Chapter 4
Trees

South China University of Technology

College of Software Engineering

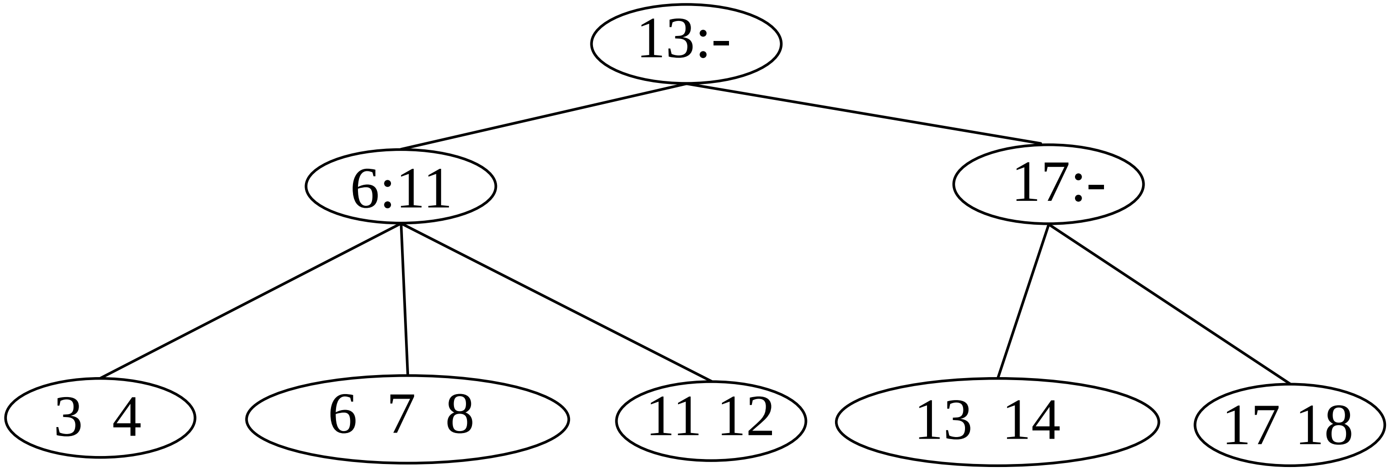
Huang Min

Chapter 4
Tree- part2

B-Tree

Beyond Binary Search Trees: Multi-Way Trees

- Example: B-tree of order 3 has 2 or 3 children per node



- Search for 8

B-Trees

B-Trees are multi-way search trees commonly used in database systems or other applications where data is stored externally on disks and keeping the tree shallow is important.

A B-Tree of order M has the following properties:

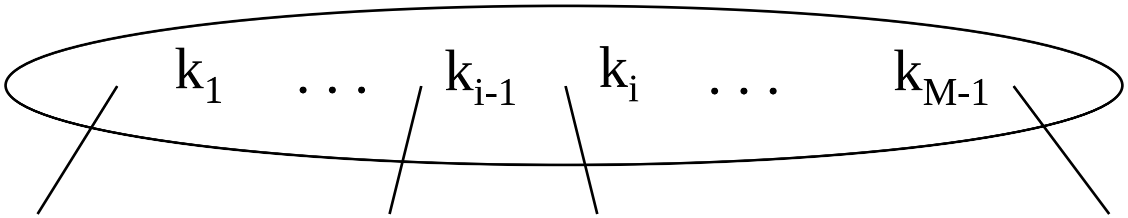
1. The root is either a leaf or has between 2 and M children.
2. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
3. All leaves are at the same depth.

All data records are stored at the leaves.
Internal nodes have “keys” guiding to the leaves.
Leaves store between $\lceil L/2 \rceil$ and L data records,
where L can be equal to M (default) or can be different.

B-Tree Details

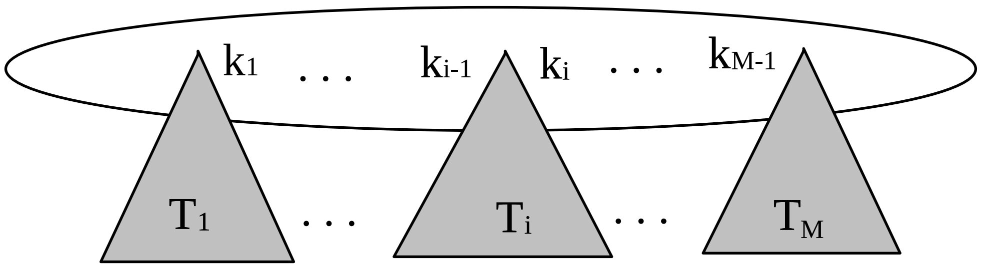
Each (non-leaf) internal node of a B-tree has:

- Between $\lceil M/2 \rceil$ and M children.
- up to M-1 keys $k_1 < k_2 < ... < k_{M-1}$



Keys are ordered so that:
$k_1 < k_2 < ... < k_{M-1}$

Properties of B-Trees



Children of each internal node are "between" the items in that node.

Suppose subtree T_i is the *i*th child of the node:

all keys in T_i must be between keys k_{i-1} and k_i

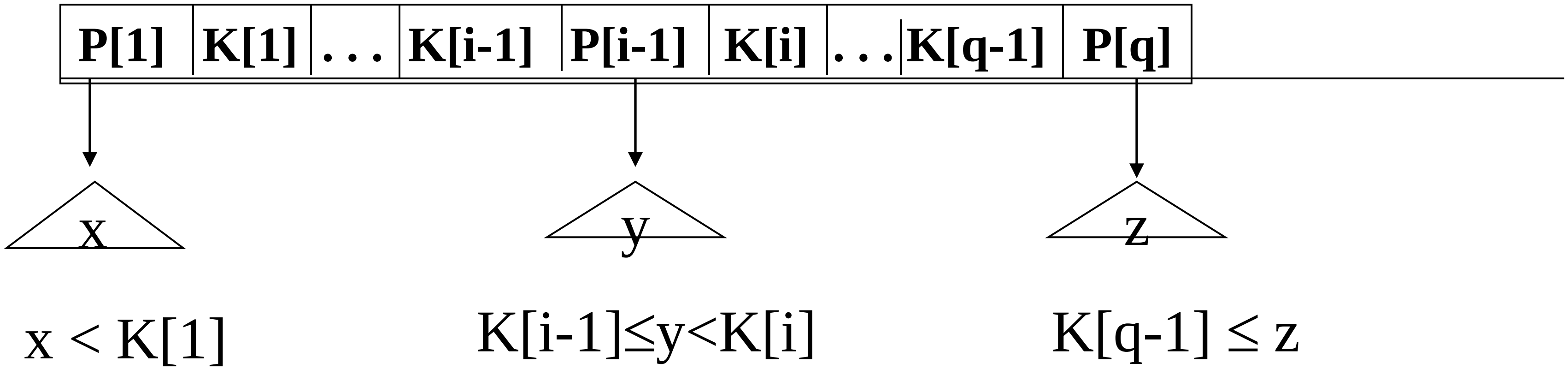
i.e. $k_{i-1} \leq T_i < k_i$

k_{i-1} is the smallest key in T_i

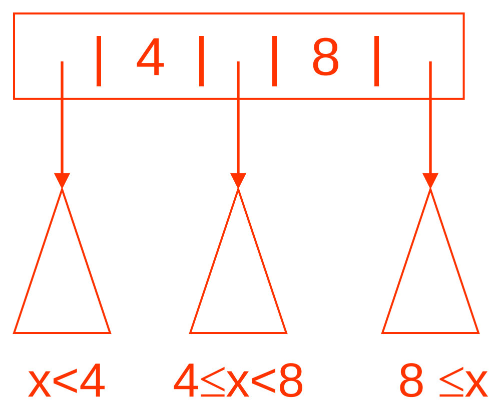
All keys in first subtree $T_1 < k_1$

All keys in last subtree $T_M \geq k_{M-1}$

B-Tree Nonleaf Node



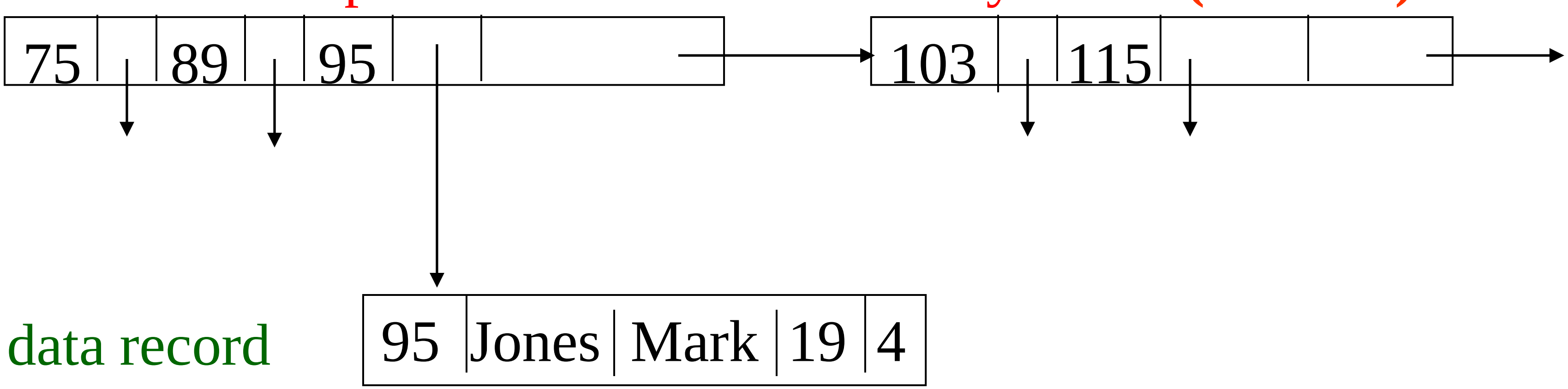
- The Ks are keys
- The Ps are pointers to subtrees.



Detailed Leaf Node Structure **(B+ Tree)**

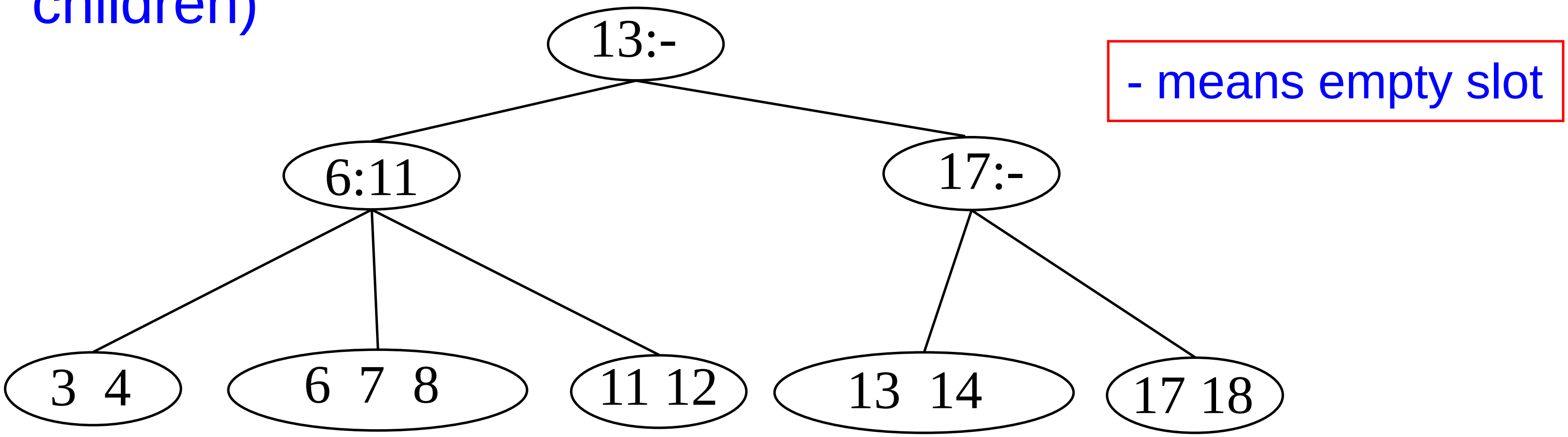
K[1]	R[1]	. . .	K[q-1]	R[q-1]	Next

- The Ks are keys (assume unique).
- The Rs are pointers to records with those keys.
- The Next link points to the next leaf in key order **(B+-tree)**.



Searching in B-trees

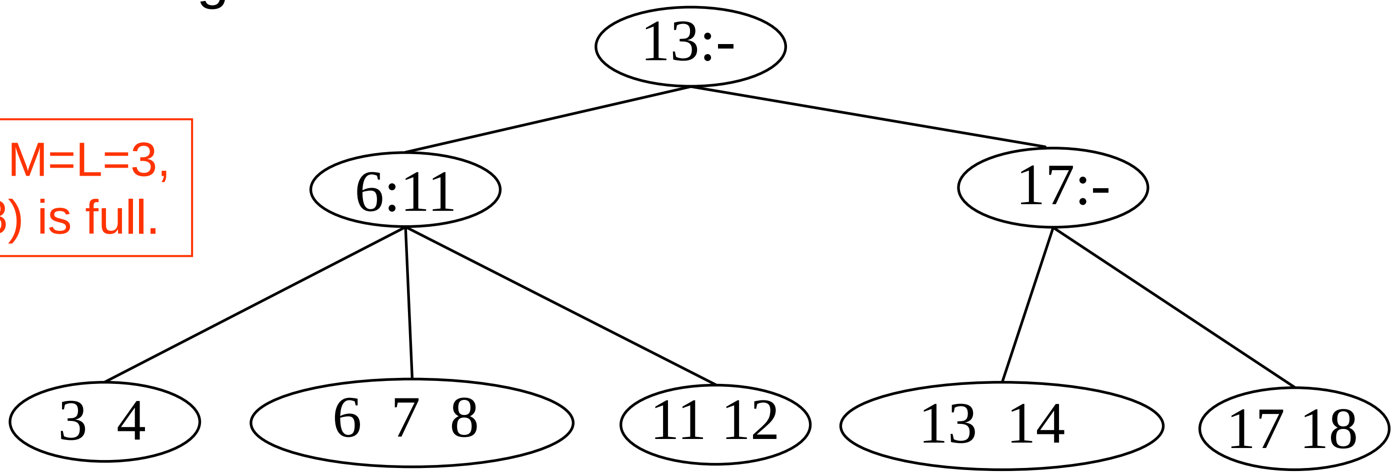
- B-tree of order 3: also known as 2-3 tree (2 to 3 children)



- Examples: Search for 9, 14, 12
- Note: If leaf nodes are connected as a Linked List, B-tree is called a B+ tree – Allows sorted list to be accessed easily

Inserting into B-Trees

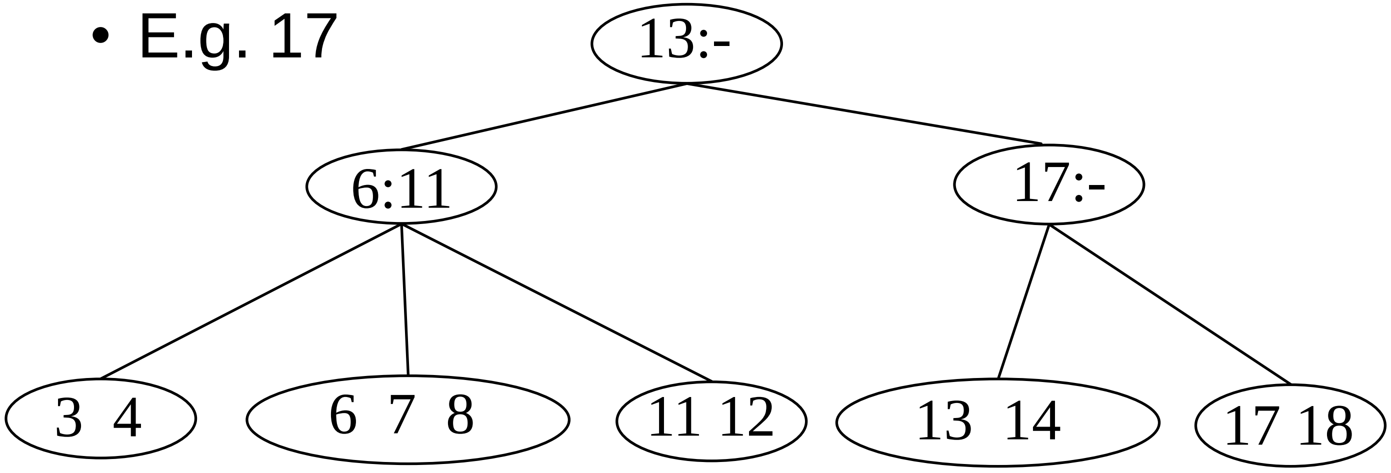
- Insert X: Do a Find on X and find appropriate leaf node
  - › If leaf node is not full, fill in empty slot with X
    - E.g. Insert 5
  - › If leaf node is full, split leaf node and adjust parents up to root node
    - E.g. Insert 9



Assume M=L=3, so (6 7 8) is full.

