2021-2022

Data Structures and Algorithms (II) – Balanced Search Trees

Dr. Dapeng Dong

The Importance of Balanced Trees

- Most of the primitive operations on search trees run in $\boldsymbol{O(h)}$ time on a tree of height $\boldsymbol{h}$

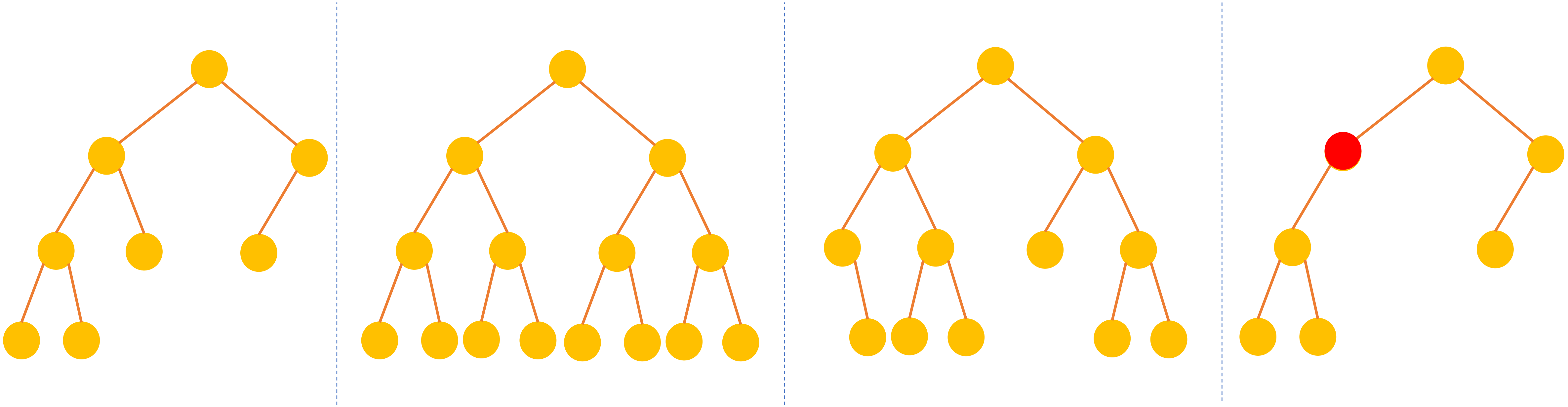
- The average height of a node in a binary search tree on $\boldsymbol{N}$ node is $\boldsymbol{\Theta}(\lg \boldsymbol{N})$

- Given $\boldsymbol{N}$ elements to be inserted in a binary search tree, the height of the tree depends on the order of the elements inserted
  - E.g., given the set of nodes with their key values from 1 – 7,
    - if the insertion order is {1, 2, 3, 4, 5, 6, 7}, the depth of the tree will be 6
    - If the insertion order is { 4, 2, 5, 1, 3, 6, 7}, the depth of the tree will be 2

- Deletion of nodes may result in an unbalanced tree

AVL Trees

- An **AVL** (Adelson-Velsky and Landis) tree is a self-balancing *binary search tree*
- For every node in the tree, the height of the left and right subtree can differ by at most one



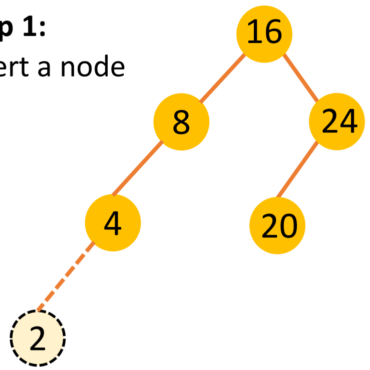
Reading: AdelsonVelskii, M., & Landis, E. M. (1963). An algorithm for the organization of information. JOINT PUBLICATIONS RESEARCH SERVICE WASHINGTON DC.

Insertion (1)

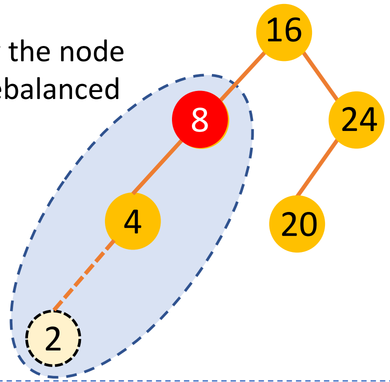
Scenario 1: insert into the **left subtree** of the **left child** of the node that is to be rebalanced.

Using Single Rotation

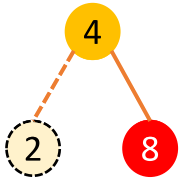
Step 1:
insert a node



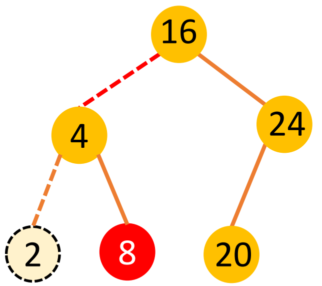
Step 2:
Identify the node to be rebalanced



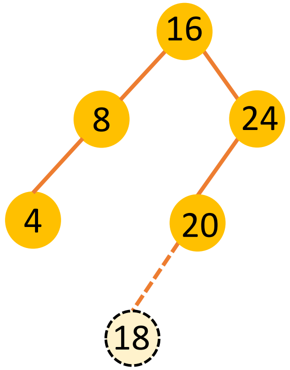
Step 3:
Rotate the subtree by lifting the left child (Node 4) of the node that is to be rebalanced (Node 8) as the new root of the subtree



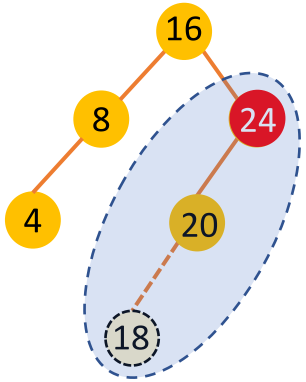
Step 4:
Make the new root of the rotated subtree (Node 4) as a child of the parent of the node to be rebalanced identified in **Step 2** (Node 16)



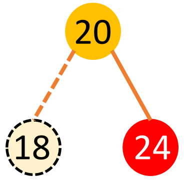
Step 1



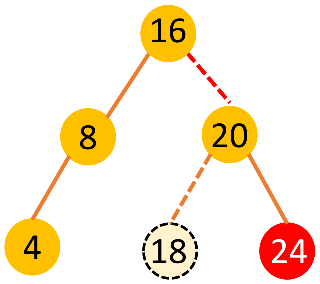
Step2



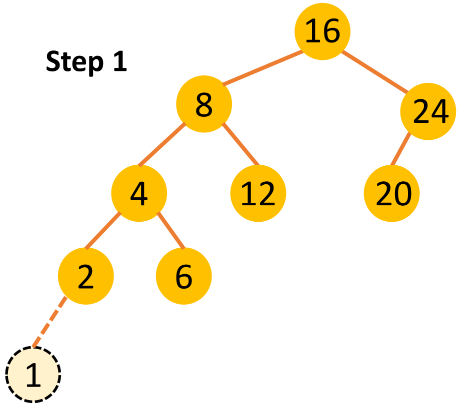
Step 3



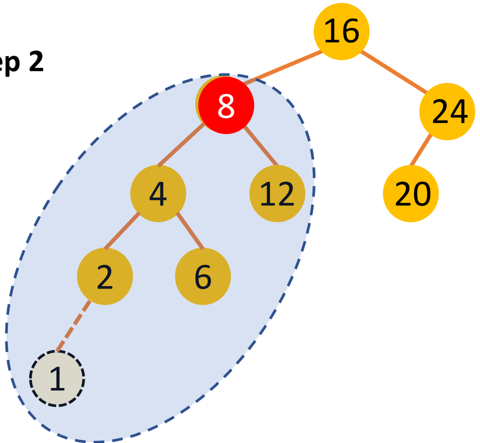
Step 4



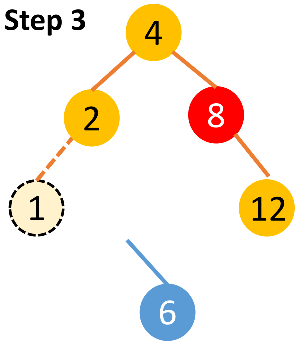
Step 1



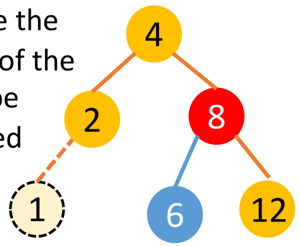
Step 2



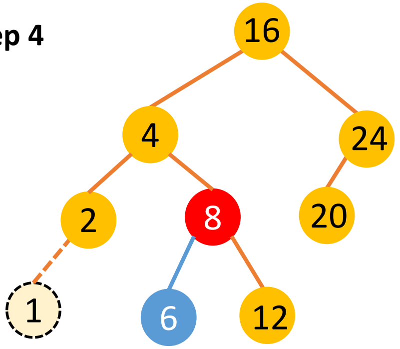
Step 3



Step 3.1:
Make the right child of the new root to be the left child of the node to be rebalanced



Step 4

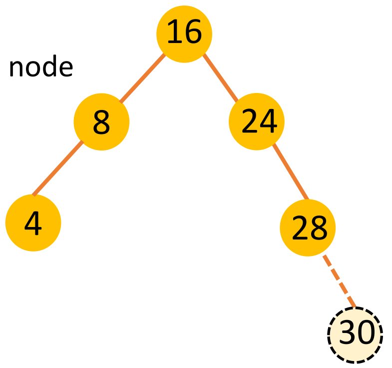


Insertion (2)

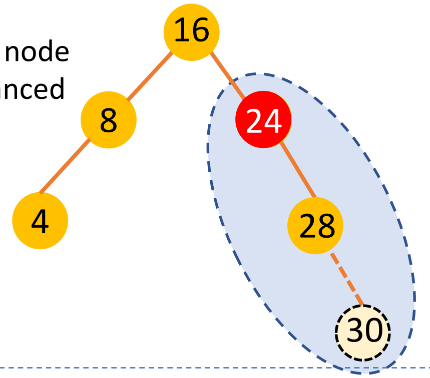
Scenario 2: insert into the **right subtree** of the **right child** of the node that is to be rebalanced.

Using Single Rotation

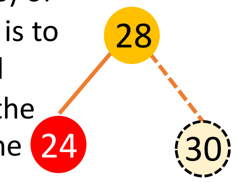
Step 1:
insert the node



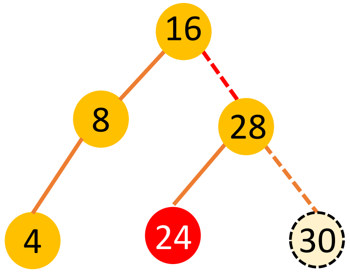
Step 2:
Identify the node to be rebalanced



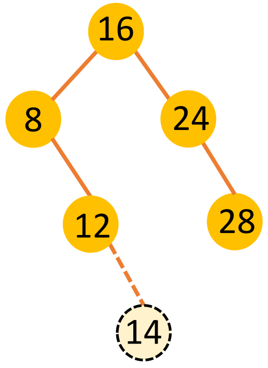
Step 3:
Rotate the subtree by lifting the right child (Node 28) of the node that is to be rebalanced (Node 24) as the new root of the subtree



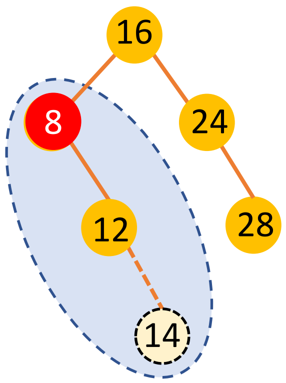
Step 4:
Make the new root of the rotated subtree (Node 28) as a child of the parent of the node to be rebalanced identified in **Step 2** (Node 16)



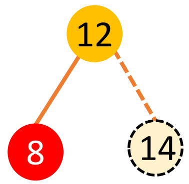
Step 1



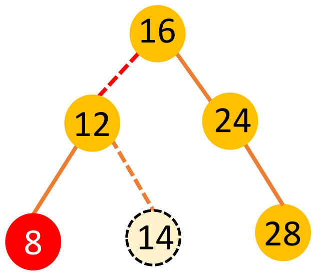
Step2



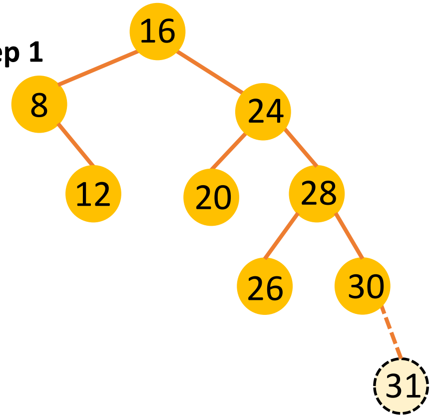
Step 3



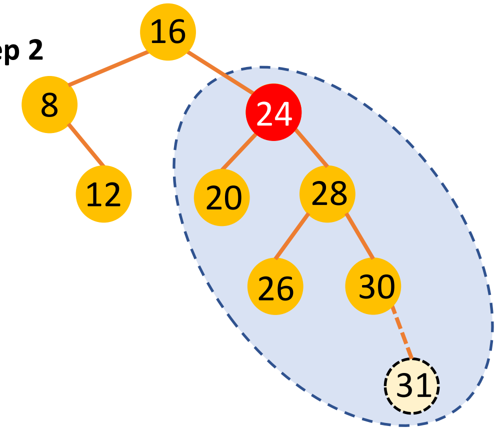
Step 4



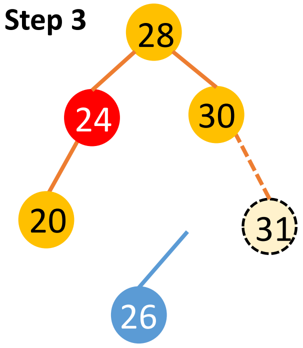
Step 1



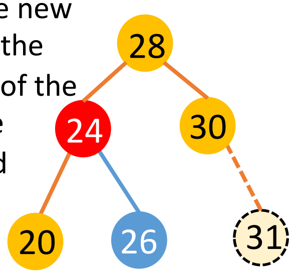
Step 2



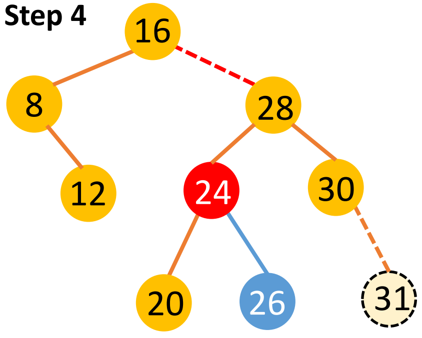
Step 3



Step 3.1:
Make the left child of the new root to be the right child of the node to be rebalanced



Step 4

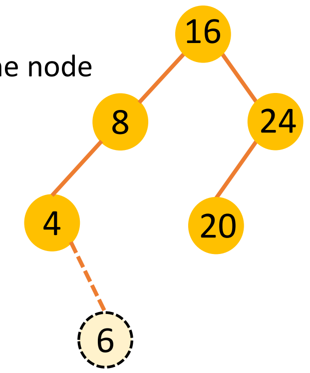


Insertion (3)

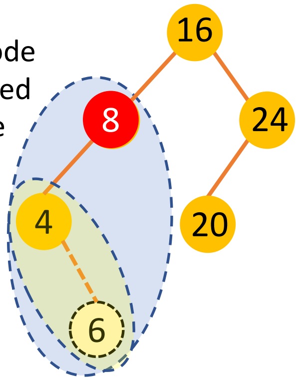
Scenario 3: insert into the right subtree of the left child of the node that is to be rebalanced.

