Searching

Linear Selection

RandPartition(A, p, r)

1. i = RANDOM(p, r)
2. exchange A[r] with A[i]
3. **return** PARTITION(A, p, r)

Randomized Selection

- RandSelect(A,p,r,i)

  If p == r then return A[p]

  q=RandPartition(A,p,r)

  k=q-p+1 /* size of A[p..q]

  If i ≤ k then return RandSelect(A,p,q,i)

  Else return RandSelect(A,q+1,r,i-k).
- First call: RandSelect(A,1,n,i).
- Returns the i-th smallest element in A[p..r].

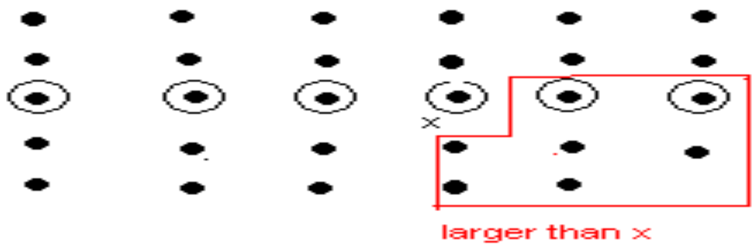
- $X_k = 1$ {the subarray A[p…q] has exactly k elements}
- $E[X_k]=1/n$
- $T(n) <= \sum_{k=1}^{n} X_k (T(\max(k-1, n-k) + O(n))$
- $E[T(n)] <= \sum_{k=1}^{n} 1/n (T(\max(k-1, n-k) + O(n))$
- $\max(k-1,n-k) = \left\{ \frac{k-1 \; if \; k>n/2}{n-k \; if \; k \le n/2} \right\}$
- $E[T(n)] <= 2/n \sum_{k=n/2}^{n-1} E(T(k) + O(n))$
- Using substitution method it can be shown, expected running time is O(n)

Complexity

- Though expected complexity is $O(n)$, worst case complexity is $O(n^2)$.
- Next we discuss, how can we make the worst case complexity linear.

Steps to find i-th smallest element
Algorithm *Select*

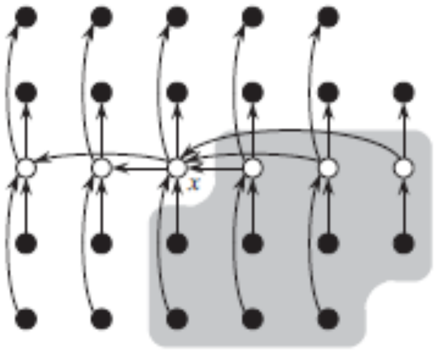
1. Divide elements in n/5 groups of 5 elements, plus at most one group with (n mod 5) elements.
2. Find median of each group:
   - Insertion sort: O(1) time (at most 5 elements).
   - Take middle element (largest if two medians).
3. Use *Select* recursively to find median x of medians.



Algorithm *Select* (cont.)

4. Partition input array around median-of-medians x. Let k be the number of elements on low side, n-k on high side.
   - $a_1, a_2, \ldots, a_k \mid a_{k+1}, a_{k+2}, \ldots, a_n$
   - $a_i < a_j$, for $1 \le i \le k$, $k+1 \le j \le n$.
5. Use *Select* recursively to:
   - Find i-th smallest element on low side, if $i \le k$
   - Find (i-k)-th smallest on high side, if $i > k$.

Analysis



- Find lower bound on number of elements greater than x.
  - At least half of medians in step 2 greater than x. Then,
  - At least half of the groups contribute 3 elements that are greater than x, except:
    - Last group (if less than 5 elements);
    - x own group.
  - Discard those two groups:
    - Number of elements greater than x is ≥ 3((n/5)/2-2)=3n/10-6.
- Similarly, number of elements smaller than x is ≥3n/10-6.
- Then, in worst case, *Select* is called recursively in Step 5 on at most 7n/10+6 elements (upper bound).

Analysis (cont.)

- Steps 1,2 and 4: O(n) time.
- Step 3: T(n/5)
- Step 5: at most T(7n/10+6)
    - $7n/10+6 < n$ for $n > 20$.
- $T(n) \leq T(\lceil n/5 \rceil)+T(7n/10+6)+O(n)$, $n > n_1$.
- Use substitution to solve:
    - Assume $T(n) \leq cn$, for $n > n_1$; find n_1 and c.

Analysis (cont.)

- $T(n) \le c\lceil n/5 \rceil + c(7n/10+6) + O(n)$

  $\le cn/5 + c + 7cn/10 + 6c + O(n)$

  $= 9cn/10 + 7c + O(n)$
- Want $T(n) \le cn$:
  - Pick c such that $c(n/10-7) \ge c_1 n$, where c_1 is constant from $O(n)$ above (n_1 = 80).

Binary Search Tree

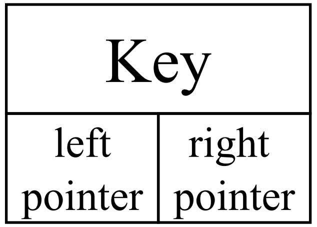
Binary Trees

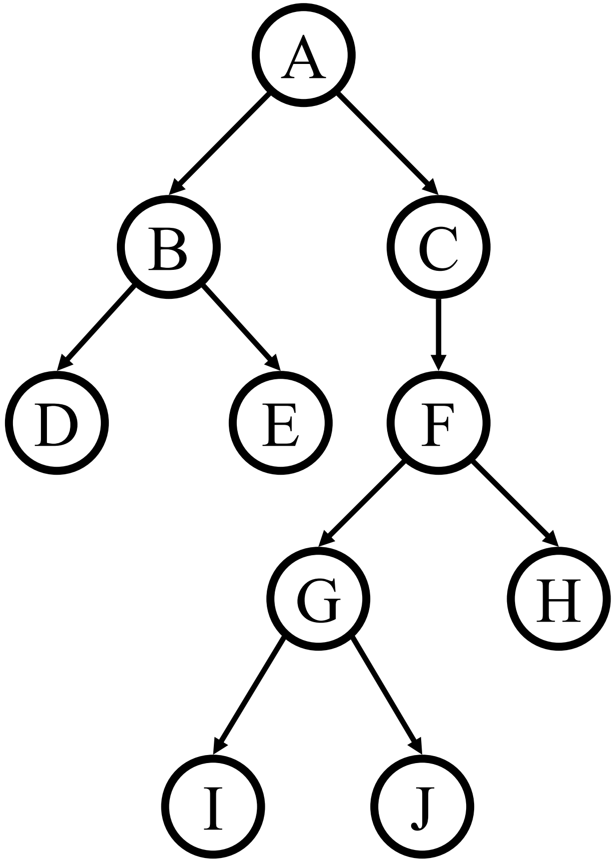
Binary tree will have

- a root
- left subtree *(maybe empty)*
- right subtree *(maybe empty)*

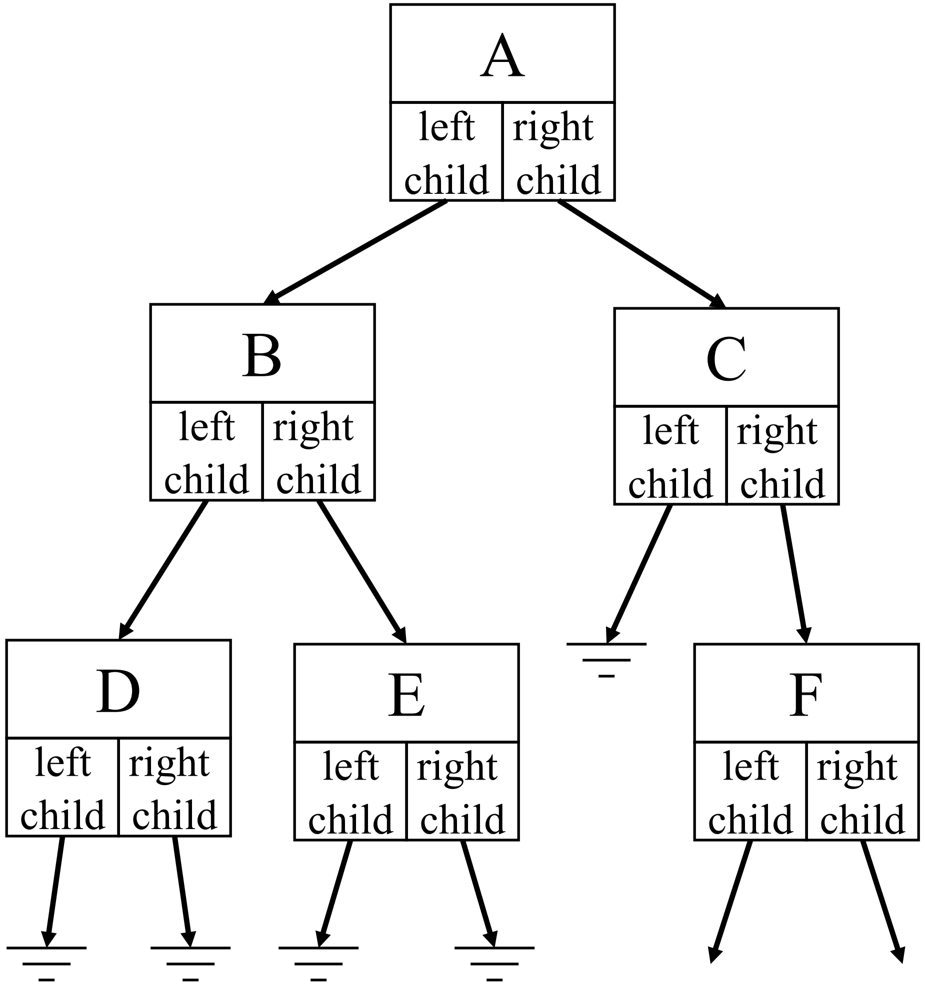
Every node can be presented in following manner

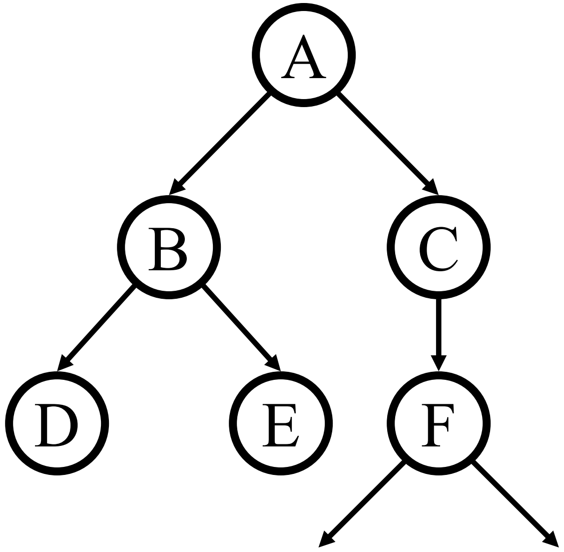
```
typedef struct s {
int key;
struct s * left;
struct s * right
} Node;
```





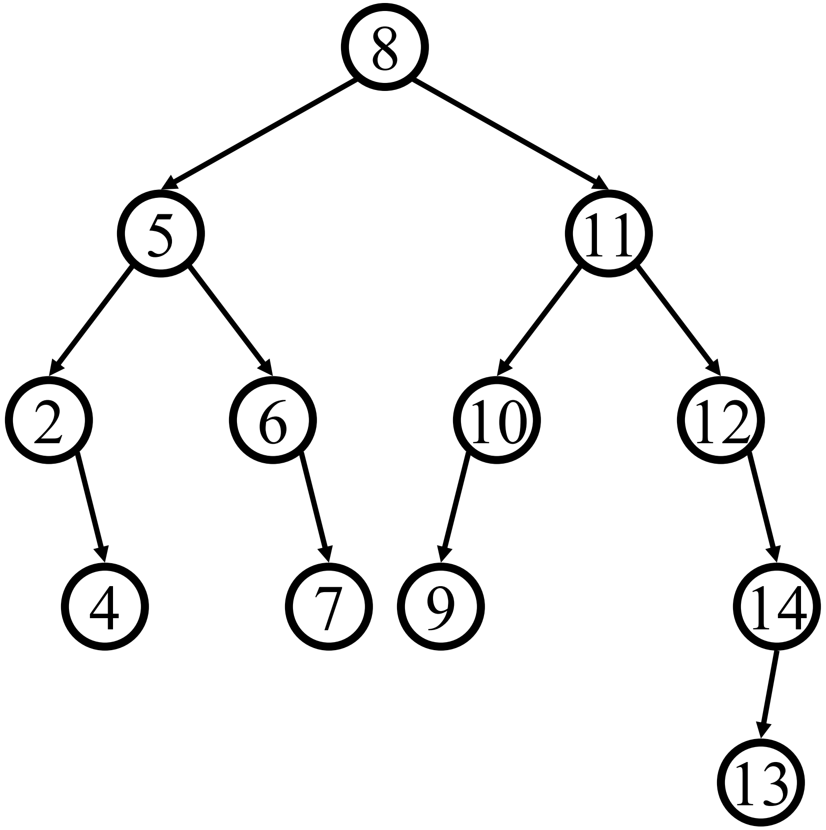
Binary Tree Representation





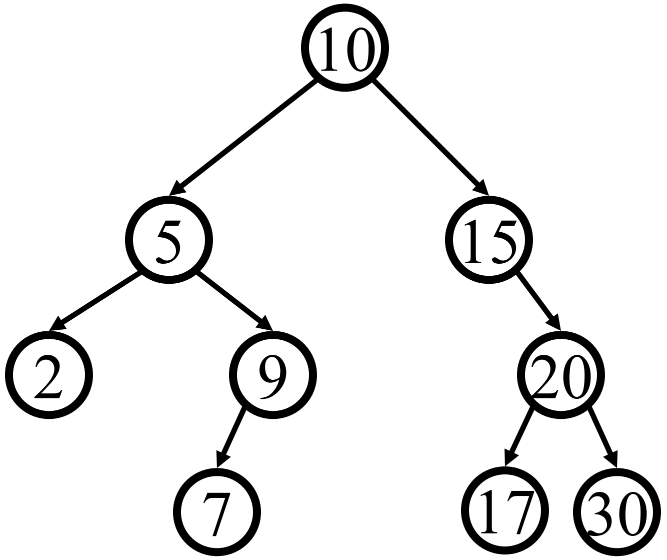
Binary Search Tree

- A binary search tree is a binary tree in which all nodes in the left subtree of a node have lower values than the node. All nodes in the right subtree of a node have higher value than the node. In summary
  - all keys in left subtree smaller than root’s key
  - all keys in right subtree larger than root’s key



Recursive Find

```
Node *
find(int data, Node * t)
{
  if (t == NULL || data == t->key) return t;
  else if (data < t->key)
    return find(data, t->left);
  else
    return find(data, t->right);
}
```



Insert

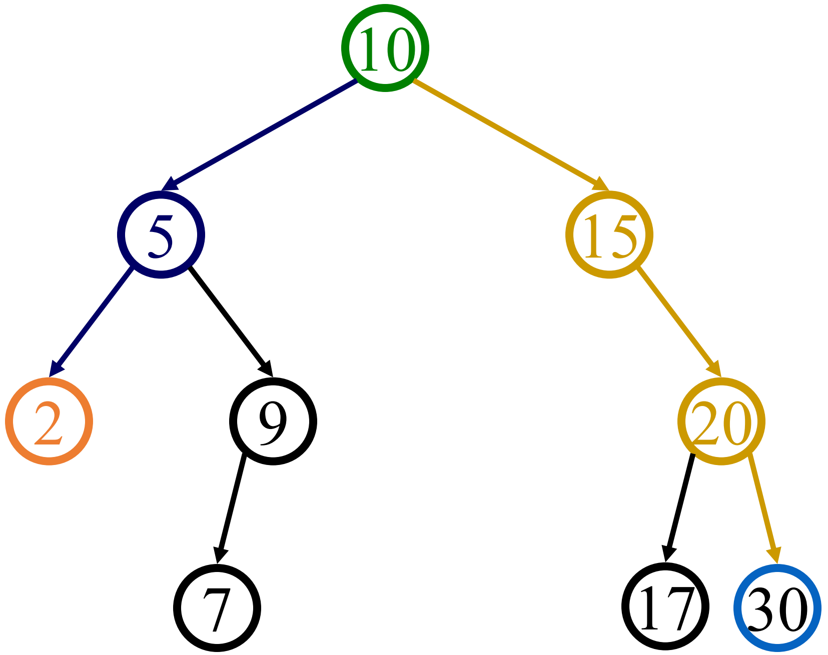
Concept:

- Proceed down tree as in Find
- If new key not found, then insert a new node at last spot traversed

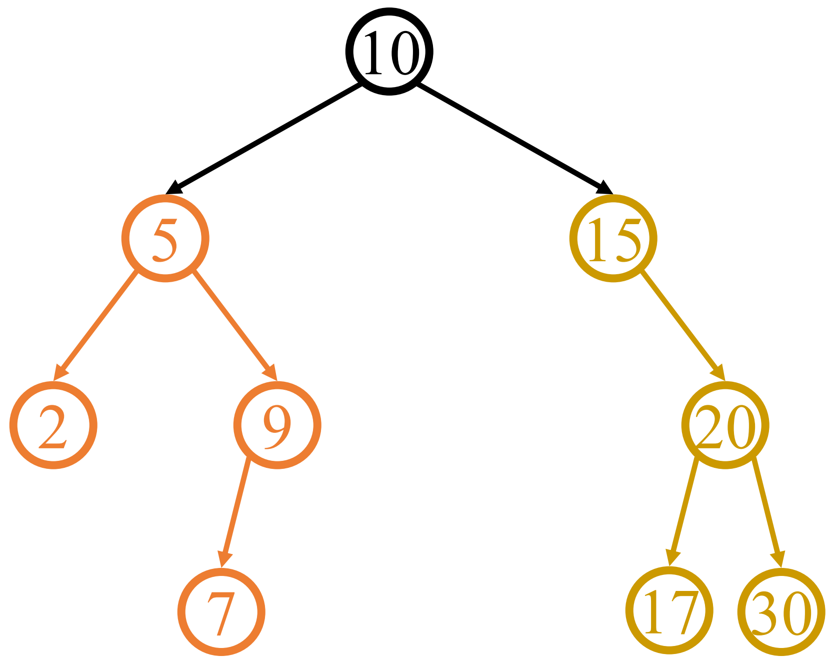
```
void
insert(int data, Node * t)
{
  if ( t == NULL ) {
    t = new Node(x);

  } else if (data < t->key) {
    insert( data, t->left );

  } else if (data > t->key) {
    insert( data, t->right );

  } else {
    // duplicate
    // handling is app-dependent
}
```

FindMin, FindMax

- Find minimum
- Find maximum



In-Order Traversal



In order listing:

2→5→7→9→10→15→17→20→30

visit left subtree

visit node

visit right subtree

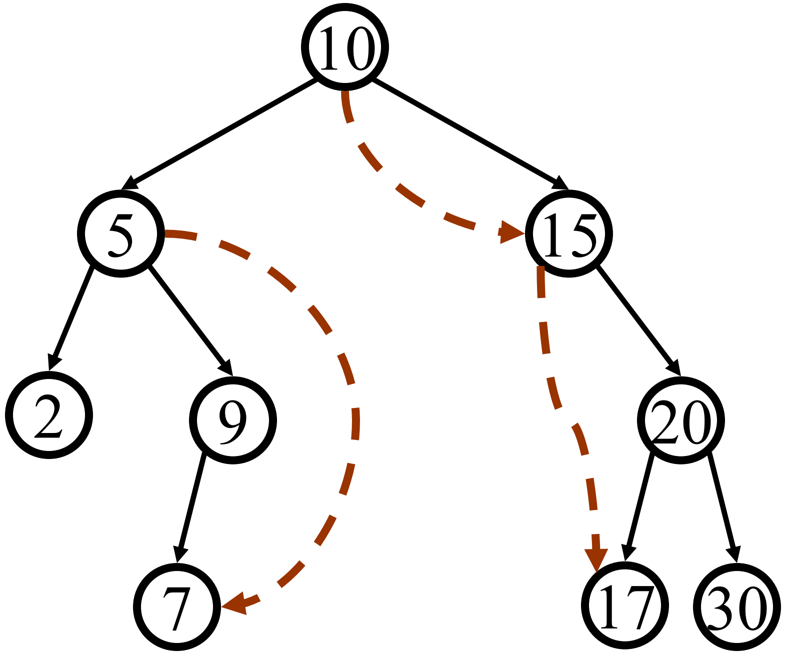
```
Proc InOrderPrint(pointer){
  pointer != NULL{
    InOrderPrint(pointer->left)
    print(data)
    InOrderPrint(pointer->right)
    }
}
```

What does this guarantee with a BST?

