fit@hcmus

DATA STRUCTURES & ALGORITHMS

Lecture 6: TREES – Part 1

Binary Tree, Binary Search Tree

Lecturer: Dr. Nguyen Hai Minh

CONTENT

- Introduction
  - Trees
  - Binary trees
  - Binary search trees
- Implementing binary trees
- Tree traversal
- Querying, insertion, deletion a binary search tree
- Balancing a tree
- Heap – Priority queue

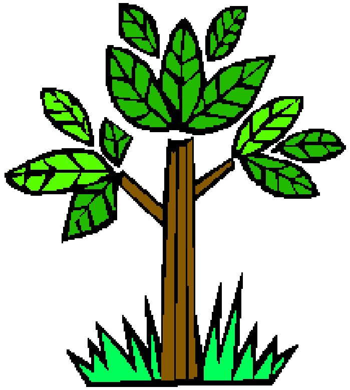
TREES

- The Tree ADT
- Tree Traversal

Introduction

- Arrays:
  - Static → inflexible
  - Search: $O(\log_2 n)$ (ordered array)
- Linked lists:
  - Dynamic → difficult to represent the hierarchical structure of objects.
  - Insert/delete: $O(1)$
- Stacks, queues:
  - Limited to one dimension

→ **Trees**

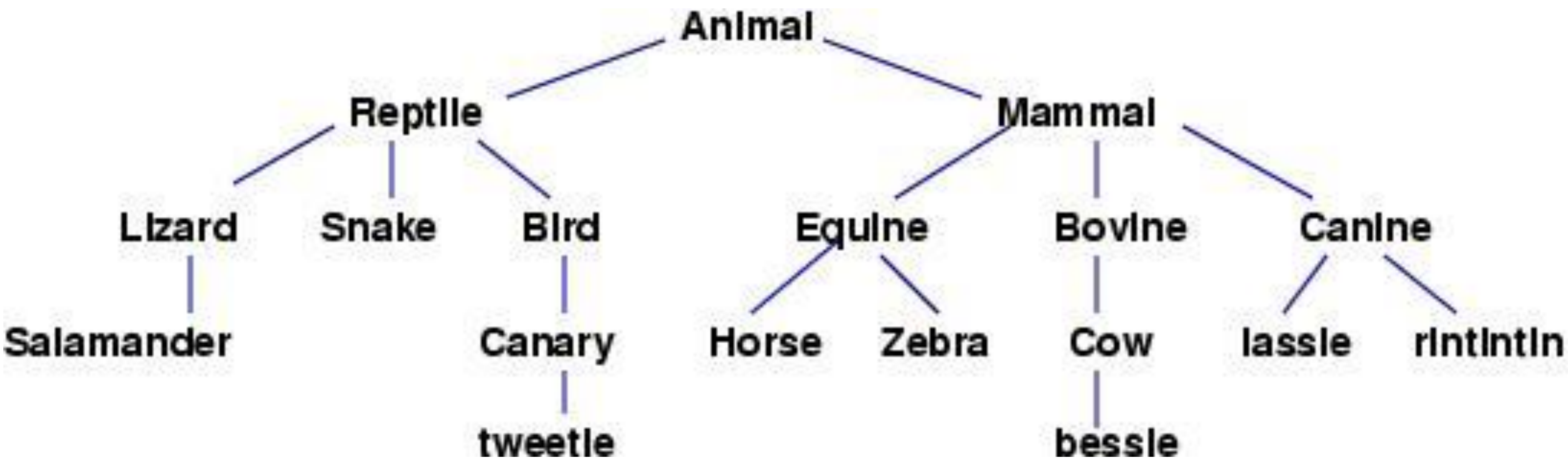


Trees

- Fundamental data storage structures used in programming.
- Combines advantages of an ordered array and a linked list.
- Searching as fast as in ordered array.
- Insertion and deletion as fast as in linked list.

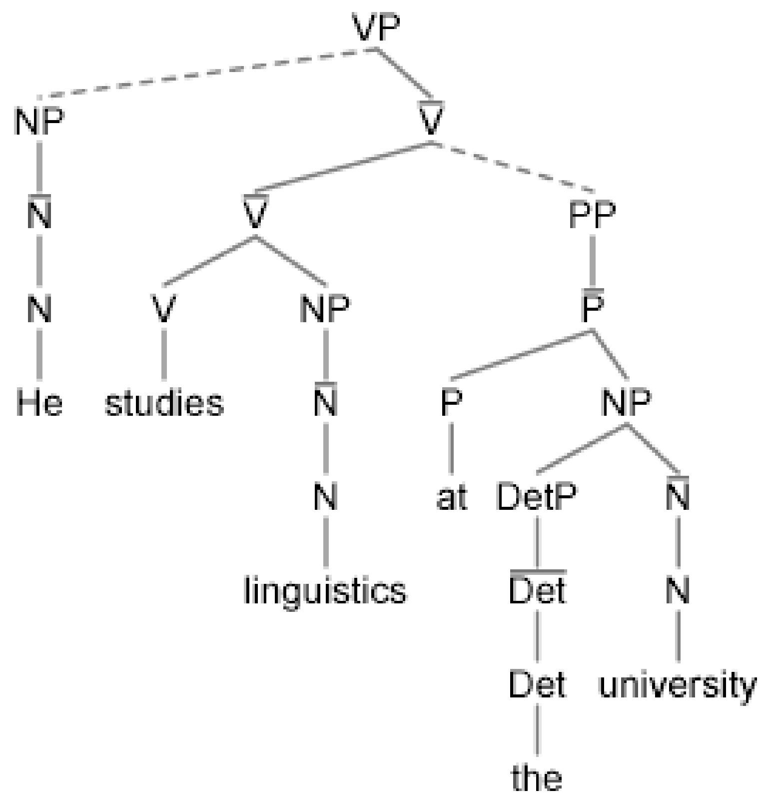
Trees – Example

- Species tree:



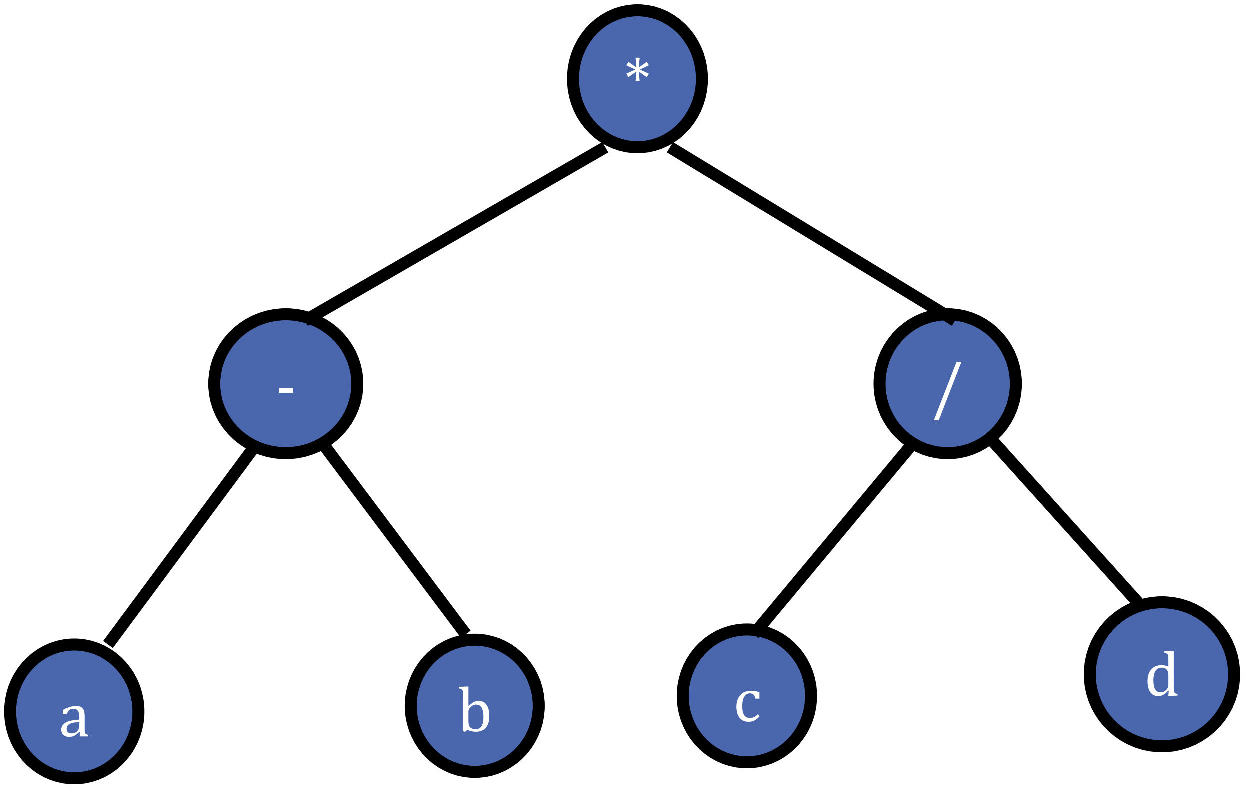
Trees – Example

- Parse tree of a sentence:



Trees – Example

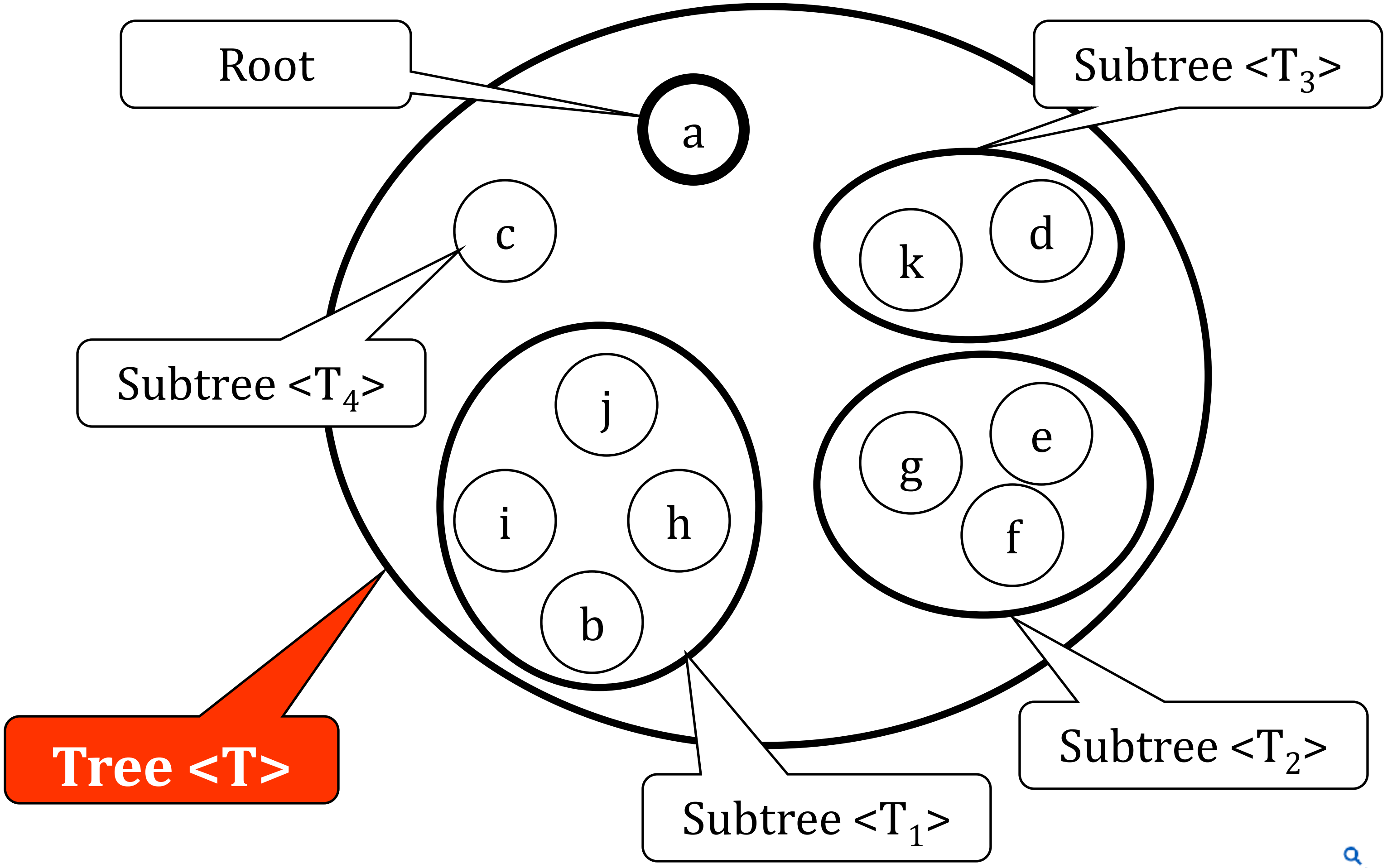
- A tree of the expression (a-b)*(c/d):



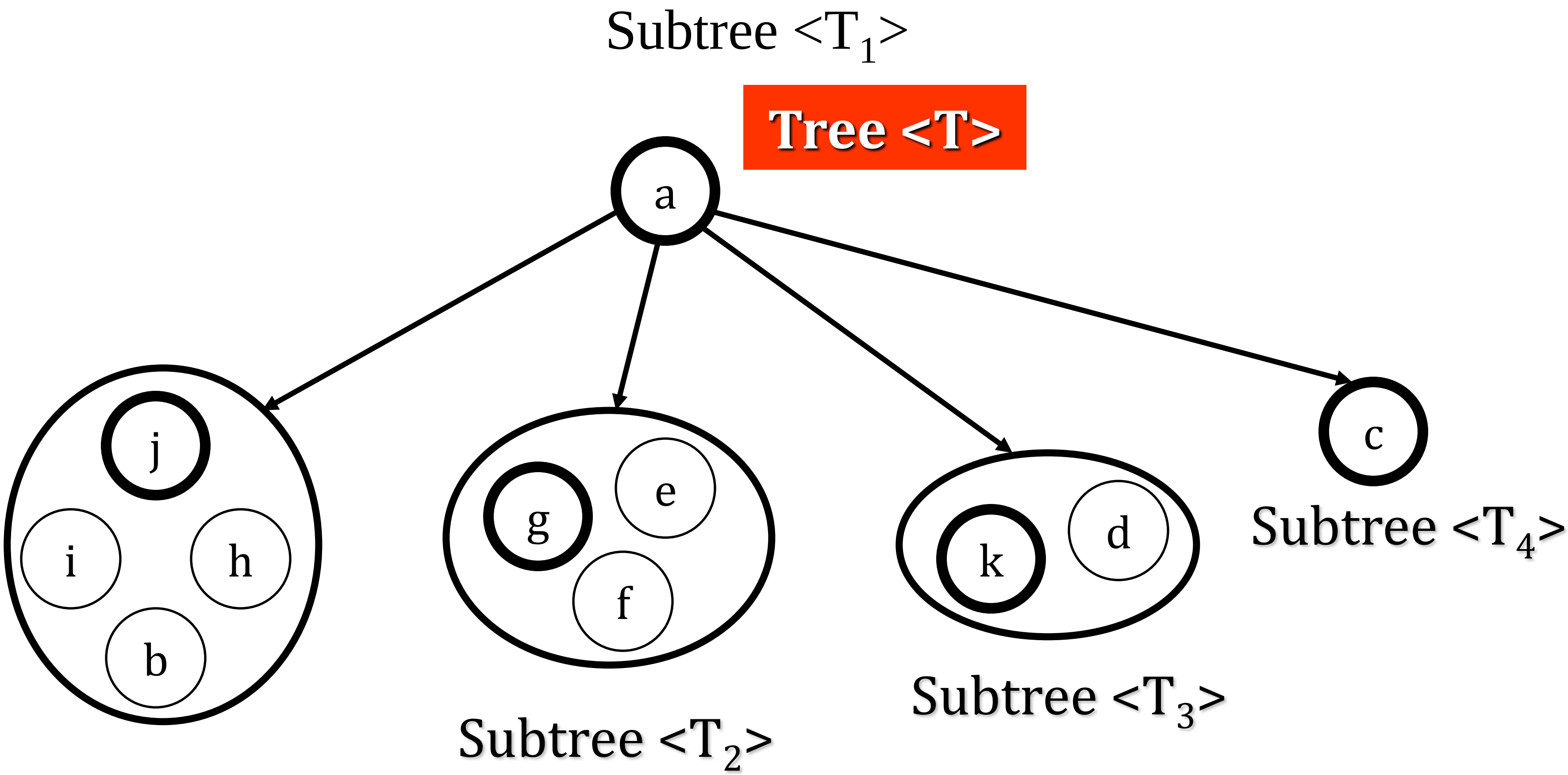
Trees – Definition

1. An *empty structure* is an *empty tree*
2. If $T_1, \ldots, T_k$ are disjointed trees, then the structure T whose root has as its children the roots of $T_1, \ldots, T_k$ is also a tree.
3. Only structures generated by 1 and 2 are trees.