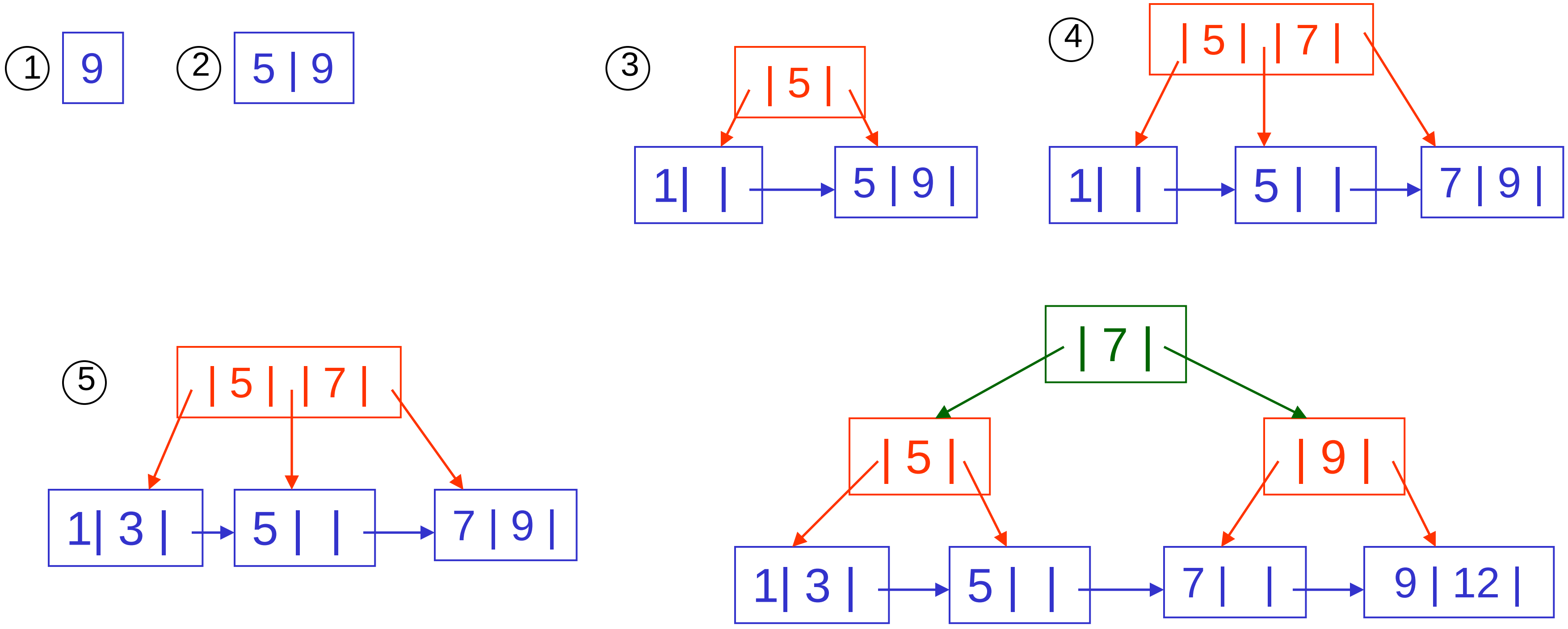
Deleting From B-Trees

- Delete X : Do a find and remove from leaf
  - › Leaf underflows – borrow from a neighbor
    - E.g. 11
  - › Leaf underflows and can't borrow – merge nodes, delete parent
    - E.g. 17

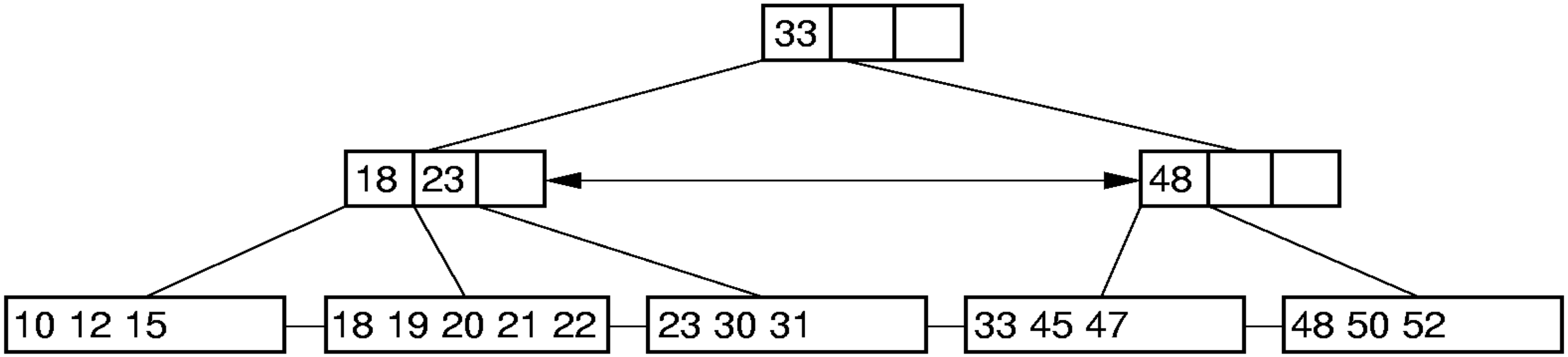


Example of Insertions into a B+ tree with M=3, L=2

Insertion Sequence: 9, 5, 1, 7, 3,12

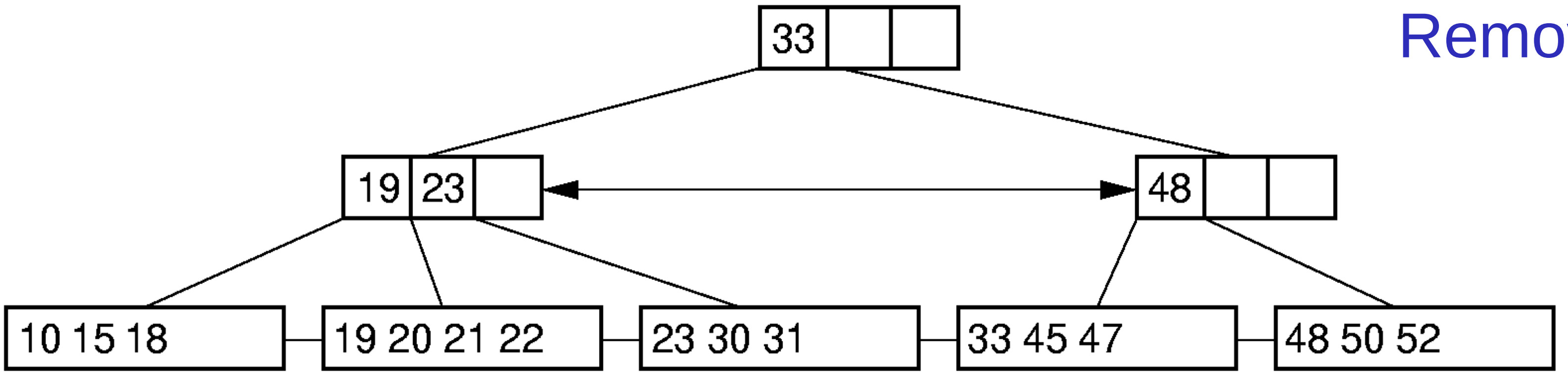


B+-Tree Deletion



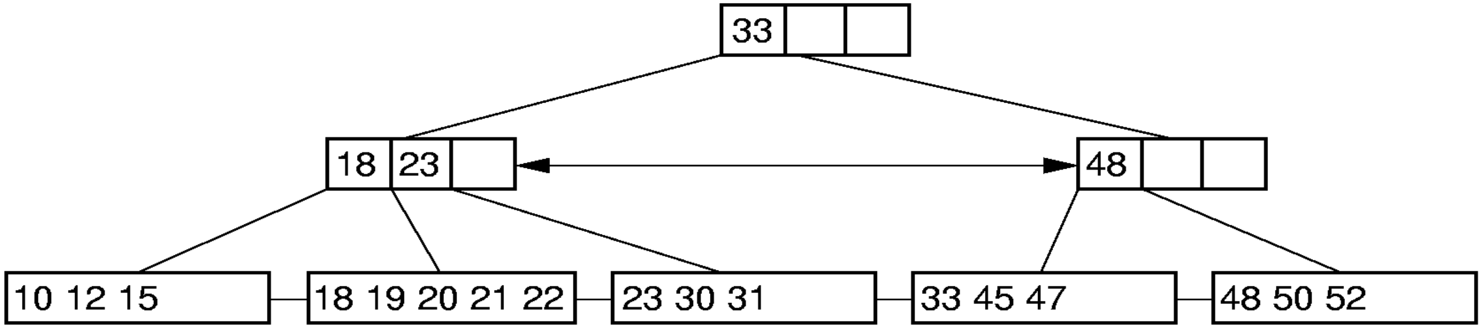
Example of a B$^+$ -tree of order four.

Remove 12

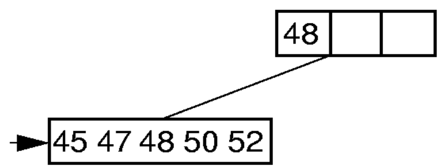


Deletion from the B$^+$ -tree via borrowing from a sibling.

B+-Tree Deletion

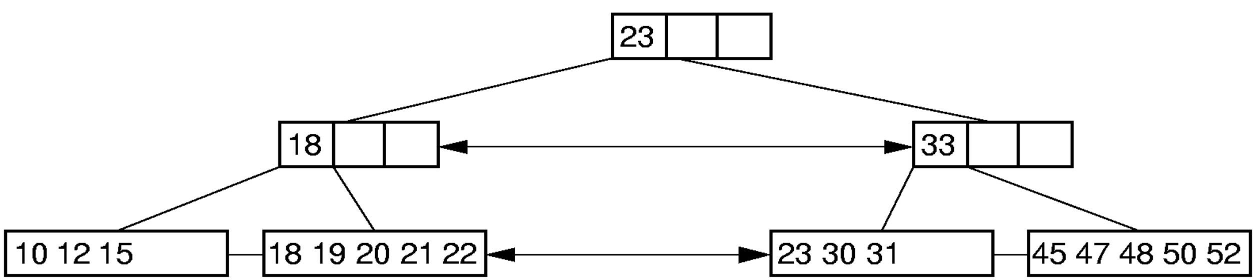


Example of a B^{+} -tree of order four.



(a)

Remove 33



(b)

Deleting the record with key value 33 from the B^{+} -tree via collapsing siblings.

AVL Trees

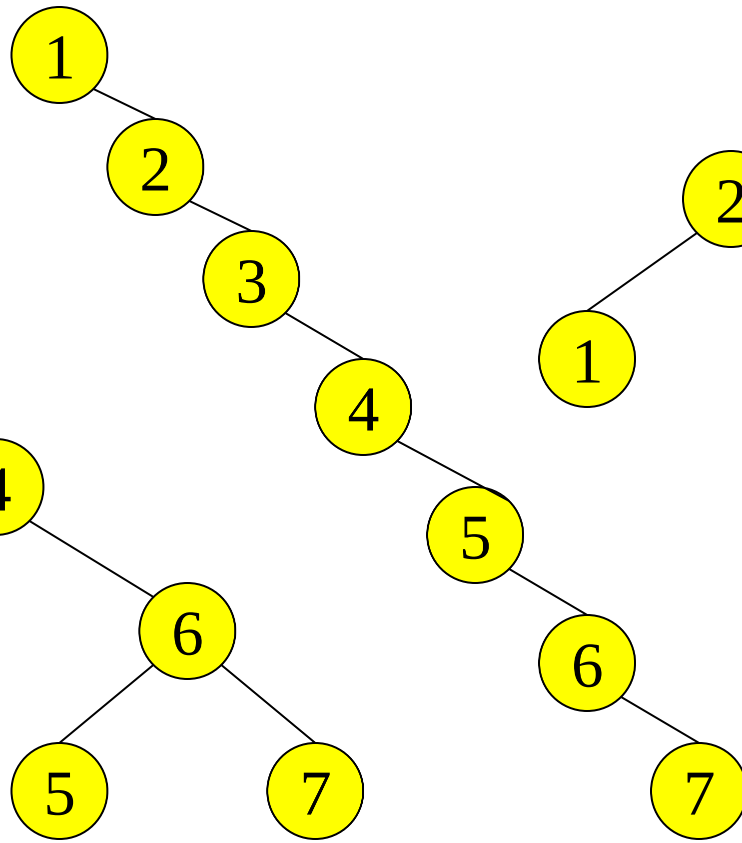
Binary Search Tree - Best Time

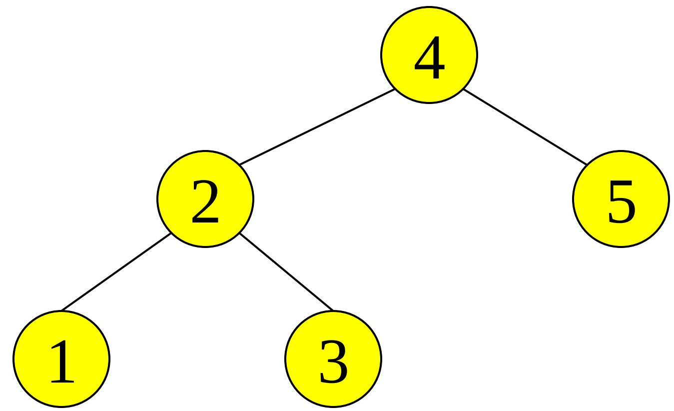
- All BST operations are O(d), where d is tree depth
- minimum d is $d = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - › What is the best case tree?
  - › What is the worst case tree?
- So, best case running time of BST operations is O(log N)

Binary Search Tree - Worst Time

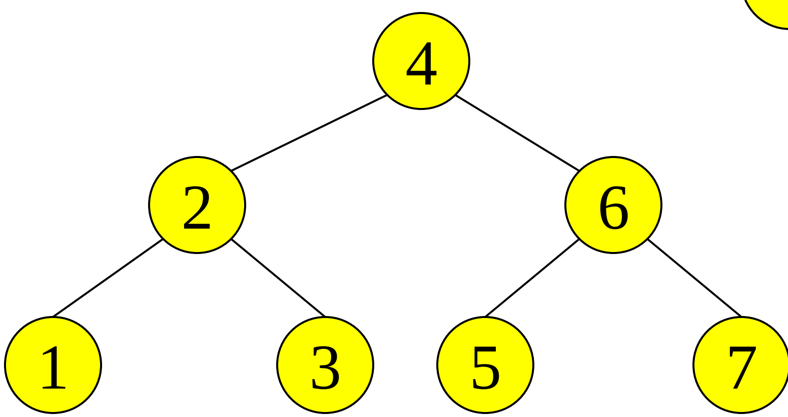
- Worst case running time is O(N)
  - What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Problem: Lack of "balance":
    - compare depths of left and right subtree
  - Unbalanced degenerate tree

Balanced and unbalanced BST





Is this “balanced”?



Approaches to balancing trees

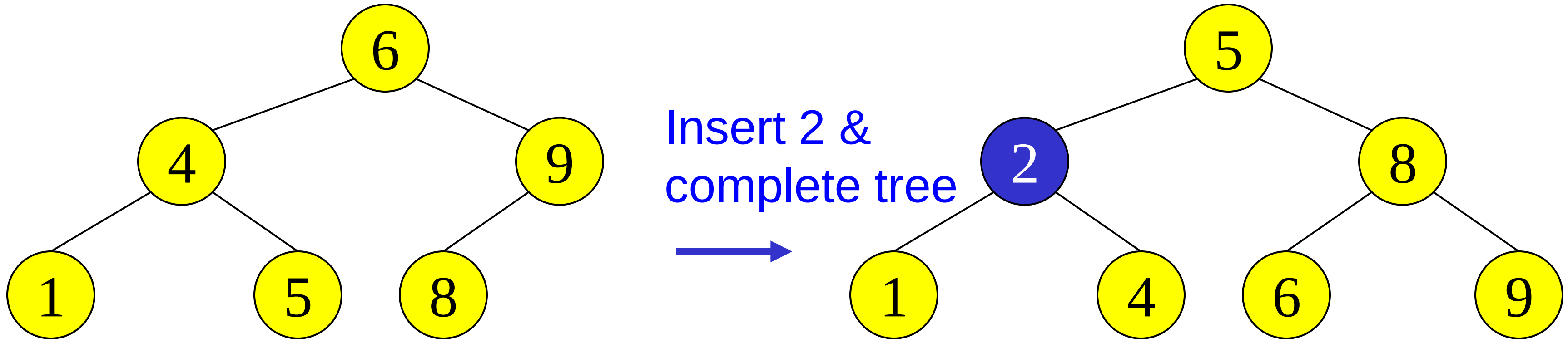
- Don't balance
  - May end up with some nodes very deep
- Perfectly balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

Perfect Balance

- Want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree

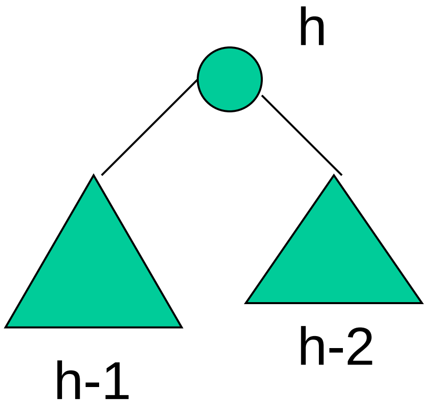


AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

Height of an AVL Tree

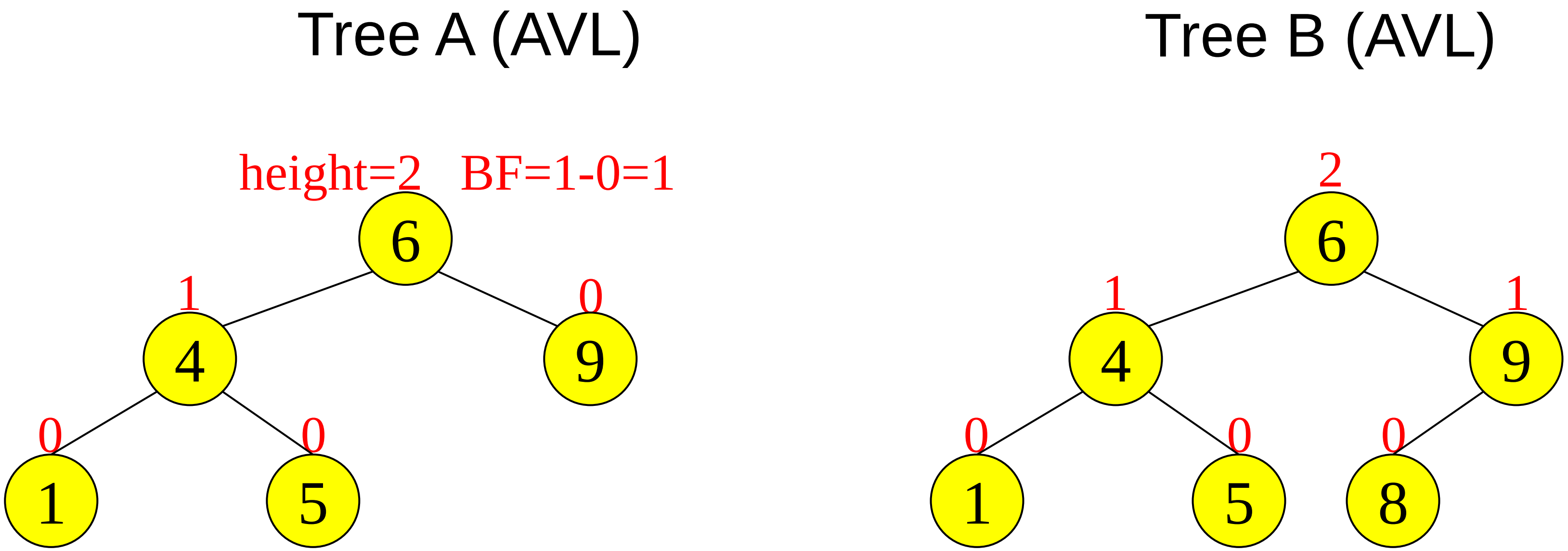
- N(h) = minimum number of nodes in an AVL tree of height h.
- Basis
  - › $N(0) = 1$, $N(1) = 2$
- Induction
  - › $N(h) = N(h-1) + N(h-2) + 1$
- Solution (recall Fibonacci analysis)
  - › $N(h) \geq \phi^h \quad (\phi \approx 1.62)$



Height of an AVL Tree

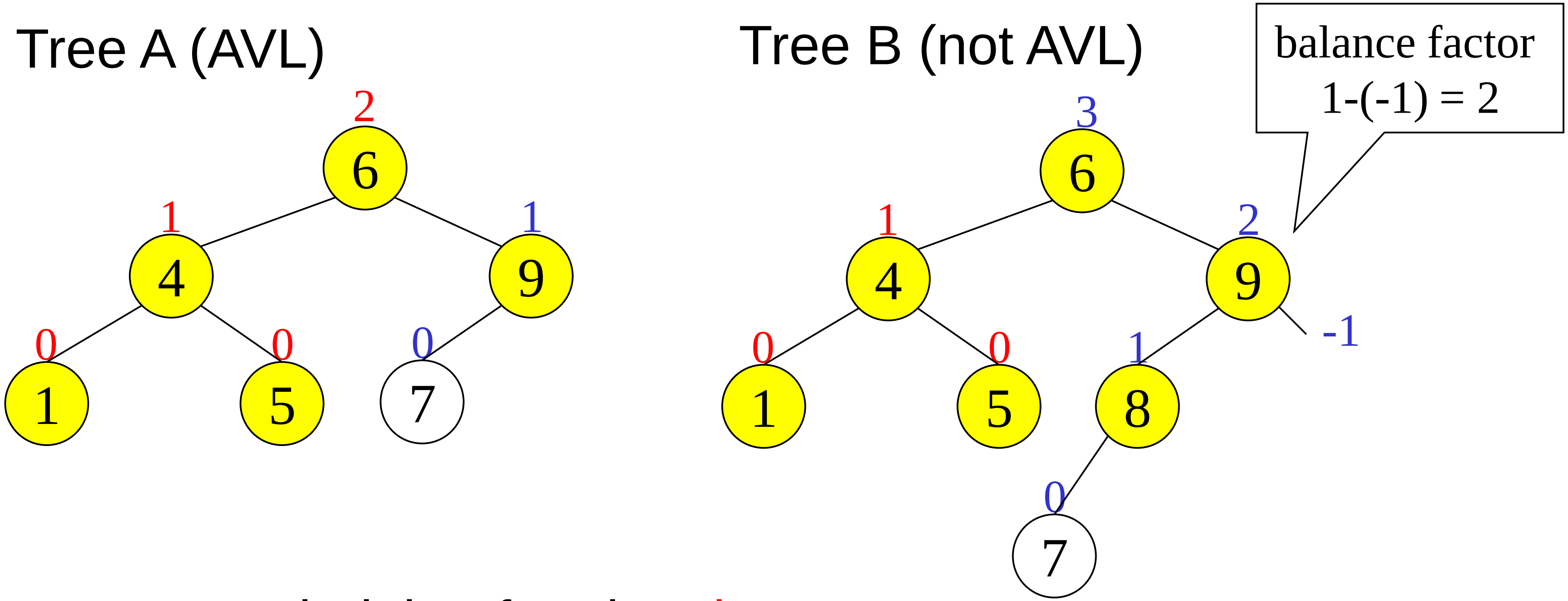
- $N(h) \geq \phi^h$ ($\phi \approx 1.62$)
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes O(logn))

