Lecture 3
Tree Data Structures

Subject Lecturer: Kevin K.F. YUEN, PhD.

Acknowledgement: Slides were offered from Prof. Ken Yiu.
Some parts has been revised and indicated.

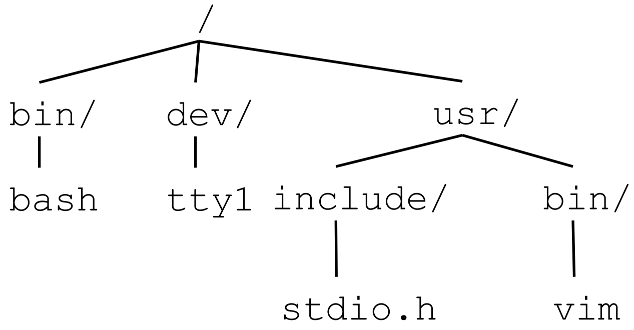
Outline

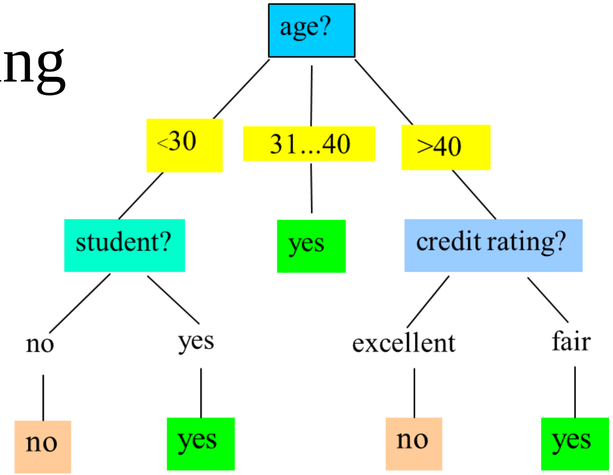
- General tree
- Binary tree
- Binary search tree
- Balanced tree: AVL tree

Applications of trees

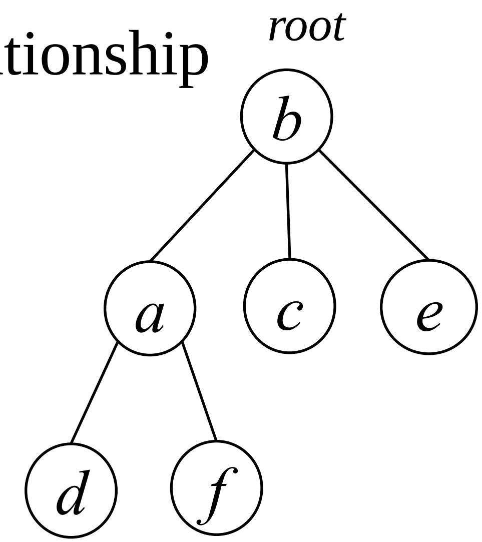
- **Tree** is a *hierarchical* data structure

- Applications
  - Structure/data modeling
    - File system, XML, organization tree
  - Database systems: B-tree, R-tree
  - Data compression: Huffman coding
  - Compilers: syntax tree
  - Data mining: decision tree





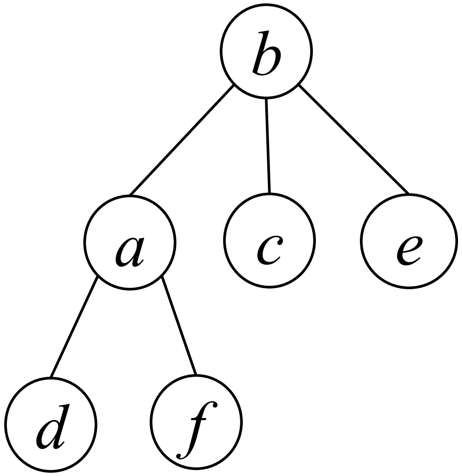
Tree definitions



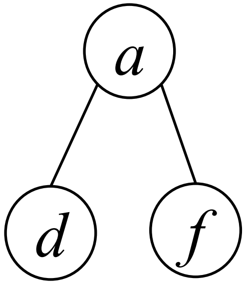
- A **tree** is a set of **nodes** that have **parent-child** relationship
  - E.g., *a*, *c*, *e* are the children of *b*
  - E.g., *b* is the parent of *a*, *c*, *e*
- **Root**: the node without parent
  - E.g., *b*
- **Internal node**: a node with at least one child
  - E.g., *b*, *a*
- **Leaf node** (a.k.a. external node): a node without any child
  - E.g., *d*, *f*, *c*, *e*
- The **ancestors** of a node *v* are all nodes *on the path* from the root to *v*, except *v* itself
  - E.g., the ancestors of *d* are *a*, *b*
- The **descendants** of a node *v* are all nodes that take *v* as their ancestor
  - E.g., the descendants of *b* are *a*, *c*, *e*, *d*, *f*

Tree definitions

Tree T



- **Depth** of a node: the number of ancestors
  - E.g., the depth of b is 0
  - E.g., the depth of d is 2
- **Height** of a tree: the maximum depth of any node
  - E.g., the height of tree T is 2
- **Subtree** rooted at node v: the tree consisting of v and its descendants (including edges connected to descendants)
  - *Example*: the subtree rooted at a

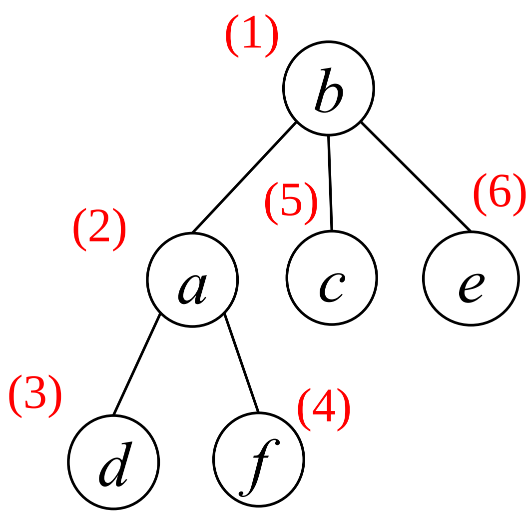


- A tree is **ordered** if there is a linear ordering for the children of each node
  - The ordering is visualized by arranging child nodes left to right
  - E.g., a, c, e are the 1st child, 2nd child, 3rd child of b, respectively
- How to traverse an ordered tree?

Preorder traversal

- **Preorder**: visit a node before its descendants
- *Example*: print a tree T by using preorder traversal
  - Implement visit(v) (at line 1) by "print *v.element*"
  - Run the algorithm with preorder (*T.root*)
    - The visiting order is indicated by the numbers in red (in brackets)
  - The result: *b a d f c e*

Tree T



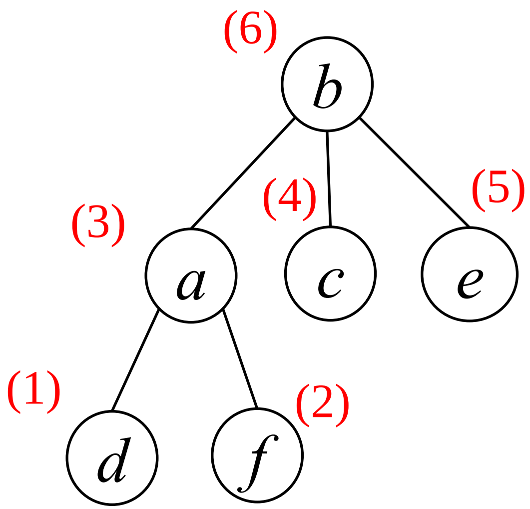
<u>preorder(v)</u>

1. visit(v)
2. for each child w of v
3. preorder(w)

Postorder traversal

- **Postorder**: visit a node after its descendants
- *Example*: print a tree T by using postorder traversal
  - Implement visit(v) (at line 3) by "print $v.element$"
  - Run the algorithm with postorder ($T.root$)
    - The visiting order is indicated by the numbers in red (in brackets)
  - The result: $d\ f\ a\ c\ e\ b$

Tree T



postorder(v)

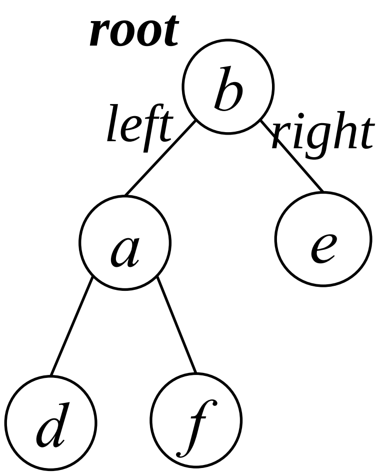
1. for each child w of v
2. postorder(w)
3. visit(v)

Outline

- General tree
- Binary tree
- Binary search tree
- Balanced tree: AVL tree

Binary tree structure

- A binary tree ***T*** is an ordered tree



- It has a root node *T.root*
- Each node *v* stores the following attributes:
  - *v.element*: data element
  - *v.left*: reference to the left child node
  - *v.right*: reference to the right child node
  - *v.p*: reference to the parent node [optional]
- A reference is set to `null` if the corresponding child node is missing
- *v* is a leaf node if *v.left* = *v.right* = `null`

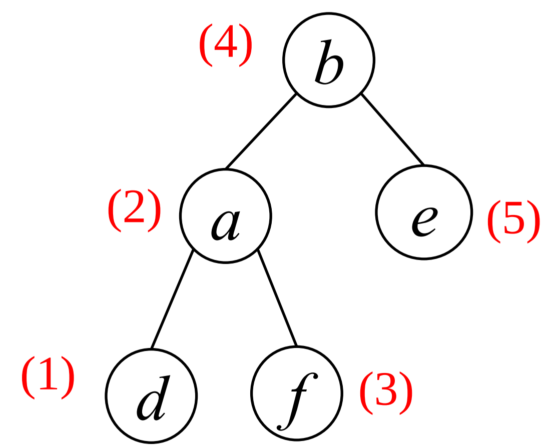
Inorder traversal

- **Inorder**: visit a node after its left subtree and before its right subtree
- *Example*: print a tree T by using inorder traversal
  - Implement visit(v) (at line 3) by "print $v.element$"
  - Run the algorithm with inorder ($T.root$)
    - The visiting order is indicated by the numbers in red (in brackets)
  - The result: $d\ a\ f\ b\ e$

<u>inorder(v)</u>

1. if $v.left \neq$ `null`
2.   inorder($v.left$)
3. visit(v)
4. if $v.right \neq$ `null`
5.   inorder($v.right$)

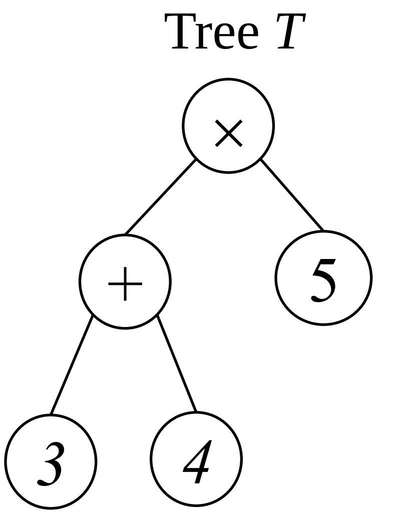
Tree T



Tree traversal: applications

- We can use a binary tree T to represent an arithmetic expression, e.g.,

$$(3 + 4) \times 5$$



- 1. How to modify the **postorder** traversal algorithm to compute the result of the expression?

- 2. How to modify the **inorder** traversal algorithm to print the expression?

