Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI2100B CSCI2100S DATA STRUCTURES

Spring 2011

AVL Tree

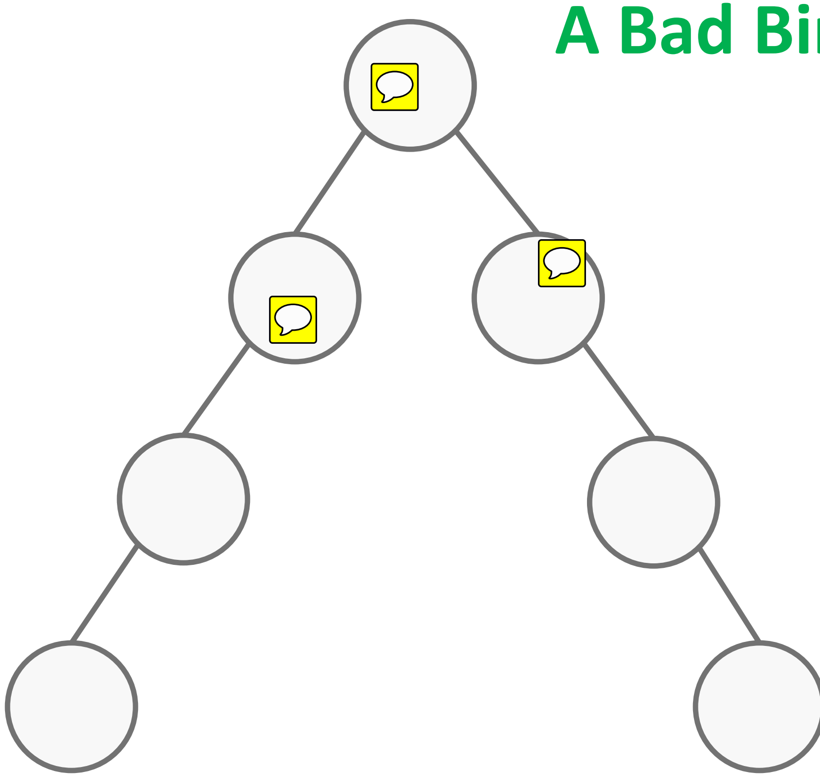
Last updated: **23/03/2011**

Tang Wai Chung, Matthew

AVL Trees

- An **AVL**(Adelson-Velskii and Landis) tree is a binary search tree with a **balance** condition.
- An AVL tree is identical to a binary search tree,
  - except that for every node in the tree, the levels of the left and right subtrees can **differ by at most 1**.
- With an AVL tree, all the tree operations can be performed in $O(\lg N)$, except insertion.

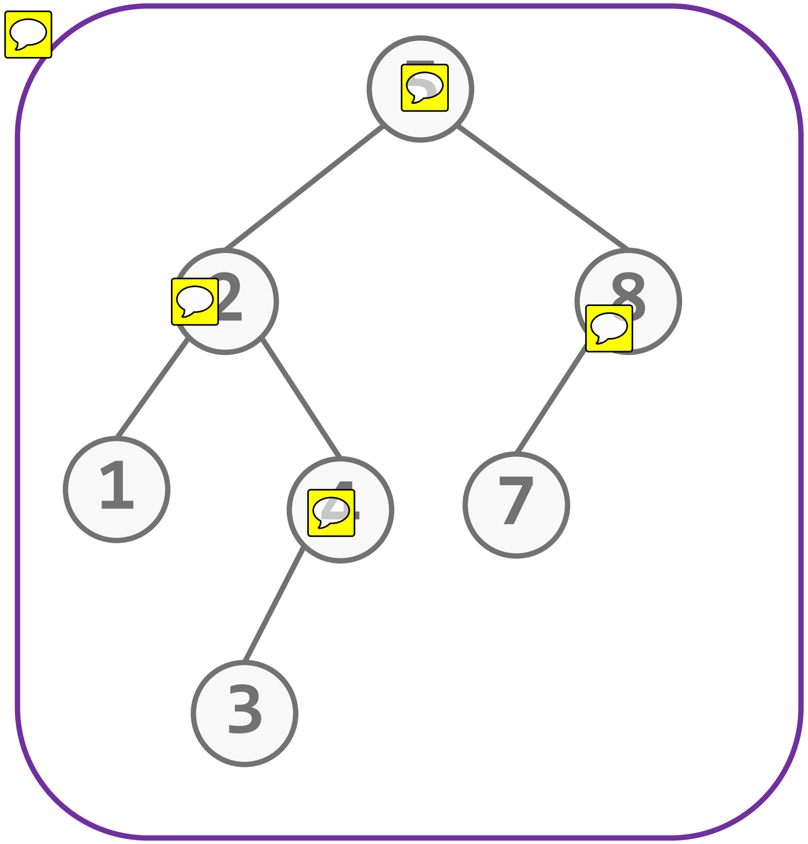
Motivation: Well Balance



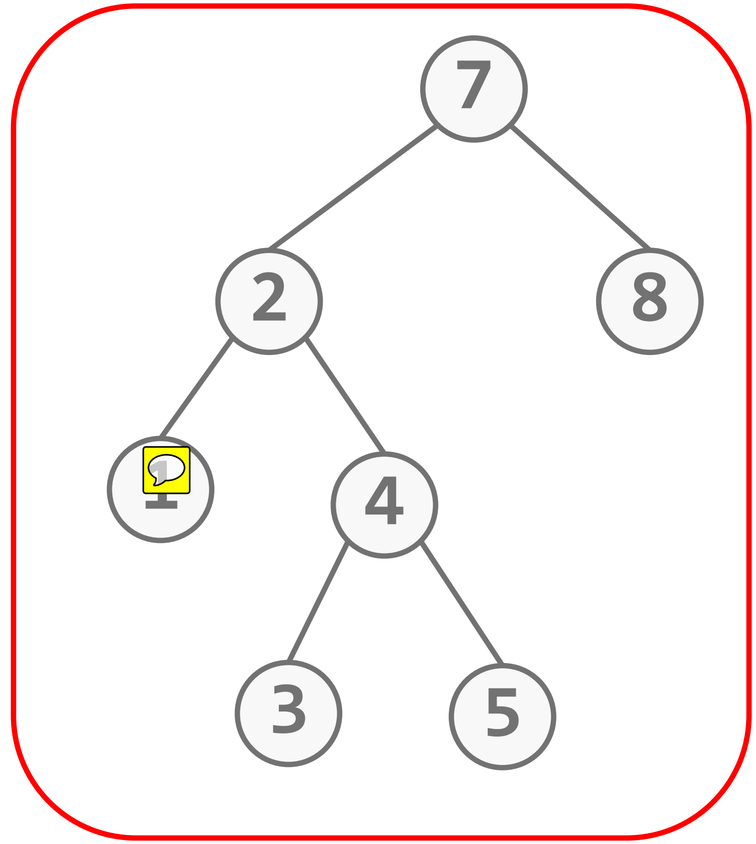
A Bad Binary Tree

Requiring balance at the root is not enough!

AVL Trees: Example



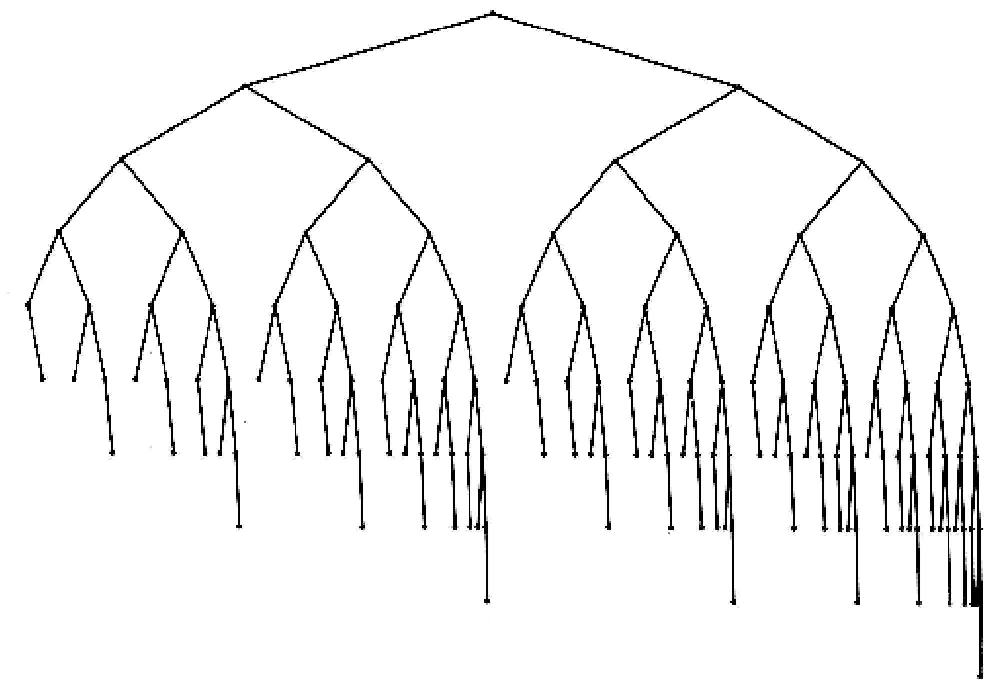
An AVL Tree



Only a BST

AVL Trees: Example (2)

- A **smallest** AVL tree of height 9.
- The construction of the smallest AVL tree of height ***H*** is to use 2 **smallest** AVL sub-trees that are of heights ***H* – 1** and ***H* – 2**.