Successor Node

Next larger node in this node's subtree

```
Node * succ(Node * t) {
  if (t->right == NULL)
    return NULL;
  else
    return min(t->right);
}
```

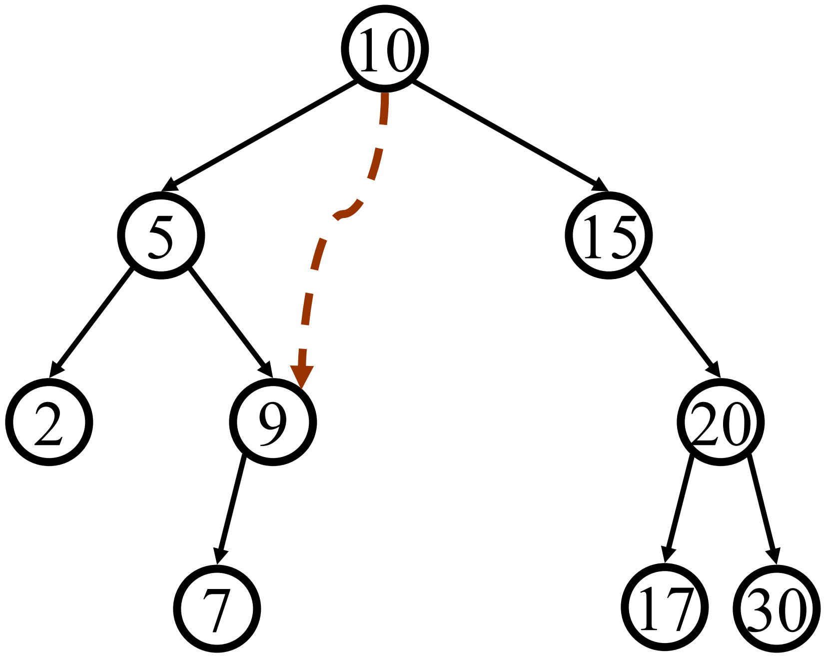


How many children can the successor of a node have?

Predecessor Node

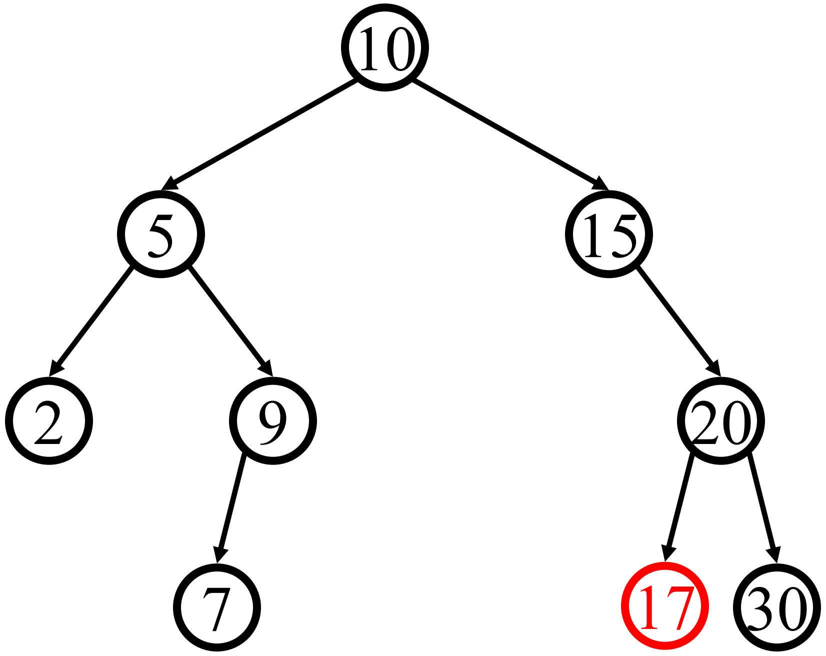
Next smaller node in this node's subtree

```
Node * pred(Node * t) {
  if (t->left == NULL)
    return NULL;
  else
    return max(t->left);
}
```



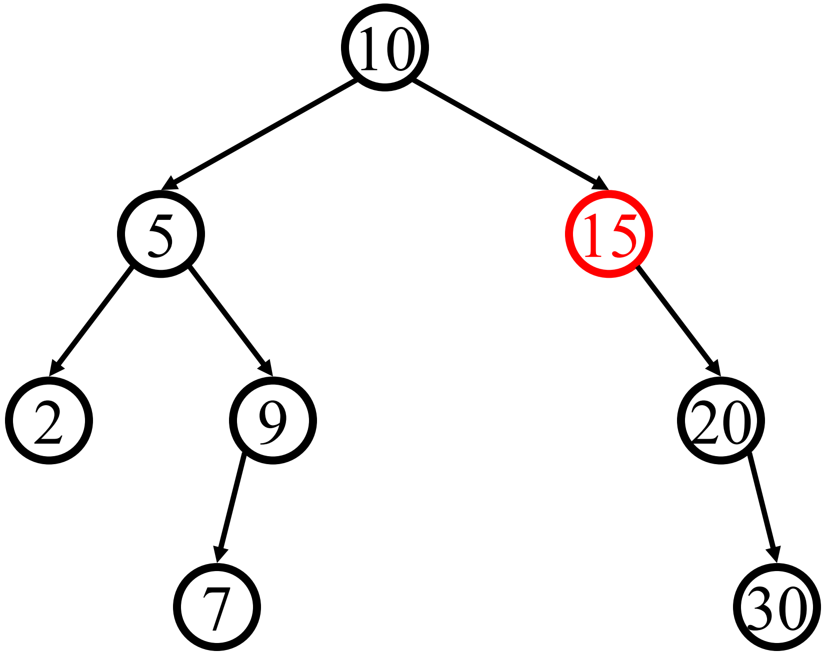
Deletion - Leaf Case

Delete(17)



Deletion - One Child Case

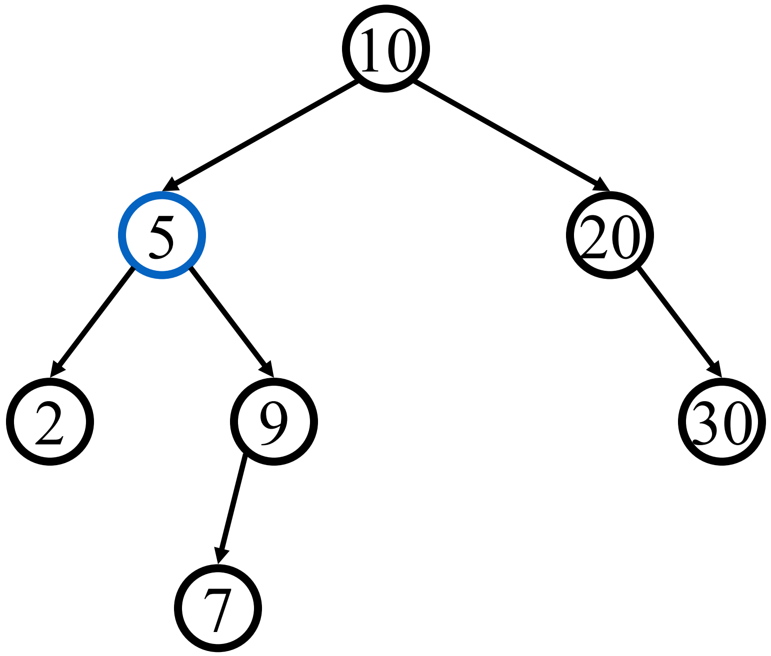
Delete(15)



Deletion - Two Child Case

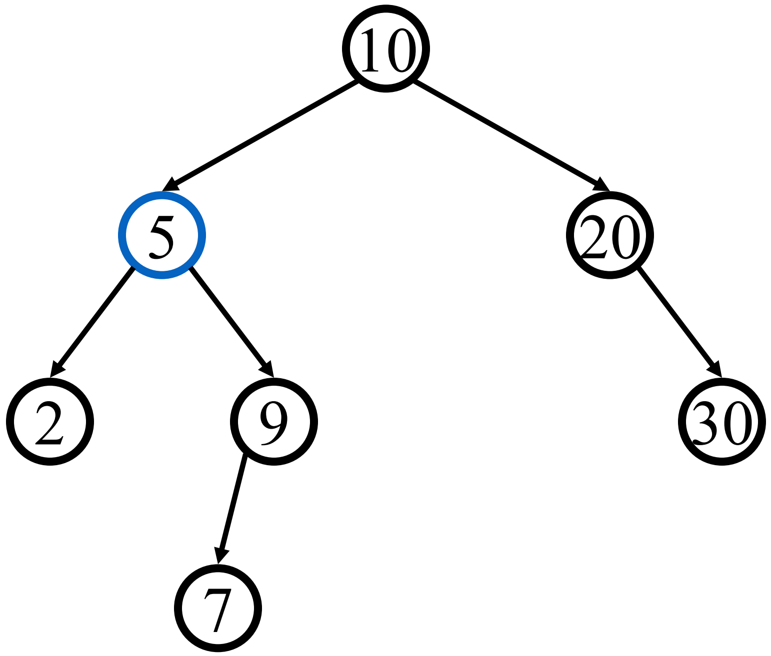
Replace node with descendant whose value is guaranteed to be between left and right subtrees: the successor

Delete(5)



Deletion - Two Child Case

Replace node with descendant whose value is guaranteed to be between left and right subtrees: the successor



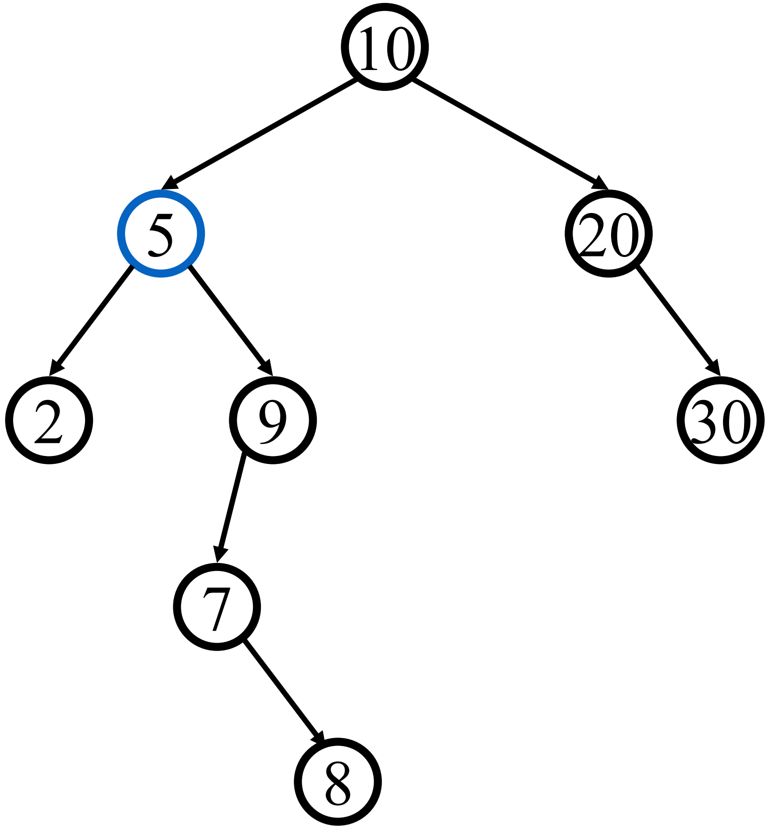
Delete(5)

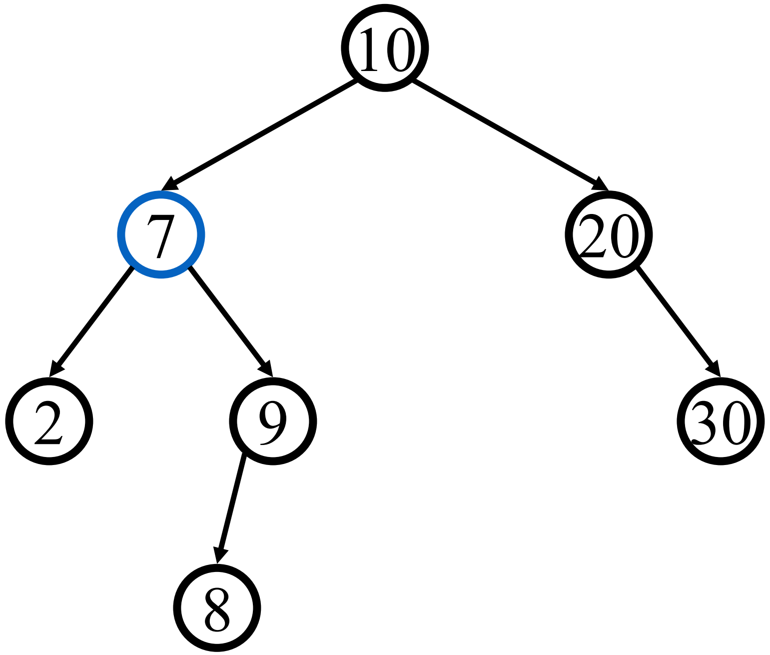
Could we have used predecessor instead?

Deletion - Two Child Case

Replace node with descendant whose value is guaranteed to be between left and right subtrees: the successor

Delete(5)





Red-Black tree

- Recall binary search tree
    - Key values in the left subtree <= the node value
    - Key values in the right subtree >= the node value
- Operations:
    - insertion, deletion
    - Search, maximum, minimum, successor, predecessor.
    - O(h), h is the height of the tree.

Red-black trees

- Definition: a binary search tree, satisfying:
    1. Every node is either red or black
    2. The root is black
    3. Every leaf is NIL and is black
    4. If a node is red, then both its children are black
    5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
- Purpose: keep the tree balanced.
- Other balanced search tree:
    - AVL tree, B-Tree

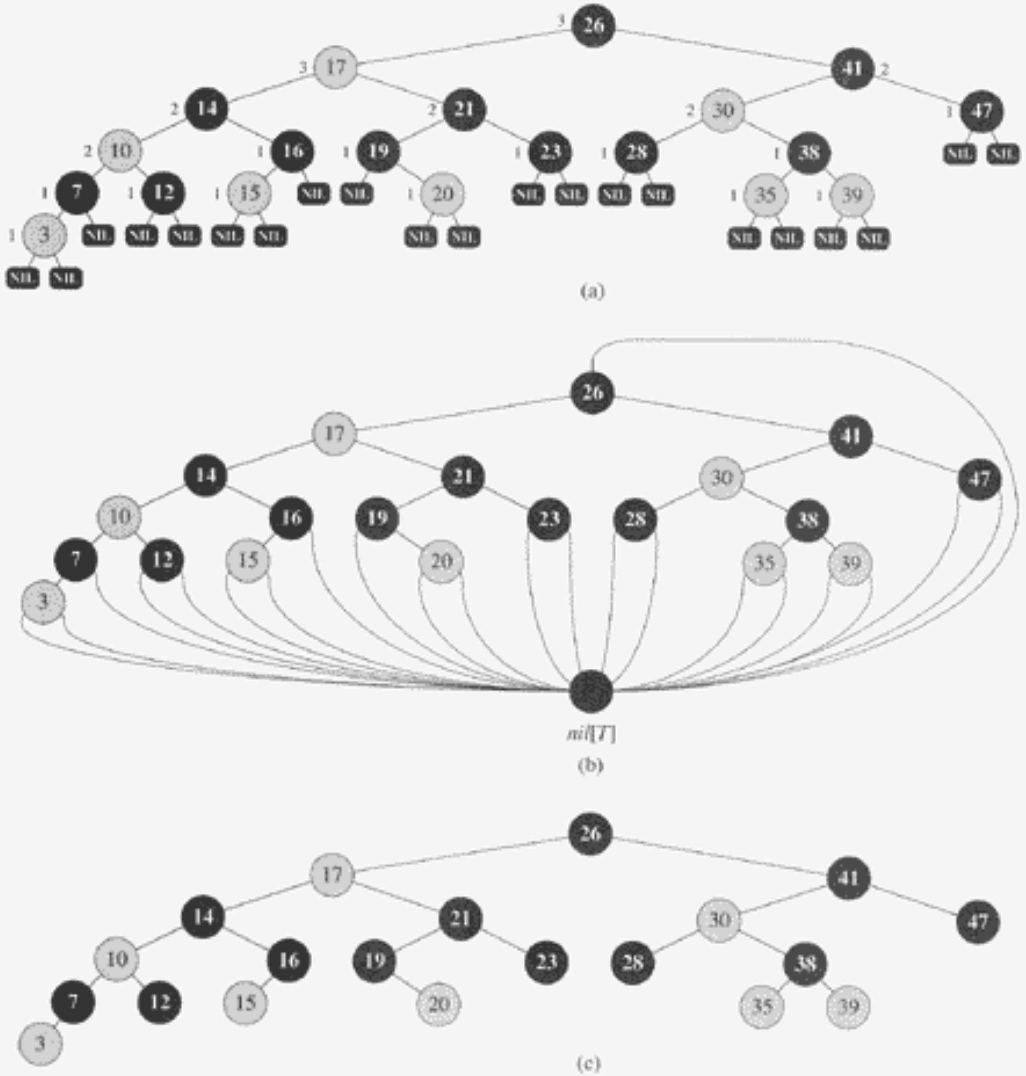
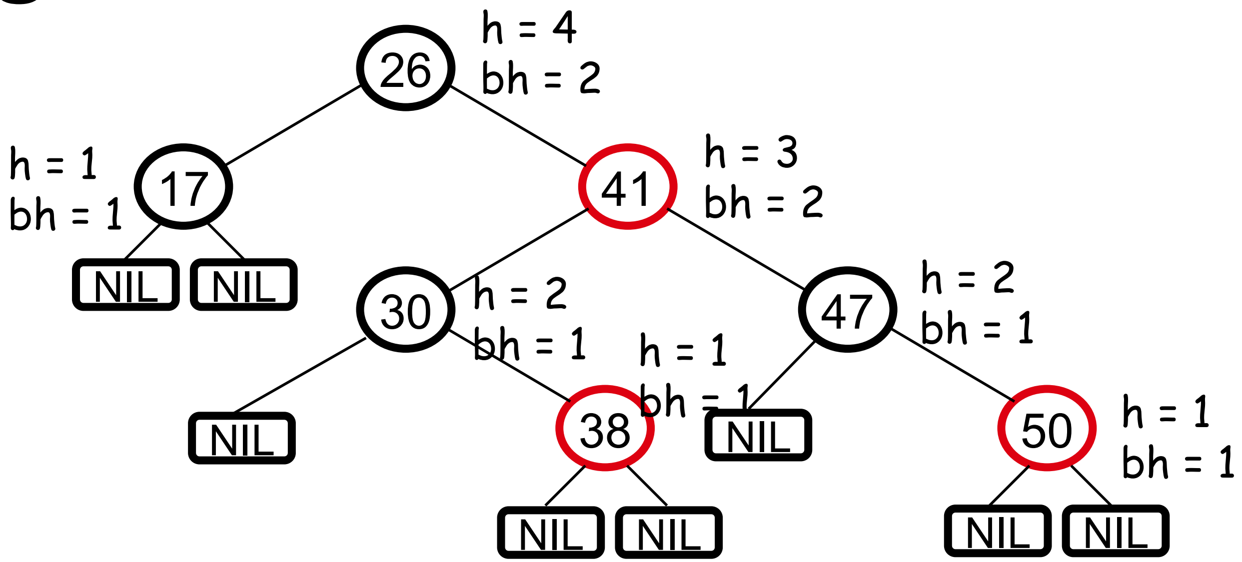


