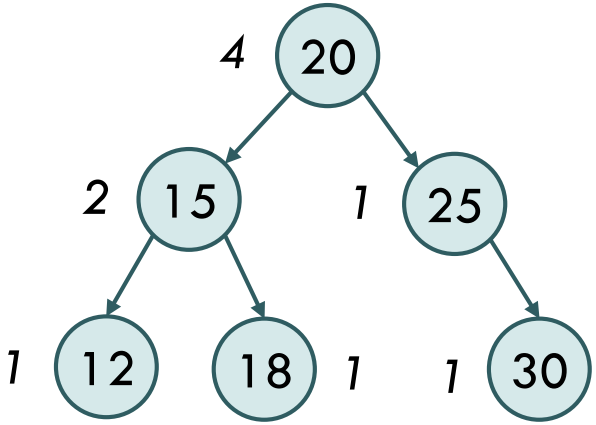
H.O.#16B
BST Supplementary
Fall 2022
Gary Chan

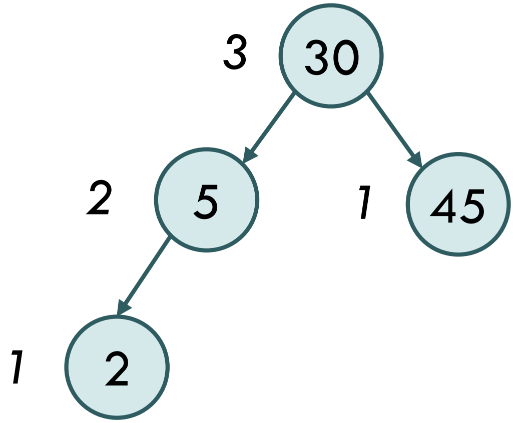
Binary Search Tree and AVL Tree

(N:12)

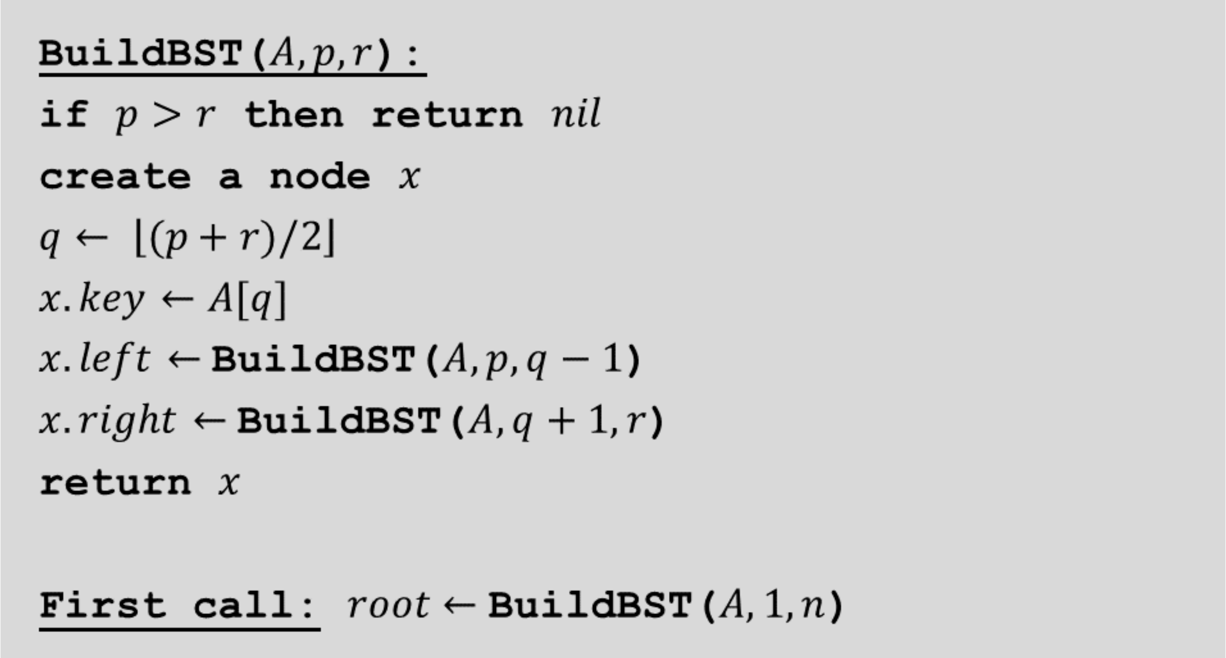
Indexed Binary Search Tree

- Derived from binary search tree by adding another field *LeftSize* to each tree node
- *LeftSize* gives the number of elements in the node's left subtree plus one
- An example (the number inside a node is the element key, while that outside is the value of *LeftSize*)
- It is the rank of the node for the search tree rooted at that node (rank is the position in the sorted order)
  - Can be used to figure out the rank of the node in the tree





Building a BST from a sorted array



```
BuildBST($A,p,r$):
if $p > r$ then return $nil$
create a node $x$
$q \leftarrow \lfloor (p+r)/2 \rfloor$
$x.key \leftarrow A[q]$
$x.left \leftarrow$ BuildBST($A,p,q-1$)
$x.right \leftarrow$ BuildBST($A,q+1,r$)
return $x$

First call: $root \leftarrow$ BuildBST($A,1,n$)
```

- Running time
  - $T(n) = 2T(n/2) + O(1)$
  - $T(n) = O(n)$
- The resulting BST
  - Any search on the tree is exactly the same as doing a binary search on A
  - The height is $O(\log n)$

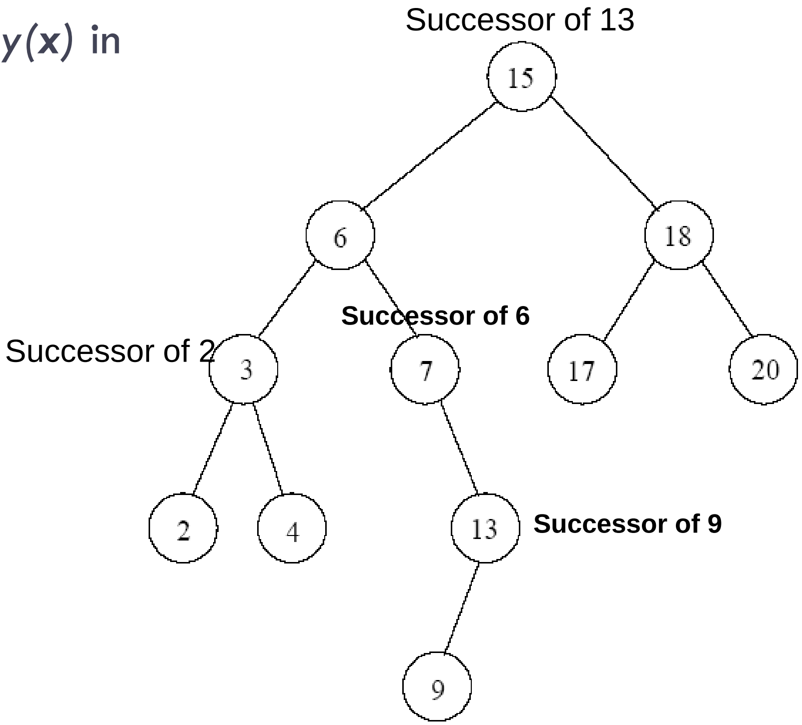
Successor

The successor of a node ***x** is*

defined as:

- The node **y**, whose key(**y**) is the successor of *key(**x**)* in sorted order

  sorted order of this tree. (2,3,4,6,7,9,13,15,17,18,20)

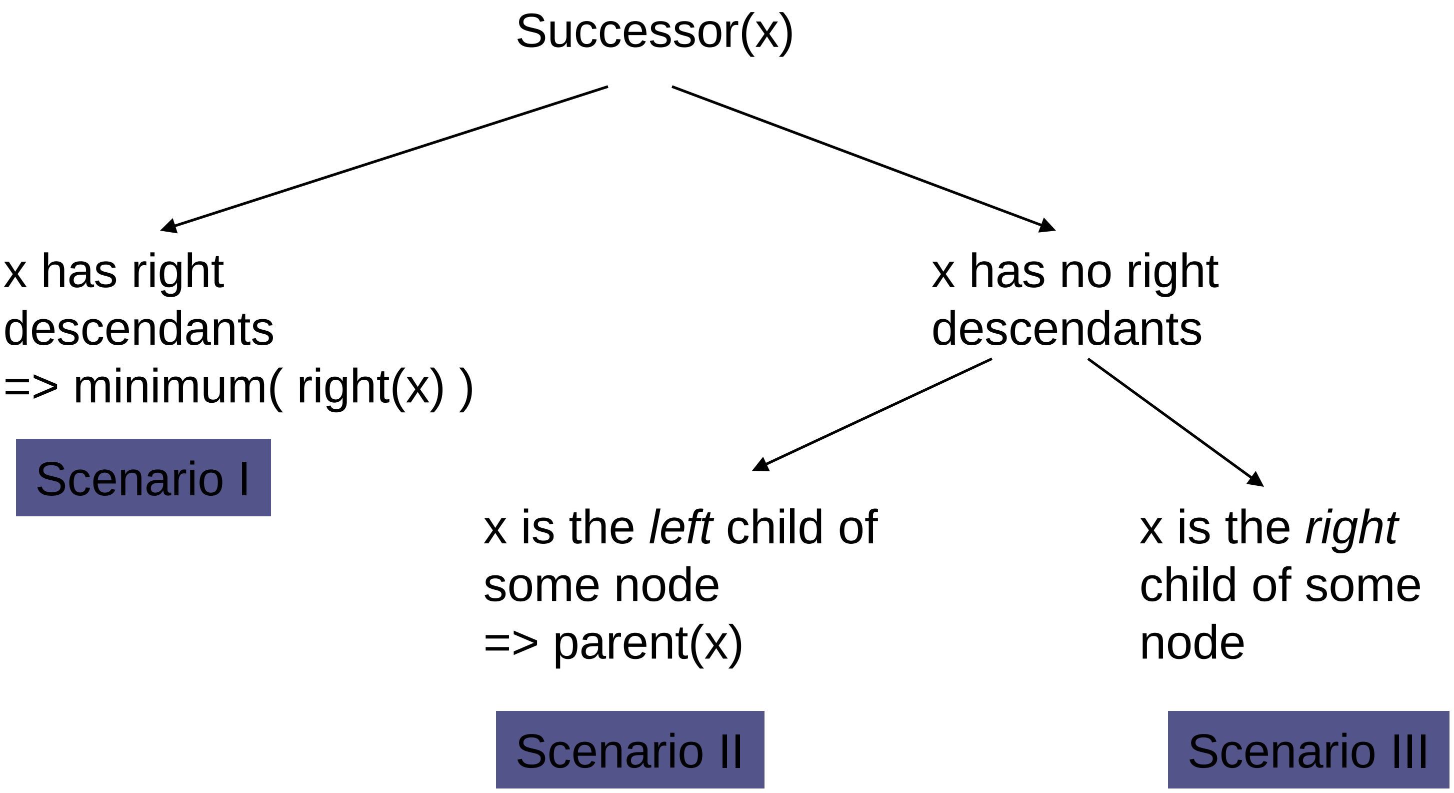


Some examples:

Which node is the successor of 2?
Which node is the successor of 9?
Which node is the successor of 13?
Which node is the successor of 20? Null

Finding Successor:

Three Scenarios to Determine Successor



Successor(x)

x has right descendants
=> minimum(right(x))

Scenario I

x has no right descendants

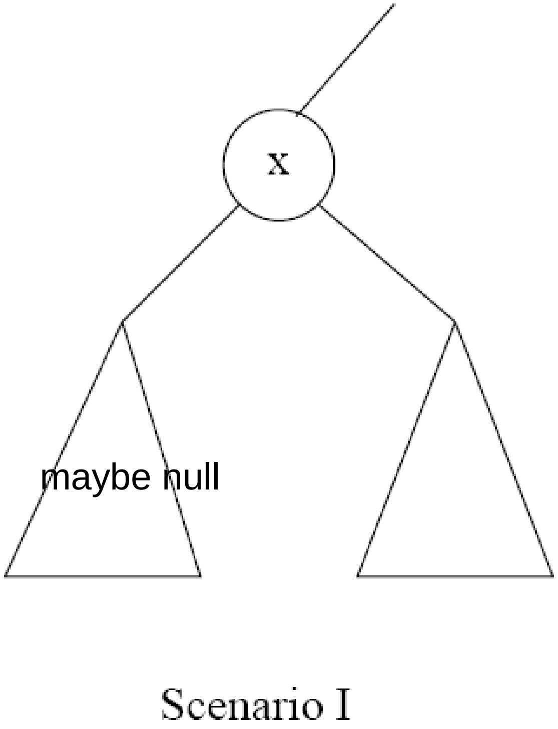
x is the *left* child of some node
=> parent(x)

Scenario II

x is the *right* child of some node

Scenario III

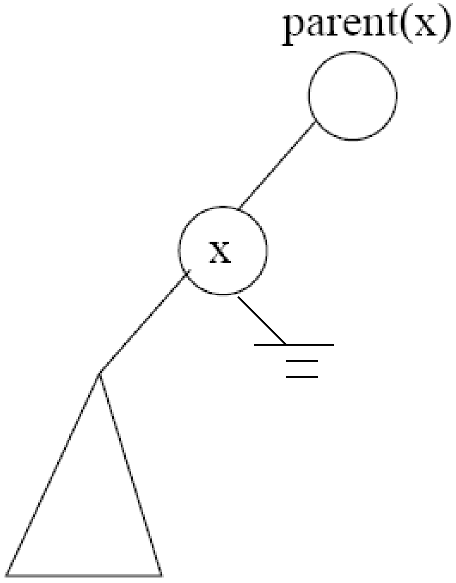
Scenario I: Node x Has a Right Subtree



By definition of BST, all items greater than **x** are in this right sub-tree.

Successor is the minimum(right(**x**))

Scenario II: Node x Has No Right Subtree and x is the Left Child of Parent (x)



Scenario II

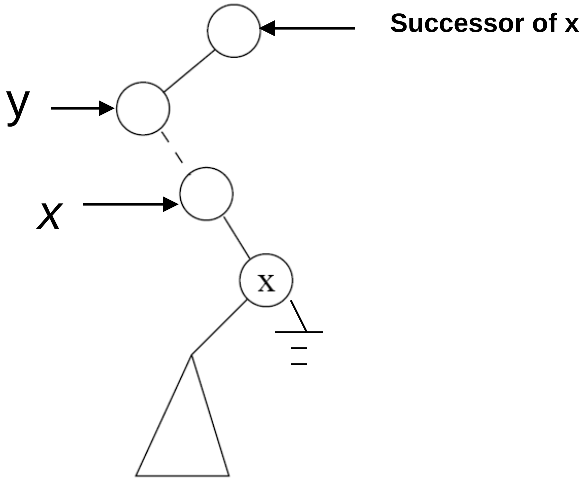
Successor is parent(**x**)

Why? The successor is the node whose key would appear in the next sorted order.

Think about traversal in-order. Who would be the successor of **x**?

The parent of x!