Node Heights



height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

Node Heights after Insert 7



height of node = h

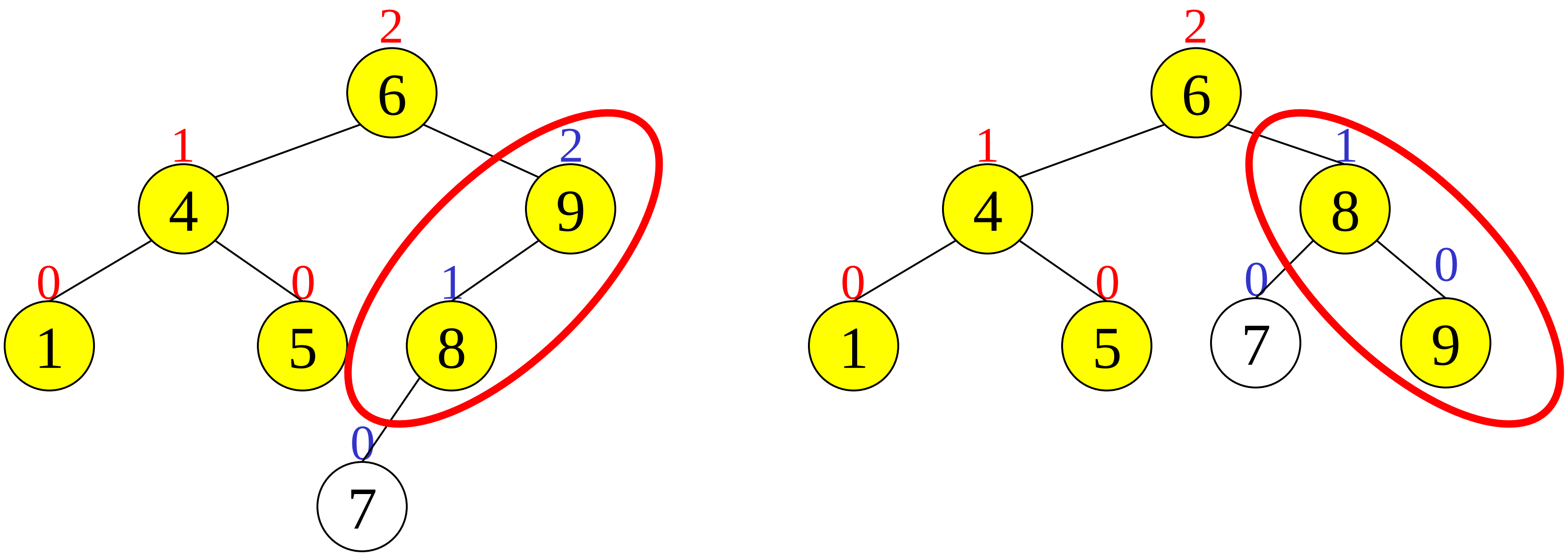
balance factor = h_{left}-h_{right}

empty height = -1

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation in an AVL Tree



Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

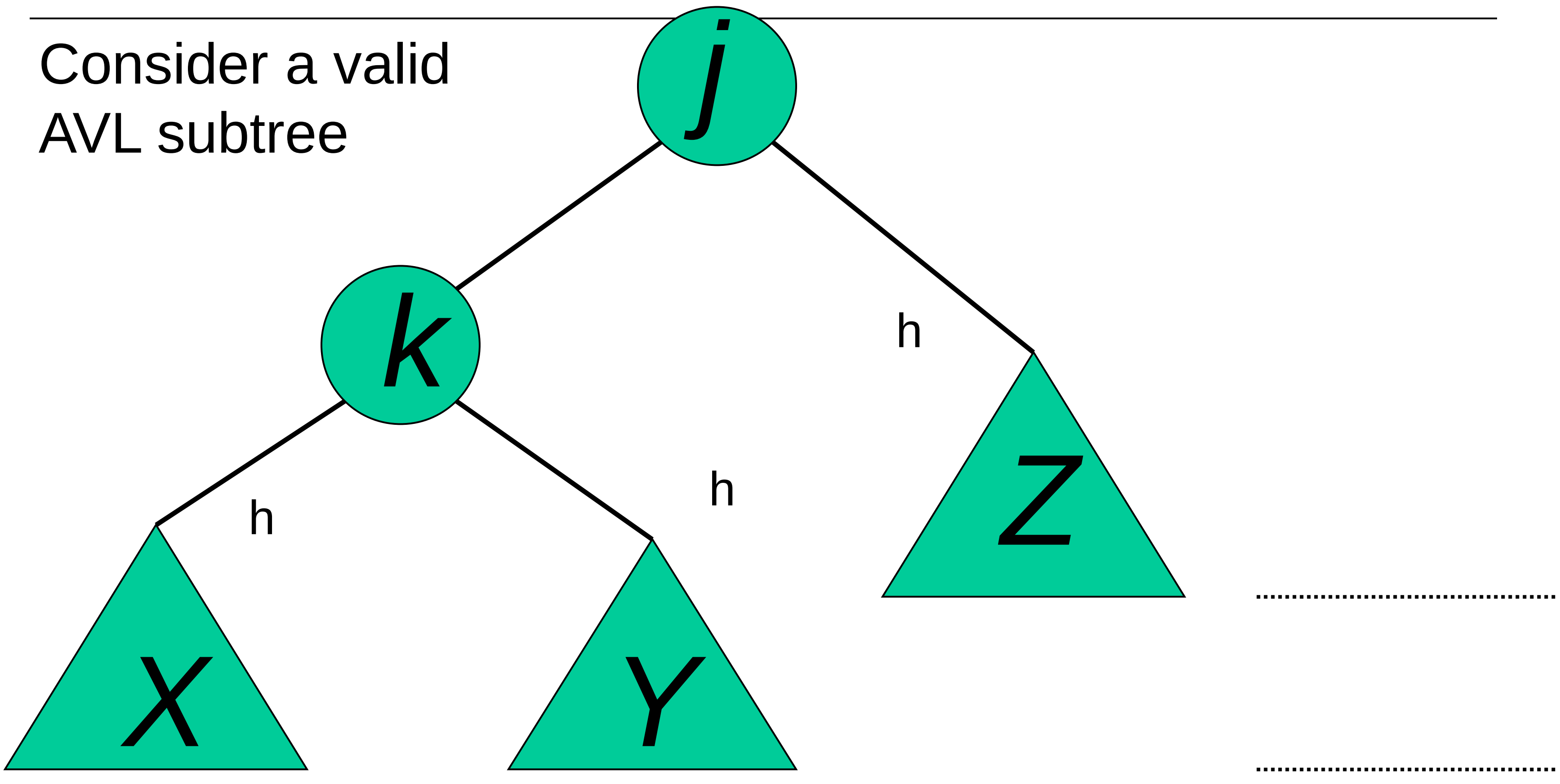
Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

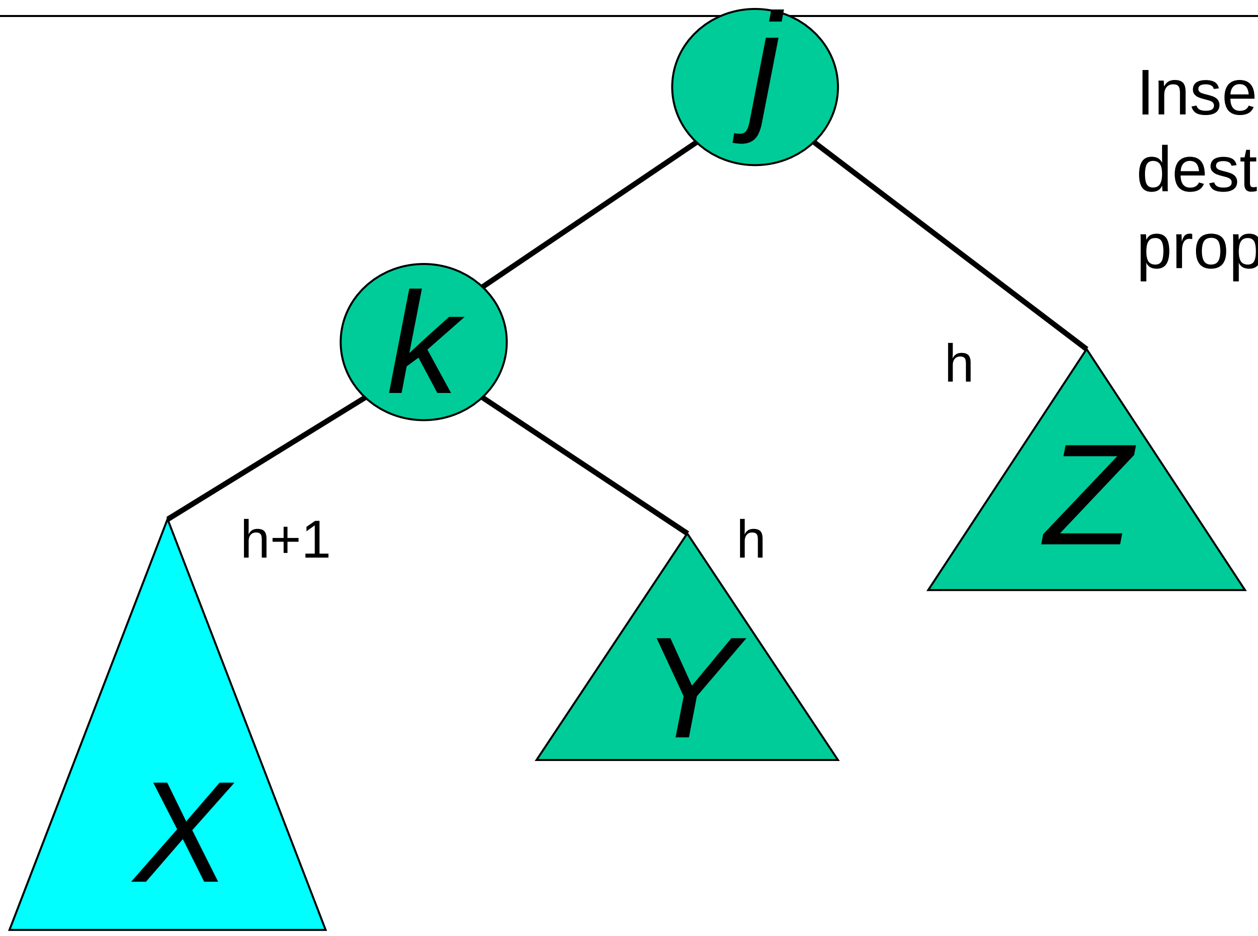
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid
AVL subtree



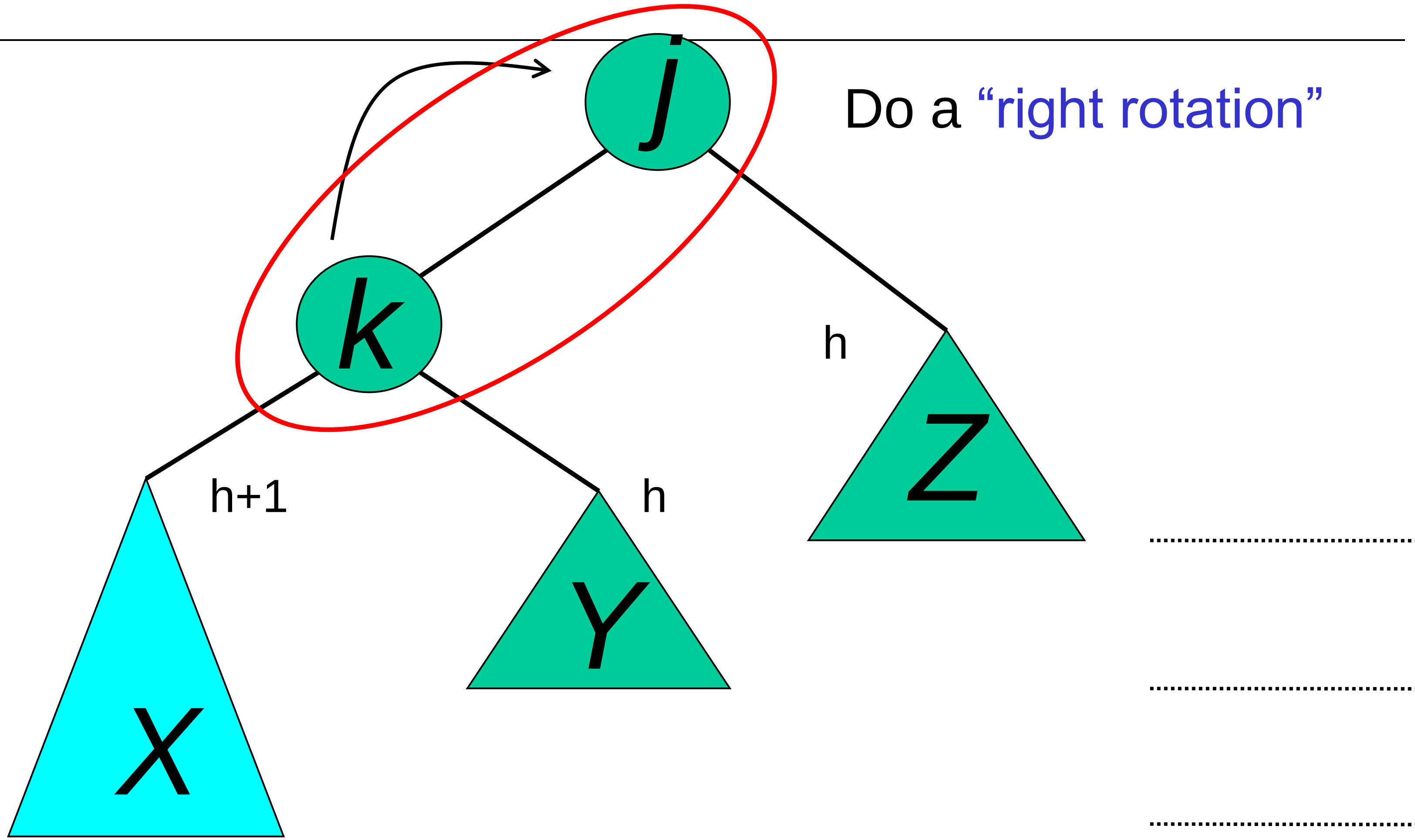
AVL Insertion: Outside Case



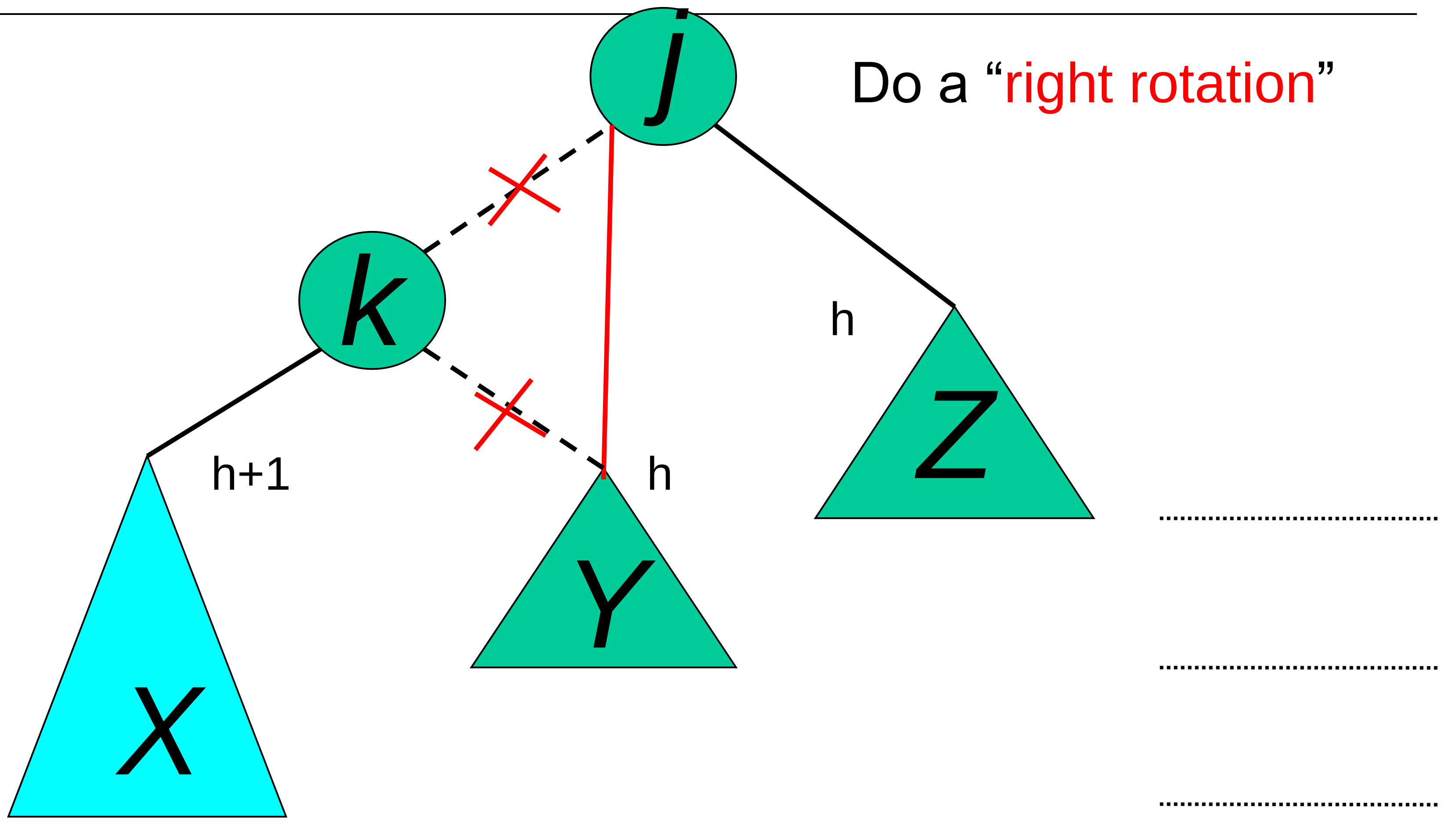
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case

