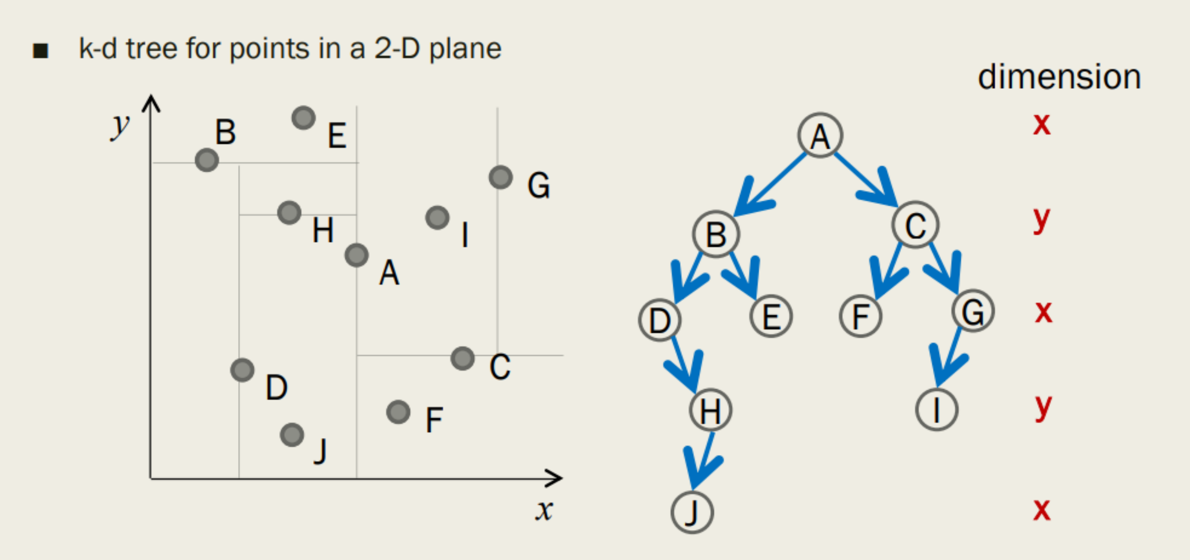
k-d tree



- binary search tree
- each level represents different dimension

insert

```
void insert(node *&root, Item item, int dim) {
    if(root == NULL) {
        root = new node(item);
    return;
 }
 if(item.key == root->item.key) // equal in all dimensions
     return;
 if(item.key[dim] < root->item.key[dim])
     insert(root->left, item, (dim+1)%numDim);
 else
     insert(root->right, item, (dim+1)%numDim);
}
```