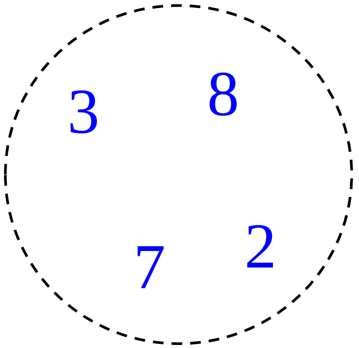
Outline

- General tree
- Binary tree
- Binary search tree
- Balanced tree: AVL tree

Binary search tree: applications

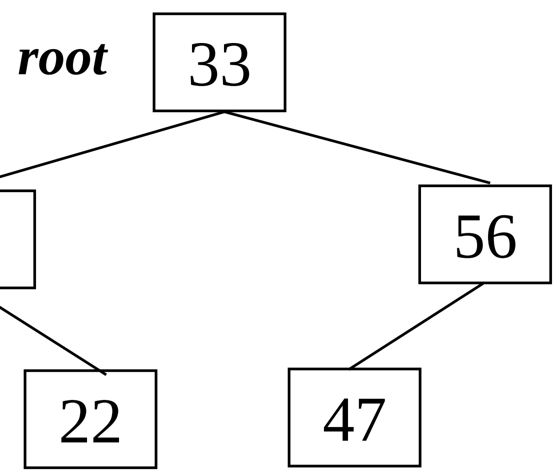
- Binary search tree
  - A data structure that supports efficient operations on a set, e.g., searching, insertion, deletion
- Applications
  - Index of items in a set
  - Dictionary
  - Browsing the data items in an order

Set S

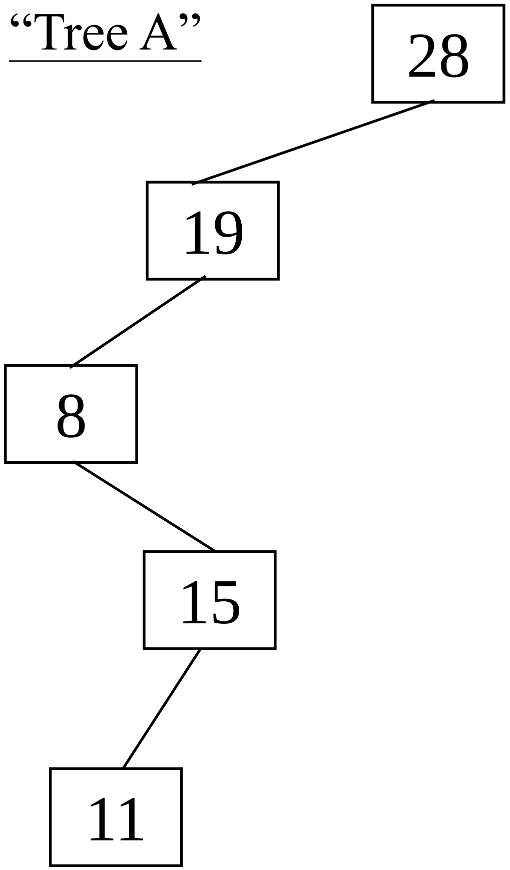


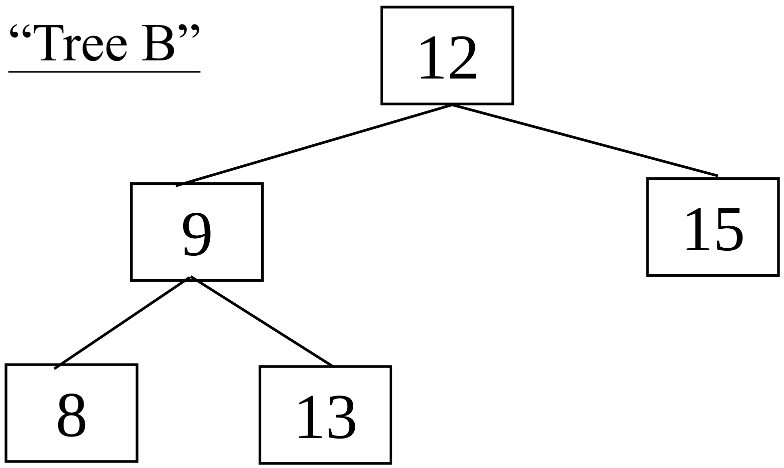
Binary search tree

- It is a binary tree:
  - each node v stores $v.key$, $v.left$, $v.right$, v,p
- that satisfies the "**binary search tree property**":
  - all keys in the left subtree of v are less than $v.key$
  - all keys in the right subtree of v are greater than $v.key$



[Question] Which tree is **not** a binary search tree?





Binary search tree: operations

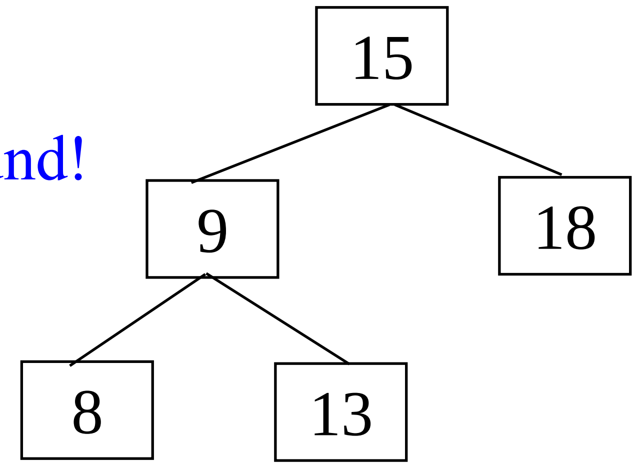
Operation	***Complexity***	***Meaning***
Search	O(h)	Search a node with a key
Minimum	O(h)	Find the minimum node
Maximum	O(h)	Find the maximum node
Insert	O(h)	Insert a key
Delete	O(h)	Delete a key

Tree height: h

However, h can be O(n) in the worst case!

Search

- Find a node with key k
  - Return `null` if there is no such node
- Example: Search ($T.root$, 13)
  - Visit the node "15", go left
  - Visit the node "9", go right
  - Visit the node "13", key found!



Search (x, k)

1. if x = `null` or $k = x.key$
2. return x
3. if $k < x.key$
4. Search($x.left$, k)
5. else
6. Search($x.right$, k)

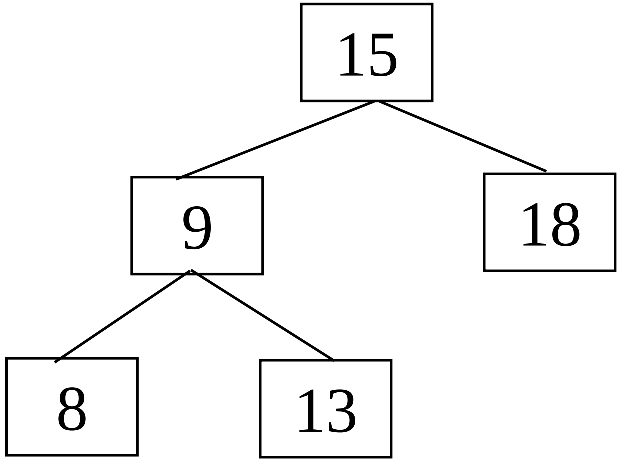
Remark: This algorithm can also be rewritten by using a while-loop

Minimum

- Find the minimum node
- Example: Minimum (*T.root*)
  - Visit the node "15", go left
  - Visit the node "9", go left
  - Visit the node "8", no left child, return the node "8"

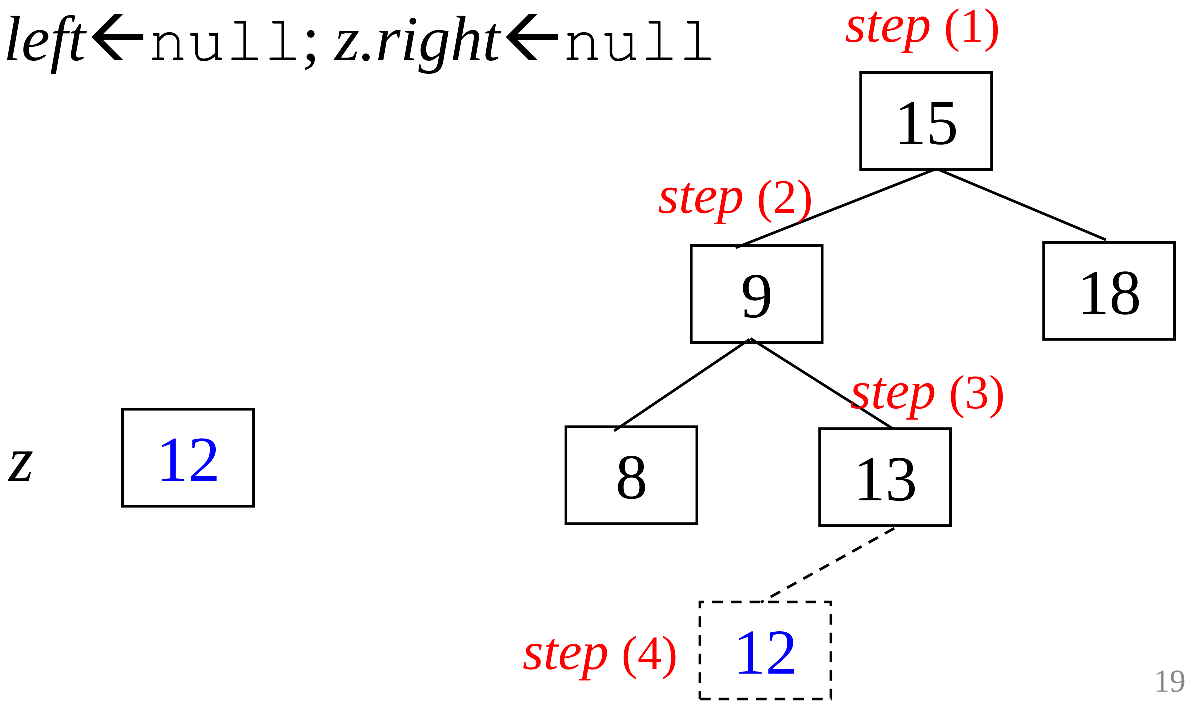
<u>Minimum (*x*)</u>

1. while $x.left \neq$ `null`
2. $\quad x \leftarrow x.left$
3. return x



Insertion: Idea

- Idea: insert a node *z* at the bottom of the tree
  - (1) Search the leaf node *y* such that it can become the parent of *z*, then
  - (2) Insert the node *z* as a child of *y*
- Example:
  - *z.key*←12; *z.left*←`null`; *z.right*←`null`
  - Insert (*T*, *z*)