Height of AVL Trees

- A binary tree of height H cannot have more than 2^H external nodes.
  - $N + 1 \leq 2^H \rightarrow \quad H \geq \lceil lg(N+1) \rceil$
- The minimum number of nodes $S(H)$ in an AVL tree of height H is given by

$$S(H) = S(H-1) + S(H-2) + 1$$

with $S(0) = 1$, $S(1) = 2$.

this relates closely with Fibonacci numbers/trees

Height of AVL Trees (2)

- Thus $S(H)$ can be found by Fibonacci series $F(N)$ and we conclude that

golden ratio

$$N \geq F(H+2) - 1 > \frac{\varphi^{H+2}}{\sqrt{5}} - 2$$

$$\log_{\varphi}(N+2) > (H+2) - \log_{\varphi}\sqrt{5}$$

The **height** of an AVL tree is at most roughly
1.44 log(N + 2) – 0.328 = O(log N)

Observations

- Since the height of an AVL tree is bounded by log *N*, all the tree operations can be performed in ***O*(log *N*)** time, except possibly insertion.
- Insertion and deletion operations need to **update the balancing information**.
- It is sometimes **difficult** since that inserting a node could **violate** the AVL tree property.
- If this is the case, then the property has to be restored before the insertion step is completed.
- The main technique to restore the balance in AVL trees is called **rotation**.

Balanced Factors *B*(•)

- **Balanced factor**: the difference between the heights of the right subtree and the left subtree.
  - +1 (+): right subtree taller
  - 0 (•): balanced
  - -1 (-): left subtree taller
- The balanced factor is kept in **every** node in order to detect out of balance condition.
- An **alternative** is to store the height of the subtree in its root.

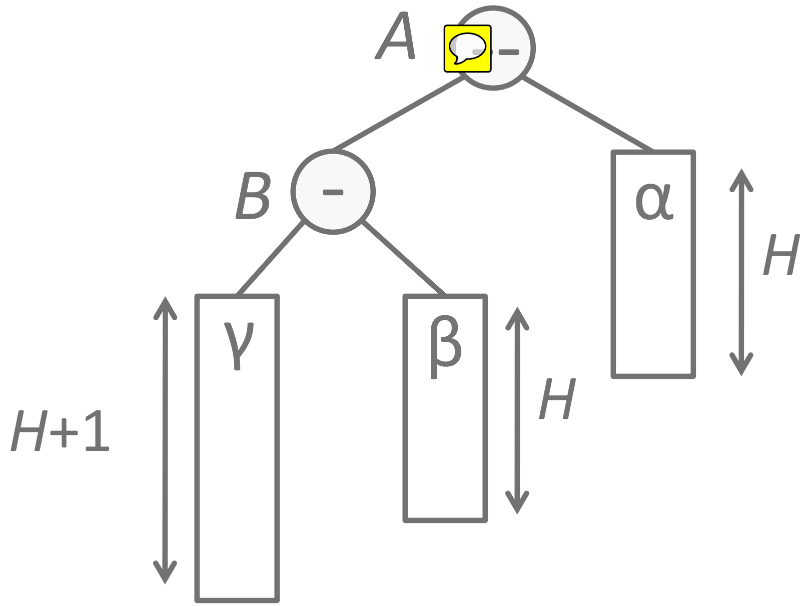
When to Rotate?

- Only nodes that are on the path from the root to the insertion point have balance altered and **possibly** violating AVL condition.
- We'll show how to **rebalance** the tree at this first node and prove this guarantees AVL property of the entire tree.
- Let the node that must be rebalanced be ***A***. Violations happen when
  - the **left** subtree of the left child of ***A*** (*LL*)
  - the **right** subtree of the left child of ***A*** (*LR*)
  - the **left** subtree of the right child of ***A*** (*RL*)
  - the **right** subtree of the right child of ***A*** (*RR*)

How to Rotate?

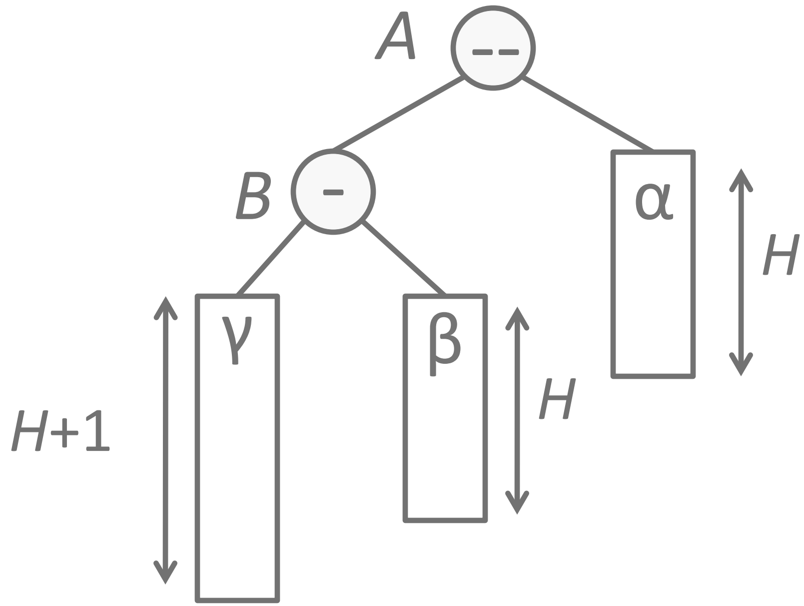
- *LL* and *RR* are mirror image symmetries w.r.t. ***A***, so do *RL* and *LR*.
- For *LL* and *RR* (insert outside), the AVL property can be fixed by a **single** rotation.
- For *LR* and *RL* (insert inside), the AVL property can be fixed by a **double** rotation.
- We will verify that these 2 operations suffice to maintain **balance** for arbitrary trees.

Violations after LL/RR Insertion



- The AVL balance property is violated at ***A***.
- The above figure shows the **ONLY** case: subtree **γ** has grown to an extra level.
- Then ***A*** violates the AVL balance property since its left subtree is 2 levels deeper than its right one.

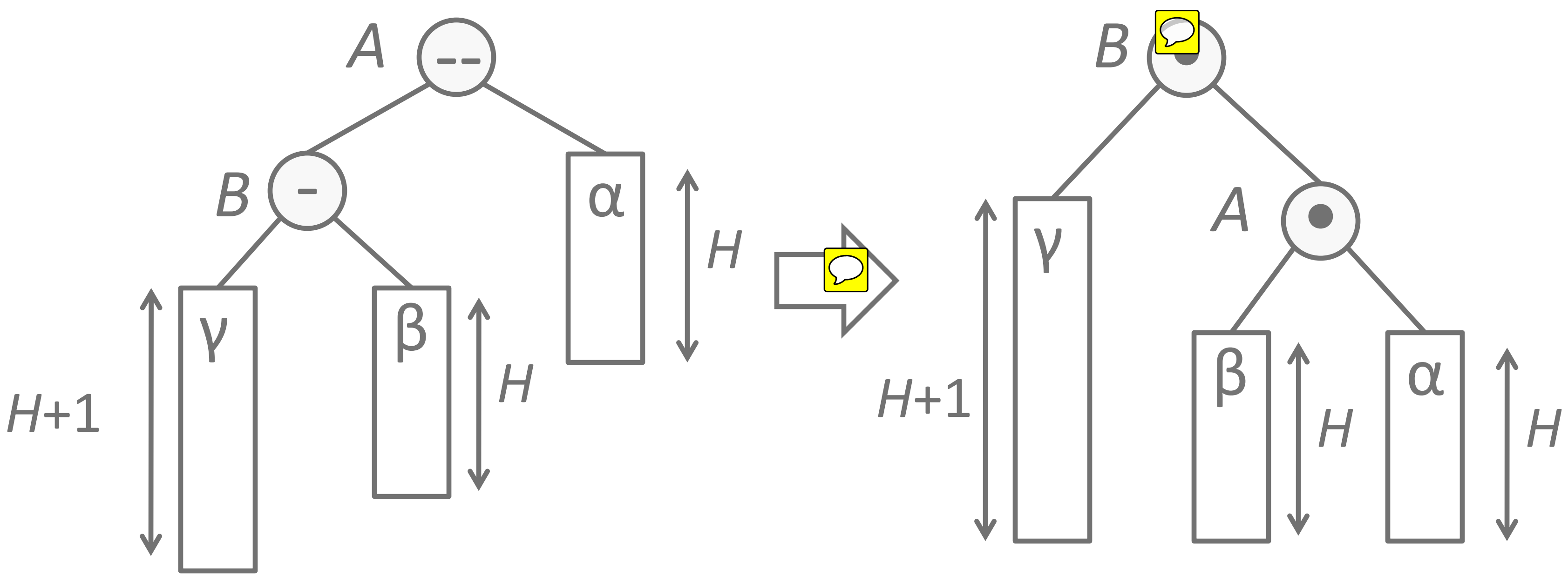
Violations after LL/RR Insertion (2)



- Why this is the **ONLY** case?
  - **γ** must be ONLY 1 level taller than **β**, otherwise the original tree is not AVL.
  - **γ** is 1 level lower than **α** due to the existence of ***B***.