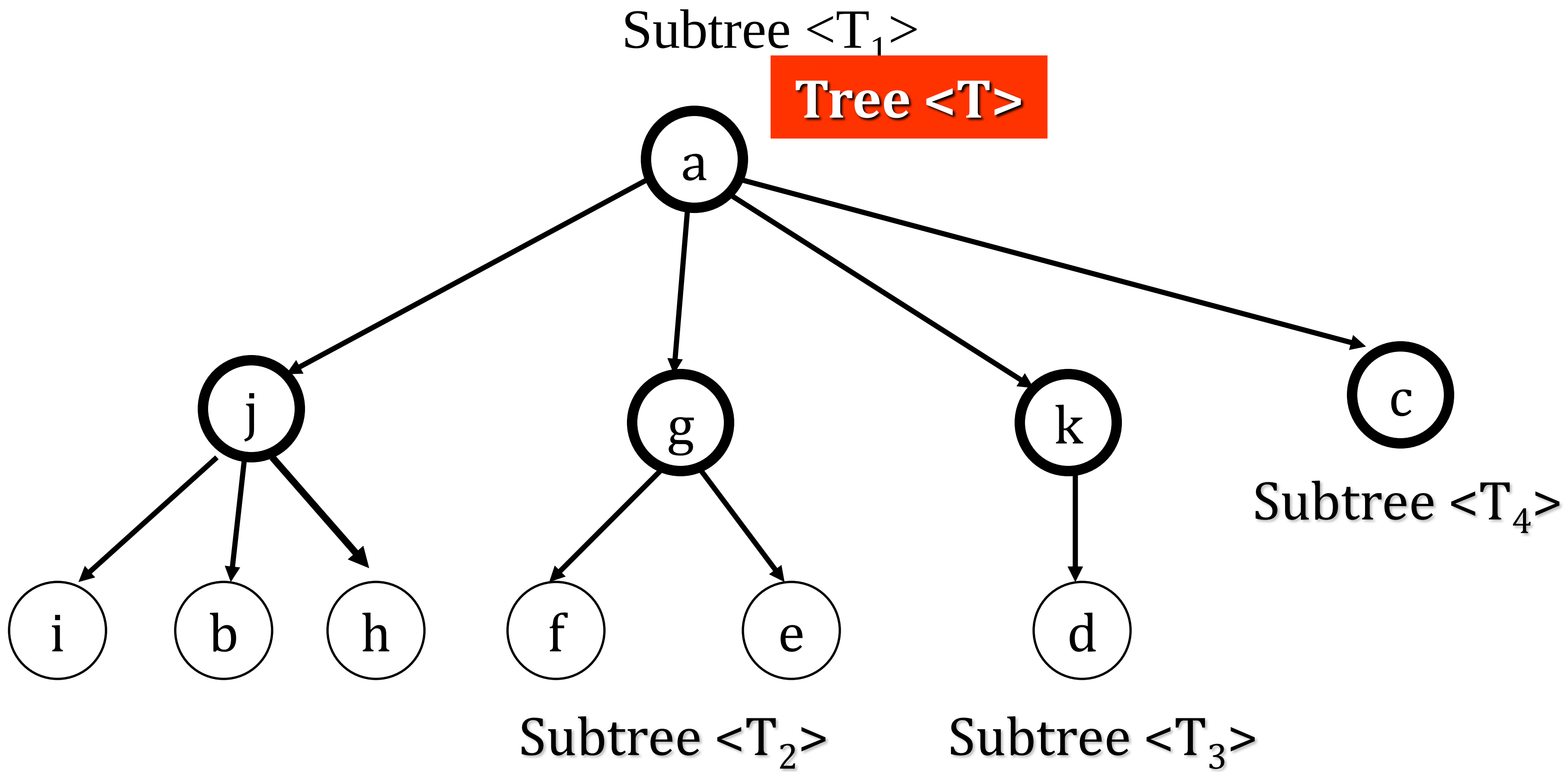
Tree ADT – Example



Tree ADT – Example



Tree ADT – Example



Trees characteristics

- Unlike natural trees, these trees are *upside down*
  - Root at the top
  - Leaves at the bottom
- Consists of *nodes* connected by *edges*.
  - Nodes often represent *entities* (complex objects) such as people, car parts etc.
  - Edges between the nodes represent the way the nodes are *related*.
- **No cycle**

Trees – Terminology

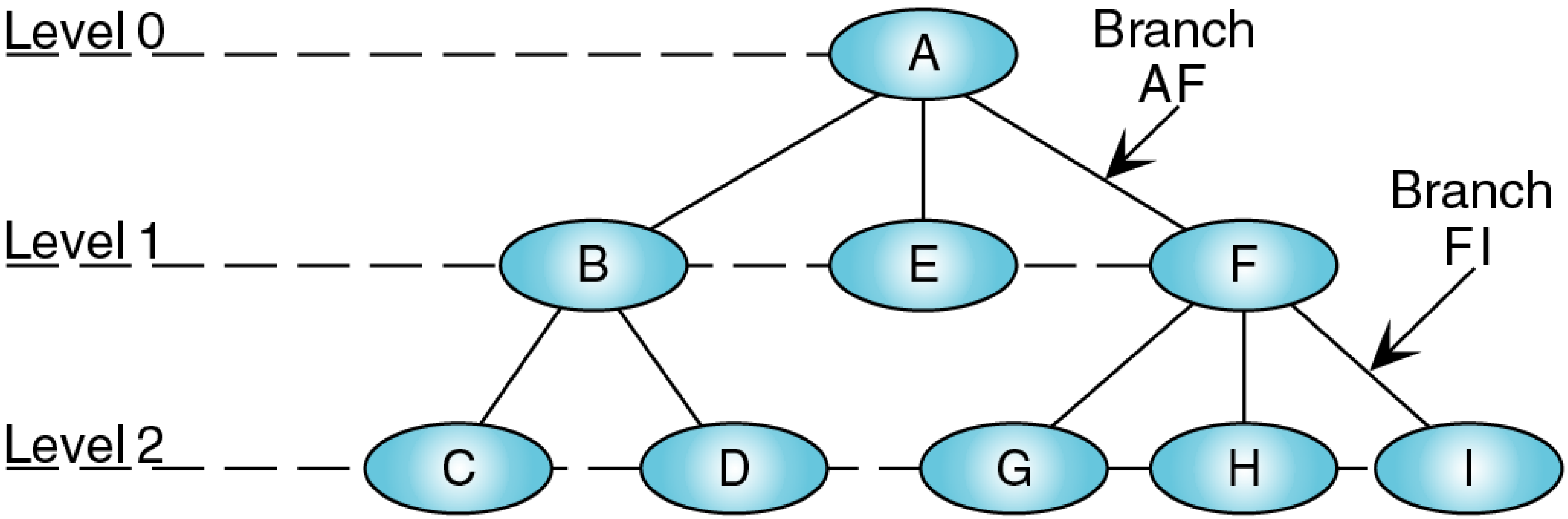
1. Node
2. Edge (Branch)
3. Parent node
4. Child node
5. Sibling nodes
6. Root node
7. Leaf node
8. Internal node
9. Degree of a node
10. Degree of a tree
11. Path
12. Subtree
13. Level/Depth
14. Height

Trees – Terminology

5. Sibling nodes: nodes that have the same parent.
8. Internal nodes: nodes that have both parent and children.
   - Special case: root is also an internal node unless it is a leaf.
9. Degree of a node: the number of its children
10. Degree of a tree: the max degree of all nodes
13. Level (or Depth) of a node *p*:
    - Level (*p*) = 0 if *p* = root
    - Level (*p*) = 1 + Level (Parent (*p*)) if *p*!=root
14. Height of a tree: the number of edges on the longest path from the root to the farthest leaf.

Trees – Terminology

- A tree with height = 2



Root: A
Parents: A, B, F
Children: B, E, F, C, D, G, H, I

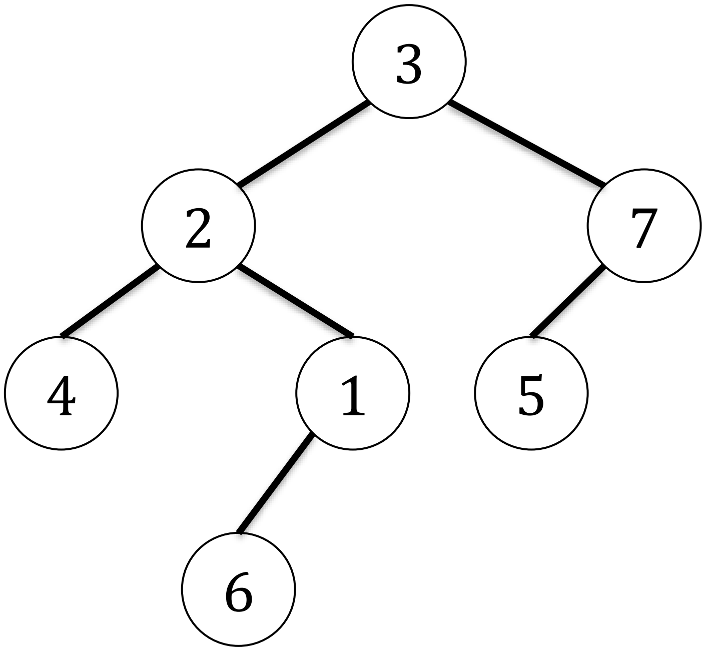
Siblings: {B, E, F}, {C, D}, {G, H, I}
Leaves: C, D, G, H, I
Internal nodes: A, B, F

Binary trees

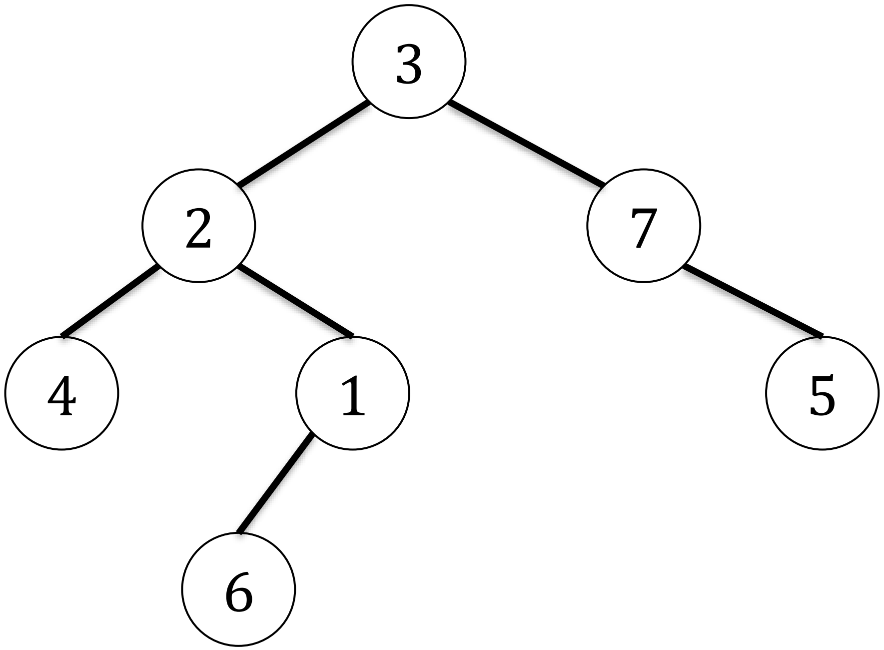
- **Definition:** A binary tree *T* is a structure defined on a finite set of nodes that either
  - contains no nodes, or
  - is composed of 3 disjoint sets of nodes:
    - a root node
    - a binary tree called its *left subtree*
    - a binary tree called its *right subtree*
- What about this definition:
  - T is a binary tree if Degree(*T*) = 2

→ not enough since in a binary tree, if a node has just one child, the position of the child (*left* child/*right* child) matters.

Binary trees – Example



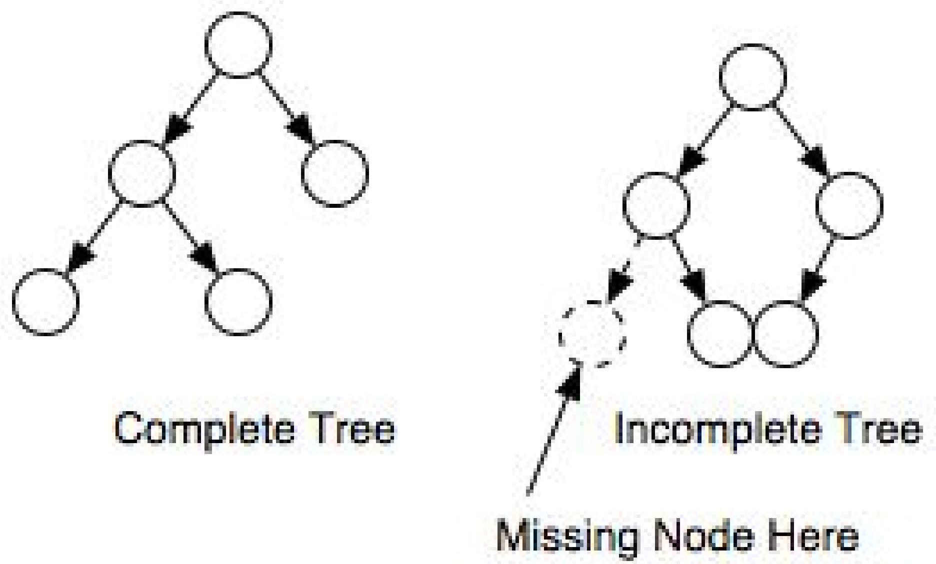
(a)



(b)

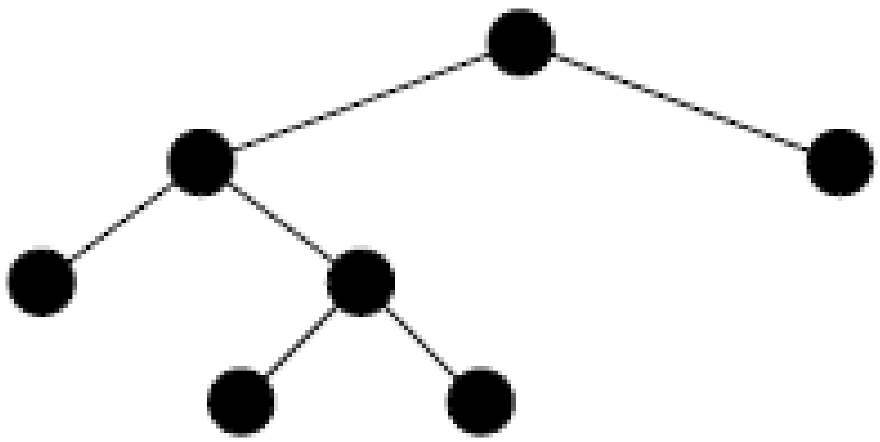
Types of binary trees

- Complete binary tree:
  - From level 0 to level h-1: the tree is completely full (maximum number of nodes)
  - The nodes at the last level are filled from left to right.



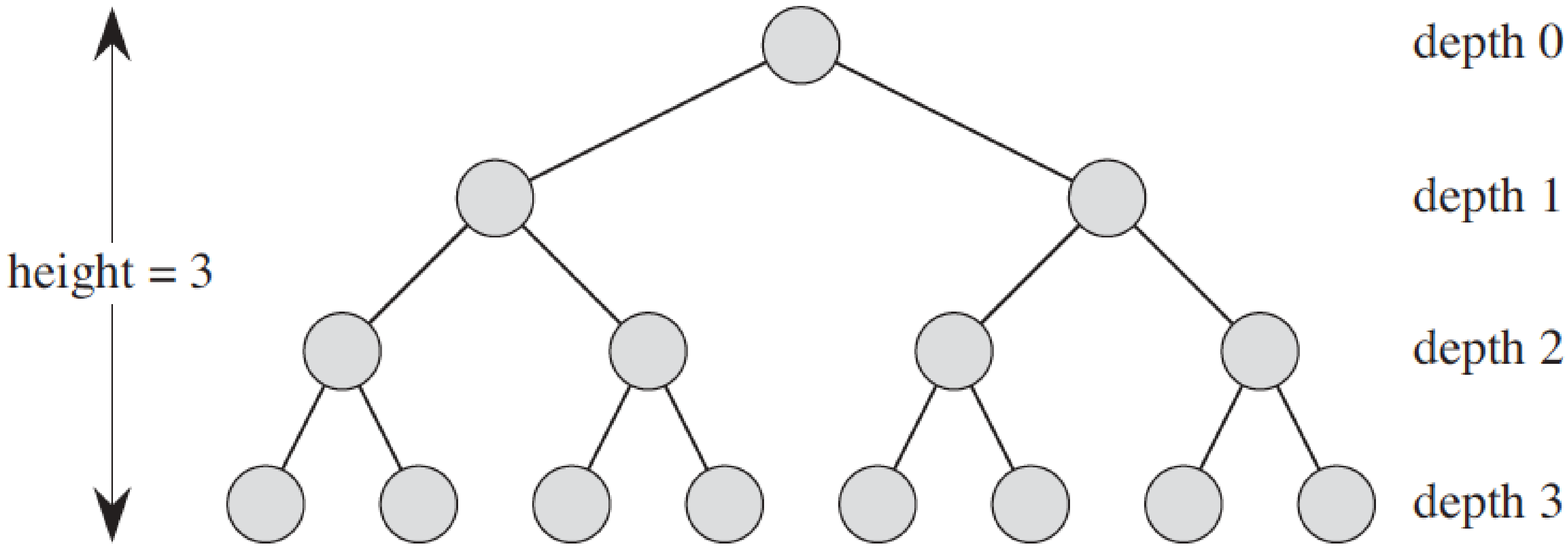
Types of binary trees

- Full binary tree:
  - Each node is either a leaf or has degree exactly 2.



Types of binary trees

- Perfect binary tree:
  - A full binary tree in which all leaf nodes are at the same level.

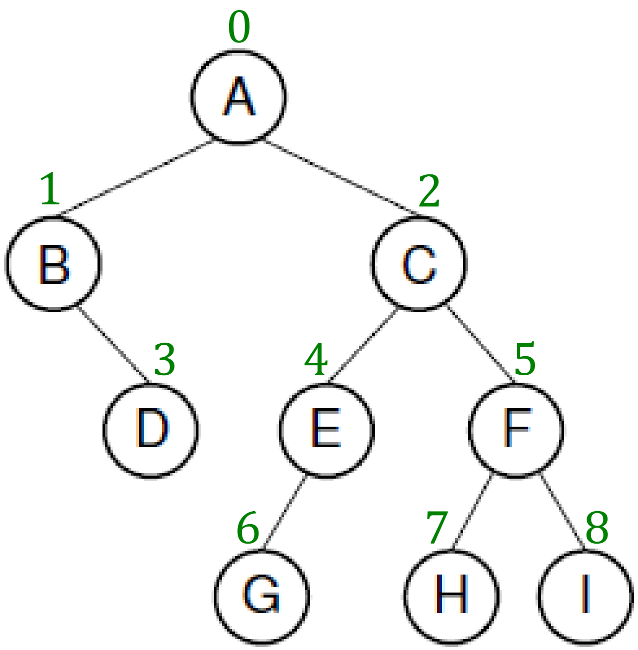


Maximum number of nodes in binary trees

Height	Nodes at one level	Nodes at all levels
0	$2^0 = 1$	$1 = 2^1 - 1$
1	$2^1 = 2$	$3 = 2^2 - 1$
2	$2^2 = 4$	$7 = 2^3 - 1$
3	$2^3 = 8$	$15 = 2^4 - 1$
10	$2^{10} = 1{,}024$	$2{,}047 = 2^{11} - 1$
13	$2^{13} = 8{,}192$	$16{,}383 = 2^{14} - 1$
h	2^h	$n = 2^{h+1} - 1$

Implement a binary tree

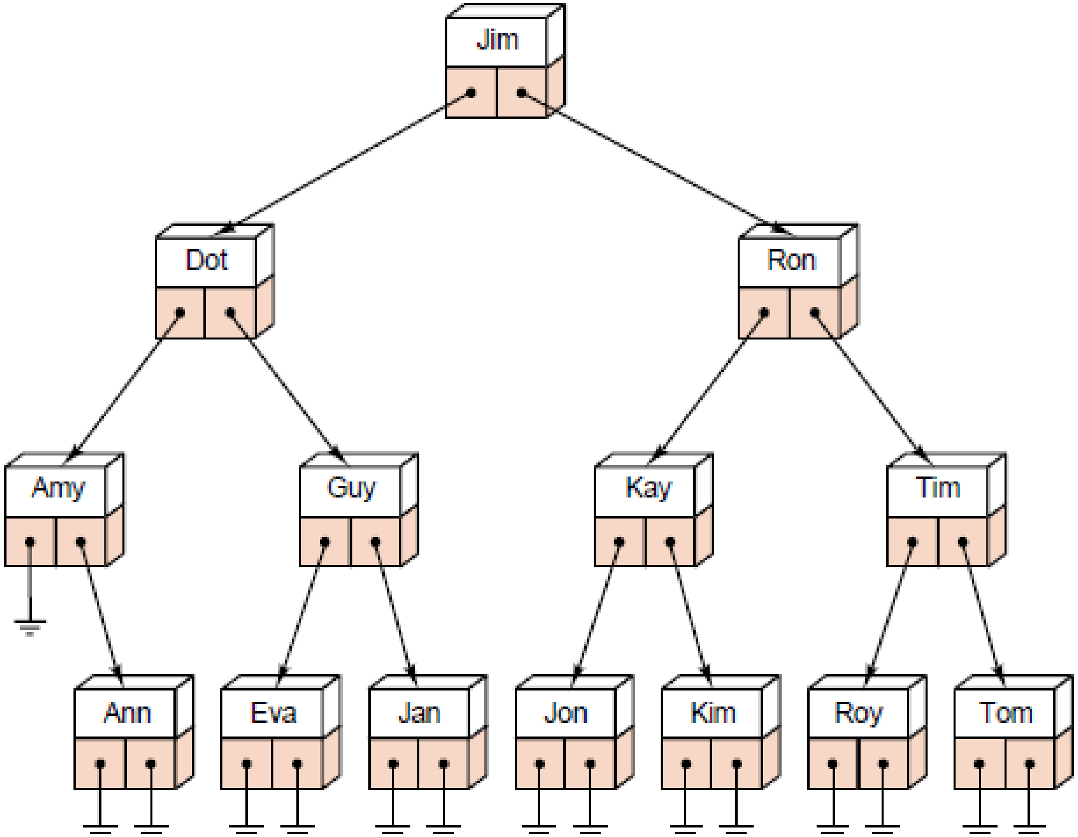
- Using an array:



index	Node	Left	Right
0	A	1	2
1	B	-1	3
2	C	4	5
3	D	-1	-1
4	E	6	-1
5	F	7	8
6	G	-1	-1
7	H	-1	-1
8	I	-1	-1