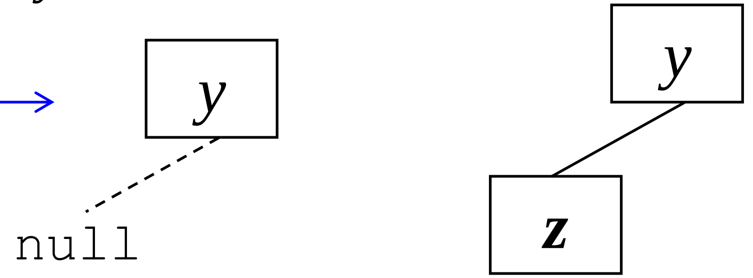


Insertion: Algorithm

Insert (T, z)

```
y ← null;    x ← T.root
while x ≠ null
    y ← x
    if z.key < x.key
        x ← x.left
    else
        x ← x.right
z.p ← y
if y = null
    T.root ← z
else if z.key < y.key
    y.left ← z
else
    y.right ← z
```

y has no left child

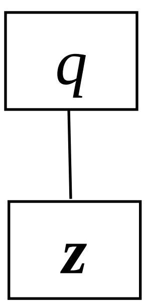


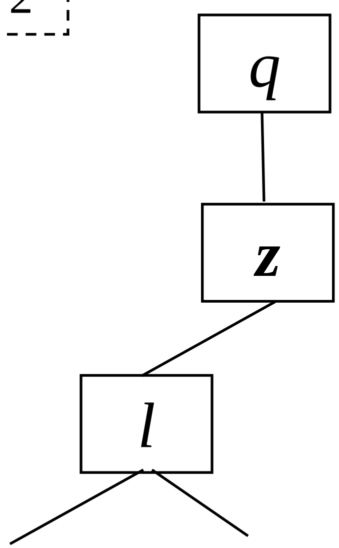
Deletion: Idea

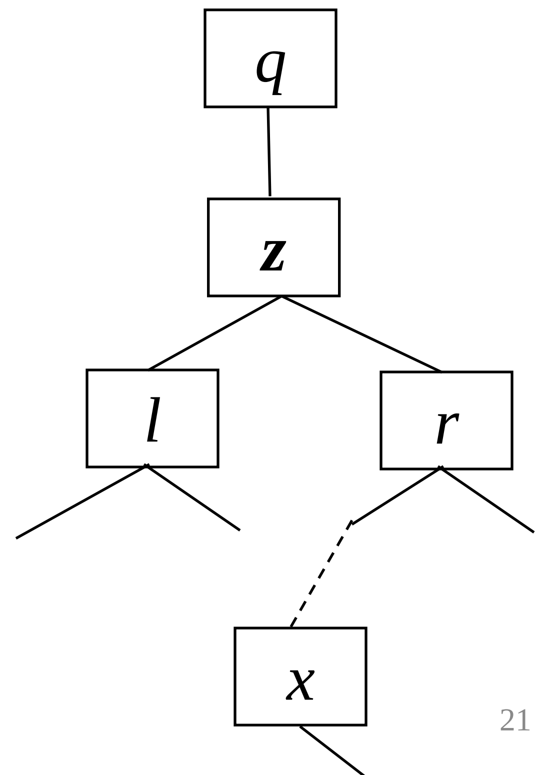
- Idea: consider three **cases** of the node *z* to be deleted
  - (1) *z* has no child: trivial
  - (2) *z* has one child: replace *z* by its child
  - (3) *z* has two children: delete the minimum node *x* of the right subtree of *z*, then replace *z* by *x*

Case 2 has two sub-cases

This must be either case 1 or case 2



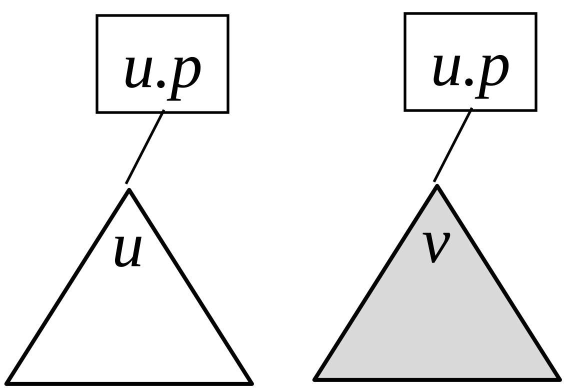




Deletion: Algorithm

<u>Transplant (*T*, *u*, *v*)</u>

if $u.p$ = `null`
    $T.root \leftarrow v$
else if $u = (u.p).left$
    $(u.p).left \leftarrow v$
else
    $(u.p).right \leftarrow v$
if $v \neq$ `null`
    $v.p \leftarrow u.p$



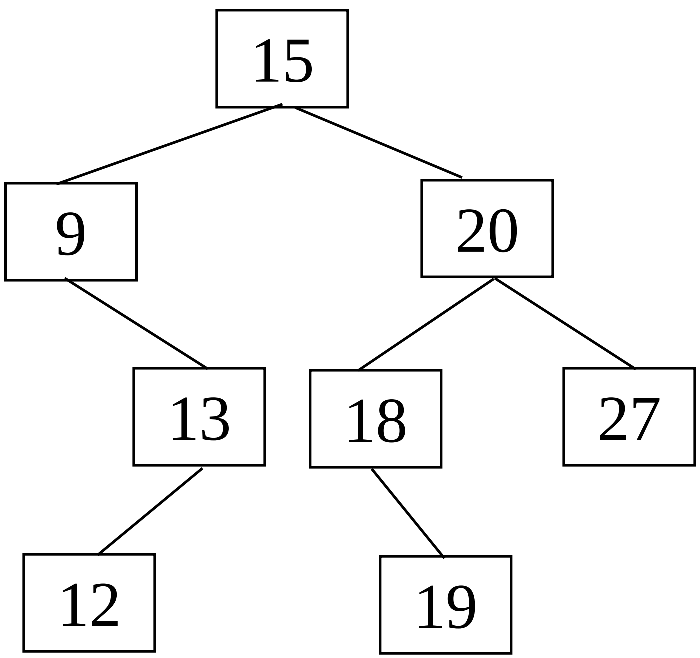
<u>**Delete** (*T*, *z*)</u>

if $z.left$ = `null`
    Transplant(T, z, $z.right$)
else if $z.right$ = `null`
    Transplant(T, z, $z.left$)
else
    $y \leftarrow$ Minimum($z.right$)
    Delete (T, y)
    replace z by y

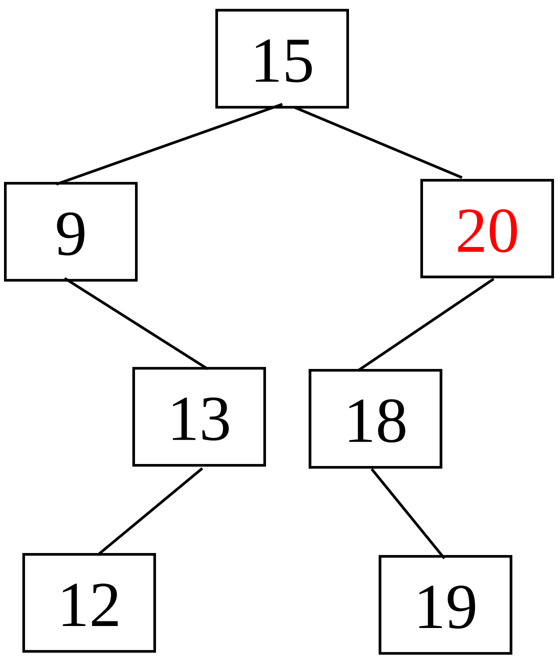
Deletion: Example 1

- Example: Delete (*T*, *node_27*)
  - Find the parent node of "27"
  - Set its right child to `null`,

    i.e., delete the node "27"



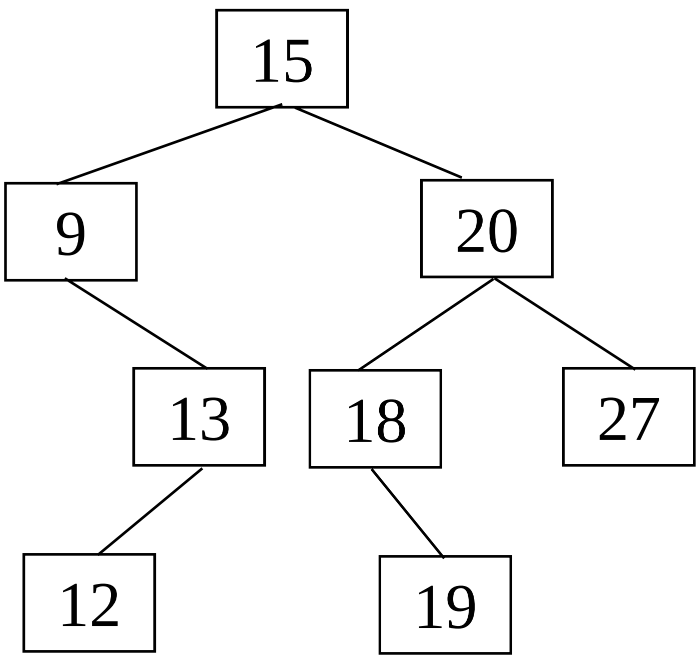
before deletion



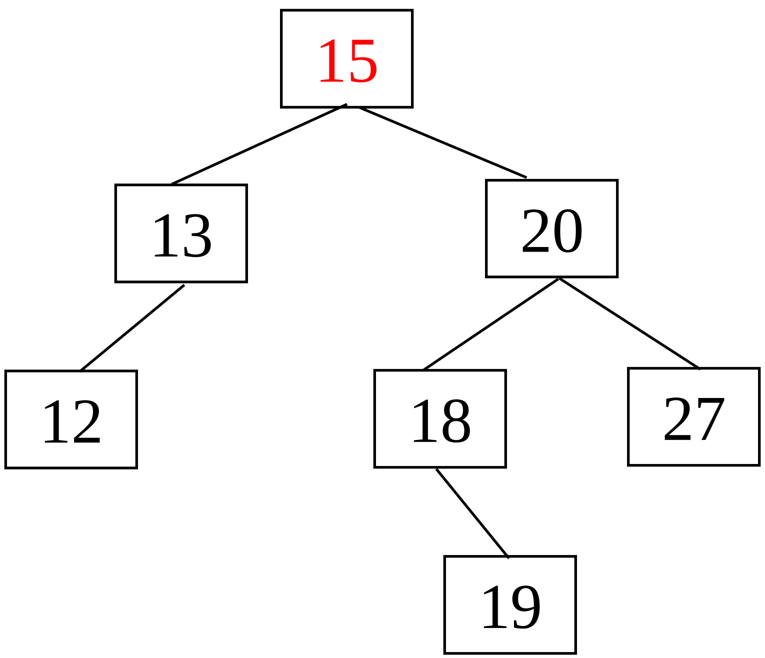
after deletion

Deletion: Example 2

- Example: Delete (*T*, *node_9*)
  - Find the parent node of “9”
  - Set its left child to the child of “9”



before deletion

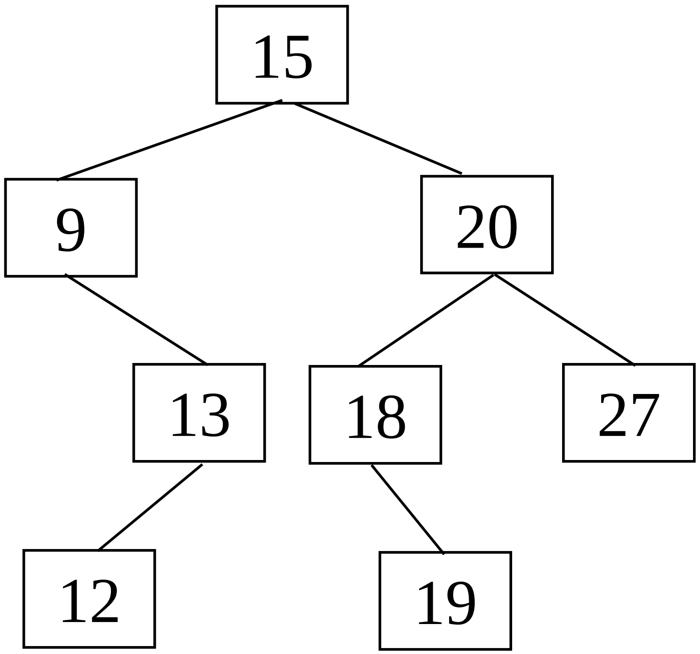


after deletion

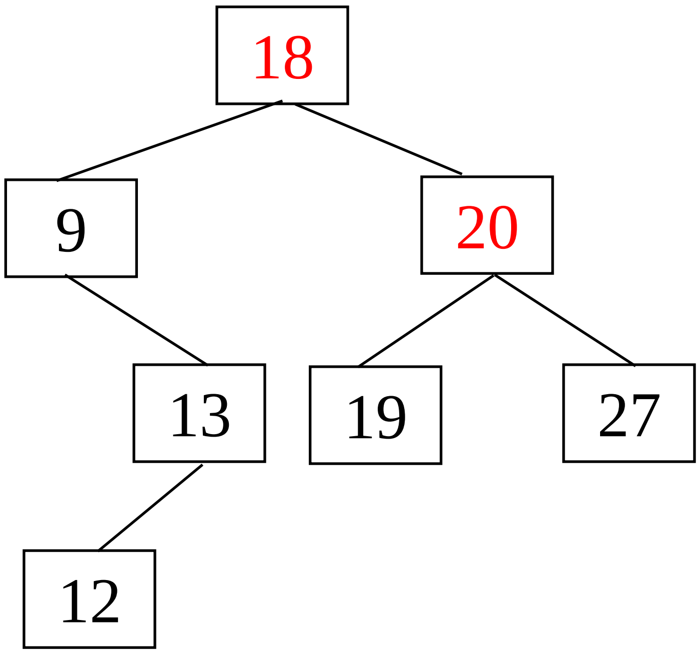
Deletion: Example 3

- Example: Delete (*T*, *node_15*)
  - Replace “15” by its successor “18”
  - Delete its successor “18”