Single Rotation: *LL*

Take ***B*** up and ***A*** down. Connect **β** to ***A***

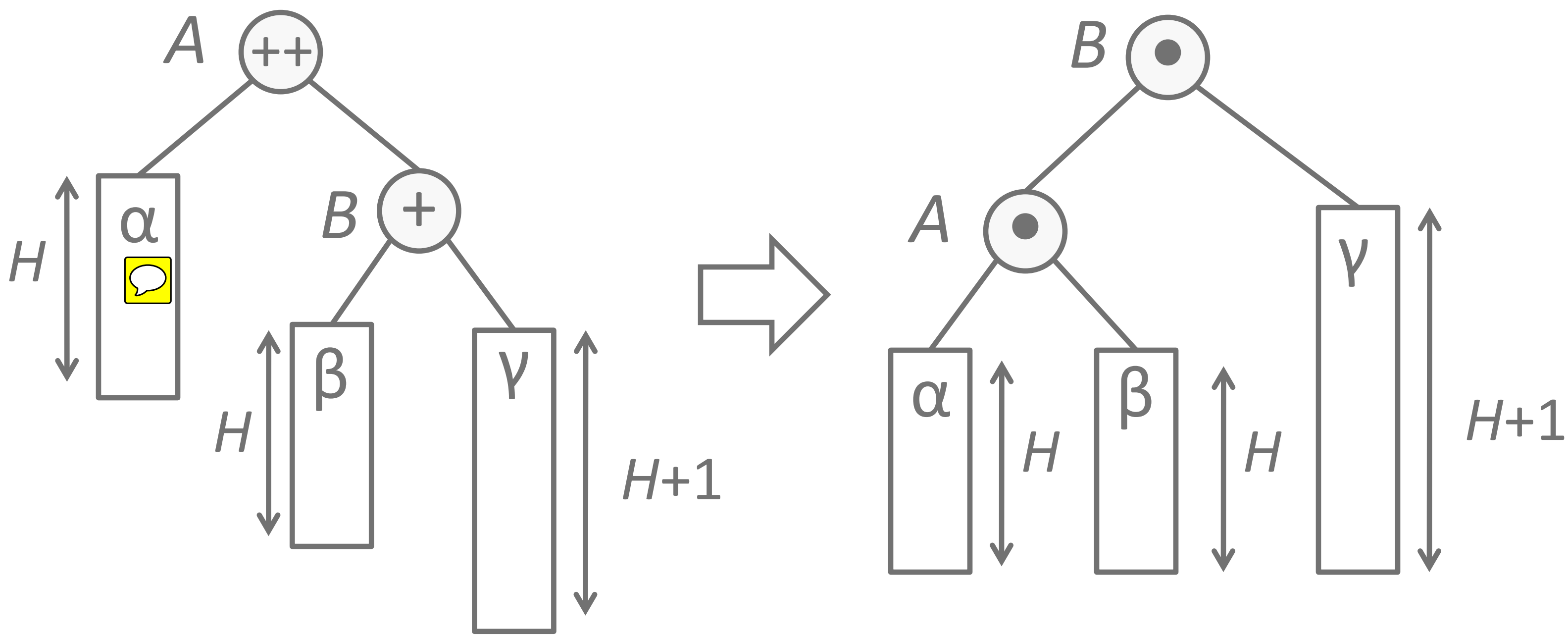


BST property: *B* < **β** < *A*

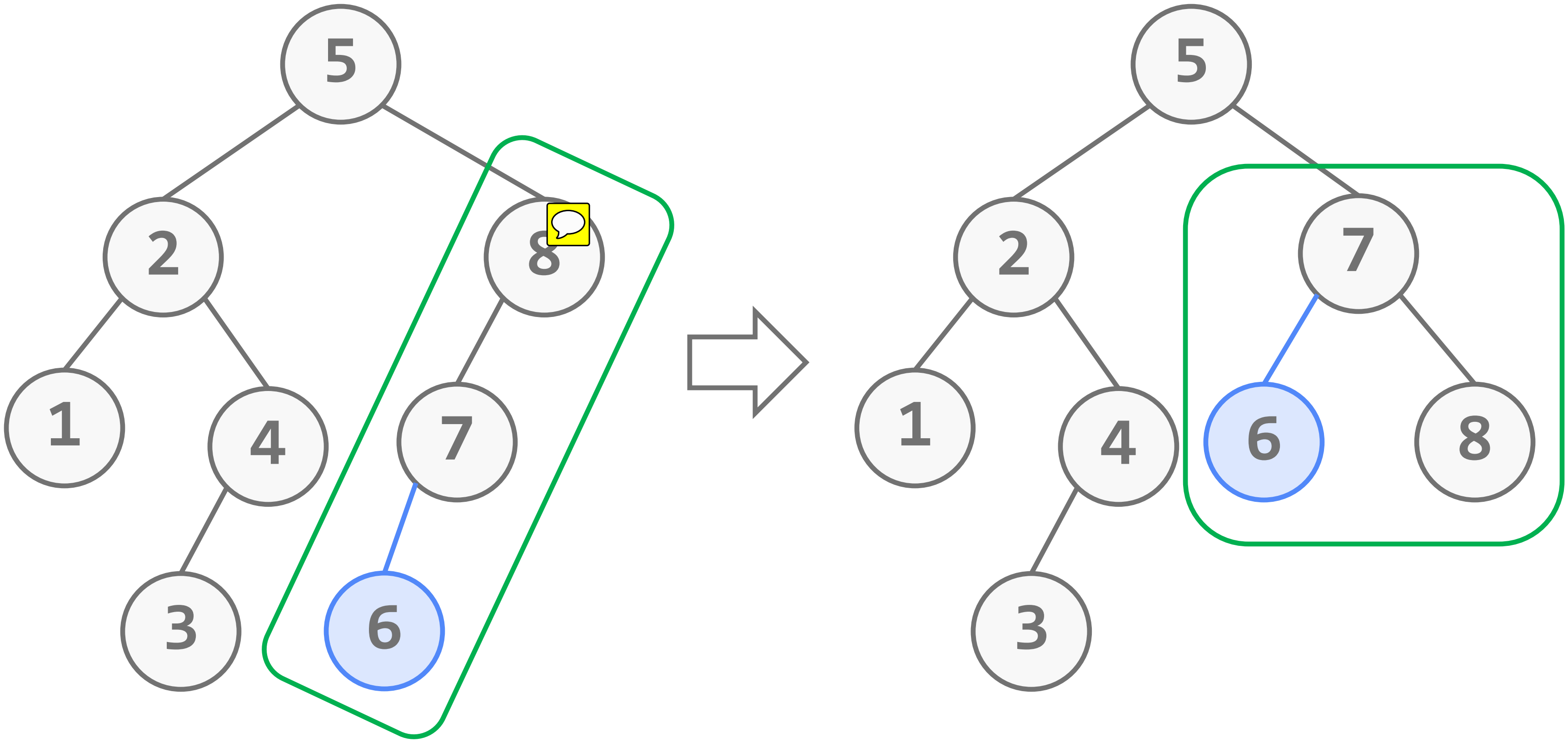
AVL Property: Balance restored. The height is the original AVL tree → no updates required.

Single Rotation: *RR*

Take ***B*** up and ***A*** down. Connect **β** to ***A***



Single Rotation: Example

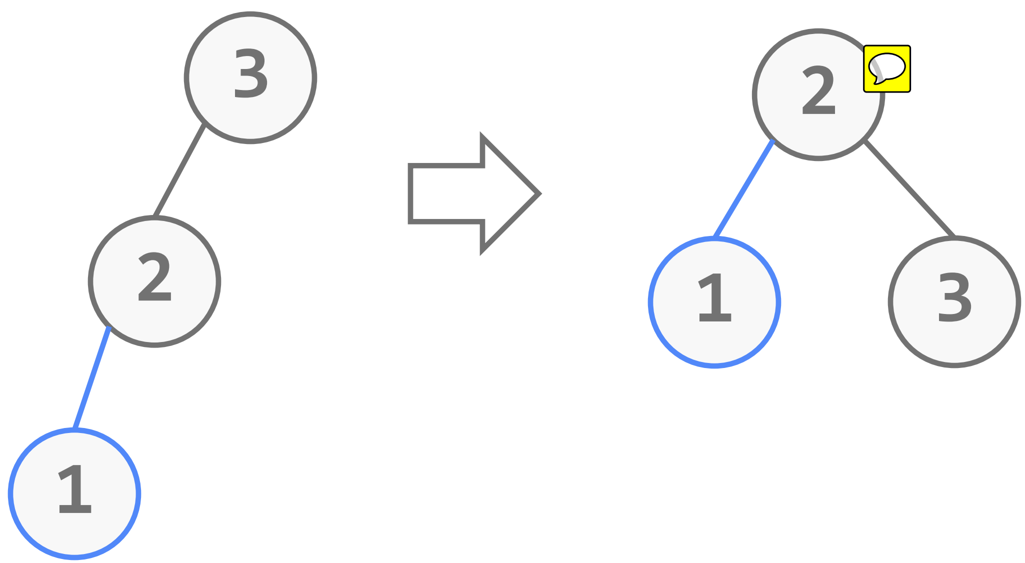


AVL property destroyed by insertion of 6, then fixed by a rotation

Building an AVL Tree

Suppose we start with an initially empty AVL tree and insert keys 3, 2, 1 and then 4 through 7 in sequence order.