Scenario III: Node x Has No Right Subtree and Is Not a Left-Child of an Immediate Parent



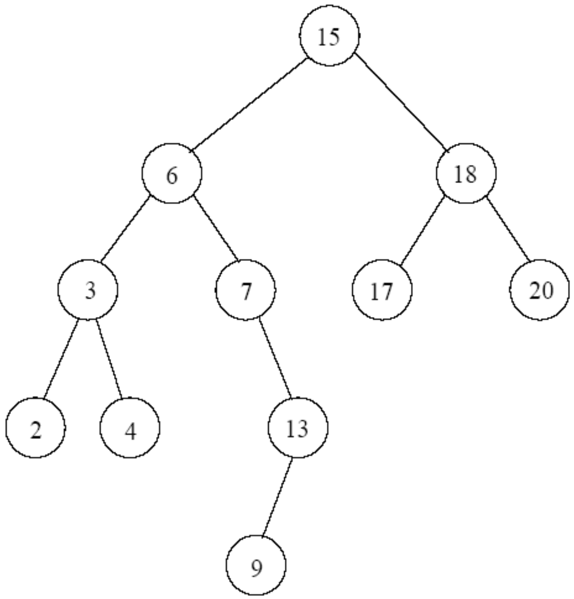
Scenario III

Keep moving up the tree until you find a parent which branches from the left().

Stated in Pseudo code.

```
$y$ := parent($x$);
while $y \neq$ NULL and $x$ = right($y$)
    do $x$ := $y$;
       $y$ := parent($y$);
```

Successor Pseudo-Codes



Verify this code
with this tree.

Find successor of
3 → 4
9 → 13
13 → 15
18 → 20

Note that parent(root) = NULL

```
Algorithm Successor($x$)
Input: $x$ is the input node.
if right($x$) $\neq$ NULL
    then return Minimum(right($x$));          Scenario I
    else
        $y$ := parent($x$);                   Scenario II
        while $y \neq$ NULL and $x$ = right($y$)  }
            do $x$ := $y$;                        } Scenario III
               $y$ := parent($y$);                }
return $y$;
```

Problem

- If we use a “doubly linked” tree, finding parent is easy.

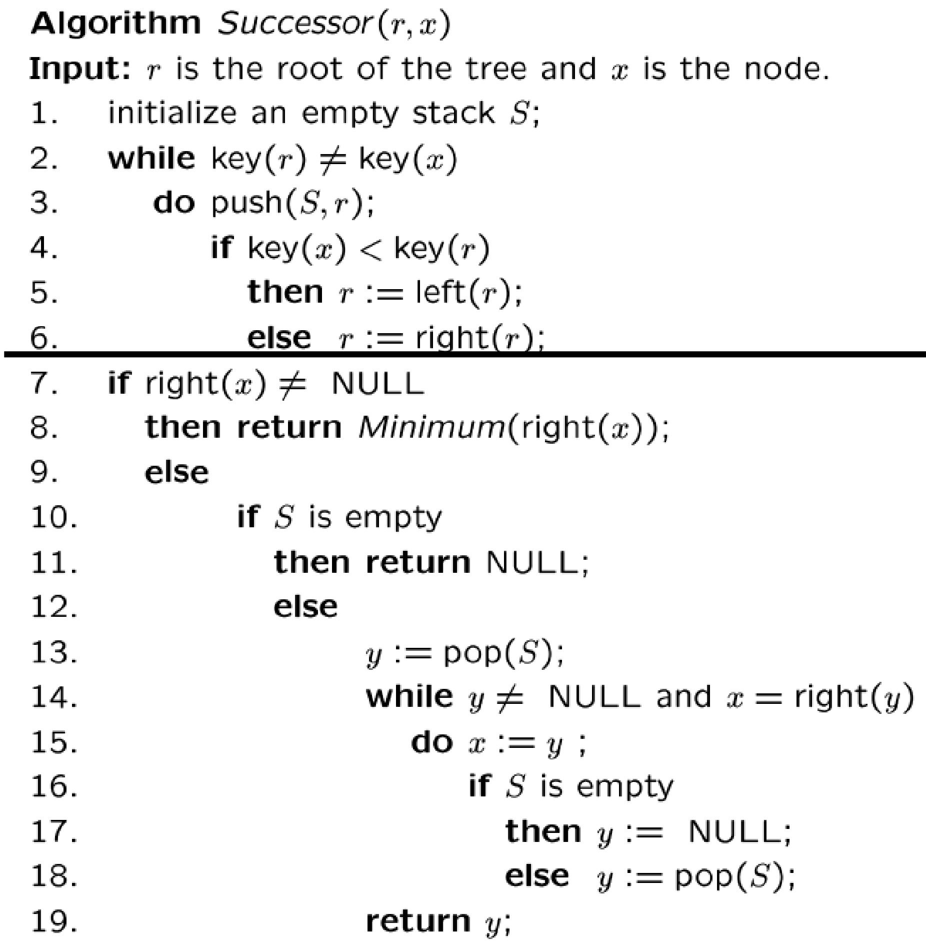
```
class Node
{
                int data;
                Node *left;
                Node *right;
                Node *parent;
};
```

- But usually, we implement the tree using only pointers to the left and right node. ☹ So, finding the parent is tricky.

```
class Node
{
                int data;
                Node *left;
                Node *right;
};
```

For this implementation we need to use a Stack.

Use a Stack to Find Successor



```
Algorithm Successor$(r, x)$
Input: $r$ is the root of the tree and $x$ is the node.
initialize an empty stack $S$;
while key$(r) \neq$ key$(x)$
    do push$(S, r)$;
        if key$(x) <$ key$(r)$
            then $r :=$ left$(r)$;
            else $r :=$ right$(r)$;
if right$(x) \neq$ NULL
    then return Minimum(right$(x)$);
    else
        if $S$ is empty
            then return NULL;
            else
                $y :=$ pop$(S)$;
                while $y \neq$ NULL and $x =$ right$(y)$
                    do $x := y$ ;
                        if $S$ is empty
                            then $y :=$ NULL;
                            else $y :=$ pop$(S)$;
                return $y$;
```

PART I

Initialize an empty Stack s.

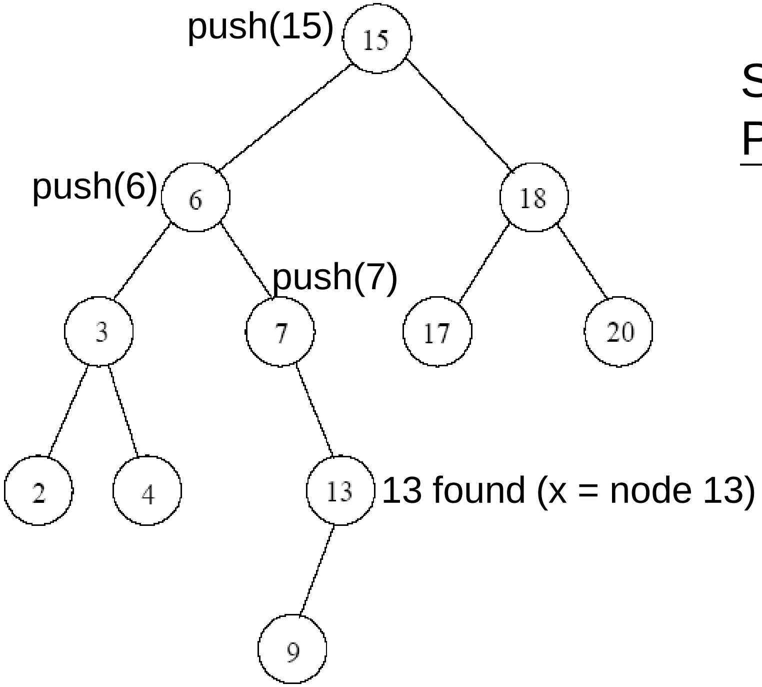
Start at the root node, and traverse the tree until we find the node **x**. Push all visited nodes onto the stack.

PART II

Once node **x** is found, find successor using 3 scenarios mentioned before.

Parent nodes are found by popping the stack!

An Example



push(15)

push(6)

push(7)

13 found (x = node 13)

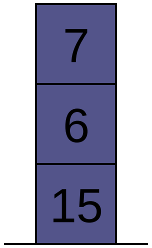
Successor(root, 13)

Part I

Traverse tree from root to find 13

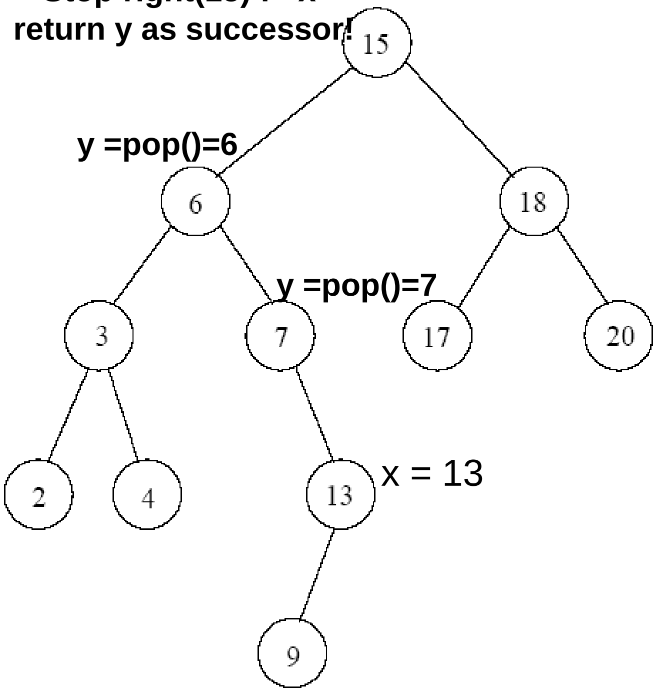
order -> 15, 6, 7, 13

```
Algorithm $Successor(r, x)$
Input: $r$ is the root of the tree and $x$ is the node.
initialize an empty stack $S$;
while key($r$) $\neq$ key($x$)
    do push($S, r$);
        if key($x$) $<$ key($r$)
            then $r :=$ left($r$);
            else $r :=$ right($r$);
```



Stack s

Example

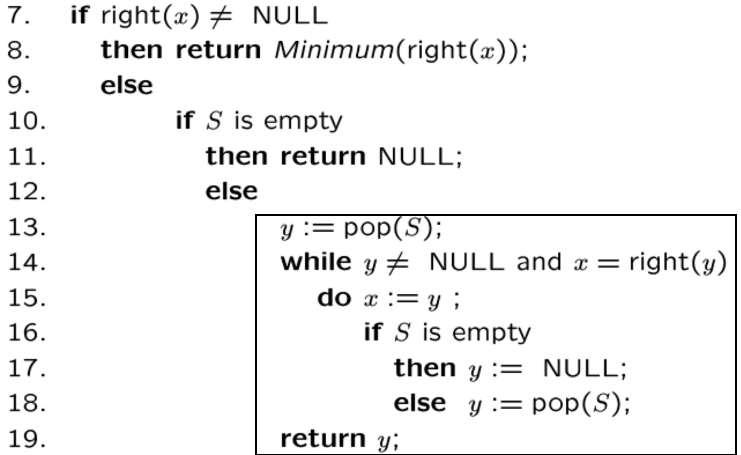


y =pop()=15
->Stop right(15) != x
return y as successor!

y =pop()=6

y =pop()=7

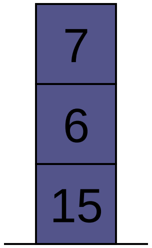
x = 13



```
if right($x$) $\neq$ NULL
   then return Minimum(right($x$));
   else
         if $S$ is empty
            then return NULL;
            else
               $y$ := pop($S$);
               while $y \neq$ NULL and $x$ = right($y$)
                  do $x$ := $y$ ;
                     if $S$ is empty
                        then $y$ := NULL;
                        else $y$ := pop($S$);
               return $y$;
```

Successor(root, 13)
Part II
Find Parent (Scenario III)

```
y=s.pop()
while y!=NULL and x=right(y)
  x = y;
  if s.isempty()
     y=NULL
  else
     y=s.pop()
  loop
return y
```



Stack s

Successor

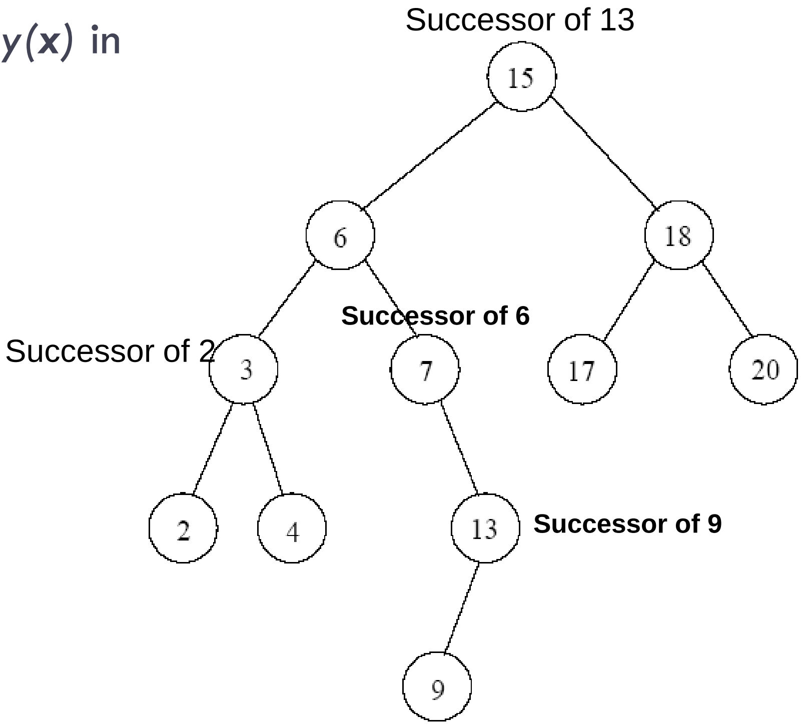
The successor of a node ***x*** *is* defined as:

- The node **y**, whose key(**y**) is the successor of *key(**x**)* in sorted order

  sorted order of this tree. (2,3,4,6,7,9,13,15,17,18,20)

Some examples:

Which node is the successor of 2?
Which node is the successor of 9?
Which node is the successor of 13?
Which node is the successor of 20? Null



Two Other Approaches for Case III

- Method 1 (if we know the distance between two keys beyond simply <, > and ==): Along the way to find x, keep the pointer to the node which has the lowest value higher than x.
- Method 2: Observe that:
  - To get to x from left(y), we always traverse right, i.e., the value is increasing beyond left(y). The value never exceeds y, as x is on the left of y.
  - If we plot the values from y to x against the nodes visited, it is hence of a "V" shape, starting from y, dropping to some low value, and then increasing gradually to x, a value just below y
  - Using stack storing the path from the root to x, we hence can detect the right turn in the reverse path simply as follows:
    - Keep popping the stack until the key is higher than the value x. This must be its successor.