    (like the case in the previous slide)



before deletion



after deletion

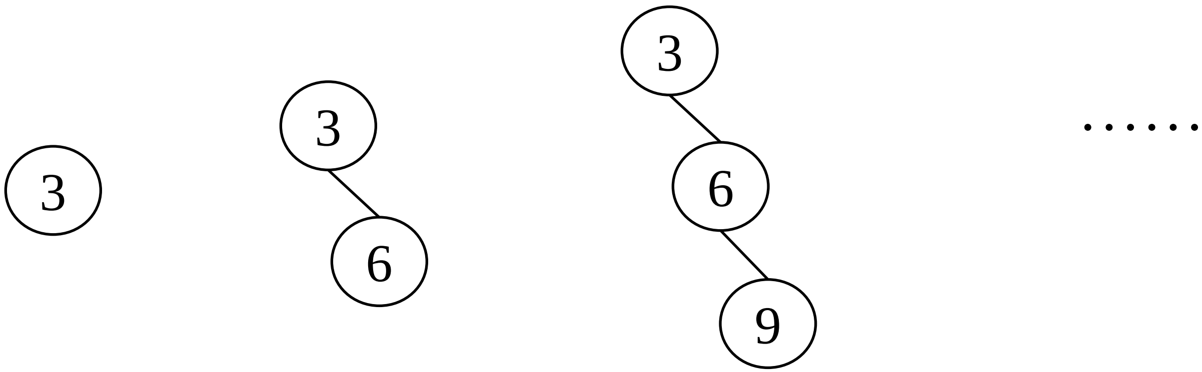
Outline

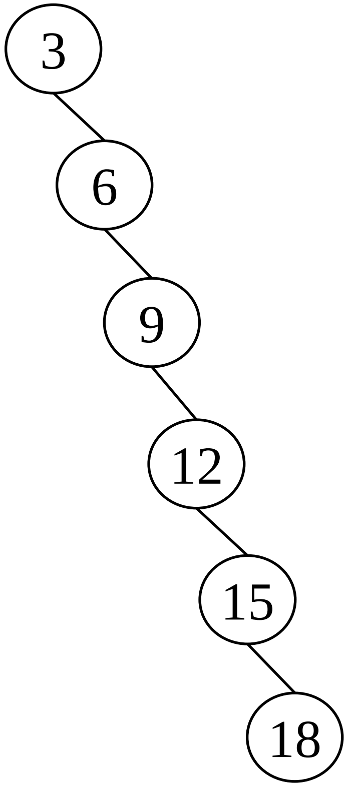
- General tree
- Binary tree
- Binary search tree
- Balanced tree: AVL tree

Unbalanced Tree: Example

- Suppose we insert the keys 3, 6, 9, 12, 15, 18 (in this order) into a binary search tree ……
- Problem: the tree is not “balanced”
  - The right subtree is much taller than the left subtree
  - Tree height h can be as large as n–1 !
  - High search time: O(h) = O(n)

    where n is the number of keys

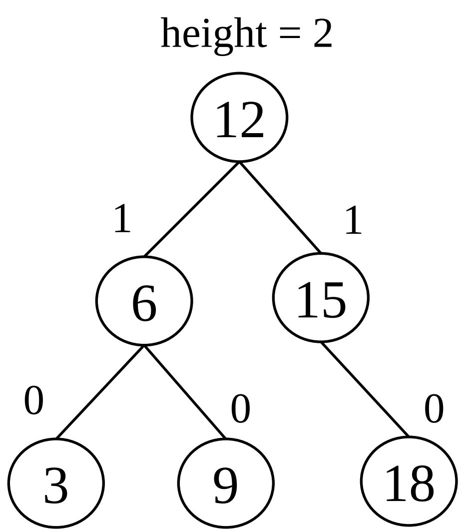




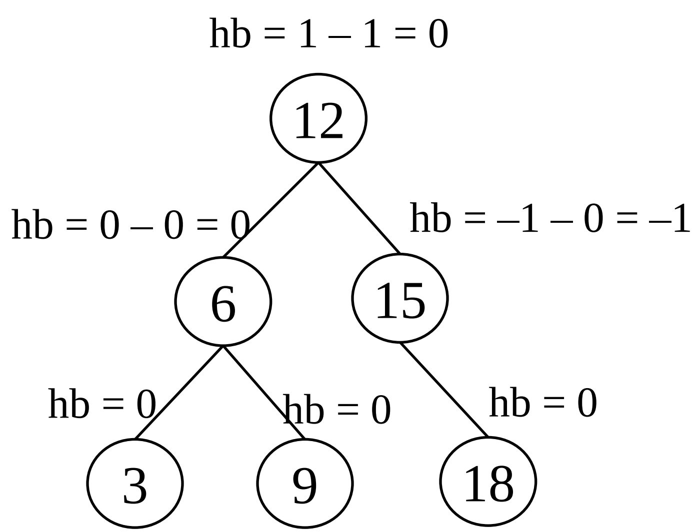
AVL Tree

- The <u>**height-balance**</u> (*hb*) of a node *x* is:
  - *x*'s left child height – *x*'s right child height
    - Special case: `null` node's height = –1
- AVL tree is a height-balanced binary search tree
  - Property: the *hb* of each node is either –1, 0, or 1

    [The property is violated if the *hb* of some node is <–1 or >1]



Heights of tree nodes

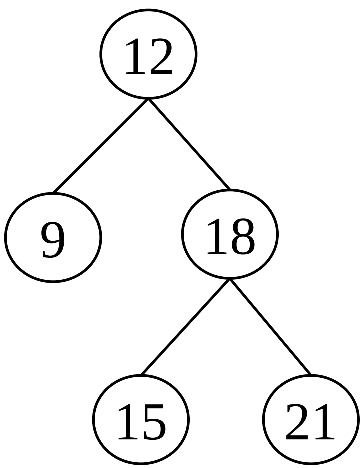


Height-balances of tree nodes

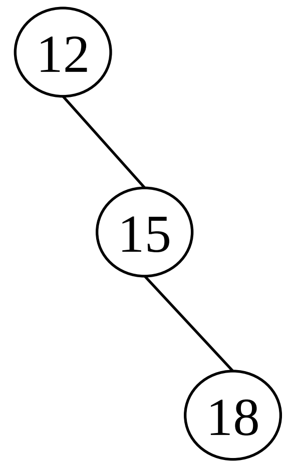
Question

- Which tree satisfies the **AVL tree property**?
  - *Hint*: compute the height-balance of each node

    (first consider nodes at low levels, then nodes at high levels)



Tree A



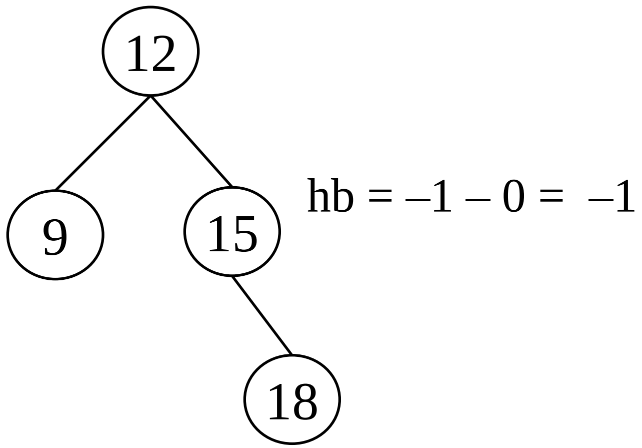
Tree B

Height of AVL Tree

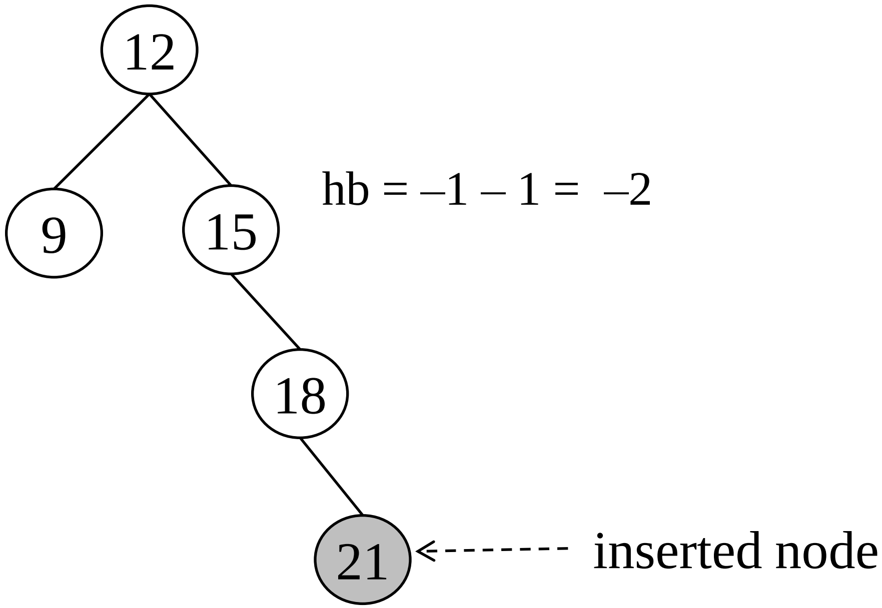
- What is the relationship between the tree height h and the number of nodes n?
- It can be proved that $h \le \log_{\varphi} n$
  - where φ is the golden ratio (1.618)
  - We skip the proof here
- Therefore, $h = O(\log n)$
- AVL tree supports fast searching, insertion, deletion: $O(\log n)$ time

Insertion

- When we insert a key (21) into an AVL tree, some node may have height-balance –2 or 2
  - This violates the AVL tree property
- How to fix this problem?