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL's have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *nil*[*T*], which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

Black-Height of a Node



- **Height of a node:** the number of edges in the **longest** path to a leaf
- **Black-height** of a node x: bh(x) is the number of black nodes (including NIL) on the path from x to a leaf, <u>not counting x</u>
- **Fields of Red-balck-tree node**: Left, right ,parent, color, key

Upper Bound on Height of a Red-Black Tree

A red-black tree with n internal nodes

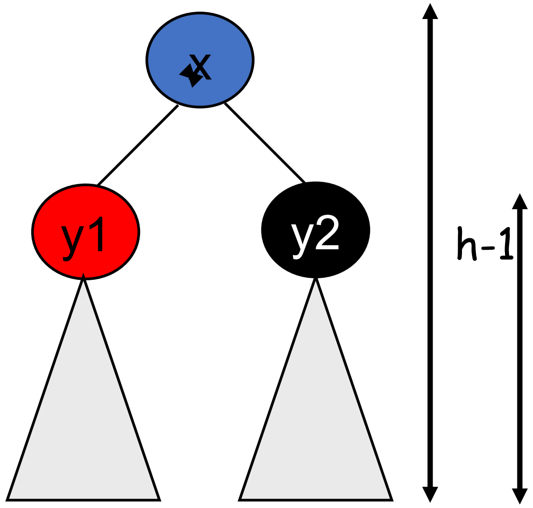
has height <u>at most</u> $2lg(n + 1)$

- Claim 1:Any node x with height $h(x)$ has $bh(x) \geq h(x)/2$
  - By property 4, at most $h/2$ **red** nodes on the path from the node to a leaf
  - Hence at least $h/2$ are **black**

- Claim 2: The sub-tree rooted at any node x contains **at least** $2^{bh(x)} - 1$ internal nodes
  - Proof:
    - By induction on **h[x]**
    - **Basis:** $h[x] = 0$
      - x is a leaf ($NIL[T]$) => $bh(x) = 0$
      - # of internal nodes: $2^0 - 1 = 0$
    - **Inductive Hypothesis:** assume it is true for $h[x]=h-1$

- **Inductive step:**
  - Prove it for $h[x]=h$
  - Let bh(x) = b, then any child y of x has:
  - bh (y) = b (if y is a red node)
  - bh (y) = b-1 (if y is a black node)
  - Height of y is h-1
  - As per inductive hypothesis y will have at least $2^{bh(y)-1}$ internal nodes
  - The sub-tree rooted at x contains at least:
  - $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$

    $= 2\cdot(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes



bh(y1)=bh(x) ≥bh(x)-1

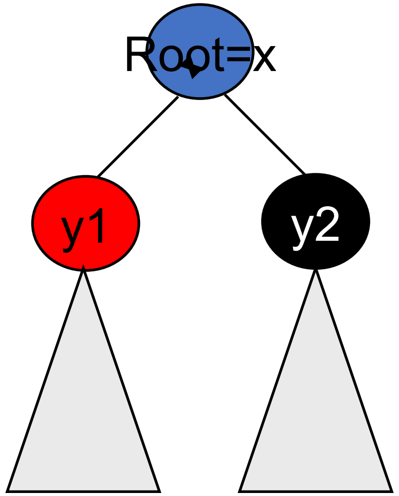
bh(y2)=bh(x)-1 ≥bh(x)-1

A red-black tree with **n** internal nodes has height at most **2lg(n + 1)**.

- **Proof:**
- n>= $2^{bh(x)}$ - 1 (using claim 2)

  >=$2^{h(x)/2}$ -1 (using claim 1)

h(x)<= 2lg(n + 1).



Operations on Red-Black-Trees

- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(h)$ time
  - They take $O(\lg n)$ time on red-black trees
- What about TREE-INSERT and TREE-DELETE?
  - They will still run on $O(\lg n)$
  - We have to guarantee that the modified tree will still be a red-black tree

INSERT

INSERT: what color to make the new node?

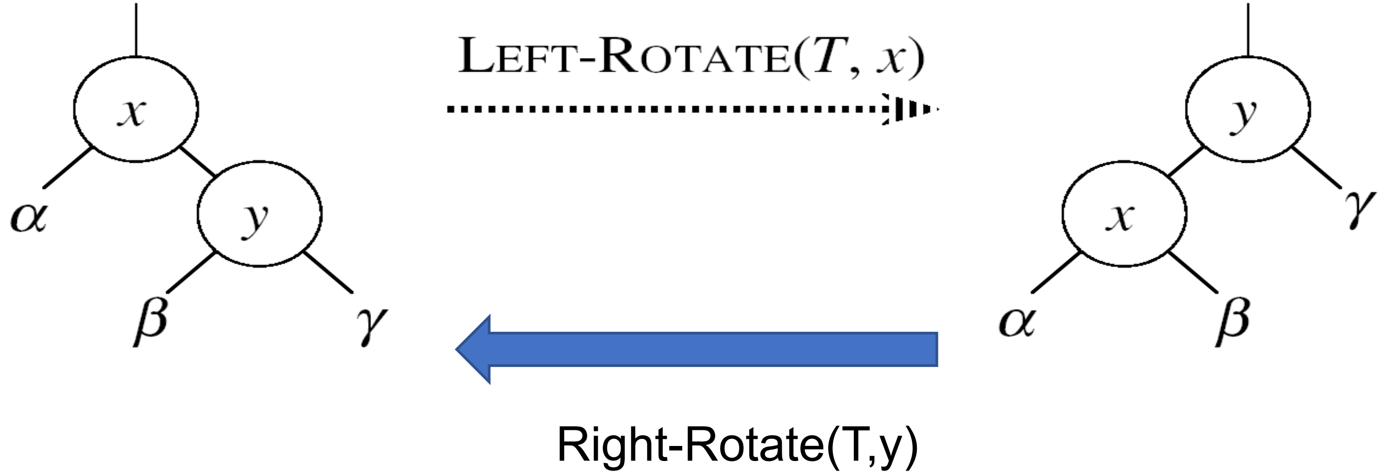
- Red?
  - Property 4 might be violated: if a node is red, then both its children are black
  - Property 2 might be violated : root node is always black
- Black?
  - Property 5 might be violated: all paths from a node to its leaves contain the same number of black nodes

Rotations

- Rotation helps in re-structuring the tree after insert and delete operations to maintain red-black tree properties.
- There are two types of rotations
  - Left rotation
  - Right rotation

Left Rotations

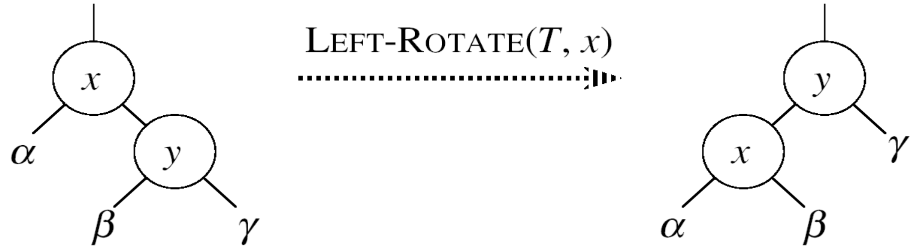
- Assumptions for a left rotation on a node x:
  - The right child of x (y) is not NIL



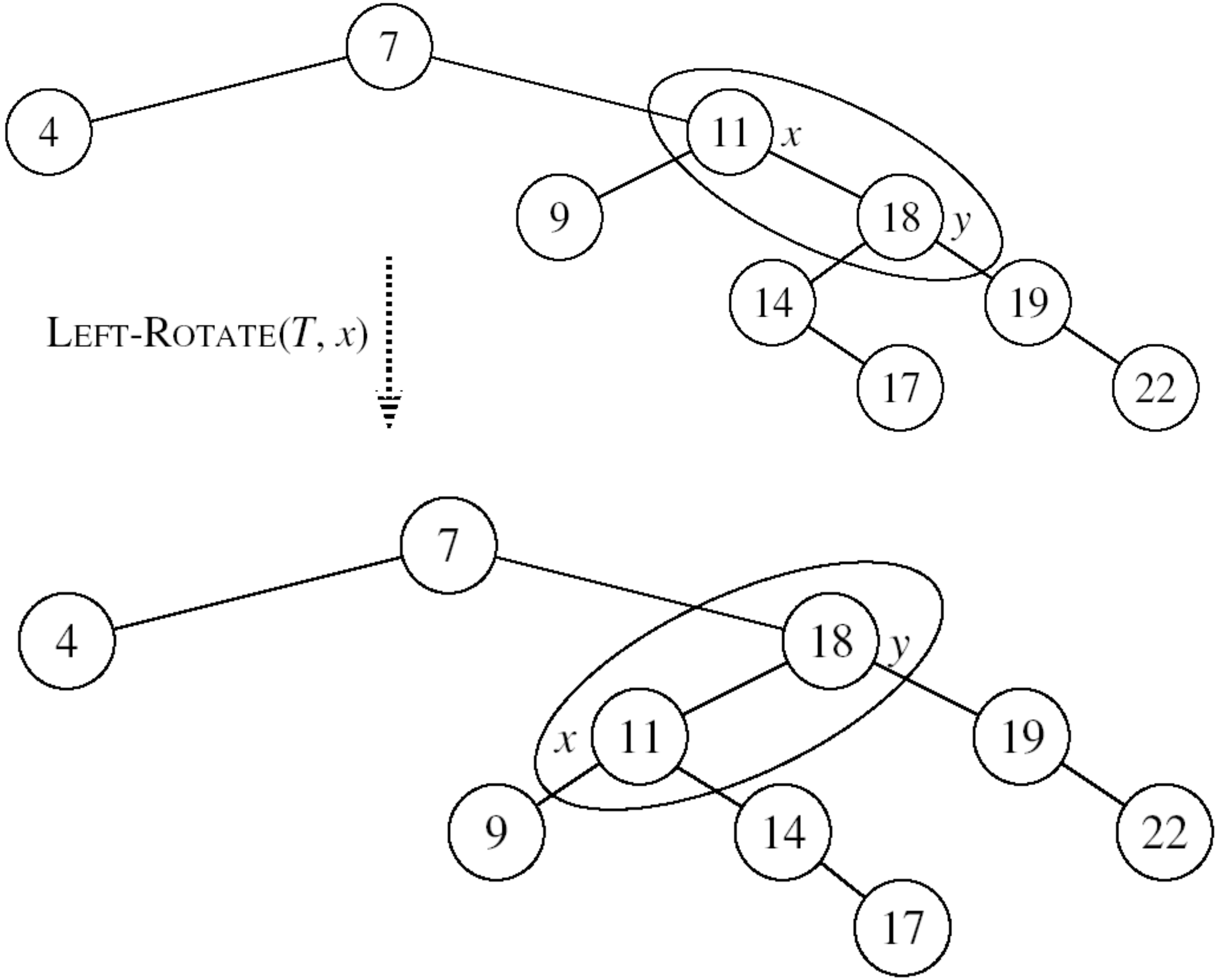
- Idea:
  - Pivots around the link from x to y
  - y's left child becomes x's right child
  - Makes y the new root of the sub-tree
    - Set X's parent as Y's parent
    - If X was left child of its parent, then make Y as left child of its parent
    - Else if X was right child of its parent then make Y as right child of its parent
  - x becomes y's left child and Y is set to new parent of X

LEFT-ROTATE(T, x)

```
y ← right[x]                  ► Set y
right[x] ← left[y]            ► y's left sub-tree becomes x's right sub-tree
if left[y] ≠ NIL
   then p[left[y]] ← x        ► Set the parent relation from left[y] to x
p[y] ← p[x]                   ► The parent of x becomes the parent of y
if p[x] = NIL
   then root[T] ← y
   else if x = left[p[x]]
           then left[p[x]] ← y
           else right[p[x]] ← y
left[y] ← x                   ► Put x on y's left
p[x] ← y                      ► y becomes x's parent
```



Example: LEFT-ROTATE



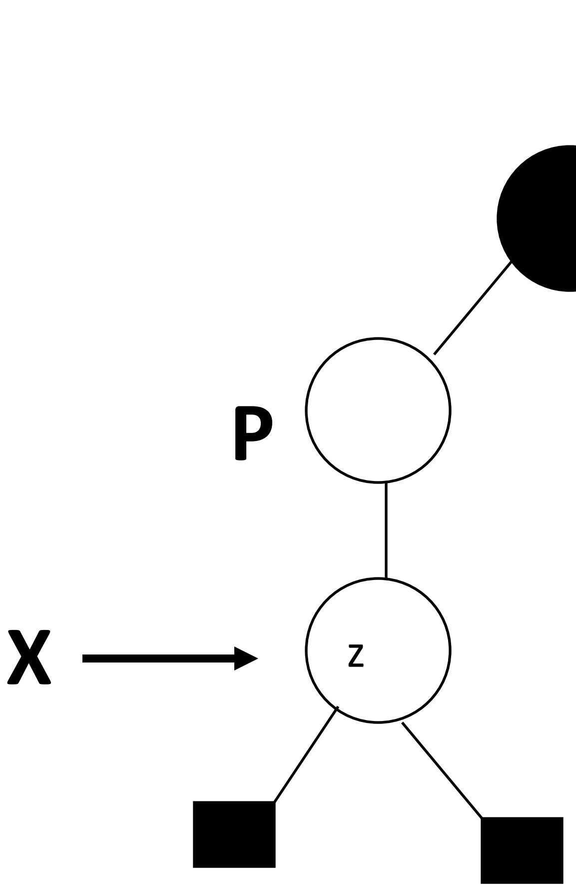
Insertion

- Goal:
  - Insert a new node z into a red-black-tree
- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node **red**
  - Restore the red-black-tree properties
    - Use an auxiliary procedure RB-INSERT-FIXUP

```
RB-INSERT(T, z)
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
     do y ← x
        if key[z] < key[x]
           then x ← left[x]
           else x ← right[x]
p[z] ← y
if y = nil[T]
   then root[T] ← z
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-INSERT-FIXUP(T, z)
```

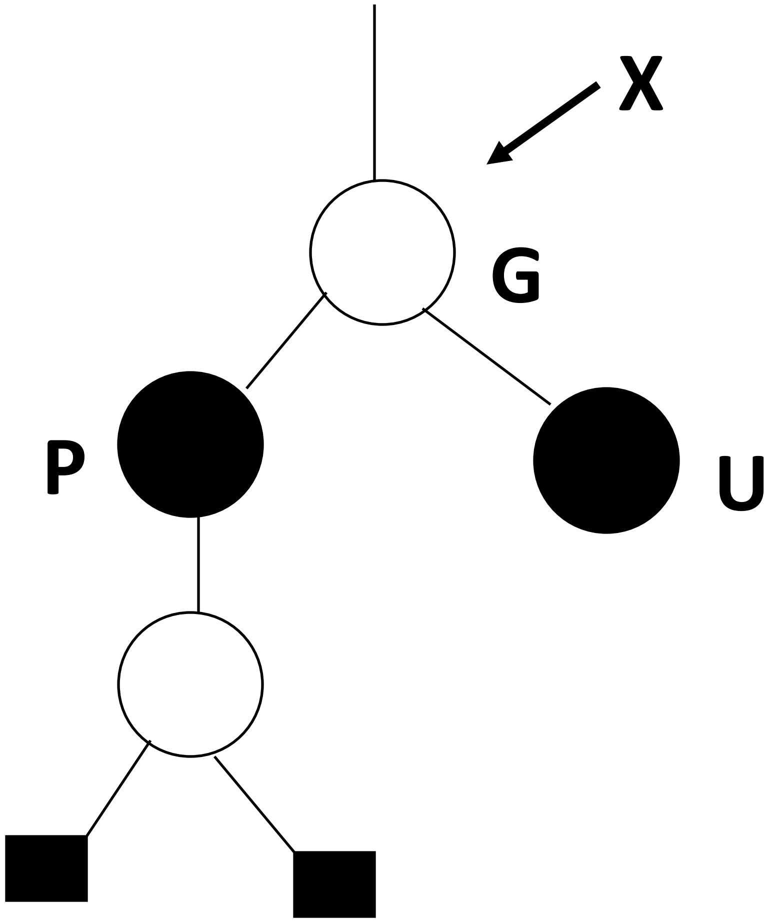
Properties violations

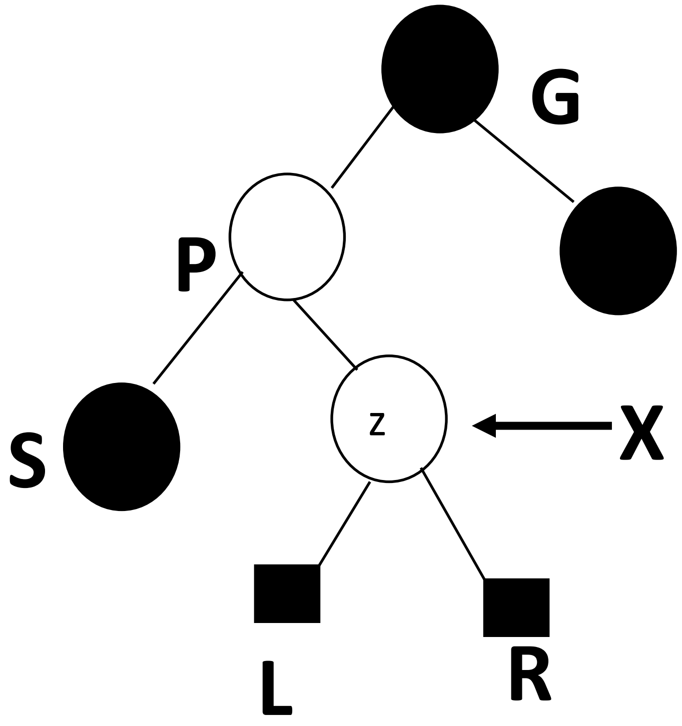
- Property 1 (each node black or red): hold
- Property 2: (root is black), not, if z is root (and colored red).
- Proper 3: (each leaf is black sentinel): hold.
- Property 4: (the child of a red node must be black), not, if z’s parent is red.
- Property 5: same number of blacks: hold

