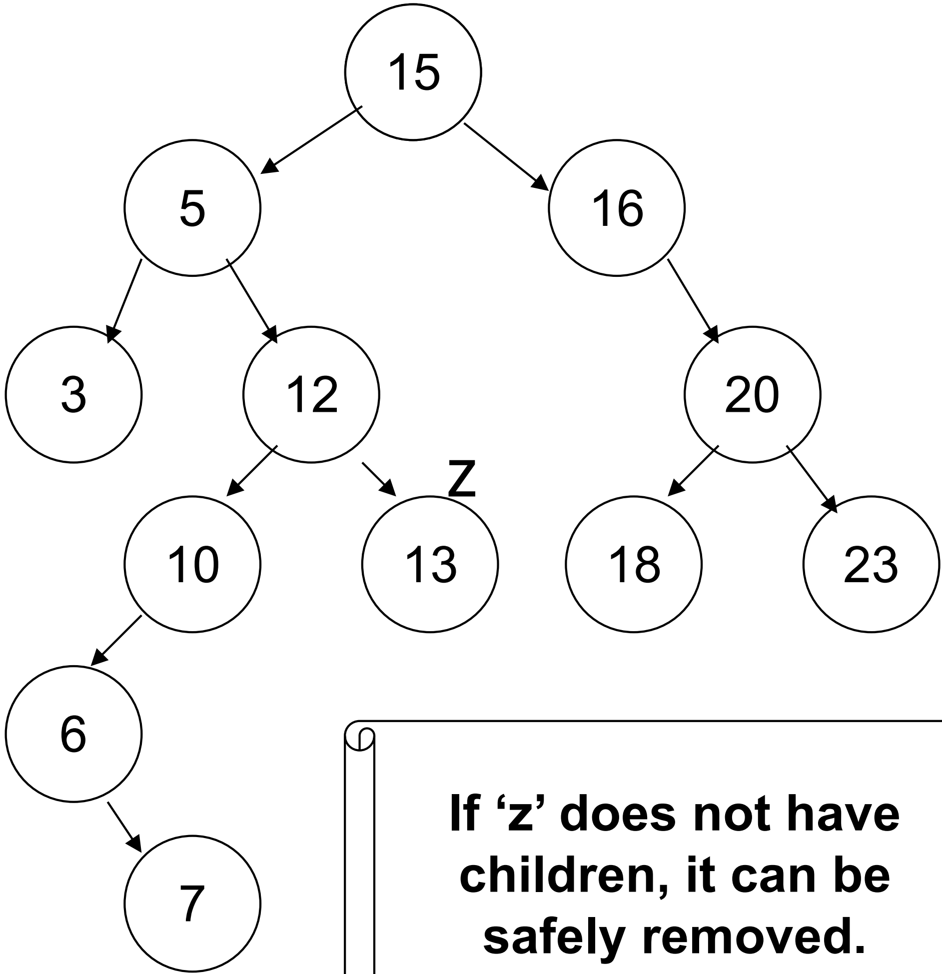
Example

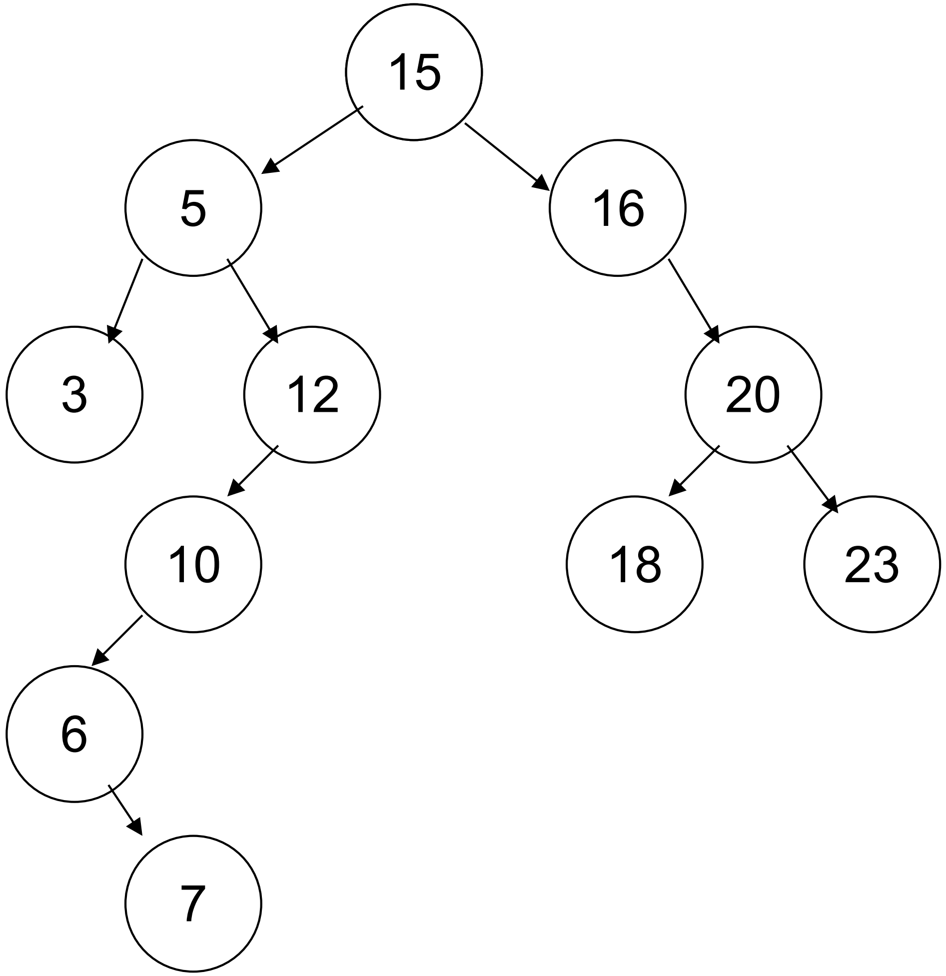


Delete

Deletion is the most complex operation on a BST. There are 3 situations, depending on the number of children of the deleted node.