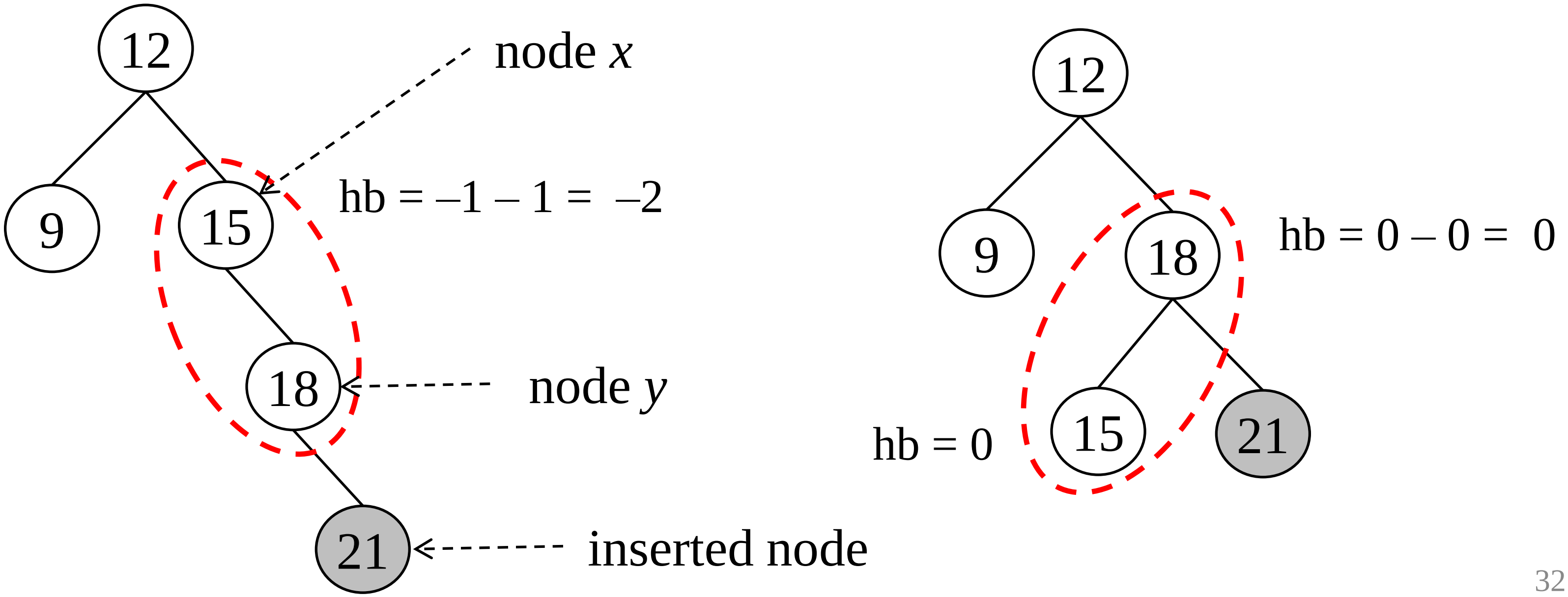
Tree before insertion
(AVL yes)



Tree after insertion
(AVL no)

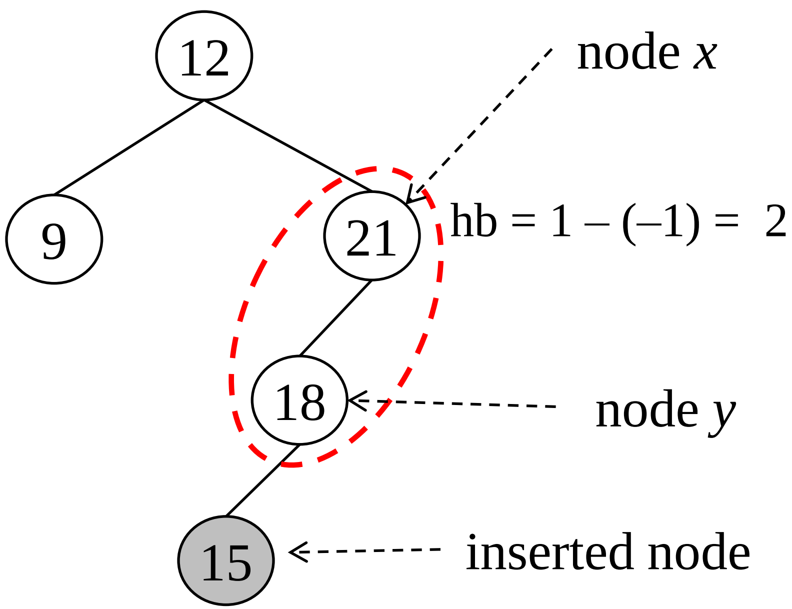
Insertion: Left Rotation

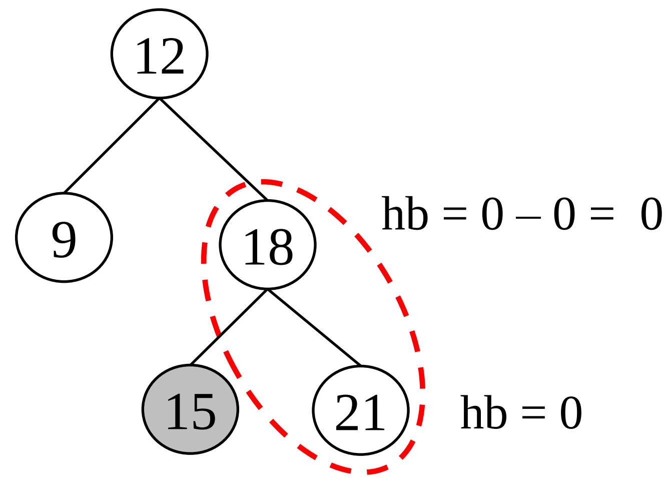
- Solution: **rotate** the node to balance that node
  - Rotate **left** if its height-balance = –2
  - *y* ← *x.right* ; *x.right* ← *y.left* ;
  - *y.left* ← *x*;
  - *x.parent.left/right* ← *y*
    - depending which node is the parent of *x*



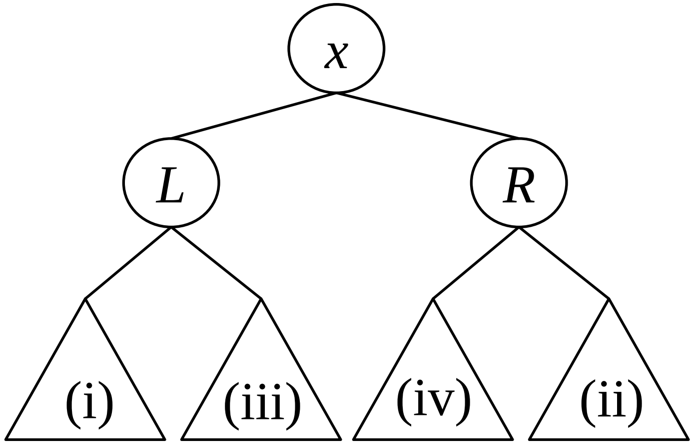
Insertion: Right Rotation

- Solution: **rotate** the node to balance that node
  - Rotate **right** if its height-balance = 2
  - *y* ← *x.left* ; *x.left* ← *y.right* ;
  - *y.right* ← *x*;
  - *x.parent.left/right* ← *y*
    - depending which node is the parent of *x*





Insertion



- After inserting a key *k*,

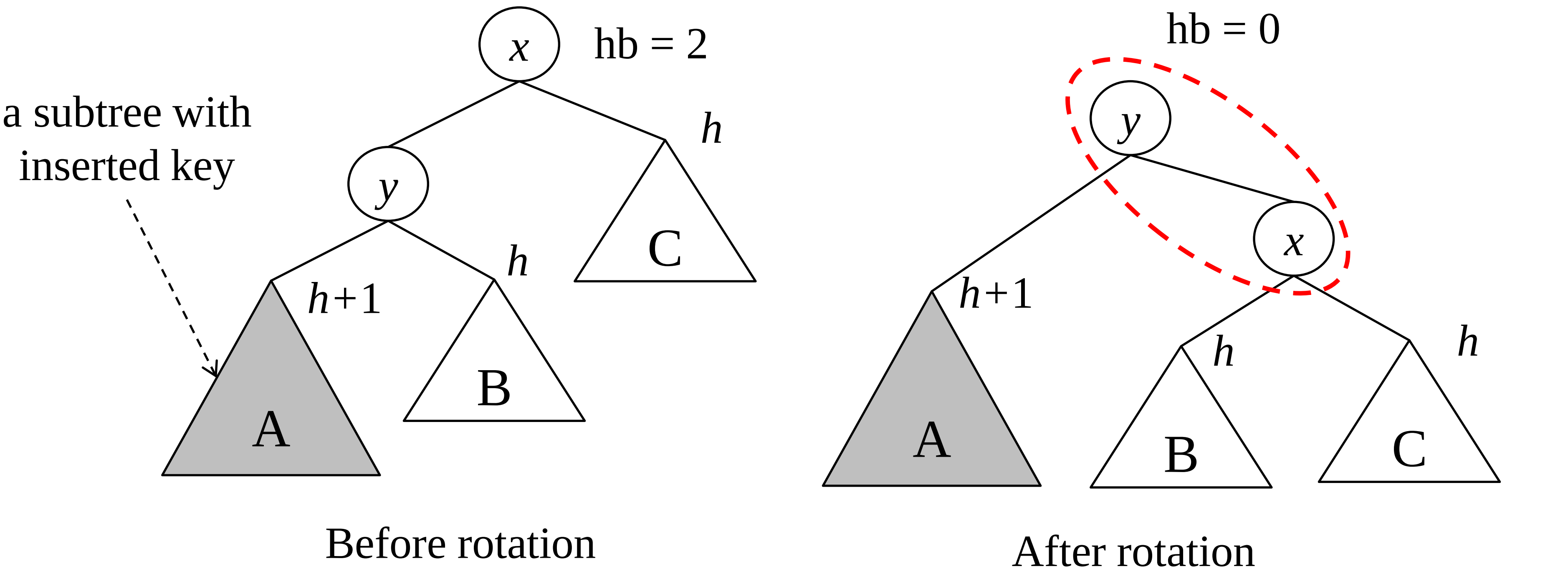
  the cases for balancing a node *x*:

  <u>*Outside cases*</u>: do a single rotation
- (i) *k* inserted into *x*'s left child's left subtree
- (ii) *k* inserted into *x*'s right child's right subtree

  <u>*Inside cases*</u>: do a double rotation
- (iii) *k* inserted into *x*'s left child's right subtree
- (iv) *k* inserted into *x*'s right child's left subtree

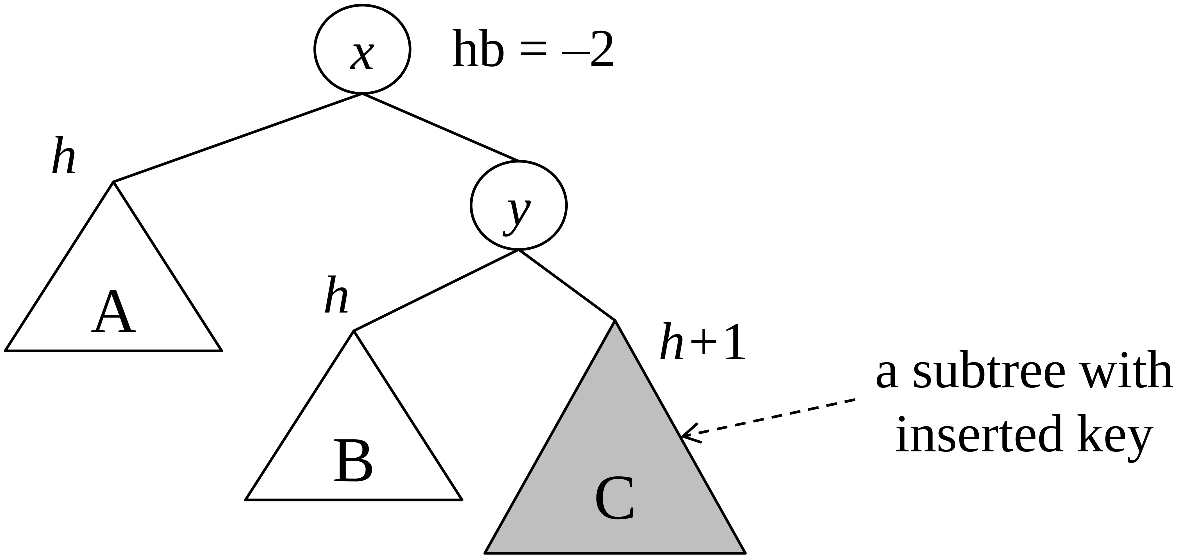
Insertion: Outside Case (i)