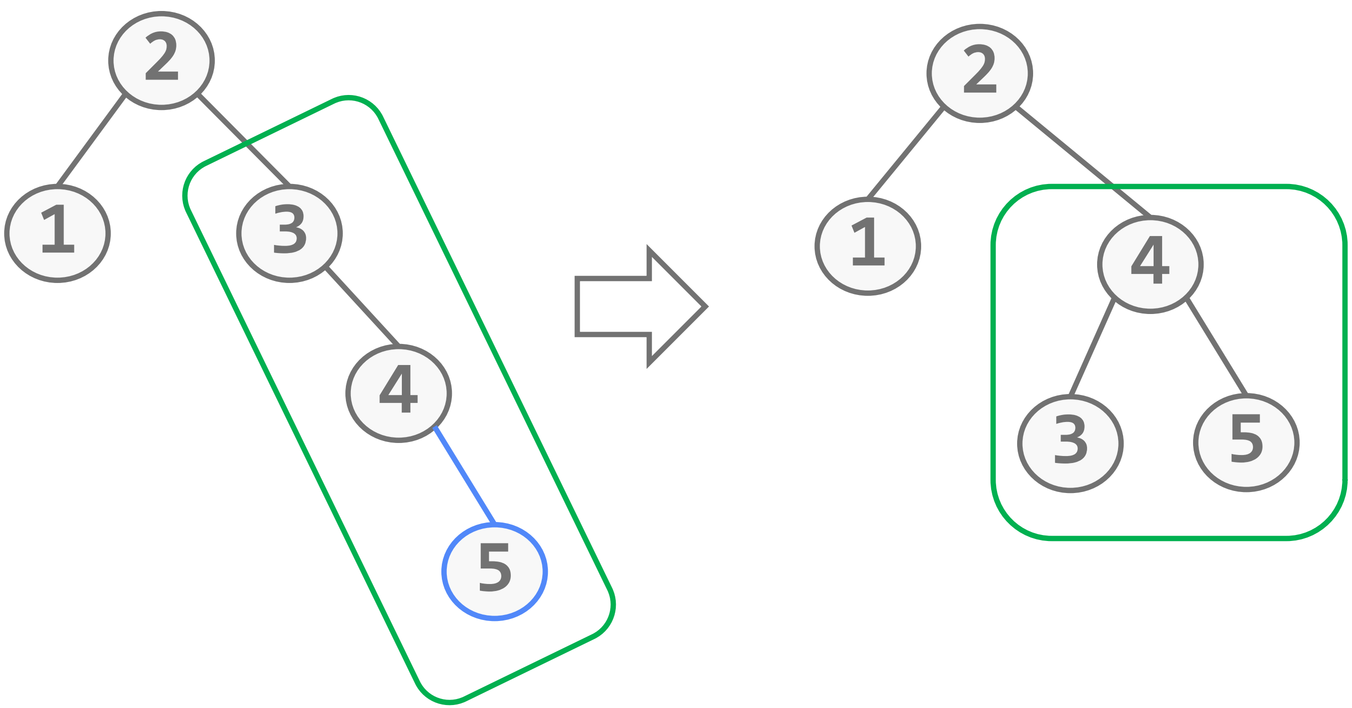
First problem comes when we want to insert 1. We perform a single rotation between the root and its left child to fix the problem.



Building an AVL Tree (2)

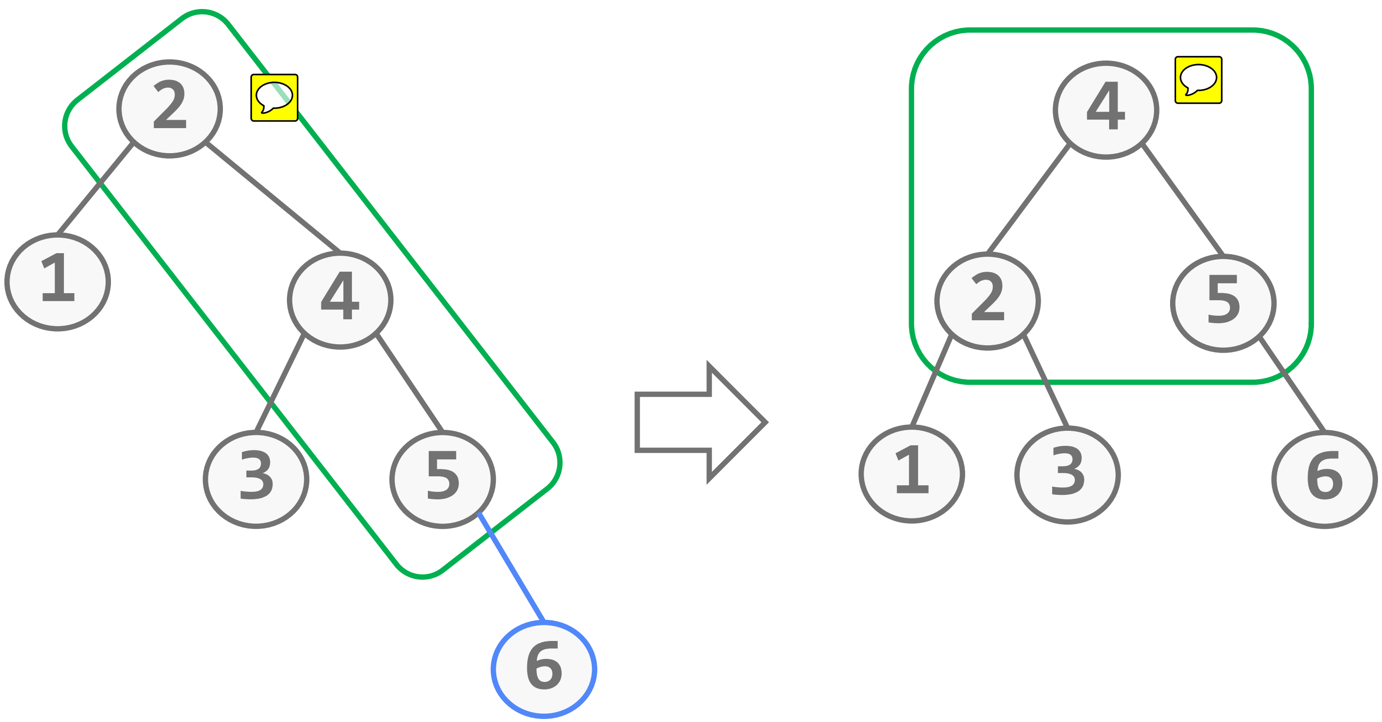
The insertion of 5 creates a violation at 3 that is fixed by a single rotation.

Don't forget to update pointers at 2. Otherwise...



Building an AVL Tree (3)

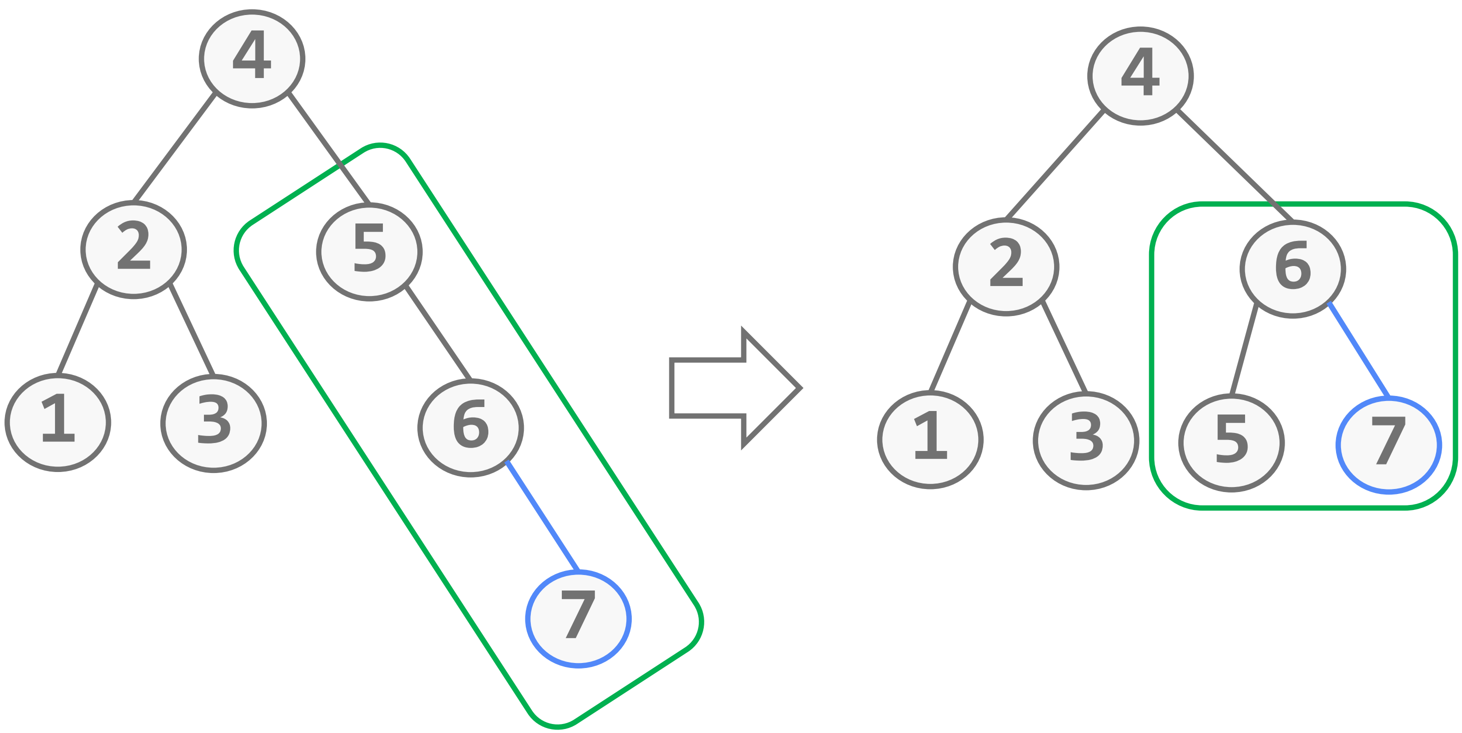
We continue to insert 6. This causes a balance problem at the root.