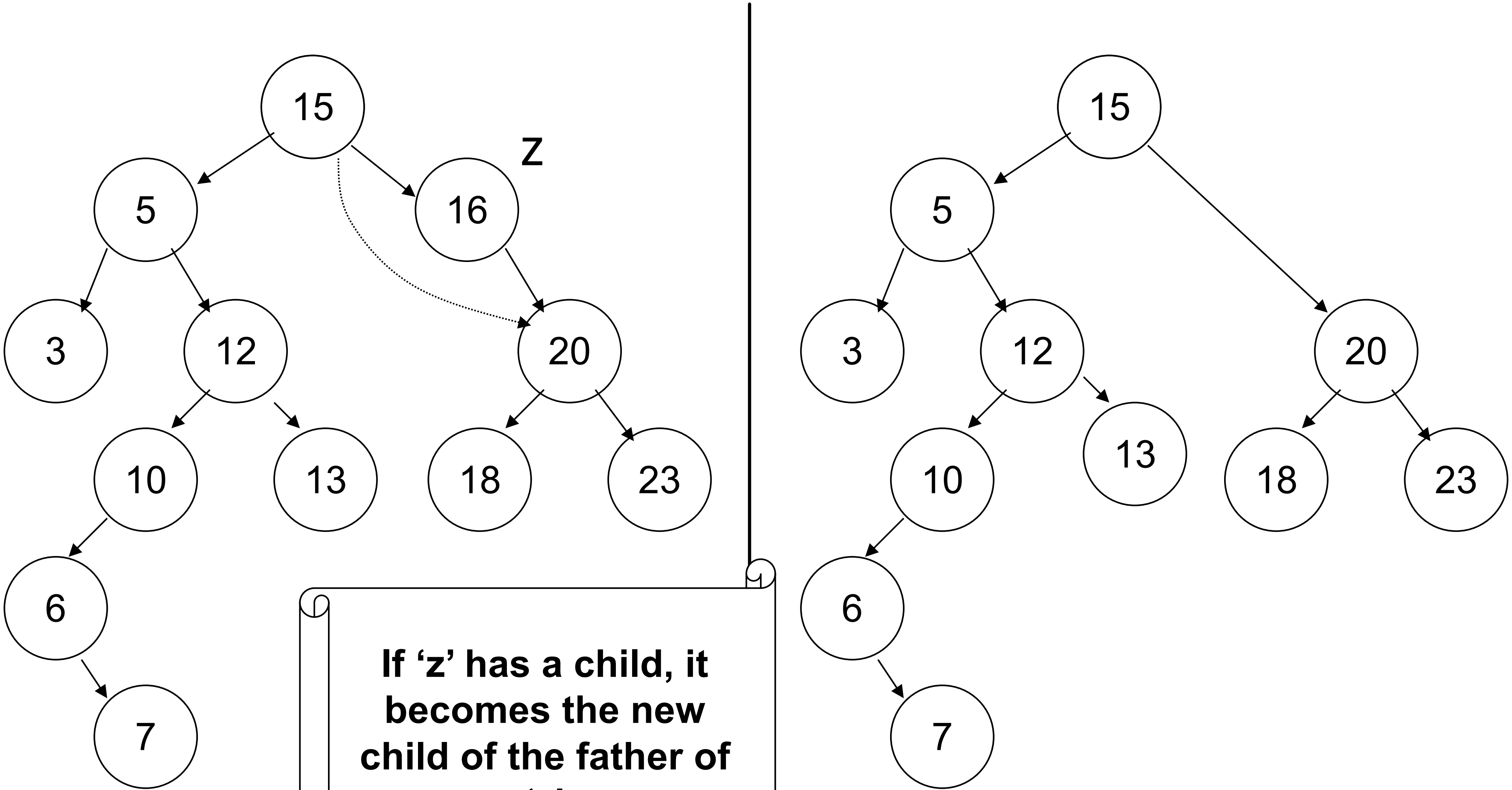
Possible cases: 0 children