Implement a binary tree

- Using pointers:



Tree traversal

- Tree traversal (or tree walk): allow us to print out all the keys in a tree.
- 3 strategies:
  - In-order traversal (LNR – Left Node Right)
  - Pre-order traversal (NLR – Node Left Right)
  - Post-order traversal (LRN – Left Right Node)

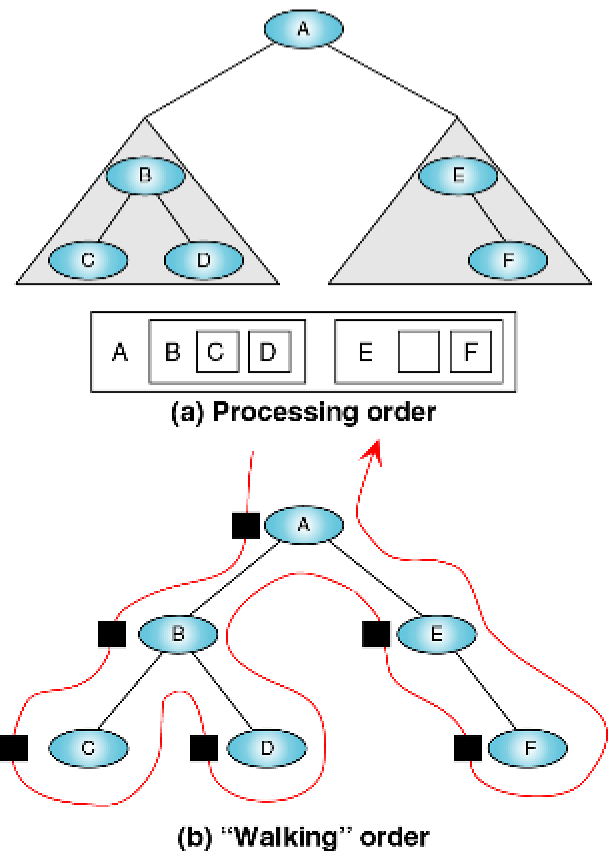
Tree traversal

```
INORDER-TREE-WALK(x)
1.  if x ≠ NIL
2.      then INORDER-TREE-WALK(x.left)
3.           print x.key
4.           INORDER-TREE-WALK(x.right)
```

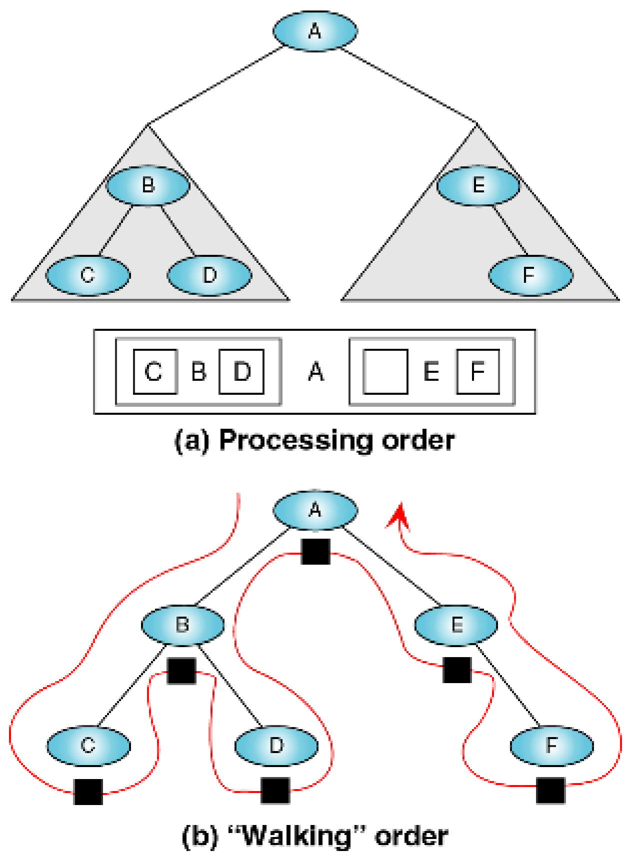
```
PREORDER-TREE-WALK(x)
1.  if x ≠ NIL
2.      then print x.key
3.           PREORDER-TREE-WALK (x.left)
4.           PREORDER-TREE-WALK (x.right)
```

```
POSTORDER-TREE-WALK(x)
1.  if x ≠ NIL
2.      then POSTORDER-TREE-WALK (x.left)
3.           POSTORDER-TREE-WALK (x.right)
4.           print x.key
```

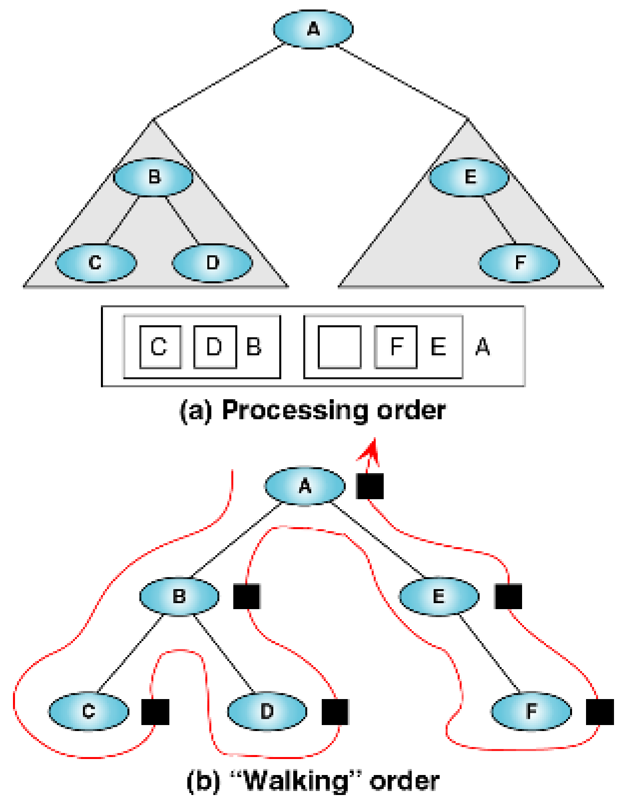
Tree traversal



Pre-order tree walk
NLR



In-order tree walk
LNR



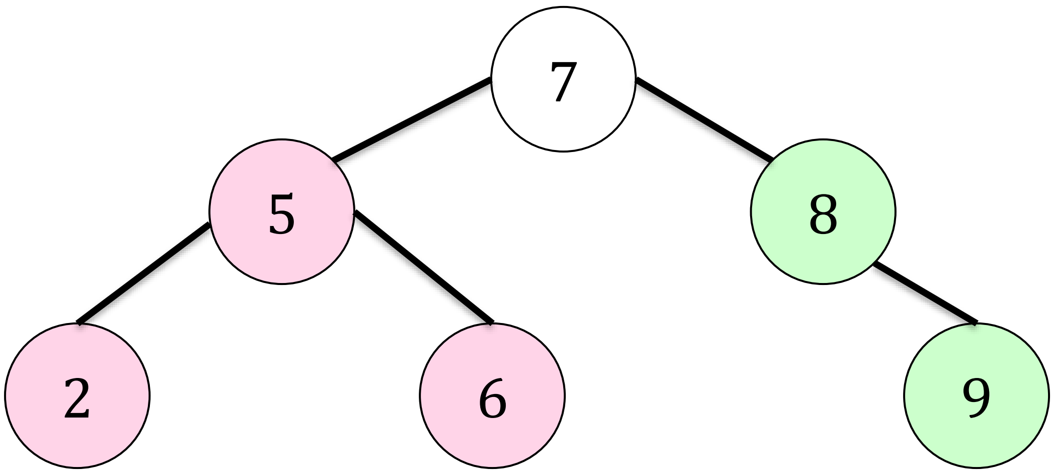
Post-order tree walk
LRN

BINARY SEARCH TREES

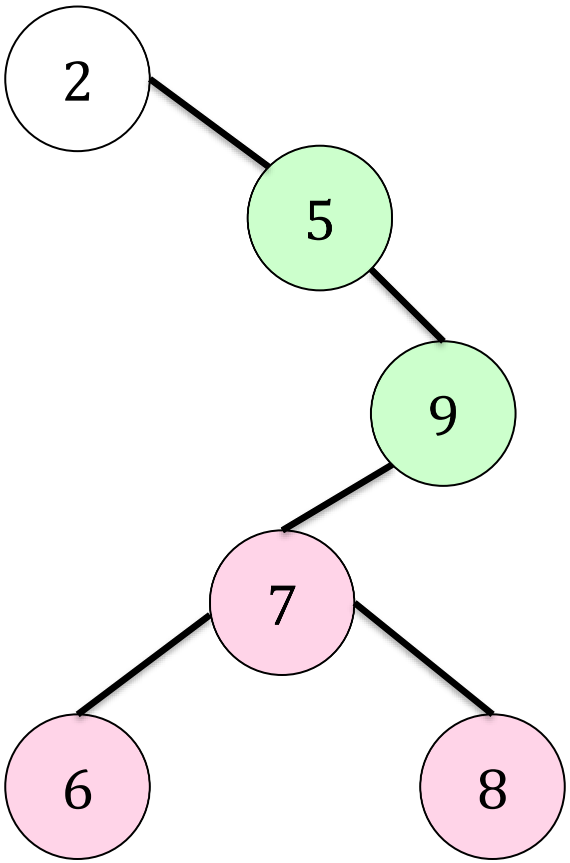
Binary search trees

- **Definition:** A binary search tree (BST) is a binary tree which storing keys in a way that satisfies the ***binary-search-tree property***:
  - Let *x* be a node in a BST
  - If *y* is a node in the left subtree of *x*, then *x.key*≥ *y.key*
  - If *y* is a node in the right subtree of *x*, then *x.key*< *y.key*
- Why using a BST?
  - Fast for basic operations: insert, delete, search

Binary search tree – Example



(a)



(b)

Querying a binary search tree

- Operations:
  - Searching
  - Minimum and maximum
  - Successor and predecessor
  - Insertion and deletion
- **Theorem.** We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in $O(h)$ time on a BST of height h.

Searching a BST

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
        return x
if k < x.key
        return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

$O(h)$

Recursive version

Searching a BST

TREE-SEARCH(x, k)

1. **while** $x \neq NIL$ and $k \neq x.key$
2. **if** $k < x.key$
3. $x = x.left$
4. **else**
5. $x = x.right$
6. **return** x

$O(h)$

Iterative version

Minimum and maximum

TREE-MINIMUM(x)

1. **while** $x.left \neq$ NIL
2. $x = x.left$
3. **return** x

$O(h)$

TREE-MAXIMUM(x)

1. **while** $x.right \neq$ NIL
2. $x = x.right$
3. **return** x

$O(h)$

Successor and predecessor

- If all keys are distinct, the **successor** of a node *x* is:
  - the node with the smallest key greater than *x.key*.
  - NIL if *x* has the largest key in the tree.

- If all keys are distinct, the **predecessor** of a node *x* is:
  - the node with the largest key smaller than *x.key*.
  - NIL if *x* has the smallest key in the tree.

Successor and predecessor

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

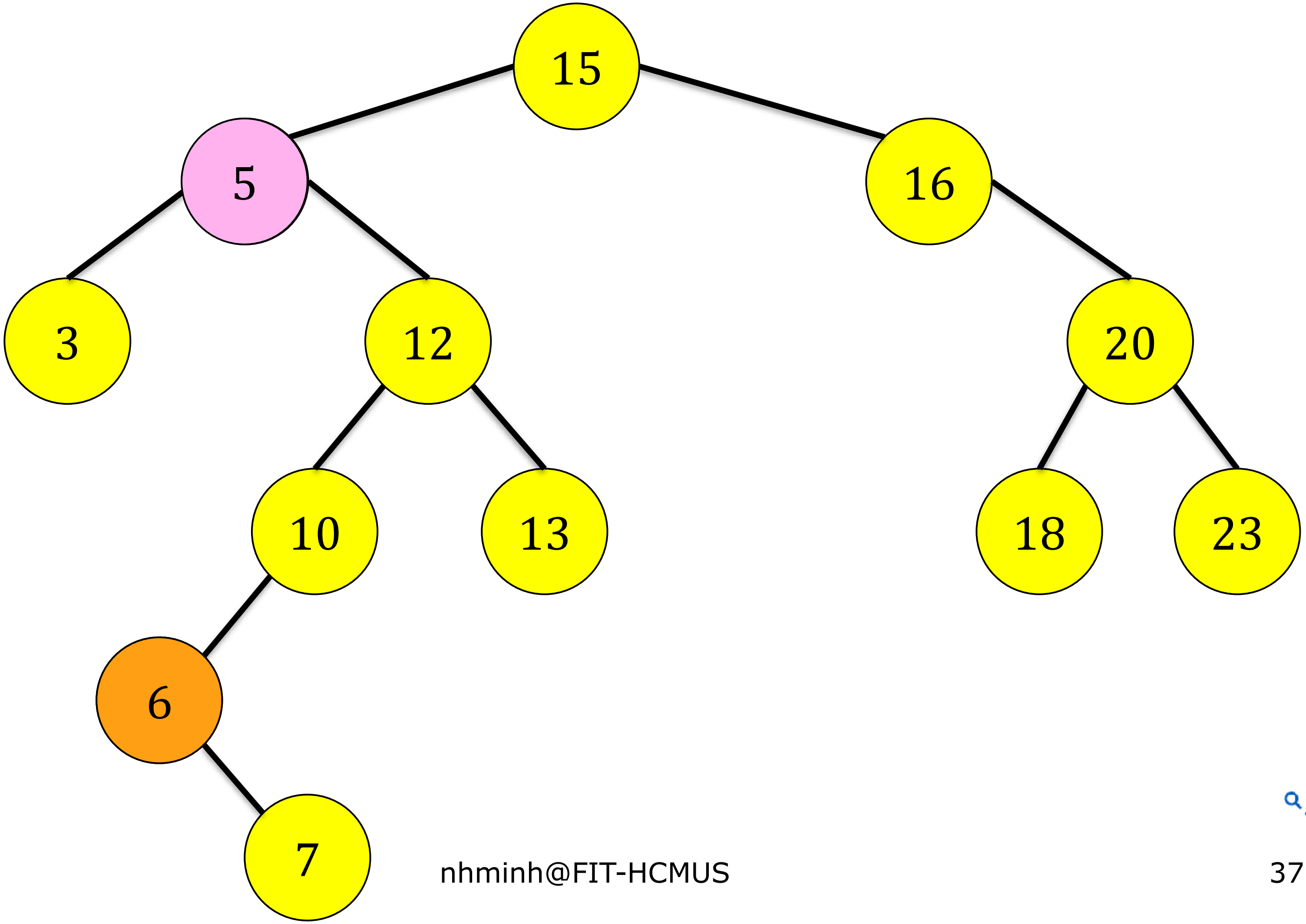
$O(h)$

TREE-PREDECESSOR(x)