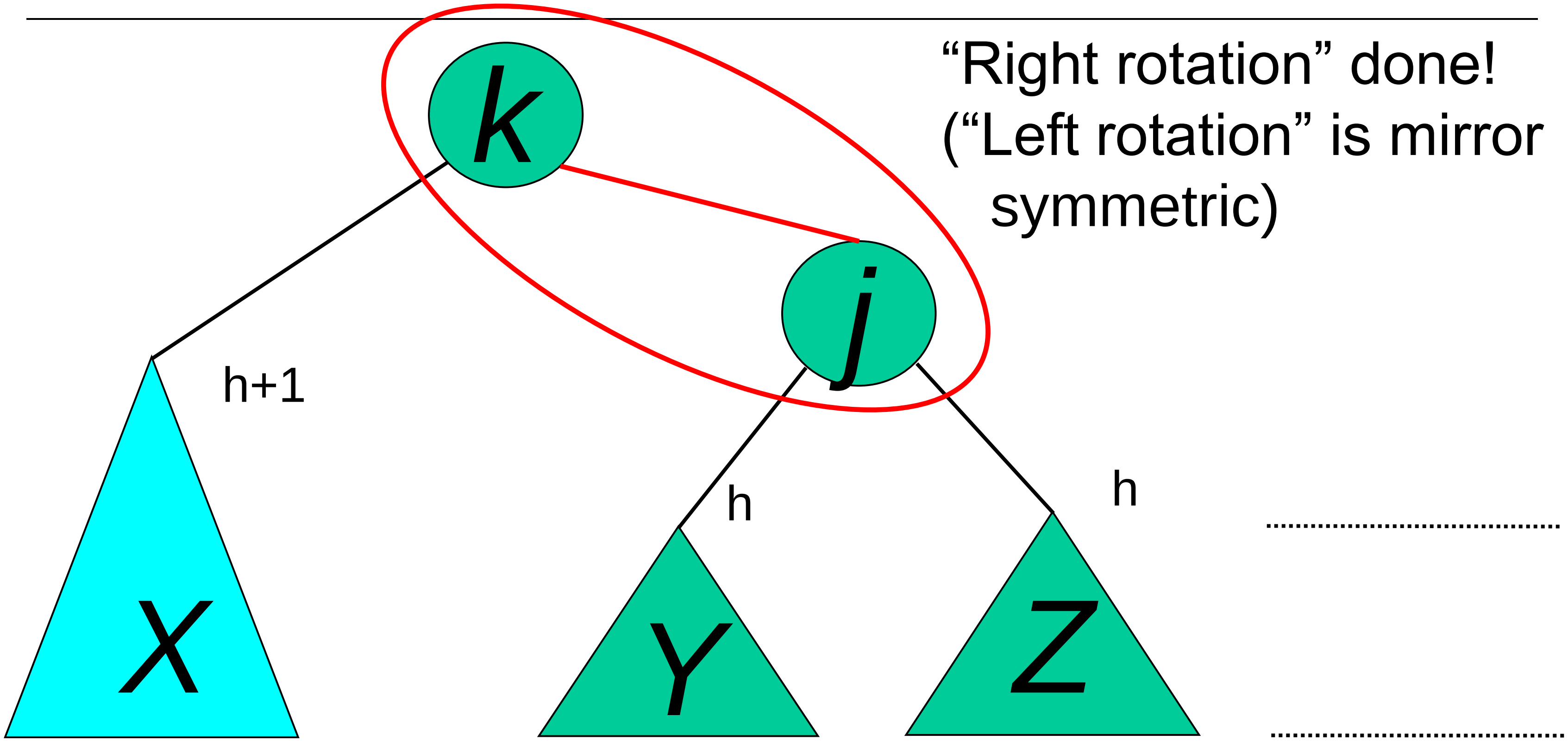
Do a "right rotation"



Single right rotation



Outside Case Completed

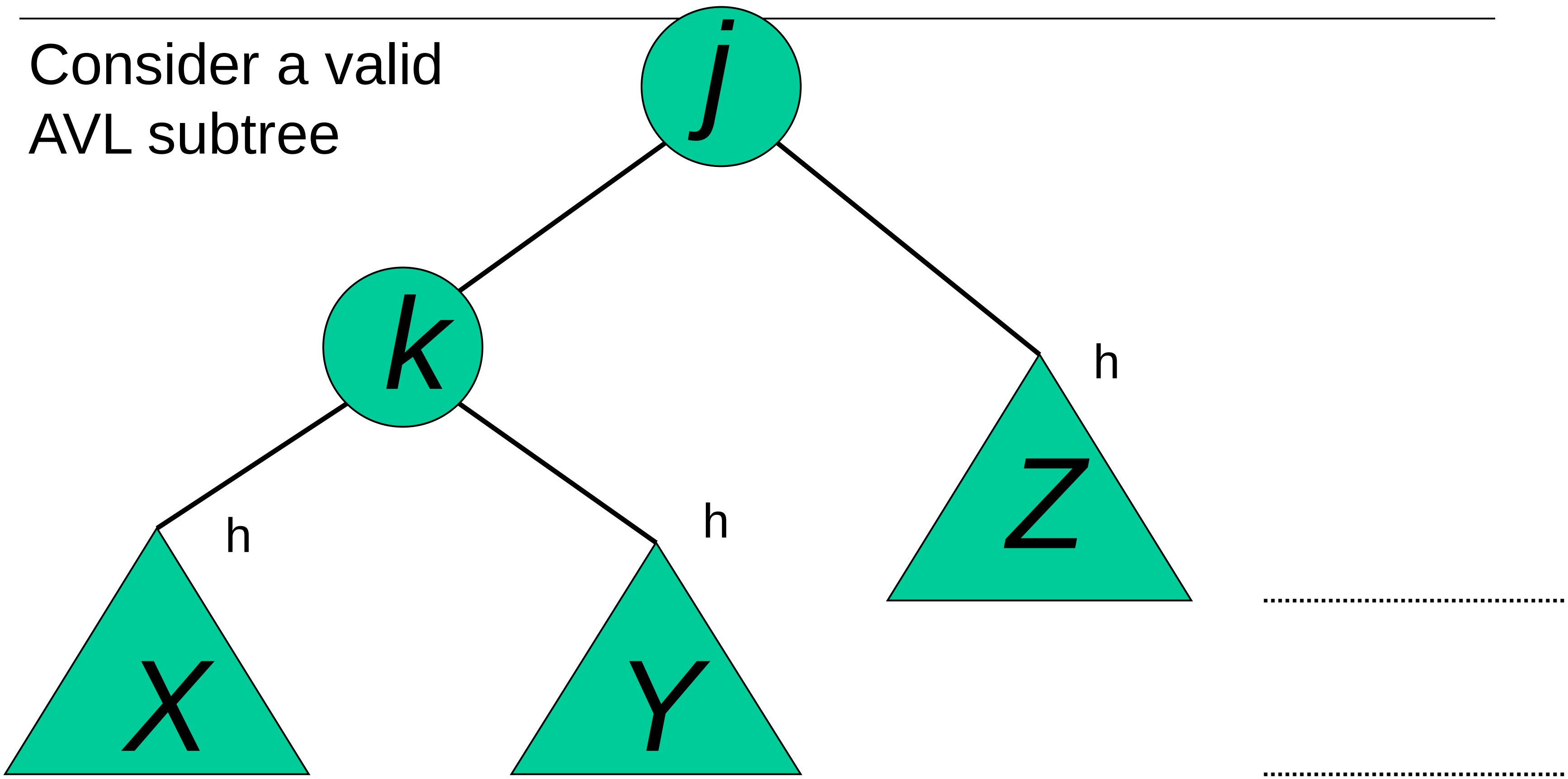


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

AVL Insertion: Inside Case

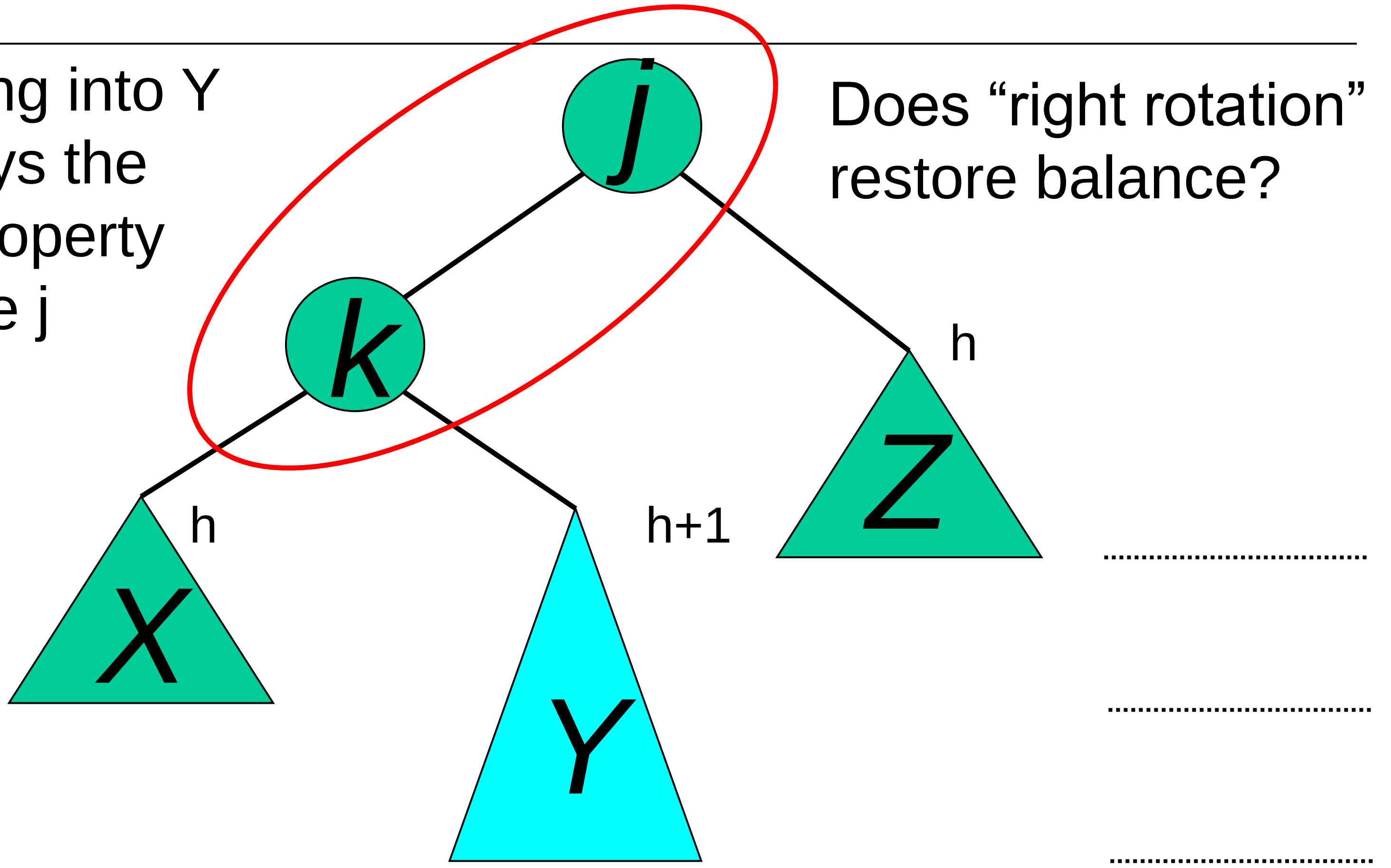
Consider a valid AVL subtree



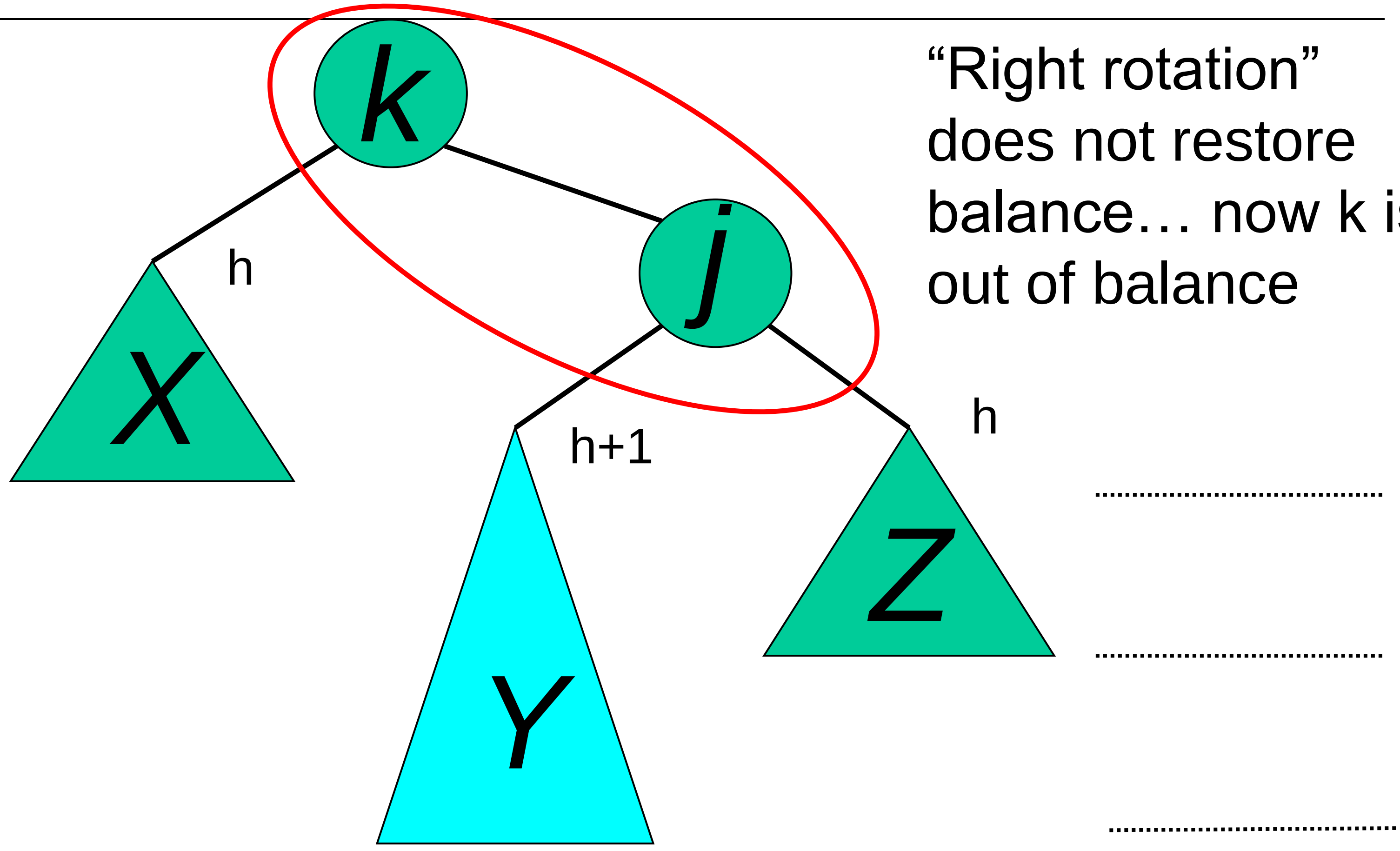
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j

Does “right rotation” restore balance?



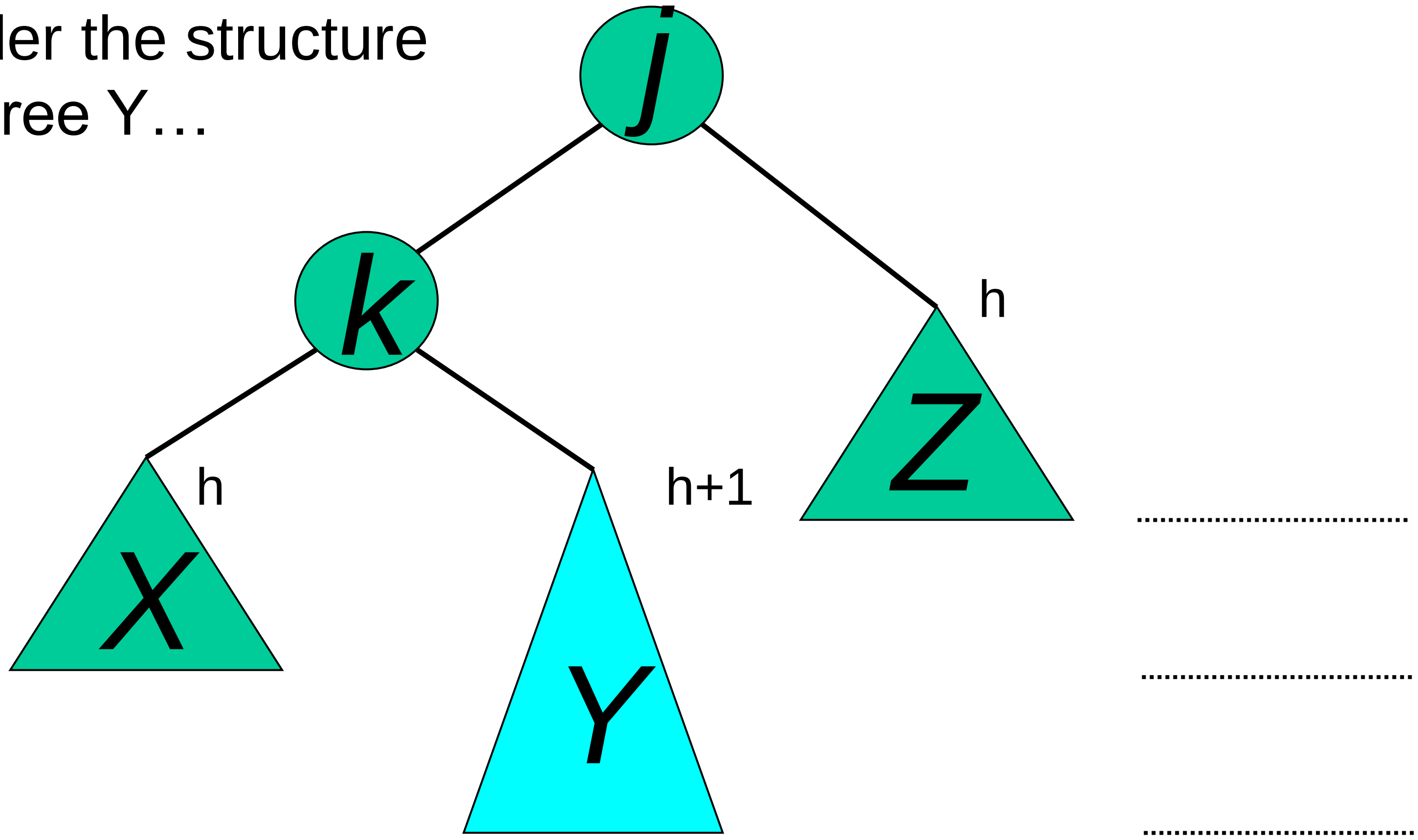
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now k is out of balance