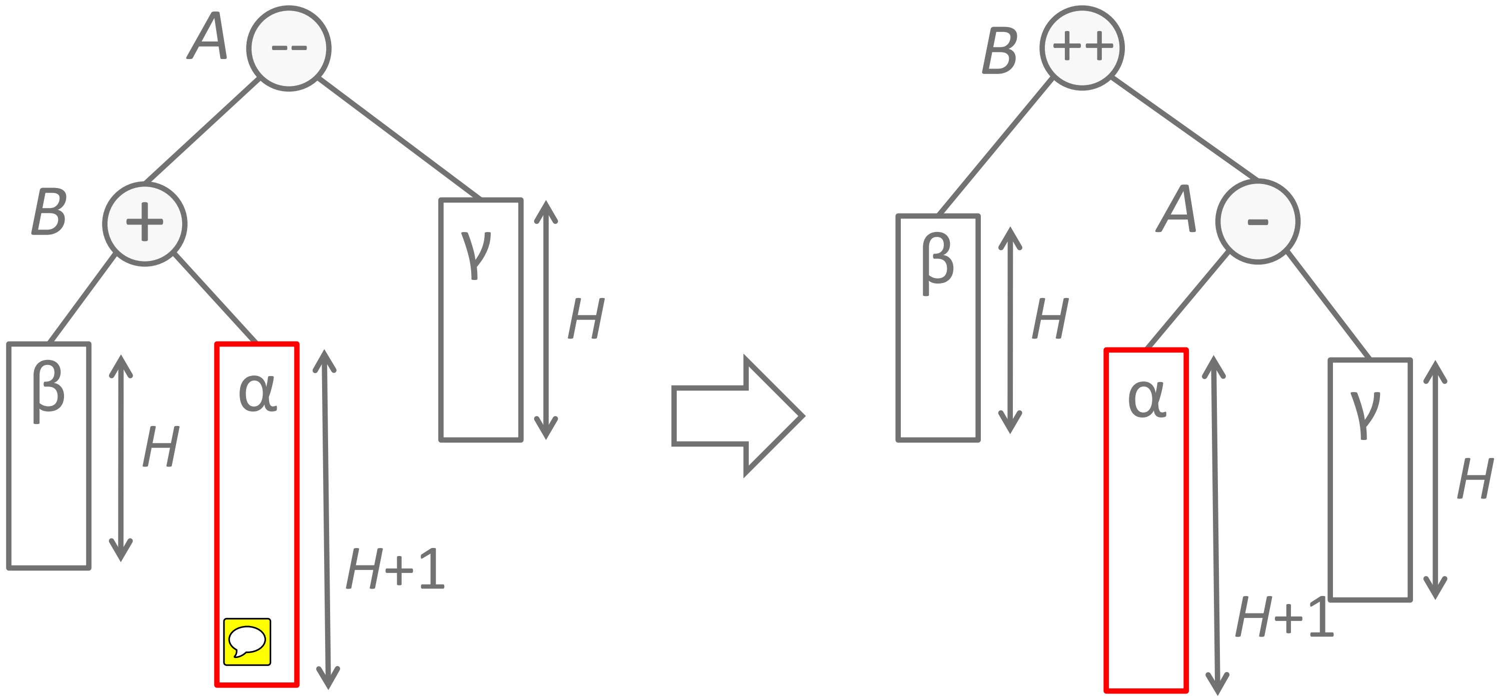
Building an AVL Tree (4)

The last key 7 causes another rotation.

The final tree is a well-balanced complete binary tree.

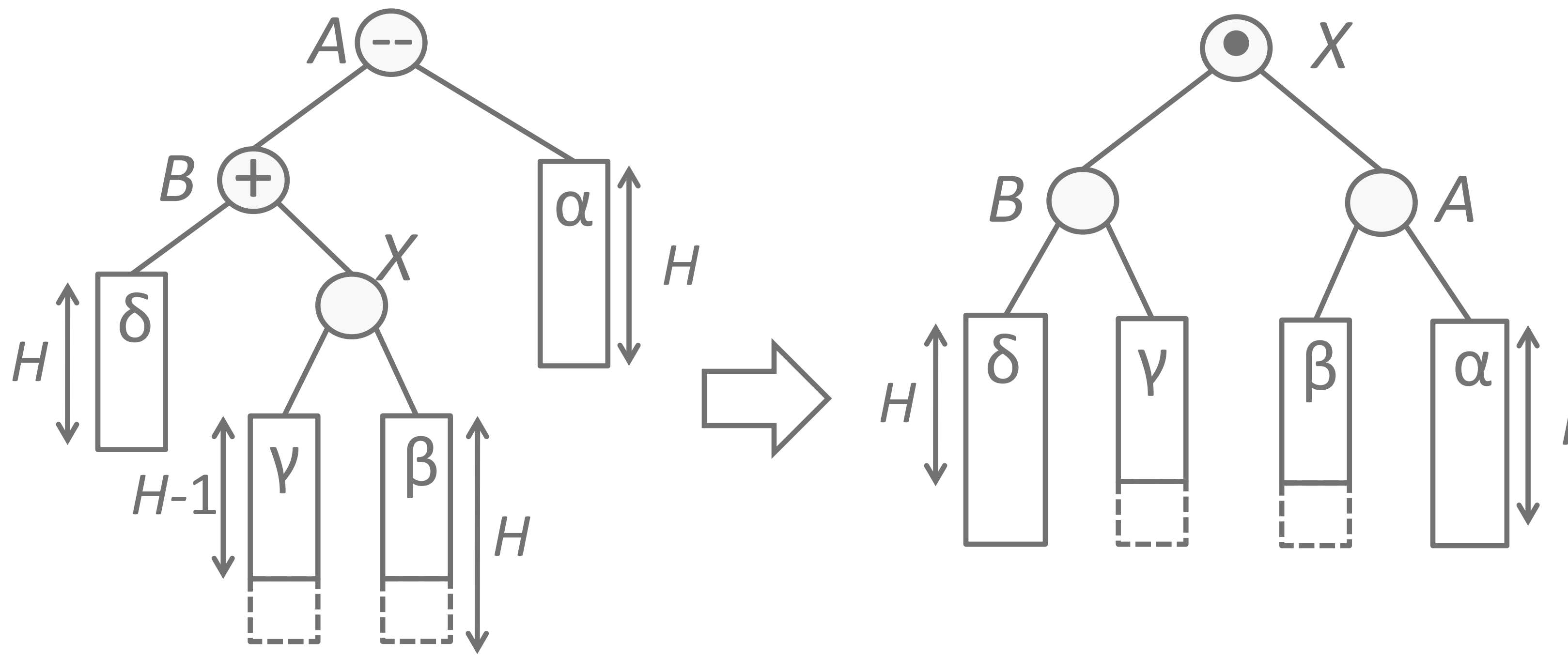


Single Rotation Fails with *LR*/*RL*



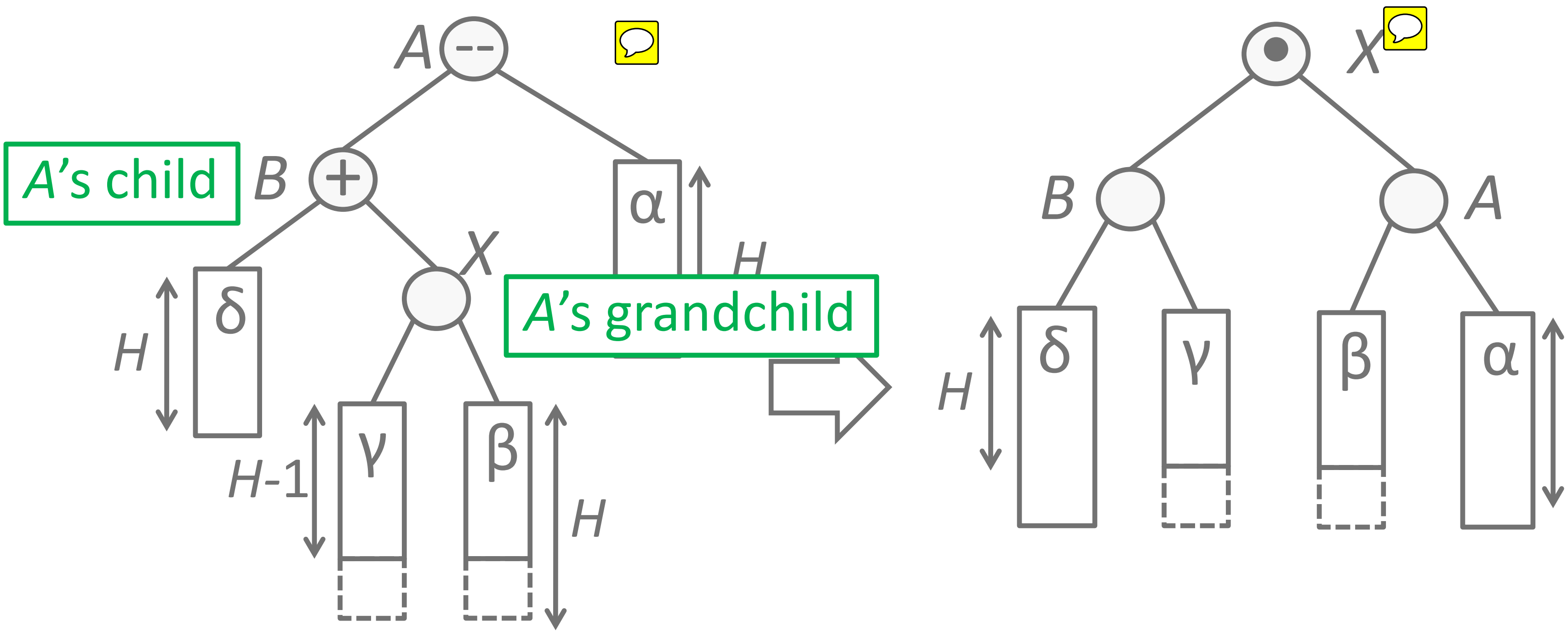
- When violation happens with *LR* or *RL*, a single rotation cannot fix the problem.
- The subtree **α** is still to deep.
- Thus we have to use **double** rotation.