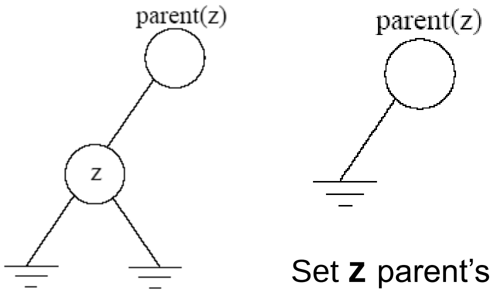
```
while (!s.empty()){
  y = s.pop(); // assume pop() returns the top value of the stack
  if( y > x)
    return y; // the successor
}
return NULL;  // empty stack; successor not found
```

BST Deletion: Delete Node z from Tree

Three cases for deletion

Case I

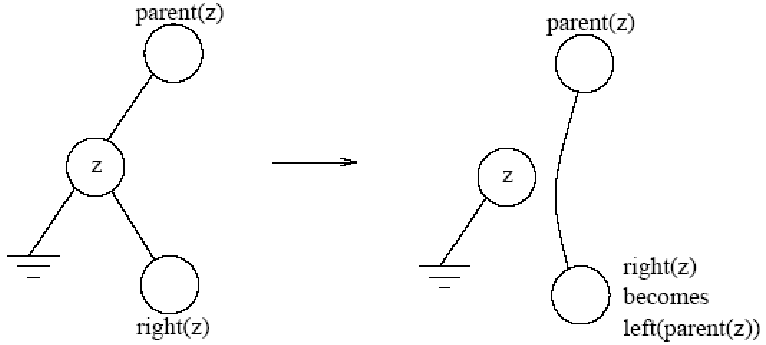
Node **z** is a leaf



Set **z** parent's pointer to **z** to NULL

Case II

Node **z** has exactly 1 (left or right) child

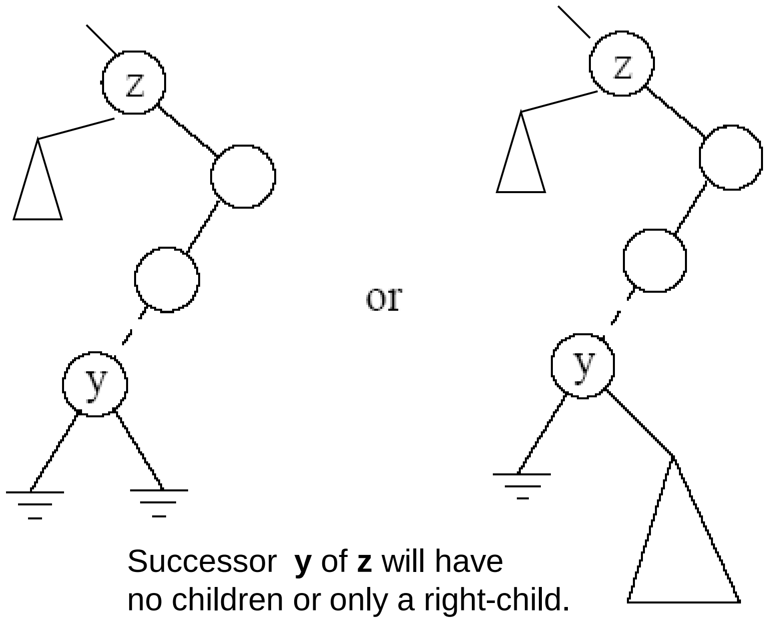


Modify appropriate parent(**z**) to point to **z**'s child (Parent adoption)

Case III: Node z Has 2 Children

Step 1.
Find successor y of 'z' (i.e. y = successor(z))

Since z has 2 children, successor is **y=minimum(right(z))**

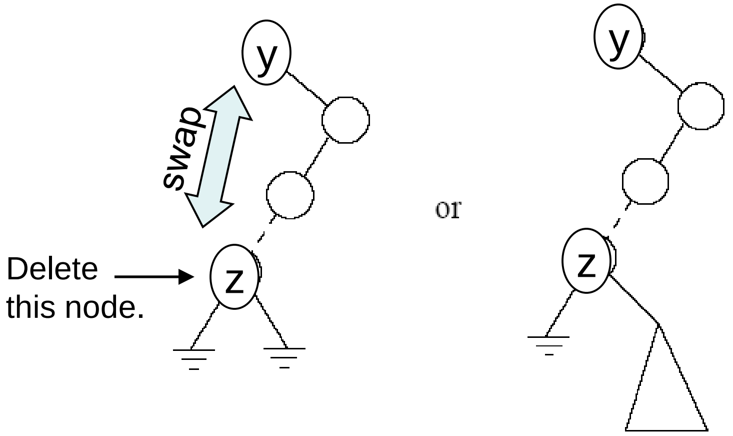


Successor **y** of **z** will have
no children or only a right-child.

Why? Look at the definition of
minimum(..)

Step 2.
Swap keys of z and y.

Now delete node y (which now has value z)!
This *deletion* is either case I or II.



Delete
this node.

Case I

Case II

(deletion of node "z" is
always going to be Case I or II)

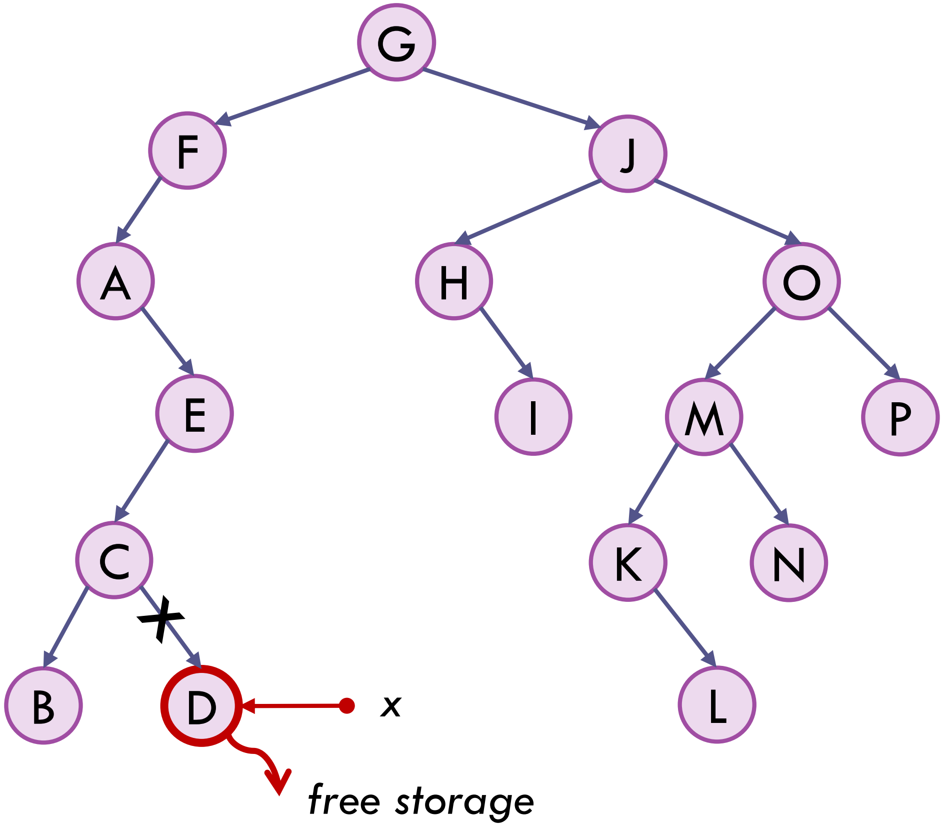
Special Case: Deleting the Root with 1 Child Descendant

- Move the root to the child

A Deletion Example

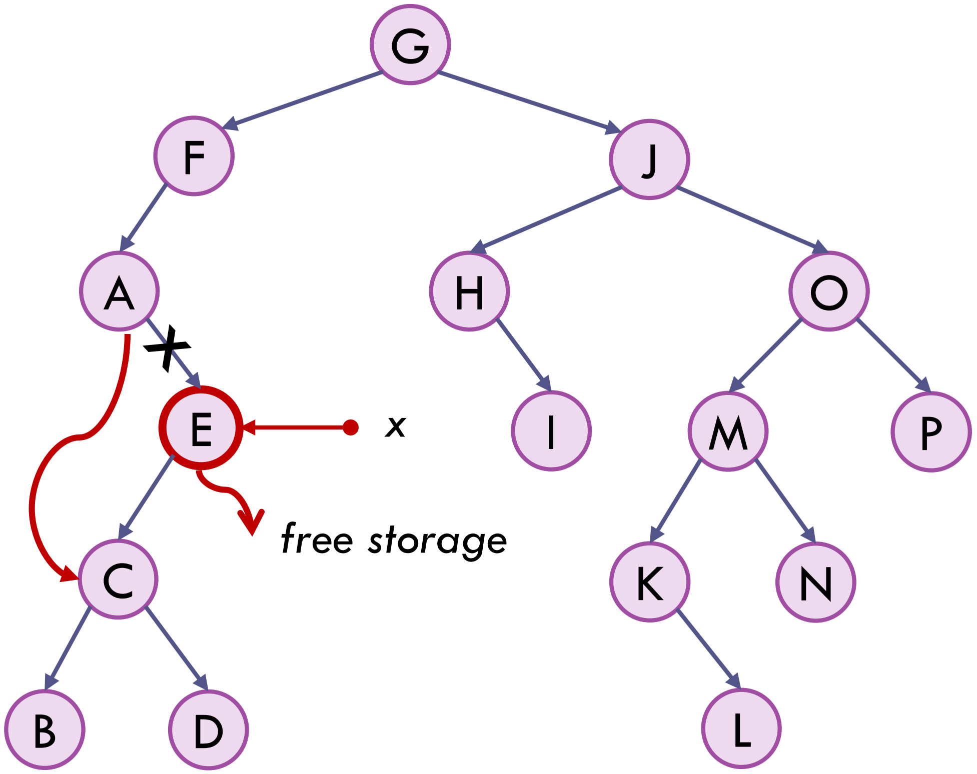
Three possible cases to delete a node x from a BST

1. The node x is a leaf



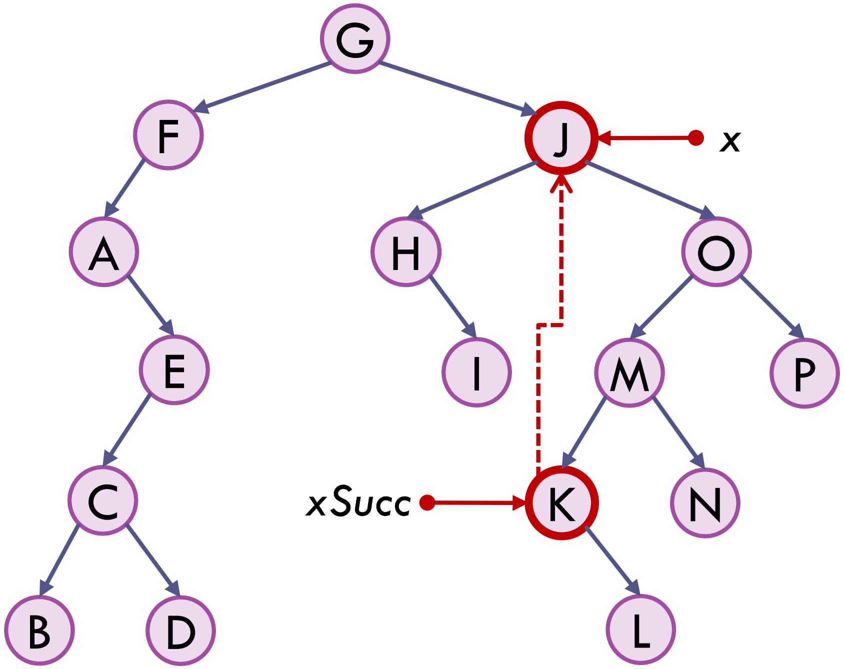
A Deletion Example (Cont.)

2. The node x has one child

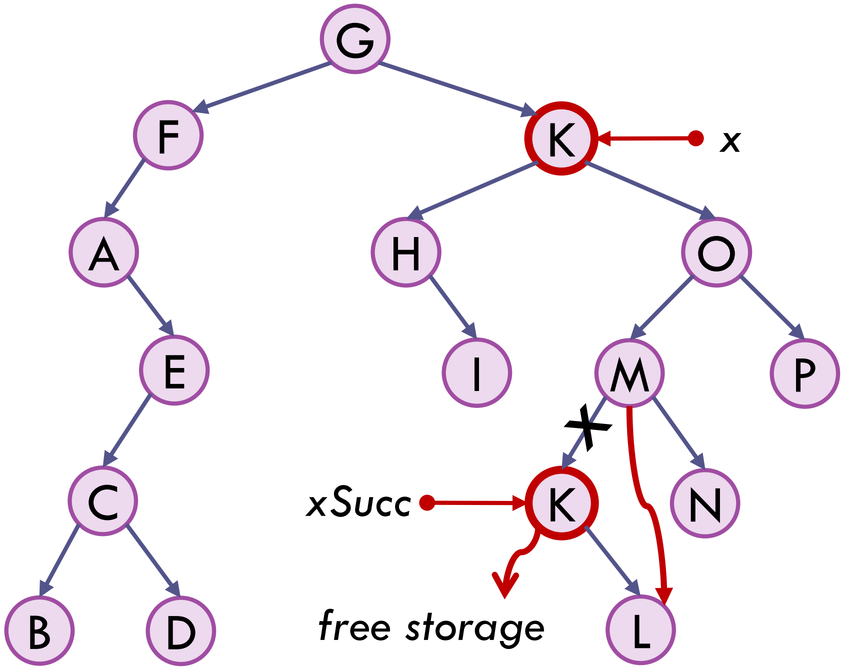


A Deletion Example (Cont.)

3. *x* has two children



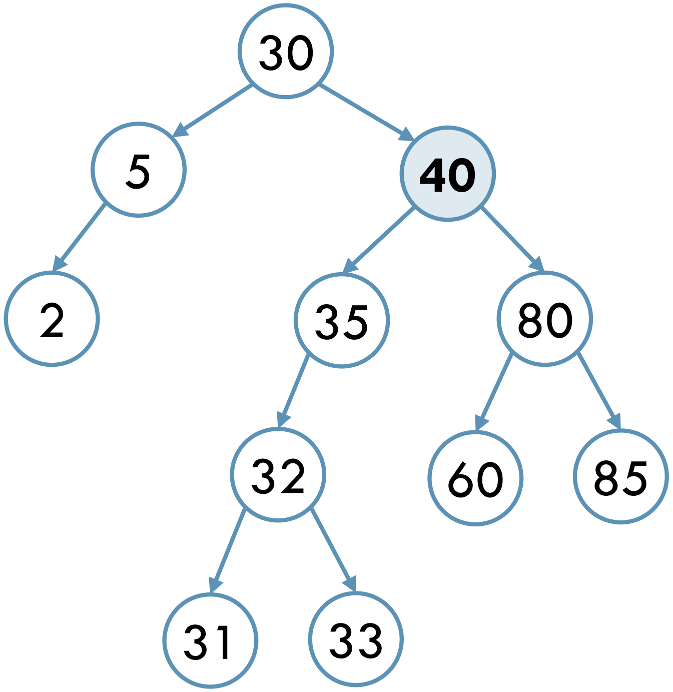
i) Replace contents of *x* with inorder successor (smallest value in the right subtree)



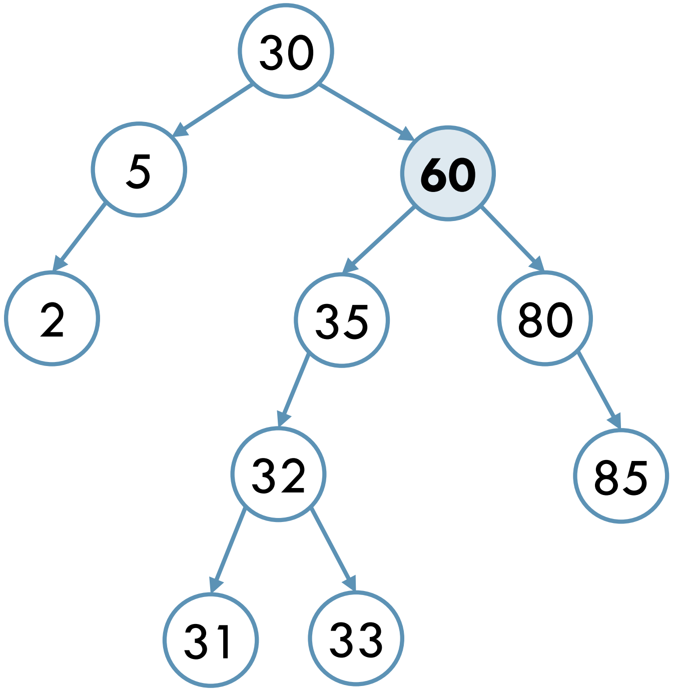
ii) Delete node pointed to by *xSucc* as described for cases 1 and 2