Using Double Rotation

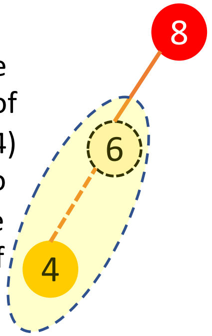
Step 1:
insert the node



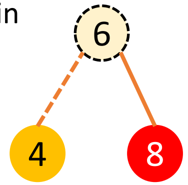
Step 2:
Identify the node to be rebalanced and its subtree



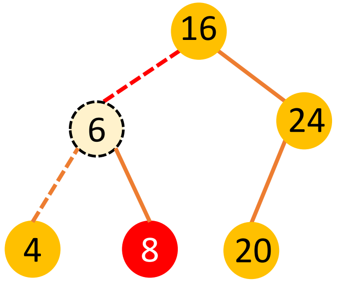
Step 3 (rotate 1):
Rotate its subtree (yellow) by lifting the right child (Node 6) of the left child (Node 4) of the node that is to be rebalanced (Node 8) as the new root of the subtree



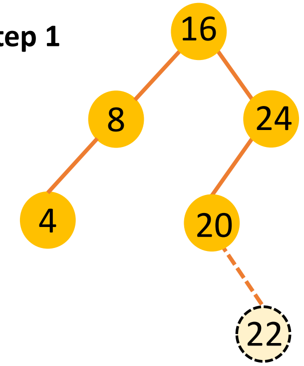
Step 4 (rotate 2):
Rotate the subtree following the steps illustrated in ***Scenario 1***



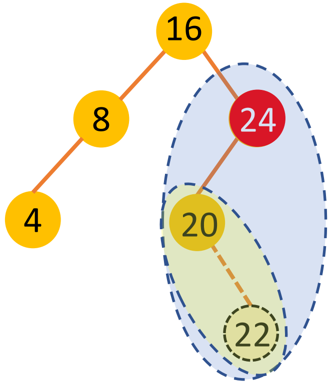
Step 5:
Make the new root as a child of the parent of the node to be rebalanced



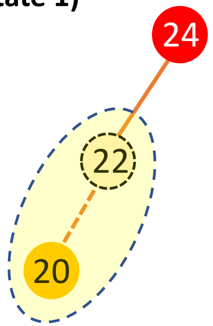
Step 1



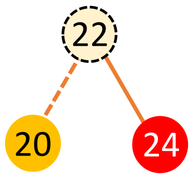
Step2



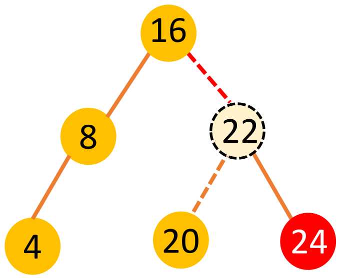
Step 3 (rotate 1)



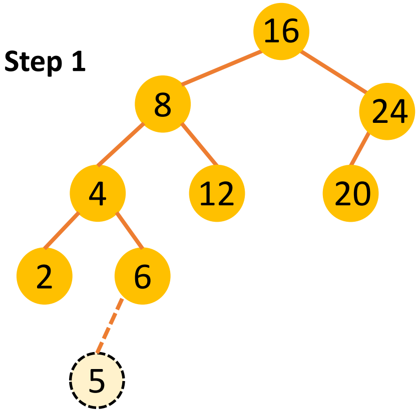
Step 4 (rotate 2)



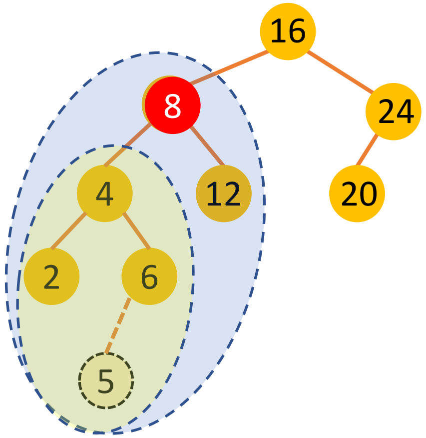
Step 5



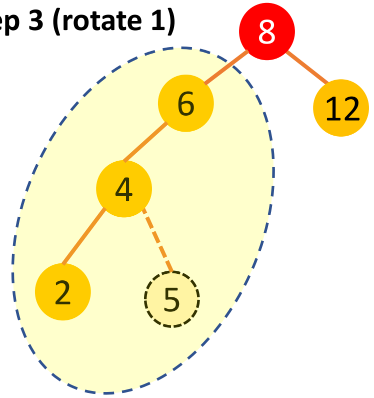
Step 1



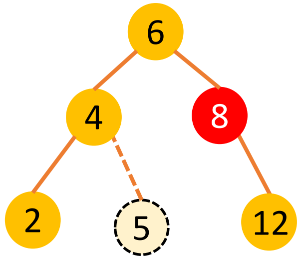
Step 2



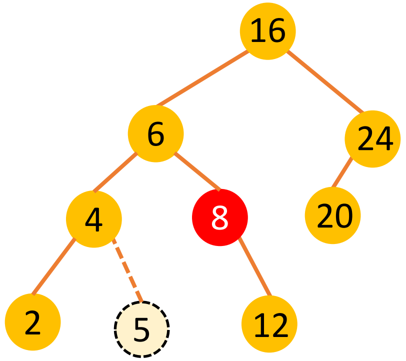
Step 3 (rotate 1)



Step 4 (rotate 2)

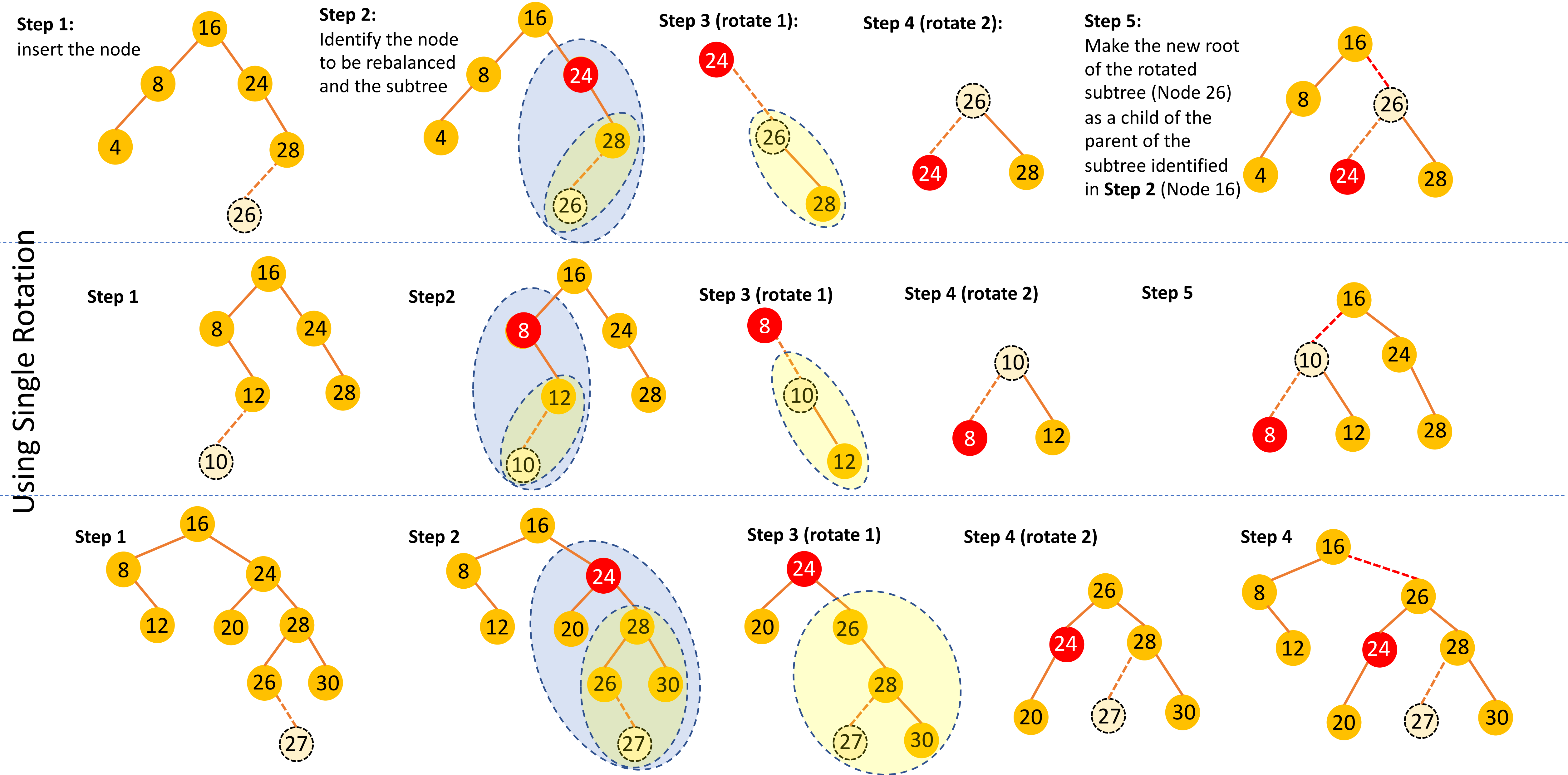


Step 5



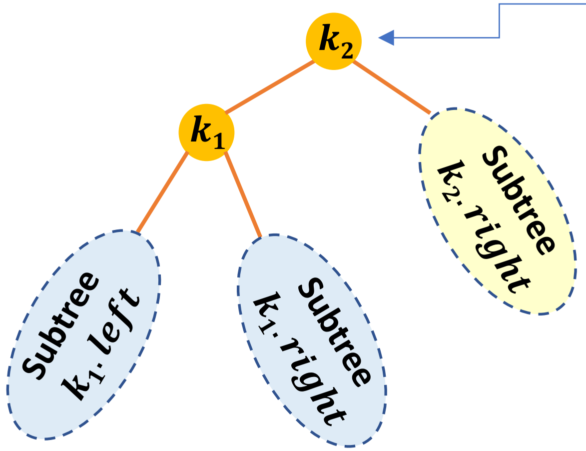
Insertion (4)

Scenario 4: insert into the **left subtree** of the **right child** of the node that is to be rebalanced.

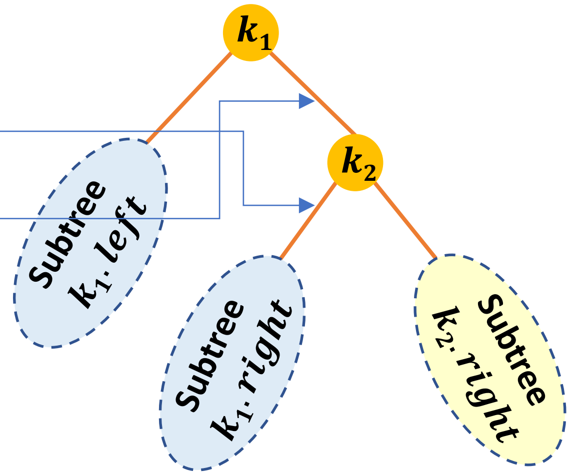


Rotation Summary (Single Rotation)

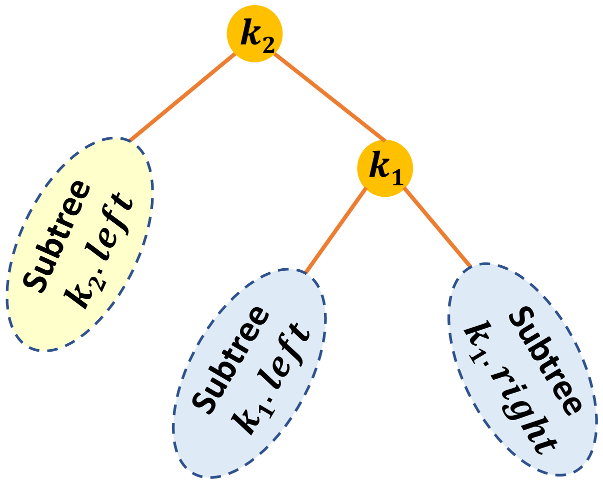
Single rotate with left child



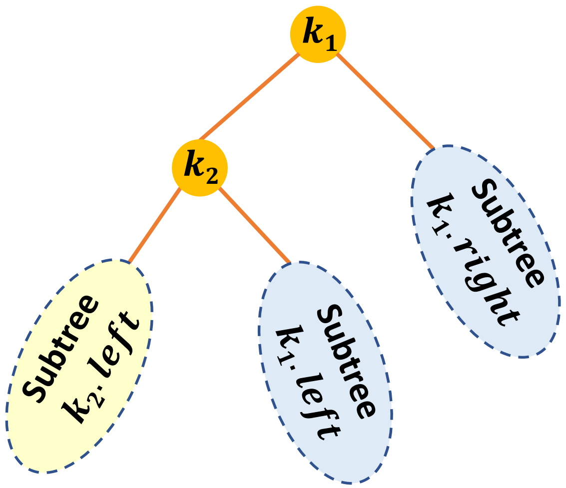

```
// Given the node to be rebalanced: Node k2
private Node rotateWithLeftChild(Node k2) {
    Node k1 = k2.leftChild; // The leftChild of k2 is k1

    // If k1 has a right child, make it the left child of k2
    k2.leftChild = k1.rightChild;

    k1.rightChild = k2;

    // Update the height of the nodes.
    // NOTE, the height increase from leaves to root
    k2.height = Math.max(height(k2.leftChild), height(k2.rightChild)) + 1;
    k1.height = Math.max(height(k1.leftChild), k2.height) + 1;
    return k1;
}
```



Single rotate with right child

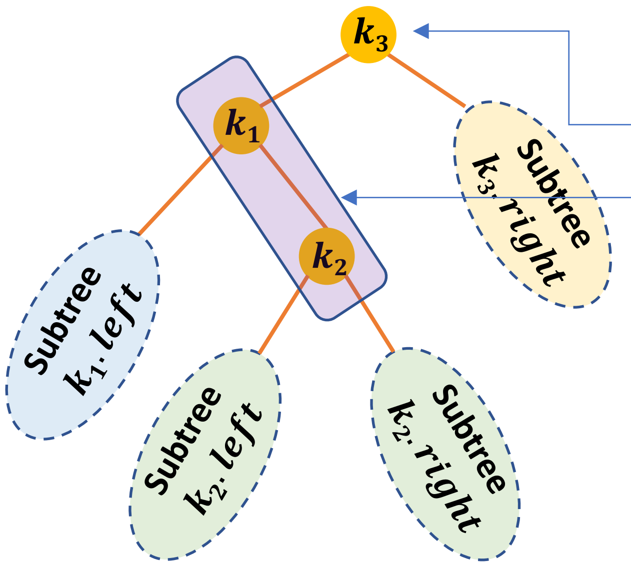



```
// Given the node to be rebalanced: Node k2
private Node rotateWithRightChild(Node k2) {
    Node k1 = k2.lrightChild; // The rightChild of k2 is k1

    // If k1 has a left child, make it the right child of k2
    k2.rightChild = k1.leftChild;

    k1.leftChild = k2;

    // Update the height of the nodes.
    // NOTE, the height increase from leaves to root
    k2.height = Math.max(height(k2.leftChild), height(k2.rightChild)) + 1;
    k1.height = Math.max(height(k1.rightChild), k2.height) + 1;
    return k1;
}
```