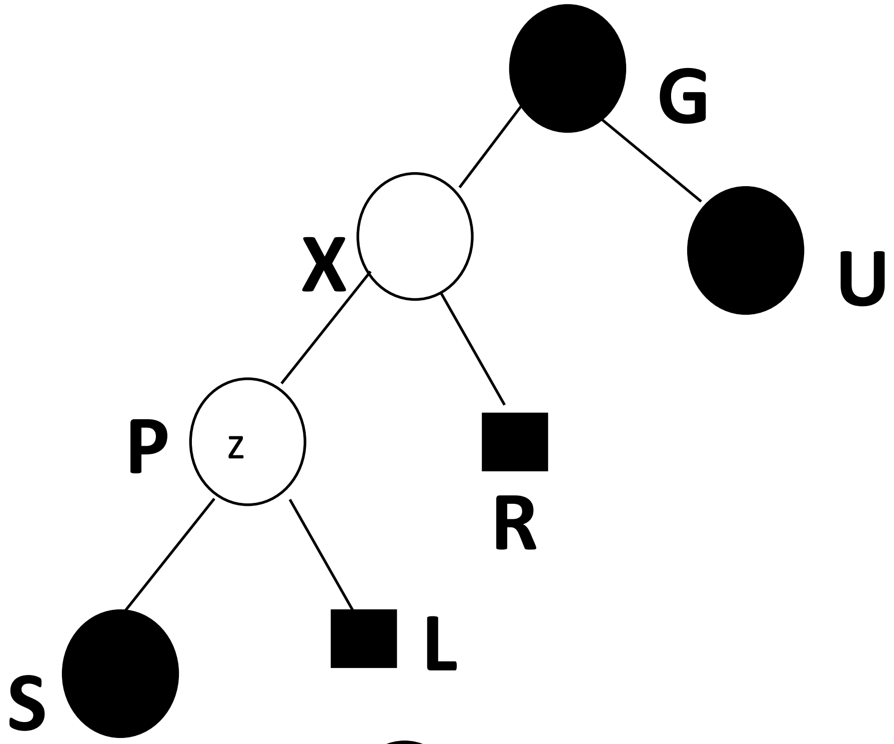


Case 1 – U is Red

Just Recolor and move up



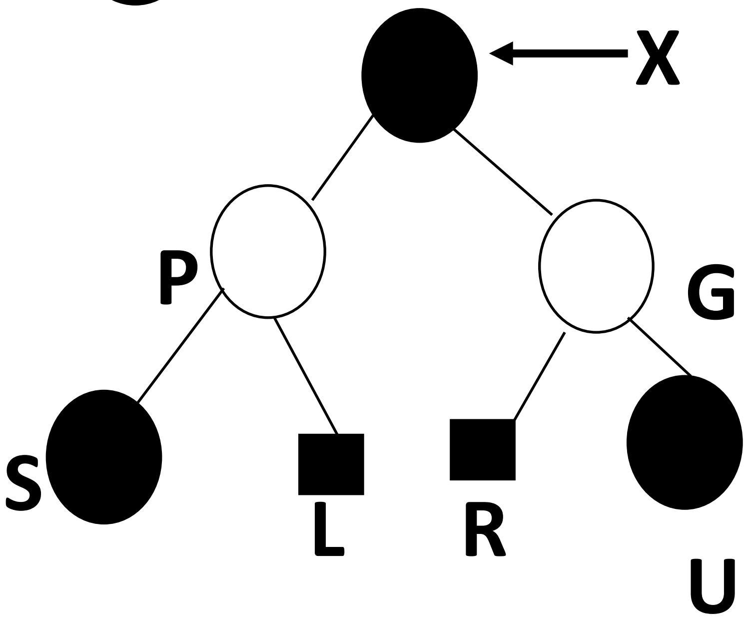


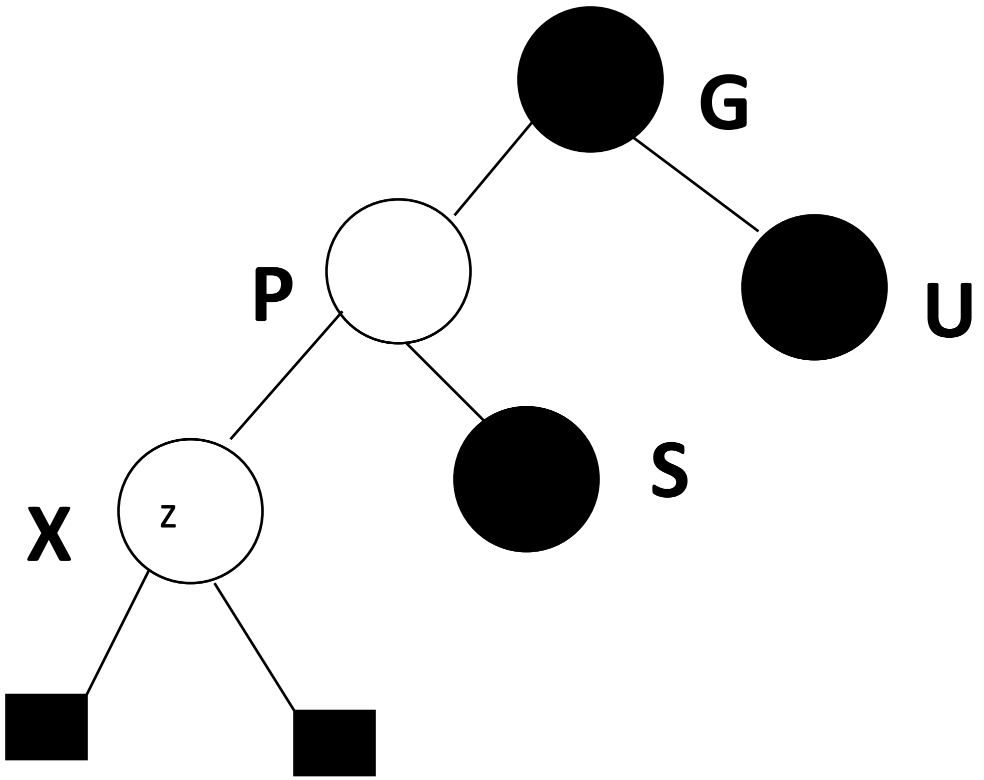


Case 2 – Zig-Zag

Single rotation to make case 3

Then Handle case 3

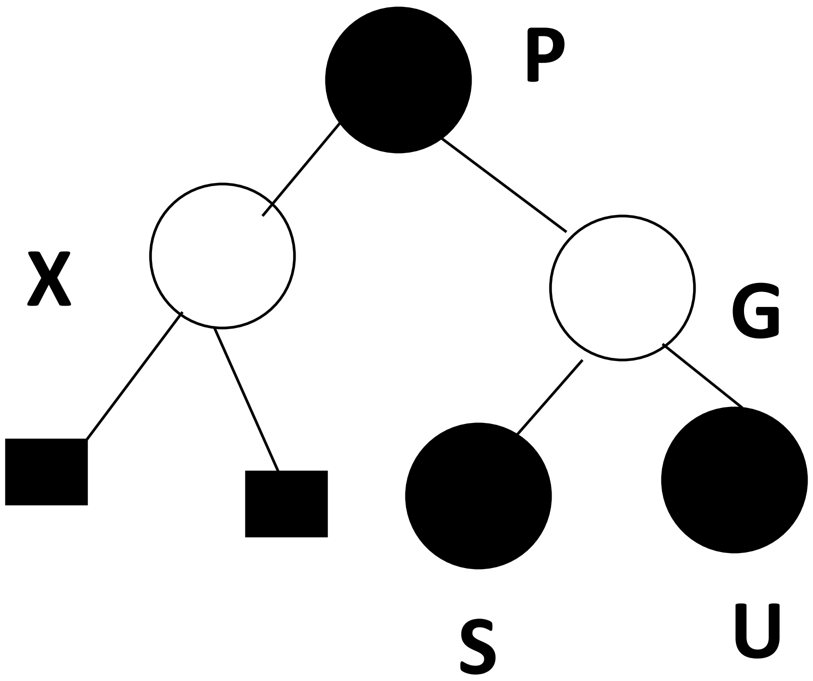




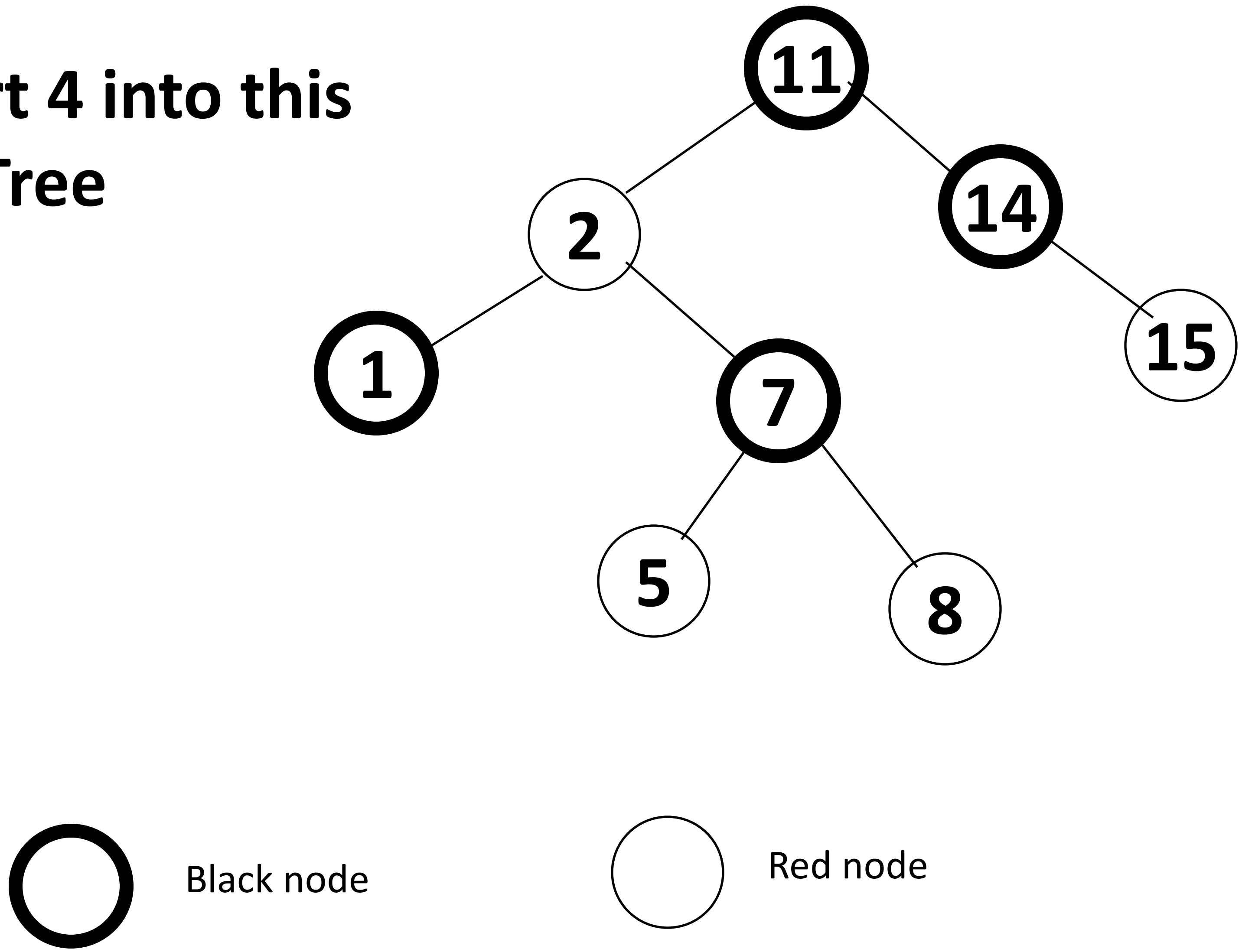
Case 3 – Zig-Zig

Single Rotate P around G

Recolor P and G



Insert 4 into this R-B Tree



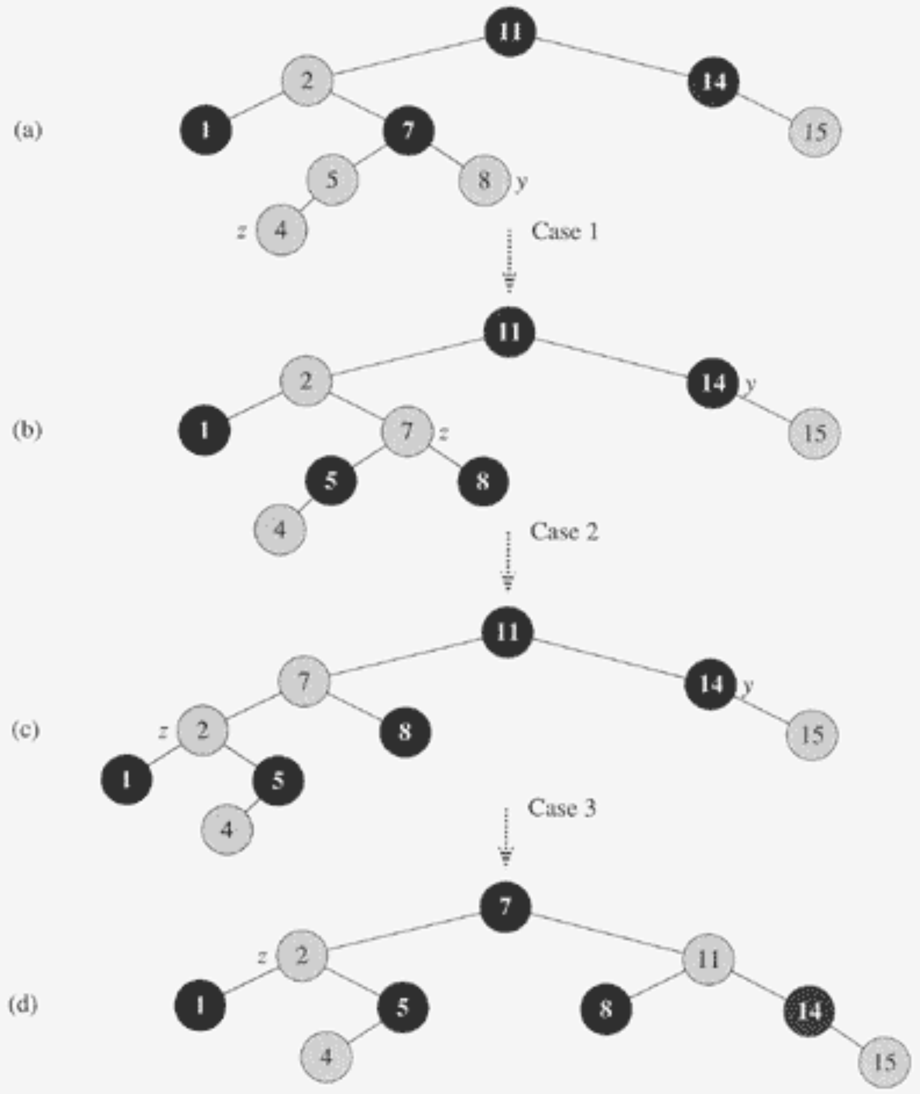


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Since z and its parent $p[z]$ are both red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $p[z]$, case 2 can be applied. A left rotation is performed, and the tree that results is shown in **(c)**. Now z is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in **(d)**, which is a legal red-black tree.

Insertion Practice

Insert the values 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty Red-Black Tree

Case 1,2,3: p[z] is the left child of p[p[z]].
Correspondingly, there are 3 other cases,
In which p[z] is the right child of p[p[z]]

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
        then y ← right[p[p[z]]]
            if color[y] = RED
              then color[p[z]] ← BLACK                    ▷ Case 1
                   color[y] ← BLACK                       ▷ Case 1
                   color[p[p[z]]] ← RED                   ▷ Case 1
                   z ← p[p[z]]                            ▷ Case 1
              else if z = right[p[z]]    //color[y]=BLACK
                      then z ← p[z]                       ▷ Case 2
                           LEFT-ROTATE(T, z)              ▷ Case 2
                   color[p[z]] ← BLACK                    ▷ Case 3
                   color[p[p[z]]] ← RED                   ▷ Case 3
                   RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
        else (same as then clause
                   with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

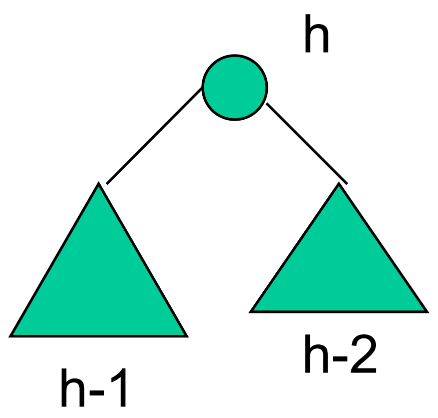
AVL Trees

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

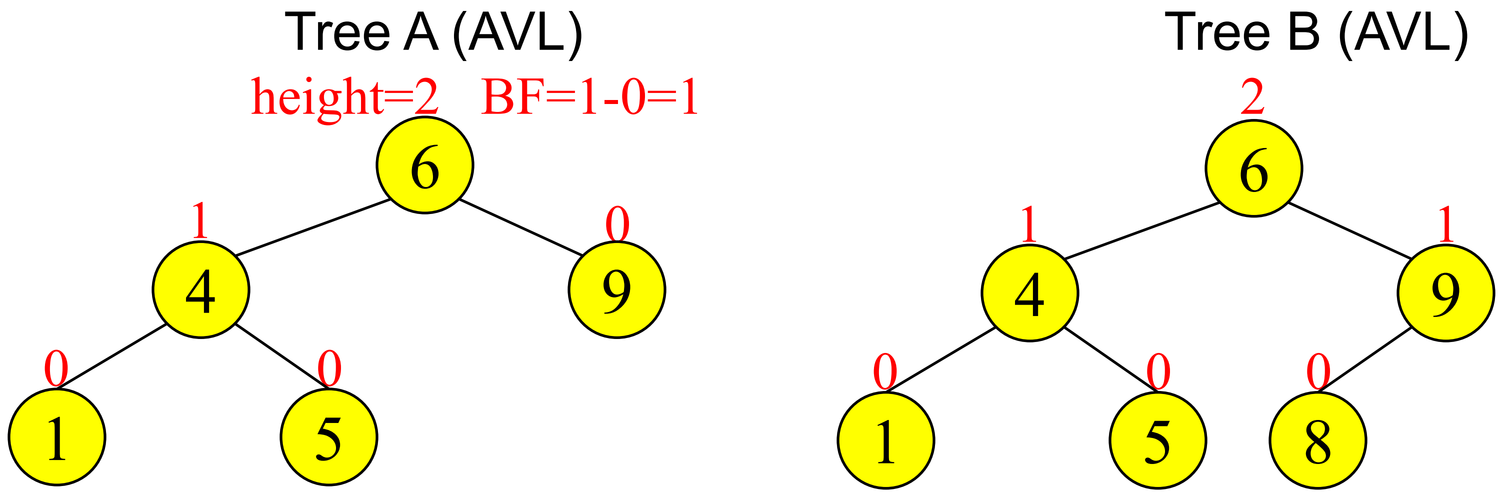
- $N(h)$ = minimum number of nodes in an AVL tree of height h.
- Basis
  - › $N(0) = 1$, $N(1) = 2$
- Induction
  - › $N(h) = N(h-1) + N(h-2) + 1$
- Solution
  - › $N(h) \geq \phi^h$ $(\phi \approx 1.62)$



Height of an AVL Tree

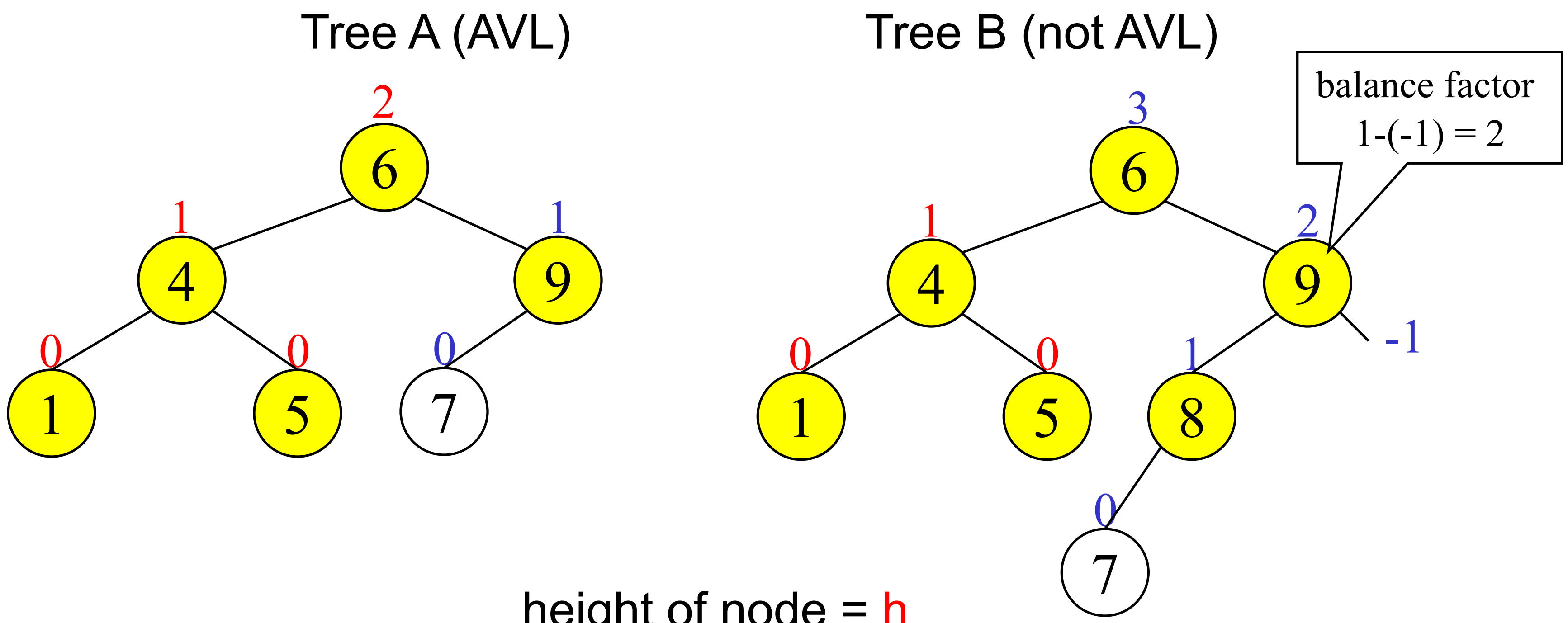
- $N(h) \geq \phi^h \quad (\phi \approx 1.62)$
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes O(logn))