- The AVL property is violated at node *x*
- Solution: do a right rotation at *x*, *y*



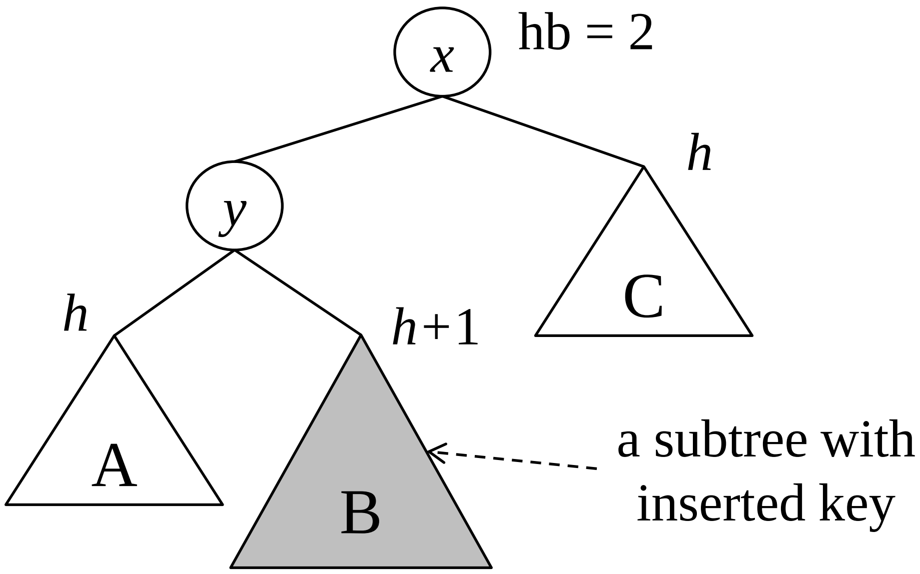
Insertion: Outside Case (ii)

- The AVL property is violated at node *x*
- The solution is similar to that for case (i)

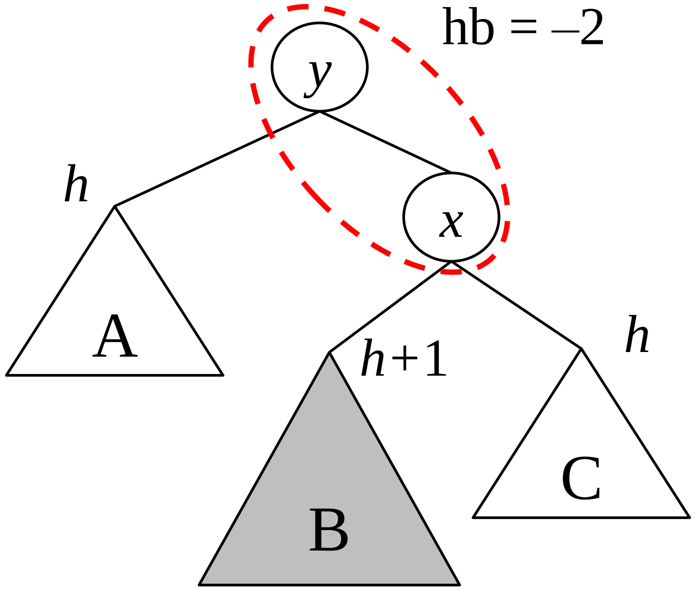


Insertion: Inside Case (iii)

- The AVL property is violated at node x
- Can we solve this by a right rotation?
  - NO! Node y will violate the property!



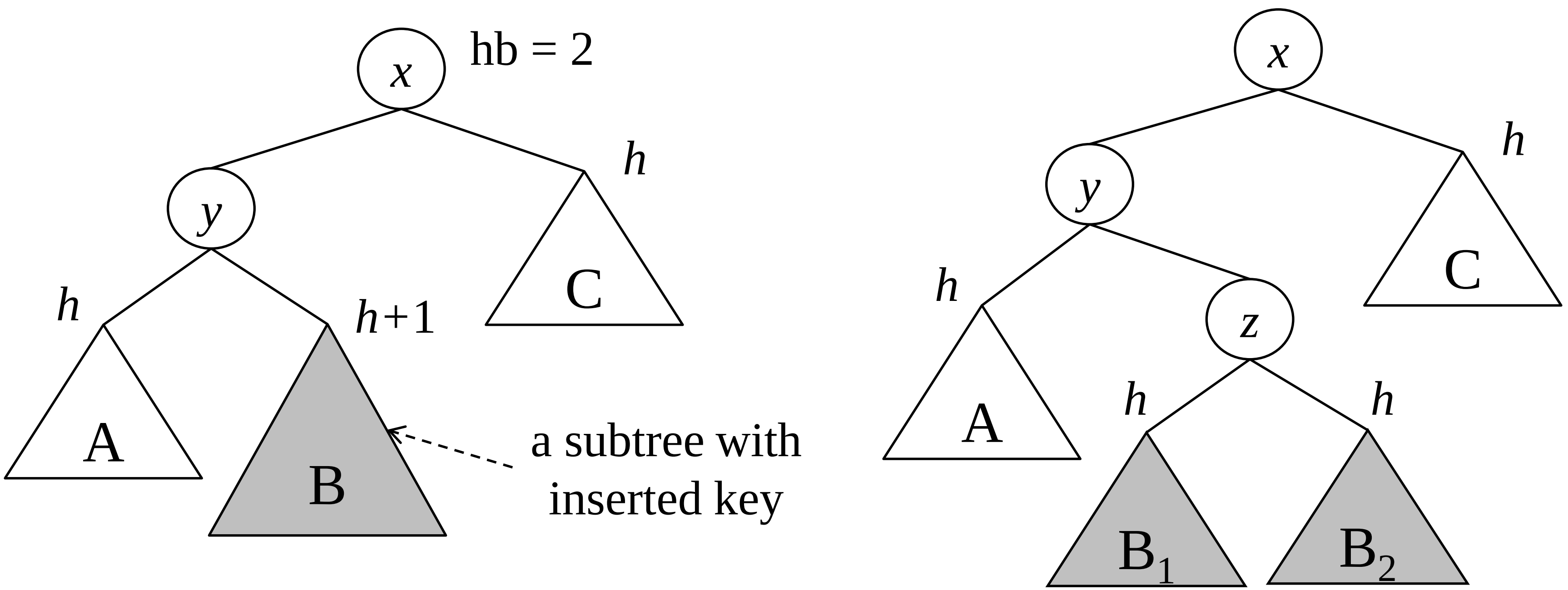
Before rotation



After rotation

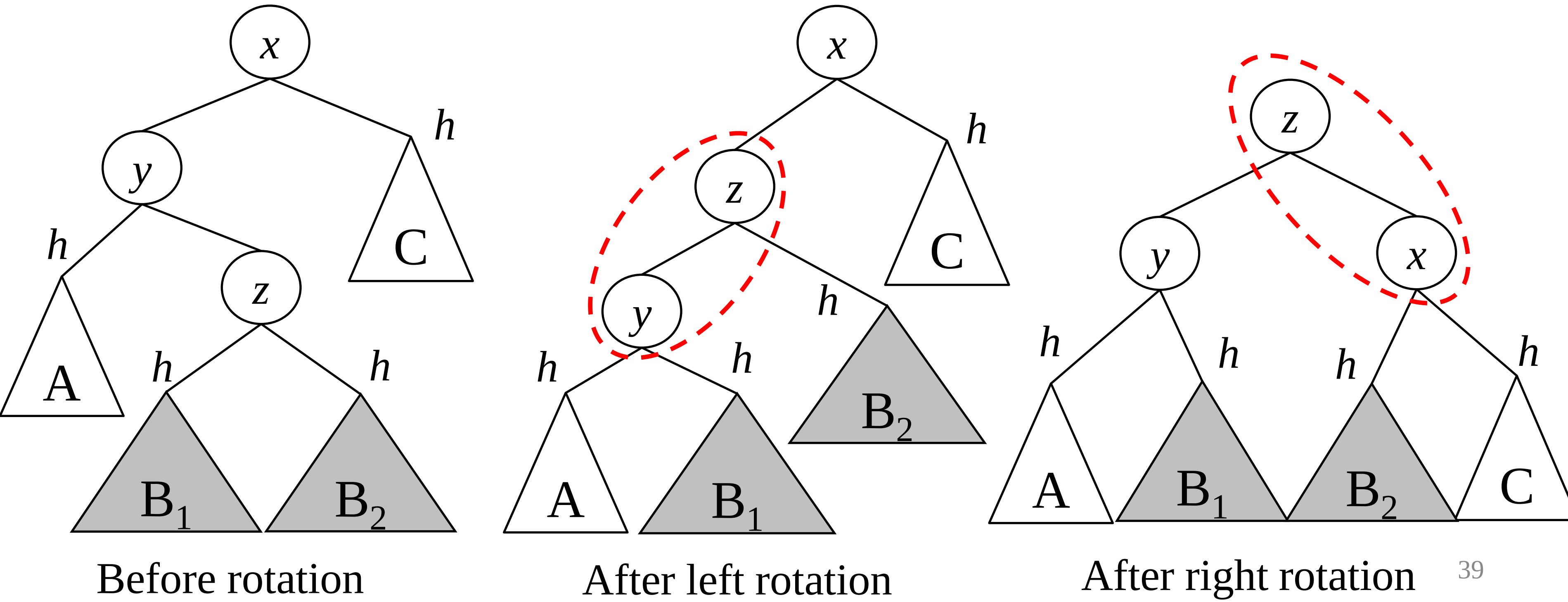
Insertion: Inside Case (iii)

- The AVL property is violated at node x
- Consider the right subtree of y
  - Let z be the root of this subtree



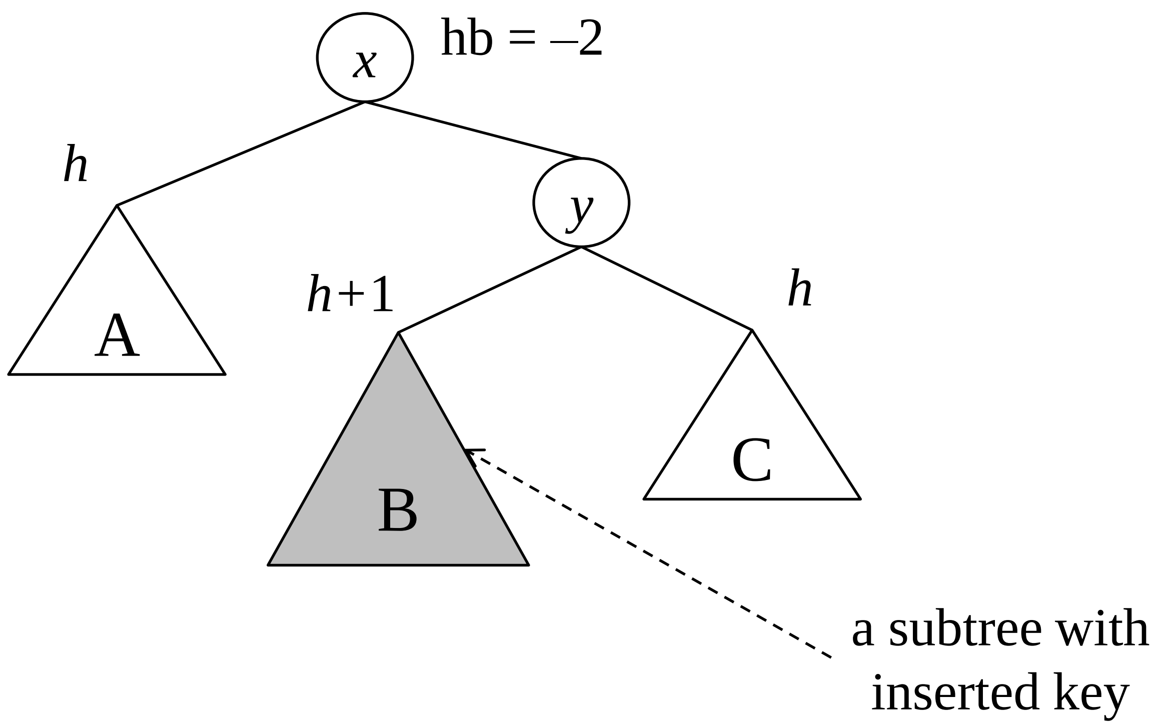
Insertion: Inside Case (iii)