Another Deletion Example

- Removing 40 from (a) results in (b) using the smallest element in the right subtree (i.e., the successor)



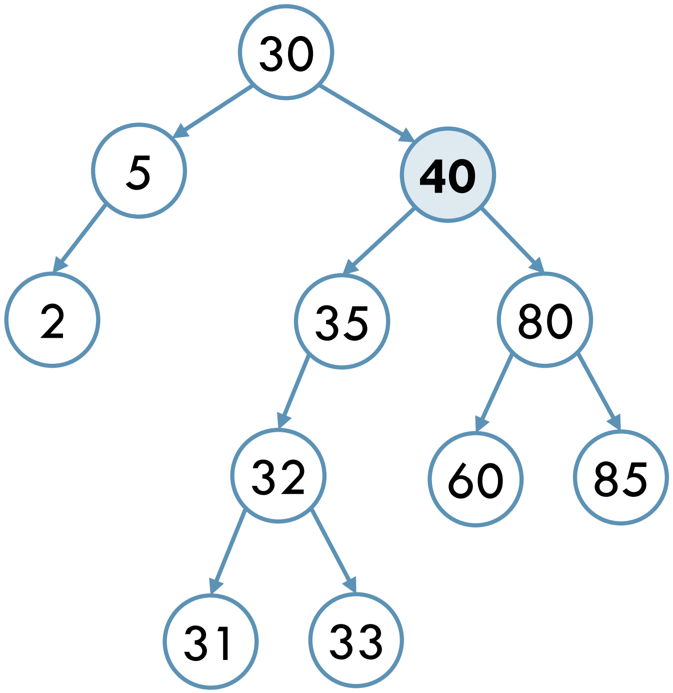
(a)



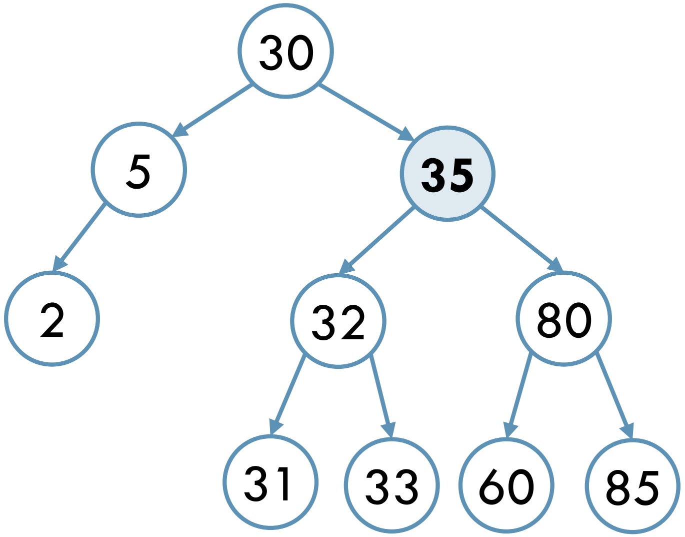
(b)

Another Deletion Example (Cont.)

- Removing 40 from (a) results in (c) using the largest element in the left subtree (i.e., the predecessor)



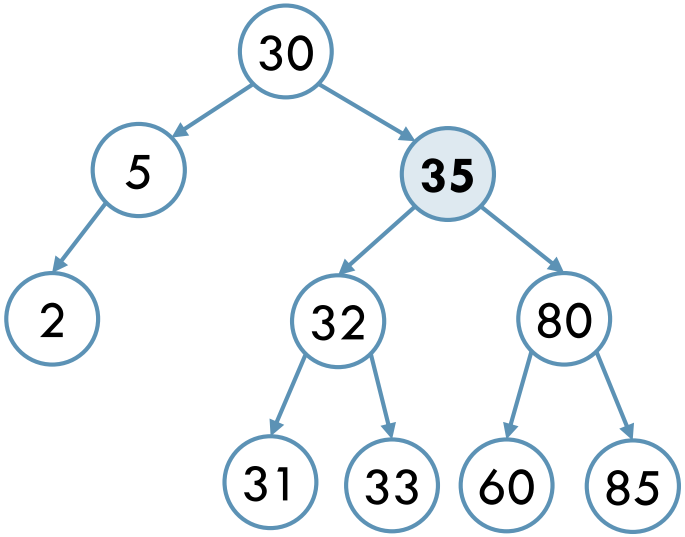
(a)



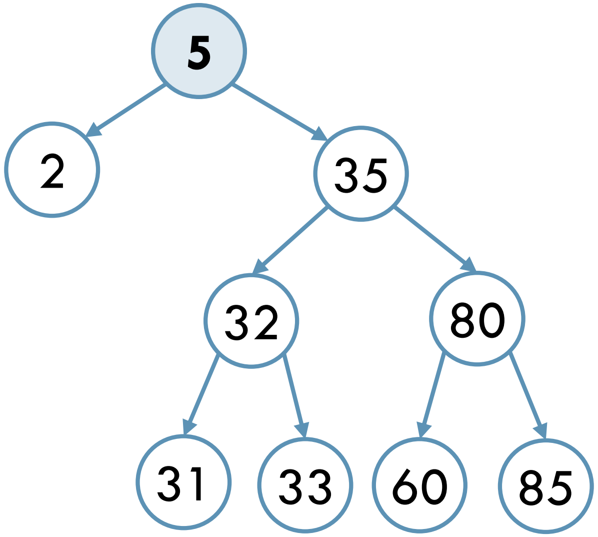
(c)

Another Deletion Example (Cont.)

- Removing 30 from (c), we may replace the element with either 5 (predecessor) or 31 (successor). If we choose 5, then (d) results.



(c)



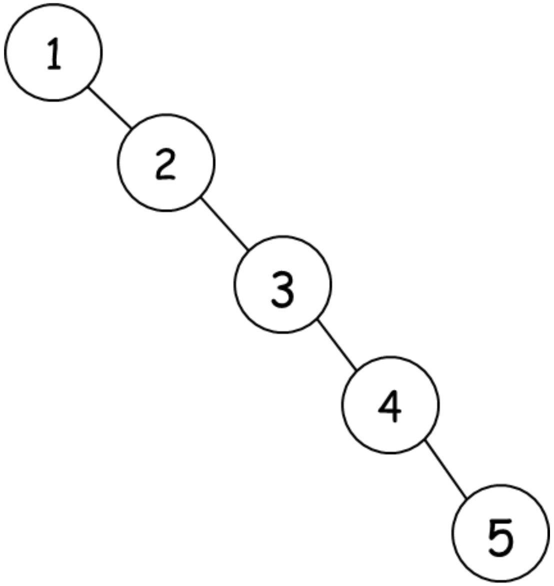
(d)

Time Complexity of Binary Search Trees

Find(x)	O(height of tree)
Min(x)	O(height of tree)
Max(x)	O(height of tree)
Insert(x)	O(height of tree)
Delete(x)	O(height of tree)
Traverse	O(N)

Problem of Lopsidedness

- Trees can be totally lopsided
- If we insert all elements in sorted order… each node would have a right child only
  - Degenerates into a linked list
- Need a way to restore "balance" so that the height is always $O(\log n)$.



Processing time affected by "shape" of tree

Tree Examples

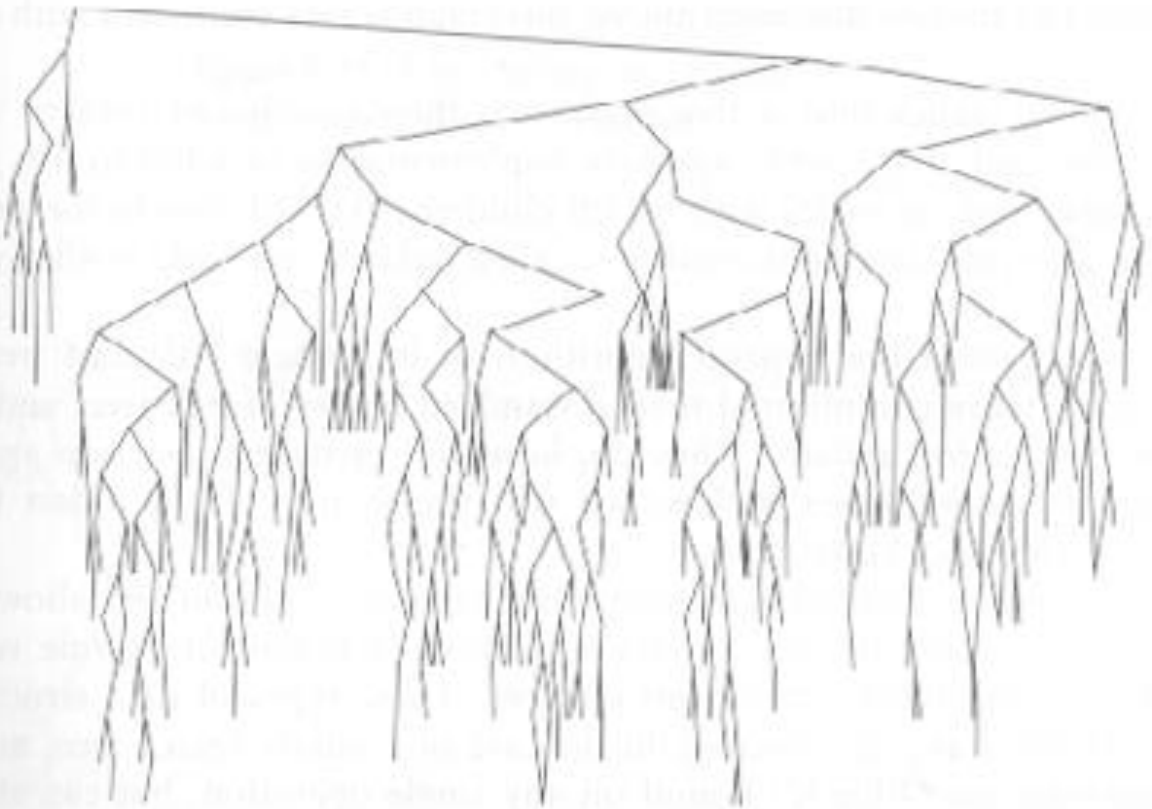
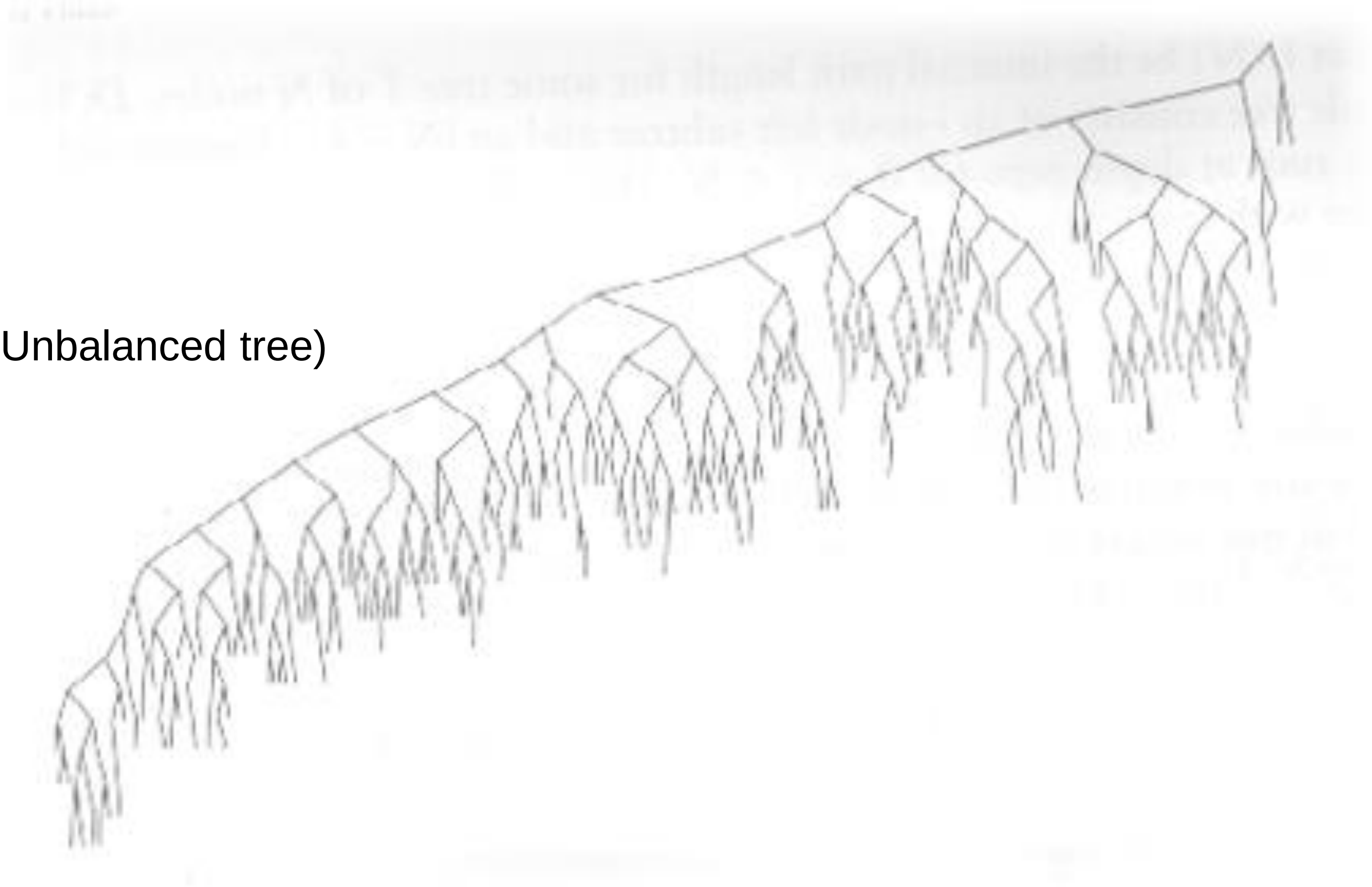


Figure 4.29 A randomly generated binary search tree

(Tree resulting from randomly generated input)