Double Rotation: *LR*



- Subtrees **β** and **γ** may have height of *H* or *H* - 1.
- BST: *B* < ***X*** < *A*, *X* < **β** < *A*, *B* < **γ** < *X*
- AVL: The max. height difference is 1. The height of the tree is same as before insertion. No updates required.

Double Rotation: *LR*

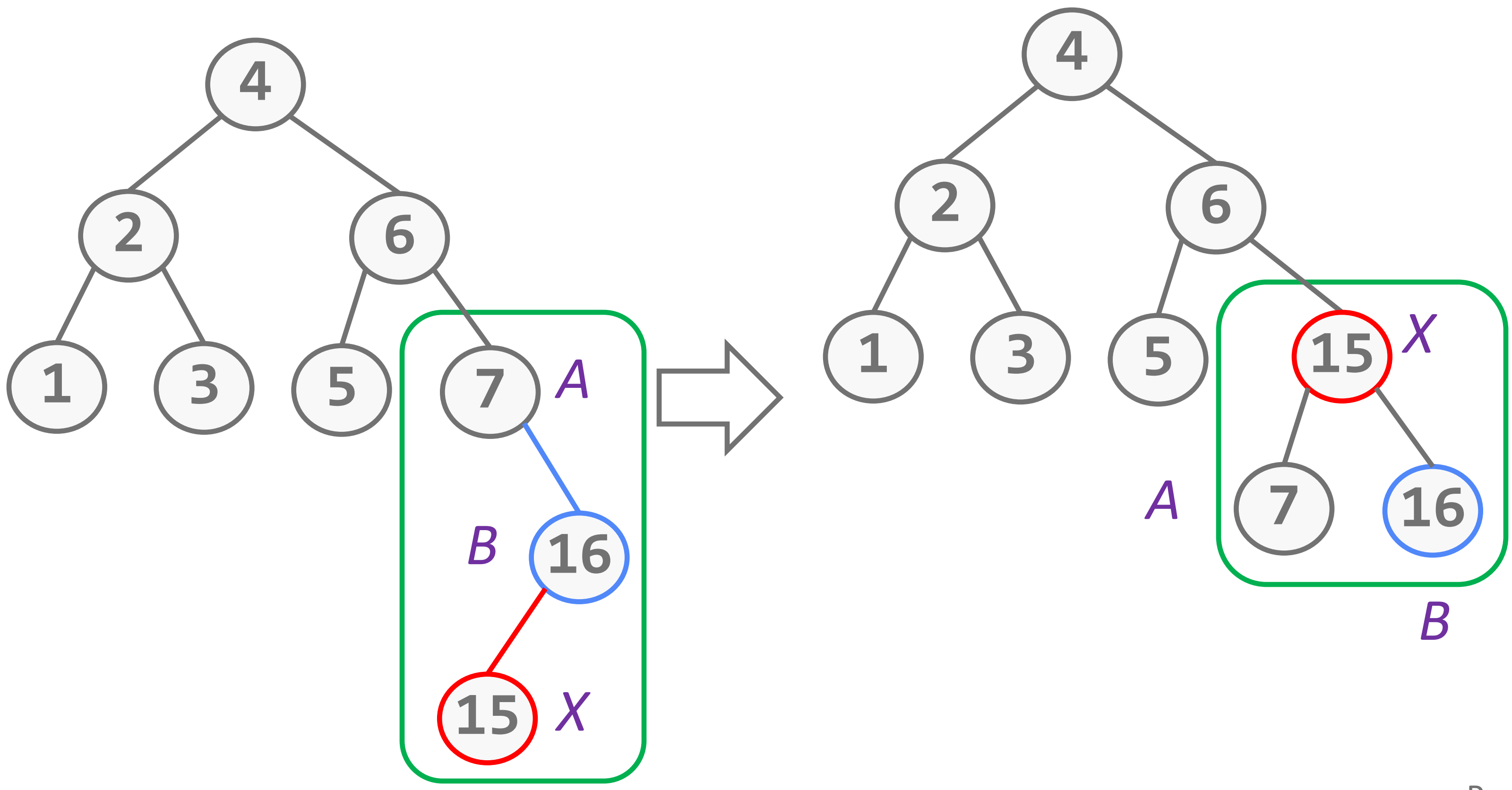


- Note: double rotation is equivalent to rotating between ***A***'s ***child*** and ***grandchild***, and then between ***A*** and its new ***child***.

Building an AVL Tree (5)

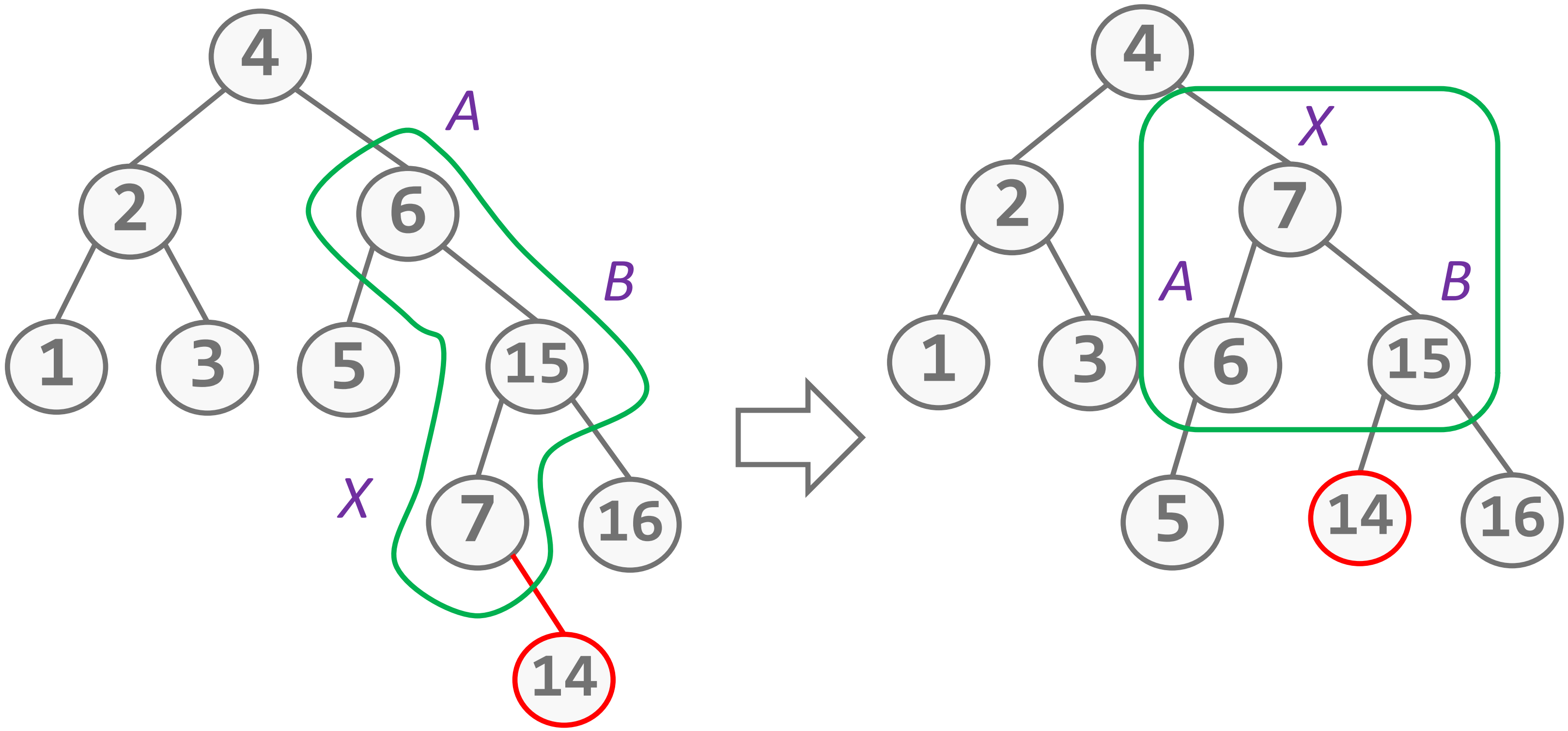
Continue the building of our AVL tree with 16 down to 14.