- The AVL property is violated at node *x*
- Solution: do a **double rotation**
  - First do a left rotation at *y*,*z*, and
  - Then do a right rotation at *x*,*z*



Insertion: Inside Case (iv)

- The AVL property is violated at node *x*
- The solution is similar to that for case (iii)



AVL Tree Insertion Algorithm

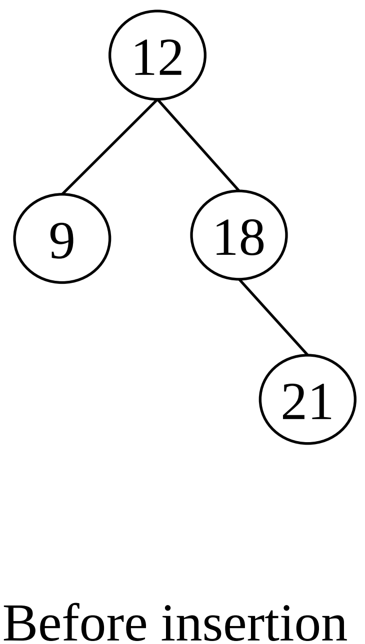
Insert (AVL-Tree *T*, Key *k*)

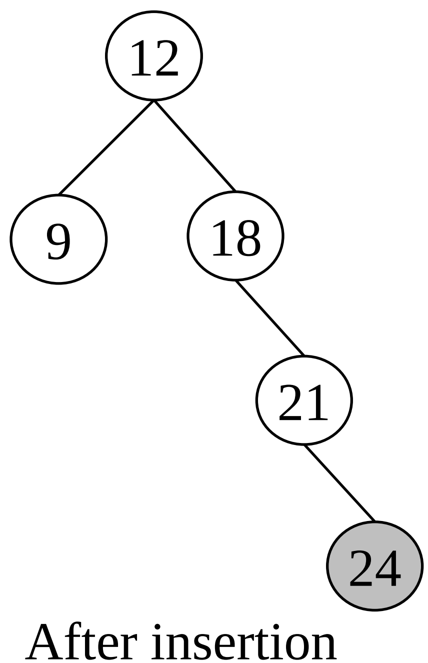
1. insert *k* into a (new) leaf node of *T*
2. let *x* be the leaf node that contains *k*
3. while *x.parent* ≠ `null`
4.   *x* ← *x.parent*   // go up the tree
5.   update *x.hb*   // update height-balance
6.   if *x.hb* = –2 or *x.hb* = 2
7.   decide the case, do rotation at *x*
8.   exit the while-loop

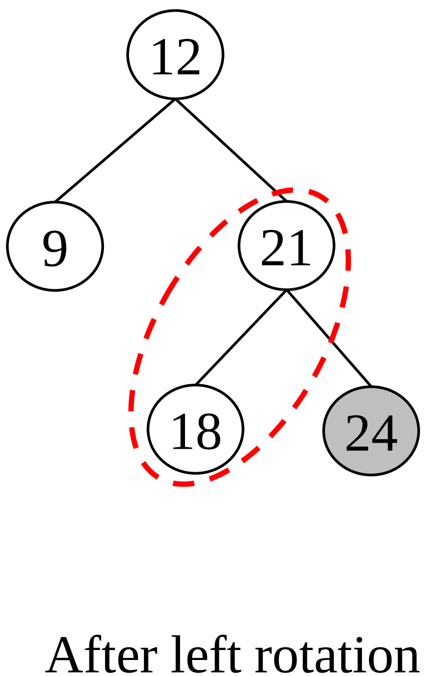
Insertion: Example 1

- Insert ‘24’ into the tree
  - Node ‘18’ has a height-balance –2
  - Do a left rotation at ‘18’, ‘21’

Which rotation case?





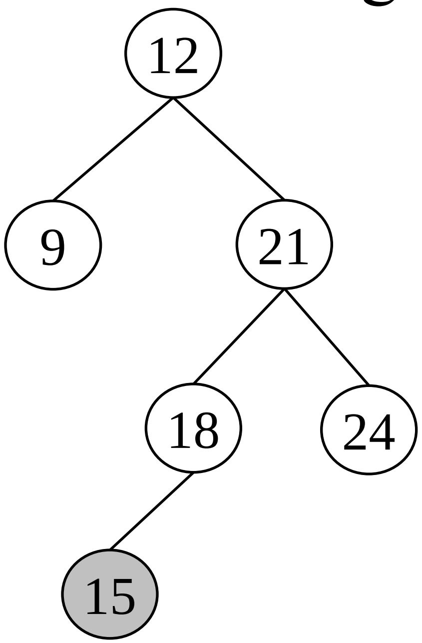


Insertion: Example 2

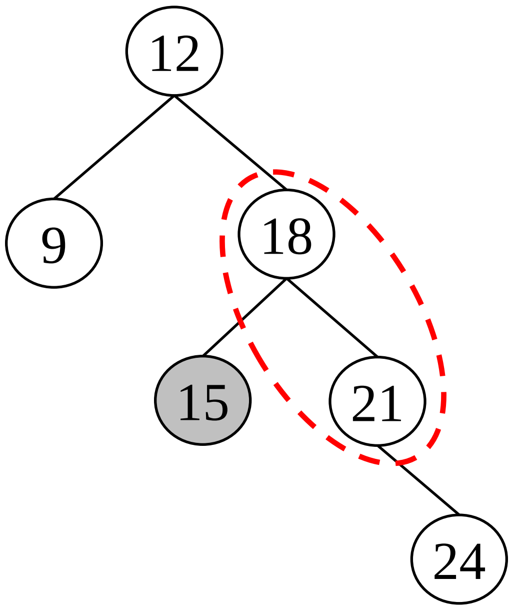
- Insert ‘15’ into the tree
  - Node ‘12’ has a height-balance –2
  - Do a double rotation:

    right rotate ‘21’, ‘18’; left rotate ‘12’, ‘18’

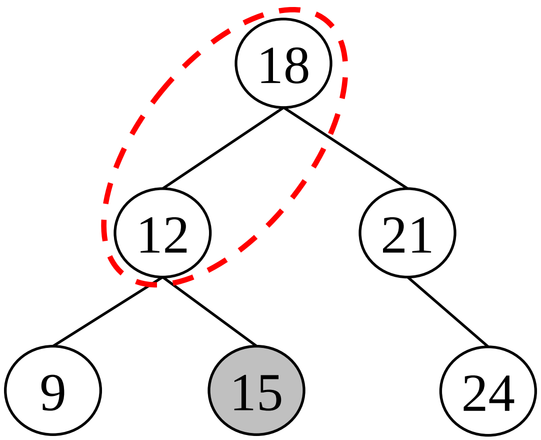
Which rotation case?



After insertion



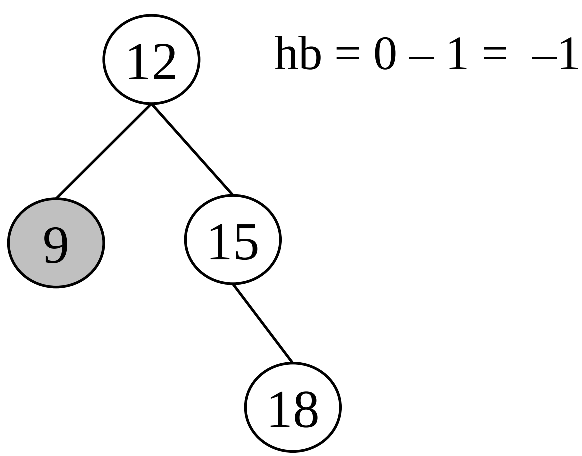
After right rotation



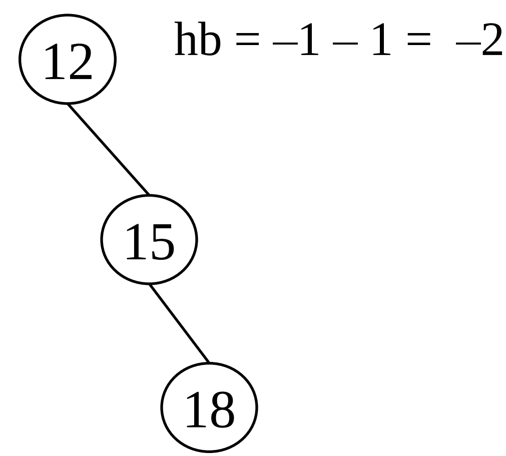
After left rotation

Deletion

- When we delete a key (9) from an AVL tree, some node may have height-balance –2 or 2
  - This violates the AVL tree property
- How to fix this problem?
  - Solution: do rotation (as we learnt before)



Tree before deletion
(AVL yes)



Tree after deletion
(AVL no)

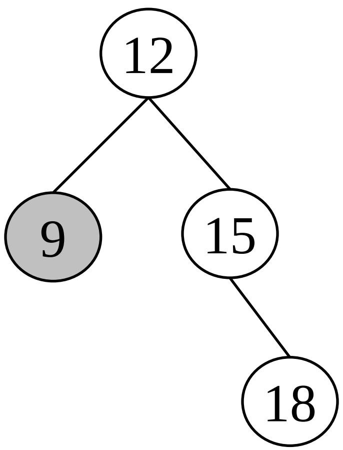
AVL Tree Deletion Algorithm

- We consider three cases when deleting node *x*:
  - 1. *x* has no child
  - 2. *x* has one child
  - 3. *x* has two child
- After deleting a node *x*:
  - Iteratively check the parent/ancestor nodes of *x*
  - Update their height-balance values
  - Do rotations if they have height-balance –2 or 2

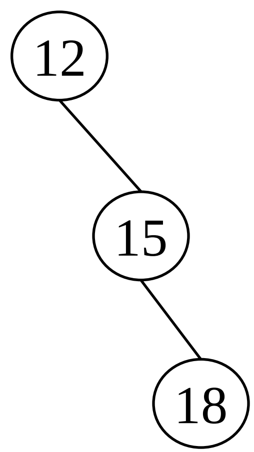
Deletion: Example 1

- Delete ‘9’ from the tree
  - ‘9’ has no children; just delete it
  - Node ‘12’ has a height-balance –2
  - Do a left rotation at ‘12’, ‘15’

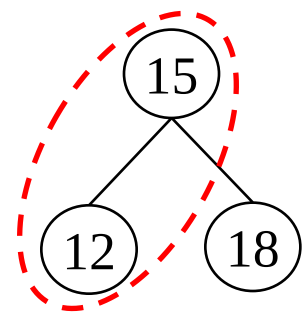
Which rotation case?