Node Heights



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

Node Heights after Insert 7



height of node = h

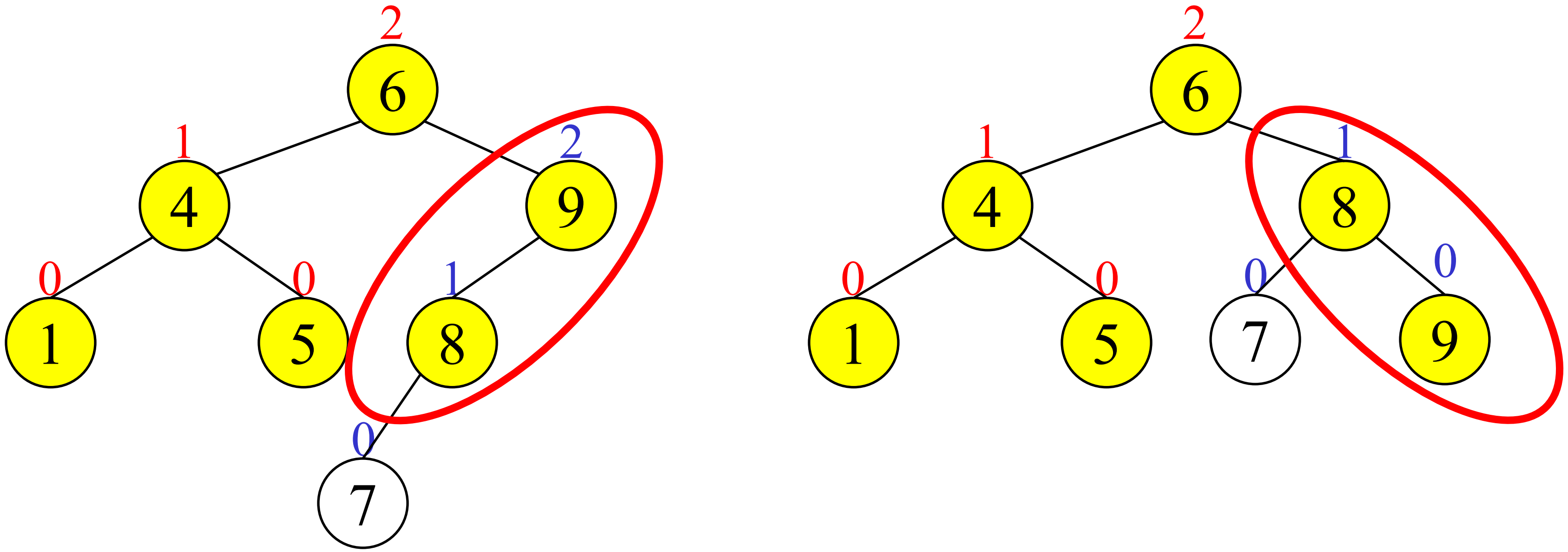
balance factor = h_{left}-h_{right}

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.

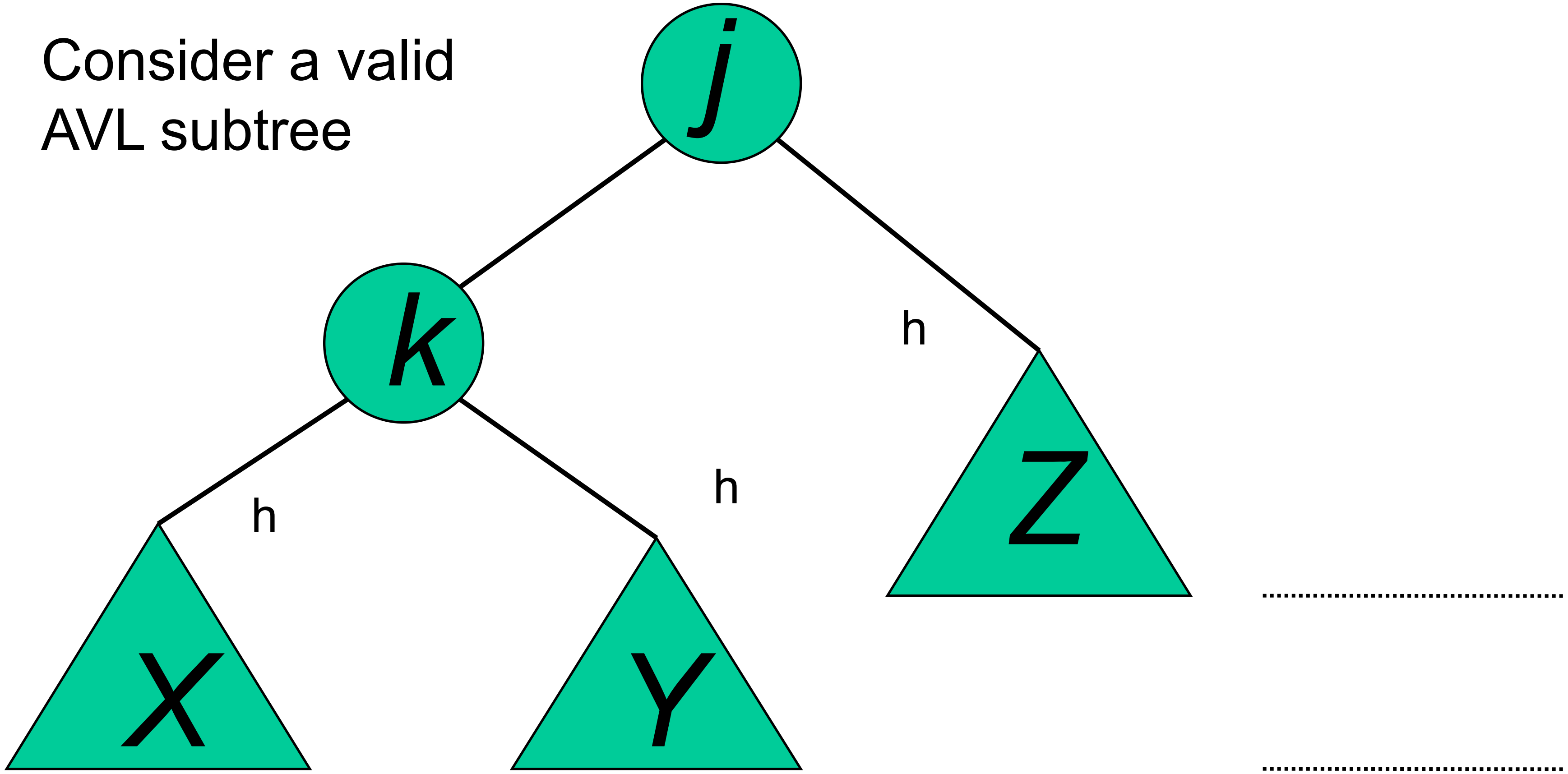
Inside Cases (require double rotation) :

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

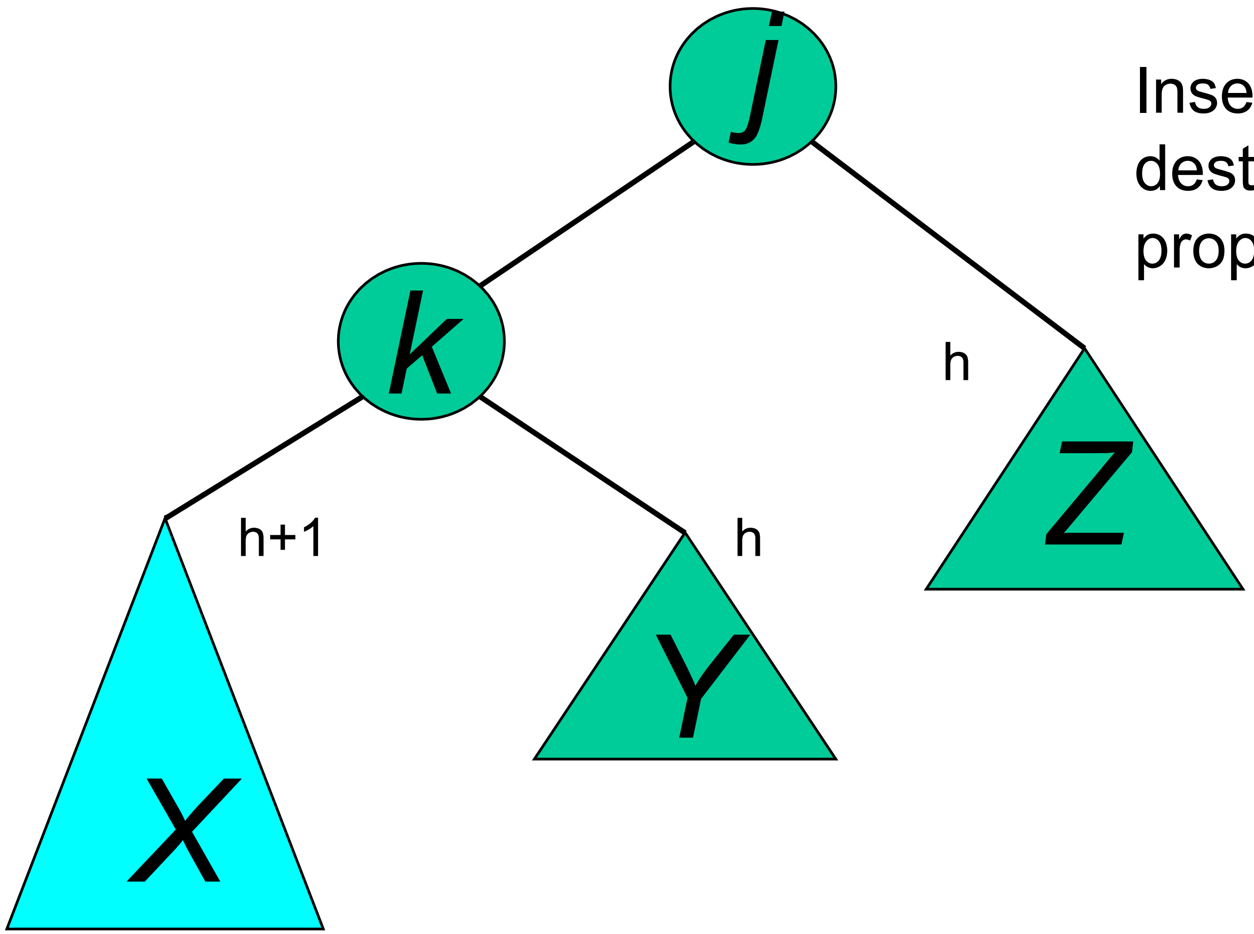
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid AVL subtree

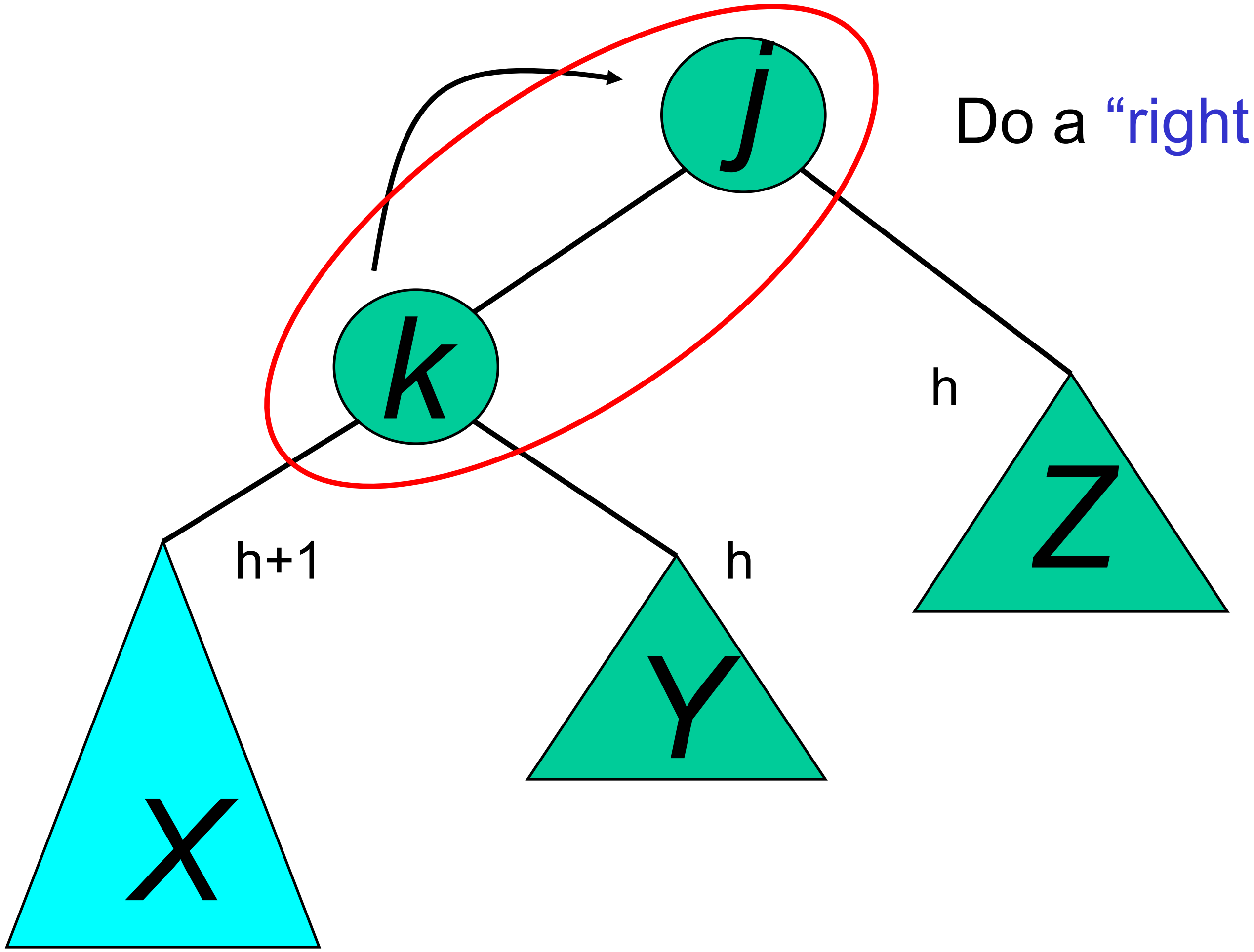


AVL Insertion: Outside Case



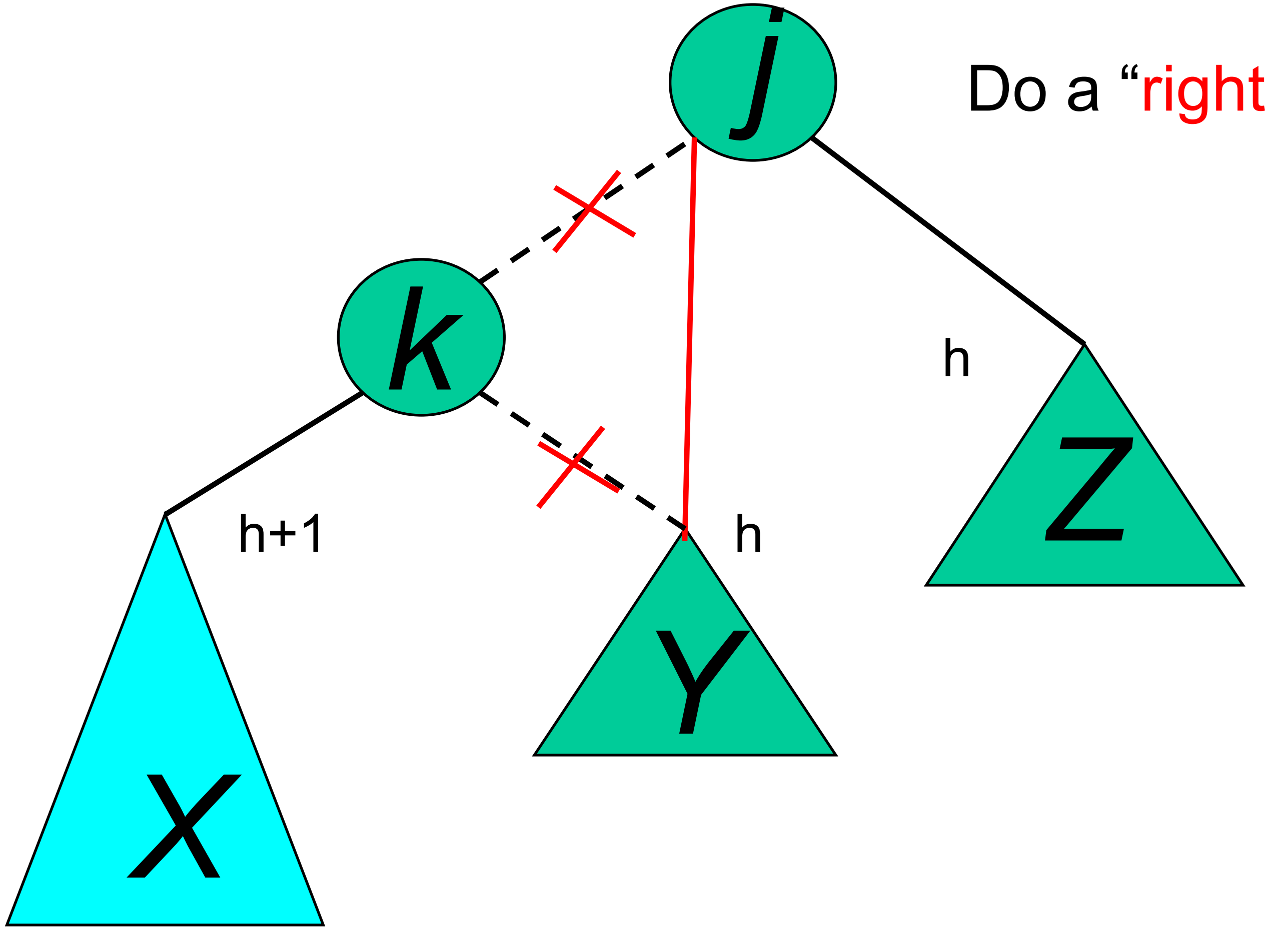
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case