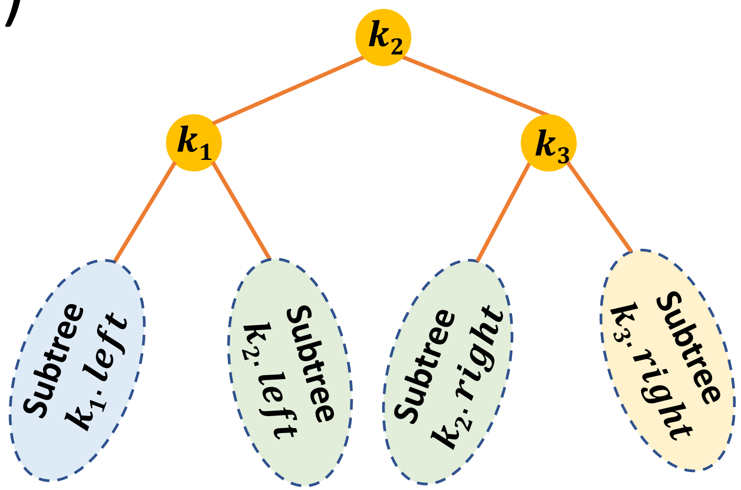


Rotation Summary (Double Rotation)

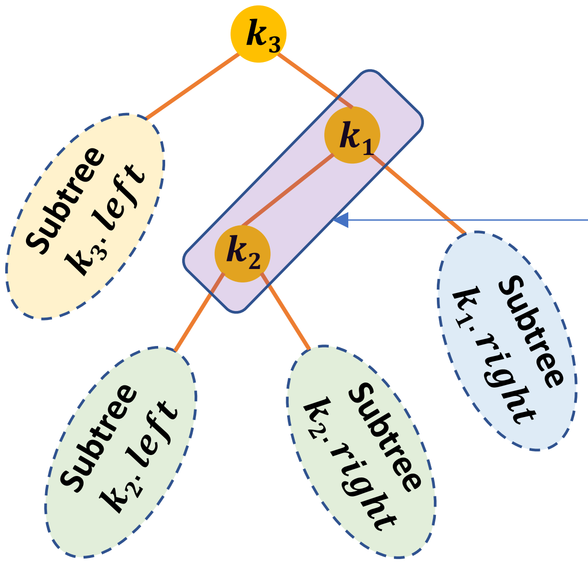
Double rotate with left child



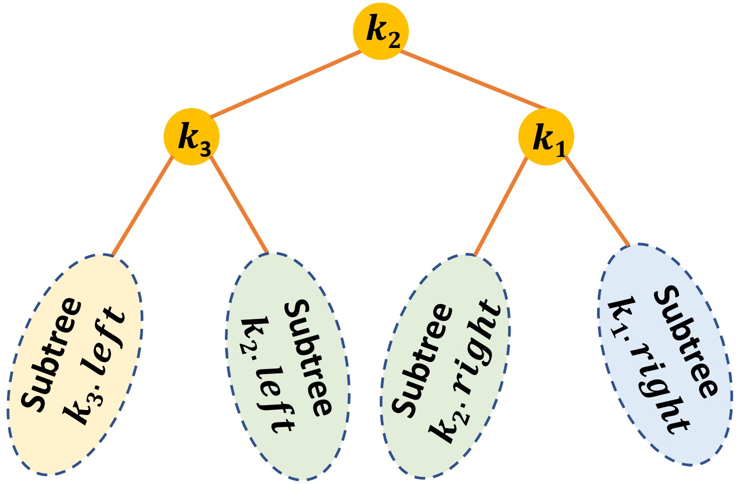

```
// Given the node to be rebalanced (Node k3)
private Node doubleWithLeftChildRightSubtree(Node k3) {

    k3.leftChild = rotateWithRightChild(k3.leftChild);

    return rotateWithLeftChild(k3);
}
```



Double rotate with right child

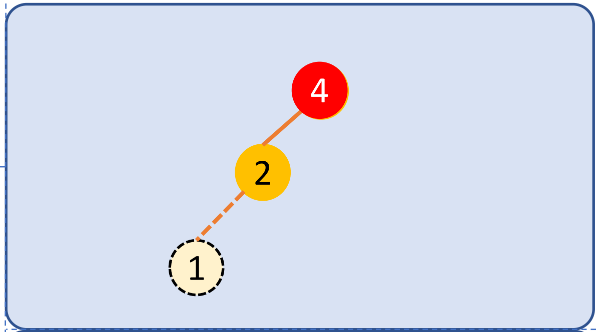


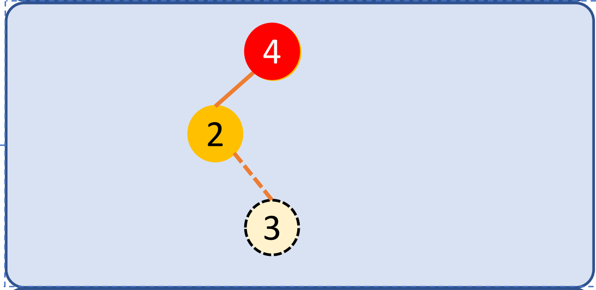

```
// Given the node to be rebalanced (Node k3)
private Node doubleWithRightChildLeftSubtree(Node k3) {

    k3.rightChild = rotateWithLeftChild(k3.rightChild);

    return rotateWithRightChild(k3);
}
```

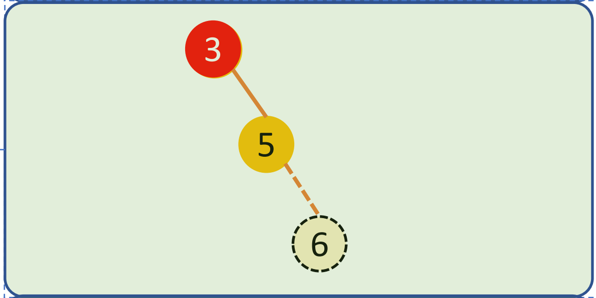


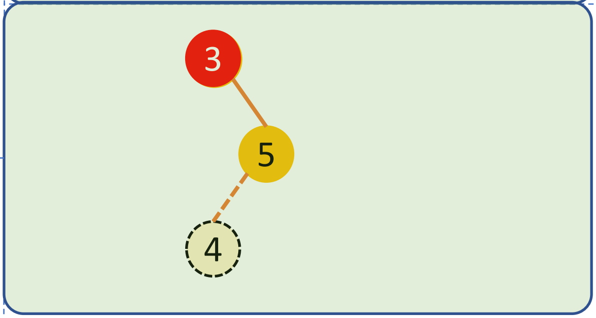
Detect When to Balance

```
// The node to be balanced
private Node balance(Node node) {

    if (node == null)
        return node;

    if (height(node.leftChild) - height(node.rightChild) > BALANCE_FACTOR) {
        if (height(node.leftChild.leftChild) >= height(node.leftChild.rightChild)) {
            node = rotateWithLeftChild(node);
        } else {
            node = doubleWithLeftChildRightSubtree(node);
        }
    } else if (height(node.rightChild) - height(node.leftChild) > BALANCE_FACTOR) {
        if (height(node.rightChild.rightChild) >= height(node.rightChild.leftChild)) {
            node = rotateWithRightChild(node);
        } else {
            node = doubleWithRightChildLeftSubtree(node);
        }
    }

    node.height = Math.max(height(node.leftChild), height(node.rightChild)) + 1;
    return node;
}
```



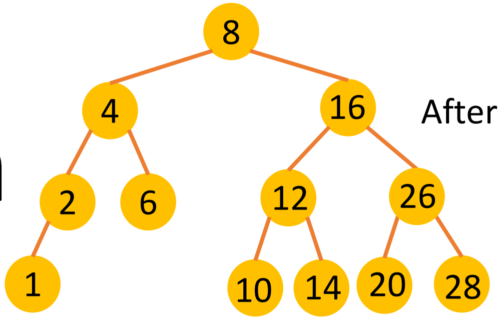


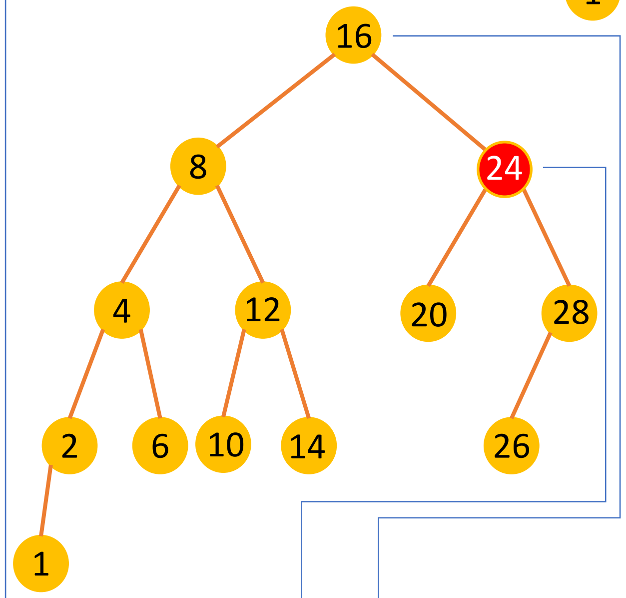




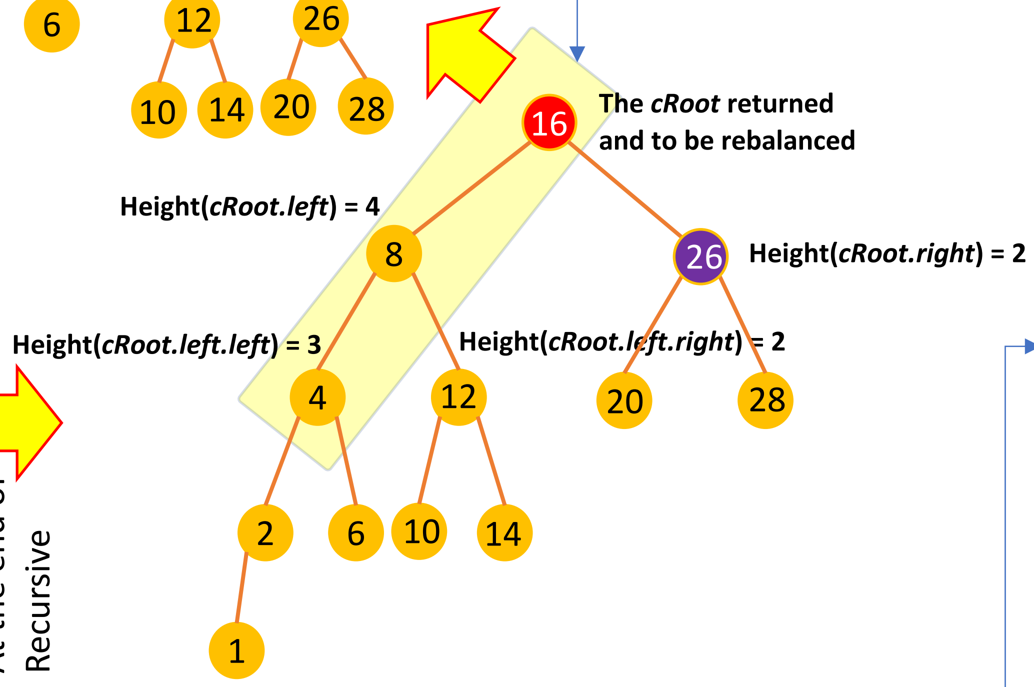
Balance Factor of a node is the height of its right subtree minus the height of its left subtree. In an AVL tree, a balance factor should not be greater than 1, i.e., $|BALANCE\_FACTOR| \leq 1$

Deletion





At the end of Recursive




```
Node BALANCE(cRoot) {
  if cRoot = NIL  return cRoot;
  if HEIGHT(cRoot.left) – HEIGHT(cRoot.right) > 1
    if HEIGHT(cRoot.left.left) >= HEIGHT(cRoot.left.right)
      cRoot = SINGLE_ROTATE_LCHILD(cRoot)
    else
      cRoot = DOUBLE_ROTATE_LCHILD_RSUBT(cRoot)
  else if HEIGHT(cRoot.right) – HEIGHT(cRoot.left) > 1
    if HEIGHT(cRoot.right.right) >= HEIGHT(cRoot.right.left)
      cRoot = SINGLE_ROTATE_RCHILD(cRoot)
    else
      cRoot = DOUBLE_ROTATE_RCHILD_LSUBT(cRoot)
  else
    UPDATE_HEIGHT(cRoot)
  return cRoot;
```

26

```
Node REMOVE(key, cRoot) { (INPUTS: key=26, cRoot=26)
  if cRoot = NIL  return cRoot;
  if key < cRoot.key
    cRoot.left = REMOVE(key, cRoot.left);
  else if key > cRoot.key
    cRoot.right = REMOVE(key, cRoot.right);
  else if cRoot.left ≠ NIL and cRoot.right ≠ NIL
    cRoot.key = FIND_MIN(cRoot.right).key;
    cRoot.right = REMOVE(cRoot.key, cRoot.right);
  else
    cRoot = (cRoot.leftChild ≠ NIL) ? cRoot.left : cRoot.right;
  return BALANCE(cRoot);  (cRoot=NIL)
```

16

```
(cRoot.right=26)
Node REMOVE(key, cRoot) { (INPUTS: key=24, cRoot=16)
  if cRoot = NIL  return cRoot;
  if key < cRoot.key
    cRoot.left = REMOVE(key, cRoot.left);
  else if key > cRoot.key                 (cRoot.right = 24)
    cRoot.right = REMOVE(key, cRoot.right);
  else if cRoot.left ≠ NIL and cRoot.right ≠ NIL
    cRoot.key = FIND_MIN(cRoot.right).key;
    cRoot.right = REMOVE(cRoot.key, cRoot.right);
  else
    cRoot = (cRoot.leftChild ≠ NIL) ? cRoot.left : cRoot.right;
  return BALANCE(cRoot); (cRoot=16)
```

24

```
Node REMOVE(key, cRoot) { (INPUTS: key=24, cRoot=24)
  if cRoot = NIL  return cRoot;
  if key < cRoot.key
    cRoot.left = REMOVE(key, cRoot.left);
  else if key > cRoot.key
    cRoot.right = REMOVE(key, cRoot.right);
  else if cRoot.left ≠ NIL and cRoot.right ≠ NIL
    cRoot.key = FIND_MIN(cRoot.right).key;  (MIN(key) = 26)
    cRoot.right = REMOVE(cRoot.key, cRoot.right);
  else                                      (cRoot.right = 28)
    cRoot = (cRoot.leftChild ≠ NIL) ? cRoot.left : cRoot.right;
  return BALANCE(cRoot); (cRoot=26 <key changed from 24 to 26>, cRoot.right=28)
```