Tree Examples

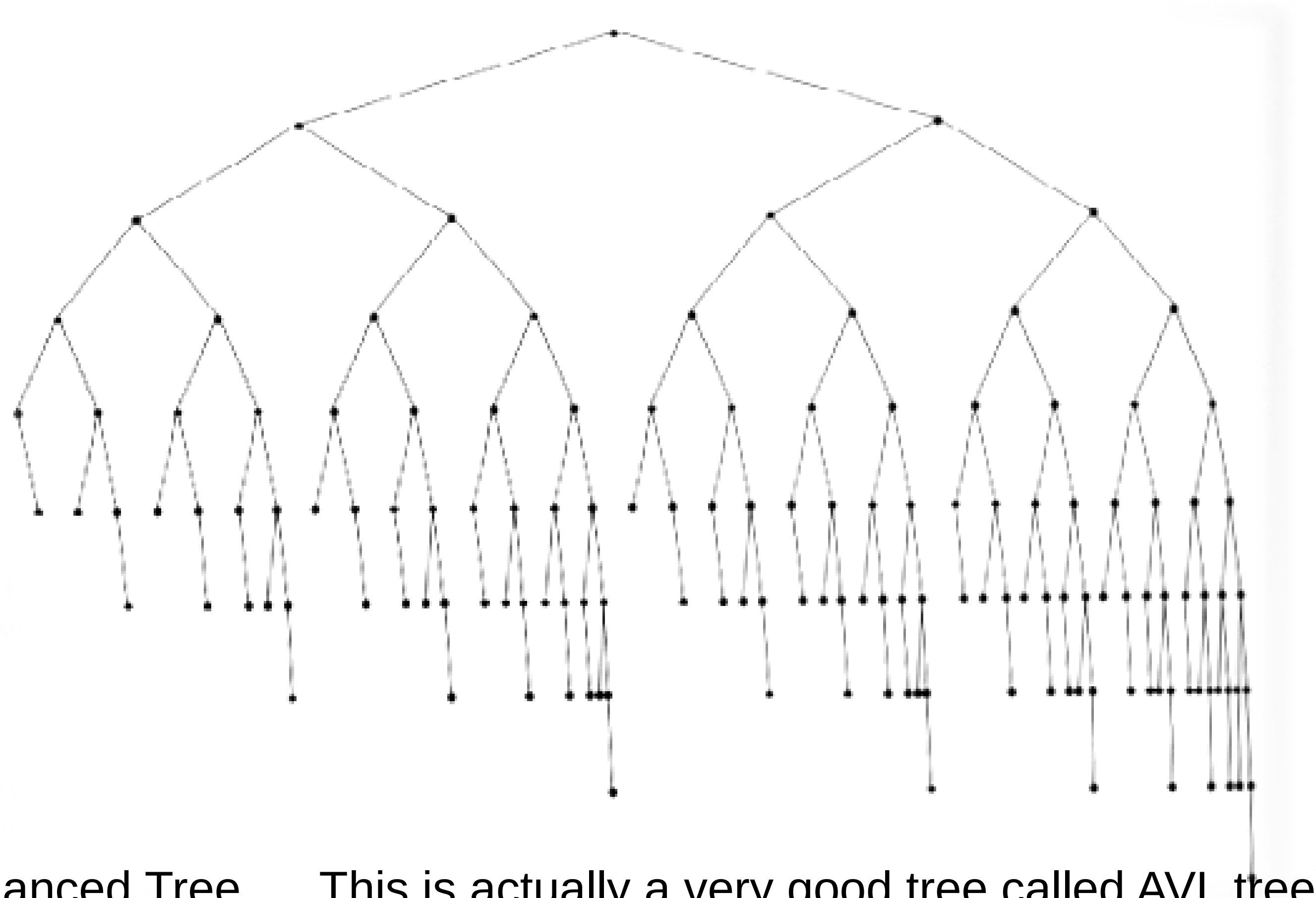


(Unbalanced tree)

How Fast is Sorting Using BST?

- n numbers (n large) are to be sorted by first constructing a binary tree and then read them in inorder manner
- Bad case: the input is more or less sorted
  - A rather "linear" tree is constructed
  - Total steps in constructing a binary tree: $1 + 2 + \ldots + n = n(n+1)/2 \sim n^2$
  - Total steps in traversing the tree: n
  - Total $\sim n^2$
- Best case: the binary tree is constructed in a balanced manner
  - Depth after adding i numbers: log(i)
  - Total steps in constructing a binary tree: $\log 1 + \log 2 + \log 3 + \log 4 + \ldots + \log n < \log n + \log n + \ldots + \log n = n \log n$
  - Total steps in traversing the tree: n
  - Total $\sim n \log n$, much faster
- It turns out that one cannot sort n numbers faster than nlog n
- For any arbitrary input, one can indeed construct a rather balanced binary tree with some extra steps in insertion and deletion
  - E.g., An AVL tree (two Soviet inventors, G. M. Adelson-Velskii and E. M. Landis, 1962)

An AVL Tree → A Rather Balanced Tree for Efficient BST Operations (See Animation)



(Balanced Tree . . This is actually a very good tree called AVL tree)

AVL Trees

(Balanced Trees)

The name comes from the inventors:
Adelson-Velskii and Landis Trees

AVL Tree Performance

- The height with *n* nodes is O(*log n)*
- For every value of *n*, there exists an AVL tree
- An n-element AVL search can be done in O(*log n*) time
- Insertion takes O(*log n*) time
- Deletion takes O(*log n*) time

Idea of AVL

- Make the binary tree a balanced tree
- What affects the tree's shape?
  - Insertion()
  - Deletion()
- So, let's modify the way we do insertions and deletions
- We need some definitions first . .

Height of a node

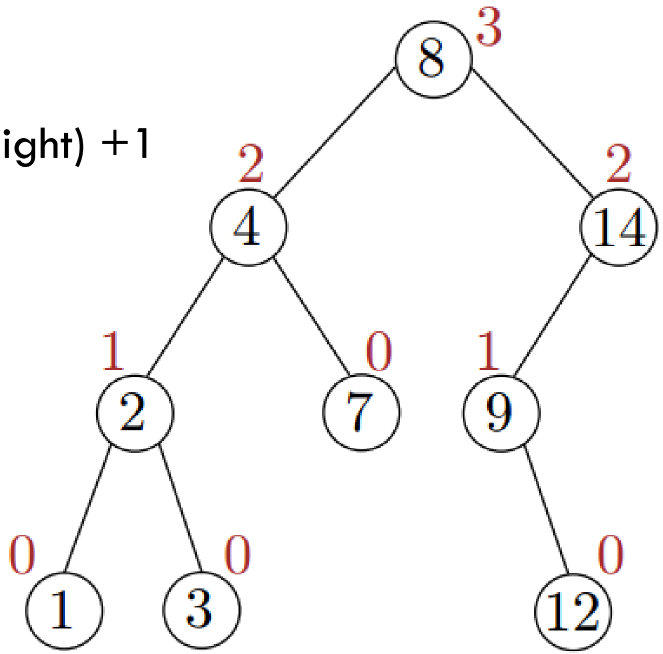
The height of a node in a tree is the number of edges on the longest downward path from the node to a leaf

Node height = max(left child height, right child height) +1

Leaves: height = 0

Tree height = root height

Empty tree: height = −1

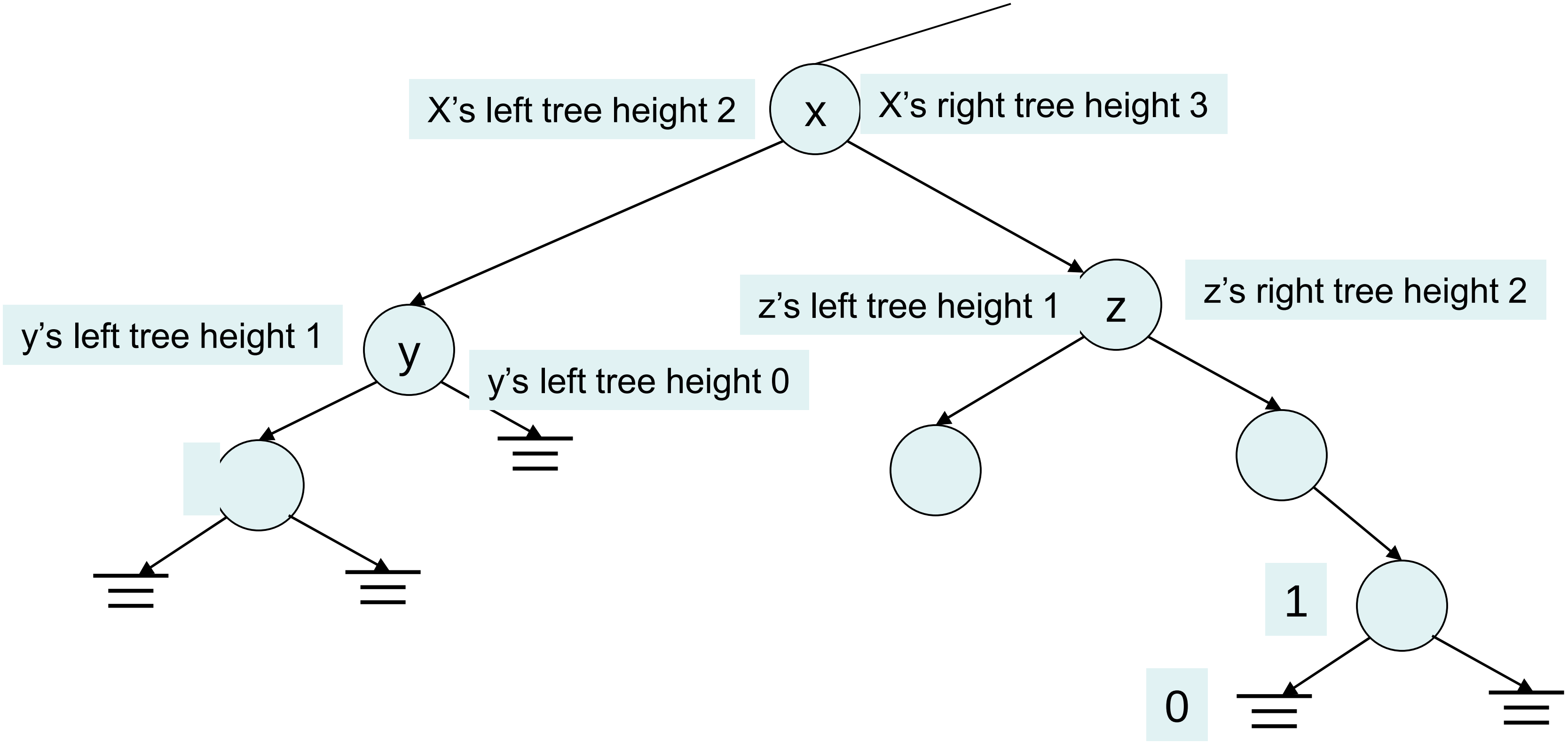


An Alternative Definition on the Height of a Node

- Height of a leaf is 1
- Height of a null pointer is 0
- The height of an internal node is the maximum height of its children plus 1

- Slightly different than our previous definition of height, which counted the number of edges

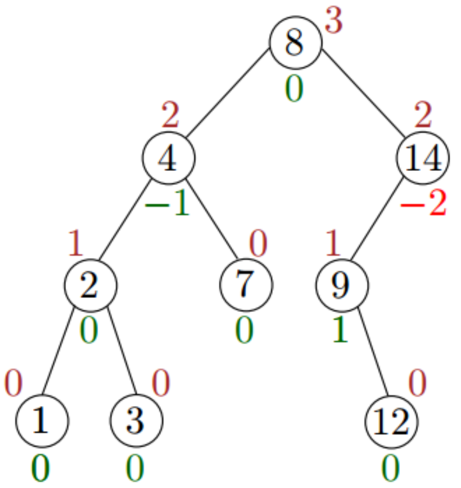
Height of Node example



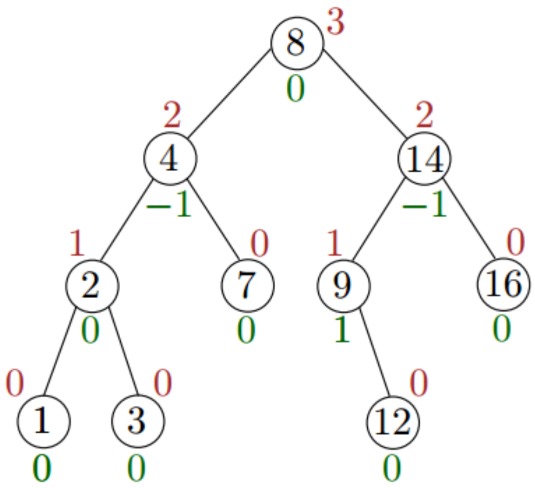
What is the height of x? Maximum child height + 1, so height(x) = 4.

Balanced Binary Search Tree: AVL Tree

- An AVL-tree is a binary search tree in which for every node in the tree, (right height – left height) differs by at most 1.



non-AVL Tree

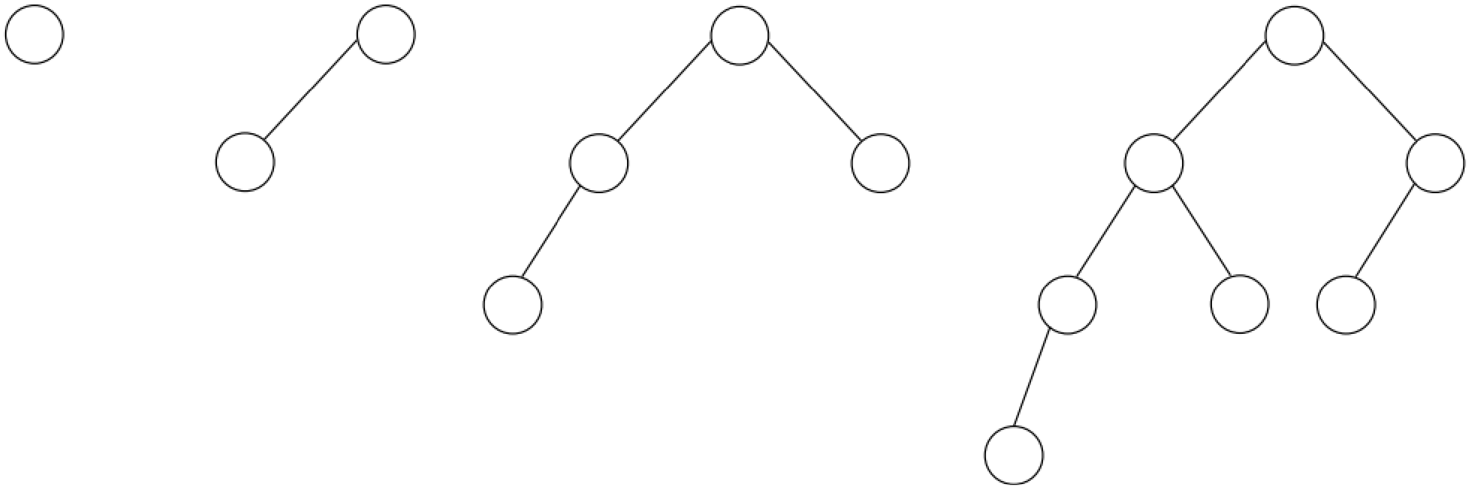


AVL Tree

- Q: Why is the height of an AVLtree $O(\log n)$?
- Q: How to maintain the AVL property after an insertion/deletion?

Height of an AVL-tree

- Observation: Equivalent to show that # nodes $\geq 2^{ch}$ for an AVL-tree of height h for some constant c.
  - Let n_h be the minimum number of nodes in an AVL tree of height h.
  - $n \geq n_h \geq 2^{ch} \Leftrightarrow h \leq \frac{1}{c} \log n = O(\log n)$
- We will prove the statement by mathematical induction



$n_0 = 1$ $\quad$ $n_1 = 2$ $\quad$ $n_2 = n_1 + n_0 + 1 = 4$ $\quad$ $n_3 = n_2 + n_1 + 1 = 7$

Height of an AVL-tree

- Claim: $n_h \geq 2^{h/2}$

Pf: (by induction on h)

$n_0 = 1, n_1 = 2$ (base case)

Recurrence: $n_h \geq \; n_{h-1} + n_{h-2} + 1$

$$\begin{aligned}
n_h &= n_{h-1} + n_{h-2} + \; 1 \\
&\geq 2n_{h-2} \qquad (n_{h-1} \geq n_{h-2} + 1) \\
&\geq 2 \cdot 2^{(h-2)/2} \qquad \text{(induction hypothesis)} \\
&= 2^{h/2}
\end{aligned}$$

Theorem: The height of an AVL-tree of n nodes is $O(\log n)$.