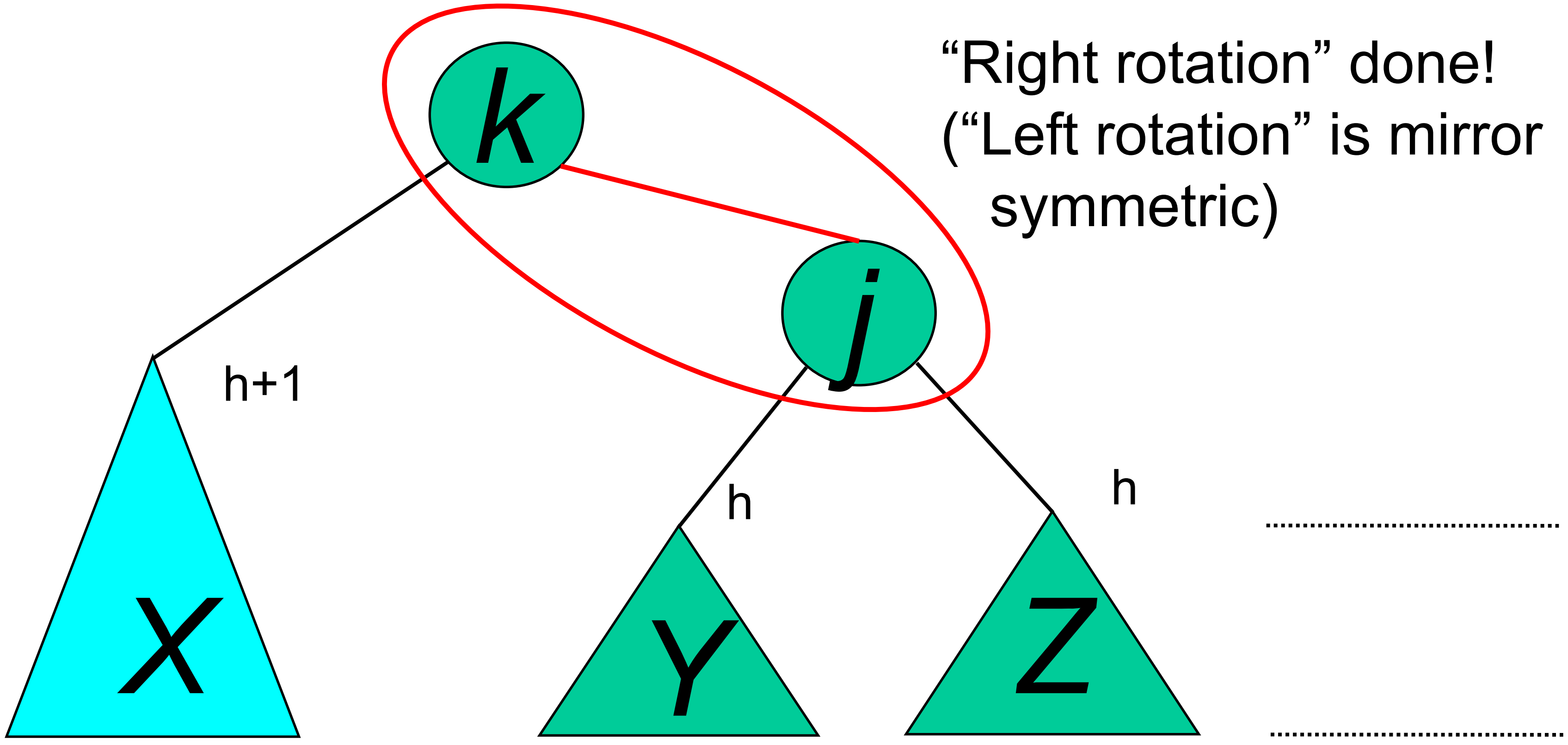
Do a "right rotation"

Single right rotation



Do a “right rotation”

Outside Case Completed

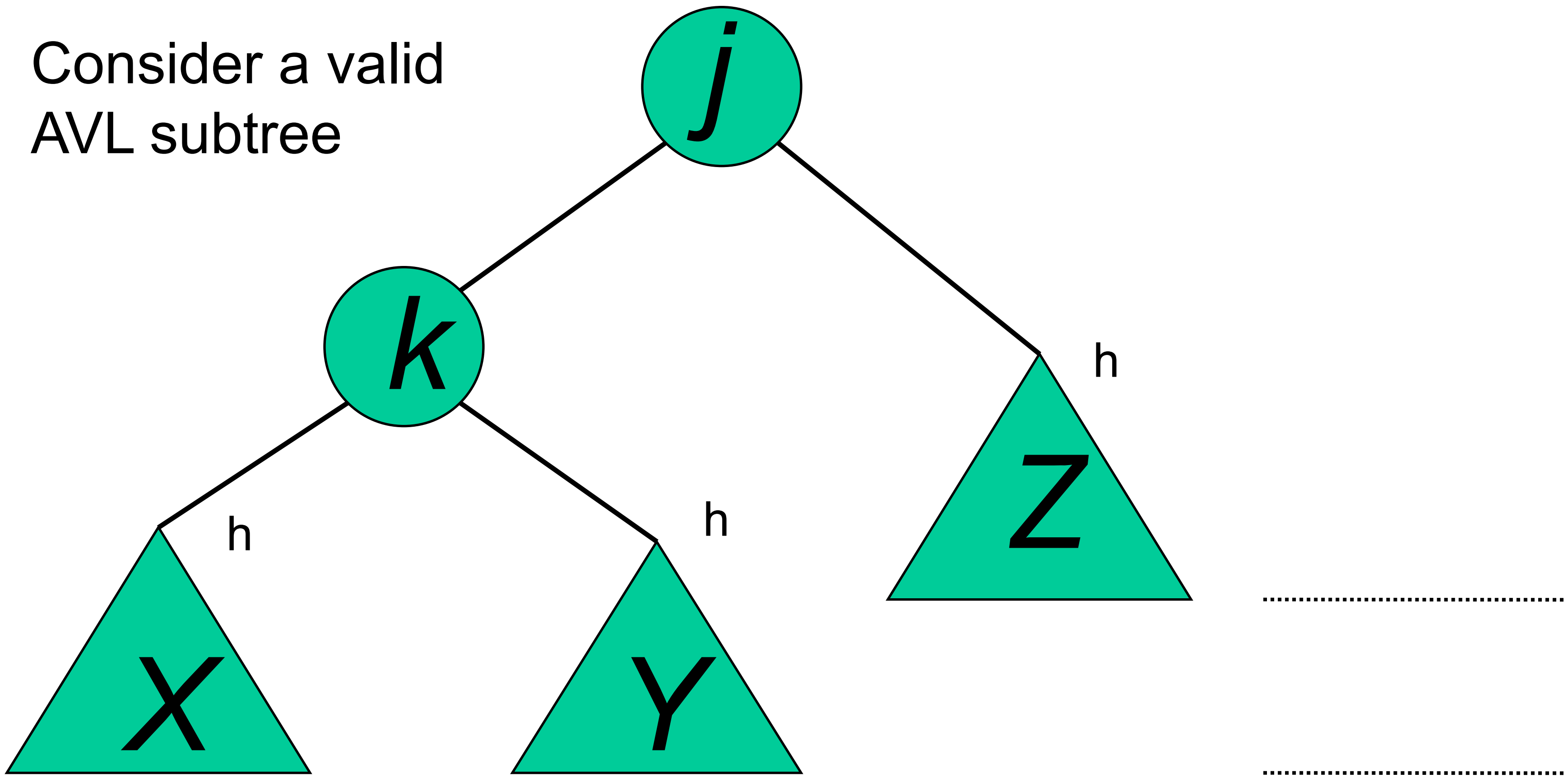


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

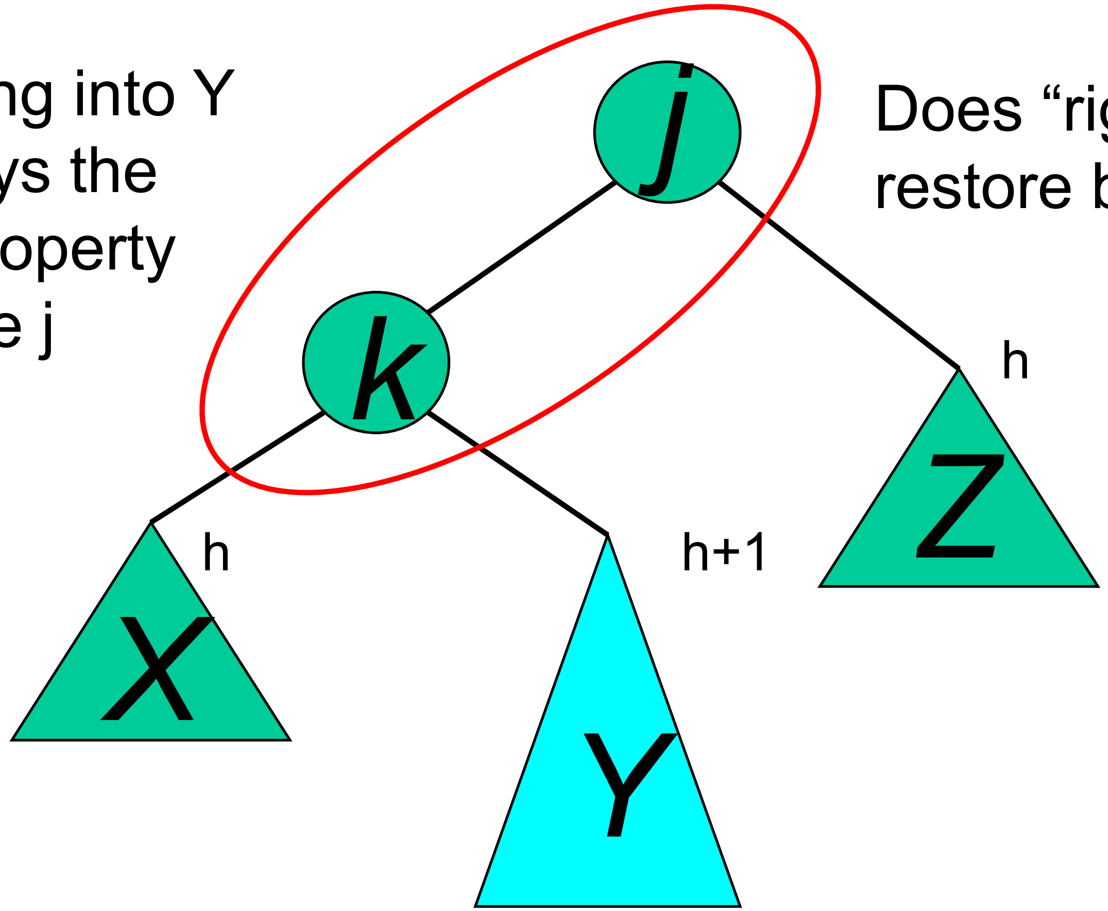
AVL Insertion: Inside Case

Consider a valid
AVL subtree



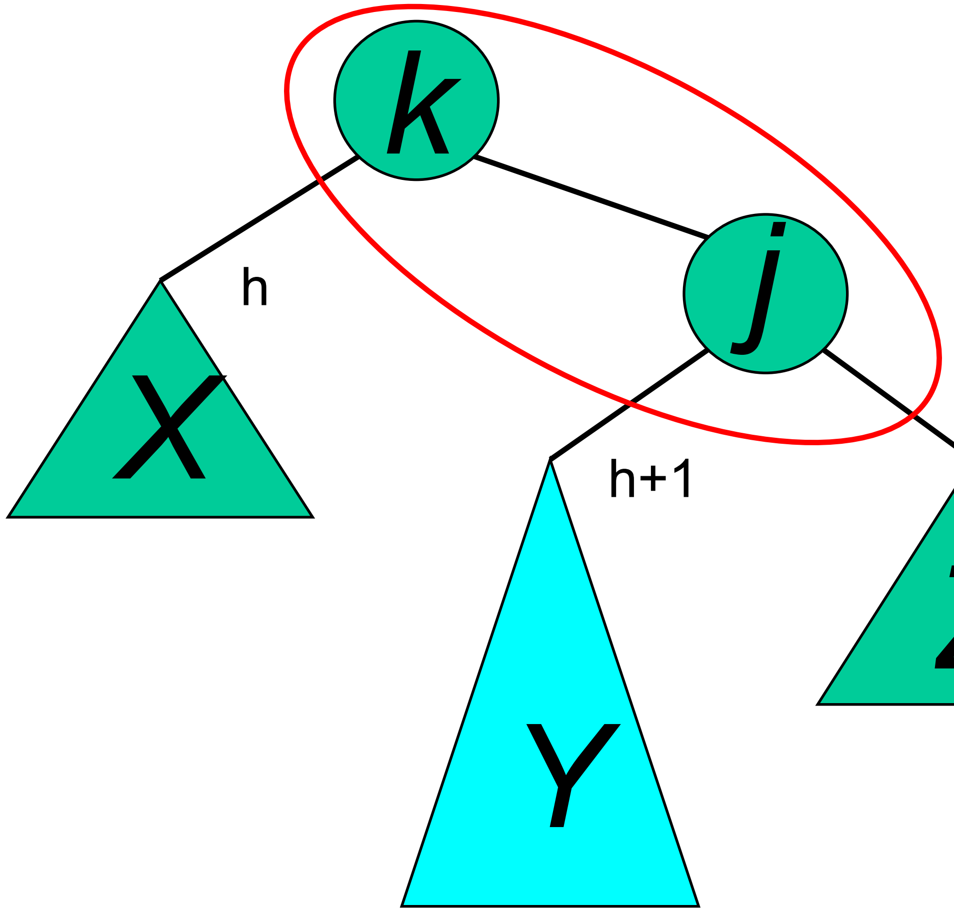
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does “right rotation” restore balance?