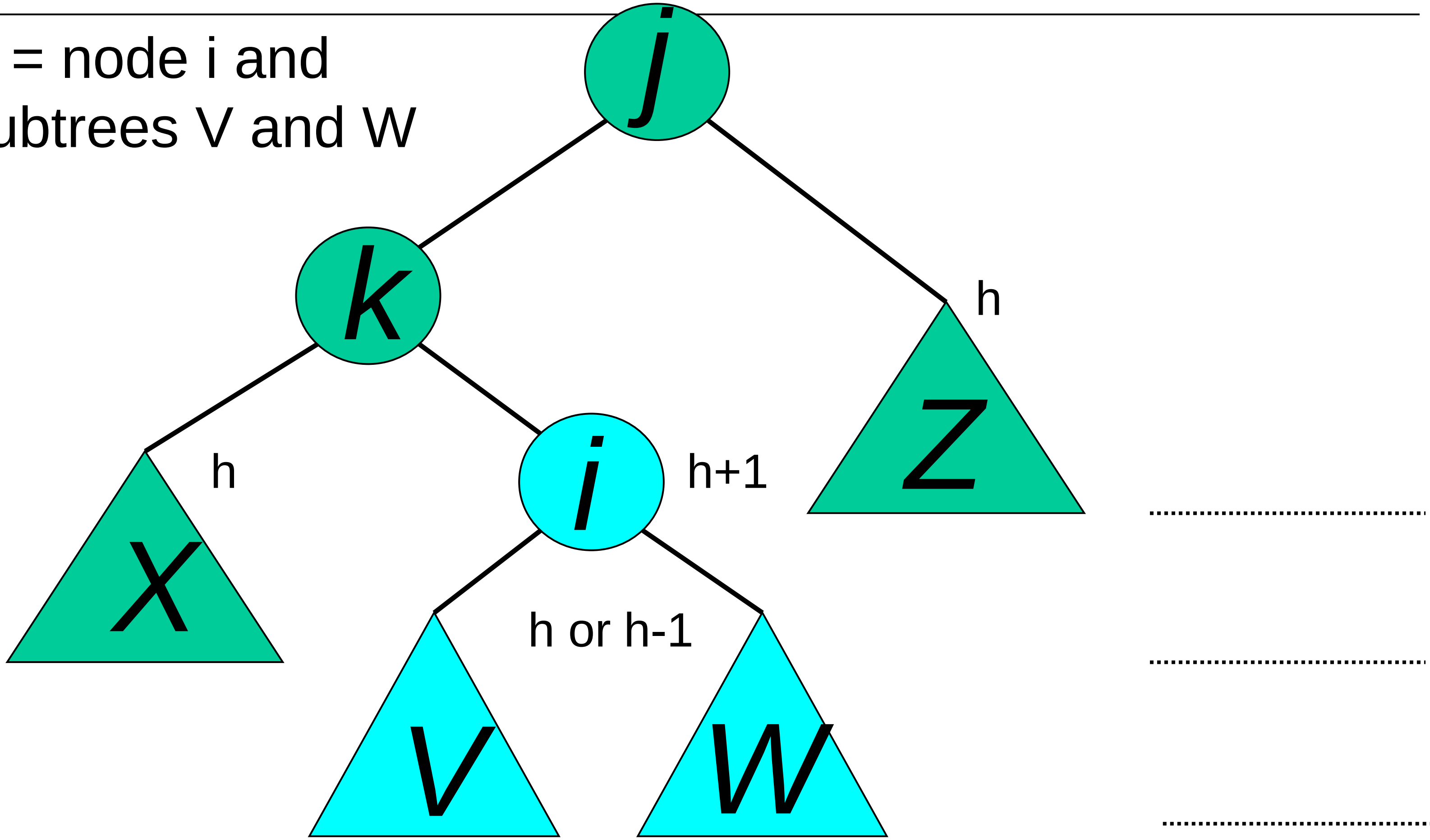
AVL Insertion: Inside Case

Consider the structure of subtree Y…

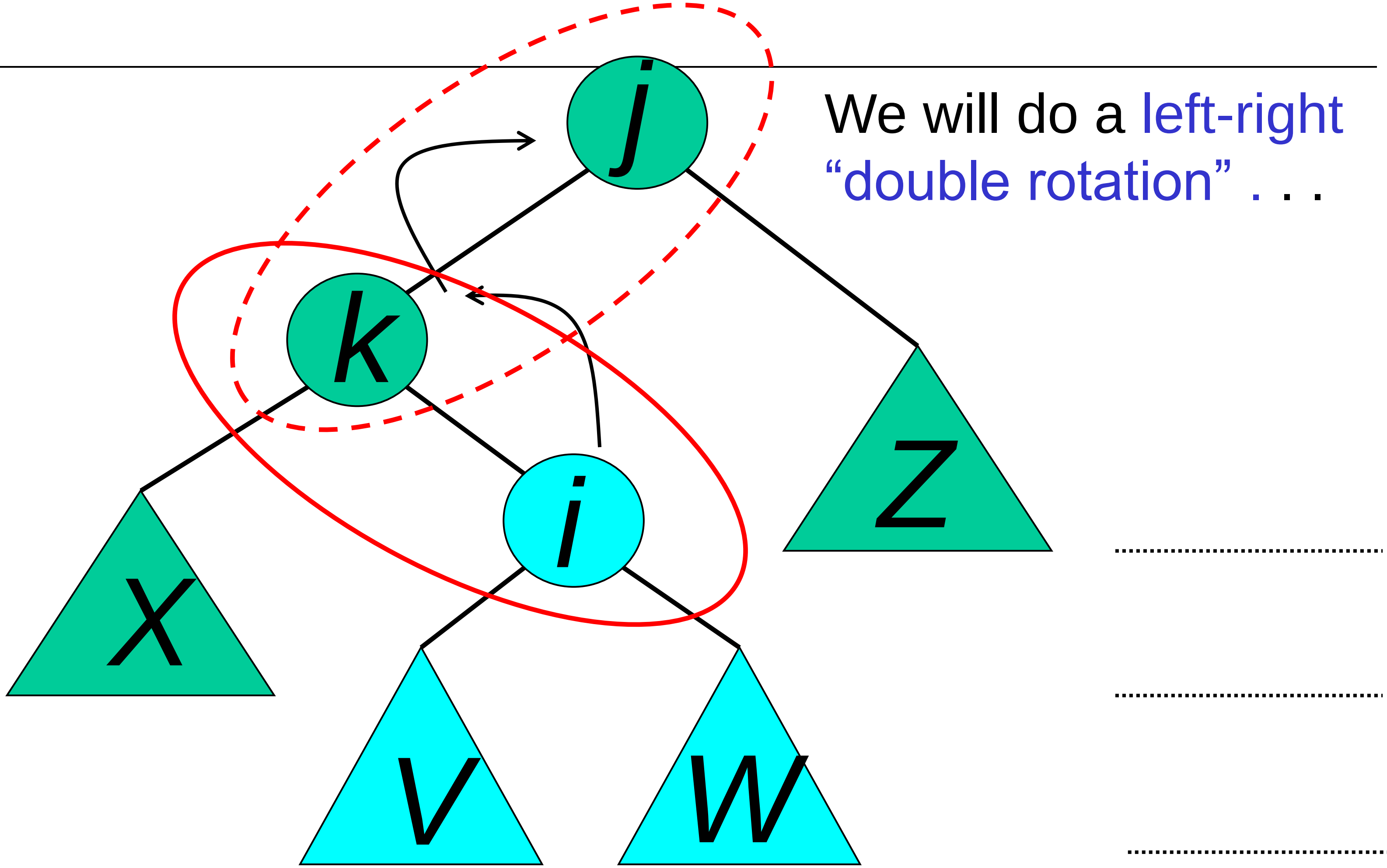


AVL Insertion: Inside Case

Y = node i and
subtrees V and W



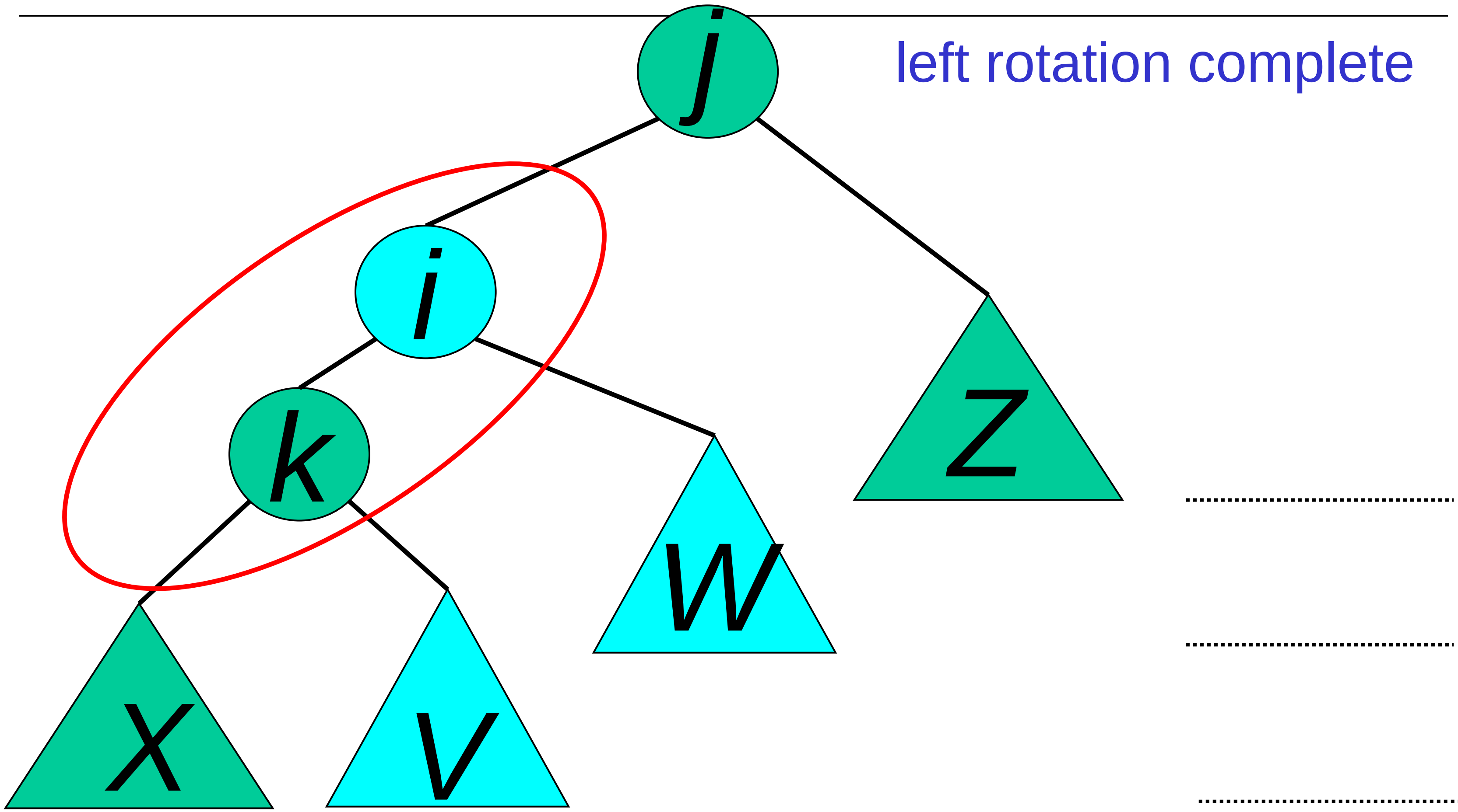
AVL Insertion: Inside Case



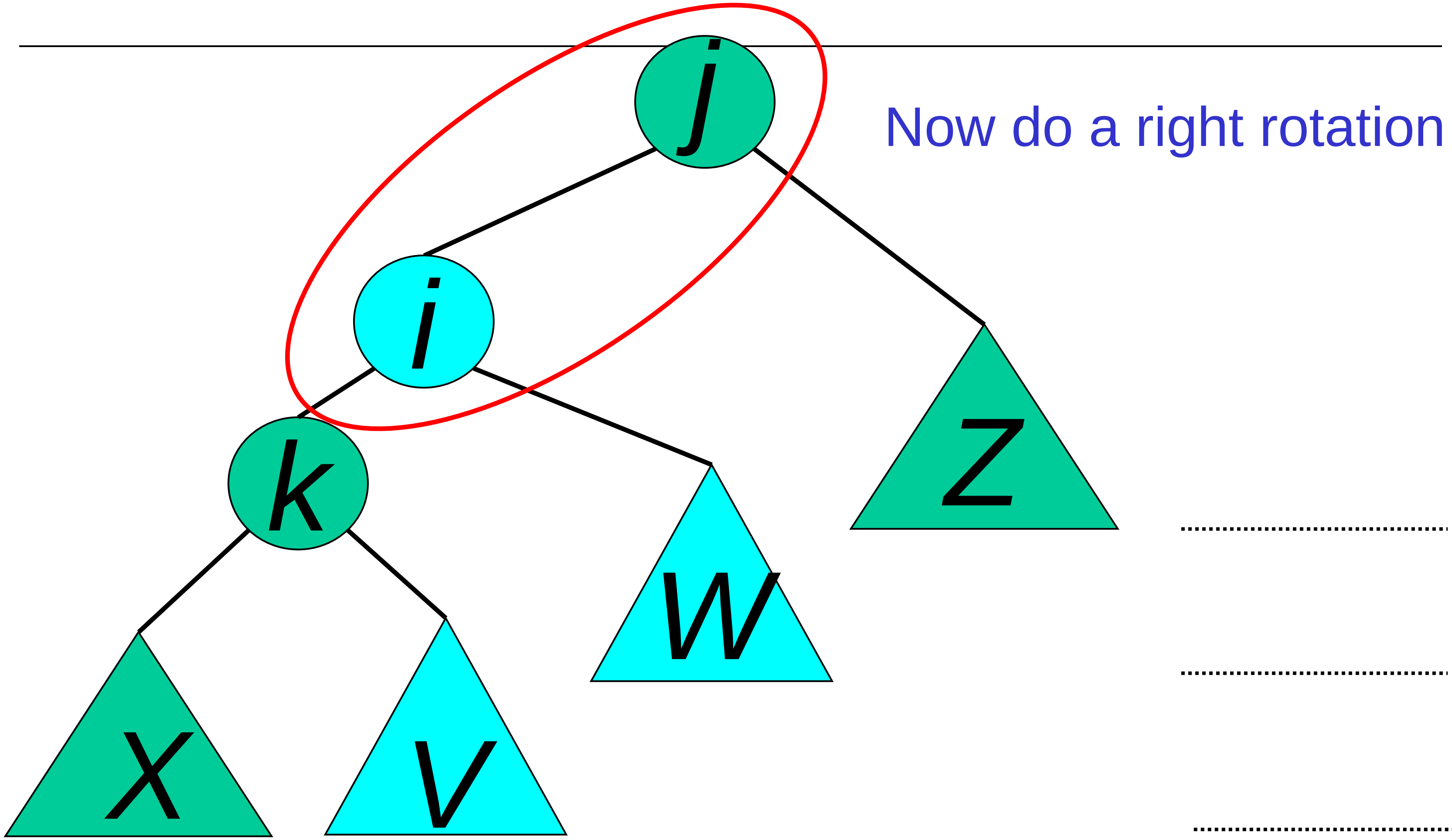
We will do a left-right "double rotation" . . .

Double rotation : first rotation

left rotation complete



Double rotation : second rotation

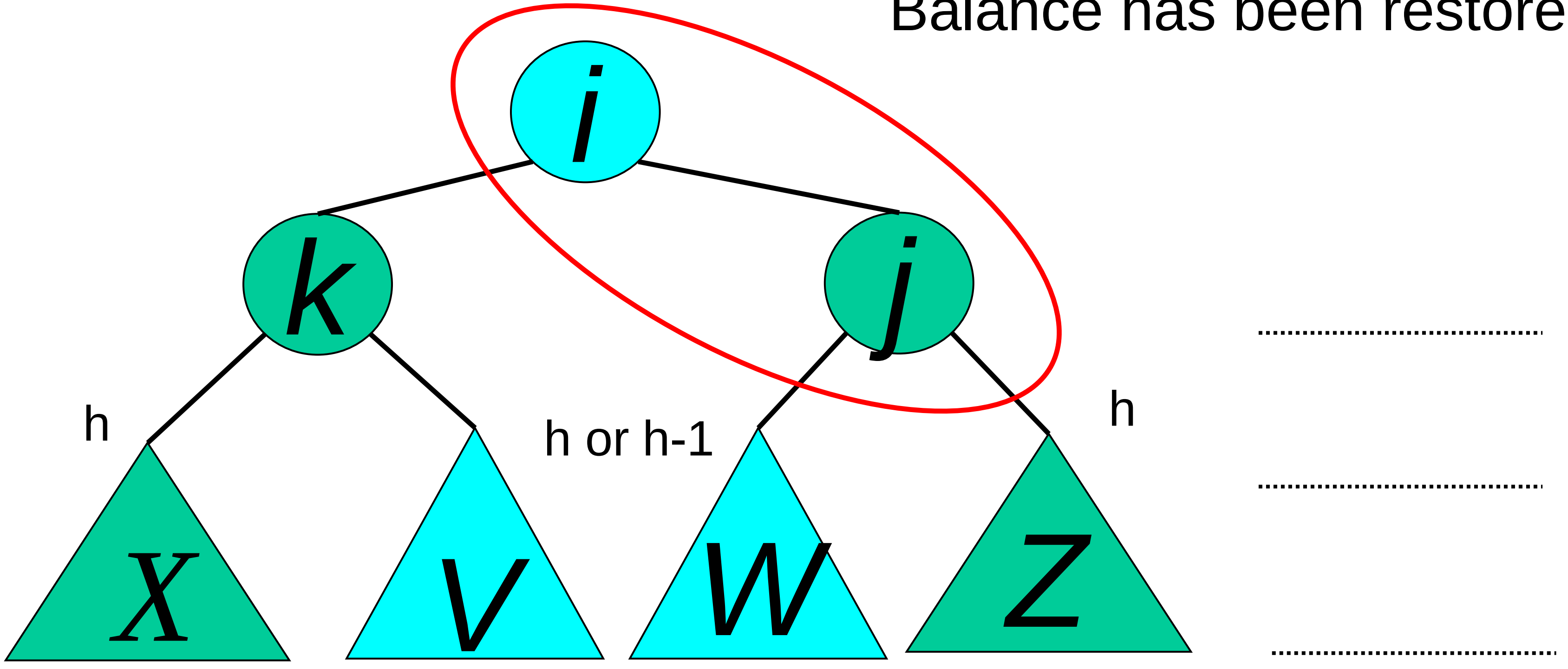


Now do a right rotation

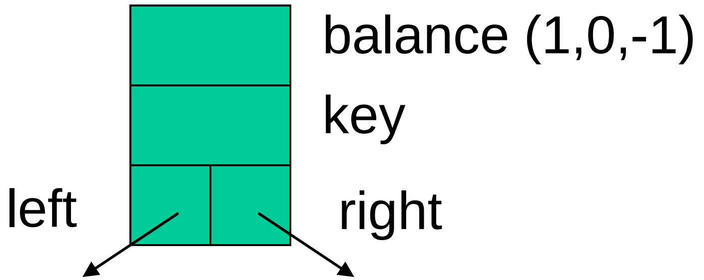
Double rotation : second rotation

right rotation complete

Balance has been restored



Implementation

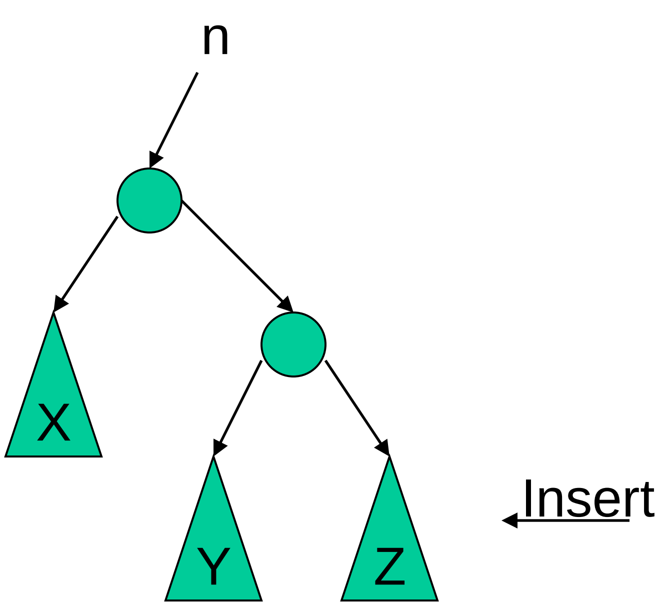


No need to keep the height; just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don't perform rotations

Once you have performed a rotation (single or double) you won't need to go back up the tree

Single Rotation

```
RotateFromRight(n : reference node pointer) {
p : node pointer;
p := n.right;
n.right := p.left;
p.left := n;
n := p
}
```

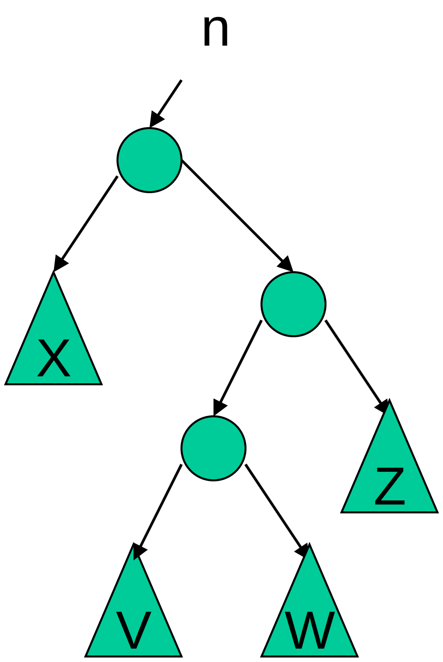


You also need to modify the heights or balance factors of n and p

Double Rotation

- Implement Double Rotation in two lines.

```
DoubleRotateFromRight(n : reference node pointer)
{
????
}
```



Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

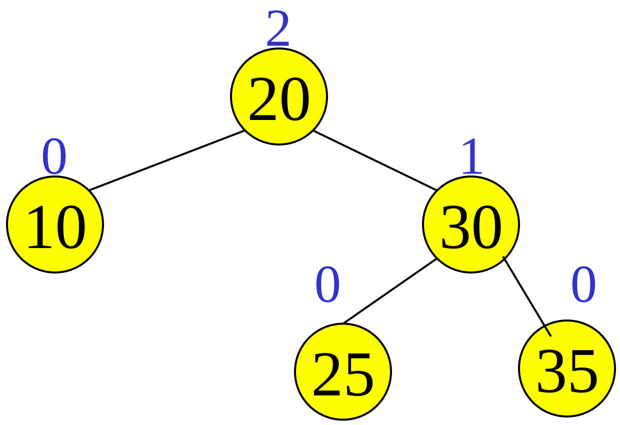
Insert in BST

```
Insert(T : reference tree pointer, x : element) : integer
{
if T = null then
  T := new tree; T.data := x; return 1;//the links to
                                        //children are null
case
  T.data = x : return 0; //Duplicate do nothing
  T.data > x : return Insert(T.left, x);
  T.data < x : return Insert(T.right, x);
endcase
}
```

Insert in AVL trees

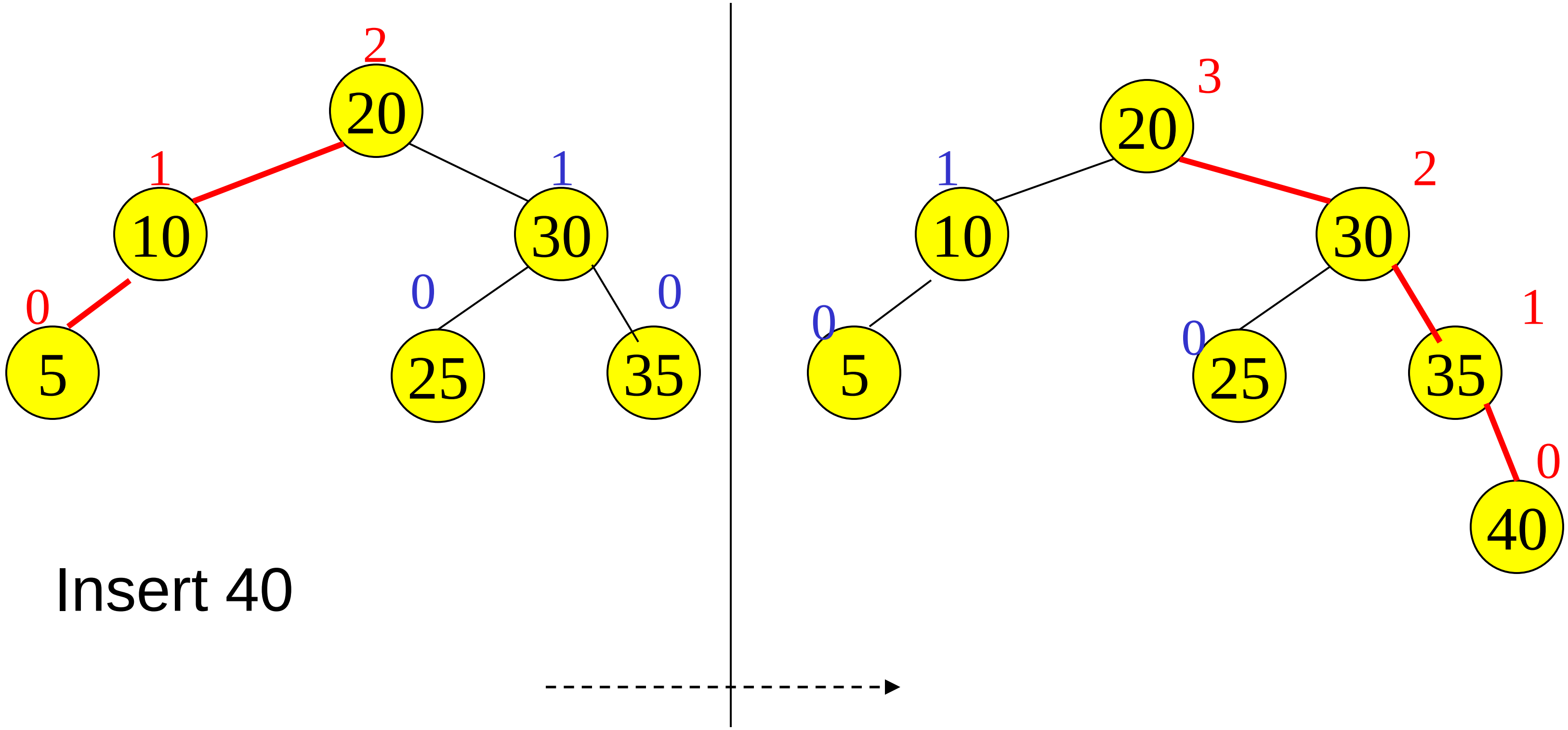
```
Insert(T : reference tree pointer, x : element) : {
if T = null then
  {T := new tree; T.data := x; height := 0; return;}
case
  T.data = x : return ; //Duplicate do nothing
  T.data > x : Insert(T.left, x);
               if ((height(T.left)- height(T.right)) = 2){
                  if (T.left.data > x ) then //outside case
                         T = RotatefromLeft (T);
                  else                       //inside case
                         T = DoubleRotatefromLeft (T);}
  T.data < x :  Insert(T.right, x);
                code similar to the left case
Endcase
  T.height := max(height(T.left),height(T.right)) +1;
  return;
}
```

Example of Insertions in an AVL Tree

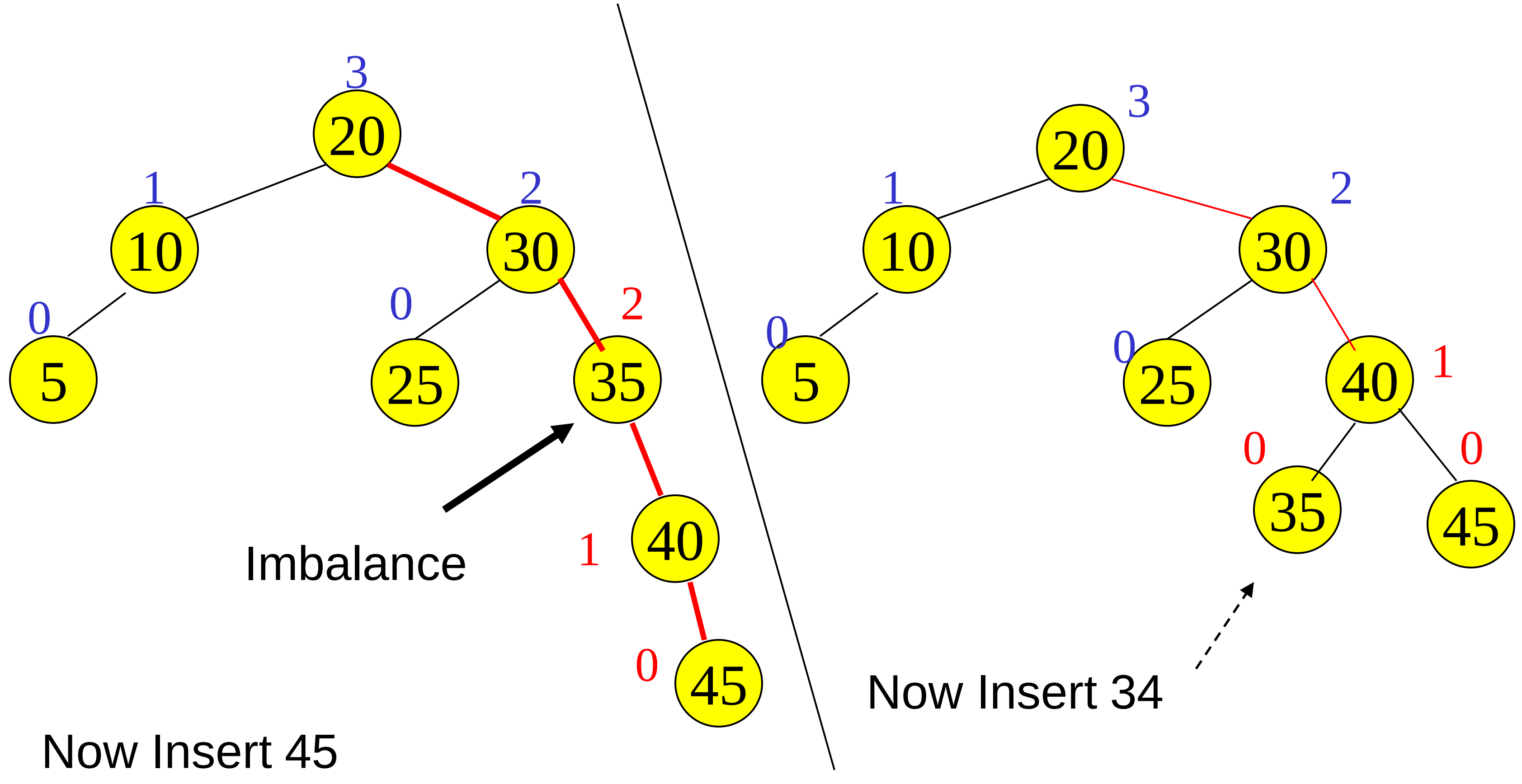


Insert 5

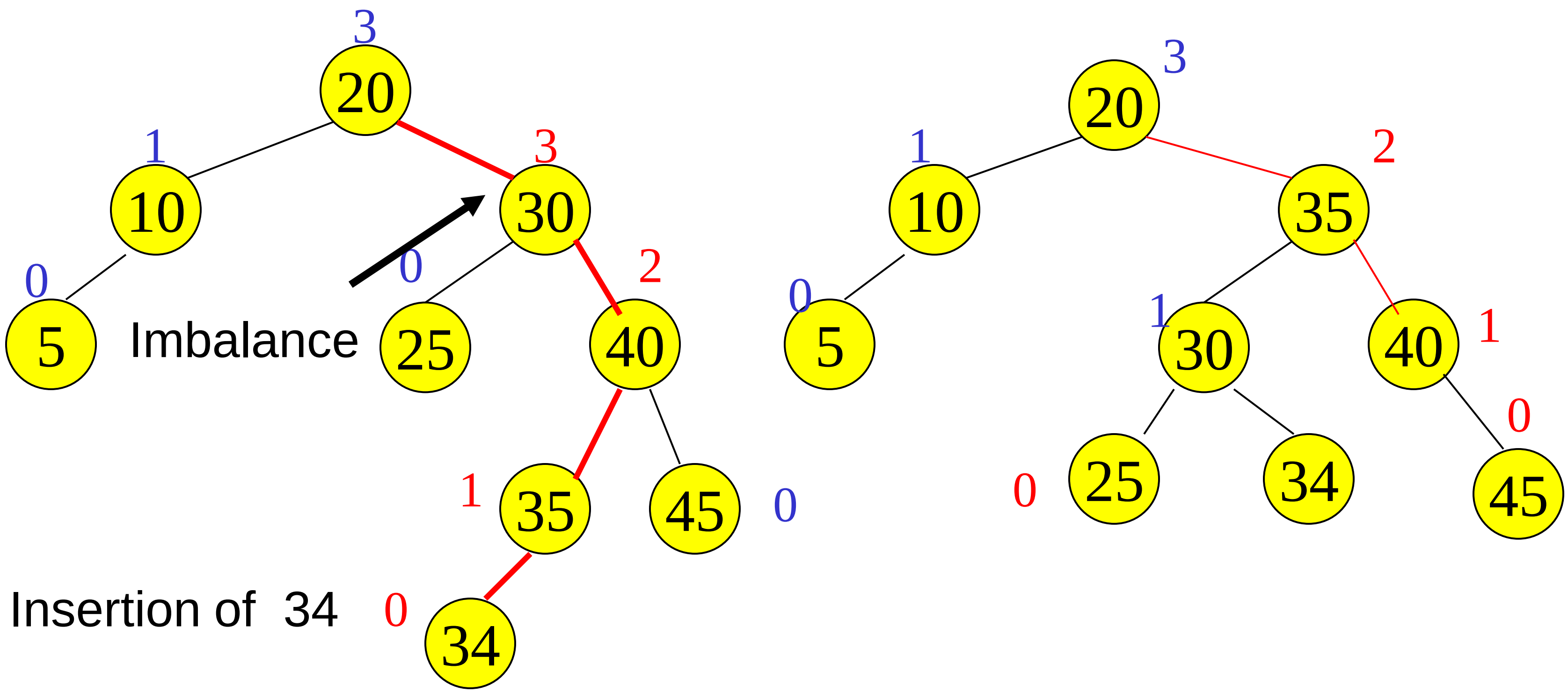
Example of Insertions in an AVL Tree



Single rotation (outside case)



Double rotation (inside case)



AVL Tree Deletion

- Similar but more complex than insertion
  - › Rotations and double rotations needed to rebalance
  - › Imbalance may propagate upward so that many rotations may be needed.

Splay Trees

Self adjusting Trees

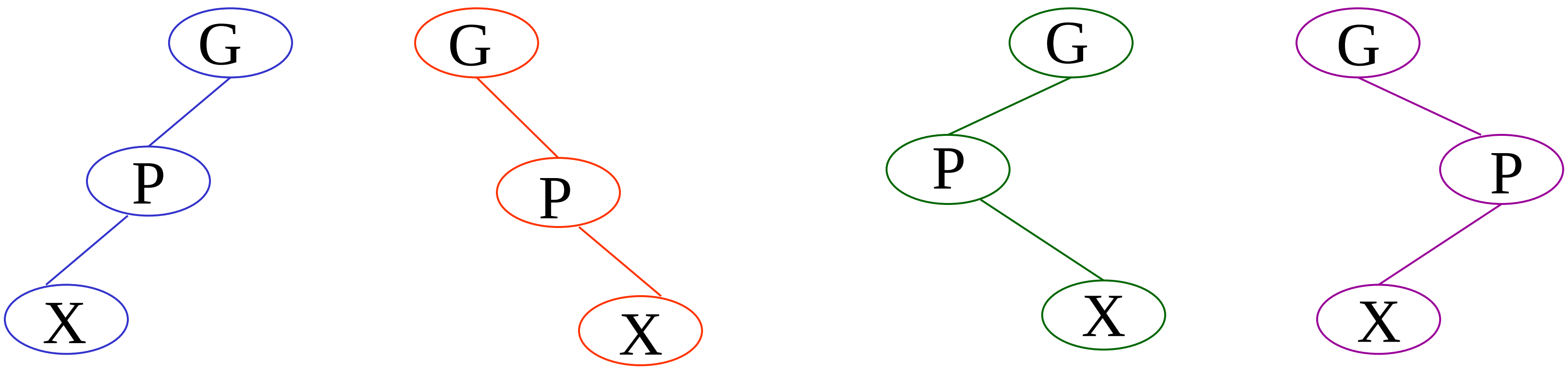
- Ordinary binary search trees have no balance conditions
  - what you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
  - tree is always balanced after an insert or delete
- Self-adjusting trees get reorganized over time as nodes are accessed
  - Tree adjusts after insert, delete, or find

Splay Trees

- Splay trees are tree structures that:
  - › Are not perfectly balanced all the time
  - › Data most recently accessed is near the root. (principle of locality; 80-20 “rule”)
- The procedure:
  - › After node X is accessed, perform “splaying” operations to bring X to the root of the tree.
  - › Do this in a way that leaves the tree more balanced as a whole

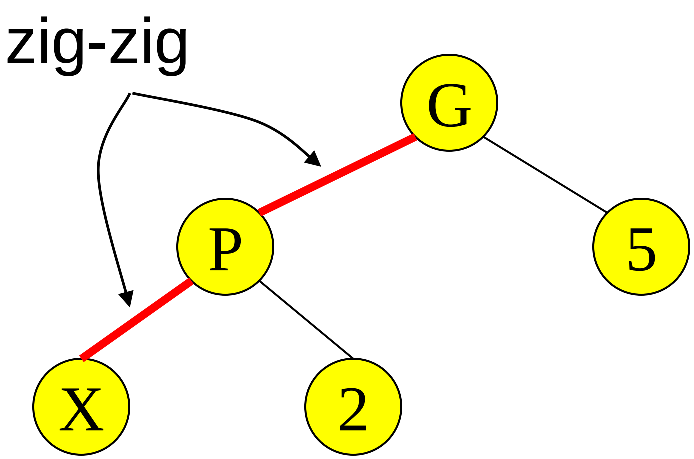
Splay Tree Terminology

- Let X be a non-root node with ≥ 2 ancestors.
  - P is its parent node.
  - G is its grandparent node.

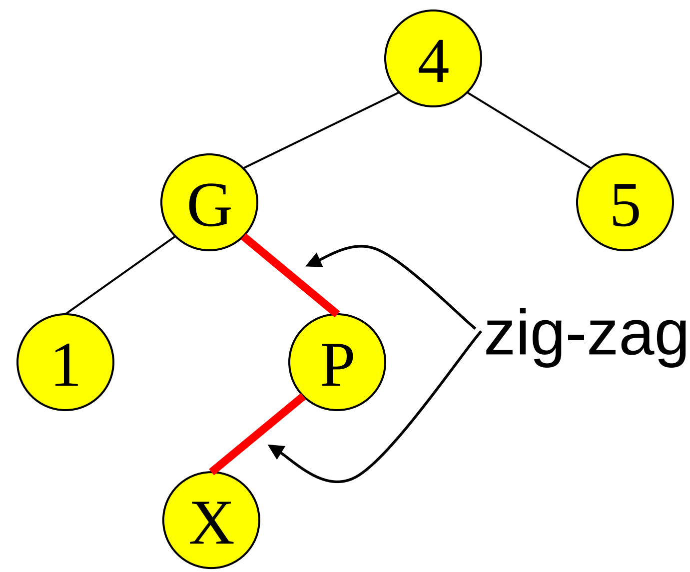


Zig-Zig and Zig-Zag

Parent and grandparent in same direction.

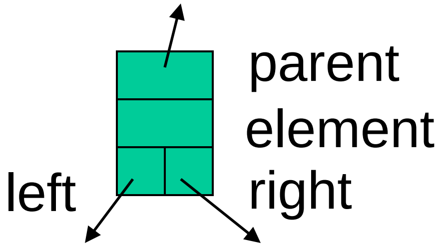


Parent and grandparent in different directions.