…

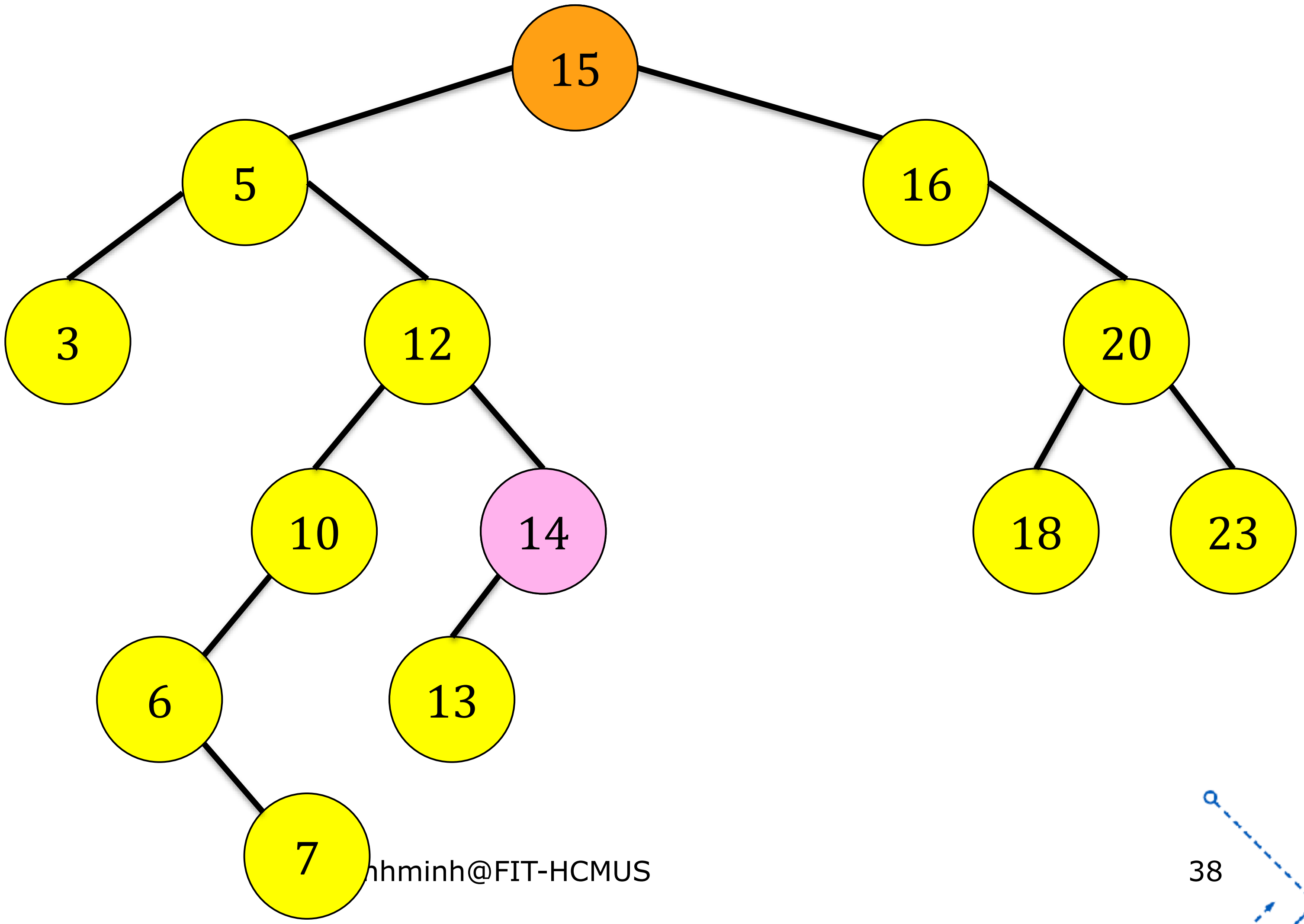
Successor – Example

- Successor of 5 is: 6



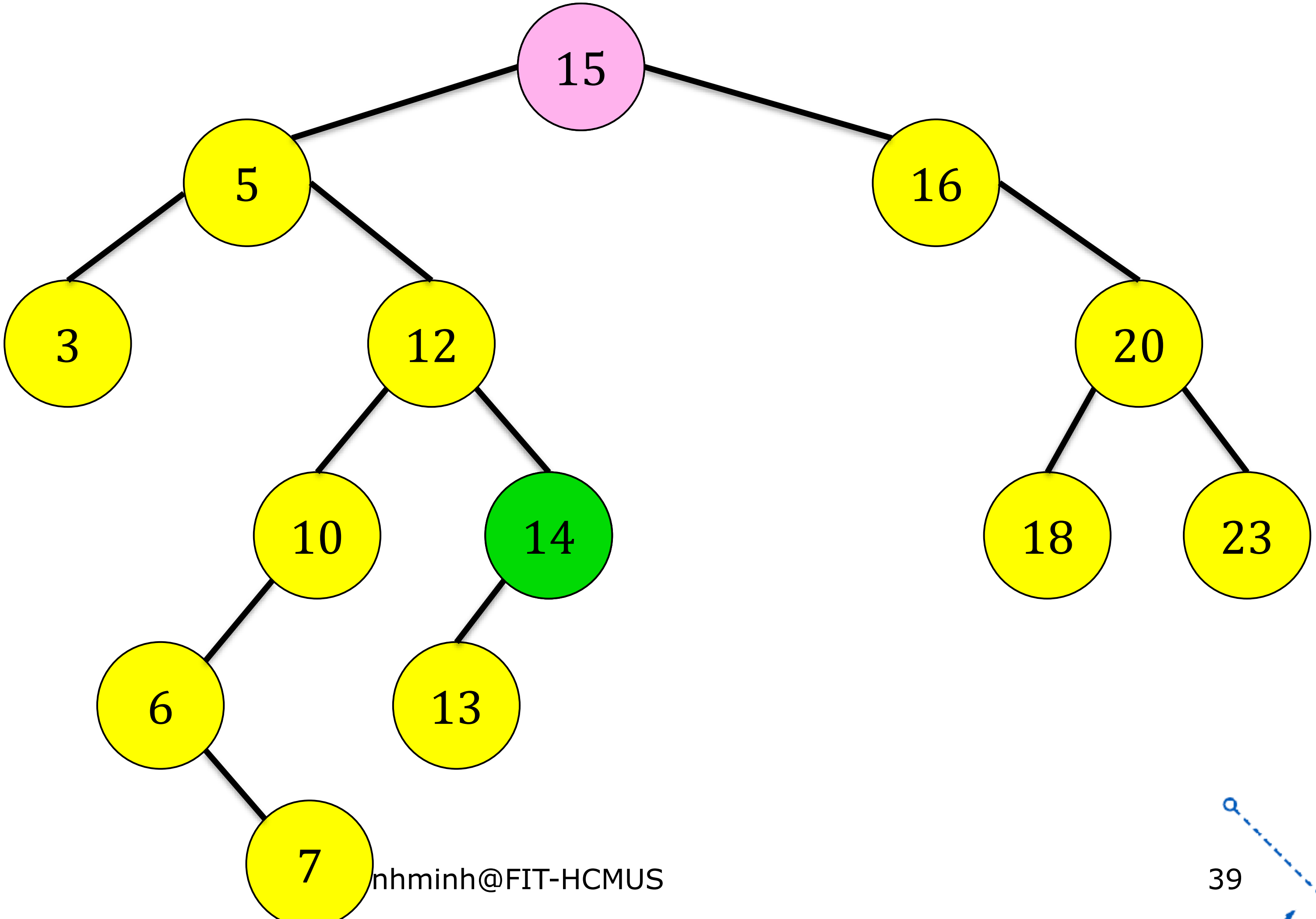
Successor – Example

- Successor of 14 is: 15



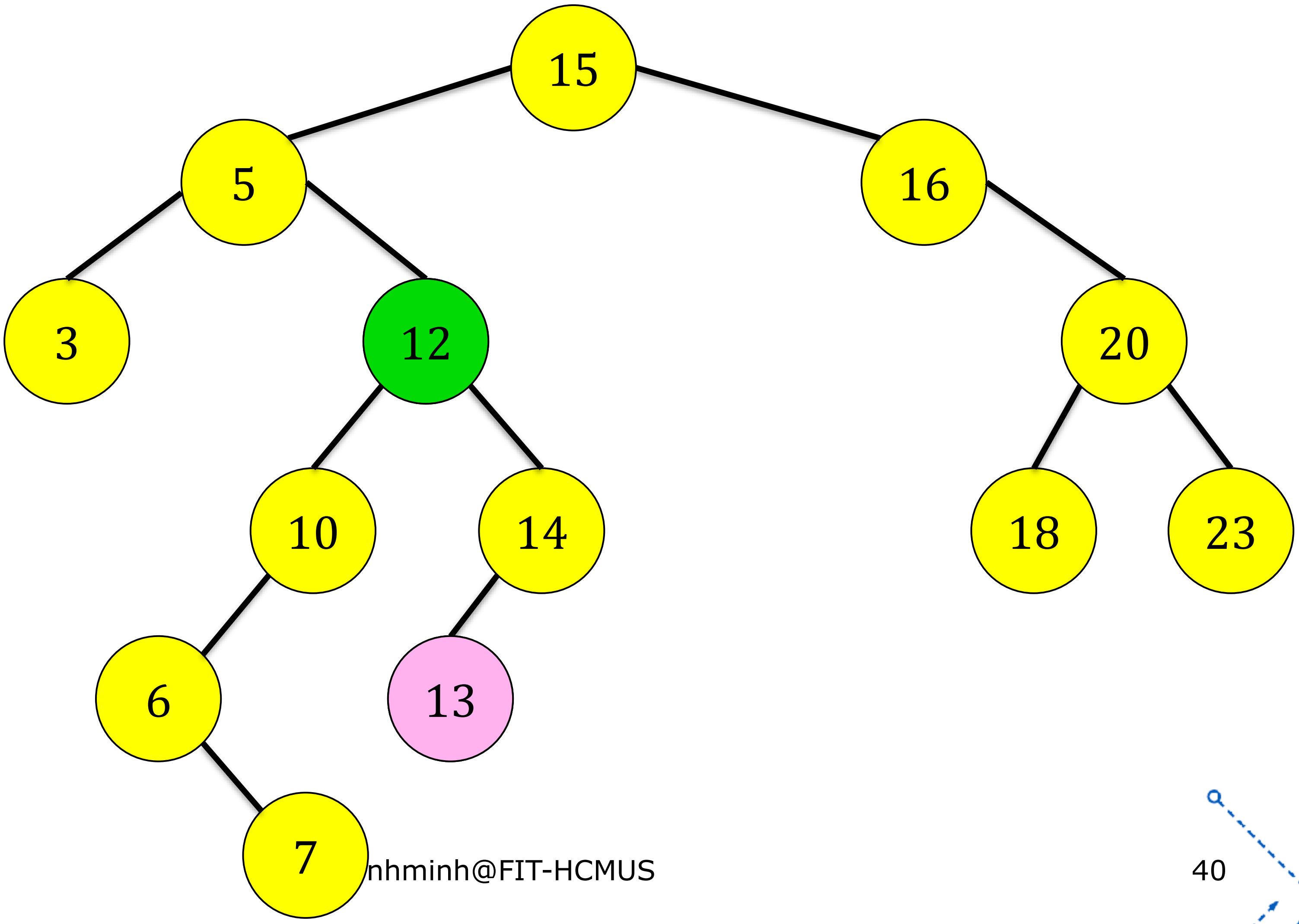
Predecessor – Example

- Predecessor of 15 is 14



Predecessor – Example

- Predecessor of 13 is 12



Insertion and deletion

- The operation of *insertion* and *deletion* cause the BST to change.
  - The data structure must be modified to reflect this change.
  - The BST property must be continued to hold.
- *Insertion*: straight-forward.
- *Deletion*: more intricate.

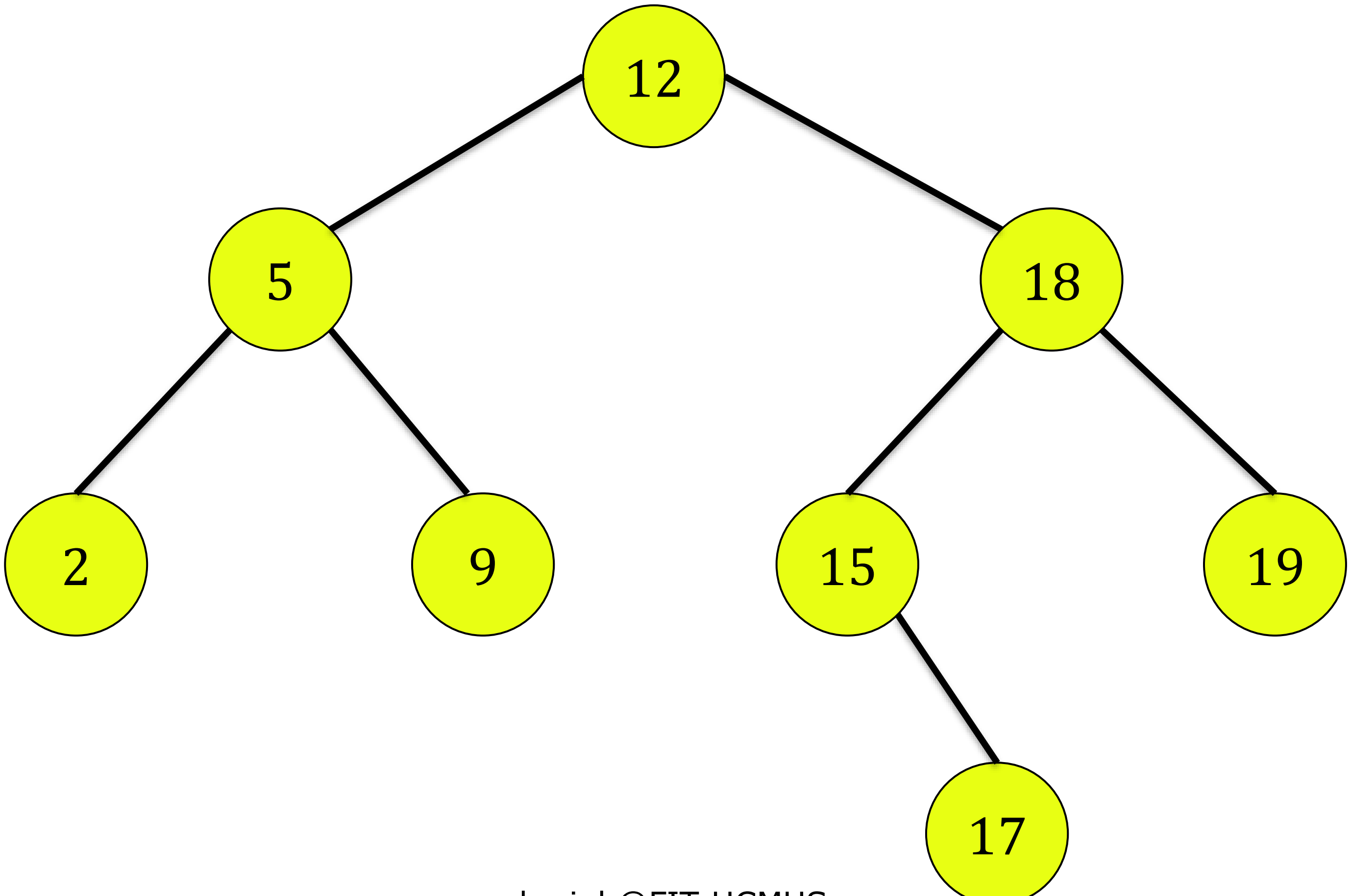
Insertion

TREE-INSERT(*T*, *z*)

```
y = NIL
x = T.root
while x ≠ NIL
        y = x
        if z.key < x.key
             x = x.left
        else x = x.right
z.p = y
if y == NIL
        T.root = z   // tree T was empty
elseif z.key < y.key
        y.left = z
else y.right = z
```