28

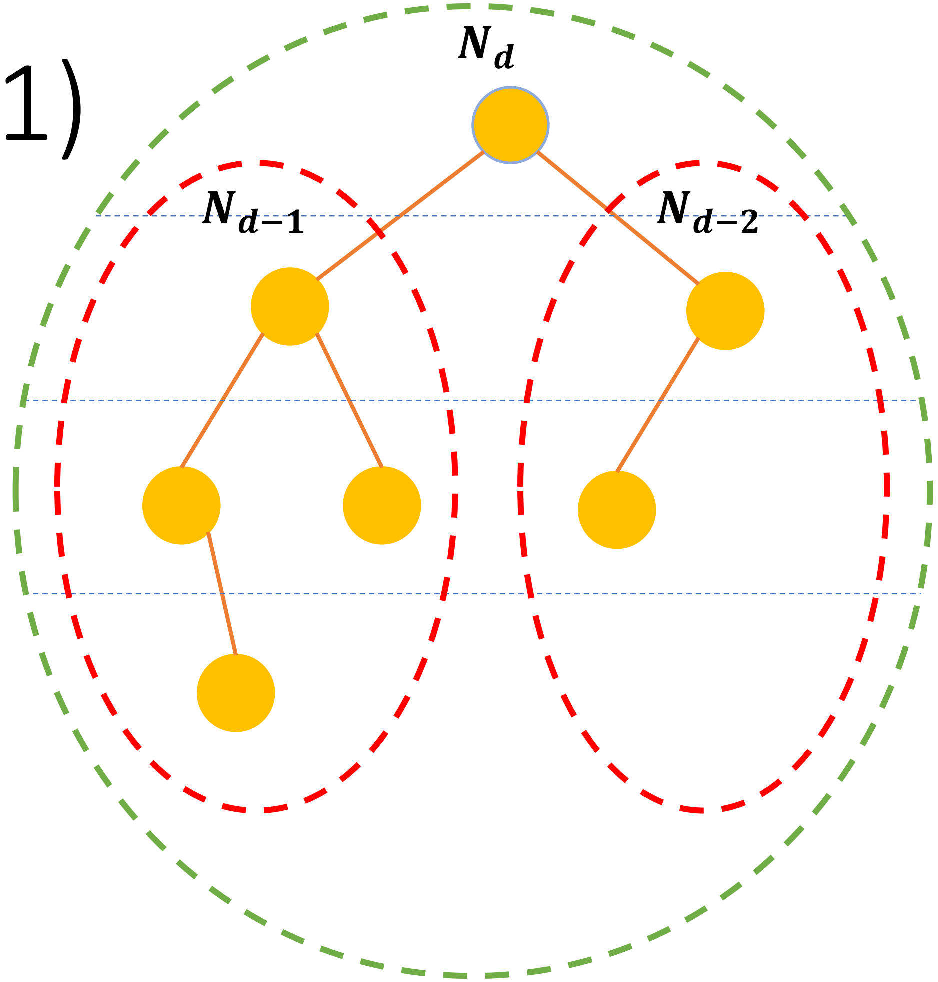
```
Node REMOVE(key, cRoot) {(INPUTS: key=26, cRoot=28)
  if cRoot = NIL  return cRoot;
  if key < cRoot.key                      (cRoot.left = 26)
    cRoot.left = REMOVE(key, cRoot.left);
  else if key > cRoot.key                 (cRoot.left=NIL,
    cRoot.right = REMOVE(key, cRoot.right);  i.e., 28.left=NIL)
  else if cRoot.left ≠ NIL and cRoot.right ≠ NIL
    cRoot.key = FIND_MIN(cRoot.right).key;
    cRoot.right = REMOVE(cRoot.key, cRoot.right);
  else
    cRoot = (cRoot.leftChild ≠ NIL) ? cRoot.left : cRoot.right;
  return BALANCE(cRoot); (cRoot=28)
```

Analysis – Maximum Depth(1)

1

Given an AVL tree $\boldsymbol{T}$, let's denote by $\boldsymbol{N_d}$ the minimum number of nodes of $\boldsymbol{T}$ with the given maximum depth $\boldsymbol{d}$, we show the following recurrence relation:

$$N_d = N_{d-1} + N_{d-2} + 1$$



2 Add $\mathbf{1}$ to both sides,

$$\underbrace{(N_d+1)}_{F_d} = \underbrace{(N_{d-1}+1)}_{F_{d-1}} + \underbrace{(N_{d-2}+1)}_{F_{d-2}}$$

An AVL tree with minimum number of nodes is a **Fibonacci Tree**.

The numbers $\boldsymbol{F_n}$ are *Fibonacci numbers* with the initial conditions:
$\boldsymbol{N_0 = 1}$ (the number of nodes at level 0 $\Rightarrow \boldsymbol{F_0 = 2}$),
$\boldsymbol{N_1 = 2}$ (the number of nodes at level 1, but with the minimum number of nodes required to form the AVL tree $\Rightarrow \boldsymbol{F_1 = 3}$).

Analysis – Maximum Depth(2)

$$F_d = F_{d-1} + F_{d-2}$$

3 Find a function of n which satisfies

$$F_d - F_{d-1} - F_{d-2} = 0$$

3.1 Given the recurrence relation above, the *characteristic polynomial* is

$$ax^2 + bx + c$$

$$a = 1$$
$$b = -1$$
$$c = -1$$

3.2 which gives the *characteristic equation*,

$$ax^2 + bx + c = 0$$

The root(s) of the characteristic equation are known as *characteristic roots*.

3.3 If the characteristic roots r_1 and r_2 are the solutions to the characteristic equation, then the solution to the recurrence relation is given by,

$$F_d = c_1 r_1^d + c_2 r_2^d$$

4 Referring to 3.2,

$$r_1 = \frac{1+\sqrt{5}}{2}; \quad r_2 = \frac{1-\sqrt{5}}{2}$$

thus,

$$F_d = c_1\left(\frac{1+\sqrt{5}}{2}\right)^d + c_2\left(\frac{1-\sqrt{5}}{2}\right)^d$$

Analysis – Maximum Depth(3)

$$F_d = c_1\left(\frac{1+\sqrt{5}}{2}\right)^d + c_2\left(\frac{1-\sqrt{5}}{2}\right)^d$$

5 Given $F_0 = 2$ and $F_1 = 3$, determine c_1 and c_2,

$$\begin{cases} 2 = c_1\left(\frac{1+\sqrt{5}}{2}\right)^0 + c_2\left(\frac{1-\sqrt{5}}{2}\right)^0 \\ 3 = c_1\left(\frac{1+\sqrt{5}}{2}\right)^1 + c_2\left(\frac{1-\sqrt{5}}{2}\right)^1 \end{cases}$$

5.1 Solve the system of equations:

$$\begin{cases} c_1 = 1 + \frac{2}{\sqrt{5}} \\ c_2 = 1 - \frac{2}{\sqrt{5}} \end{cases}$$

6 Going back to equation 2,

$$F_d = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^d + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^d$$

$$N_d = F_d - 1$$

hence,

$$= \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^d + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^d - 1$$

Analysis – Maximum Depth(4)

$$\boldsymbol{N_d = \left(1+\frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^d + \left(1-\frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^d - 1}$$

7 Approximating,

$$\frac{N_d+1}{1.89} = \left(\frac{1+\sqrt{5}}{2}\right)^d$$

$$\Rightarrow \log_{\frac{1+\sqrt{5}}{2}}\left(\frac{N_d+1}{1.89}\right) = d$$

$$\Rightarrow d < \log_{\frac{1+\sqrt{5}}{2}}(N_d+1) = \frac{\log_2(N_d+1)}{\log_2\frac{1+\sqrt{5}}{2}}$$

$$\Rightarrow d < 1.44\log_2(N+1)$$

thus, $\boldsymbol{d = O(\log_2 N)}$

The *asymptotic upper bound* of the depth of an AVL tree having $\boldsymbol{N}$ nodes is $\boldsymbol{O(log_2 N)}$.

具有N节点的AVL树的深度的渐近上界是$Olog2N$

A

When $\boldsymbol{d} \to \infty$, $\boldsymbol{\left(1-\frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^d \to 0}$

B

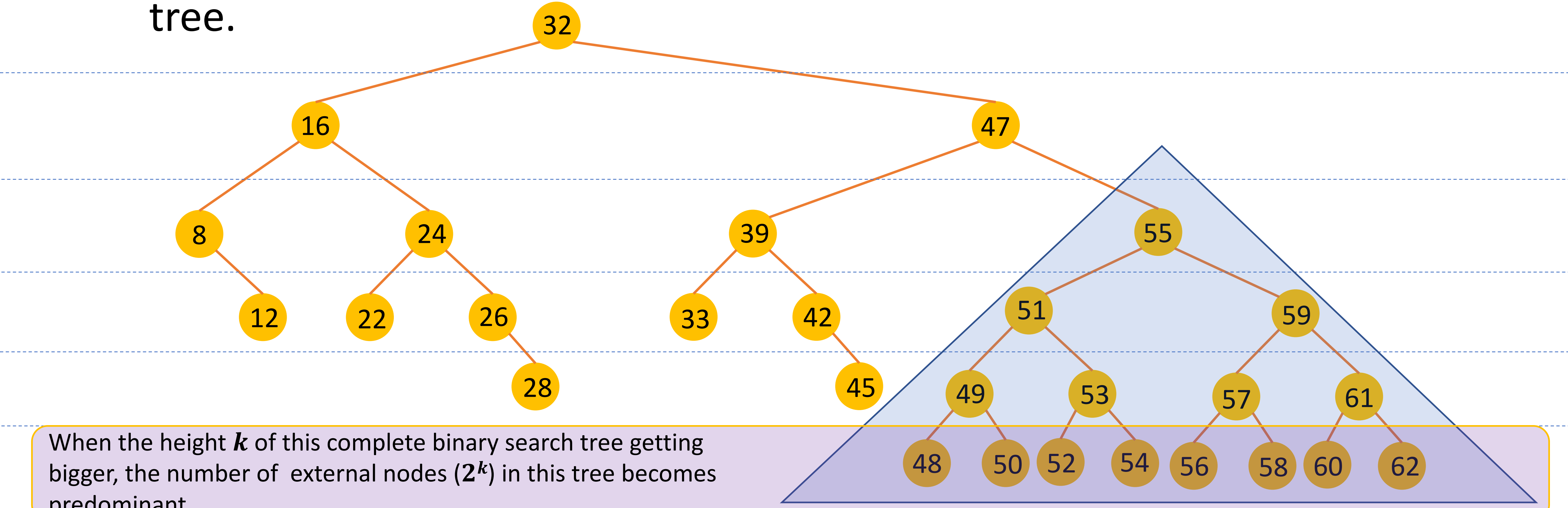
$$\boldsymbol{\left(1+\frac{2}{\sqrt{5}}\right) \approx 1.89}$$

C

$$\frac{1}{\log_2\frac{1+\sqrt{5}}{2}} \approx 1.44$$

Potential Issues with AVL Trees

- It is possible to construct a side-*overweighted* AVL, which has all paths asymptotically reaching to the maximal possible height of the tree.



When the height $\boldsymbol{k}$ of this complete binary search tree getting bigger, the number of external nodes ($\mathbf{2}^{\boldsymbol{k}}$) in this tree becomes predominant.

Summary

- In an AVL tree, for every node in the tree, the height of the left and right subtree can differ by at most one, i.e., height-balanced

- A node to be deleted can be *marked* as deleted, thus no rebalancing is needed

- Search, insertion, and deletion take $\boldsymbol{O}(\log_{\mathbf{2}} \boldsymbol{N})$ time in both the average and worst cases

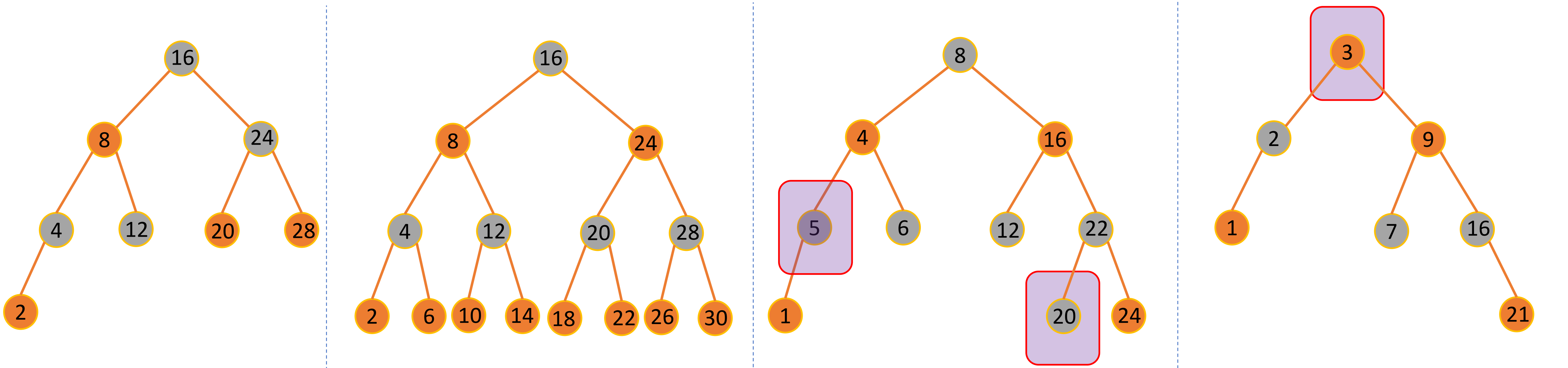
- In some situations, AVL trees can be side-overweighted, which results in a relatively poor performance

Red-Black Trees

- Red-black tree is an alternative to the AVL tree, thus a red-black tree is also a self-balancing binary search tree
- Operations on red-back tree take $O(lgN)$ time in the worst case, and the height of a red-black tree is at most $2\lg(N+1)$
- Each node in red-back tree needs an extra bit to store the colour of the node, which can be either Red or Black
- Red-black trees ensure that no path from the root to a leaf is more than twice as long as any other, thus the tree is *approximately balanced*

Properties of Red-black Trees

1. Every node is either **red** or **black**.
2. The root is **black**.
3. If a node is **red**, both its children will be coloured **black**.
4. For each node, all simple path from the node to descendent leaves contain the same number of **black** nodes.



Red-Black Trees: Guibas, L. J., & Sedgewick, R. (1978, October). A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)* (pp. 8-21). IEEE.

Insertion (1)

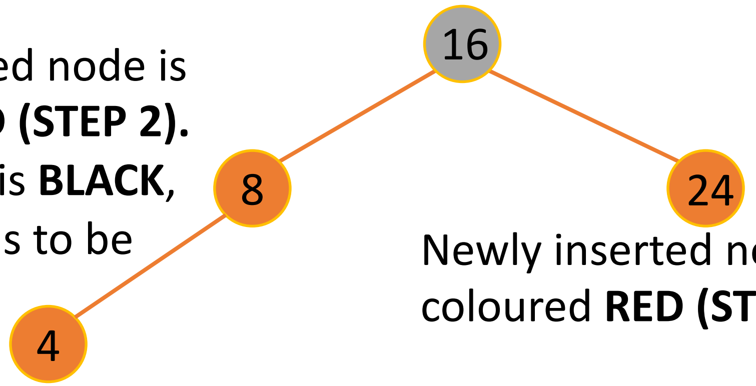
- **Rules:**
  1. red or black.
  2. black root, black *null* node.
  3. red node → black children.
  4. the same number of black nodes.

Steps:

1. Insert a node following the insertion rules used in the binary search tree
2. Colour the newly inserted node **red**
3. If rule(s) is violated, fix the colour from top to bottom

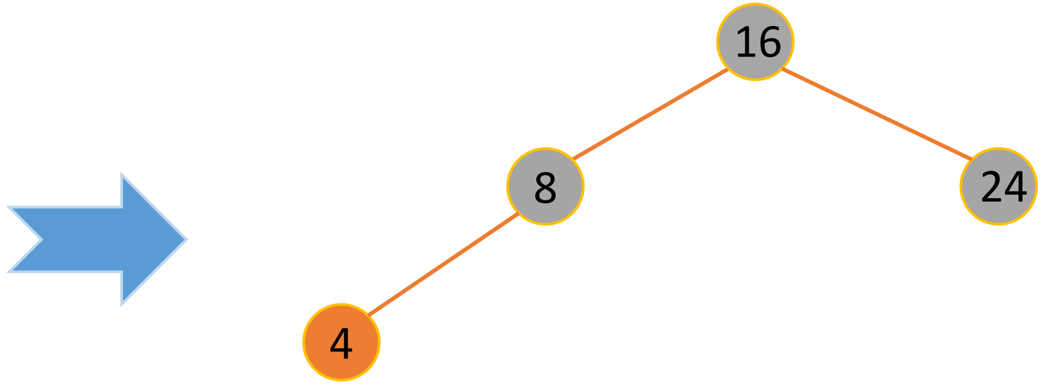
It can only be coloured **BLACK (RULE 2)**

Newly inserted node is coloured **RED (STEP 2).** If the parent is **BLACK**, nothing needs to be changed.