Inserting 16 is easy. Adding 16 generates a RL case, which is solved with a double rotation.



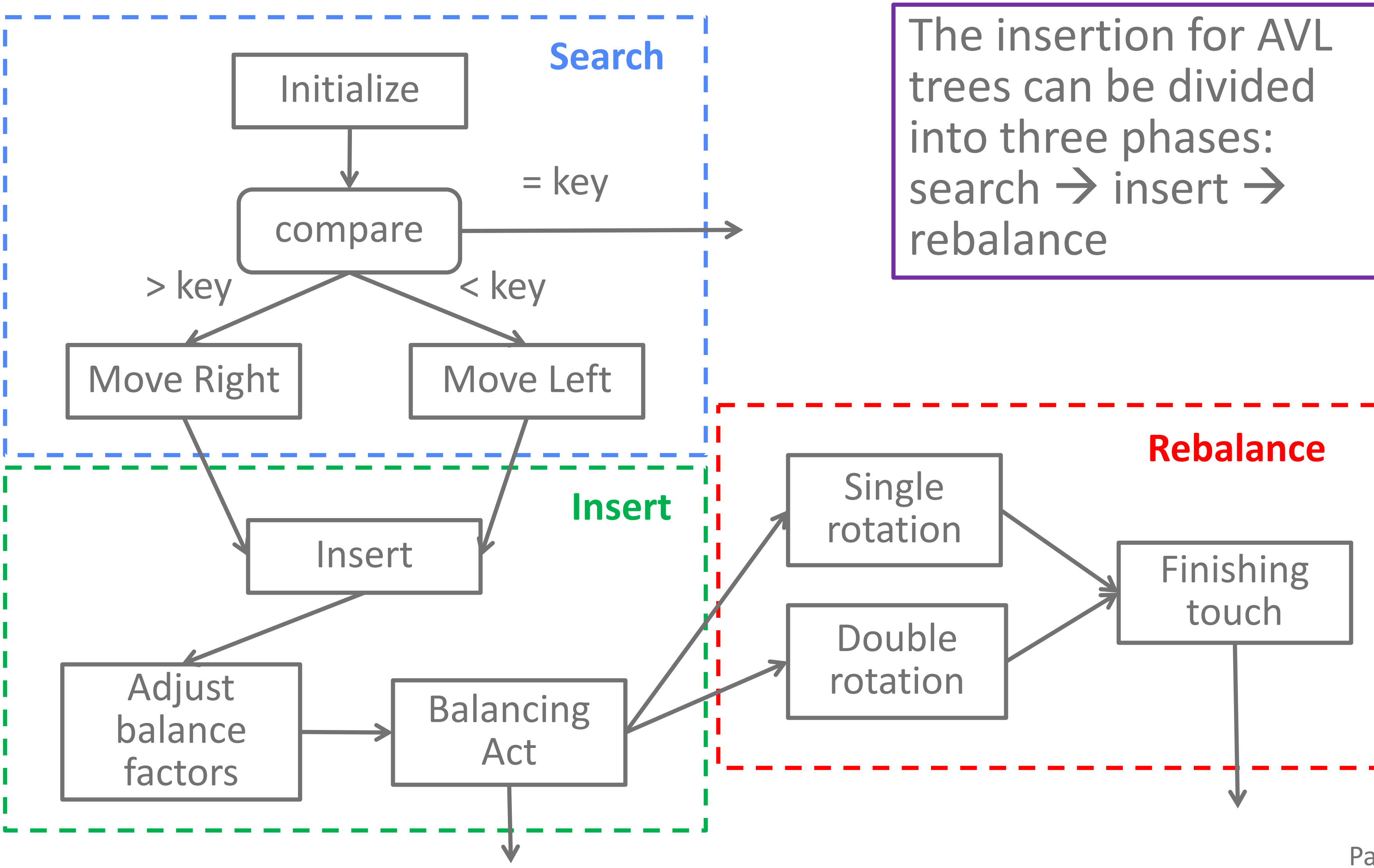
Building an AVL Tree (6)

Inserting 14 requires another double rotation on 6, 15 and 7.



You may continue the process by trying to inserting 13 down to 10, then 8 and 9.

AVL Trees: Insert



The insertion for AVL trees can be divided into three phases: search → insert → rebalance

AVL Trees: Insert (2)

Search & Insert

```
/* [Search] & [Insert]
 * (t, s): closest unbalanced point along the search path */
    for (t = h, s = p = h->r; k != p->k; p = q){
        a = k < p->k ? -1 : +1;
        if ((q = p->link[a]) == NULL){
            p->link[a] = q = avl_new_node(k);
            break;
        }
         if (q->b)
            t = p, s = q;
     }
    if (k == p->k) /* key found */
         return p;
```

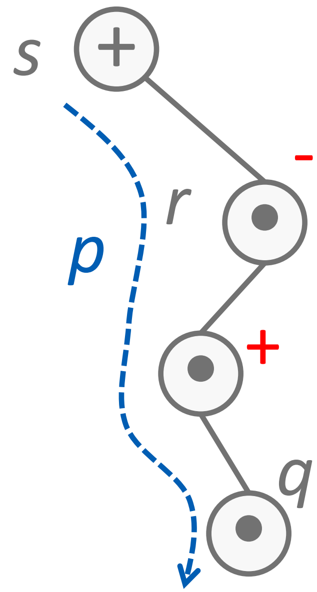
```
Note:
p->link[-1] = p->l
p->link[+1] = p->r
```

When the key is found, just return the pointer and terminate the insertion.

AVL Trees: Insert (3)

Adjust balanced factors

```
/* [Adjust]
 * Start from s, update balance factors
 * until we reach the new node q.   */
    a = k < s->k ? -1 : +1;
    r = p = s->link[a];
    while (p != q) {
        p->b = k < p->k ? -1 : +1;
        p    = k < p->k ? p->l : p->r;
    }
```



(s, r) points to where rotation may happen.

s refers to *A* while r refers to *B*

AVL Trees: Insert (4)

Balancing acts

```
    if (s->b == 0){
        s->b = a;
        h->l = (avl_t *)(((int)h->l) + 1);
        return q;
     }
     else if (s->b == -a){
         s->b = 0;
         return q;
     }
```

Case (i): tree grown higher (s is the root node)

Case (ii): tree grown more balanced
balanced factor is opposite to the side of insertion.

AVL Trees: Insert (5)

Rotations

```
if (r->b == a){ /* single R */
    p = r;
    s->link[a] = r->link[-a];
    r->link[-a] = s;
    s->b = r->b = 0;
}
else if (r->b == -a){ /* double R */
    p = r->link[-a];
    r->link[-a] = p->link[a];
    p->link[a] = r;
    s->link[a] = p->link[-a];
    p->link[-a] = s;
    s->b = p->b == a ? -a : 0;
    r->b = p->b == -a ? a : 0;
    p->b = 0;
}
/* final touch */
t->link[s == t->r ? +1 : -1] = p;
```

s: node A
r: node B

s: node A
r: node B
p: node X

Don't forget to link back the rotated subtree back to the main trunk.

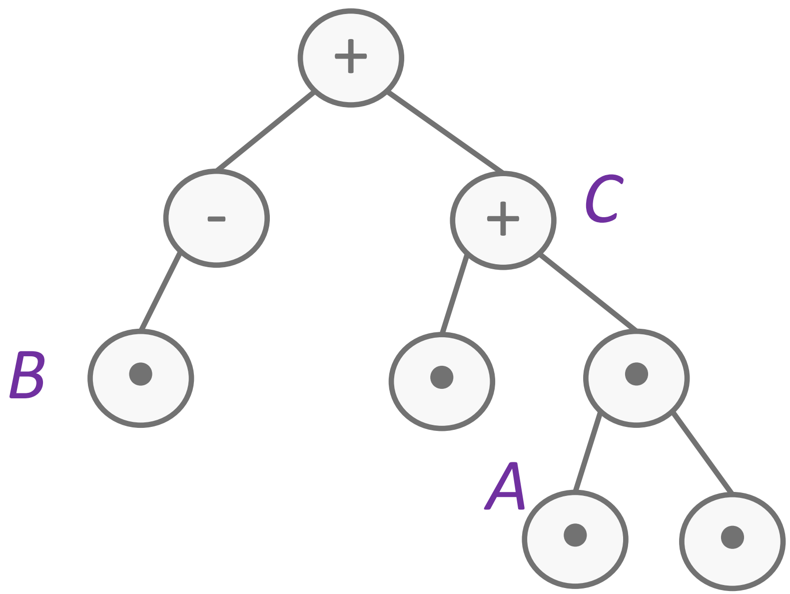
AVL Trees: Delete

- We are not going to discuss ***delete*** for AVL trees since it is much more complicated then ***insert***.
- Use **lazy** deletion is a good option if deletions are relatively infrequent.

AVL Trees: Delete

- First recall that the ***delete*** procedure in BST:
  - Find the node storing the key
    - If the node has no or 1 child, delete it directly.
    - If the node has 2 children, go and find the minimum nodes in its right subtree. Copy the key and remove that node.
- In AVL trees, we can repeat the same procedure and then **rebalance** the tree after deletion (just like AVL ***insert***).
- Let us investigate an example first.