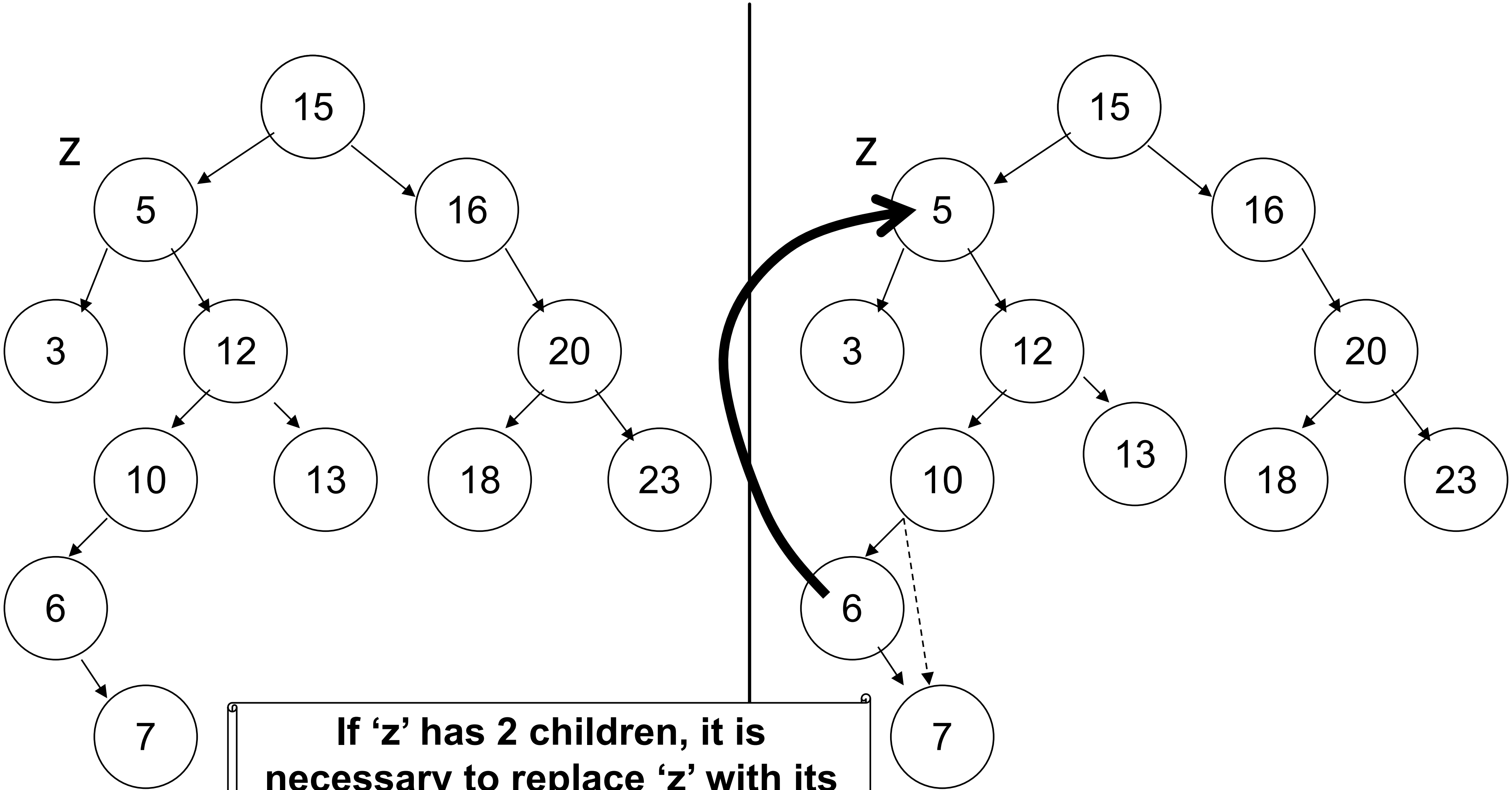
If 'z' does not have children, it can be safely removed.