Splay Tree Operations

1. Helpful if nodes contain a parent pointer.

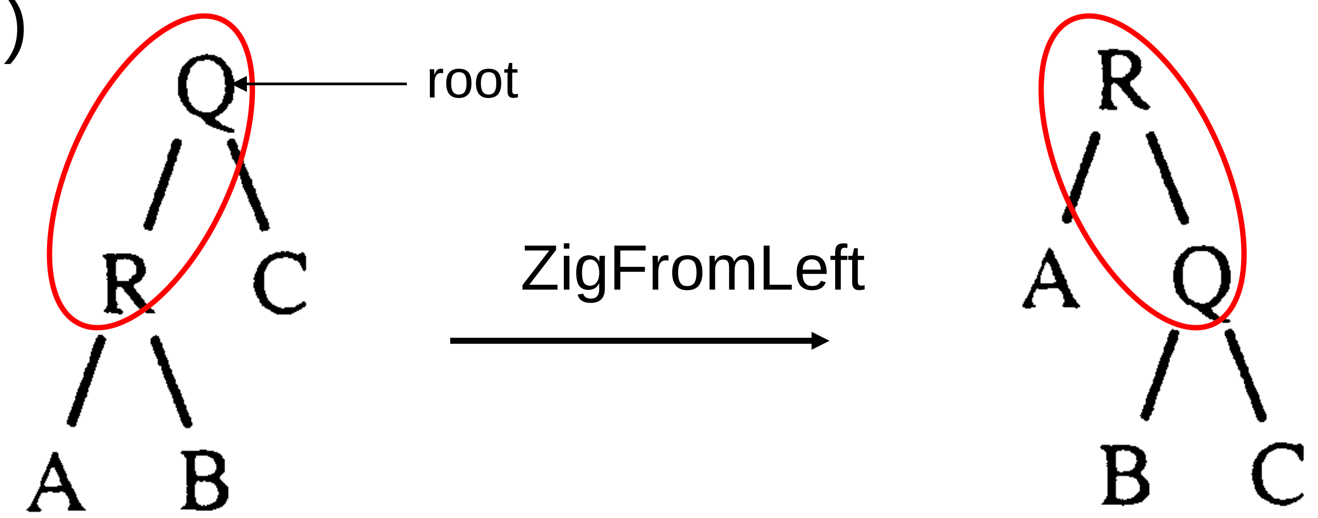


2. When X is accessed, apply one of six rotation routines.

- Single Rotations (X has a P (the root) but no G)
  ZigFromLeft, ZigFromRight
- Double Rotations (X has both a P and a G)
  ZigZigFromLeft, ZigZigFromRight
  ZigZagFromLeft, ZigZagFromRight

Zig at depth 1 (root)

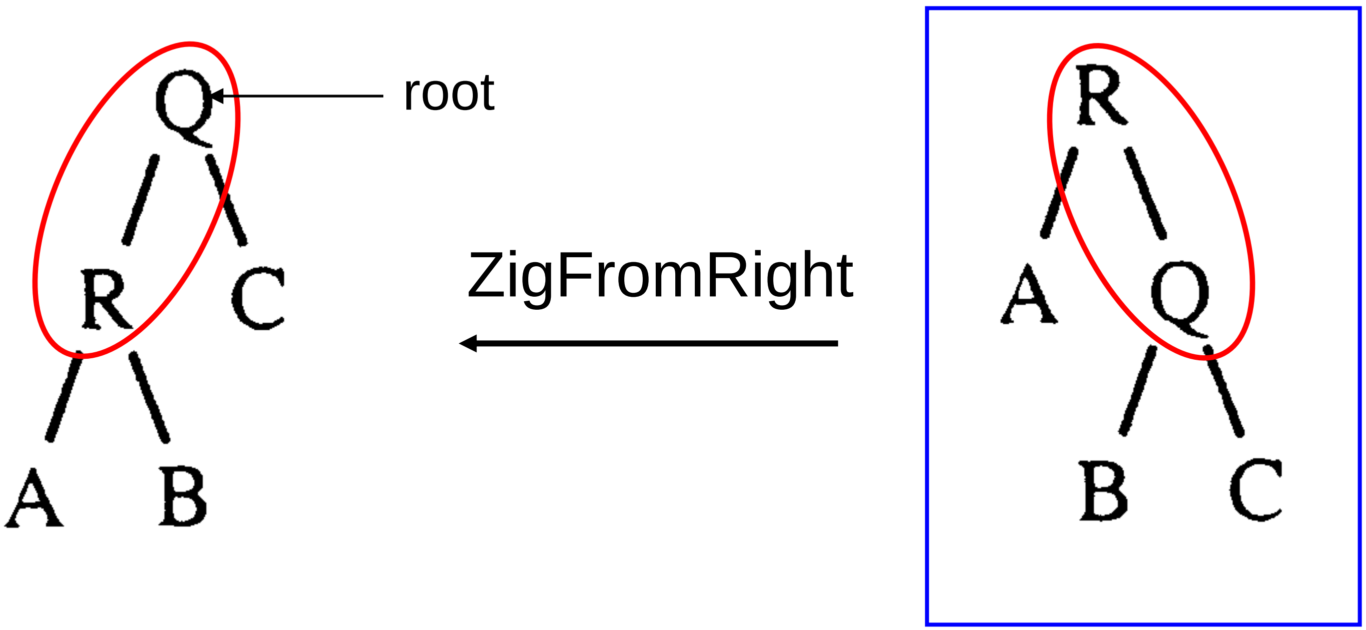
- "Zig" is just a single rotation, as in an AVL tree
- Let R be the node that was accessed (e.g. using Find)



- ZigFromLeft moves R to the top →faster access next time

Zig at depth 1

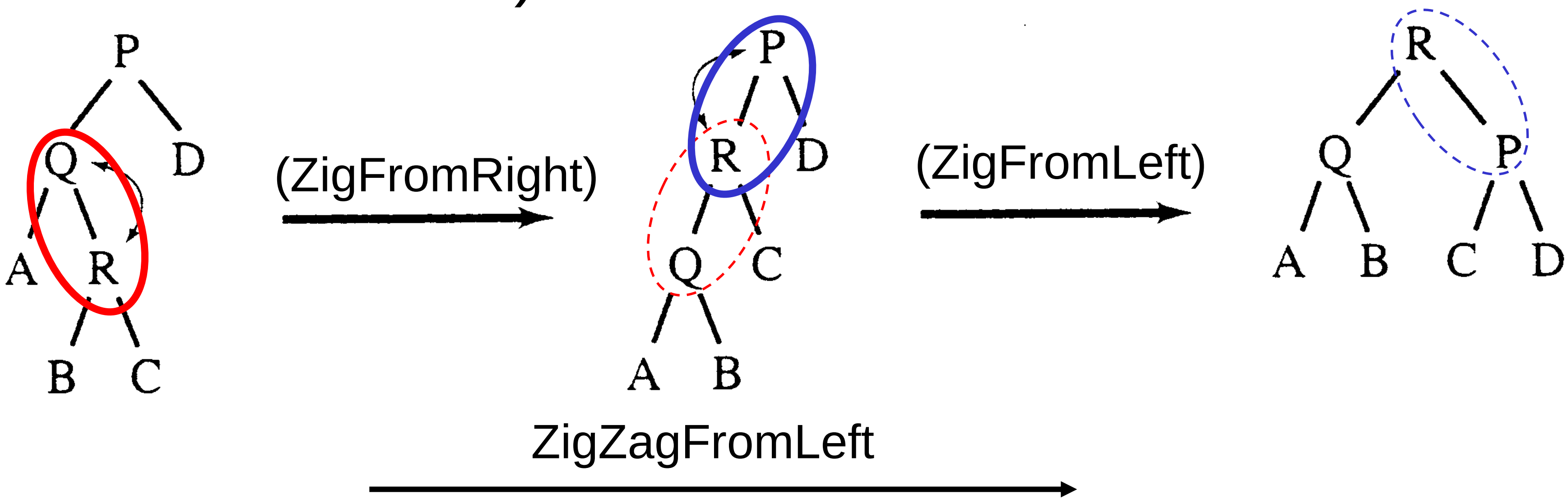
- Suppose Q is now accessed using Find



- ZigFromRight moves Q back to the top

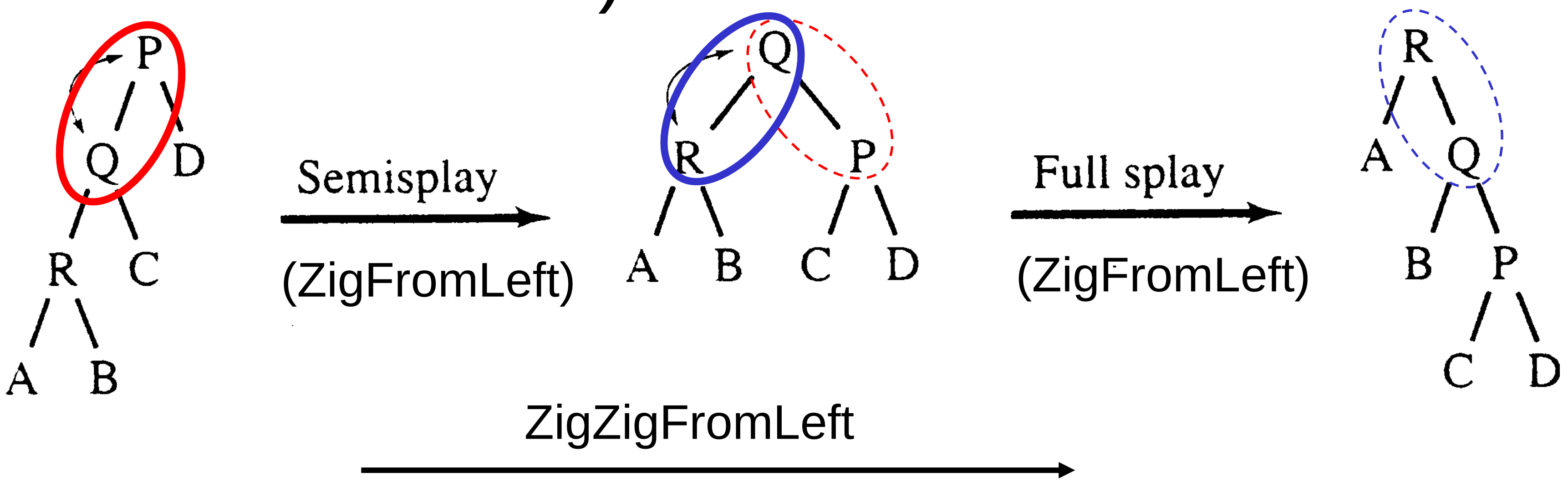
Zig-Zag operation

- "Zig-Zag" consists of two rotations of the opposite direction (assume R is the node that was accessed)

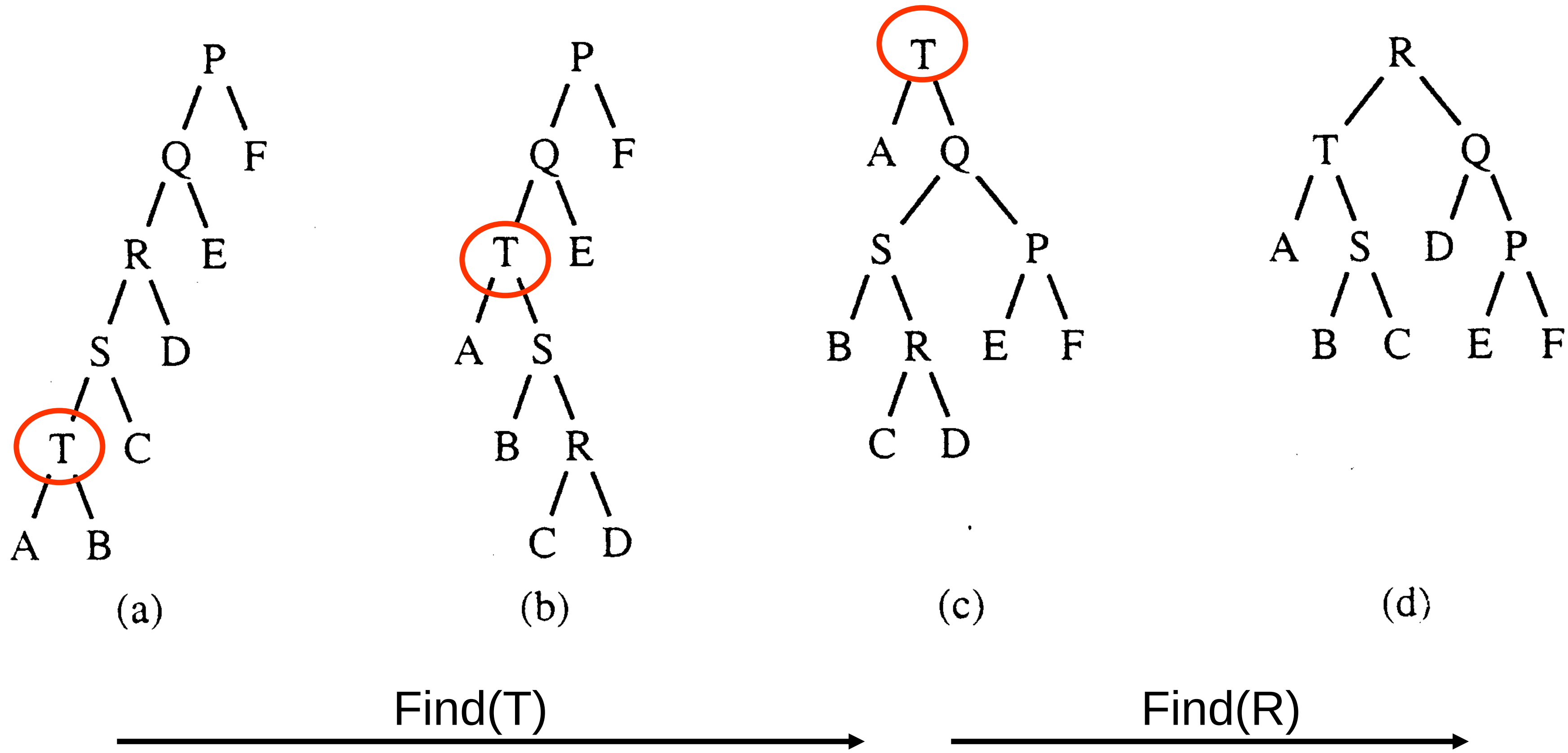


Zig-Zig operation

- "Zig-Zig" consists of two single rotations of the same direction (R is the node that was accessed)



Decreasing depth - "autobalance"

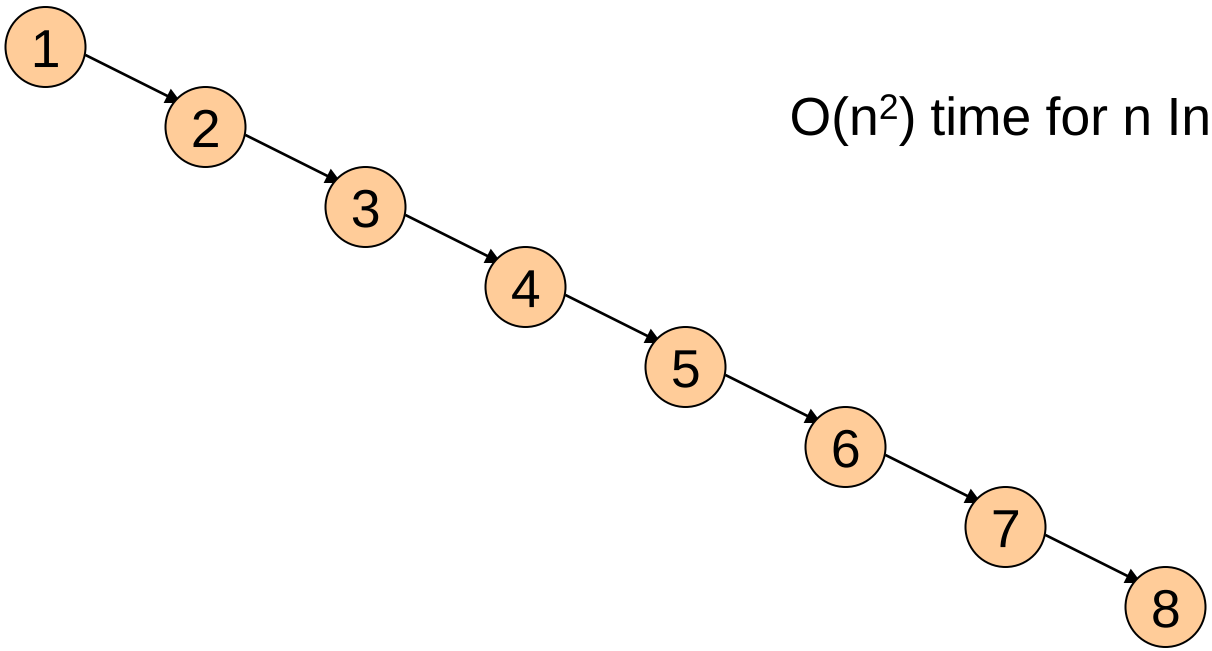


Splay Tree Insert and Delete

- Insert x
  - Insert x as normal then splay x to root.
- Delete x
  - Splay x to root and remove it. (note: the node does not have to be a leaf or single child node like in BST delete.) Two trees remain, right subtree and left subtree.
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree.

Example Insert

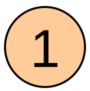
- Inserting in order 1,2,3,…,8
- Without self-adjustment