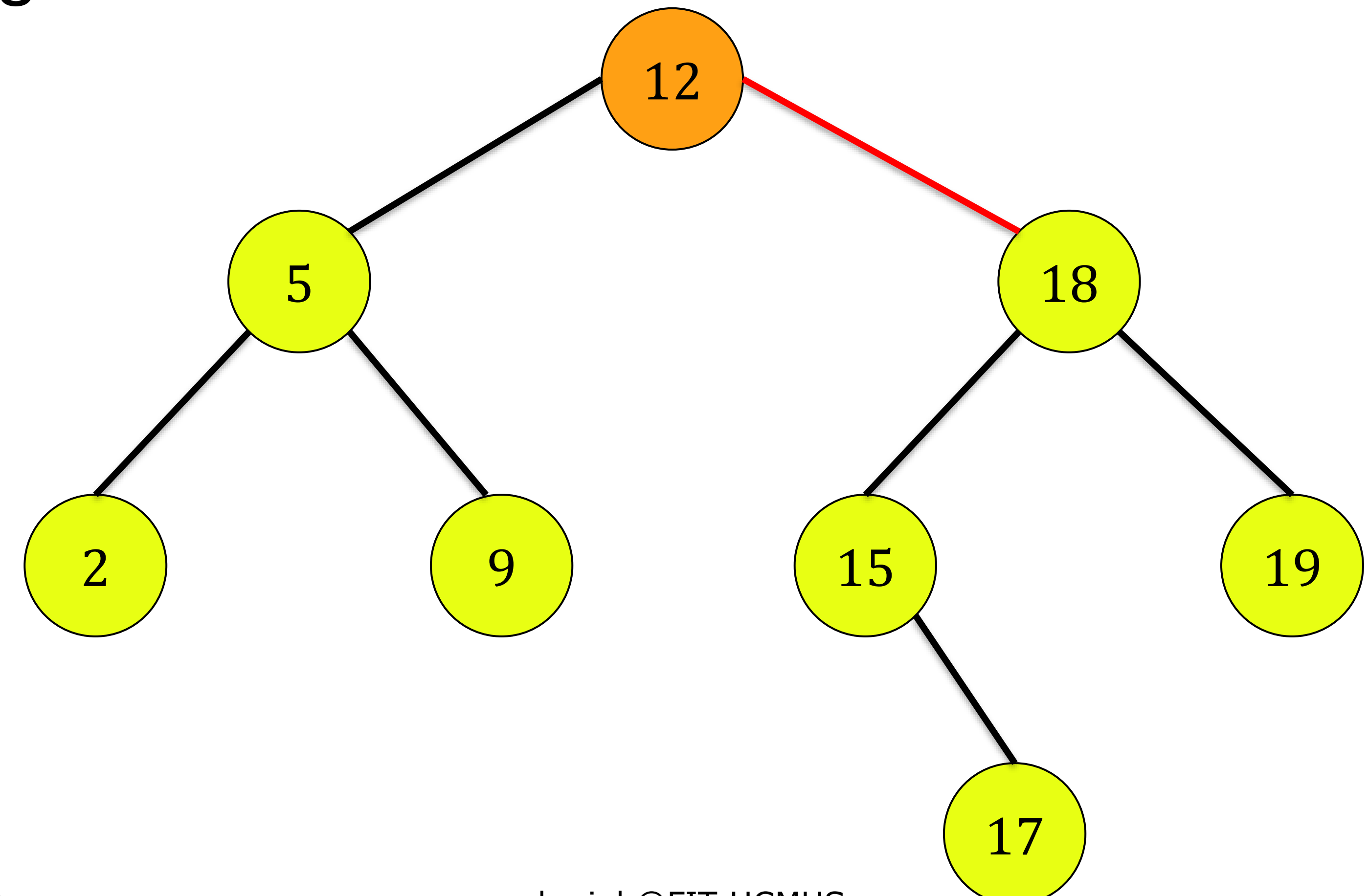
Insertion – Example

- Insert node 13 to the BST



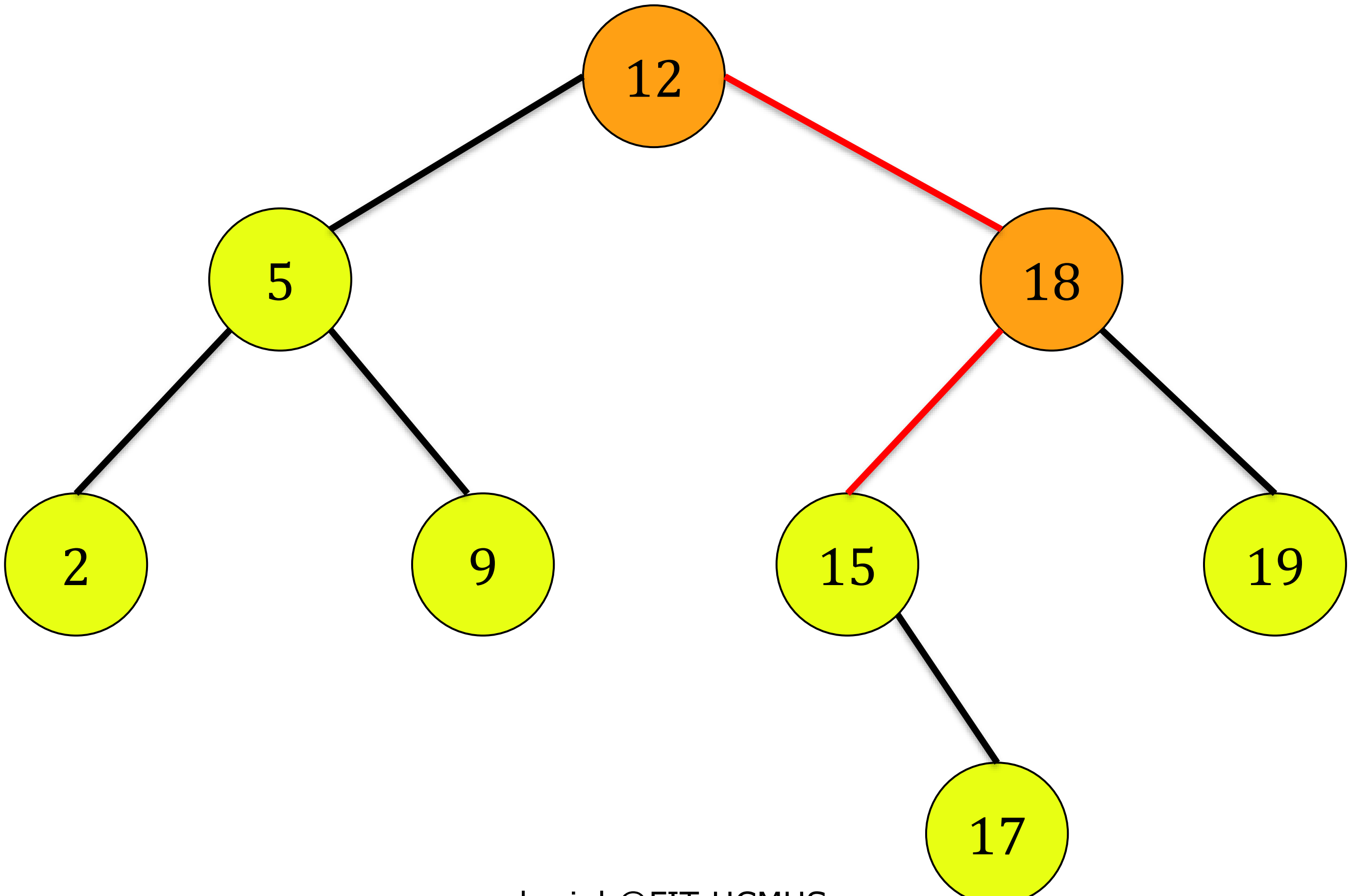
Insertion – Example

- Insert node 13 to the BST: 13>12 → go to the right



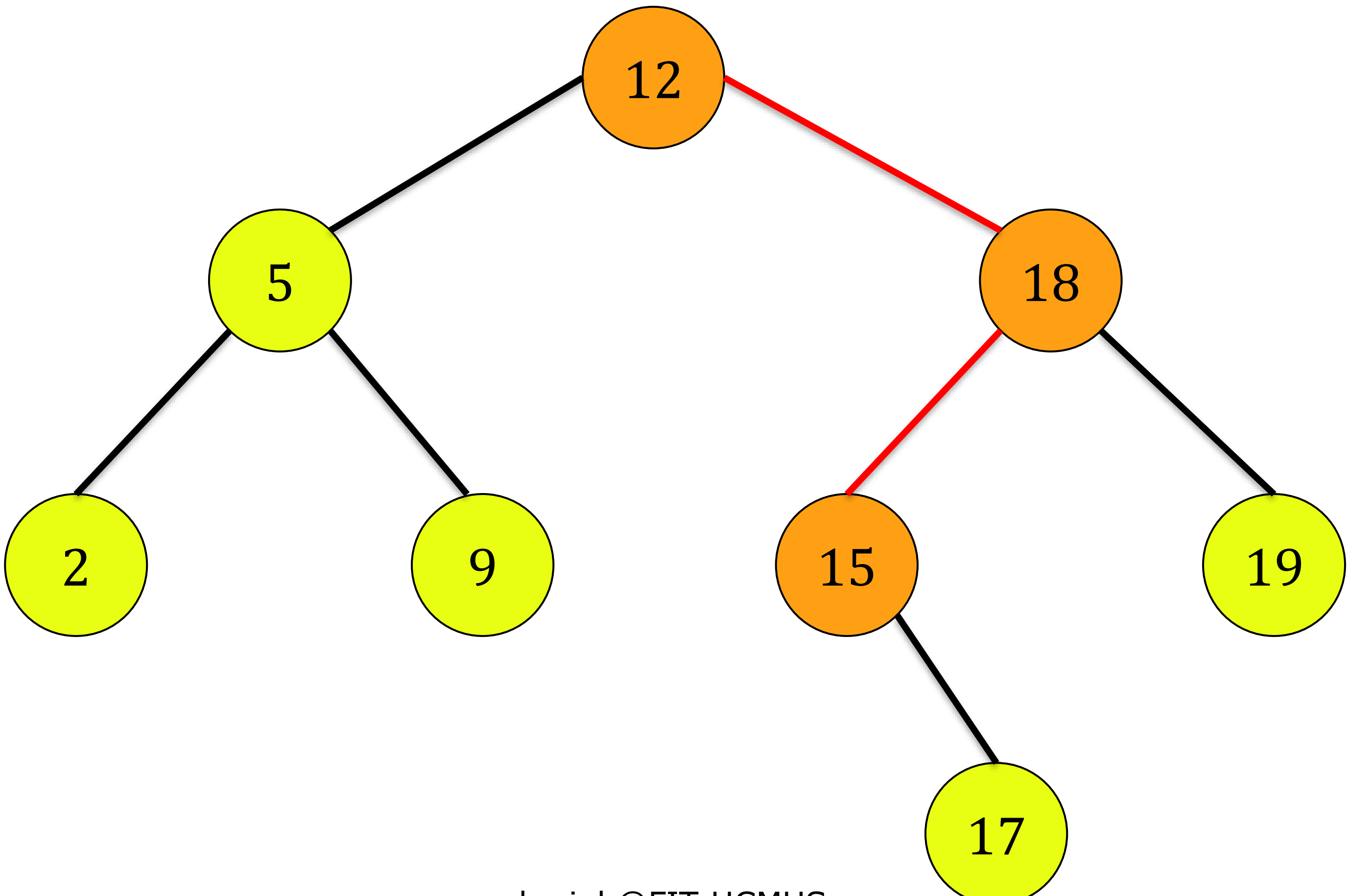
Insertion – Example

- Insert node 13 to the BST: 13<18 → go to the left



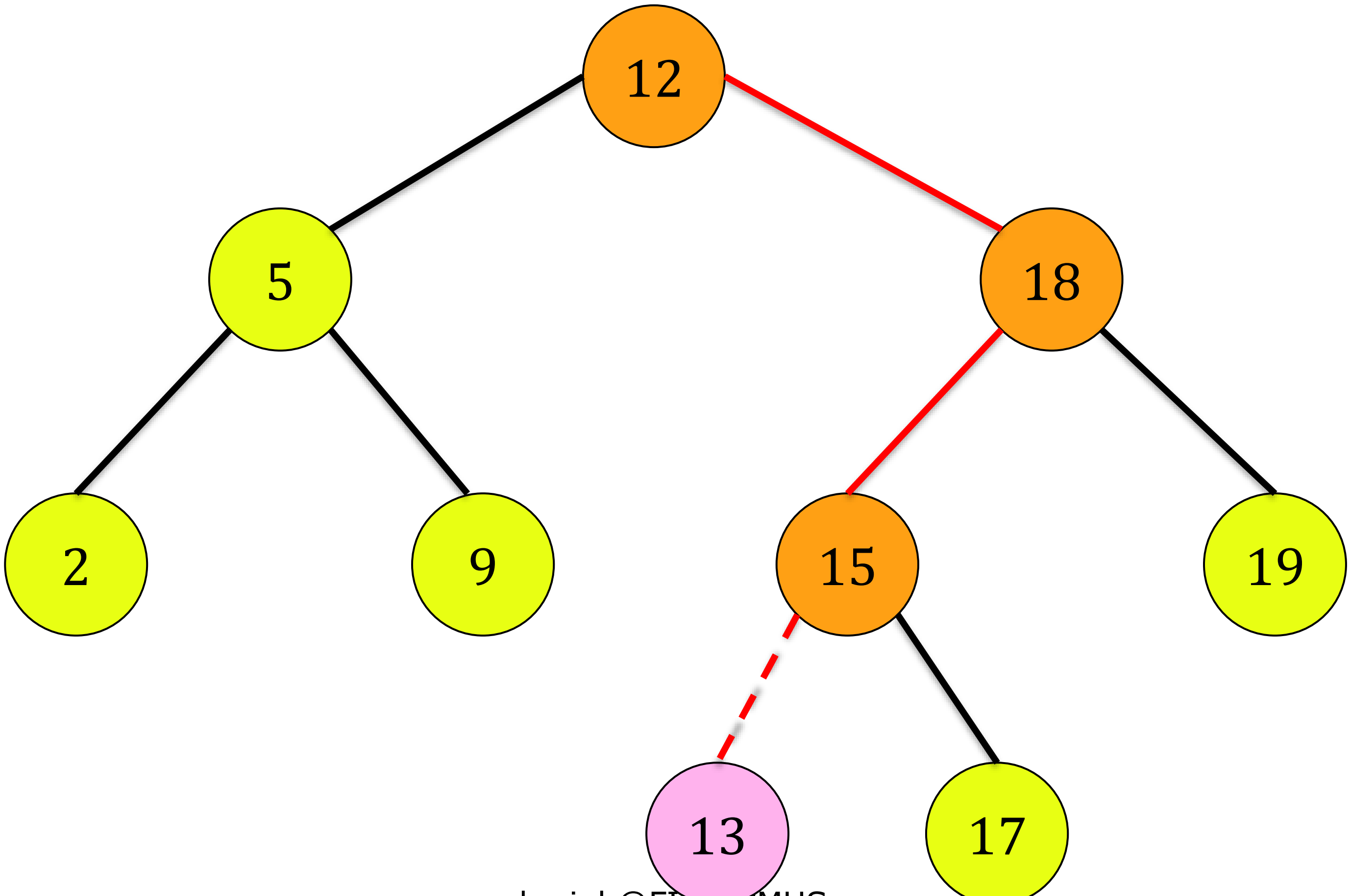
Insertion – Example

- Insert node 13 to the BST: 13<15 → go to the left



Insertion – Example

- Insert node 13 to the BST: left of 15 is NIL → insert 13 as the left child of 15



Deletion

- Deleting a node *z*: 3 cases:
  1. *z* has no child (leaf node)

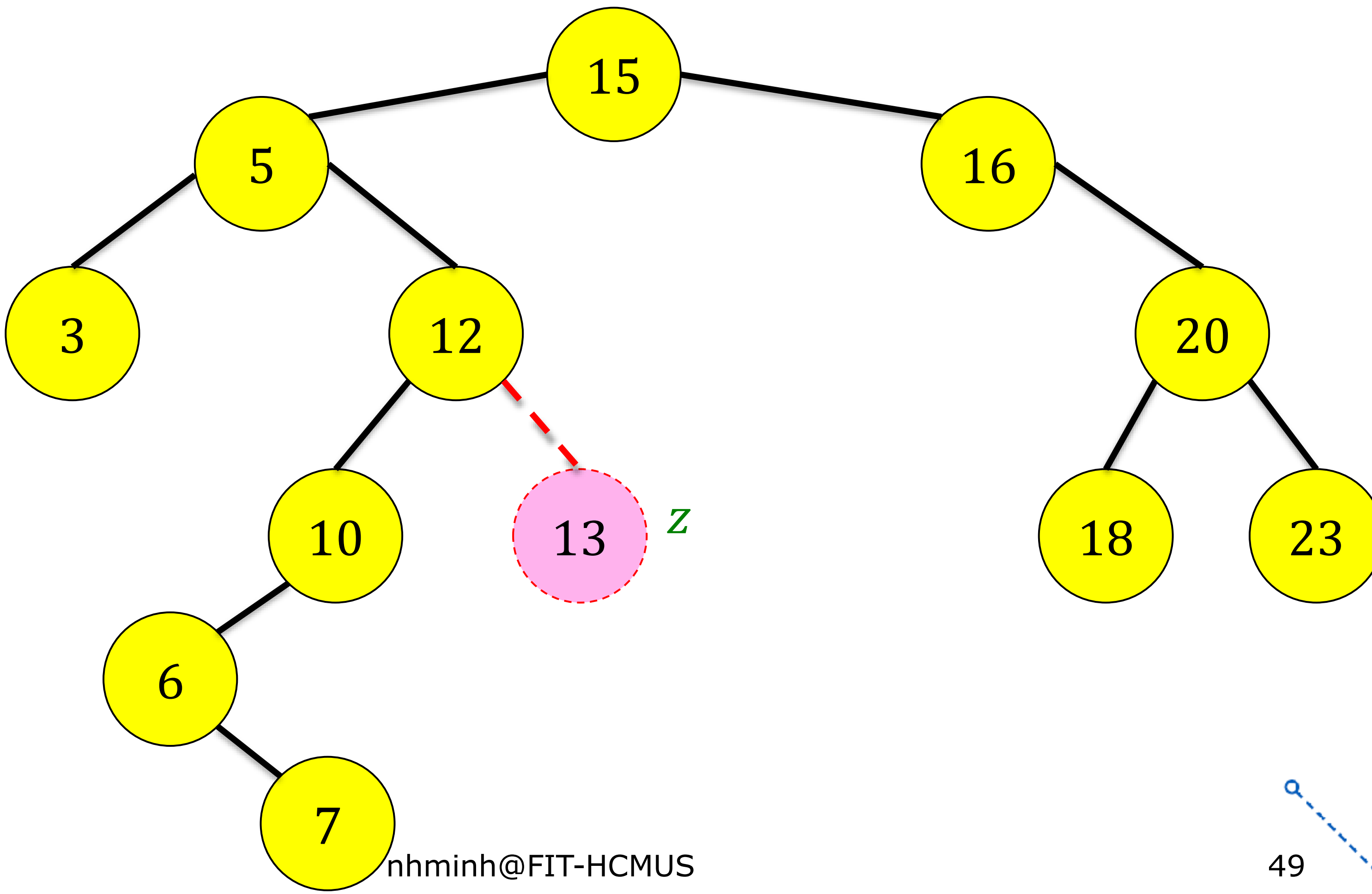
     → simply remove it
  2. *z* has one child

     → replace *z* by its child
  3. *z* has two children

     → find its successor (or predecessor): y – must be in z's right (or left) subtree and has no left (right) child. Replace z.key by y.key, then delete y.