Possible cases: 1 child



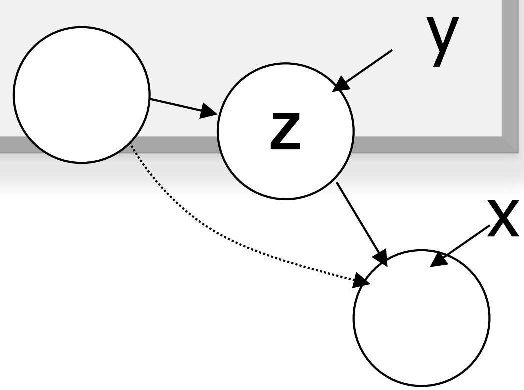
If ‘z’ has a child, it becomes the new child of the father of ‘z’

Possible cases: 2 children



If 'z' has 2 children, it is necessary to replace 'z' with its successor

Tree-Delete (I)

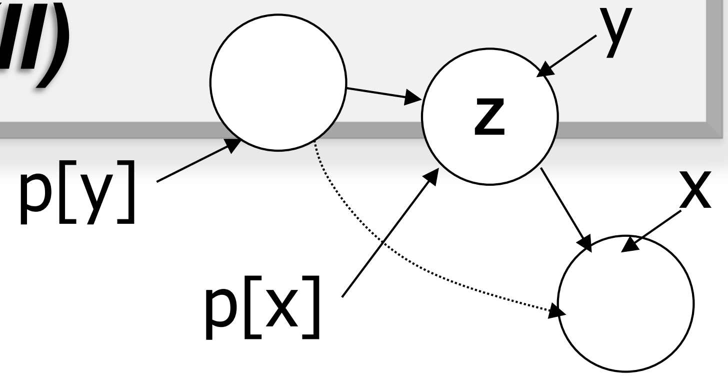



```
Tree-Delete(T, z)
    if left[z]=NULL or right[z]=NULL
        then y ← z
        else y ← Tree-Successor(z)
    if left[y] ≠ NULL
        then x ← left[y]
        else x ← right[y]
```

y: node to delete

x: only child of y

Tree-Delete (II)




```
if x ≠ NULL
        then p[x] ← p[y]
if p[y] = NULL
        then root[T] = x
        else if y = left[p[y]]
                then left[p[y]] ← x
                else right[p[y]] ← x
```

Update x's father

y is the root? Then x becomes the root

If not, link x to the father of y

Tree-Delete (III)

```
if y ≠ z
        then key[z] ← key[y]
                fields[z] ← fields[y]
return y
```

Possibly, copy the information of the successor of the node to be deleted

Complexity

The complexity of any update operation on a tree (insert or delete) is O(h).