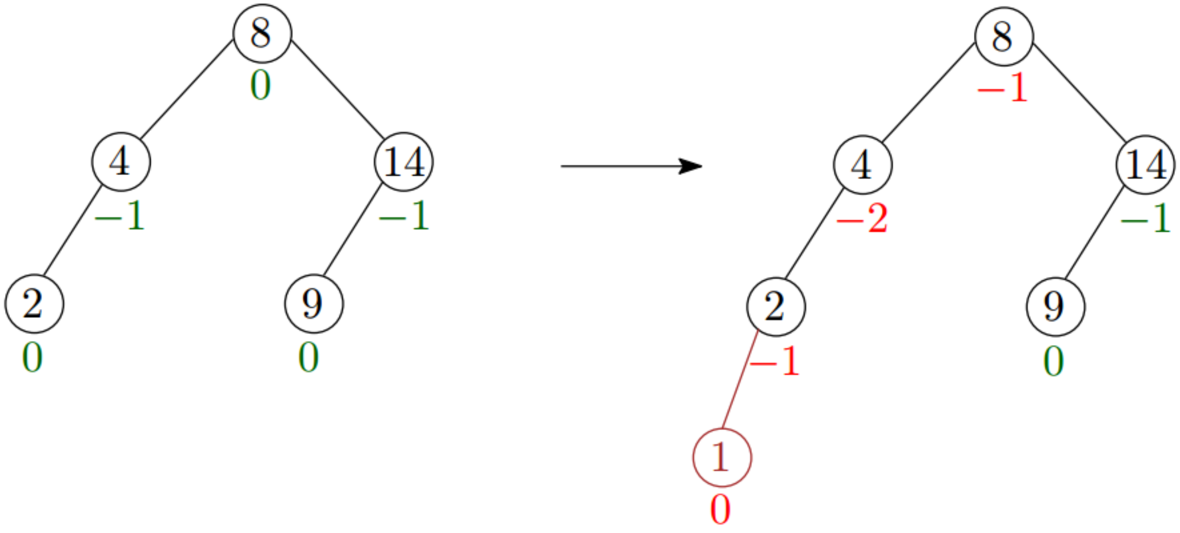
How does the AVL tree work?

- After insertion and deletion we will examine the tree structure and see if any nodes violates the AVL tree property
  - If the AVL property is violated, it means the heights of left(x) and right(x) differ by exactly 2

- If it does violate the property we can modify the tree structure using "rotations" to restore the AVL tree property

Restoring balance after an insertion

- After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered
  - Because only those nodes have their subtrees altered



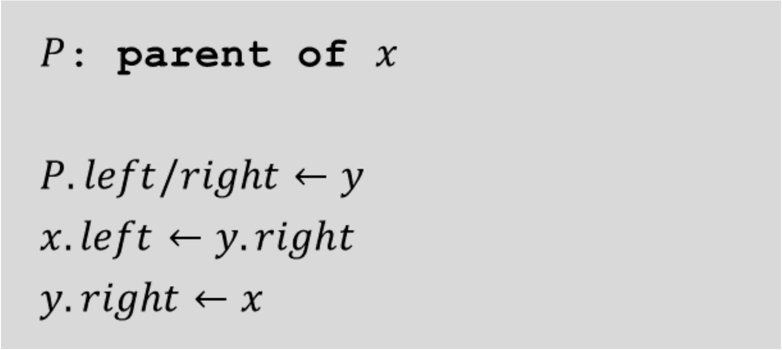
Insert 1

- Idea:
  - Update the heights of these nodes from the insertion node
  - Stop when we find the lowest node A violating AVL tree property
  - We will fix the tree at A.

Rotations

- Two types of rotations
  - Single rotations at the imbalance point
    - two nodes are “rotated”
  - Double rotations at the imbalance point
    - three nodes are “rotated”
- We’ll see them first and see when to use them later

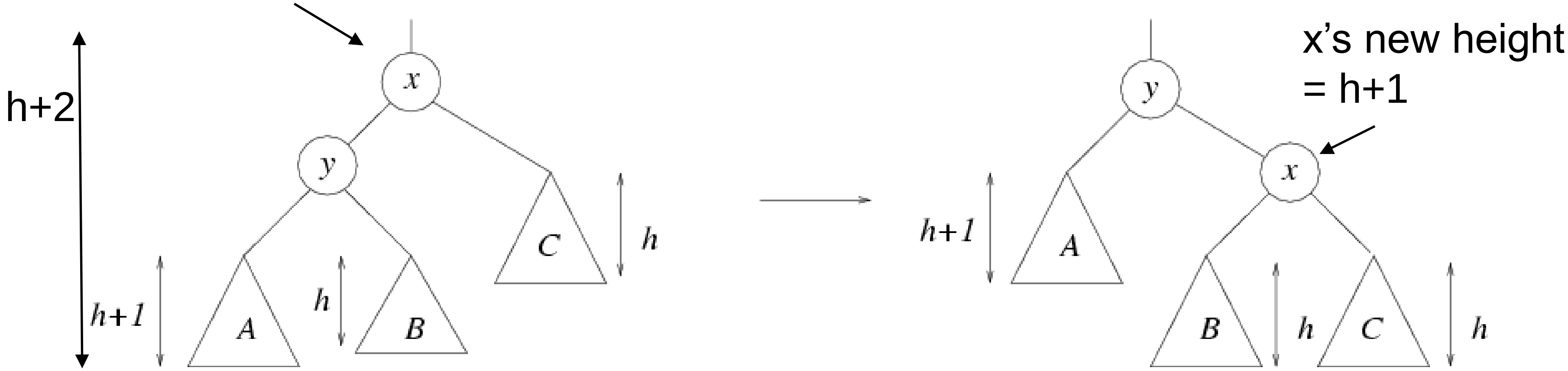
The Left-left Single Rotation (Case 1): Height(left(left(x))) = h+1



```
P: parent of x

P.left/right ← y
x.left ← y.right
y.right ← x
```

First imbalance point from the leave:
left(x) – right(x) = 2

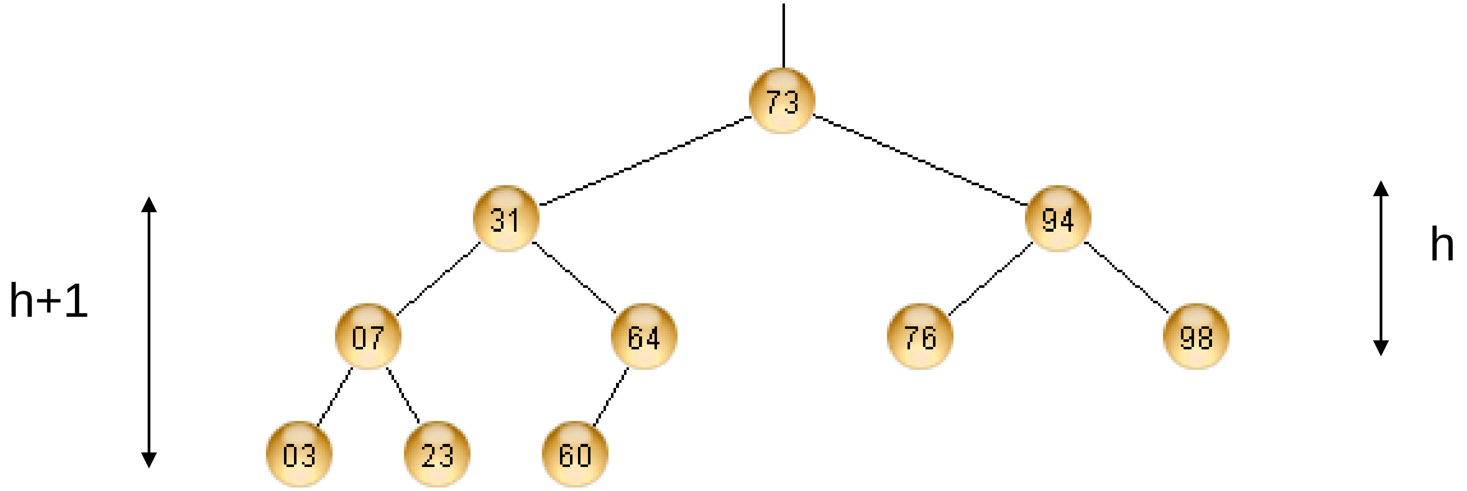


x's new height = h+1

Rotate **x** with the left child of **y**
(pay attention to the resulting sub-trees positions)

(Note that the height of B can be h+1, in which case x's new height would be h+2)

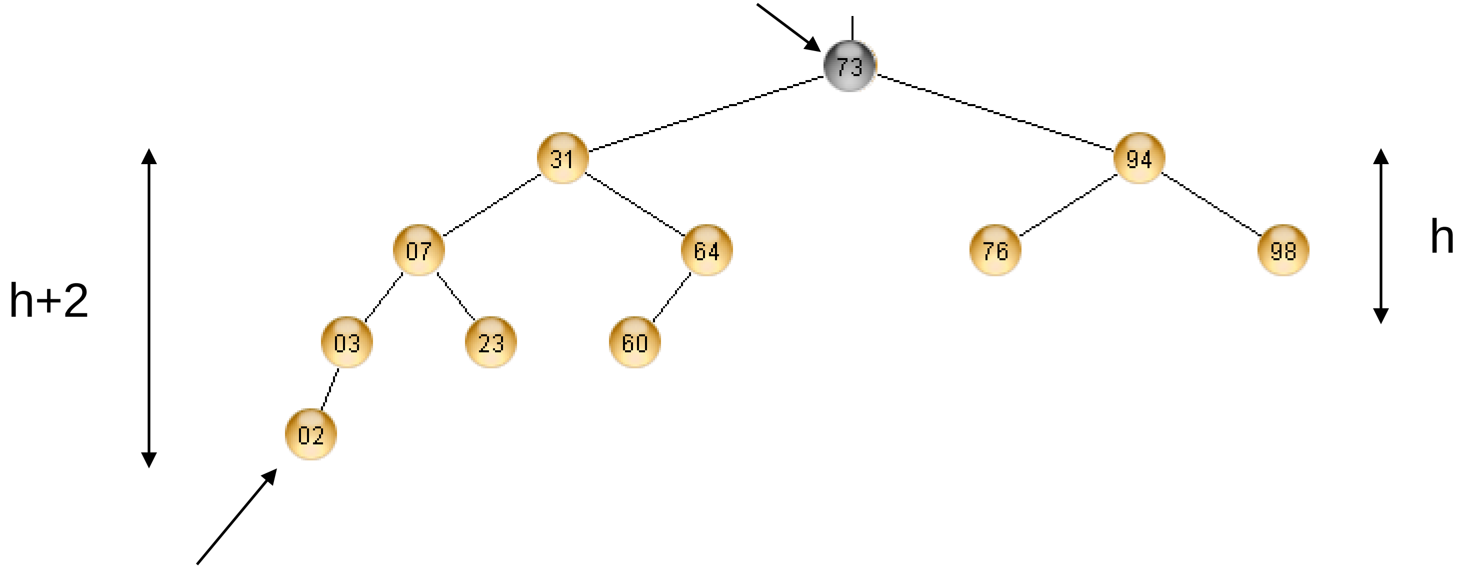
Single Rotation - Example



Tree is an AVL tree by definition.

Add a node 02

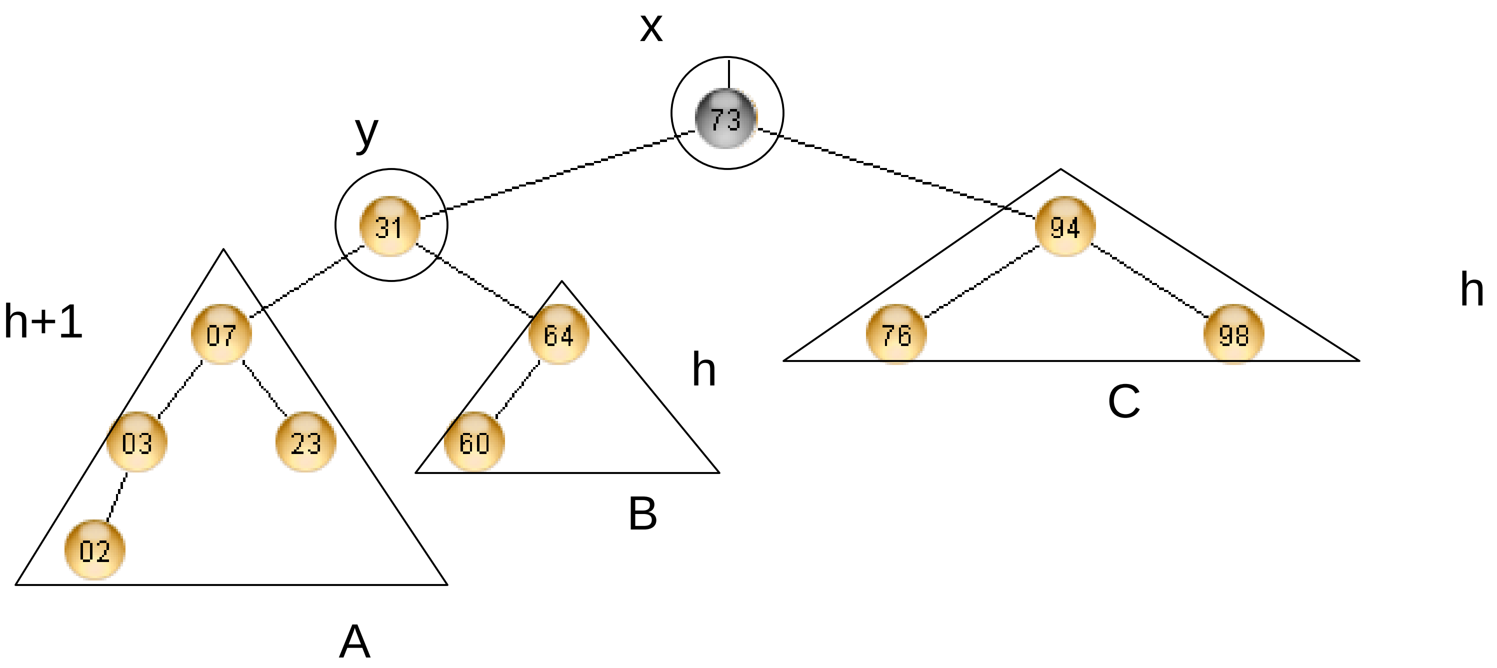
First imbalance point from the leave



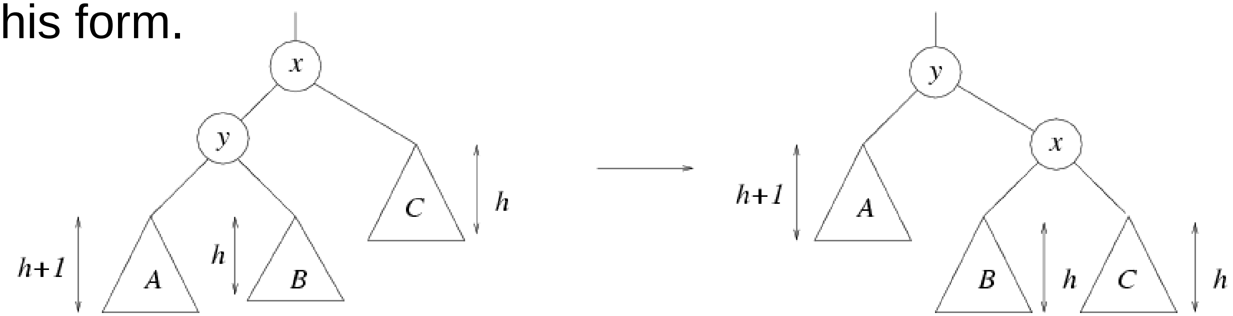
Node 02 added

Tree violates the AVL definition!
Perform rotation.