AVL Trees: Delete (2)



When you delete a node on the left(right) subtree, apparently the right(left) subtree becomes taller.

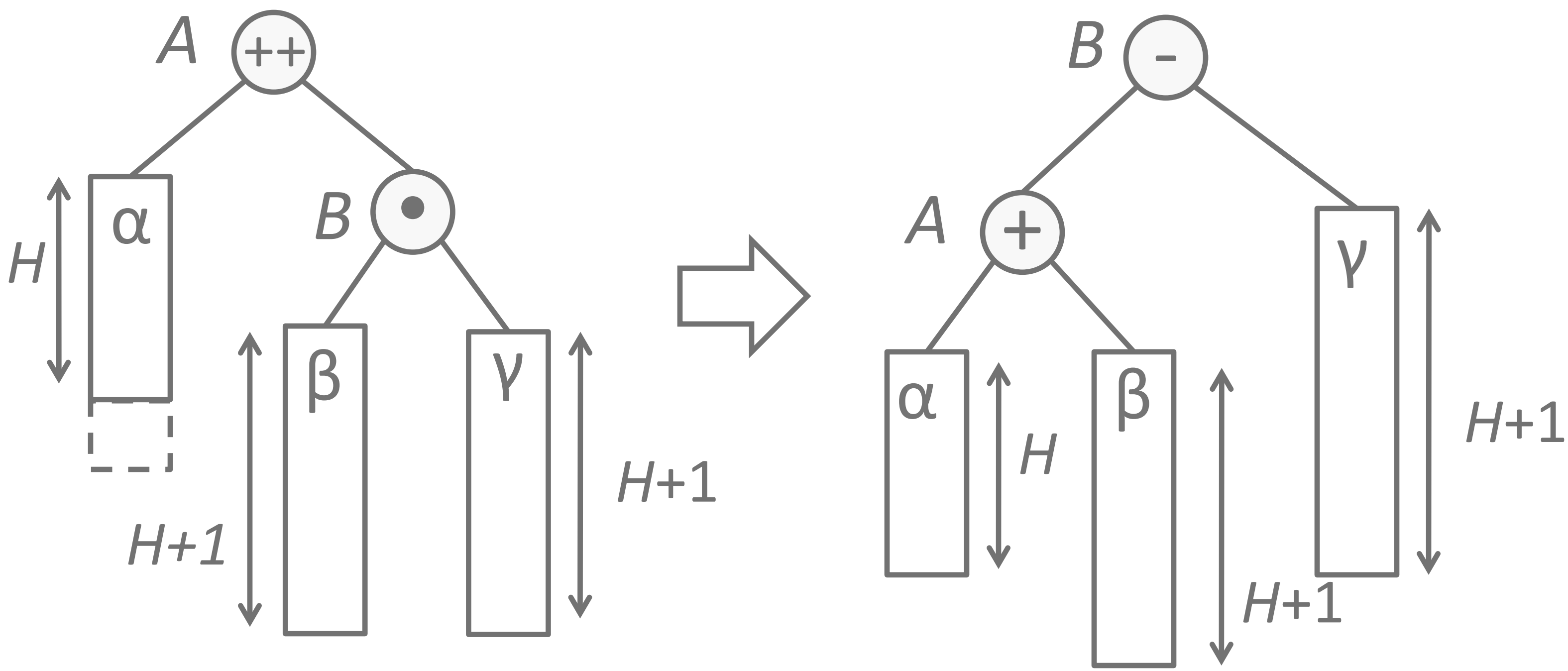
- Imagine that you are going to delete *A*, *B* or *C* from the AVL tree.
- How will you adjust the balanced factors after node deletion?
- A & C: + + -, B: (++) • → **rotation is required**!

AVL Trees: Delete (3)

- Case study
  - <u>$B(P) = a$</u>: the ***a*** subtree is taller and you delete the node on that side
    The tree becomes balanced but the height of the tree may have decreased.
  - <u>$B(P) = 0$</u>: the tree is balanced and you delete the node on either side
    The –a subtree becomes taller. Still okay.
  - <u>$B(P) = -a$</u>: the ***a*** subtree is taller and you delete the node on another side
    Rebalance needed

AVL Trees: Delete (4)

- An extra case where rotation is needed.



AVL Trees: Delete (5)

- The most tedious part of ***delete*** is that we may need to rotate up to **log *N*** times to rebalance the whole tree.
- The **worse** case: deleting rightmost node of the Fibonacci tree
  The **average** case: 0.21 rotations
- Therefore we need to keep track of the search path using a stack.
  - **$(P_0$=h, a_0=1), (P_1, a_1), ..., $(P_j=P, a_j$=NULL)**
  - After deletion, track back the path, adjusting the balanced factors and rebalancing the tree if necessary.

AVL Trees: Delete (6)

Search & Push

stack routines are skipped.

```
stack_init();
stack_push(h, 1);

for (p = h->r; p; p = p->link[a]){
    a = k < p->k ? -1 : +1;
    if (k == p->k){
        a = p->l == NULL ? -1 : +1;
        if (p->l && p->r){
            k = p->k = avl_find_min(p->r)->k;
            a = 1;
        }
    }
    stack_push(p, a);
}
/* [Delete] */
stack_pop(&d, &tmp_a);
stack_top(&p, &a);
p->link[a] = d->link[-tmp_a];
```

Note:

find_min **is the same as the one we used in BST.**

AVL Trees: Delete (7)

Balancing acts

```
    /* [Balancing acts] */
    while (!stack_is_empty()){
        stack_pop(&s, &a);
        if (s == h){ /* the header node */
            s->l = (avl_t *)((int)s->l - 1);
            return d; /* done */
        }
        if (s->b == a){ /* tree grows more balanced */
            s->b = 0;
            continue;
        }
        if (s->b == 0){ /* tree grows shorter */
            s->b = -a;
            return d; /* done */
        }
```

If we reach the header node, the tree has decreased in height.

AVL Trees: Delete (8)

Rotations

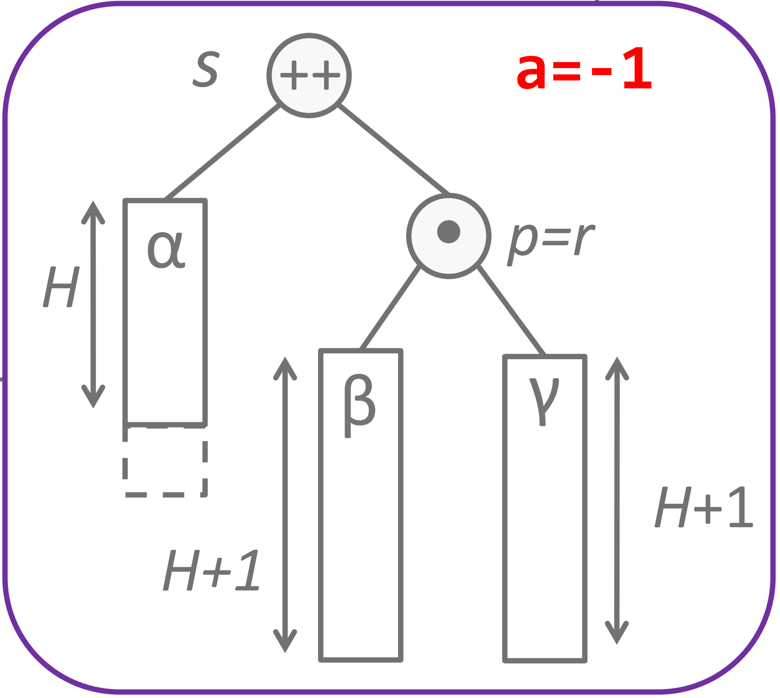
```
        a = -a; /* rotate on opposite side */
        r = s->link[a];
        if (r->b == a){ /* single R */
            p = r;
            s->link[a] = r->link[-a];
            r->link[-a] = s;
            s->b = r->b = 0;
            stack_top(&s, &a);
            s->link[a] = p;
        }
        else if (r->b == -a){ /* double R */
            p = r->link[-a];
            r->link[-a] = p->link[a];
            p->link[a] = r;
            s->link[a] = p->link[-a];
            p->link[-a] = s;
            s->b = p->b == a ? -a : 0;
            r->b = p->b == -a ? a : 0;
            p->b = 0;
            stack_top(&s, &a);
            s->link[a] = p;
        }
```

AVL Trees: Delete (9)

Rotations: extra case

```
        else if (r->b == 0){ /* single R & terminate */
            p = r;
            s->link[a] = r->link[-a];
            r->link[-a] = s;
            s->b = a;
            r->b = -a;

            stack_top(&s, &a);
            s->link[a] = p;
            break;
        }
    }
    return d;
}
```



AVL Trees: Analysis

- Rotations are **constant-time**. $O(1)$
- ***search***: go down the tree, farest to the leaves. $O(H) = O(\log N)$
- ***insert***: preceded with ***search***. Balancing acts and rotations are $O(1) \rightarrow O(H) = O(\log N)$
- ***lazy-delete:*** $O(1)$
- ***delete***: identify deletion path $\log N$, followed by at most $\log N$ rotations to rebalance the tree $\rightarrow$ $O(\log N)$

Summary

- Define the concept of **balanced** tree
- Introduce constant-time **rotation** techniques to restore tree balance while preserving search tree property.
- Use AVL trees as an application of rotations
- Reveal the **implementation** skills for AVL trees.
- Analysis the **worse case** performance of AVL trees insertion and deletion.