Outline

- Trees introduction
- Binary search trees
- Traversing algorithms
- Searching a BST
- Insert and Delete in a BST
- Tree balancing

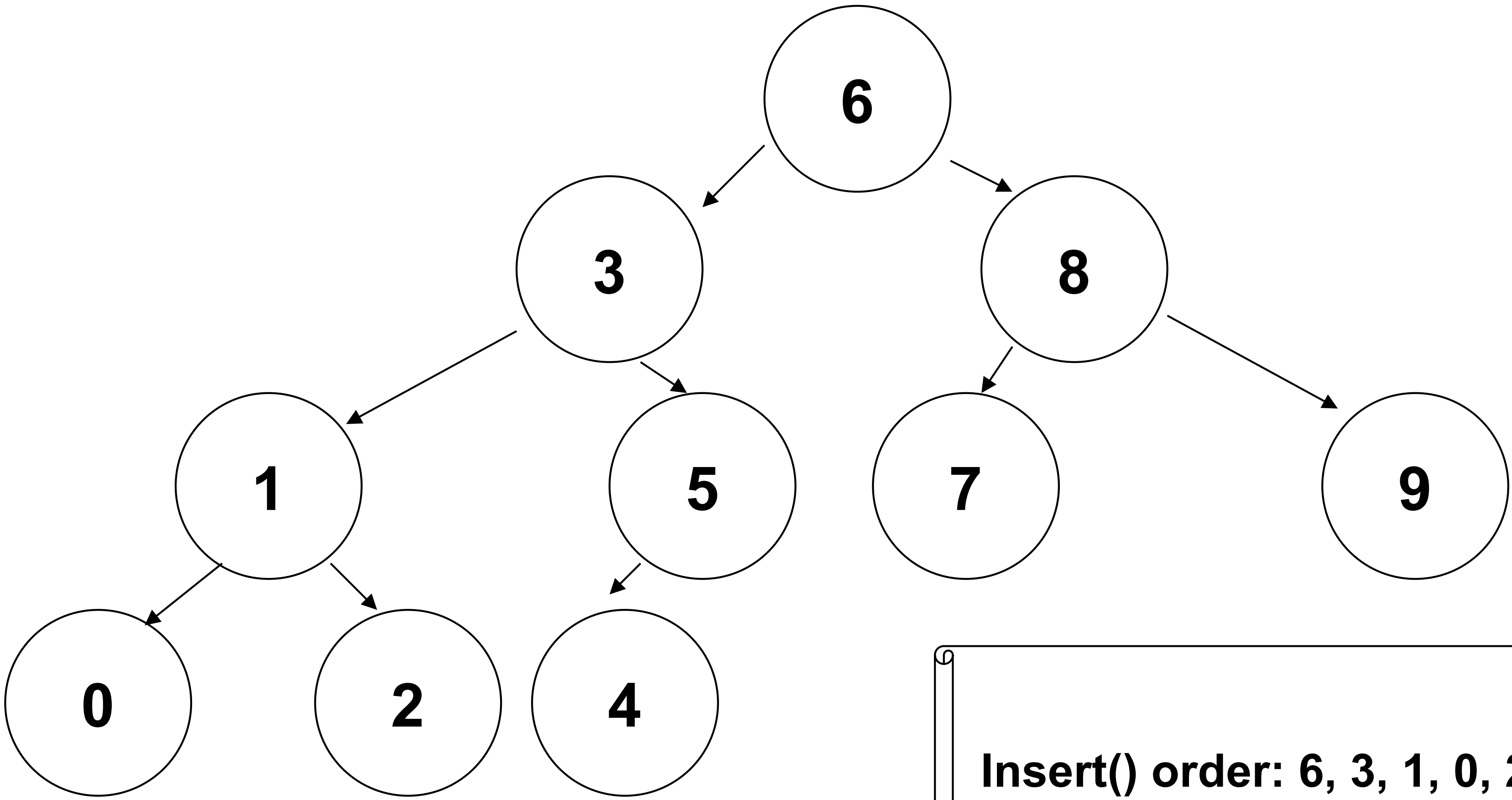
Tree balancing

- Complexity is O(h), where h is the tree height.
  - A balanced tree has
    - $h = \log_2 n$
  - A totally unbalanced tree has
    - $h = n$
  - Therefore the operations on a BST have a variable complexity between $O(\log_2 n)$ and $O(n)$