Newly inserted node is coloured **RED (STEP 2)**

Newly inserted node is coloured **RED**, but it violates **RULE 3**



When the parent is RED:

CASE 1: During the insertion of a node (in this example, Node 4), moving down to the bottom (following the BST insertion rule), if the left child (in this example, Node 8) and right child (in this example, Node 24) of a node (in this example, Node 16) are both **RED**,

1. change the colour of the node (Node 16) to **RED**, but if it is the root node, change the colour back to **BLACK** immediately.
2. change its left child (Node 8) and right child (Node 24) both to **BLACK**

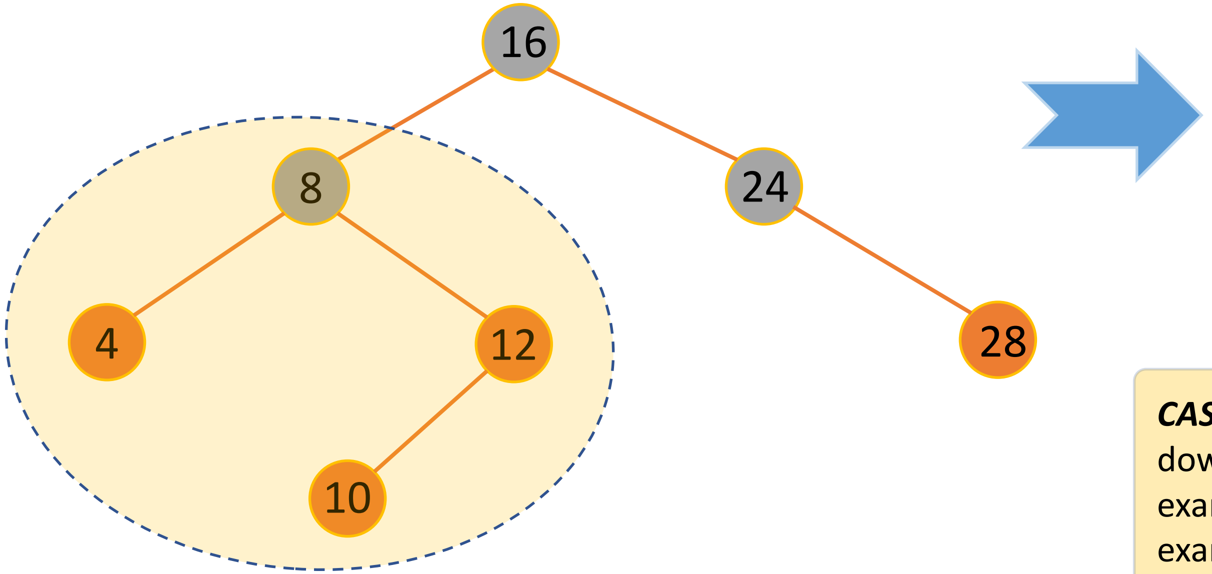
Insert { **16, 8, 24, 4**, 12, 28, 10, 14, 9, 30 }

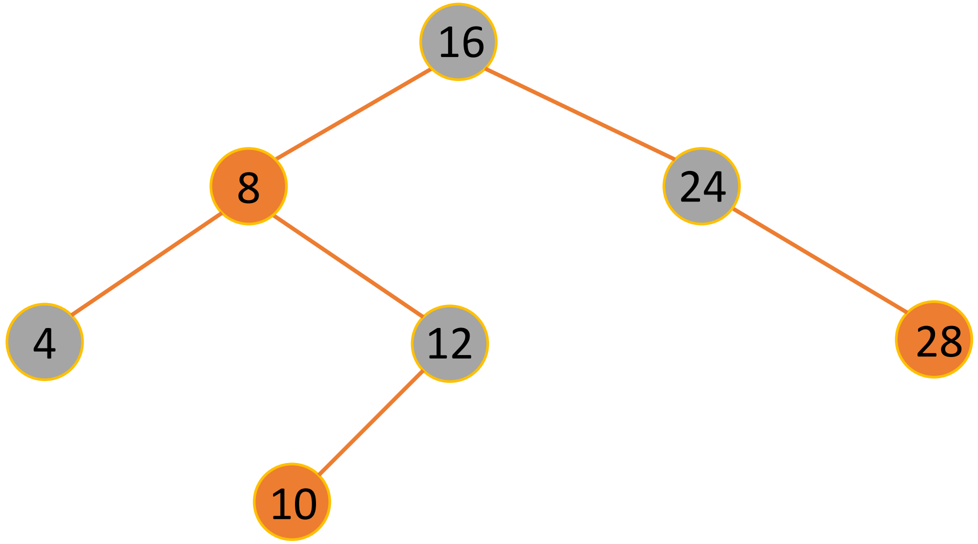
Insertion (2)

- **Rules:**
  1. red or black.
  2. black root, black *null* node.
  3. red node → black children.
  4. the same number of black nodes.

Steps:

1. Insert a node following the insertion rules used in the binary search tree
2. Colour the newly inserted node **red**
3. If rule(s) is violated, fix the colour from bottom to top





Newly inserted node is coloured **RED**, but it violates **RULE 3**

Insert { 16, 8, 24, 4, **12, 28, 10**, 14, 9, 30 }

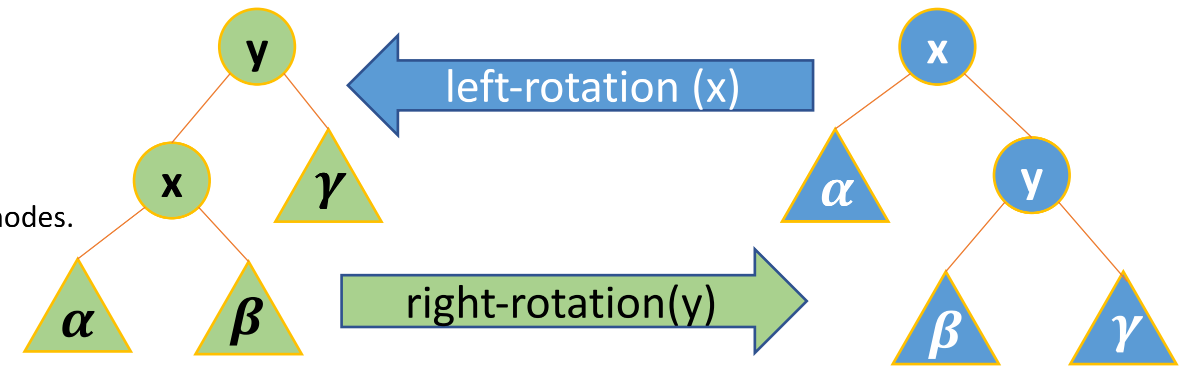
CASE 1: During the insertion of a node (in this example, Node 10), moving down to the bottom (following the BST insertion rule), if the left child (in this example, Node 4) and right child (in this example, Node 12) of a node (in this example, Node 8) are both **RED**,

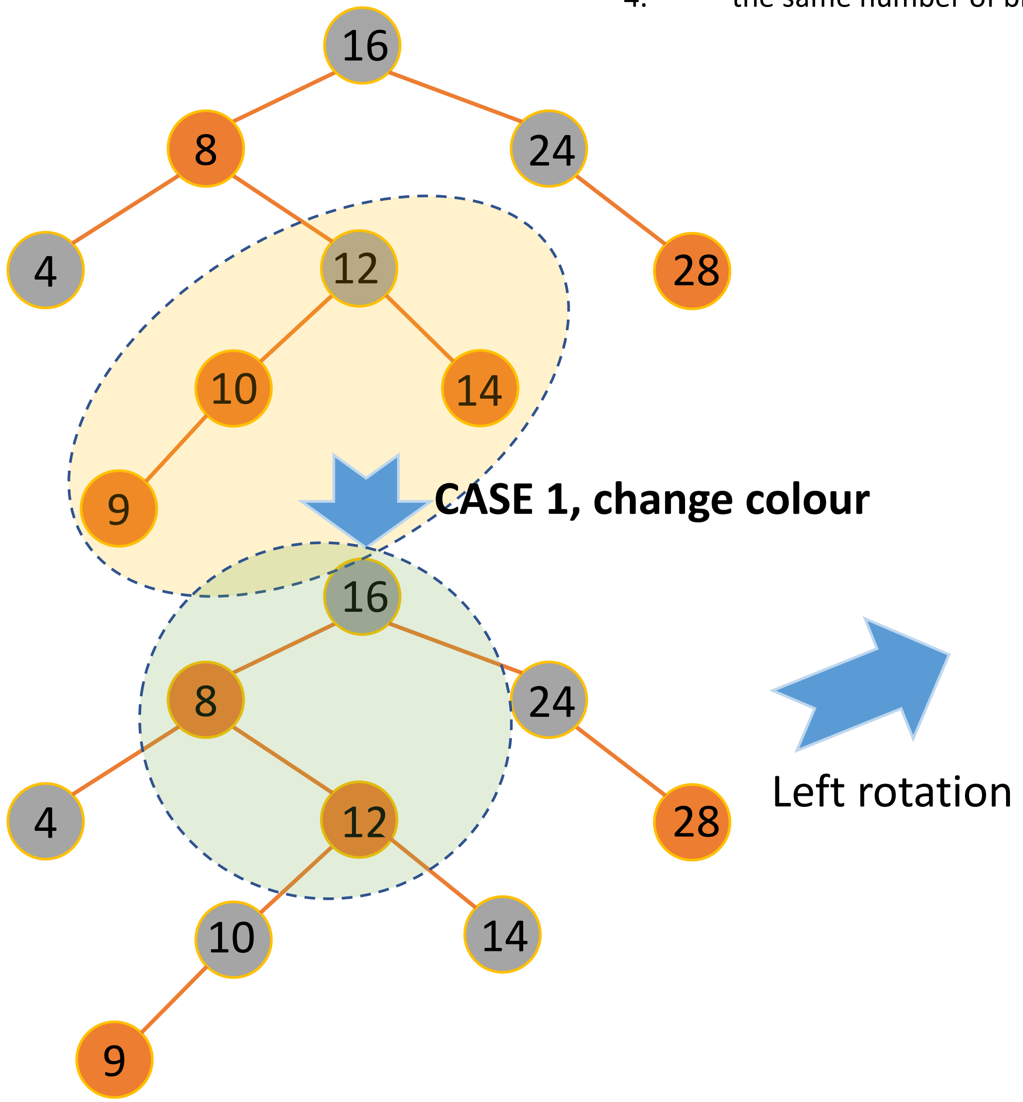
1. change the colour of the node (Node 8) to **RED**, but if it is the root node, change the colour back to **BLACK** immediately.
2. change its left child (Node 4) and right child (Node 12) both to **BLACK**

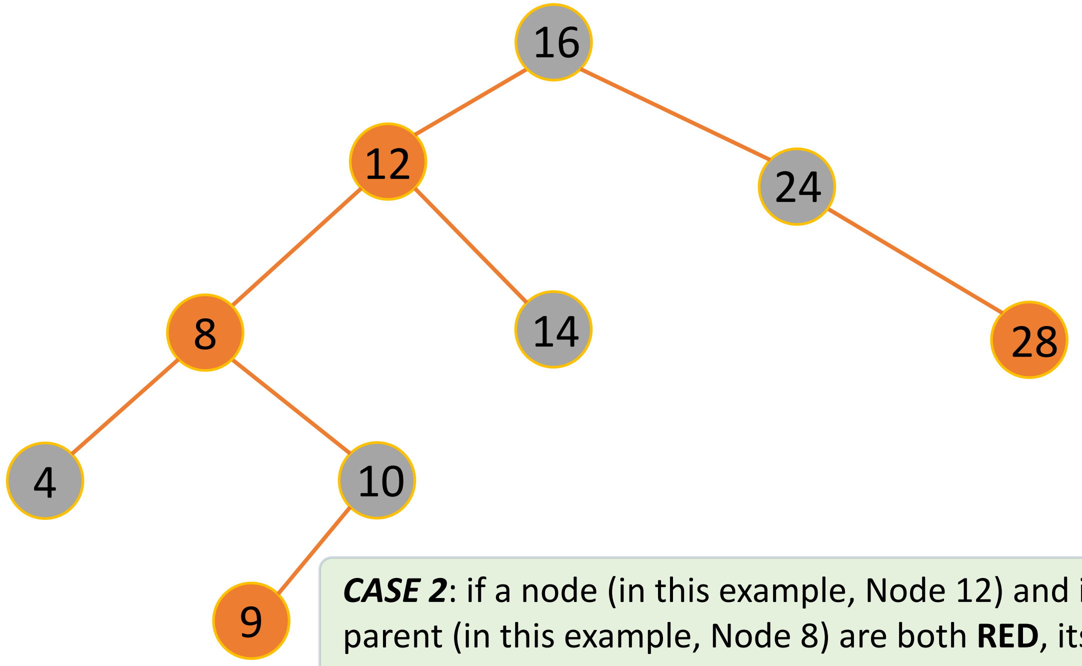
Insertion (3)

Rules:

1. red or black.
2. black root, black *null* node.
3. red node → black children.
4. the same number of black nodes.







CASE 2: if a node (in this example, Node 12) and its parent (in this example, Node 8) are both **RED**, its sibling is **BLACK**, and its parent, grandparent and itself form a zig-zag shape:

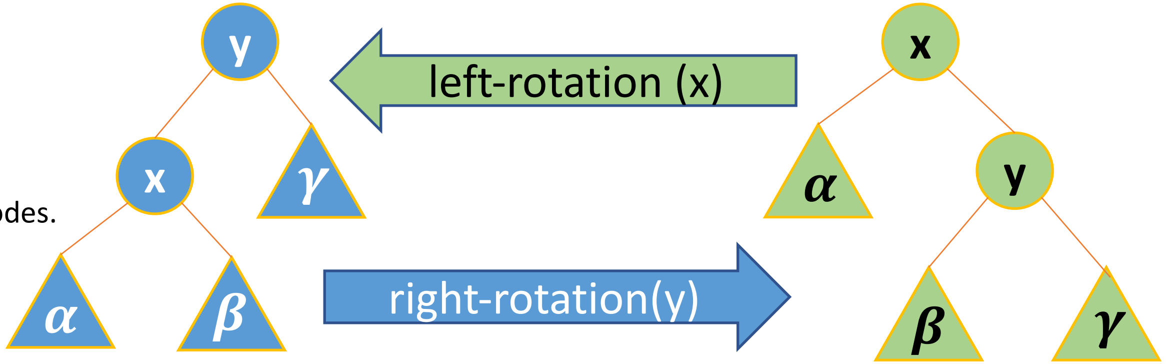
1. perform a double rotation:
   - left-rotation
   - Right-rotation
2. change the new root to BLACK and its children to **RED**