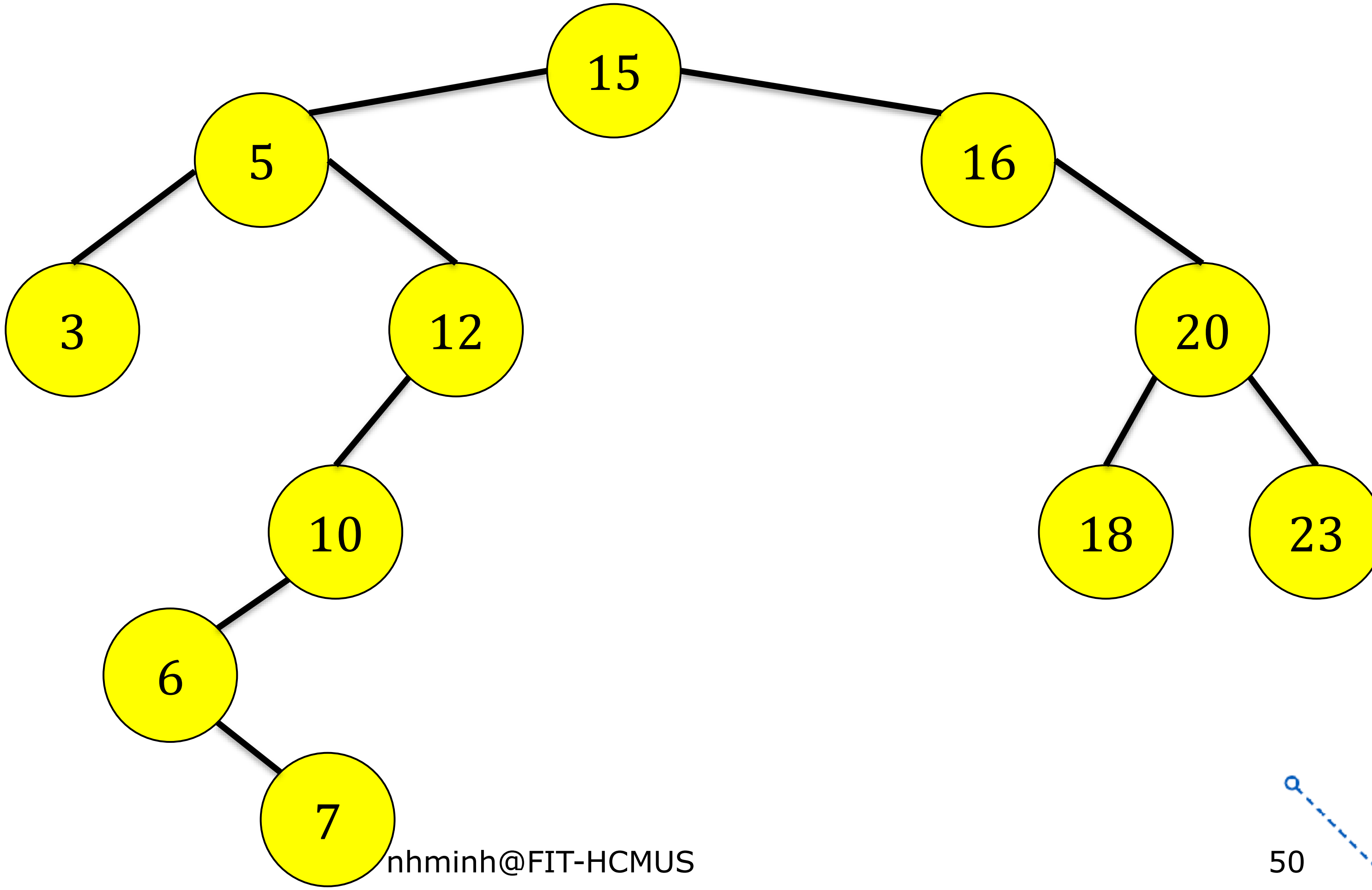
Deletion

- *z* has no child (leaf node): simply remove it



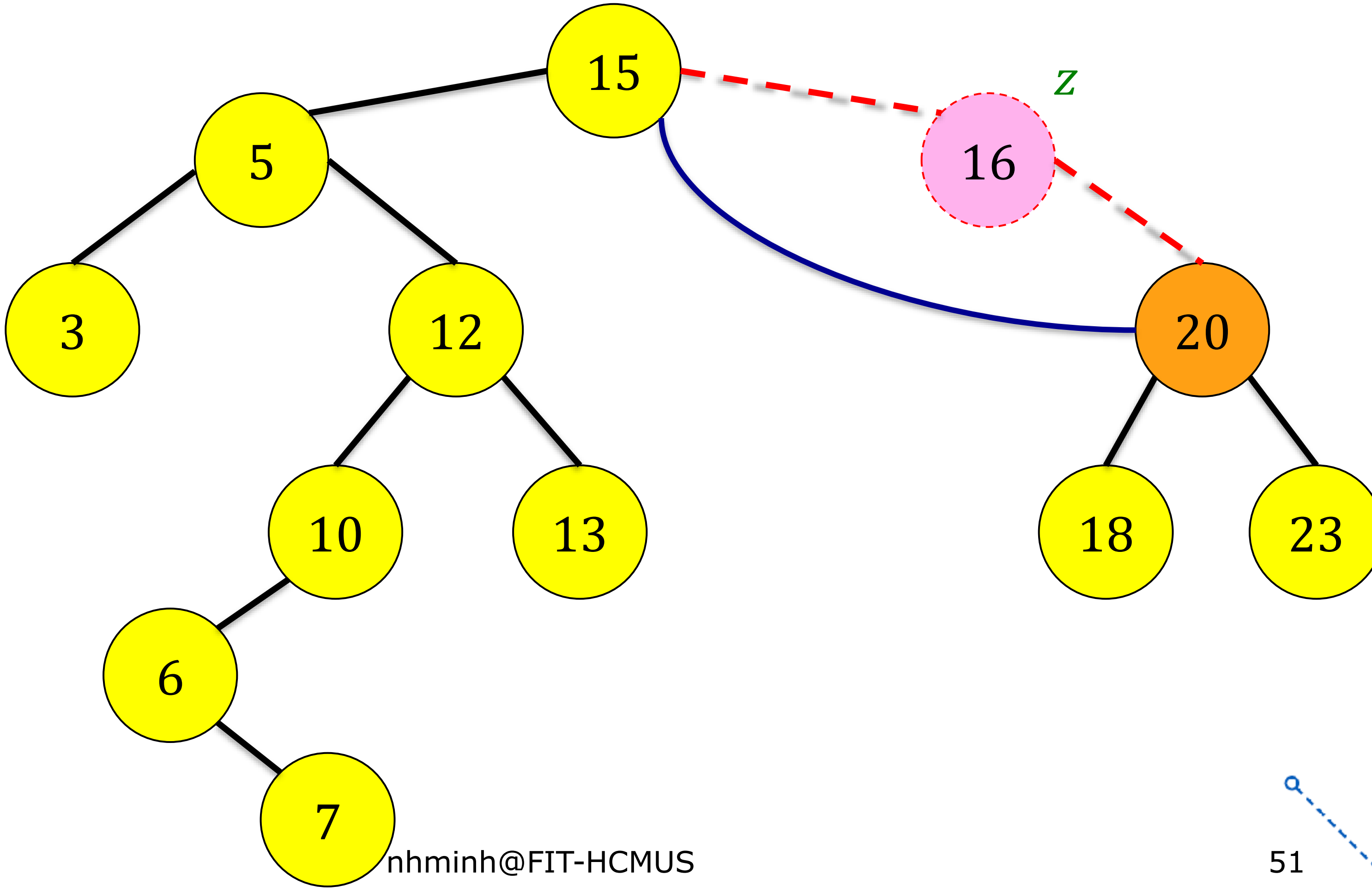
Deletion – case 1

- *z* has no child (leaf node): simply remove it



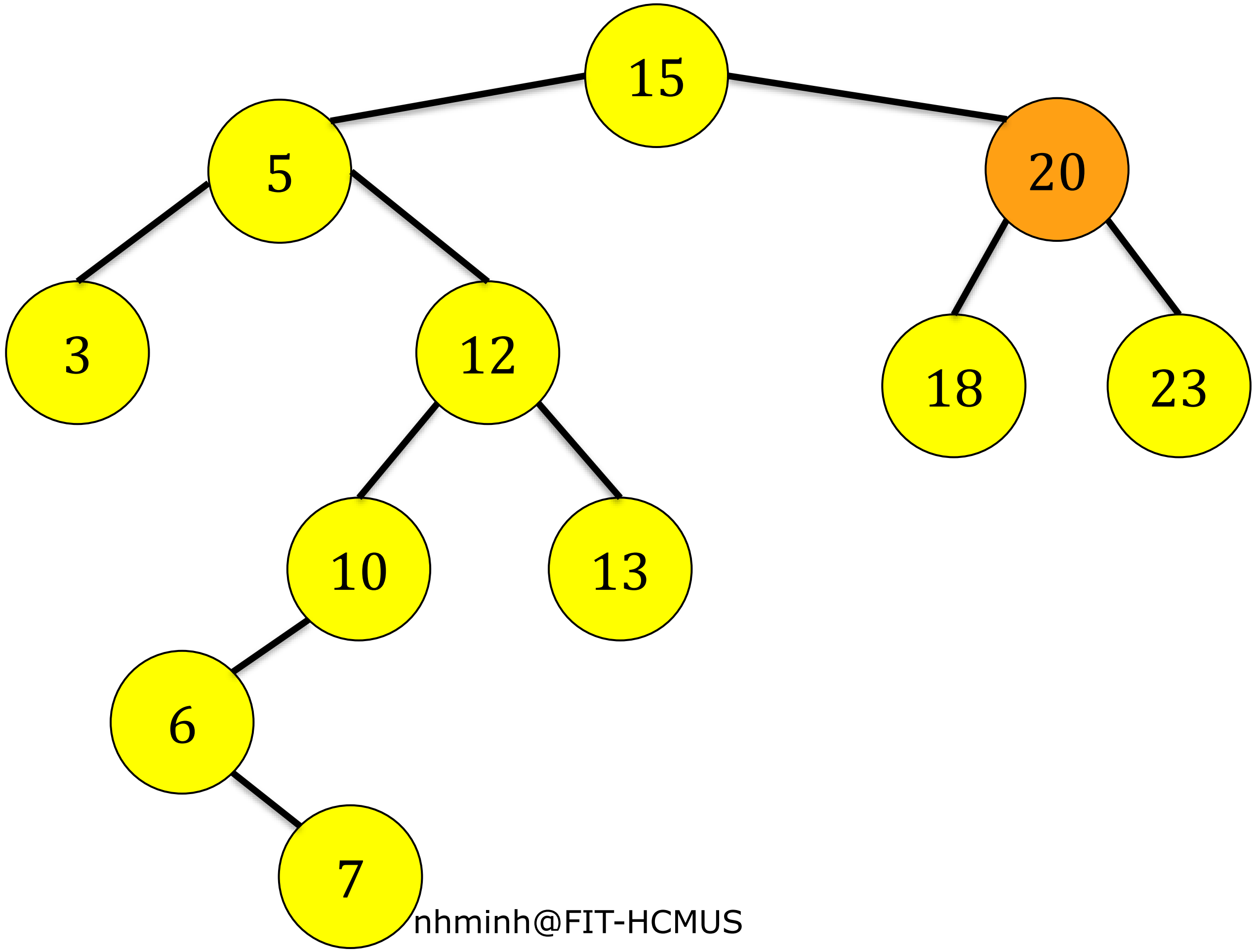
Deletion – case 2

- z has 1 child: replace z by its subtree



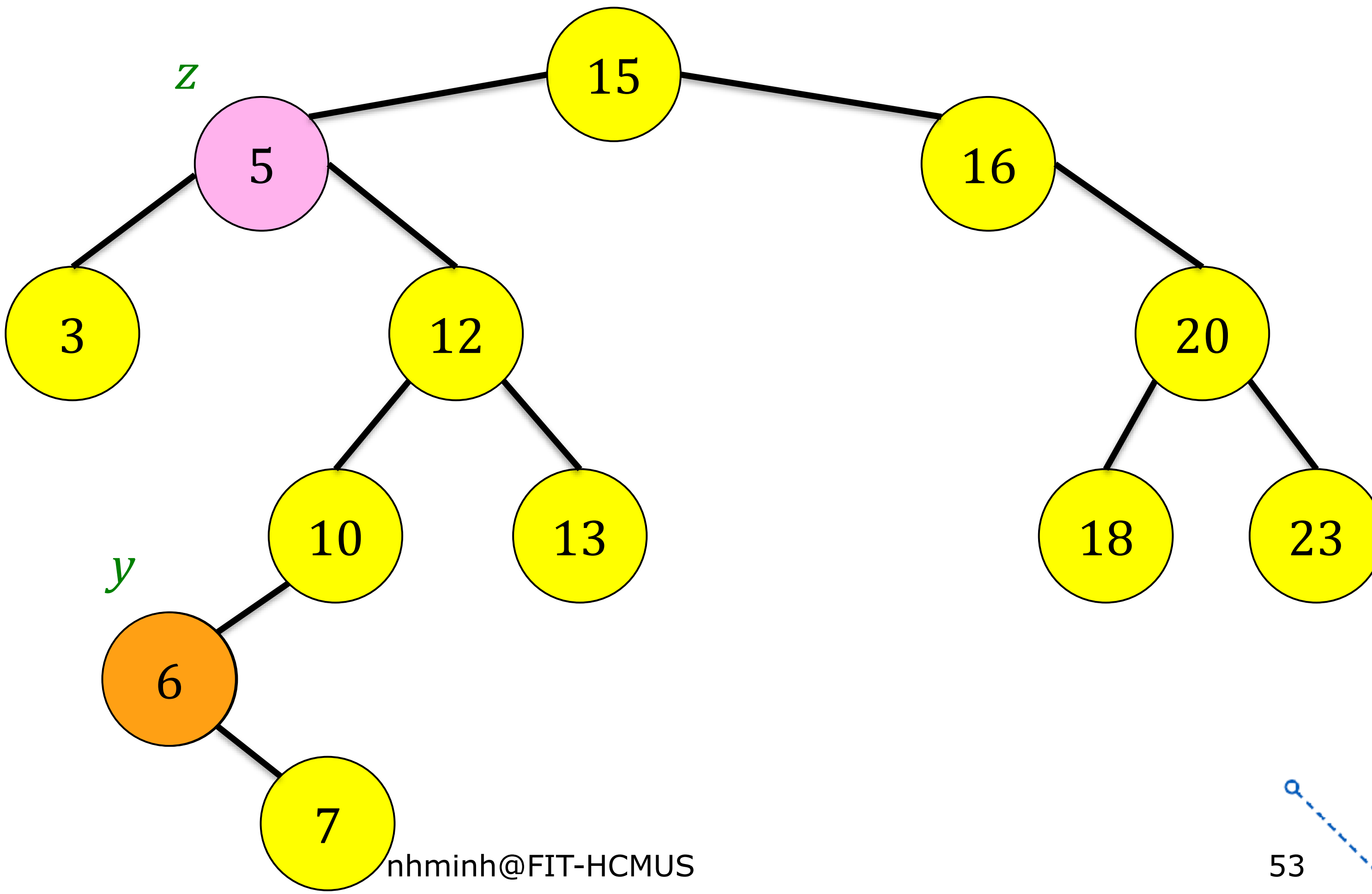
Deletion – case 2

- *z* has 1 child: replace *z* by its subtree



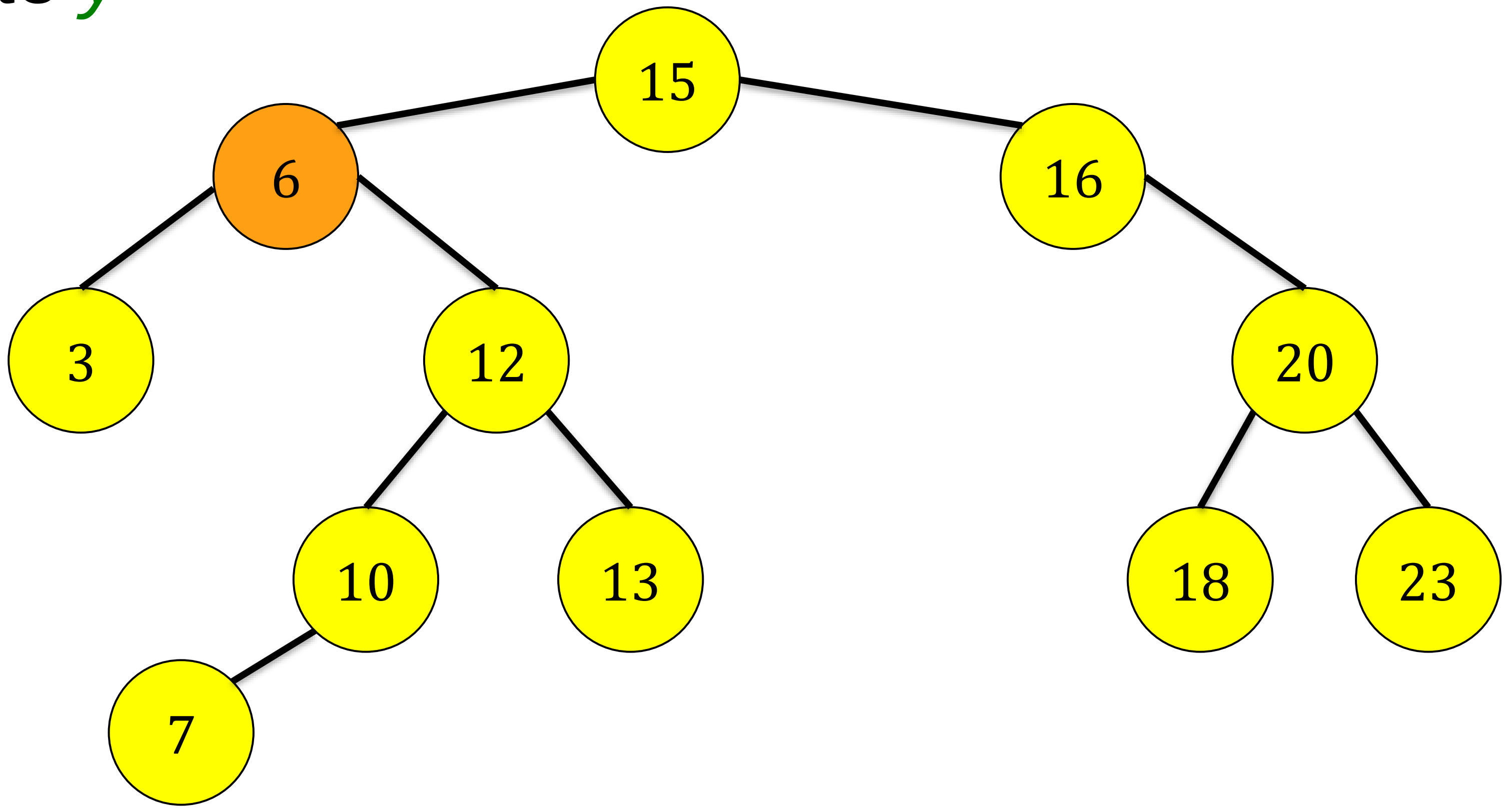
Deletion – case 3

- *z* has 2 children: find *z*'s successor *y*



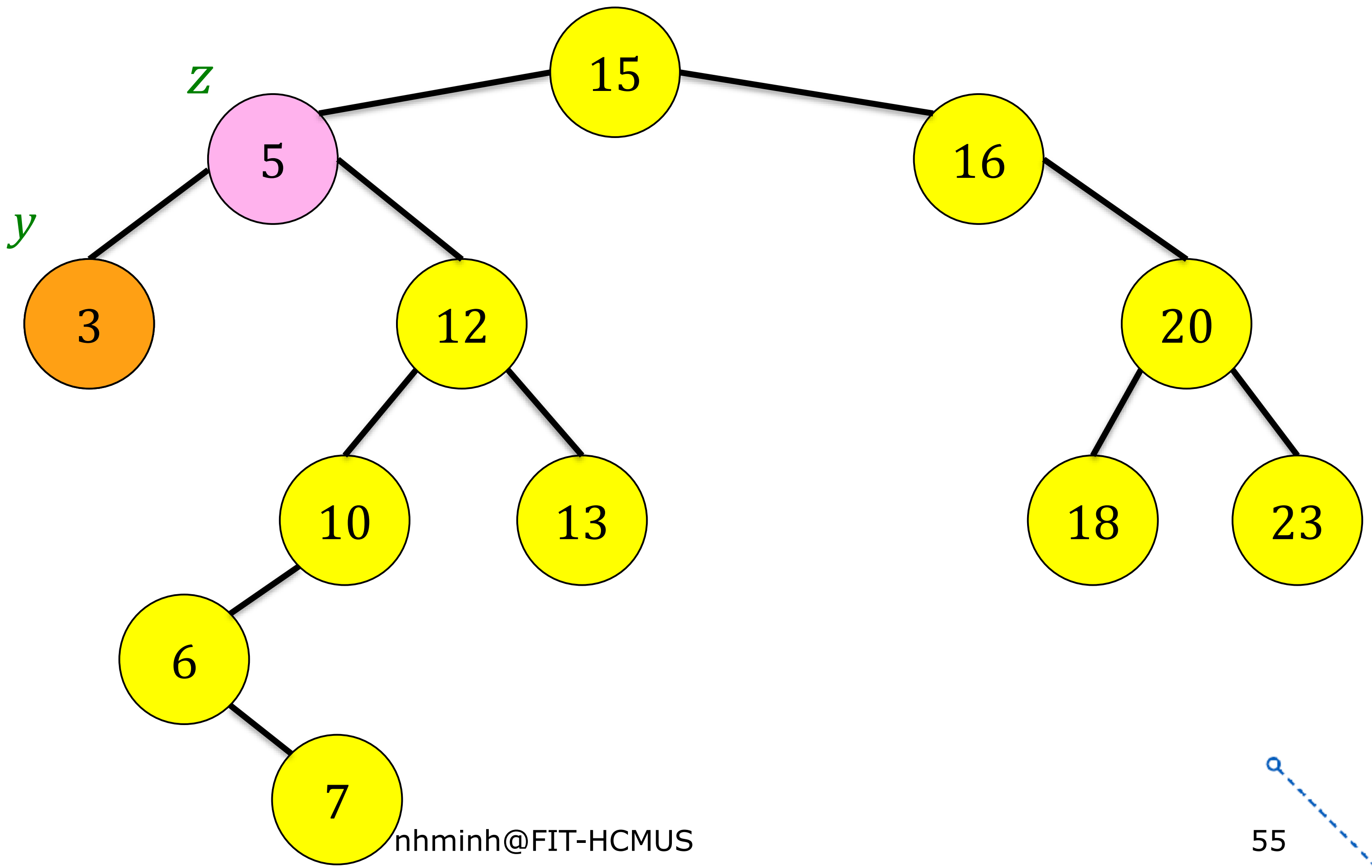
Deletion – case 3

- *z* has 2 children: replace *z.key* by *y.key*, then delete *y*



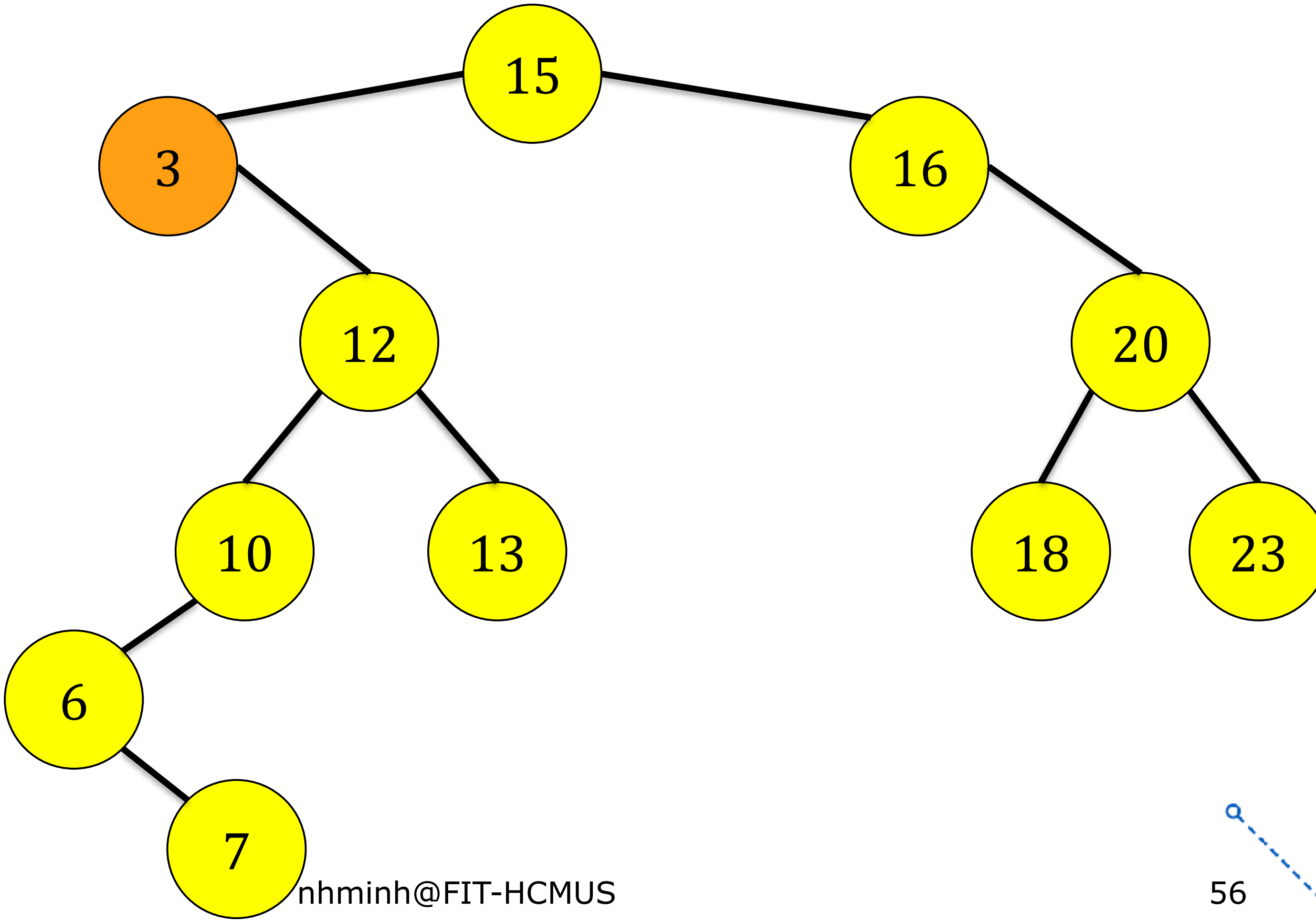
Deletion – case 3

- z has 2 children: find z's predecessor y



Deletion – case 3

- *z* has 2 children: replace *z.key* by *y.key*, then delete *y*

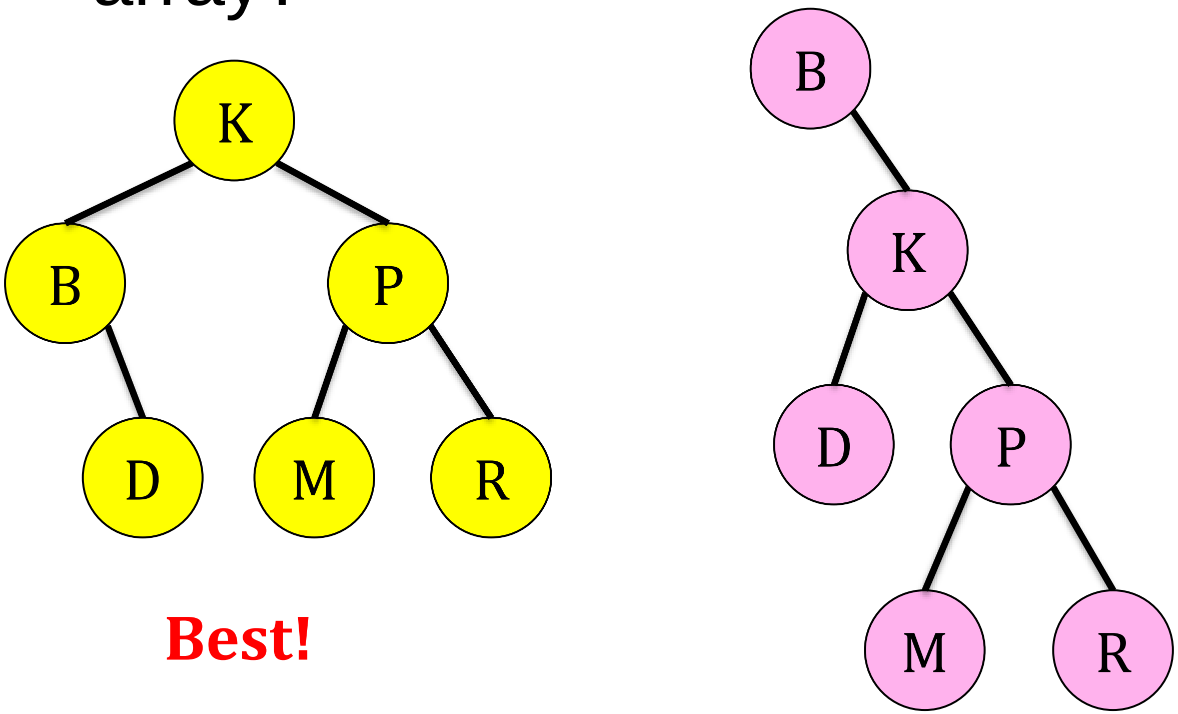


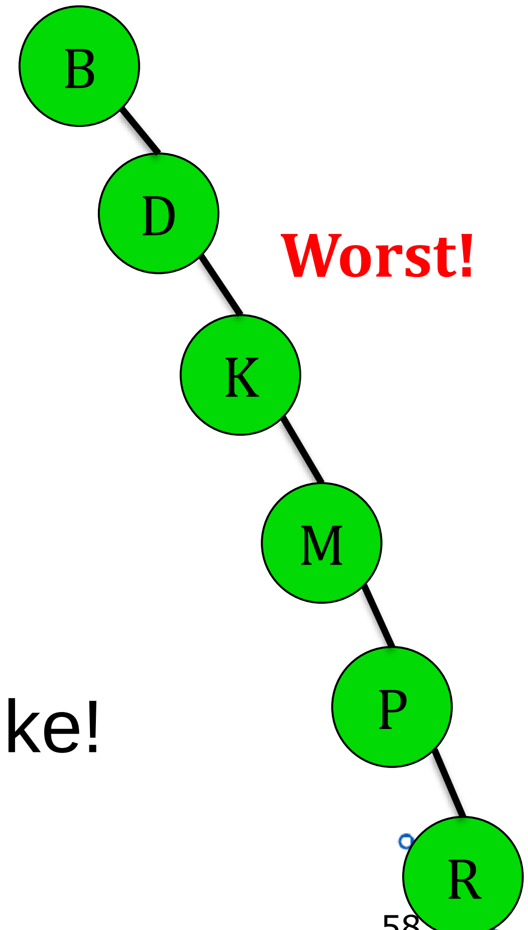
BST Analysis

	BST (*)	Ordered array	Linked list
Searching	$O(\log_2 n)$	$O(\log_2 n)$	O(n)
Insertion	$O(\log_2 n)$	O(n)	O(1)
Deletion	$O(\log_2 n)$	O(n)	O(1)
Memory to store 1 element	Sizeof(key)+8	Sizeof(key)	Sizeof(key)+4

Balancing a tree

- Is searching a BST tree as fast as an ordered array?





- It depends on what the tree looks like!

  → ***Balanced tree*** is the best!

Balancing a tree

- **Definition**. A binary tree is *height-balanced* or simply *balanced* if the **difference in height** of both subtrees of any node in the tree is either **zero** or **one**.
- A tree is *perfectly balanced* if every path from root to leaf has same length.
- Techniques:
  1. Reordering data themselves and then building a tree.
  2. Constantly restructuring the tree when elements arrive and lead to an unbalanced tree.