Exercise

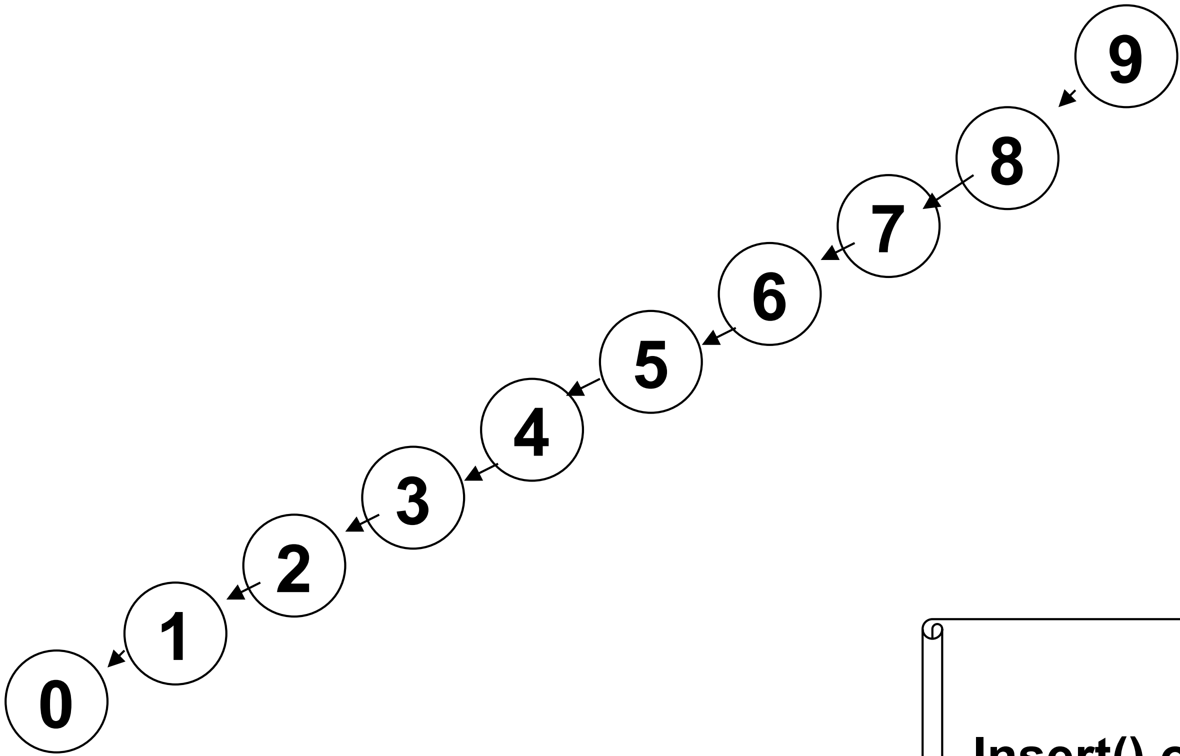
- **We want to build a BST storing all numbers between 0 and 9.**
  - **In which sequence do we have to insert the nodes in order to have a balanced tree?**
  - **In which sequence do we have to insert the nodes in order to have a totally unbalanced tree?**

Solution (I)