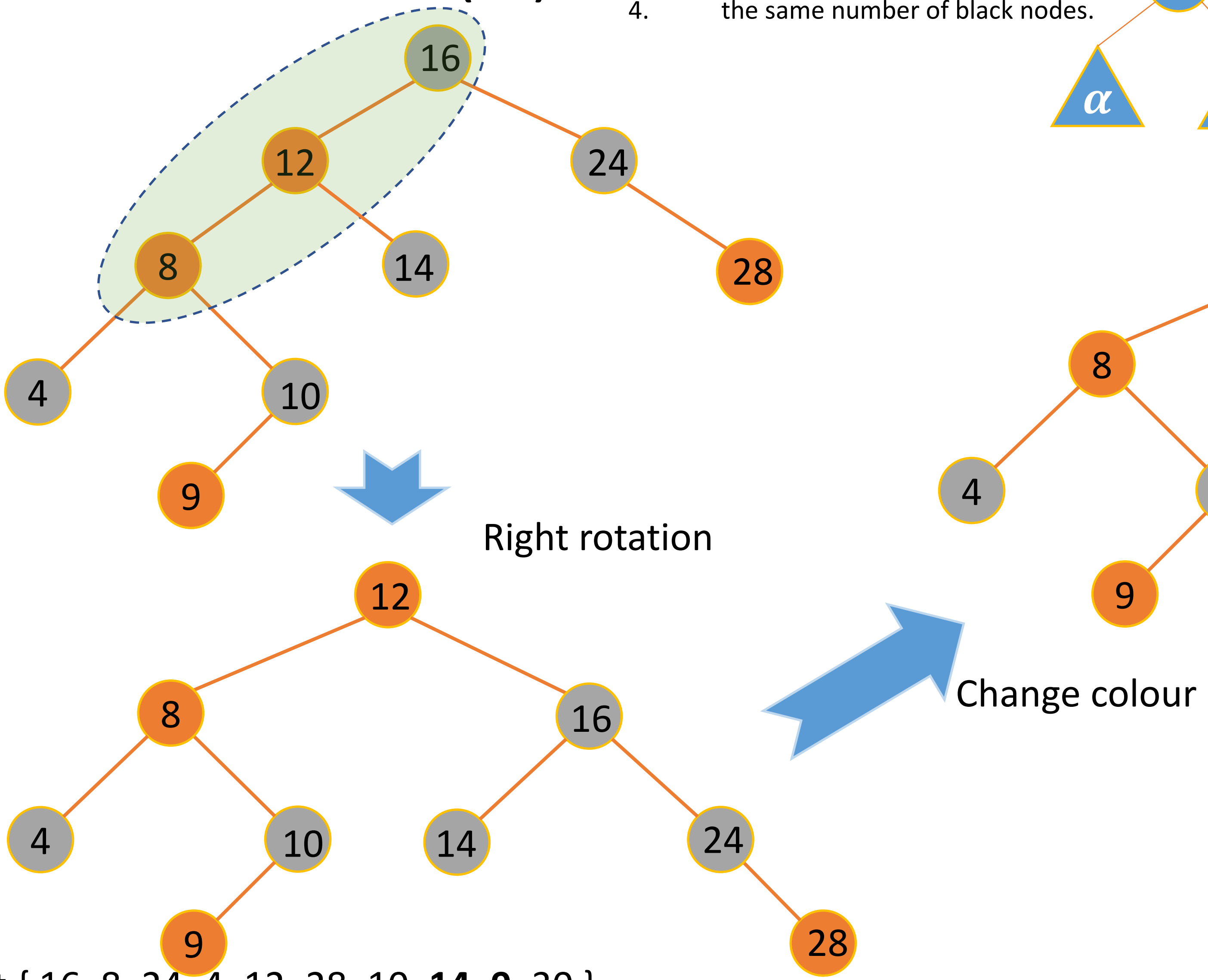
Insert { 16, 8, 24, 4, 12, 28, 10, **14, 9**, 30 }

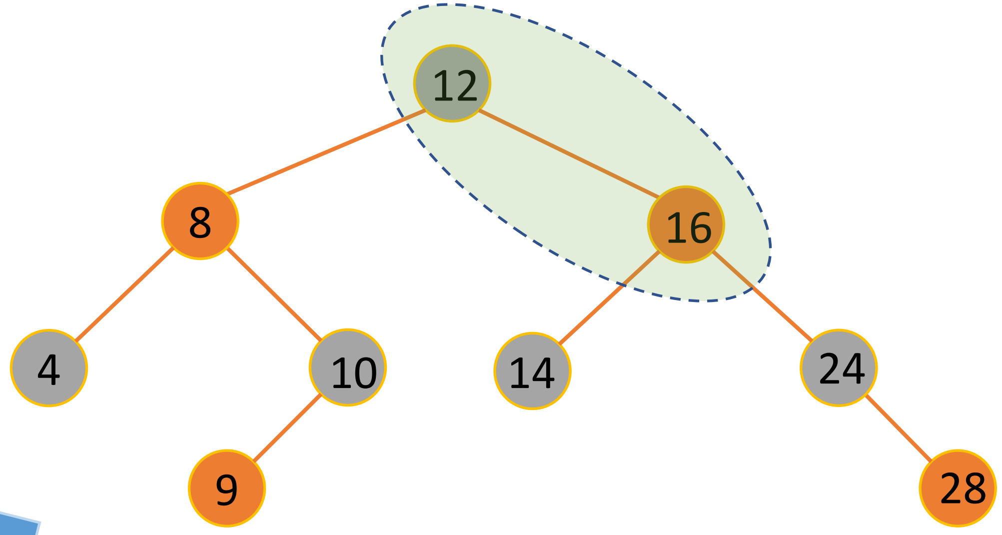
Insertion (4)

Rules:

1. red or black.
2. black root, black *null* node.
3. red node → black children.
4. the same number of black nodes.







CASE 2: if a node (in this example, Node 12) and its parent (in this example, Node 8) are both **RED**, its sibling is **BLACK**, and its parent, grandparent and itself form a zig-zag shape:

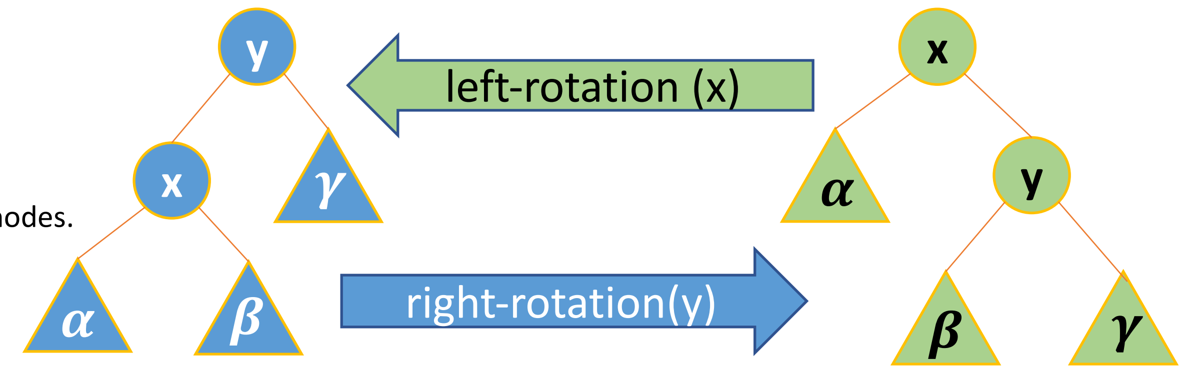
1. perform a double rotation:
   - left-rotation
   - Right-rotation
2. change the new root to BLACK and its children to **RED**

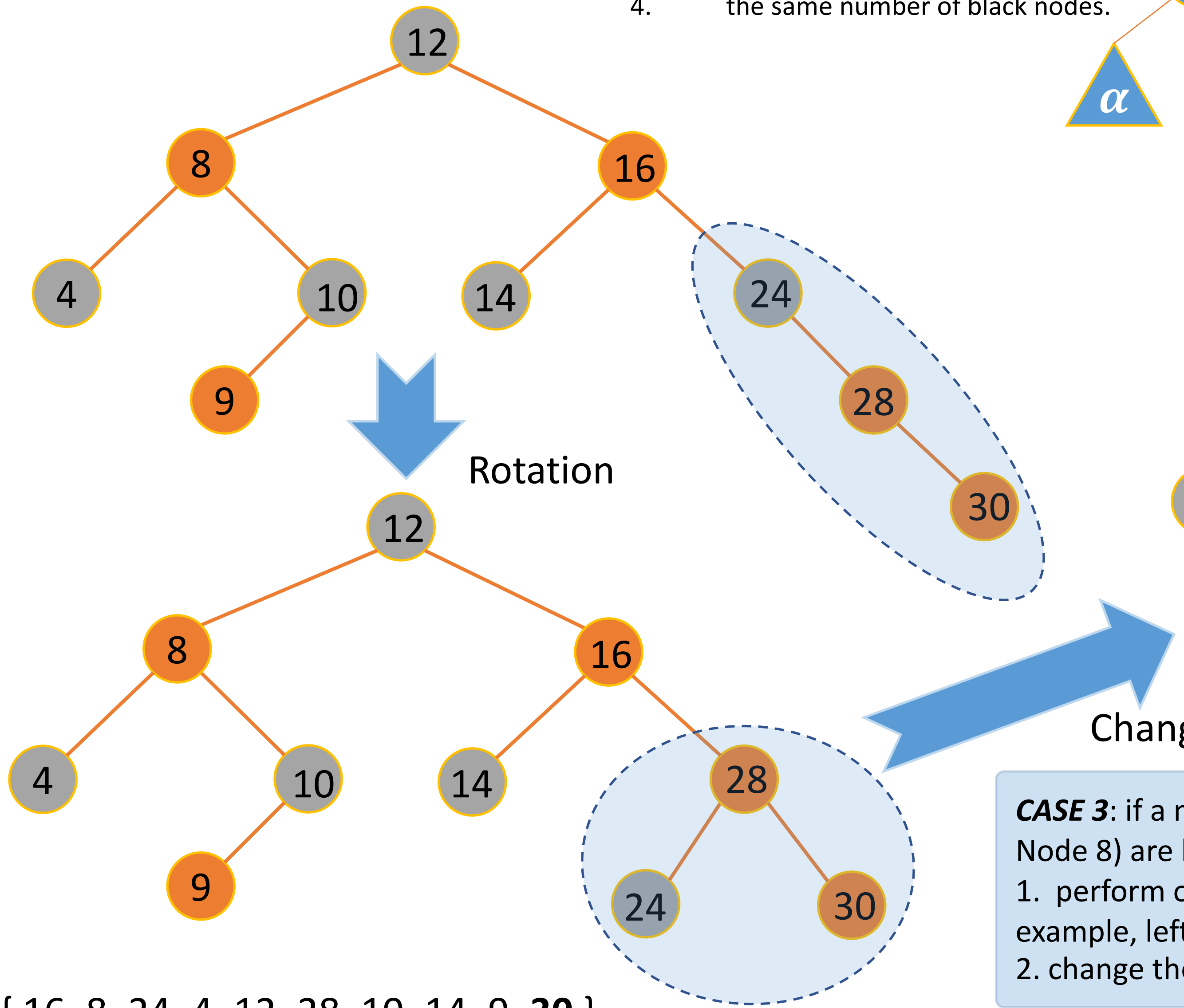
Insert { 16, 8, 24, 4, 12, 28, 10, **14, 9**, 30 }

Insertion (5)

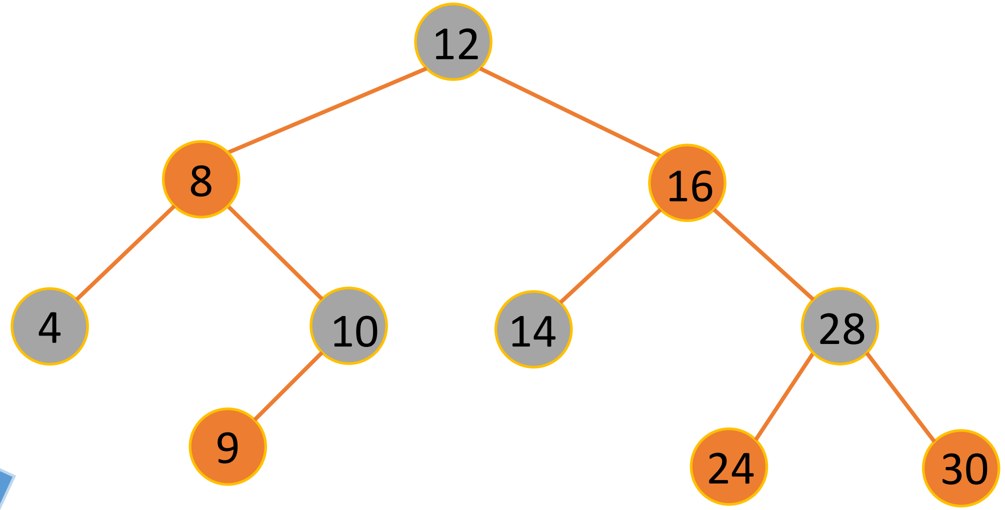
Rules:

1. red or black.
2. black root, black *null* node.
3. red node → black children.
4. the same number of black nodes.





Change colour

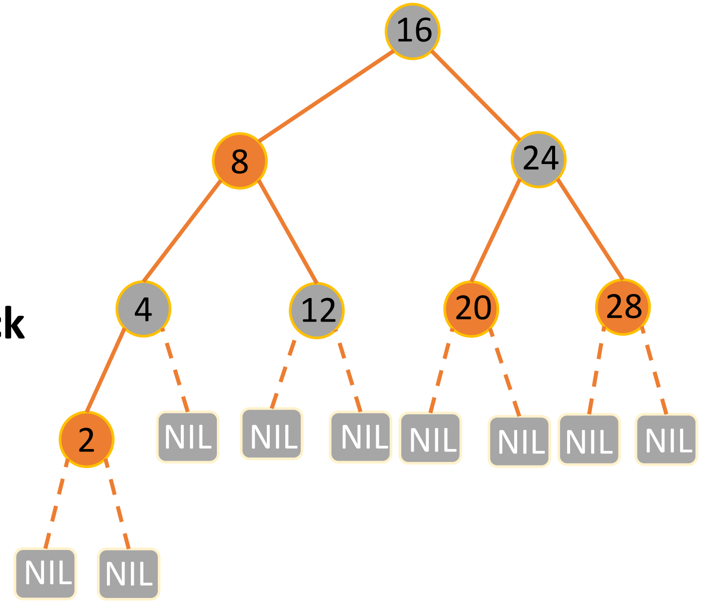


CASE 3: if a node (in this example, Node 12) and its parent (in this example, Node 8) are both **RED**, its sibling is **BLACK:**

1. perform one rotation (left or right depending on the situation, in this example, left-rotation)
2. change the new root to **BLACK** and its children to **RED**

Insert { 16, 8, 24, 4, 12, 28, 10, 14, 9, **30** }

Analysis – Maximum Depth(1)



1 Assuming all $null$ reference nodes are external nodes and coloured **black**

Let $\boldsymbol{bh(x)}$ denotes the number of **black** nodes on any simple path from a node $\boldsymbol{x}$ (but not including x) to its descendent leaf node.

2 According to the ***Property 4***, i.e., for each node, all simple path from the node to descendent leaves contain the same number of **black** nodes.

Thus, the number of internal nodes of the subtree rooted at $\boldsymbol{x}$ is at least $\boldsymbol{2^{bh(x)} - 1}$

A Internal nodes for k-ary complete tree

3 According to the ***Property 3***, i.e., if a node is **red**, both its children will be coloured **black**.

$$\sum_{i=0}^{h-1} k^i = \frac{1 - k^h}{1 - k}$$

Thus, $\boldsymbol{bh(x)}$ is at least half of the height ($\boldsymbol{h}$) of $\boldsymbol{x}$, $\Rightarrow \boldsymbol{N \geq 2^{\frac{h}{2}} - 1}$

Analysis – Maximum Depth(2)

$$N \geq 2^{\frac{h}{2}} - 1$$

4 Rearranging and taking logarithms on both sides,

$$h \leq 2\lg(N+1)$$

5 Since the asymptotic efficiency of binary search tree operations can run in $O(h)$ time,

BST Operations
SEARCH(K key)
MINIMUM( )
MAXIMUM( )
SUCCESSOR(K key)
PREDECESSOR(K key)
INSERT(K key, V value)
DELETE(K key)

Thus, operations on a red-black tree on N nodes take $O(\lg N)$ time in the worst case.