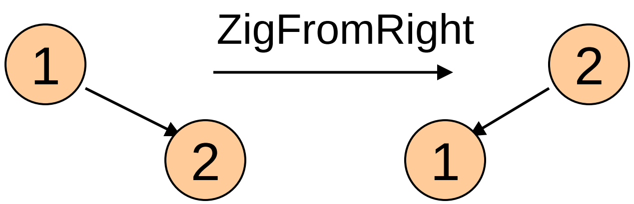
O(n^2) time for n Insert

With Self-Adjustment

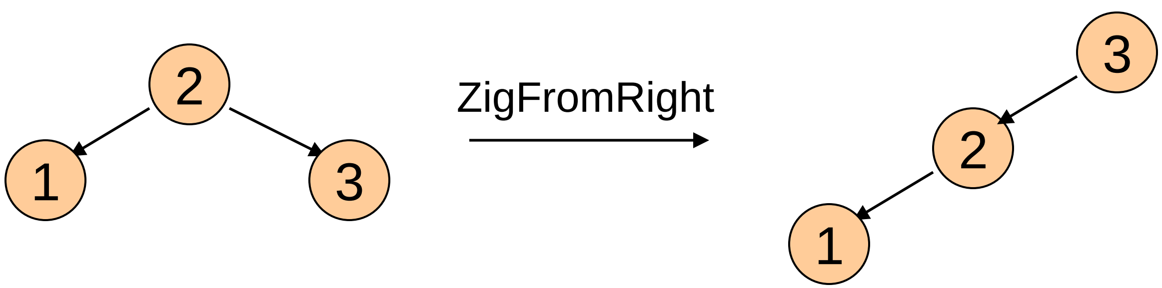
1



2

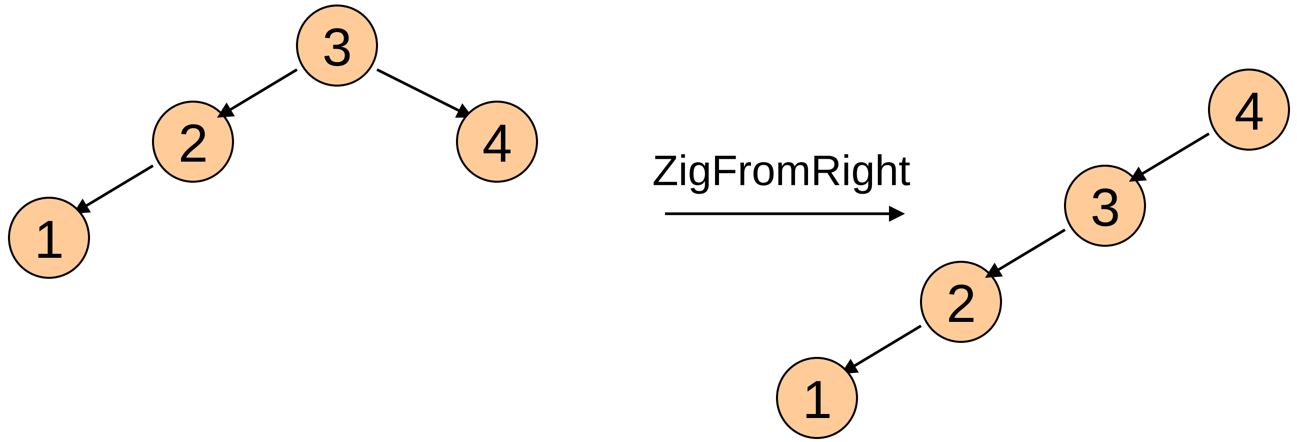


3



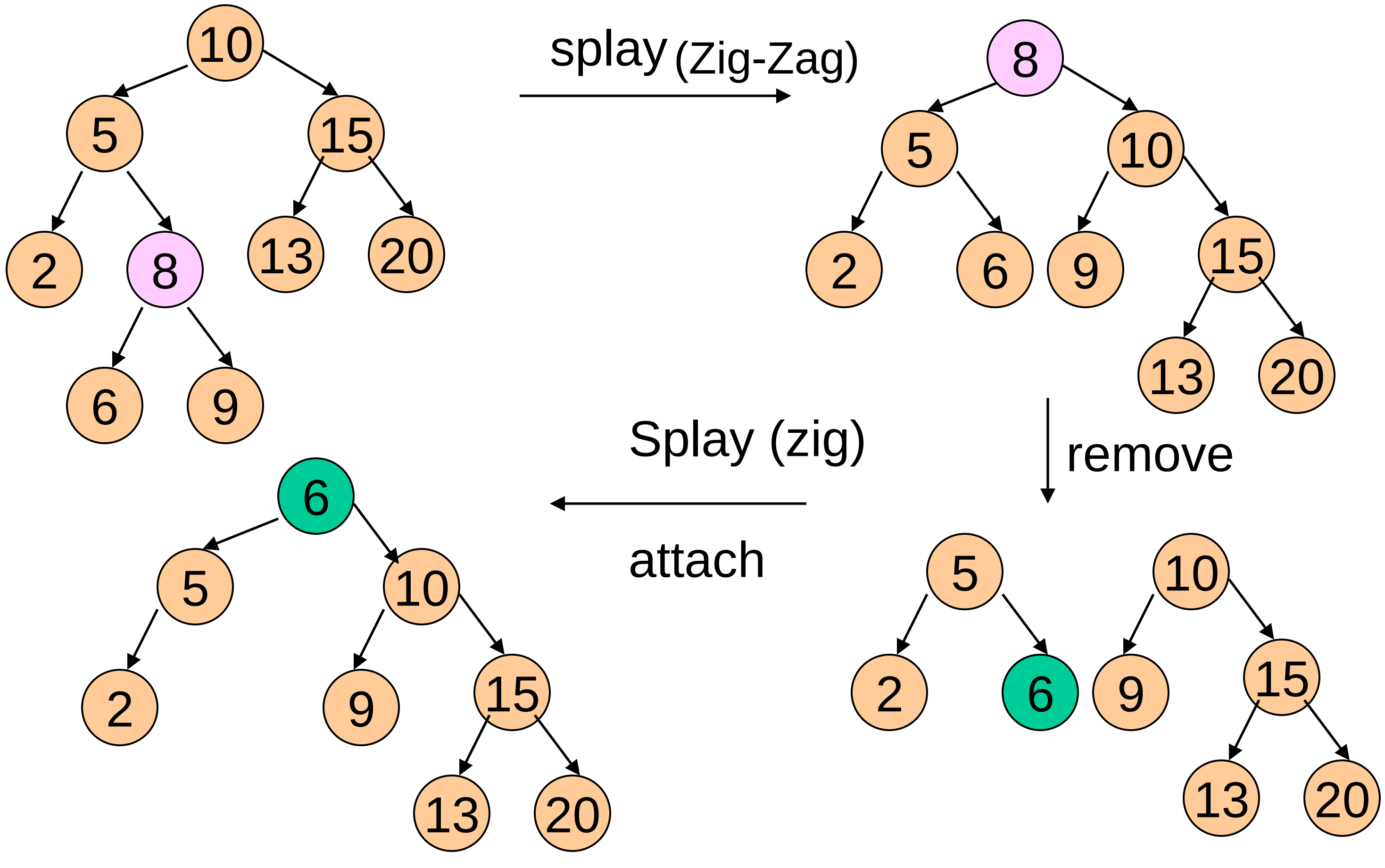
With Self-Adjustment

4



Each Insert takes O(1) time therefore O(n) time for n Insert!!

Example Deletion



Analysis of Splay Trees

- Splay trees tend to be balanced
  - › M operations takes time O(M log N) for M $\geq$ N operations on N items. (proof is difficult)
  - › Amortized O(log n) time.
- Splay trees have good “locality” properties
  - › Recently accessed items are near the root of the tree.
  - › Items near an accessed one are pulled toward the root.

Summary of Search Trees

- Problem with Binary Search Trees: Must keep tree balanced to allow fast access to stored items

- AVL trees: Insert/Delete operations keep tree balanced
- Splay trees: Repeated Find operations produce balanced trees
- Multi-way search trees (e.g. B-Trees):
  - More than two children per node allows shallow trees; all leaves are at the same depth.
  - Keeping tree balanced at all times.
  - Excellent for indexes in database systems.

Summary

1. 二叉树的定义

定义：是n（n≥0）个结点的有限集合，由一个根结点以及两棵互不相交的、分别称为左子树和右子树的二叉树组成。

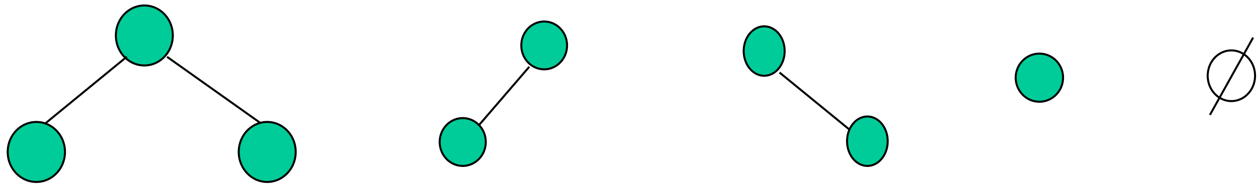
逻辑结构：　一对二（1：2）

基本特征:

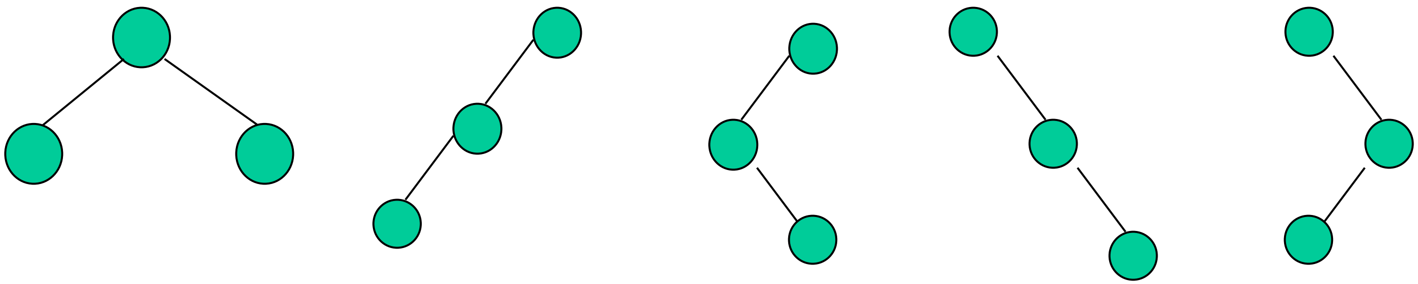
① 每个结点最多只有两棵子树（不存在度大于2的结点）；

② 左子树和右子树次序不能颠倒（有序树）。

基本形态:



具有3个结点的二叉树可能有几种不同形态?



5种

具有n个结点的二叉树可能有几种不同形态?

答：设具有n个结点的所有不同形态的二叉树有b[n]种，则 b[n]=C(2n,n)/(n + 1) (n=1,2,3,...)

证明如下：

考虑n个结点。除去根，剩下n-1个结点.对左子树有b[k]种方式。对右子树有b[n-1-k]种方式，由乘法原理，则 b[n]=sum[k=0…n-1](b[k]*b[n-1-k]) 由于b[1]=1

而这正好是Catalan数。

Catalan数（卡特兰数）：

令h(0)=1,h(1) = 1,catalan数满足递归式：

h(n)= h(0)*h(n-1) + h(1)*h(n-2) + ... + h(n-1)h(0) (其中n>=2)

该递推关系的解为：

h(n)=C(2n,n)/(n + 1) (n=1,2,3,...)

2. 二叉树的性质

性质1：在二叉树的第i层上至多有2^i个结点（i>=0）。

性质2：高度为k的二叉树至多有2^k-1个结点（k>0）。

性质3：具有k个节点的完全二叉树的高度为$\lceil \log_2(k+1) \rceil$（k>=0）。

问：高度为9的二叉树中至少有______个结点。

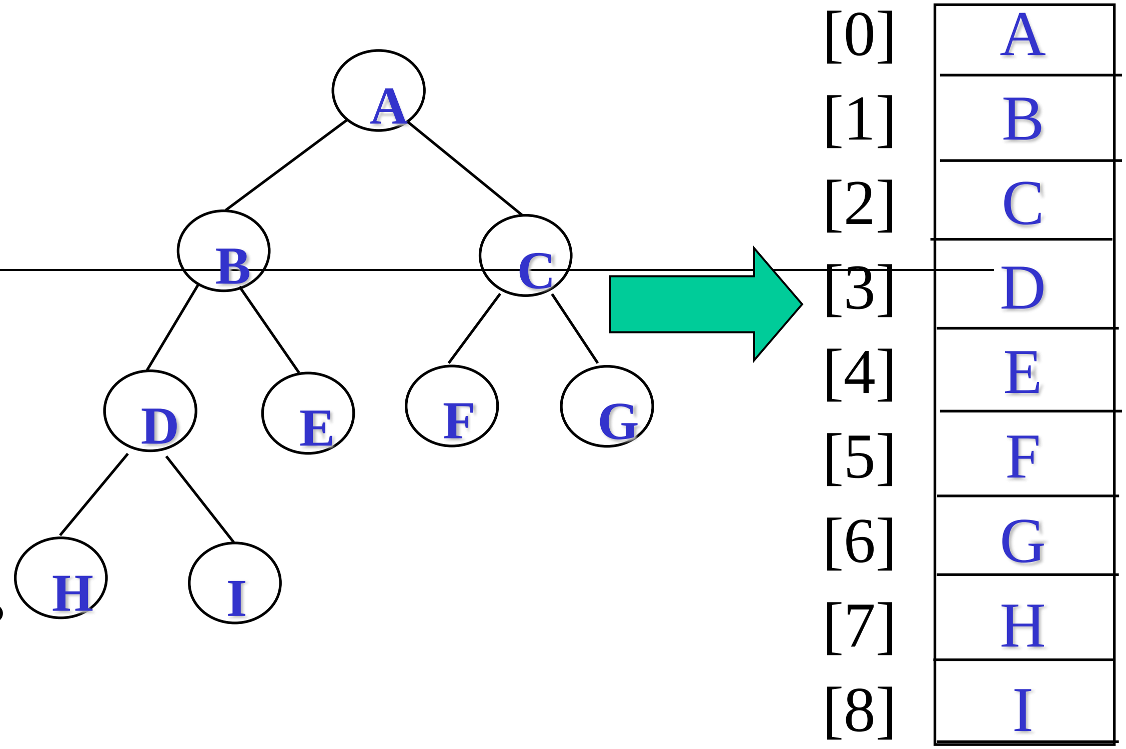
A）2^9　　B）2^8　　C）9　　D）2^9-1

答案：C

3. 二叉树的存储结构

一、顺序存储结构

按二叉树的结点“自上而下、从左至右”编号，用一组连续的存储单元存储。



问：顺序存储后能否复原成唯一对应的二叉树形状？

答：若是完全二叉树则可以做到唯一复原。

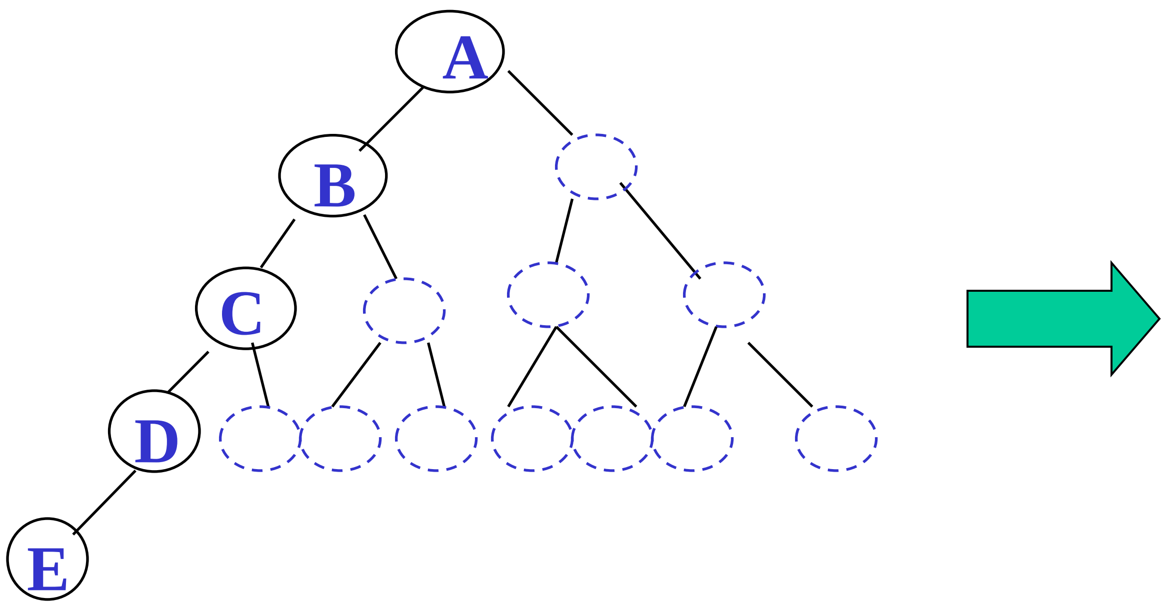
而且有规律：下标值为i的双亲，其左孩子的下标值必为2i+1，其右孩子的下标值必为2i＋2（即性质5）

例如，对应[2]的两个孩子必为[5]和[6],即C的左孩子必是F,右孩子必为G。

不是完全二叉树怎么办?

答：**一律转为完全二叉树!**

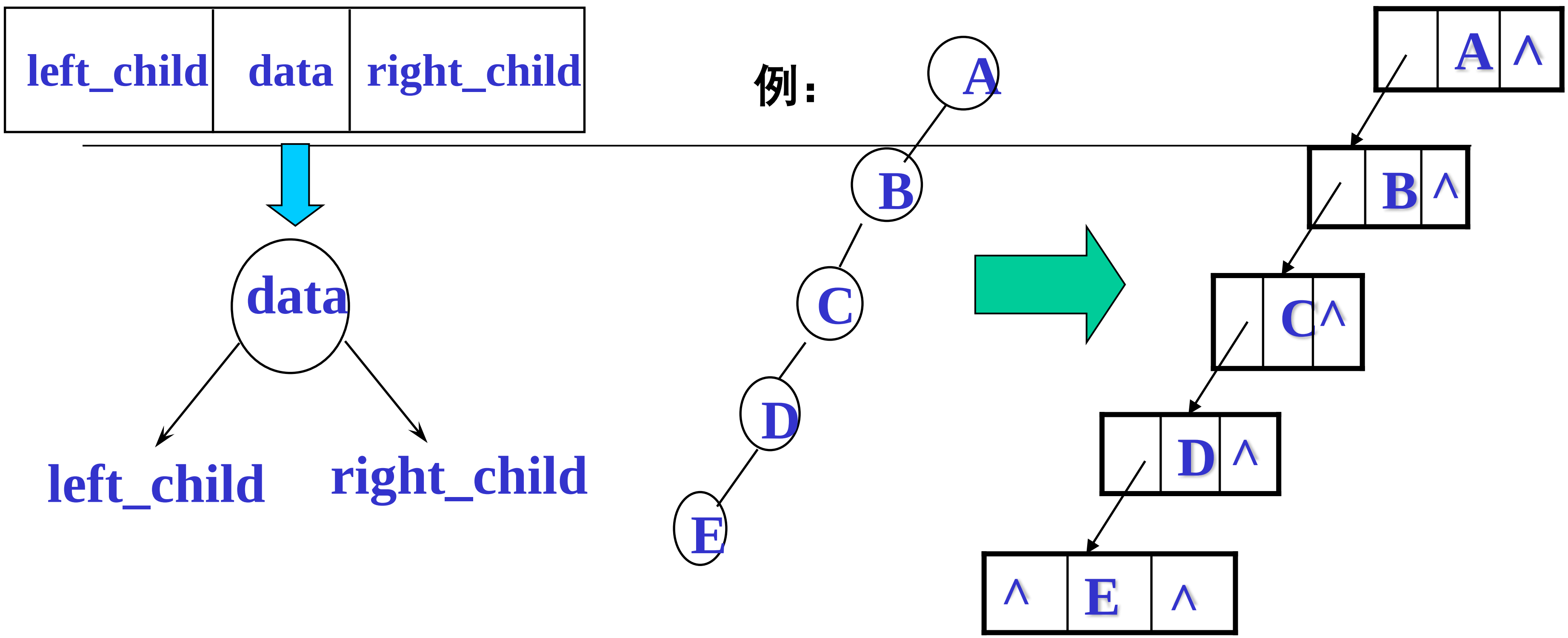
将各层空缺处统统补上“虚结点”，其内容为空。



[0]	A
[1]	B
[2]	∧
[3]	C
[4]	∧
[5]	∧
[6]	∧
[7]	D
[8]	∧
.	…
[15]	E

缺点：①浪费空间；②插入、删除不便

二、链式存储结构　用二叉链表即可方便表示。



一般从根结点开始存储。
（相应地，访问树中结点时也只能从根开始）

注：如果需要倒查某结点的双亲，可以再增加一个双亲域（直接前趋）指针，将二叉链表变成三叉链表。

4、遍历二叉树（Traversing Binary Tree）

遍历定义——指按某条搜索路线遍访每个结点且不重复（又称周游）。

遍历用途——它是树结构插入、删除、修改、查找和排序运算的前提，

是二叉树一切运算的基础和核心。

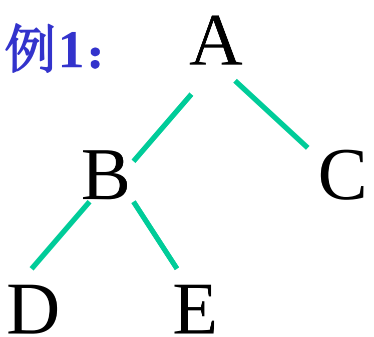
遍历规则——

- ❖ **二叉树由根、左子树、右子树构成，定义为D、 L、R**
- ❖ **D、 L、R的组合定义了六种可能的遍历方案：**

  LDR， LRD， DLR， DRL， RDL， RLD
- ❖ **若限定先左后右，则有三种实现方案：**

DLR	**LDR**	**LRD**
先 (根)序遍历	**中 (根)序遍历**	**后(根)序遍历**

例1:



先序遍历的结果是： **A B D E C**

中序遍历的结果是： **D B E A C**

后序遍历的结果是： **D E B C A**

遍历的算法实现：用递归形式

5. 二叉查找树

特点：“左小右大”，按中序遍历得到由小到大的排列

主要操作

检索(find)——

折半查找

插入(insert) ——

通过折半查找找到插入的位置（插入某个叶结点或在待插入方向上没有子结点的分支结点）

删除最小值结点(deletemin)——

删除整个树中最左边的结点，若该结点有右子树，则将其父结点中原来指向被删结点的指针改为指向其右子树。

删除给定值结点(remove)——若被删除节点的左右子结点非空，则用其右子树中的最小节点取代被删除结点。

优点：提高检索、插入、删除等操作的效率，平均情况下$\theta(\log n)$.