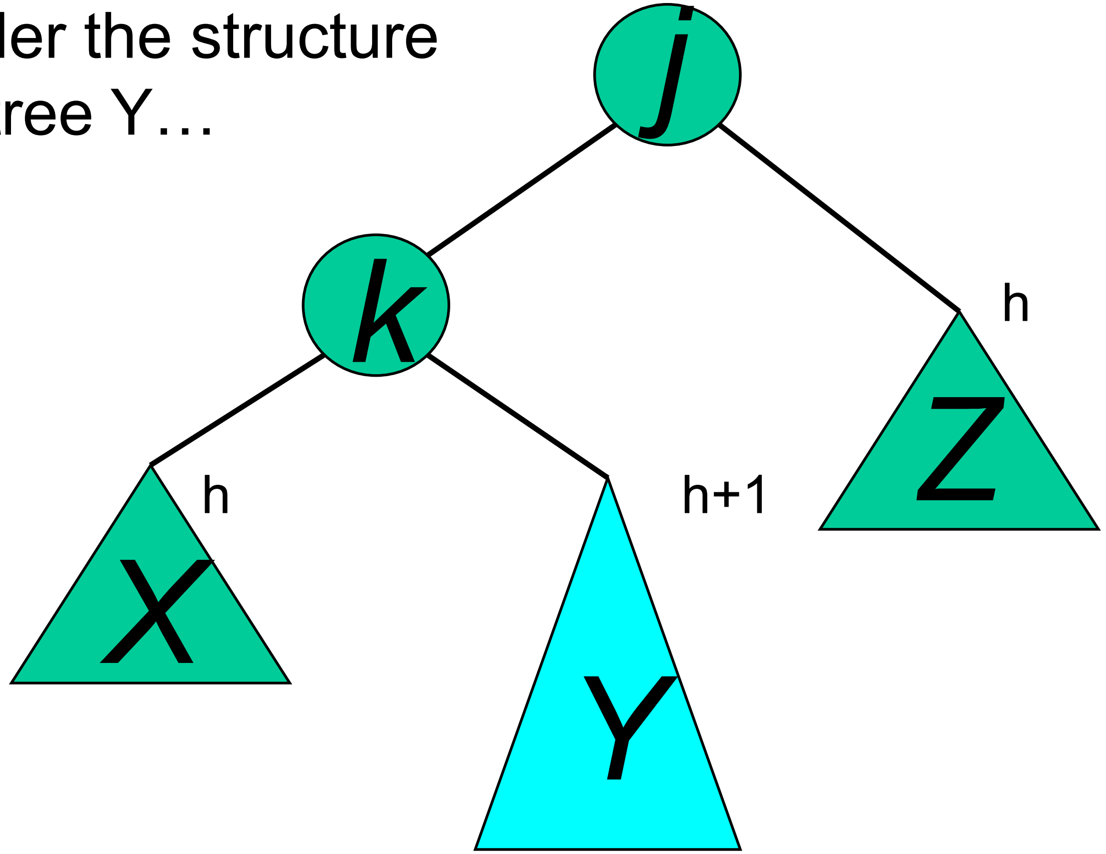
AVL Insertion: Inside Case



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

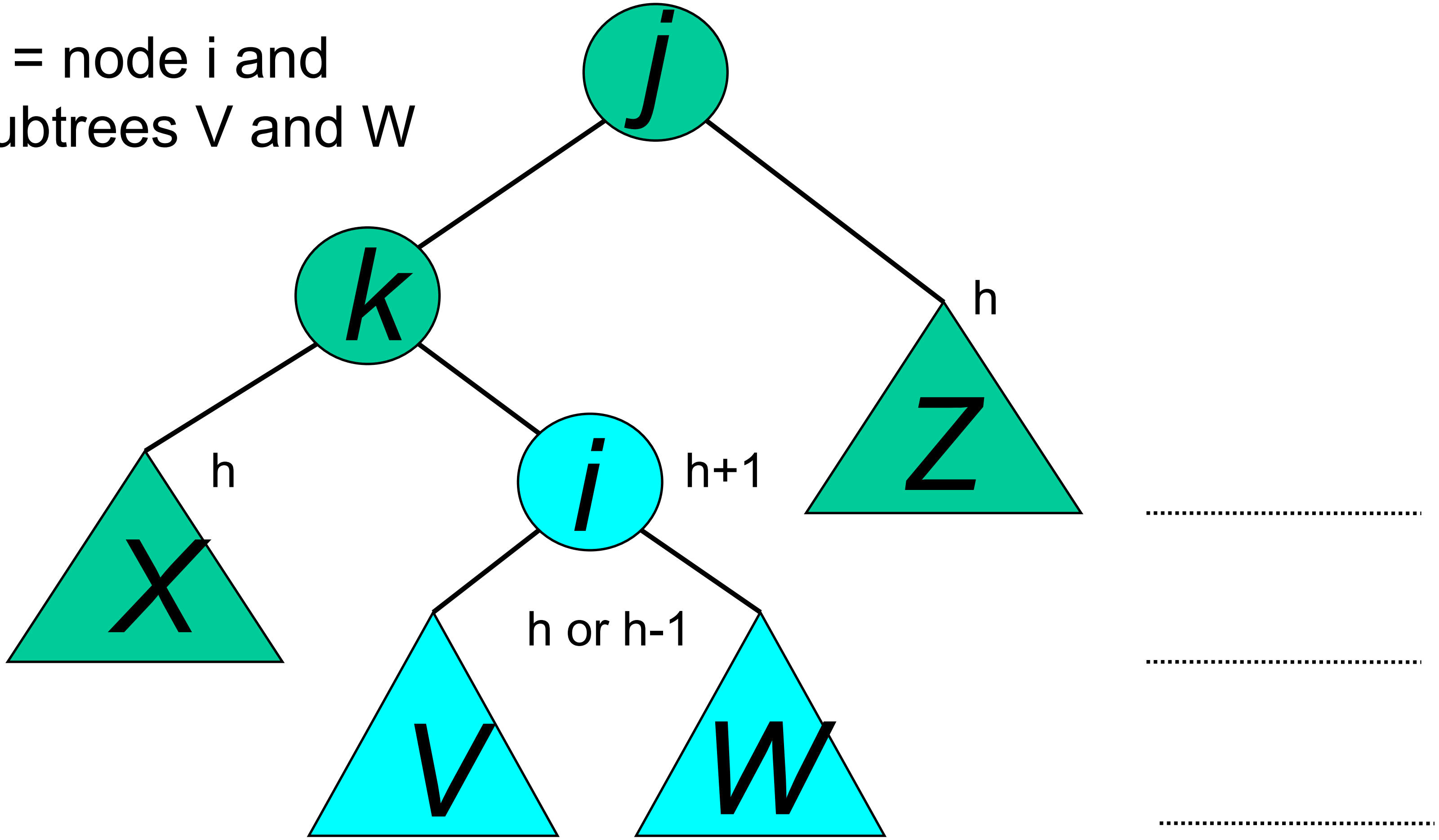
AVL Insertion: Inside Case

Consider the structure of subtree Y…

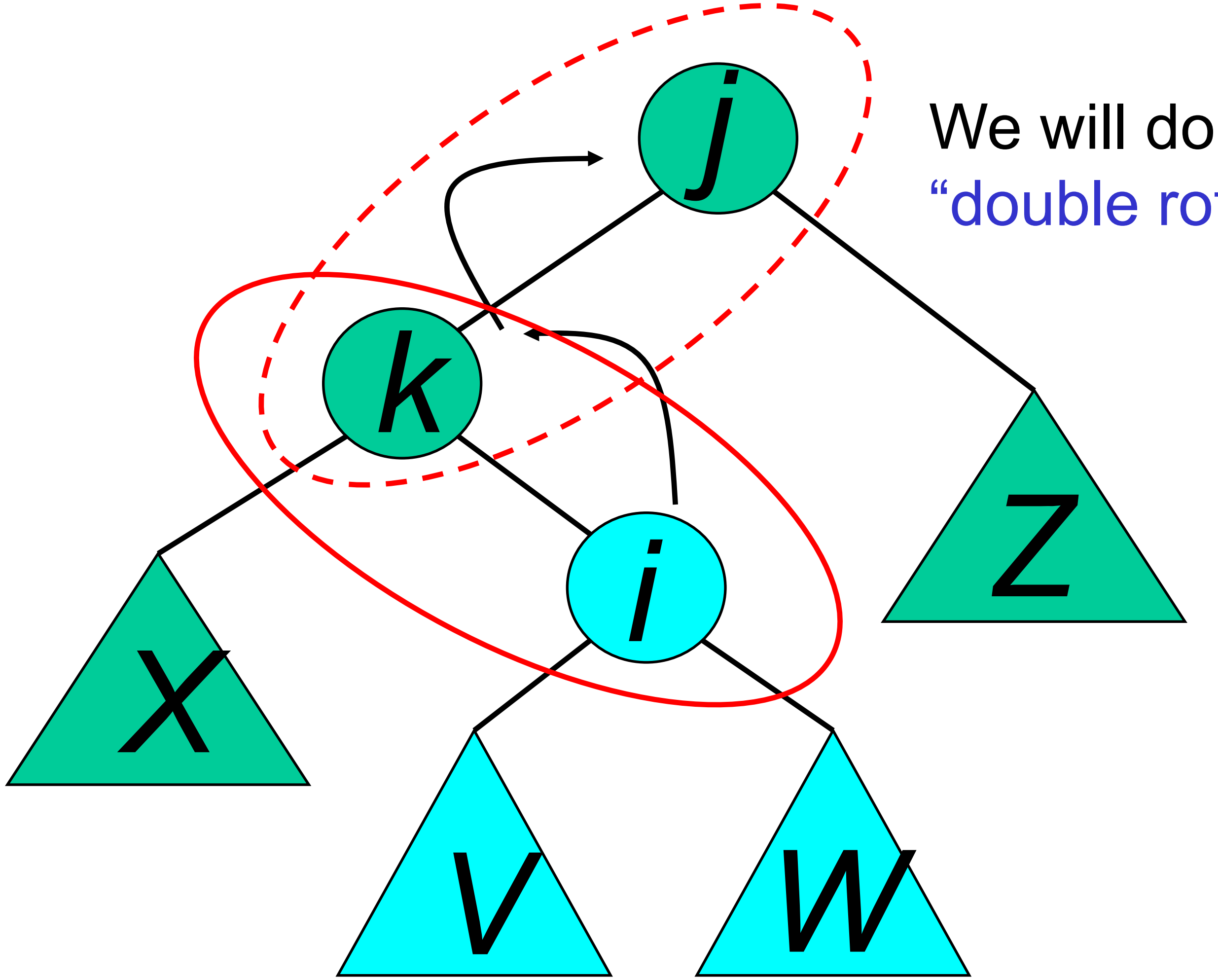


AVL Insertion: Inside Case

Y = node i and
subtrees V and W



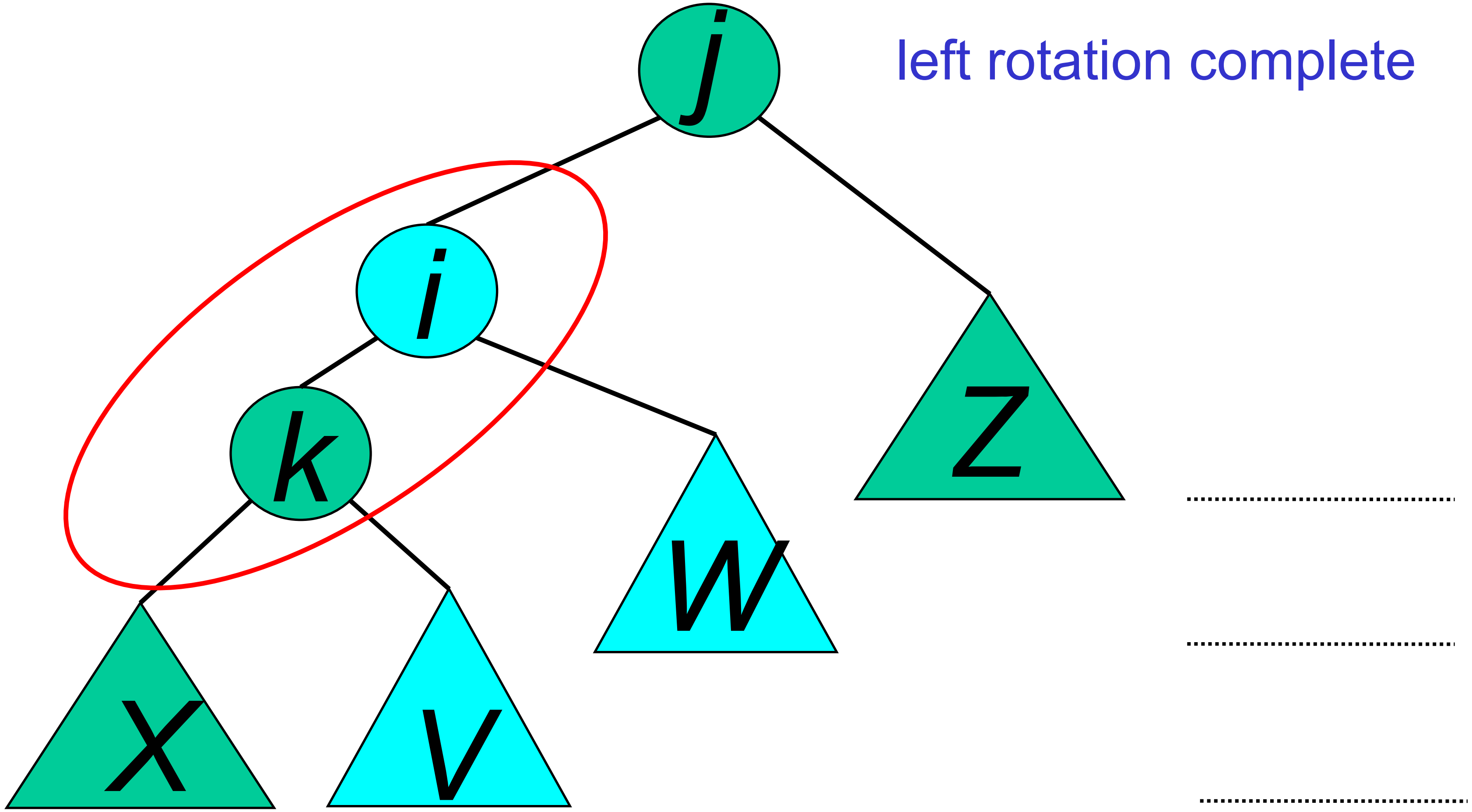
AVL Insertion: Inside Case



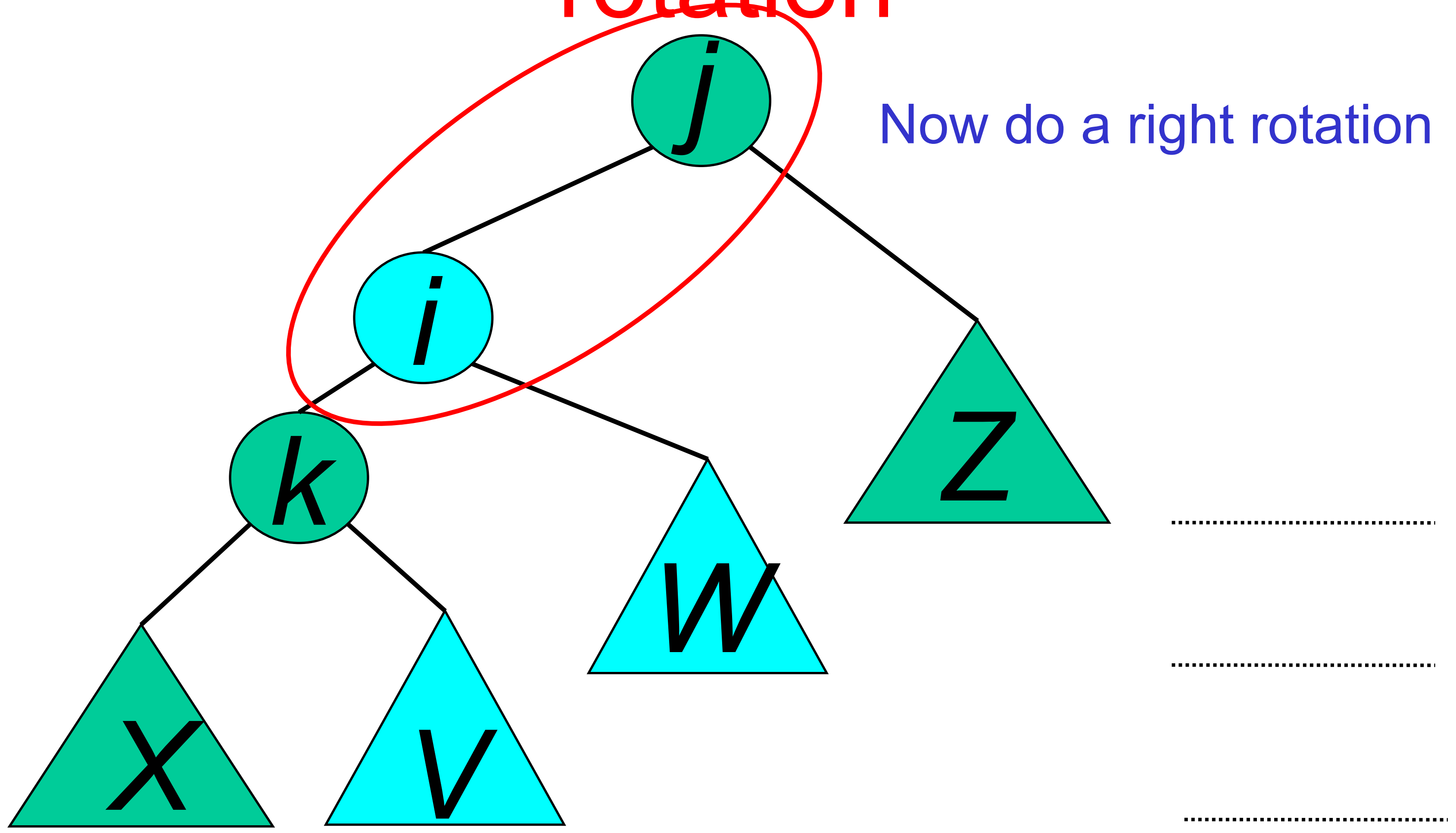
We will do a left-right "double rotation" . . .

Double rotation : first rotation

left rotation complete



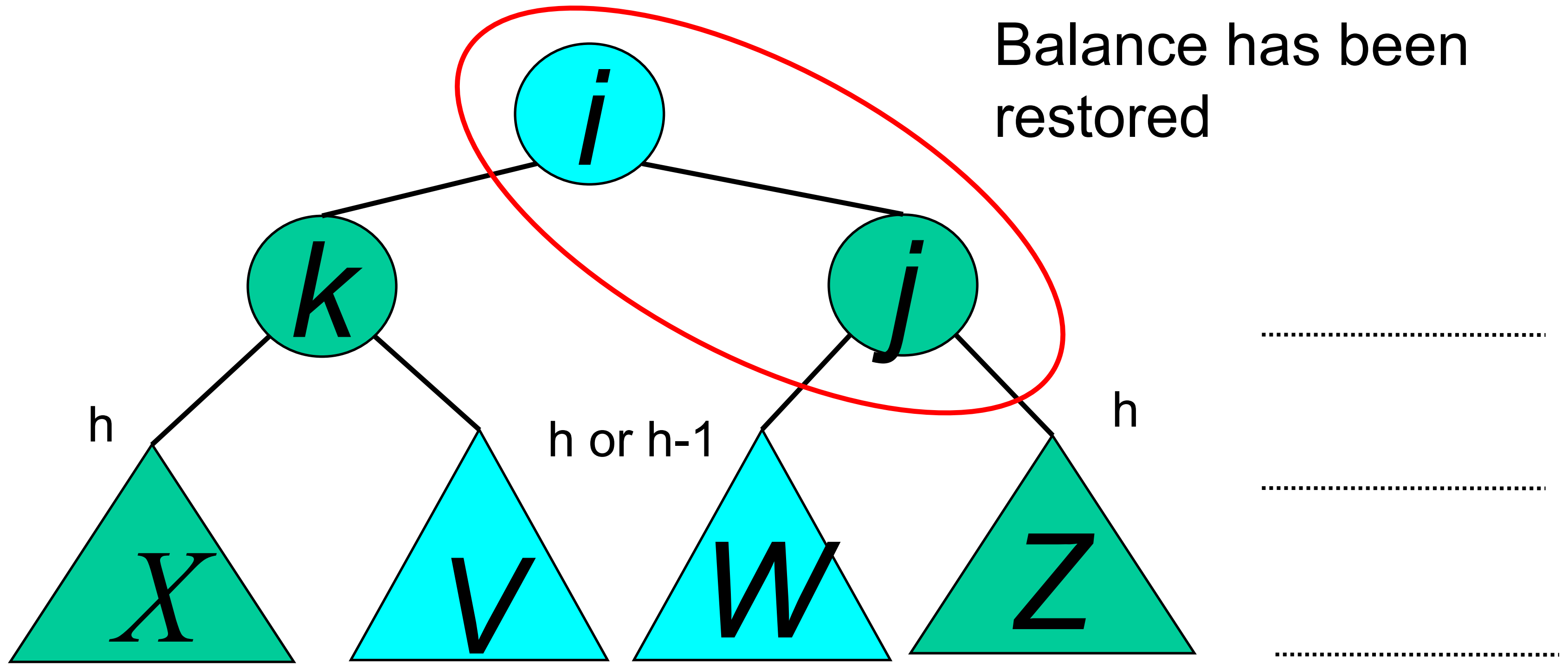
Double rotation : second rotation



Double rotation : second rotation

right rotation complete

Balance has been restored

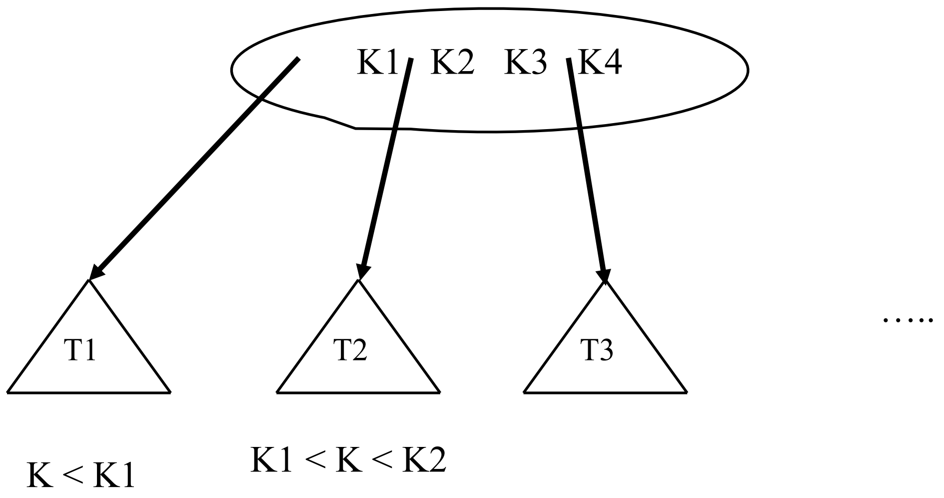


B-Trees

Motivation

- When data is too large to fit in main memory, then the number of disk accesses becomes important.
- A disk access is unbelievably expensive compared to a typical computer instruction.
- One disk access is worth about 200,000 instructions.
- The number of disk accesses will dominate the running time.
- Our goal is to devise a multiway search tree that will minimize file accesses

m-ary Trees



- A node contains multiple keys.
- If each node has m children & there are n keys then the average time taken to search the tree is $\log_m n$.

Searching m-ary Trees

```
for (i==1;i<=m-1;i++) {
    visit subtree to left of k_i
    visit k_i
}
visit subtree to right of k_{m-1}
```

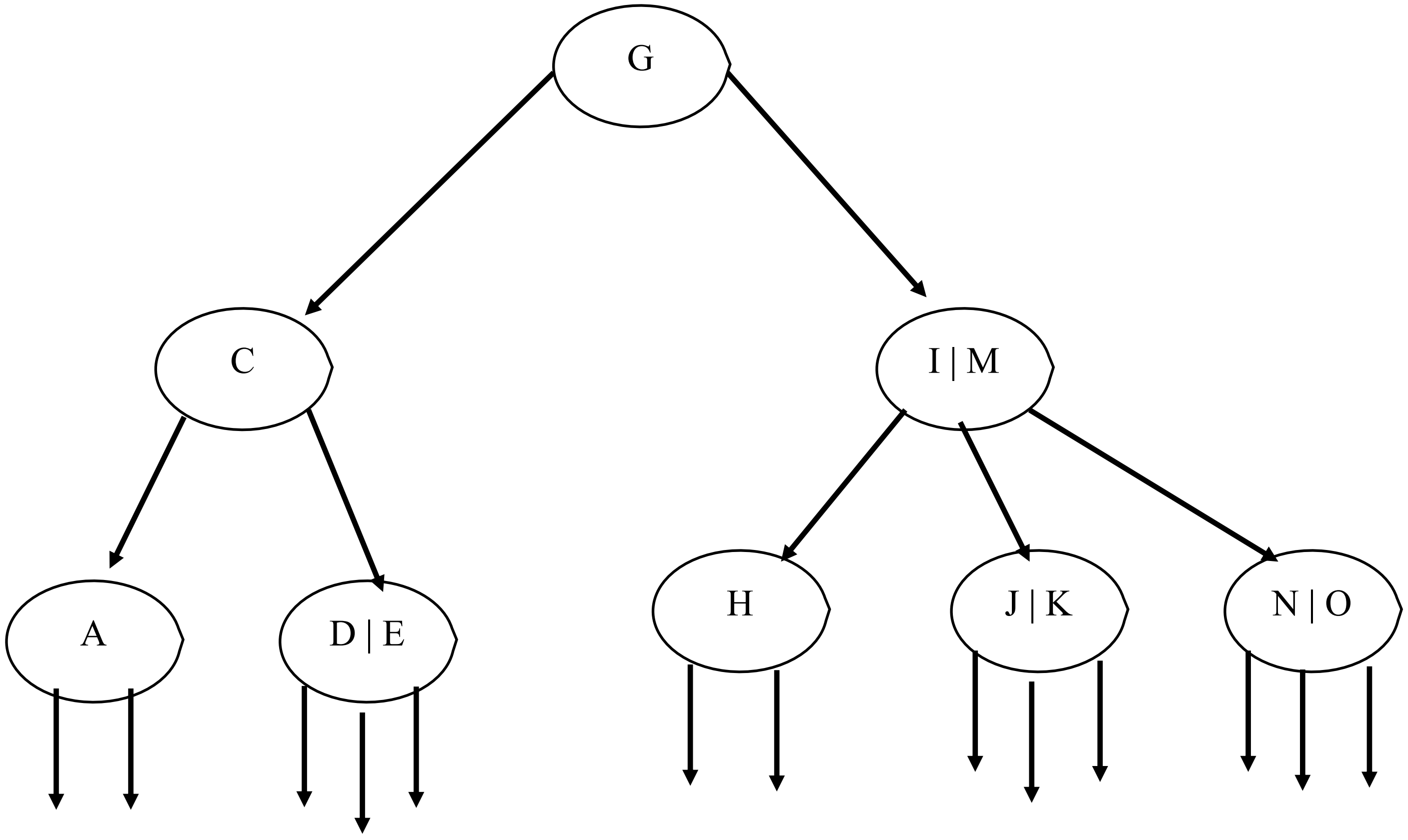
B-Trees & Efficiency

- Used in Mac, NTFS, OS2 for file structure.
- Allow insertion and deletion into a tree structure, based on $\log_m n$ property, where m is the order of the tree.
- The idea is that you leave some key spaces open. So an insert of a new key is done using available space (most cases).
  - Ideal for disk based operations.