Summary

- A red-black tree is self-balancing binary search tree 自平衡二叉搜索树
- One extra bit is needed for each node to store the colour of the node
- A node can only be coloured either red or black
- Red-black trees have relatively low overhead for insertion
- In practice, rotations occur relatively infrequently compared to AVL trees
- Using red-black trees can avoid side-overweighted subtrees

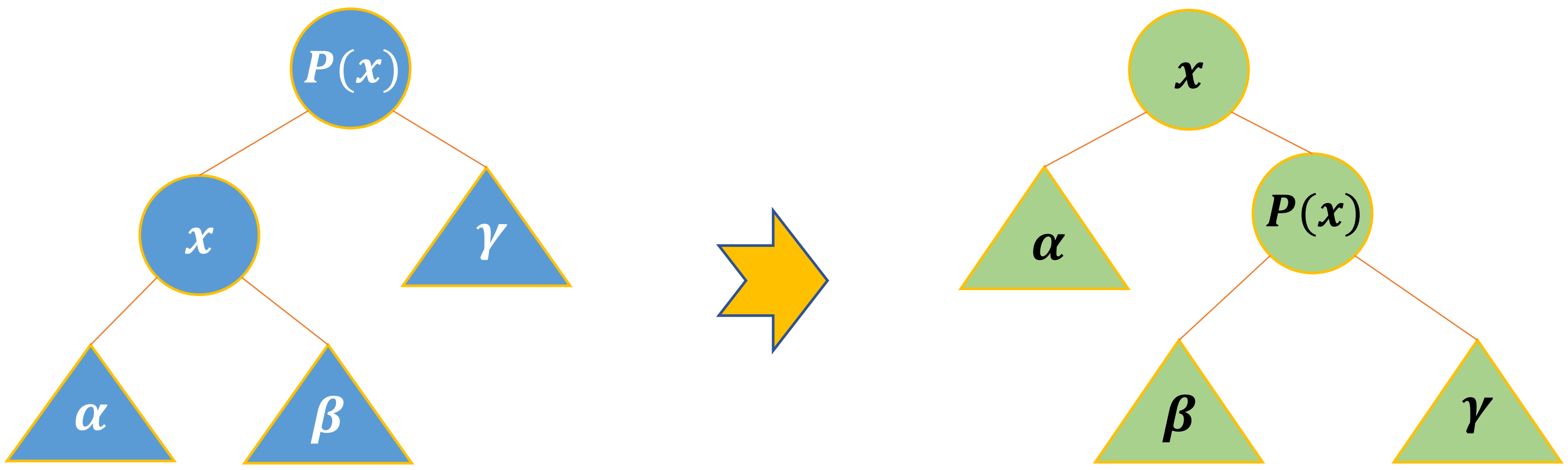
Splay Trees 伸展树

- A splay tree guarantees that any $\boldsymbol{M}$ consecutive tree operations starting from an empty tree take at most $\boldsymbol{O}(\boldsymbol{M}\mathbf{log}\boldsymbol{N})$ time
- Splay trees are based on the fact that the $\boldsymbol{O}(\boldsymbol{N})$ worst-case time per operation for BST is not bad, as long as it occurs relatively infrequently
- The basic idea of the splay tree is that after a node is accessed, the node is pushed to the root by a series of AVL tree rotations
- In a splay tree, if a node is deep, there are many nodes on the path that are also relatively deep, and by splaying we can make future accesses faster on all these nodes

Splaying Steps (1)

- **Case 1 (zig):** if $P(x)$, the parent of x, is the tree root, rotate the edge joining x with $P(x)$.

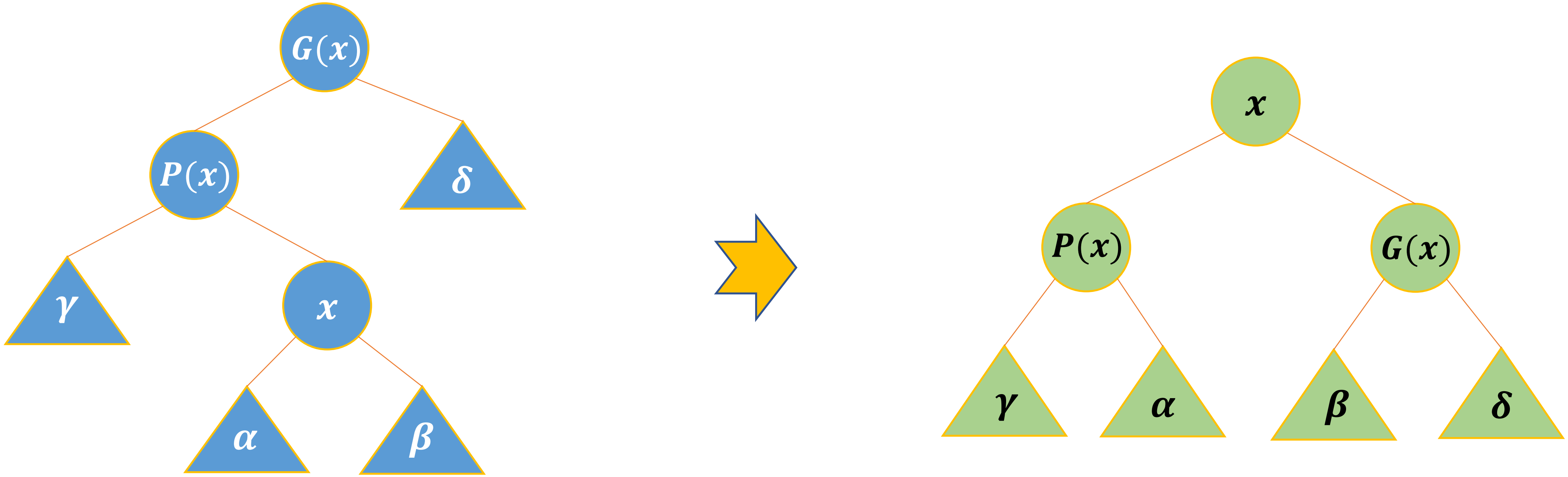
若x的父节点$P(x)$为根节点，则旋转x与P的连线(x)。



Splaying Steps (2)

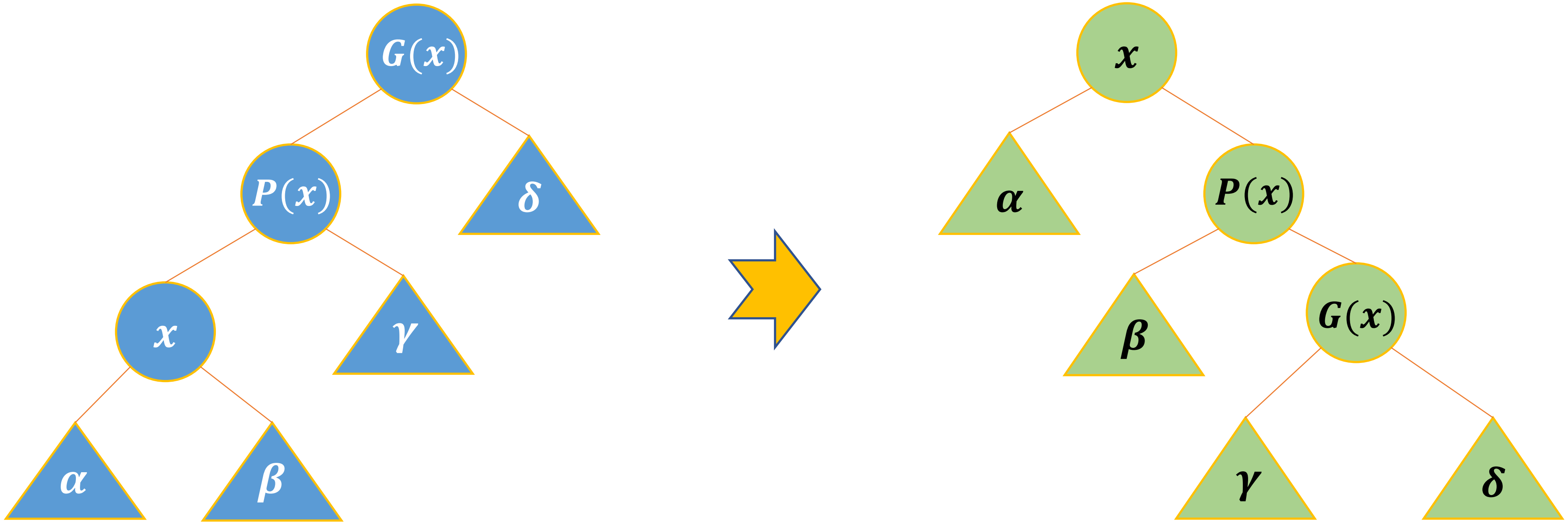
如果$\boldsymbol{P(x)}$不是根,和$\boldsymbol{x}$左孩子和$\boldsymbol{P(x)}$是一个正确的孩子,或者相反,旋转的边缘加入$\boldsymbol{x}$ $\boldsymbol{P(x)}$,然后旋转加入$\boldsymbol{x}$边缘与新$\boldsymbol{P(x)}$

- **Case 2 (zig-zag):** if $\boldsymbol{P(x)}$ is not the root, and $\boldsymbol{x}$ is a left child and $\boldsymbol{P(x)}$ is a right child, or vice-versa, rotate the edge joining $\boldsymbol{x}$ with $\boldsymbol{P(x)}$, then rotate the edge joining $\boldsymbol{x}$ with the new $\boldsymbol{P(x)}$.

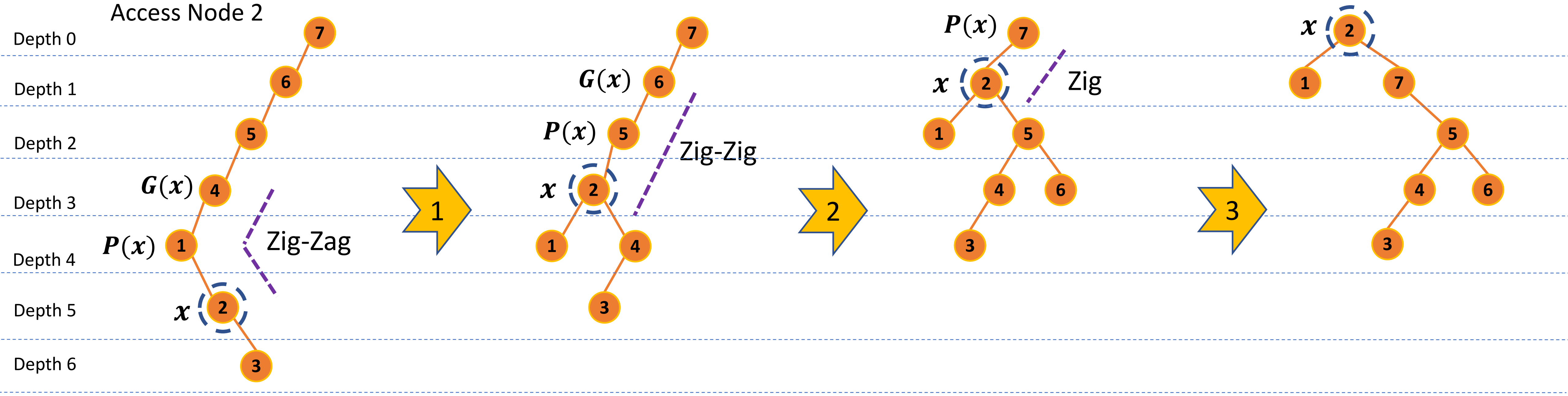


Splaying Steps (3)

- **Case 3 (zig-zig):** if $\boldsymbol{P(x)}$ is not the root, and $\boldsymbol{x}$ and $\boldsymbol{P(x)}$ are both left or right children, rotate the edge joining $\boldsymbol{P(x)}$ with its grandparent $\boldsymbol{G(x)}$, then rotate the edge joining $\boldsymbol{x}$ with $\boldsymbol{P(x)}$.



Properties of Splaying



1. Splaying a node $\boldsymbol{x}$ of depth $\boldsymbol{d}$ takes $\boldsymbol{\theta(d)}$ time, i.e., time proportional to the time to access $\boldsymbol{x}$.

2. Splaying not only moves $\boldsymbol{x}$ to the root, but *roughly* halves the depth of every node along the access path.

Amortized Complexity of Splaying (1)

1 Recall that the time required for any tree operation on node $\boldsymbol{x}$ is proportional to the depth of the node, however, due to the splaying operations, the configuration of the data structure changes over time,

Thus, the idea is to assign to each possible configuration a ***potential***, and the amortized time $\boldsymbol{a}$ of an operation can therefore be defined by

$$\boldsymbol{a} = \boldsymbol{t} + \boldsymbol{\Phi}' - \boldsymbol{\Phi}$$

where t is the actual time of the operation, $\boldsymbol{\Phi}$ is the potential before the operation, and $\boldsymbol{\Phi}'$ is the potential after the operation.

2 The total time of a sequence of $\boldsymbol{m}$ operation can be estimated by

$$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} (a_j + \Phi_{j-1} - \Phi_j) = \sum_{j=1}^{m} a_j + \Phi_0 - \Phi_m$$

Amortized Complexity of Splaying (2)

3 To define the potential of a splay tree, we first define the size $S(x)$ of a node x in the tree to be the number of descendants of node x (including the node x). E.g., $S(2) = 4$

4 Let's define the *rank* $R(x)$ of node x to be $\log_2 S(x)$. E.g., $R(2) = 2$

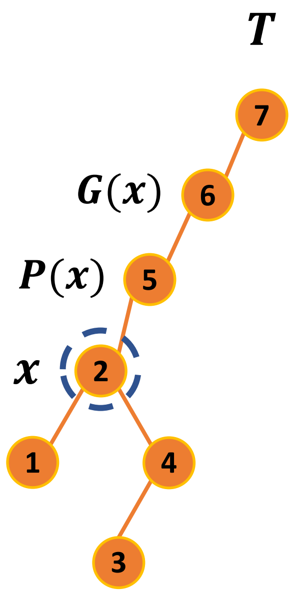
5 The potential function for a tree T is therefore defined as

$$\Phi(T) = \sum_{x \in T} \log_2 S(x) = \sum_{x \in T} R(x)$$

E.g., $\Phi(T) = R(1) + R(2). + R(3). + R(4) + R(5) + R(6) + R(7)$

$$= \log_2 1 + \log_2 4 + \log_2 1 + \log_2 2 + \log_2 5 + \log_2 6 + \log_2 7$$

$$\approx 10.714$$



Amortized Complexity of Splaying (zig)

6 **Access Lemma.** The amortized time to splay a tree with root T at a node x is at most $3(R(T) - R(x)) + 1 = O(\log_2 \frac{S(T)}{S(x)})$

6.1 **Case 1 (zig).** One operation is done, thus the amortized time (AT_{zig}) of the step is

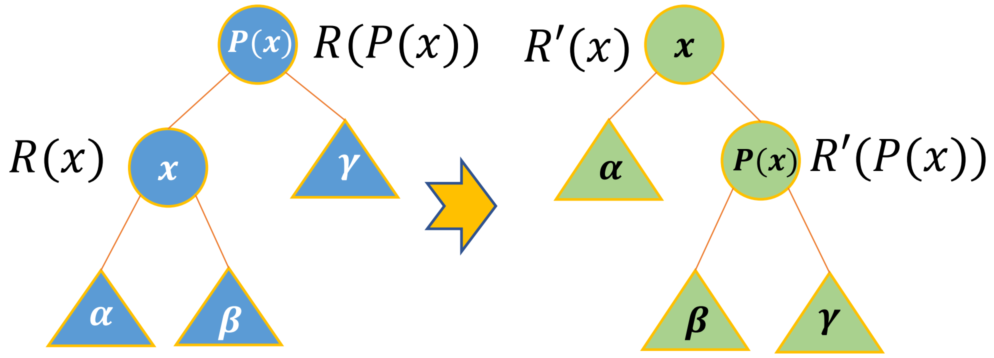
$$AT_{zig} = \underbrace{R'(x) - R(x) + R'(P(x)) - R(P(x))}_{\text{A zig rotation only affects the rank of } x \text{ and its parent } P(x)} + \overbrace{1}^{\text{Only } 1 \text{ operation is needed in a zig rotation}}$$