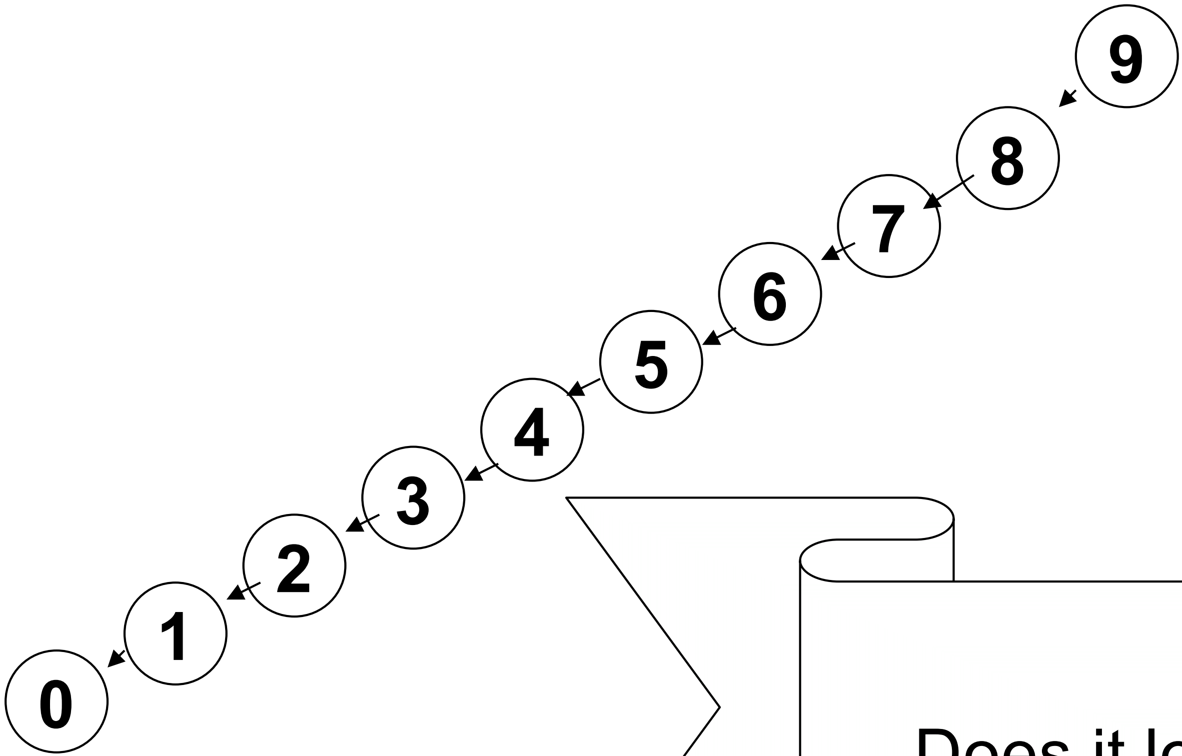
Insert() order: 6, 3, 1, 0, 2, 5, 4, 8, 7, 9

Solution (II)



Insert() order: 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

Solution (II)



Does it look like a balanced one?

Solution (II)

BST are not balanced by definition. In order to have a balanced tree you must implement balancing techniques

Self-balancing BST

References

- A.V. Aho, J.E. Hopcroft, J.D. Ullman:
  "Data Structures and Algorithms,"
  Addison Wesley, Reading MA (USA), 1983
  pp. 75-106
- G.H. Gonnet:
  "Handbook of Algorithms and Data Structures,"
  Addison Wesley, Reading MA (USA), 1984, pp. 69-117
- J. Esakow. T. Weiss
  "Data structure: an advanced approach using C,"
  Prentice Hall, Englewood Cliffs NJ (USA), 1982, pp. 38-59

References

- E. Horowitz, S. Sahni:
  "Fundamentals of Computer Algorithms,"
  Pittman, London (UK), 1978
  pp. 203-271
- R. Sedgewick:
  "Algorithms in C,"
  Addison Wesley, Reading MA (USA), 1990
  pp. 35-50
- C.J. Van Wyk:
  "Data Structures and C Programs," Addison Wesley, Reading MA (USA), 1988
  pp 159-176

References

- **M.A. Weiss:**
  "Data Structures and Algorithm Analysis,"
  The Benjamin/Cummings Publishing Company,
  Redwood City, CA (USA), 1992, pp. 87-98
- **R.J. Wilson:**
  "Introduzione alla teoria dei grafi,"
  Cremonese, Roma 1978, pp. 57-76
- **N. Wirth:**
  "Algorithms + Data Structures = Programs,"
  Prentice Hall, Englewood Cliffs NJ (USA), 1976
  pp. 169-263

Малые Автюхи, Калинковичский район, Республики Беларусь