Definition of a B-Tree

- Def: B-tree of order m is a tree with the following properties:
  - The root has at least 2 children, unless it is a leaf.
  - No node in the tree has more then m children.
  - Every node except for the root and the leaves have at least $\lceil m/2 \rceil$ children.
  - All leaves appear at the same level.
  - An internal node with k children contains exactly k-1 keys.

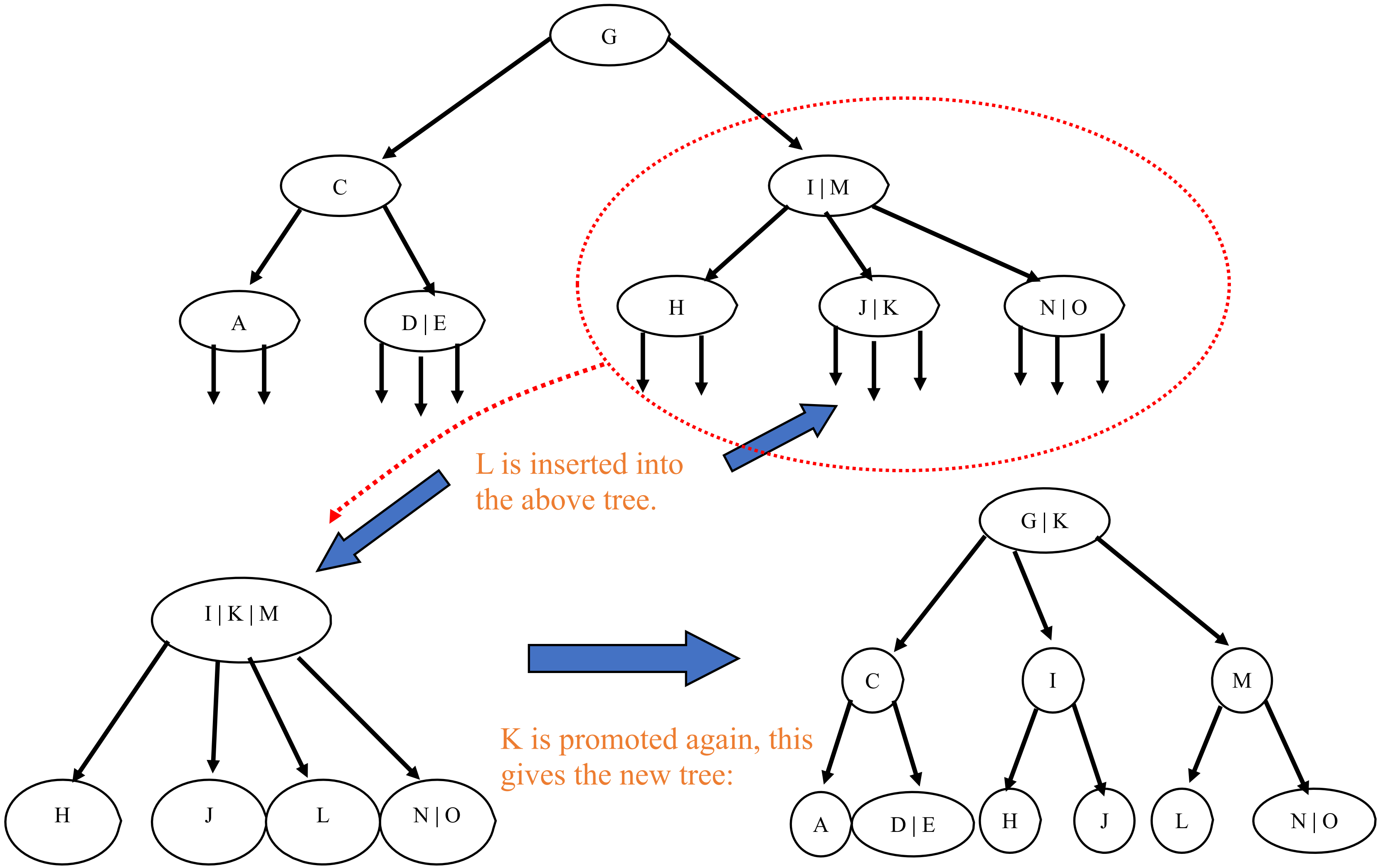
2-3 Trees



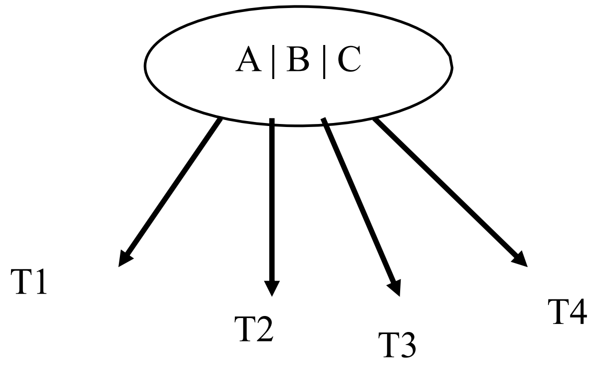
Insertion

- Insert k_i into B-tree of order m.
  - We find the insertion point (in a leaf) by doing a search.
  - If there is room then enter k_i.
  - Else, promote the middle key to the parent & split the node into nodes around the middle key.
- If the splitting backs up to the root, then
  - Make a new root containing the middle key.
- Note: the tree grows from the leaves, balance is always maintained.

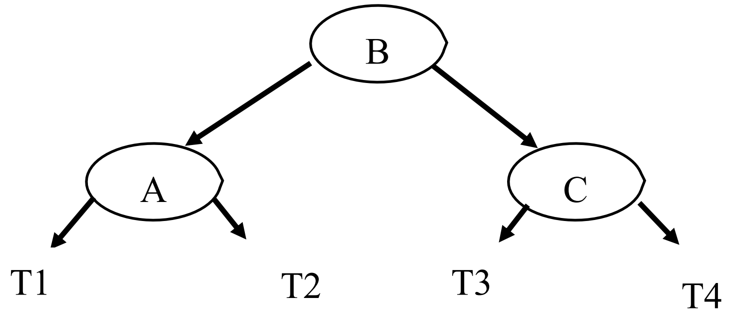
Insertion Example



Splitting Nodes



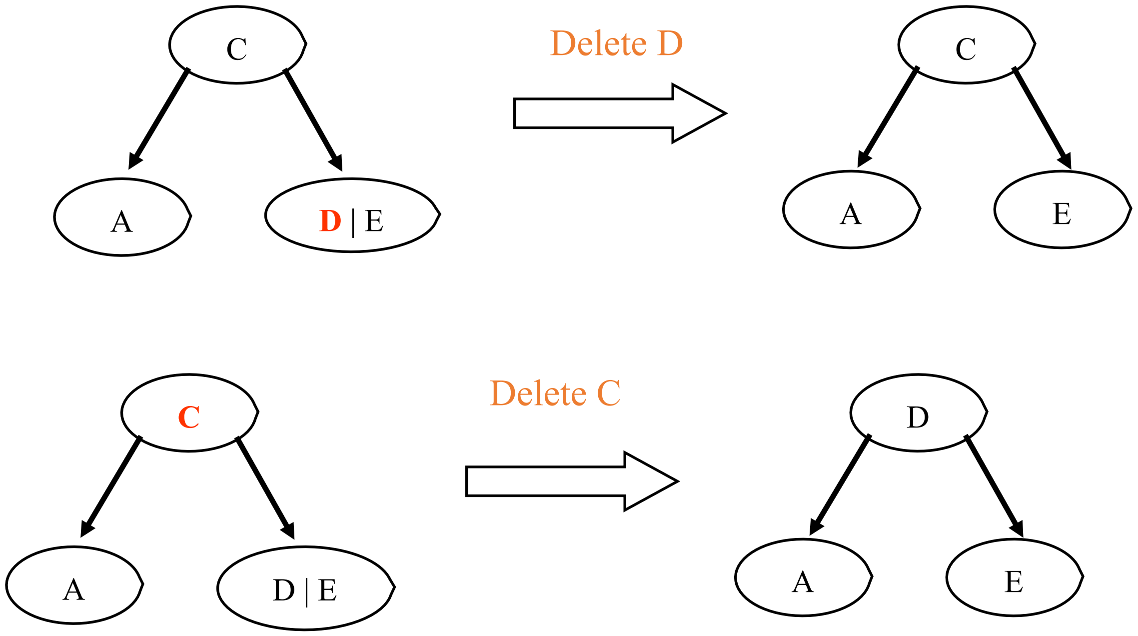
- Middle key is promoted
- Creating a new root



Deletion

- If the entry to be deleted is not in a leaf, swap it with its successor (or predecessor) under the natural order of the keys. Then delete the entry from the leaf.
- If leaf contains more than the minimum number of entries, then one can be deleted with no further action.

Deletion Example 1



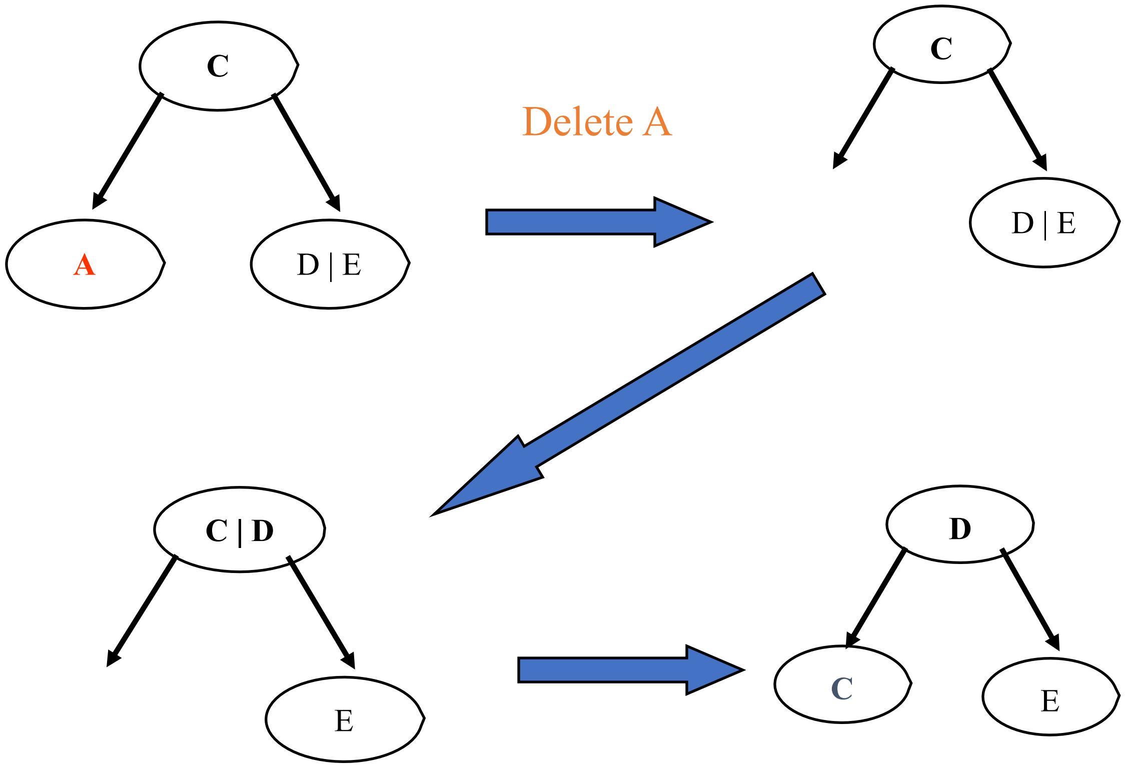
Successor is promoted, Element D

C is Deleted.

Deletion Cont...

- If the node contains the minimum number of entries, consider the two immediate siblings of the parent node:
- If one of these siblings has more than the minimum number of entries, then redistribute one entry from this sibling to the parent node, and one entry from the parent to the deficient node.
  - This is a rotation which balances the nodes
  - Note: all nodes must comply with minimum entry restriction.

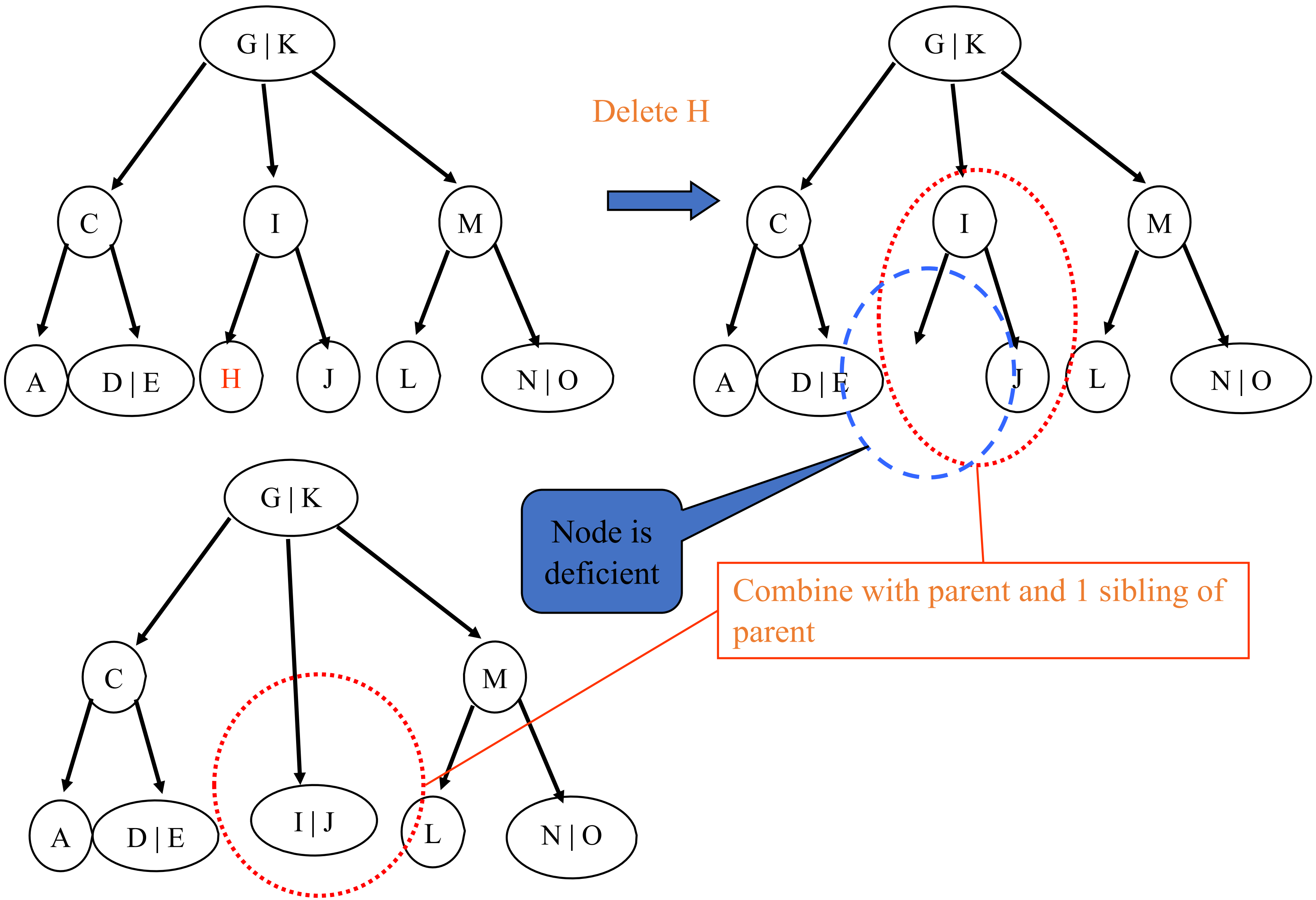
Deletion Example 2



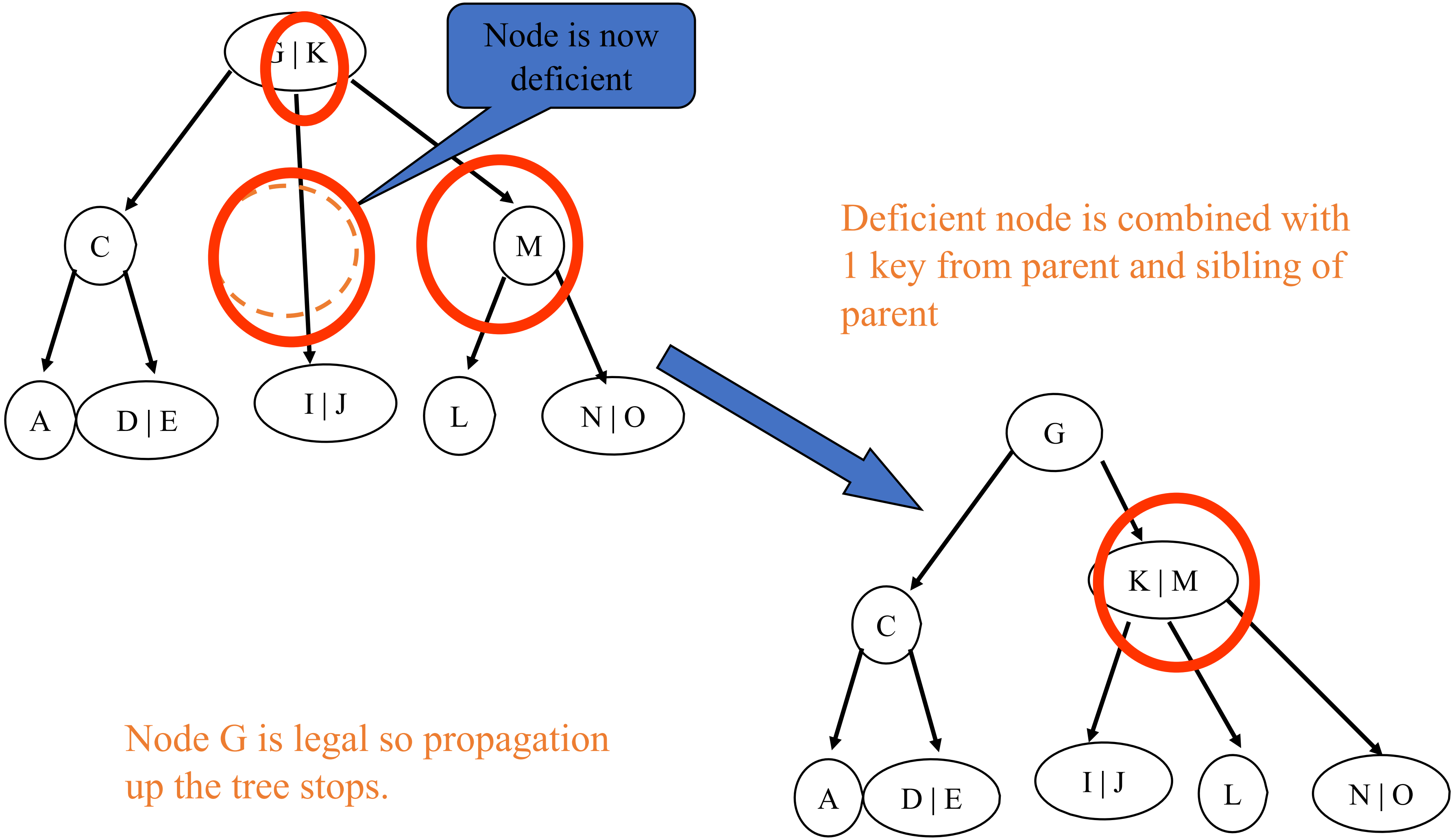
Deletion Cont...

- If both immediate siblings have exactly the minimum number of entries, then merge the deficient node with one of the immediate sibling node and one entry from the parent node.
- If this leaves the parent node with too few entries, then the process is propagated upward.

Deletion Example 3



Deletion Example 3 Cont..



Deficient node is combined with 1 key from parent and sibling of parent

Node G is legal so propagation up the tree stops.

Review of Deletions

- All Deletions take place in leaf nodes
  - To delete a internal key swap it with its successor or predecessor which is a leaf.
  - Then Delete
- Deficient Nodes are legalized by:
  - Rotation with a sibling and parent.

  OR
  - Combining with key from parent and sibling
  - Propagating up the tree until a legal node is encountered.