Balancing a tree – using sorted array

- Steps to balance a tree:
  - Store all data in an array.
  - Sort the array
  - The root is in the middle of the array.
  - The left child of the root is in the middle of the first subarray (from first element → root)
  - The right child of the root is in the middle of the second subarray (from the root → the last element)

Balancing a tree using sorted array – Example

- Stream of data:
- Sorted data:

5	1	9	8	7	0	2	3	4	6

0	1	2	3	4	5	6	7	8	9

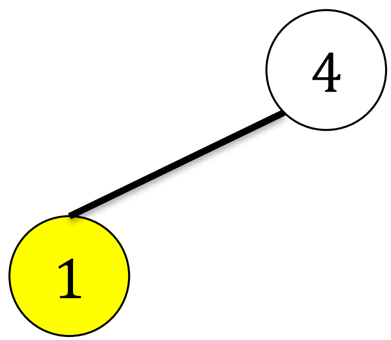
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9



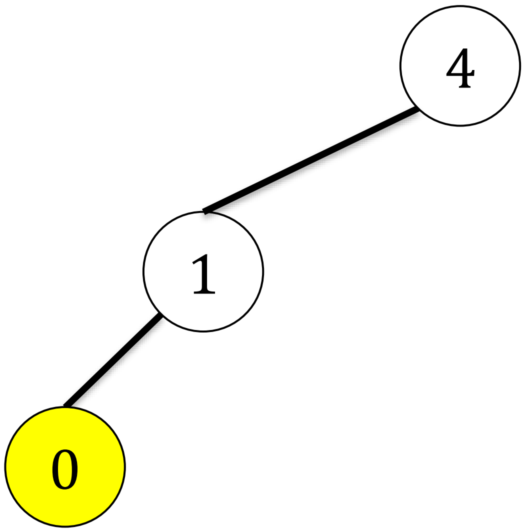
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9



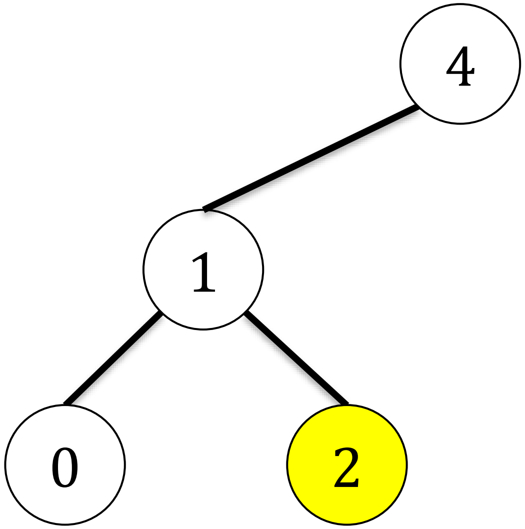
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9



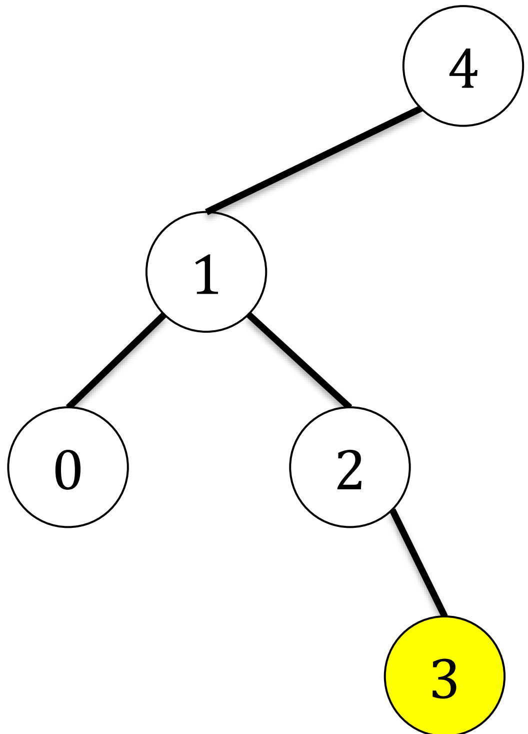
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9



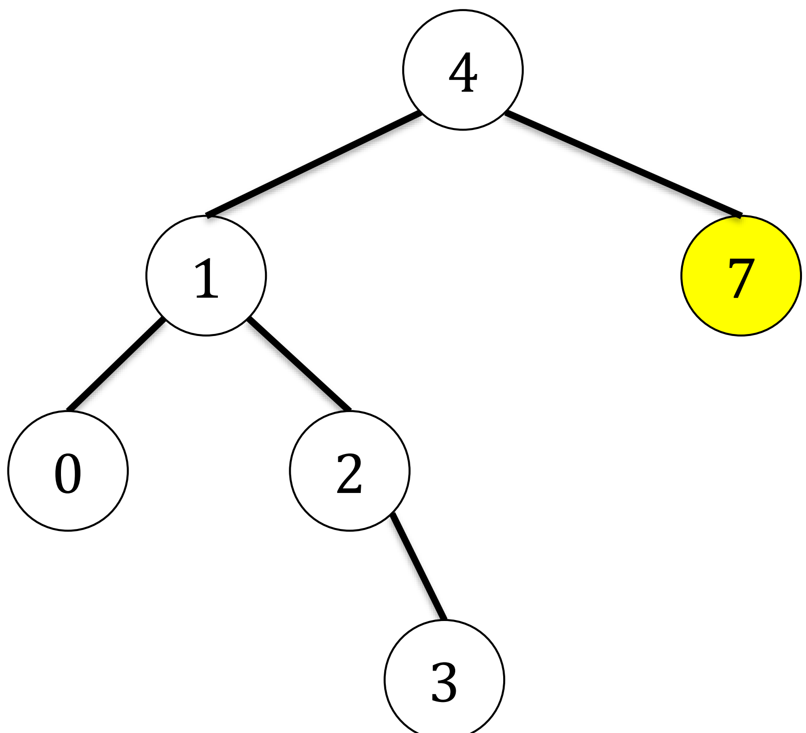
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9



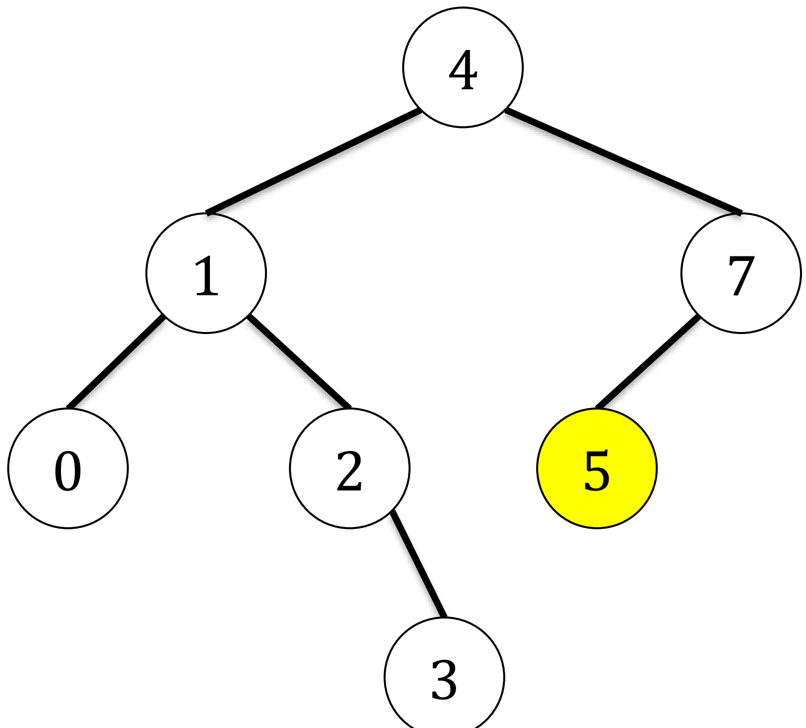
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9



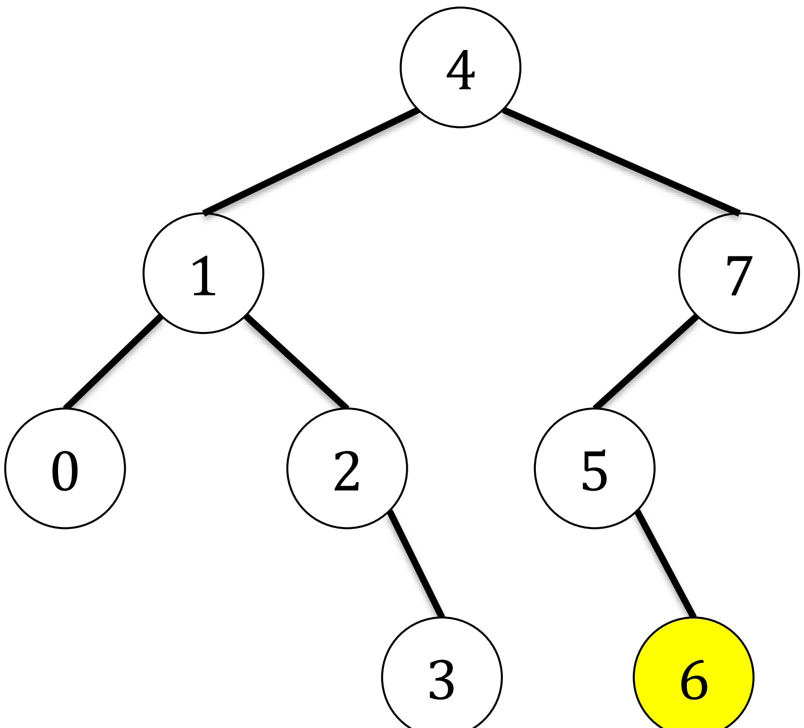
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9



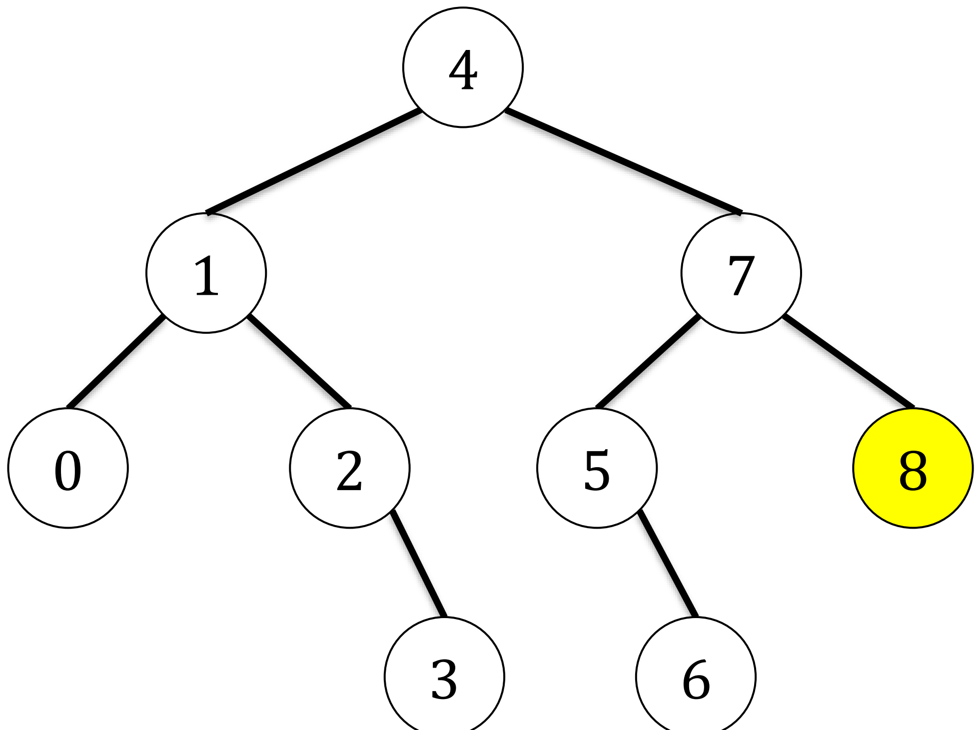
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9



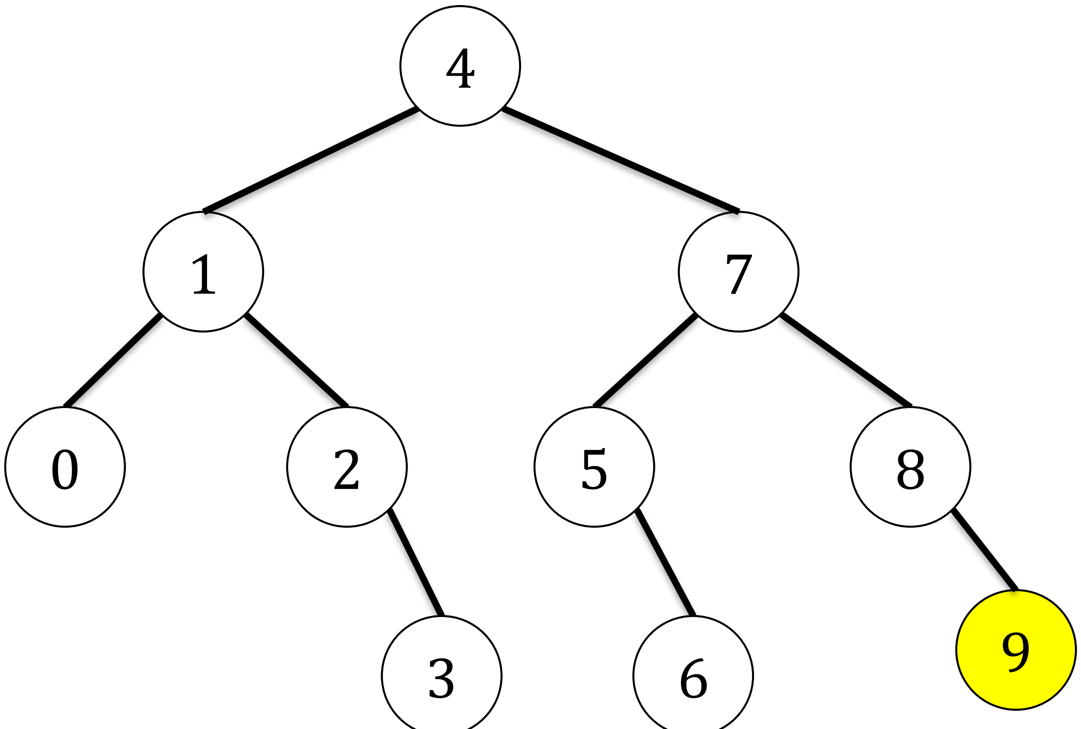
Balancing a tree using sorted array – Example

- Stream of data:

5	1	9	8	7	0	2	3	4	6

- Sorted data:

0	1	2	3	4	5	6	7	8	9

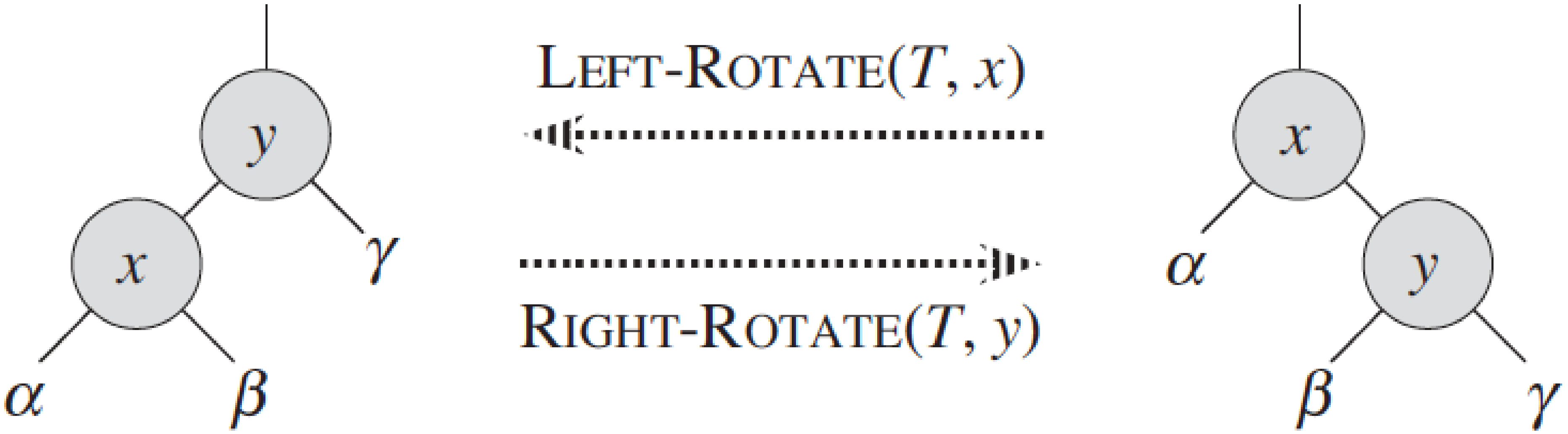


Balancing a tree using sorted array

- Drawback:
  - All data must be put in an array before the tree can be created.
  - Unsuitable when the tree has to be used while the data are still coming.