6.1.1 Given that $R(P(x)) \geq R'(P(x))$, thus

$$AT_{zig} \leq R'(x) - R(x) + 1$$

6.1.2 Given that $R(x) \leq R'(x)$,

$$AT_{zig} \leq 3(R'(x) - R(x)) + 1$$



Amortized Complexity of Splaying (zig-zag 1)

6.2 **Case 2 (zig-zag).** Two operations are done, thus the amortized time ($AT_{zig\text{-}zag}$) of the step is

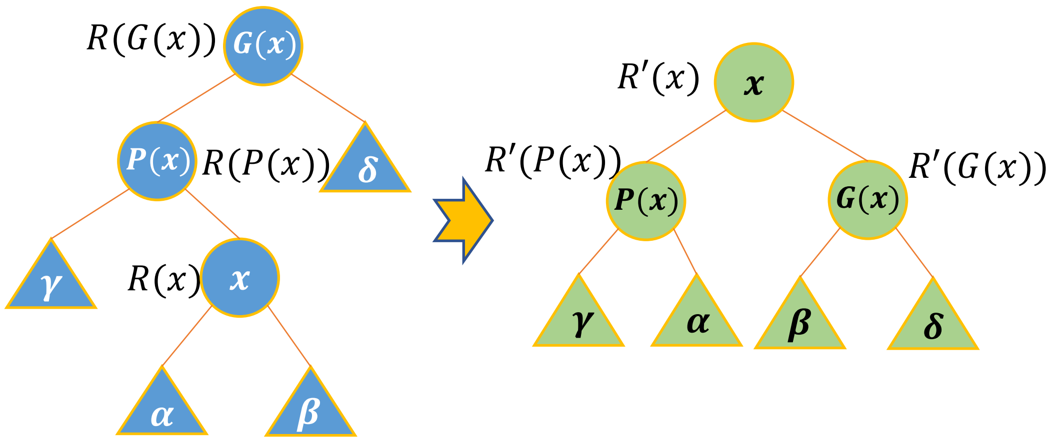
$$AT_{zig\text{-}zag} = \underbrace{R'(x) - R(x) + R'(P(x)) - R(P(x)) + R'(G(x)) - R(G(x))}_{\text{A zig-zag rotation only affects the rank of } x\text{, its parent } P(x) \text{ and its grandparent } G(x)} + \overbrace{2}^{\text{2 operations are needed in a zig-zag rotation}}$$

6.2.1 Given that $R'(x) = R(G(x))$

$$AT_{zig\text{-}zag} = R'(P(x)) + R'(G(x)) - R(x) - R(P(x)) + 2$$

6.2.2 Given that $R(P(x)) \geq R(x)$

$$AT_{zig\text{-}zag} \leq R'(P(x)) + R'(G(x)) - 2R(x) + 2$$



Amortized Complexity of Splaying (zig-zag 2)

$$AT_{zig\text{-}zag} \leq R'(P(x)) + R'(G(x)) - 2R(x) + 2$$

6.2.3 Given that $S'(P(x)) + S'(G(x)) \leq S'(x)$

$$\Rightarrow \log_2 S'(P(x)) + \log_2 S'(G(x)) \leq 2\log_2 S'(x) - 2$$

$$\Rightarrow R'(P(x)) + R'(G(x)) \leq 2R'(x) - 2$$

$$\Rightarrow AT_{zig\text{-}zag} \leq 2R'(x) - 2R(x)$$

6.2.4 Since $R'(x) \geq R(x)$, we obtain,

$$AT_{zig\text{-}zag} \leq 3(R'(x) - R(x))$$

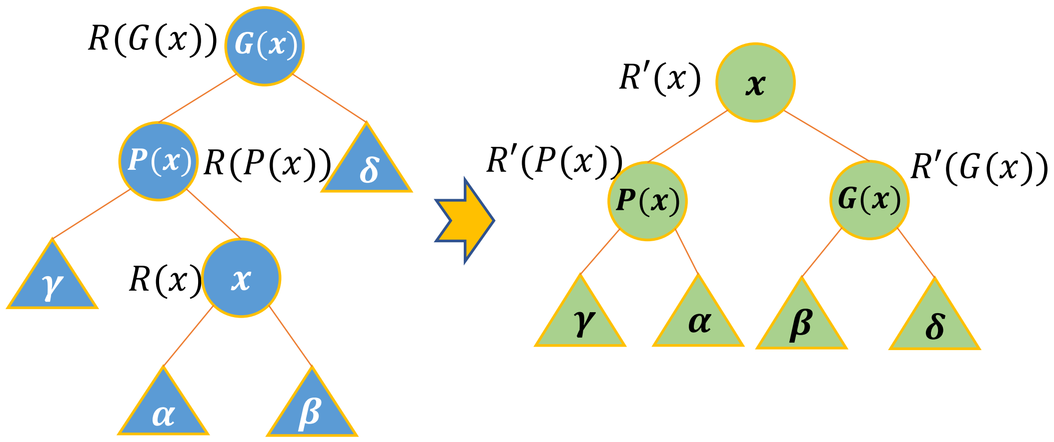
A According to the Arithmetic-Geometric Mean Inequality,

$$\sqrt{ab} \leq \frac{a+b}{2}$$

Let $c = a + b \quad \Rightarrow ab \leq \frac{c^2}{4}$

Taking logarithms of both sides,

$$\log a + \log b \leq 2\log_2 c - 2$$



Amortized Complexity of Splaying (zig-zig 1)

6.3 **Case 3 (zig-zig).** Two operations are done, thus the amortized time ($\boldsymbol{AT_{zig\text{-}zig}}$) of the step is

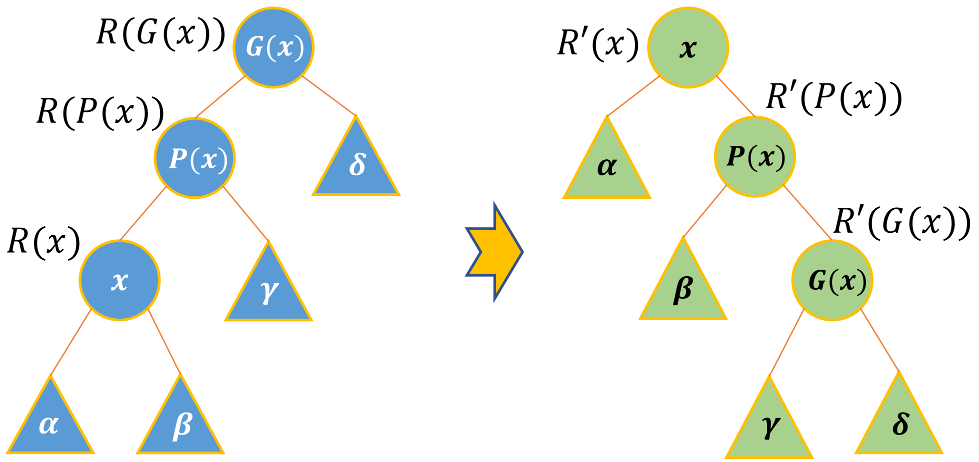
$$\boldsymbol{AT_{zig\text{-}zig}} = \underbrace{\boldsymbol{R'(x) - R(x) + R'(P(x)) - R(P(x)) + R'(G(x)) - R(G(x))}}_{\text{A zig-zig rotation only affects the rank of } \boldsymbol{x}\text{, its parent } \boldsymbol{P(x)} \text{ and its grandparent } \boldsymbol{G(x)}} + \overbrace{\boldsymbol{2}}^{\text{2 operations are needed in a zig-zig rotation}}$$

6.3.1 Given that $\boldsymbol{R'(x) = R(G(x))}$

$$\boldsymbol{AT_{zig\text{-}zig} = R'(P(x)) + R'(G(x)) - R(x) - R(P(x)) + 2}$$

6.3.2 Given that $\boldsymbol{R'(x) \geq R'(P(x))}$ and $\boldsymbol{R(x) \leq R(P(x))}$

$$\boldsymbol{AT_{zig\text{-}zig} \leq R'(x) + R'(G(x)) - 2R(x) + 2}$$



Amortized Complexity of Splaying (zig-zig 2)

$$AT_{zig\text{-}zig} \leq R'(x) + R'(G(x)) - 2R(x) + 2$$

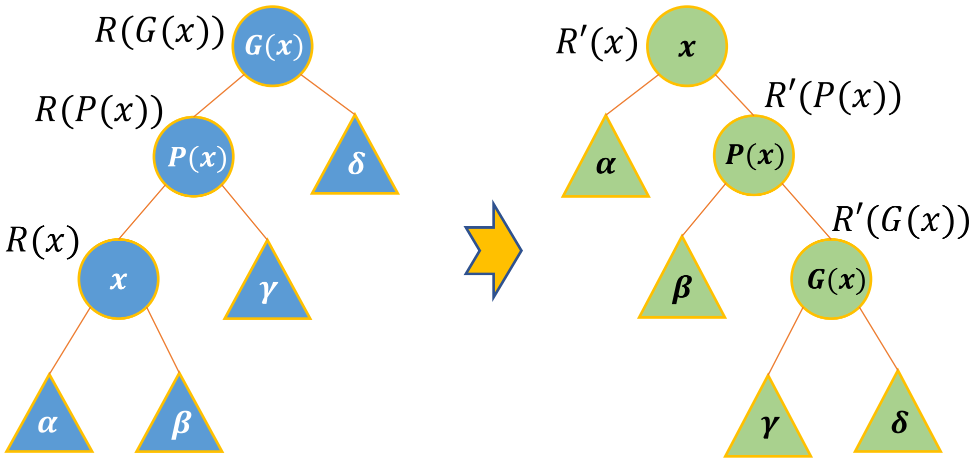
6.3.3 Since $S(x) + S'(G(x)) \leq S'(x)$, according to A $\log a + \log b \leq 2\log_2 c - 2$

$$\Rightarrow R(x) + R'(G(x)) \leq 2R'(x) - 2$$

$$\Rightarrow AT_{zig\text{-}zig} \leq 2R'(x) - 2R(x)$$

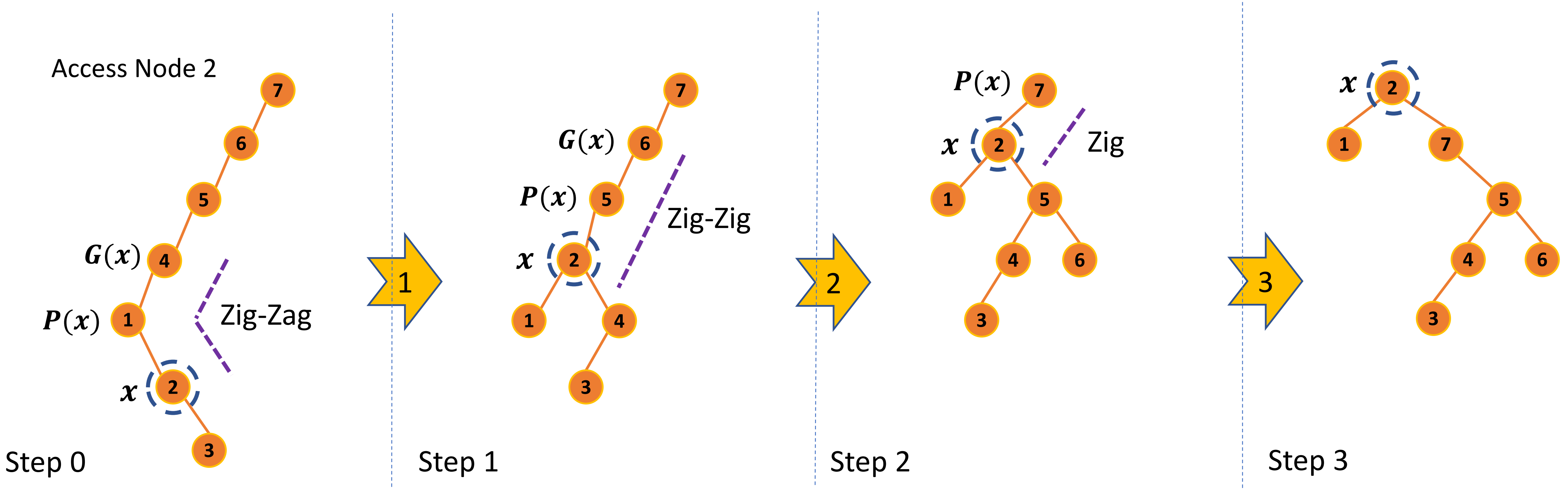
6.3.4 Since $R'(x) \geq R(x)$, we obtain,

$$AT_{zig\text{-}zig} \leq 3(R'(x) - R(x))$$



Example of Splaying

The amortized cost of an entire splay is the sum of the amortized costs of each splay step.



$3(R_{step\text{-}1}(2) - R_{step-0}(2))$

Zig-Zag

$3(R_{step\text{-}2}(2) - R_{step-1}(2))$

Zig-Zig

$3(R_{step\text{-}3}(2) - R_{step-2}(2)) + 1$

Zig

$$3(R_{step\text{-}1}(2) - R_{step-0}(2)) + 3(R_{step-2}(2) - R_{step-1}(2)) + 3(R_{step\text{-}3}(2) - R_{step-2}(2)) + 1 = 3(R_{step\text{-}3}(2) - R_{step-0}(2)) + 1$$

Summary

- In general, adding up the amortized costs of all rotations, the total amortized cost to splay at node $\boldsymbol{x}$ is at most:

$$\mathbf{3}\big(\boldsymbol{R}(\boldsymbol{T}) - \boldsymbol{R}(\boldsymbol{x})\big) + \mathbf{1} = \mathbf{3}(\mathbf{log}\,\boldsymbol{S}(\boldsymbol{T}) - \mathbf{log}\,\boldsymbol{S}(\boldsymbol{x})) + \mathbf{1}$$

Since $\mathbf{log}\,\boldsymbol{S}(\boldsymbol{T}) \geq \mathbf{log}\boldsymbol{S}(\boldsymbol{x})$ and recall that $\boldsymbol{S}(\boldsymbol{T})$ represents the number of descendent of the root and the root itself, i.e., the number of nodes of the tree $\boldsymbol{T}$, which is $\boldsymbol{N}$

thus we obtain an amortized upper bound $\boldsymbol{O}(\mathbf{log}\boldsymbol{N})$

- Insertion and deletion of a node takes $\boldsymbol{O}(\mathbf{log}\boldsymbol{N})$ amortized time
- In the analysis, the potentials for nodes can be arbitrary values, but fixed

B-Trees

- The asymptotic analysis assumes an entire data structure is in the main memory of a computer, when data size is larger than the capacity of the main memory, part of the data structure must be stored on disk. If this happens, the asymptotic efficiency of algorithms will become meaningless, because disk access is much slower than accessing data stored in memory.

- B-tree and its variants, such as B^{+}-trees and B*-trees, were developed to improve the efficiency in tree operations.

Properties of B-trees

A B-tree of degree $\boldsymbol{M}$ is an $\boldsymbol{M}$-ary tree

- The data items are stored at external nodes
- Each internal node store up to $\boldsymbol{M - 1}$ keys to guide the searching
- Keys are stored in nondecreasing order
- The root is either a leaf or has between $\mathbf{2}$ and $\boldsymbol{M}$ children
- All internal nodes (except the root) have between $\left\lceil \frac{\boldsymbol{M}}{\mathbf{2}} \right\rceil$ and $\boldsymbol{M}$ children
- All external nodes are at the same depth and have between $\left\lceil \frac{\boldsymbol{L}}{\mathbf{2}} \right\rceil$ and $\boldsymbol{L}$ data items

- Note, there are many variants of B-trees, each of which has slightly different properties. The above properties are for a popular variant of B-tree called B^+-tree.

B+-tree Example