Before deletion



After deletion

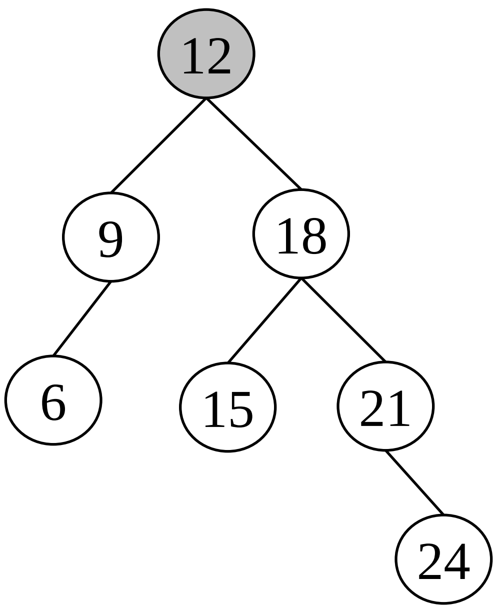


After left rotation

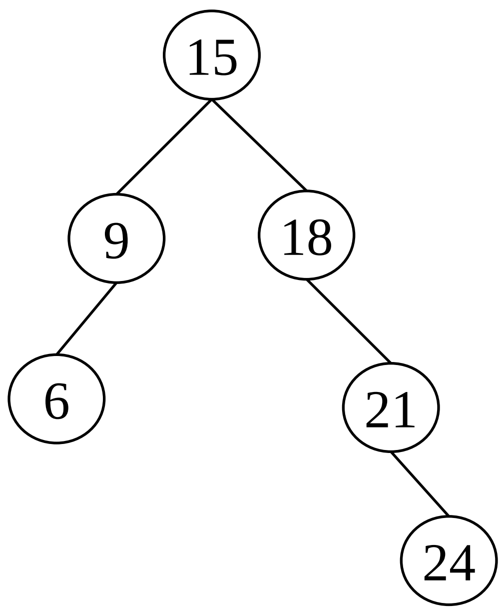
Deletion: Example 2

- Delete ‘12’ from the tree
  - Replace ‘12’ by its successor ‘15’, then delete old ‘15’
  - Node ‘18’ has a height-balance –2
  - Do a left rotation at ‘18’, ‘21’

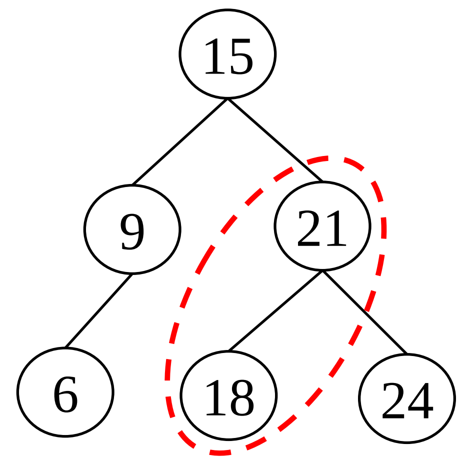
Which rotation case?



Before deletion



After deletion



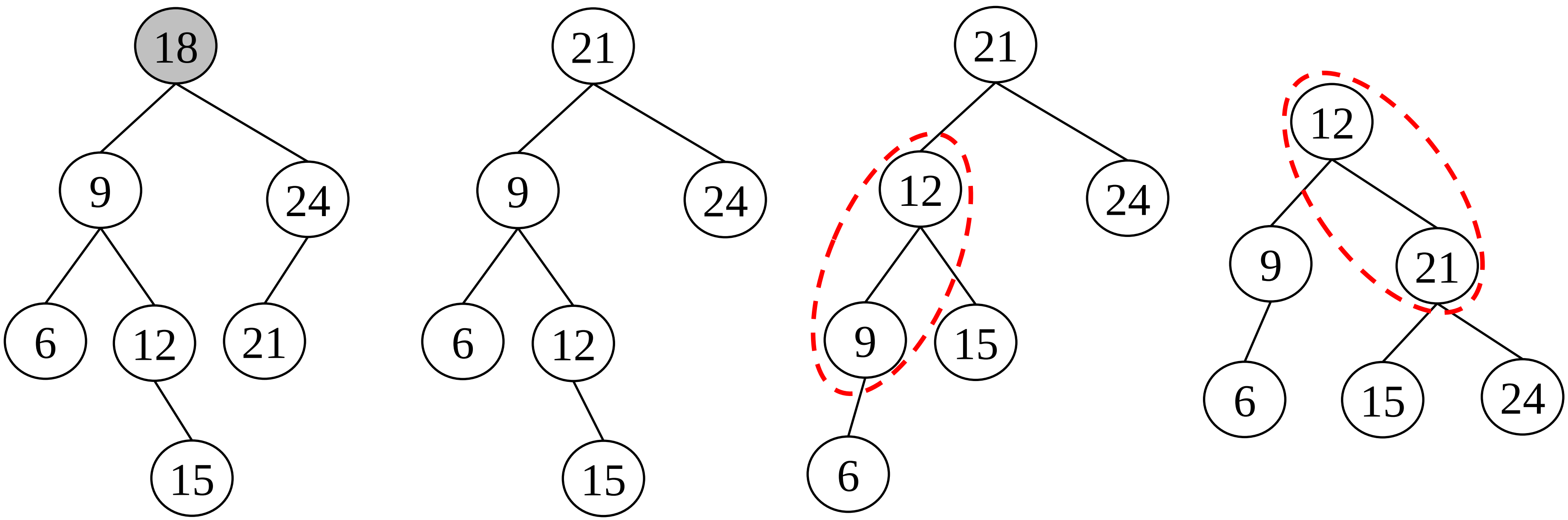
After left rotation

Deletion: Example 3

- Delete ‘18’ from the tree
  - Replace ‘18’ by its successor ‘21’, then delete old ‘18’
  - Node ‘21’ has a height-balance 2
  - Do a double rotation :

Which rotation case?

left rotate ‘9’, ‘12’; right rotate ‘21’, ‘12’

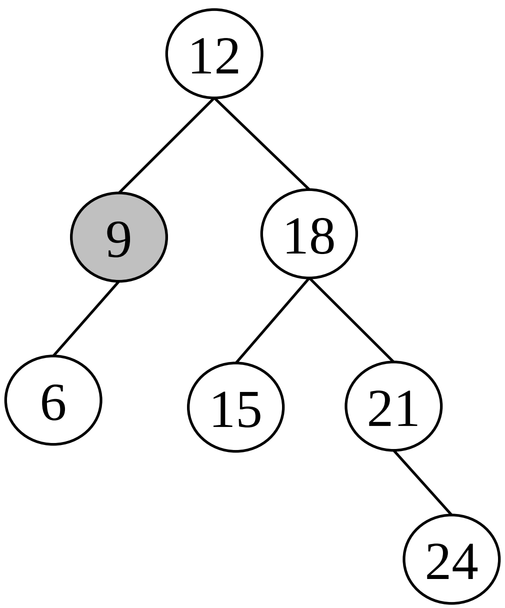


Before deletion After deletion After left rotation After right rotation

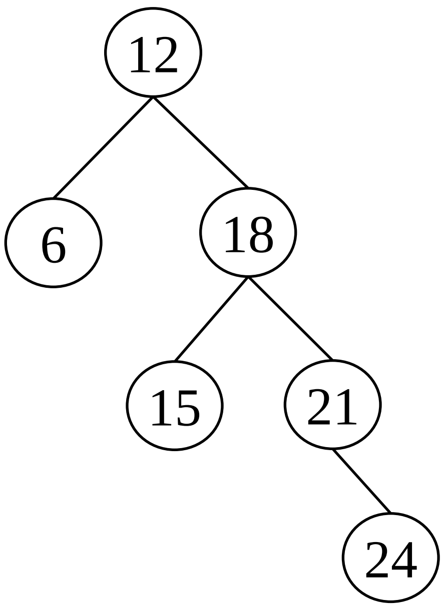
Deletion: Example 4

- Delete '9' from the tree
  - Node '12' has a height-balance –2
  - Do a left rotation at '12', '18'

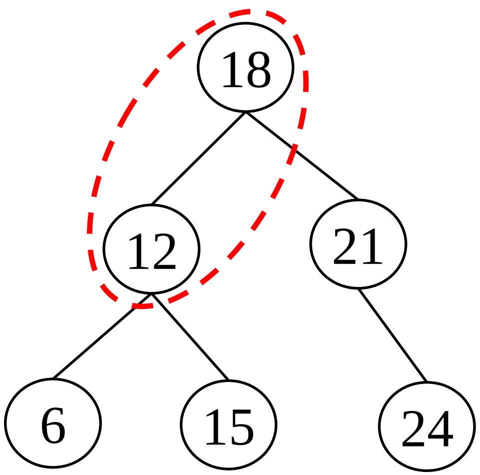
Which rotation case?



Before deletion



After deletion



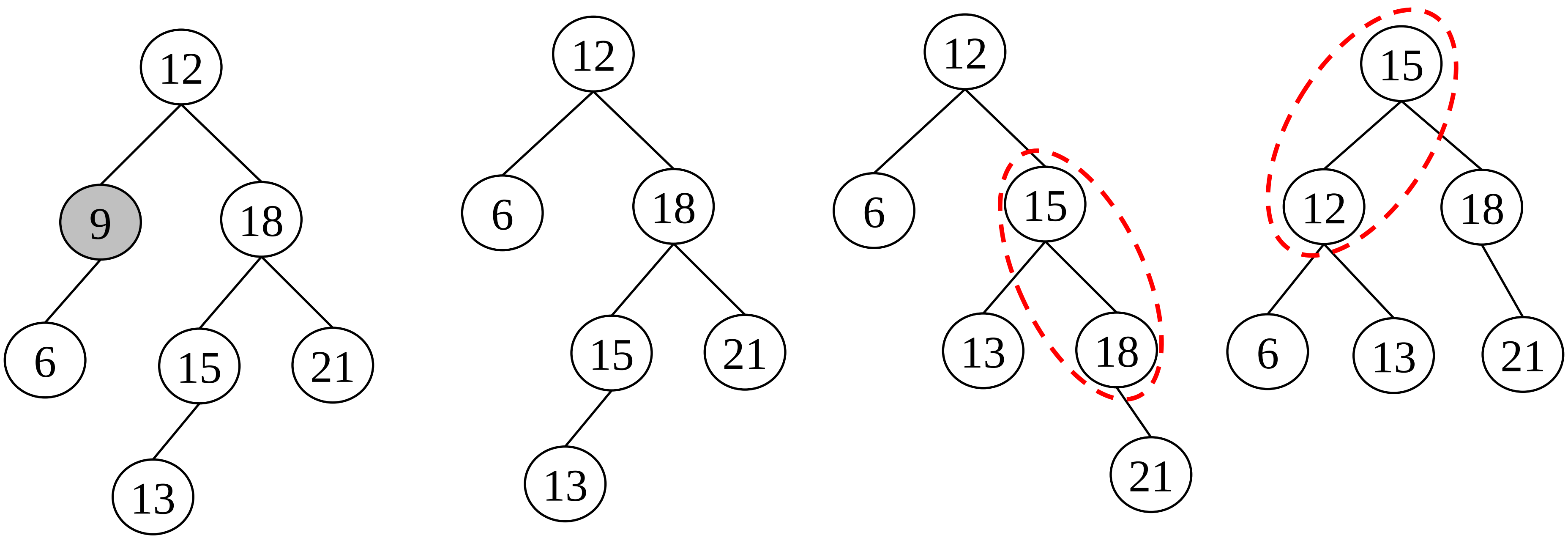
After left rotation

Deletion: Example 5

- Delete ‘9’ from the tree
  - Node ‘12’ has a height-balance –2
  - Do a double rotation:

    right rotate ‘18’, ‘15’; left rotate ‘12’, ‘15’

Which rotation case?



Before deletion After deletion After right rotation After left rotation

Summary

- Binary tree
  - Tree structure
  - Operation: Tree traversal
- Binary search tree
  - Binary search tree property
  - Operations: search, insert, delete
- Balanced tree
  - AVL tree property
  - Insert, delete
    - Apply single/double rotations to balance the tree
- Please read Chapters 8 and 11 in the book "*Data Structures and Algorithms in Java*", 6^{th} Edition