Balancing a tree – DSW algorithm

- Devised by Colin Day and later improved by Quentin F. Stout and Bette L. Warren.
  - No sorting required
  - Using tree rotation (left/right rotation)



Balancing a tree – DSW algorithm

- Devised by Colin Day and later improved by Quentin F. Stout and Bette L. Warren.
  1. Transfigure an arbitrary BST into a linked list like tree called *backbone* or *vine*.
  2. This tree is transformed into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent.

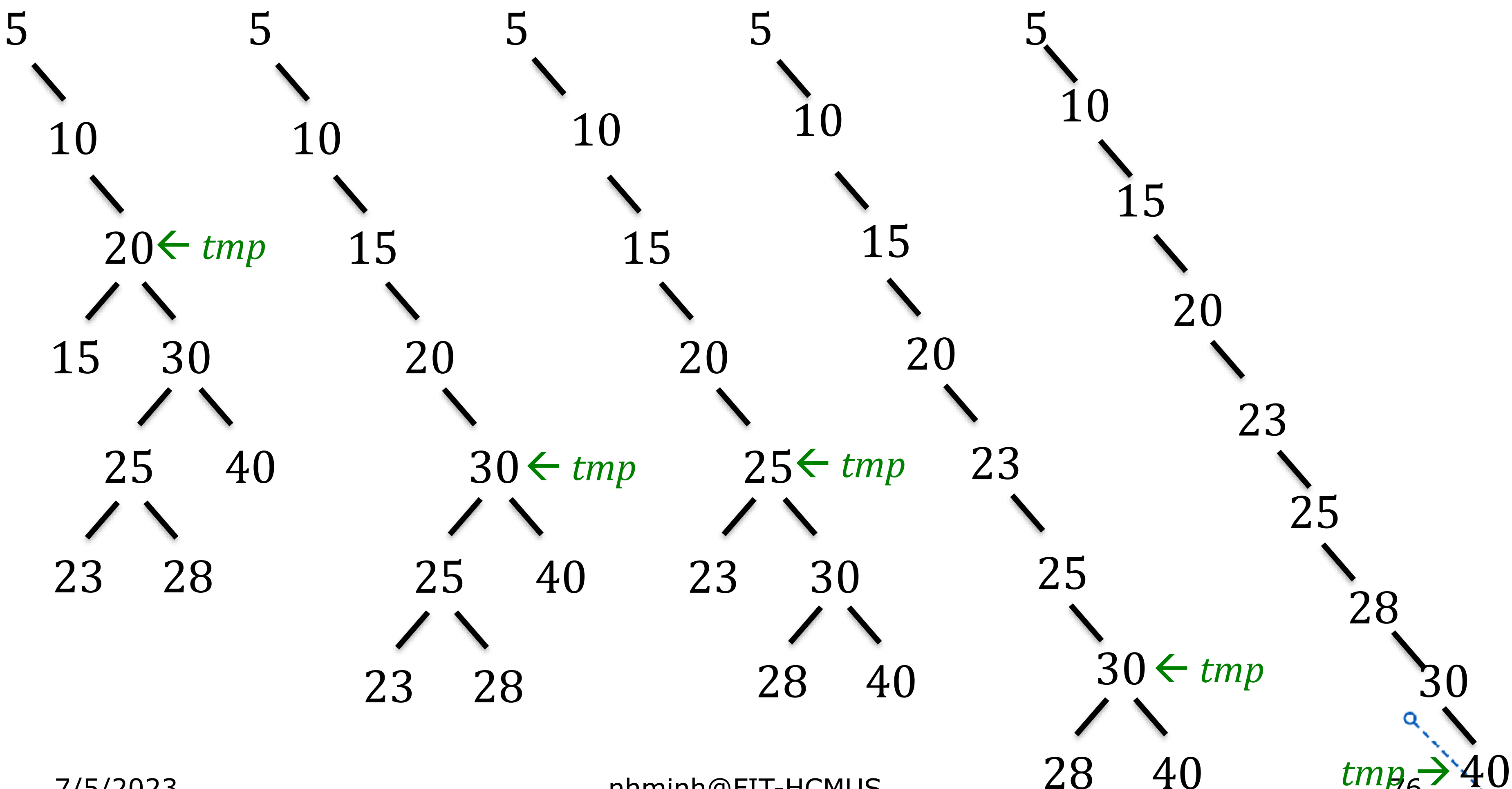
Balancing a tree – DSW algorithm

- Step 1: Transforming a BST into a backbone

```
createBackbone(root)
   tmp = root;
   while (tmp != 0)
      if tmp has a left child
         rotate this child about tmp// hence the left child
                                    // becomes parent of tmp;
         set tmp to the child that just became parent
      else set tmp to its right child
```

Balancing a tree – DSW algorithm

- Step 1: Transforming a BST into a backbone



Balancing a tree – DSW algorithm

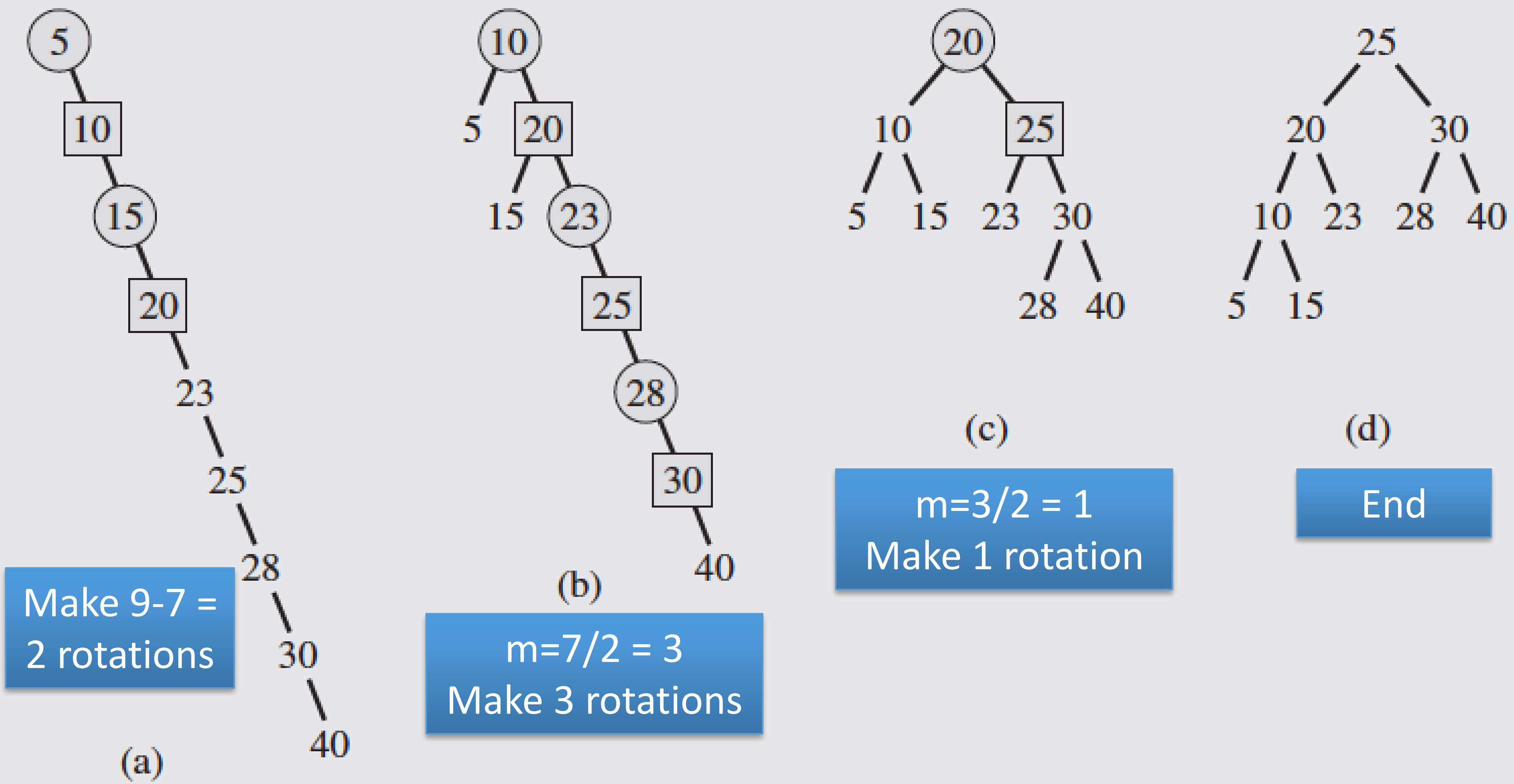
- Step 2: Transform the backbone into a perfectly balanced tree

```
createPerfectTree()
   n = number of nodes;
   m = 2^⌊log2(n+1)⌋−1;
   make n−m rotations starting from the top of backbone;
   while (m > 1)
      m = m/2;
      make m rotations starting from the top of backbone;
```

Here $m = 2^{\lfloor \log 2(n+1) \rfloor} - 1$.

  - *n-m: the number of nodes we expect on the bottommost level.*

Balancing a tree – DSW algorithm



Rotate a tree

```
LEFT-ROTATE(T, x)   //assume that x.right ≠ T.nil
y = x.right                 //set y
x.right = y.left            //turn y's left subtree to x's right subtree
if y.left ≠ T.nil
        y.left.p = x
y.p = x.p                   //link x's parent to y
if x.p == T.nil
        T.root = y
elseif x == x.p.left
        x.p.left = y
else x.p.right = y
y.left = x                  //put x on y's left
x.p = y
```

$\boxed{O(1)}$

Heap

- A particular kind of binary tree:
  - The value of each node ≥ the values of its children (MAX-HEAP).
  - The tree is perfectly balanced, all leaves in the last level are all in the leftmost positions.
- Characteristics of heaps:
  - Review in Lecture 2 (Heapsort)
- Applications of a heap:
  - Heapsort
  - Priority queue

Priority queue

- **In which circumstances the FIFO of a queue is not good?**
  - Pregnant women, the elderly, kids, disabled people
  - Emergency
  - Police
  - Fire fight
  - Elevator
  - …
- A ***priority queue*** is necessary!

Implementing a priority queue

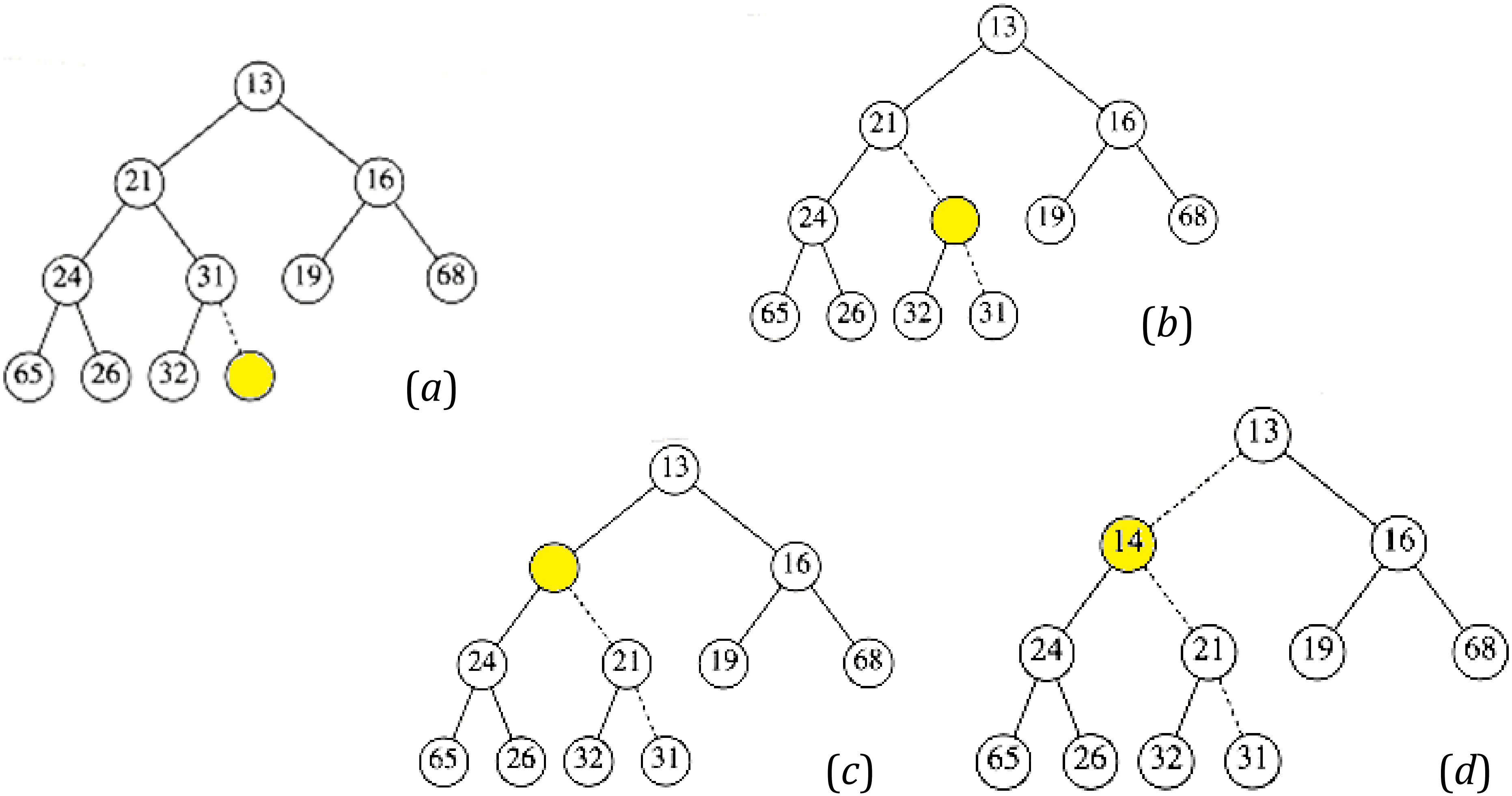
- Ordered array:
  - Insert: $O(n)$
  - Delete-min: $O(1)$
- Linked list
  - Insert: $O(1)$
  - Delete-min: $O(n)$
- Binary search tree(*)
  - Insert: $O(\log_2 n)$
  - Delete-min: $O(\log_2 n)$

(*): *balanced BST*

- Heap:
  - Insert: $O(\log_2 \mathrm{n})$
  - Delete-min: $O(\log_2 \mathrm{n})$

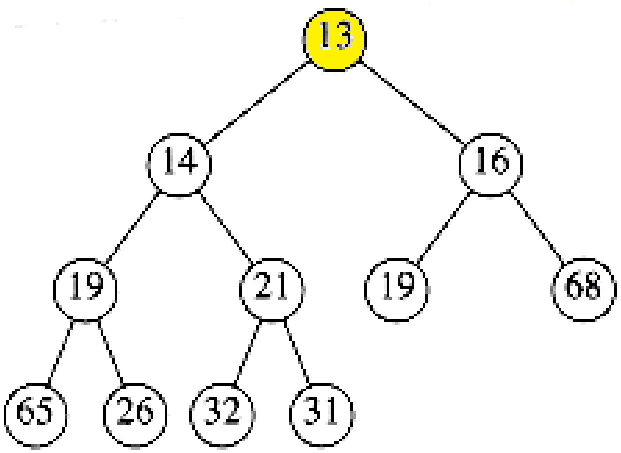
Implementing a priority queue using a heap

- Insert 14 to the heap:

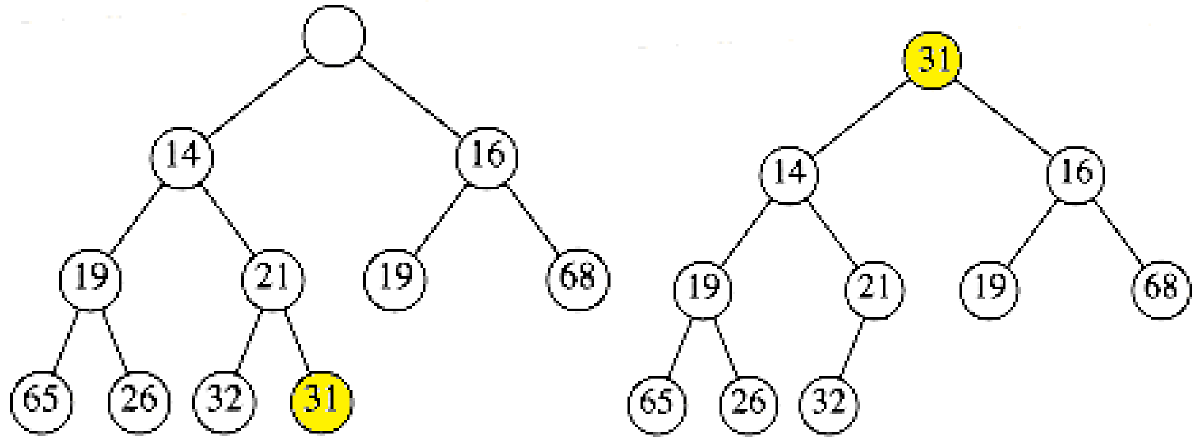


Implementing a priority queue using a heap

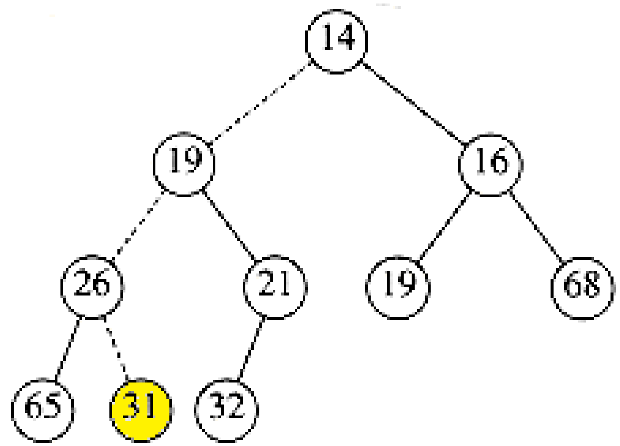
- Delete-min:



(a) Remove the root



(b) Replace by the last node



(c) HEAPIFY

What's next?

- After today:
  - Read textbook 1 – Chapter 12 (page 418~)
  - Read textbook 2 – Chapter 15, 16 (page 452~)
  - Do Homework 6
- Next class:
  - Midterm Examination.
  - Topic: Lecture 1 to Lecture 6

Q&A