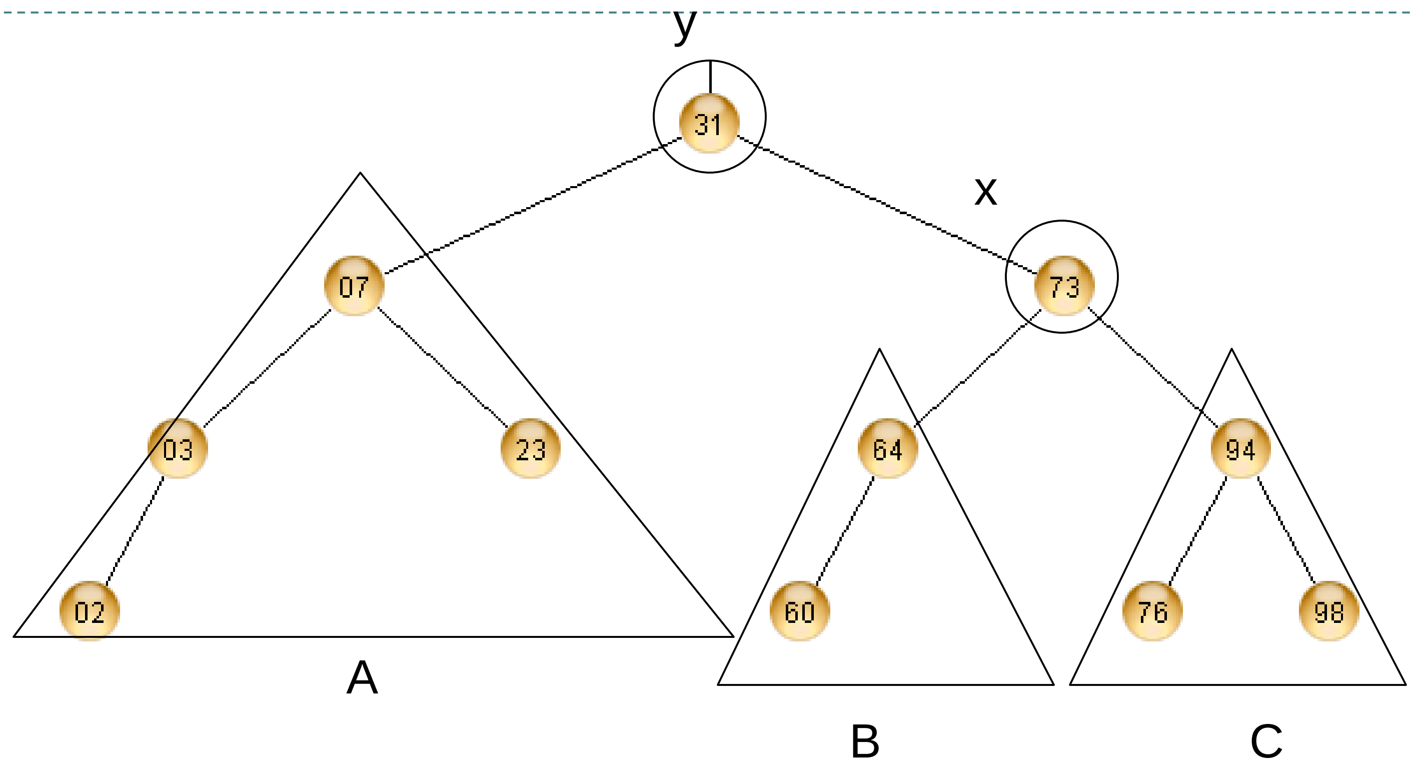
Example



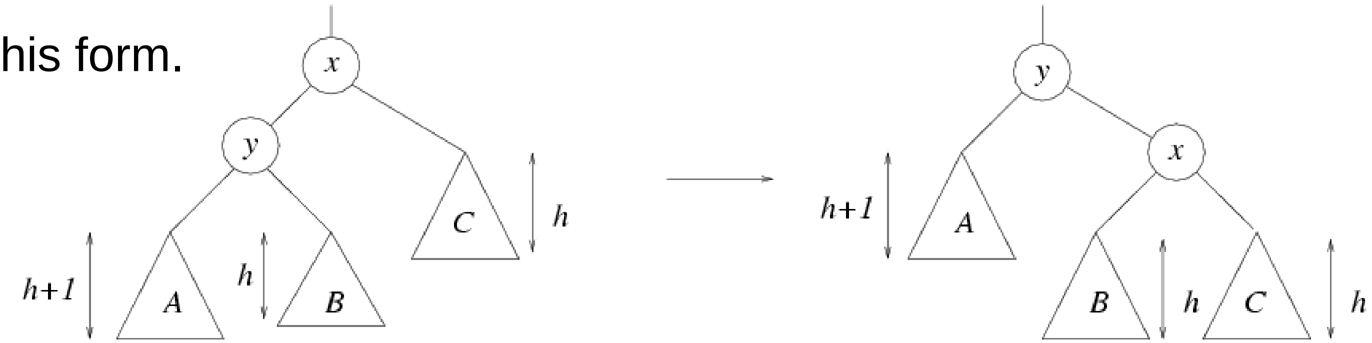
Tree has this form.



Example – After Rotation

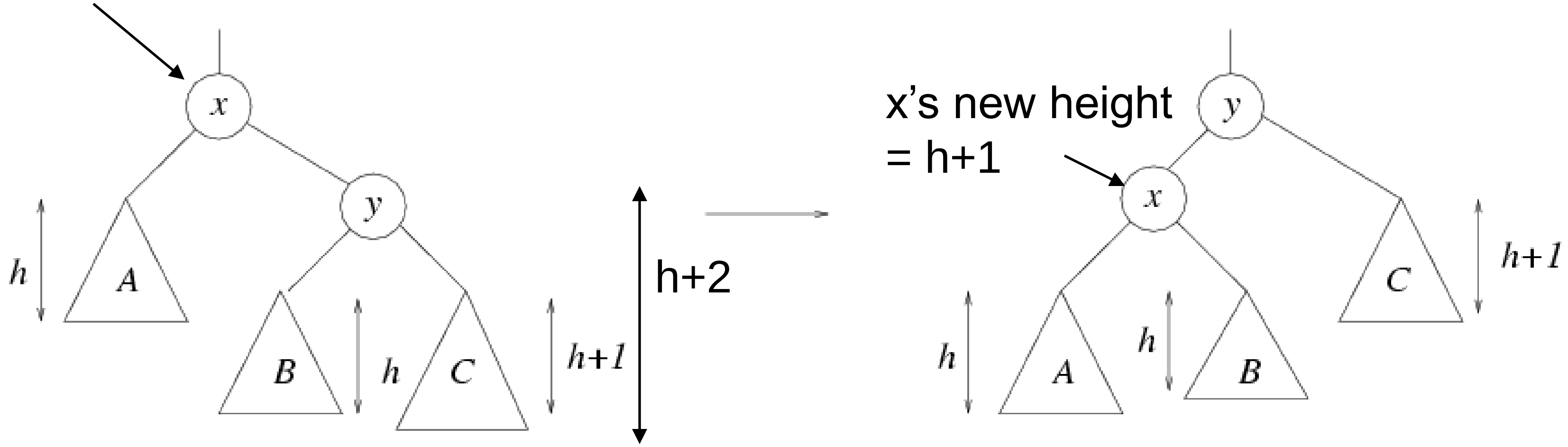


Tree has this form.



Right-Right Single Rotation (Case 2): Height(right(right(x))) = h+1

AVL property is first violated at x from the leave:
left(x) – right(x) = -2



h+2

x’s new height
= h+1

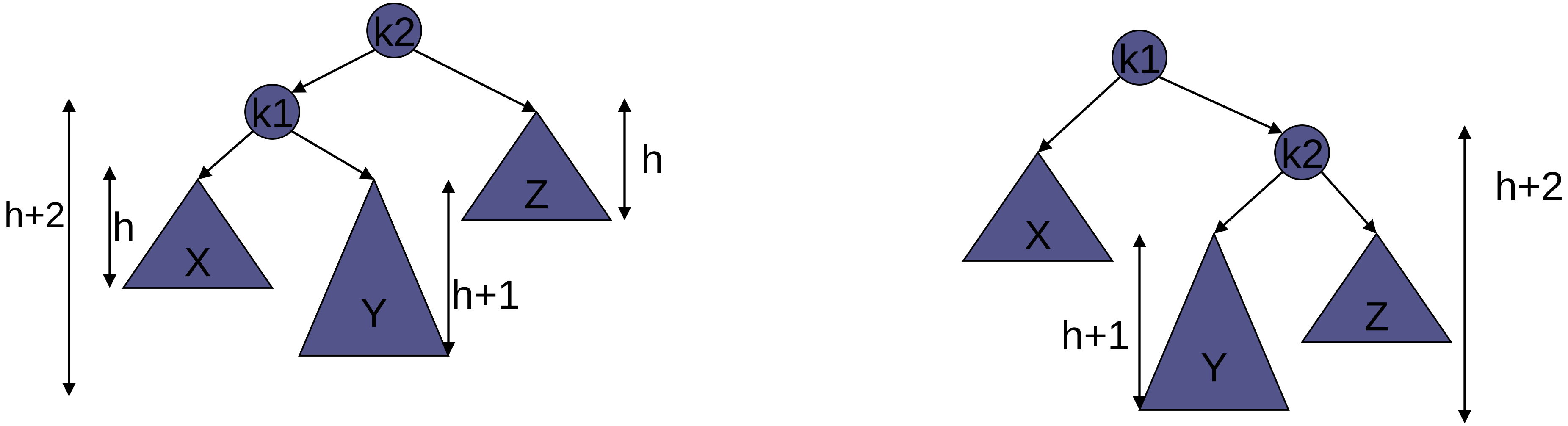
Rotate **x** with the right child of **y**

(pay attention to the resulting sub-trees positions)

(Note that the height of B can be h+1, in which case x’s new height would be h+2)

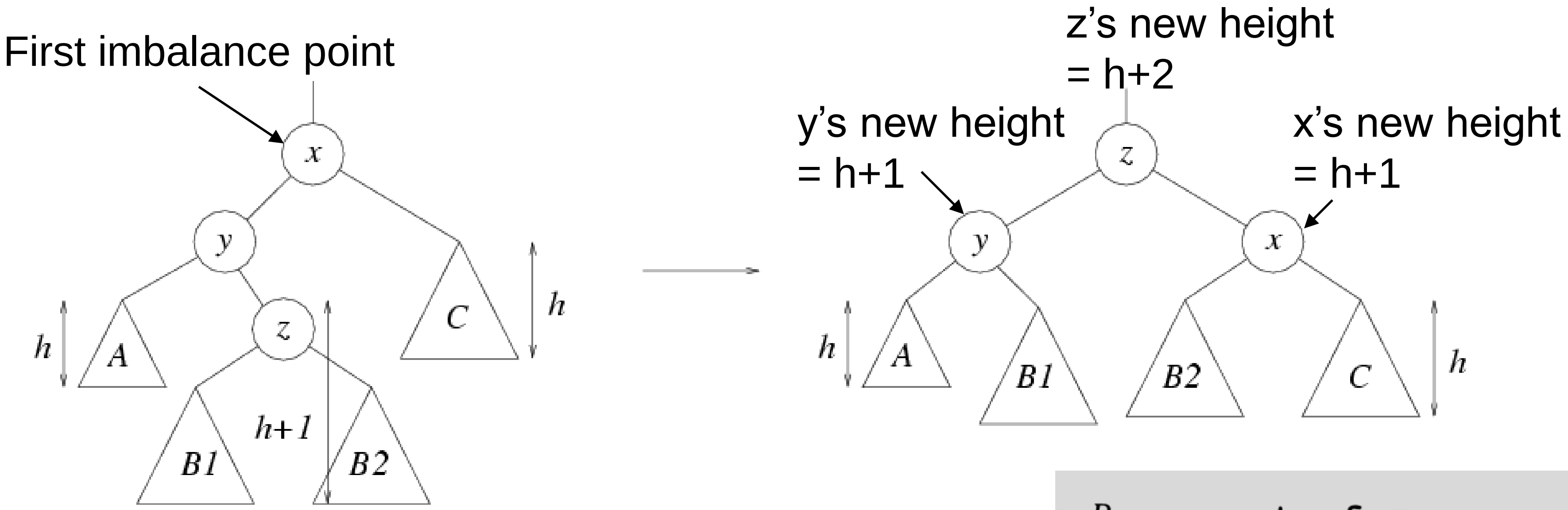
Single Rotation

- Sometimes a single rotation fails to solve the problem



- In such cases, we need to use a double-rotation

Right-Left Double Rotations (Case 3): Height(right(left(x))) = h+1



First imbalance point

z's new height = h+2

y's new height = h+1

x's new height = h+1

Double-rotate **x** with its left child **y** <u>**and**</u> **y's** right child **z**
(pay attention to the resulting sub-trees positions)

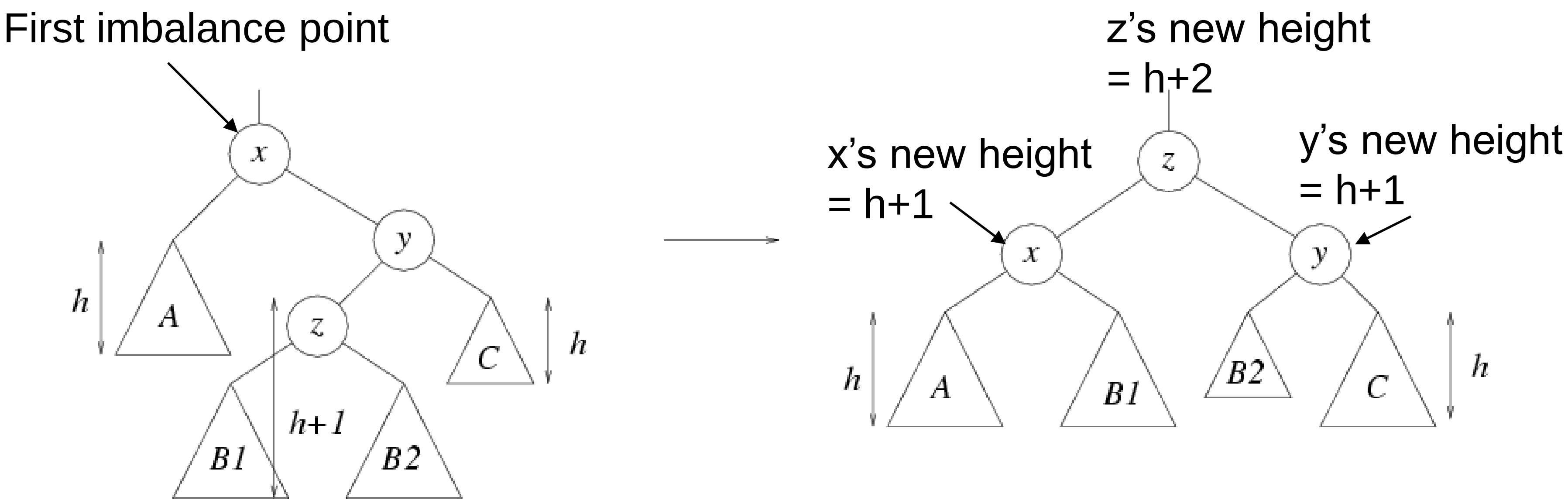
Involves 2 single rotations on z:

1. Single rotate z upwards with y, pushing y down
2. Single rotate z upwards again with x, pushing x down in another branch

```
P: parent of x

P.left/right ← z
x.left ← y.right
y.right ← x.left
z.left ← y
z.right ← x
```

Double Rotations: Case 4
Height(left(right(x))) = h+1



First imbalance point

z’s new height
= h+2

x’s new height
= h+1

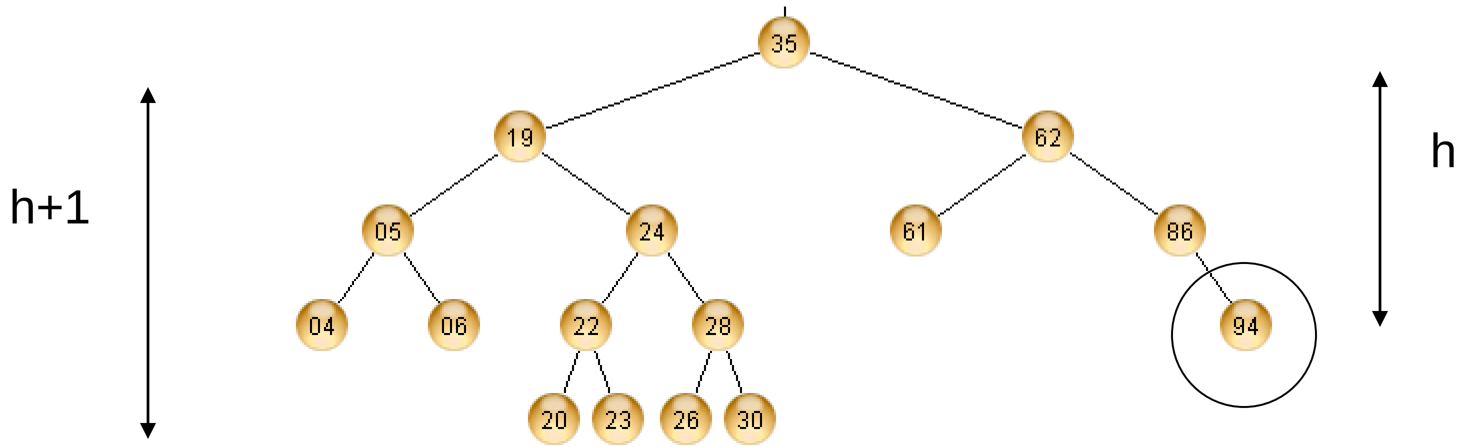
y’s new height
= h+1

Double-rotate **x** with its right child **y** **<u>and</u>** **y’s** left child **z**

(pay attention to the resulting sub-trees positions)

Involves 2 single rotations on z (similar as before):
Single rotate z upwards with y, pushing y down
Single rotate z upwards again with x, pushing x down in another branch

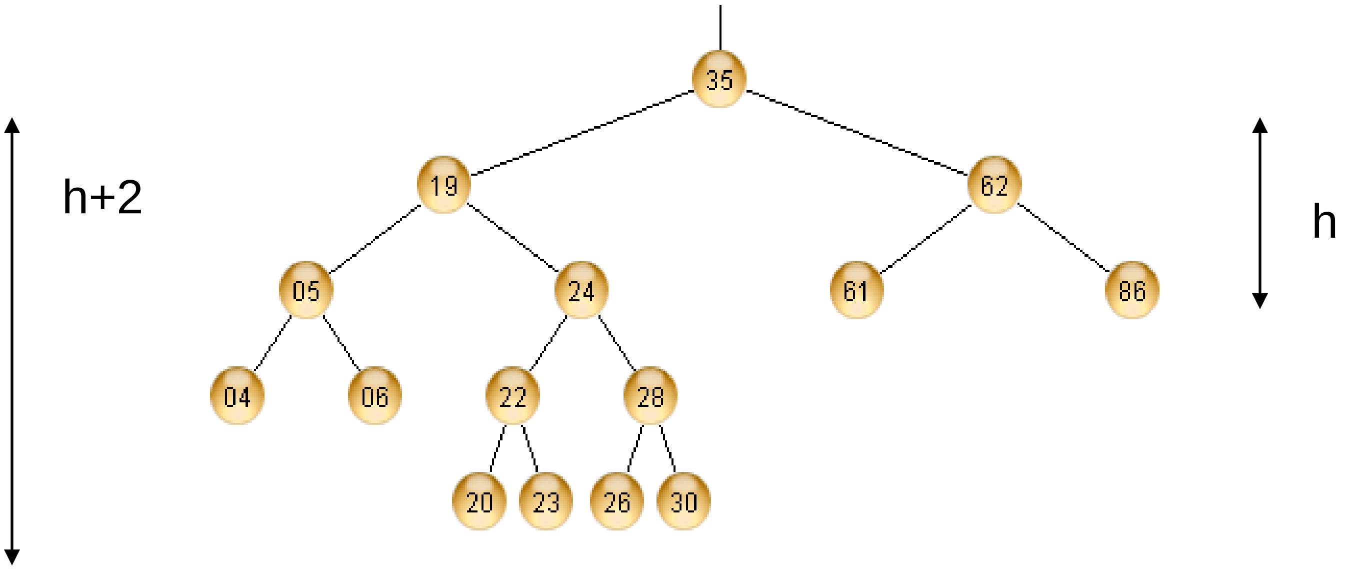
Double Rotation - Example



Delete node 94

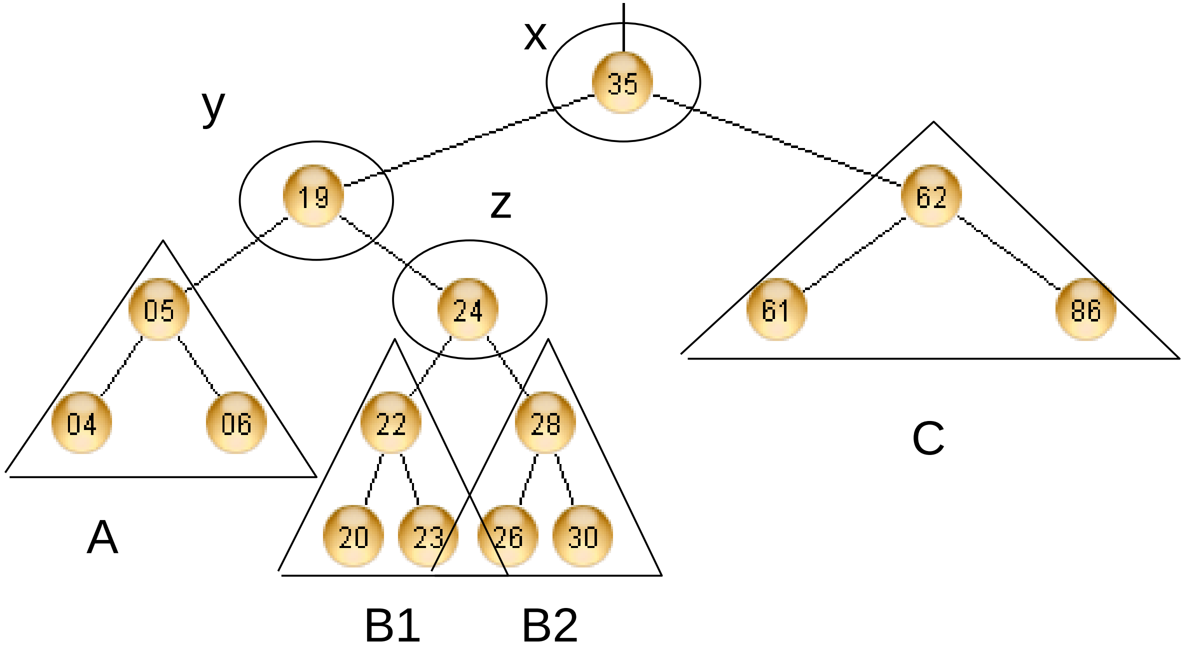
Tree is an AVL tree by definition.

Example

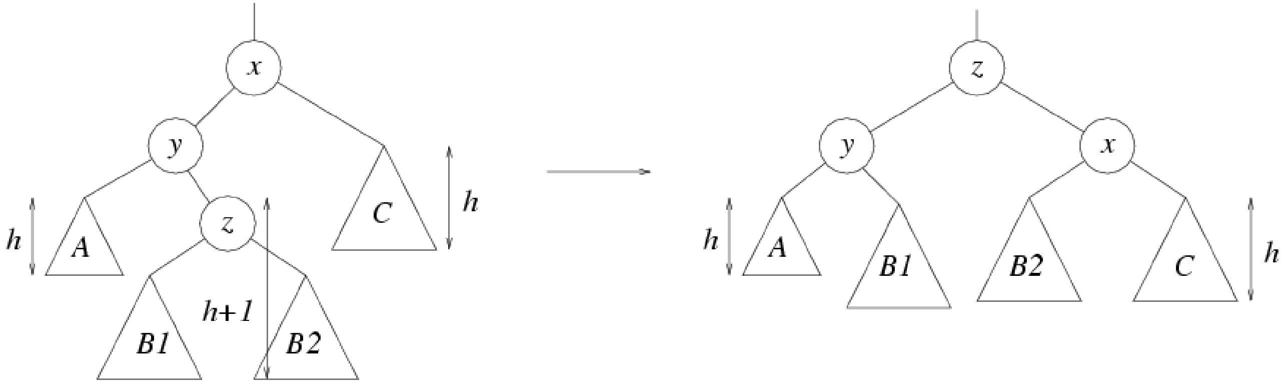


AVL tree property is violated.

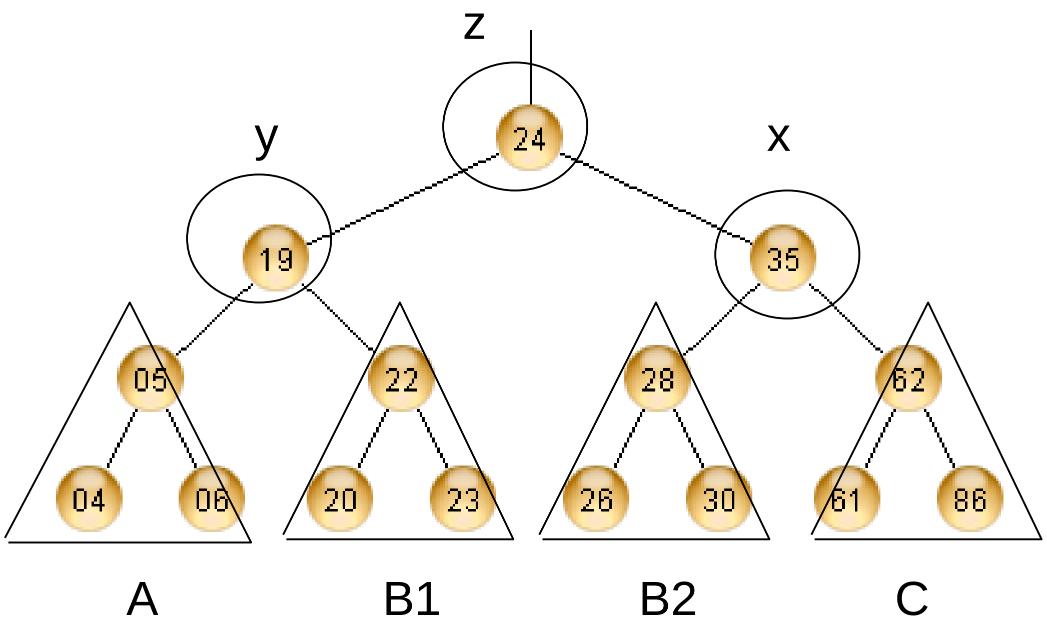
Example



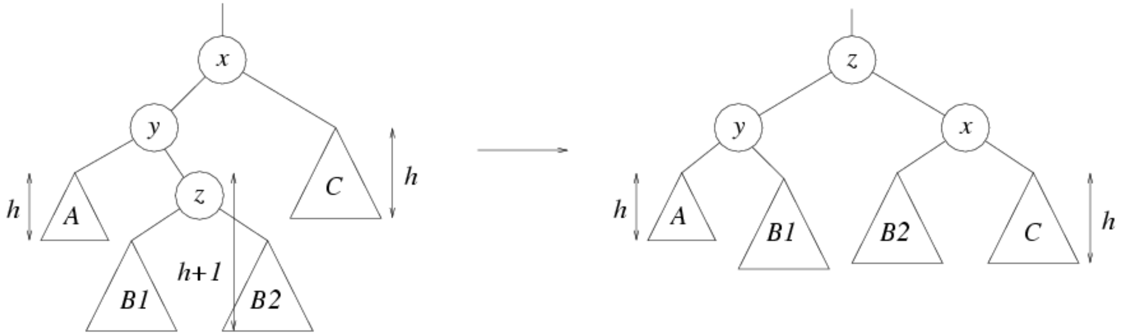
Tree has this form.



After Double Rotation



Tree has this form



Insertion

Part 1. Perform normal BST insertion

Part 2. Check and correct AVL properties

Trace back on the path from the inserted leaf all the way towards the root:

- Check to see if heights of left(x) and right(x) differ at most by 1
- If not, we know x is the imbalance point (the height of x is h+3)
  - □ If left(x) is higher (h+2), then
    - □ If left(left(x)) is of height h+1, we single rotate with x's left child, i.e., left(x) (case 1)
    - □ Otherwise [right(left(x)) is higher (h+1)] we double rotate with x's left child, i.e., left(x) (case 3)
  - □ Otherwise, height of right(x) is longer (h+2)
    - □ If right(right(x)) is of height h+1, then we rotate with x's right child, i.e., right(x) (case 2)
    - □ Otherwise [left(right(x)) is higher (h+1)] we double rotate with x's right child, i.e., right(x) (case 4)

* Rotations may stop somewhere leading to the root. Remember to make the rotated node the new child of parent(x)

Insertion

- The time complexity to perform a rotation is O(1)
- The time complexity to find a node that violates the AVL property is dependent on the height of the tree (which is log(N))
- The height of a node can be found in O(N) time.
- The height of a node can also be more efficiently stored in a node, and dynamically updated locally each time insertion or deletion occurs. In this way, the height can be accessed in O(1) time.
  - In this case, the insertion takes O(log n) time

Deletion

- Perform normal BST deletion

- Perform exactly the same checking as for insertion to restore the tree property

Note

- There are other variations in the way AVL trees are implemented. These notes present a nice way that treats insertion and deletion the same.

- All implementations have the same idea, detect an "imbalance" in height for a node and perform corrections via single or double rotations.

- Red-black tree (more complicated, but more efficient in terms of space; see textbook)

Summary AVL Trees

- Maintains a balanced tree
- Modifies the insertion and deletion routine
  - Performs single or double rotations to restore structure
  - Requires a little more work for insertion and deletion
  - But, since trees are mostly used for searching
    - More work for insert and delete is worth the performance gain for searching
- Guarantees that the height of the tree is O(logn)
  - The guarantee directly implies that functions find(), min(), and max() will be performed in O(logn)