search

similarly to insert

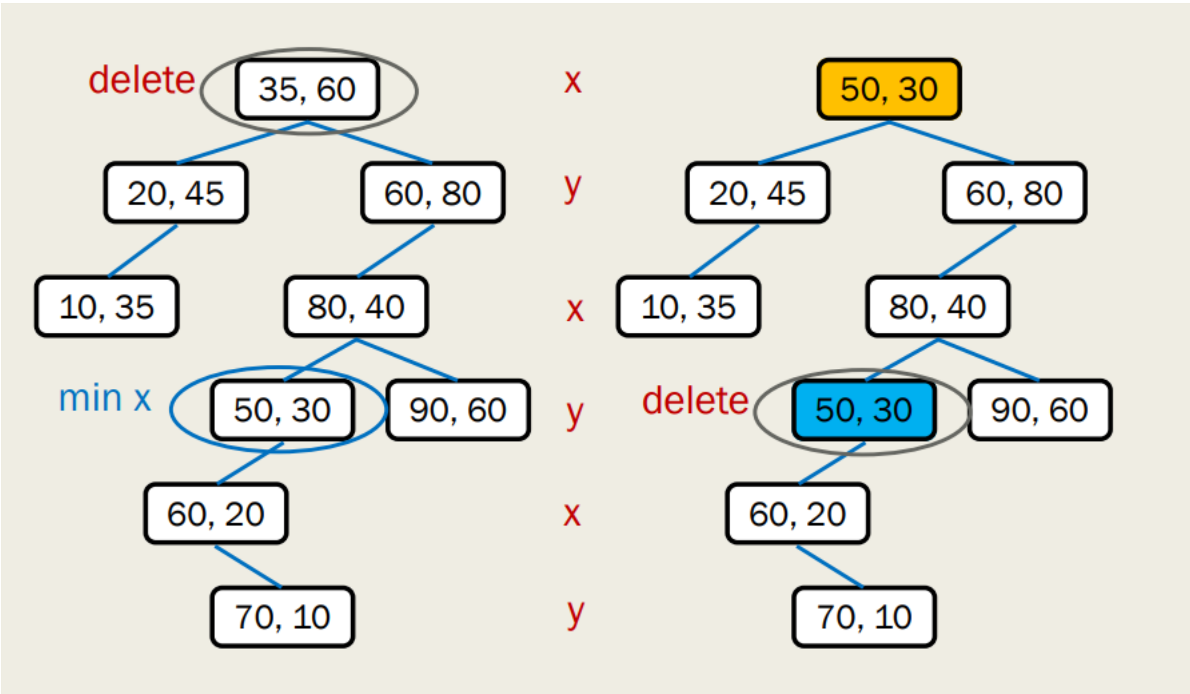
```
node *search(node *root, Key k, int dim) {
    if(root == NULL) return NULL;
    if(k == root->item.key)
        return root;
    if(k[dim] < root->item.key[dim])
        return search(root->left, k, (dim+1)%numDim);
    else
        return search(root->right, k, (dim+1)%numDim);
}
```

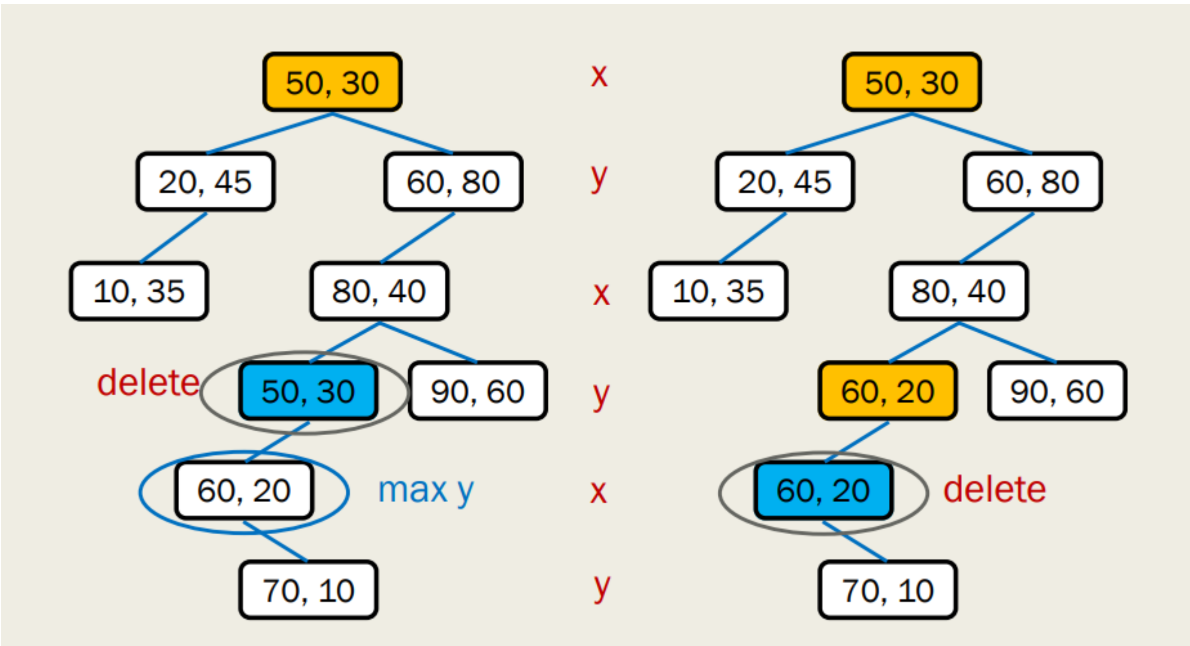
remove

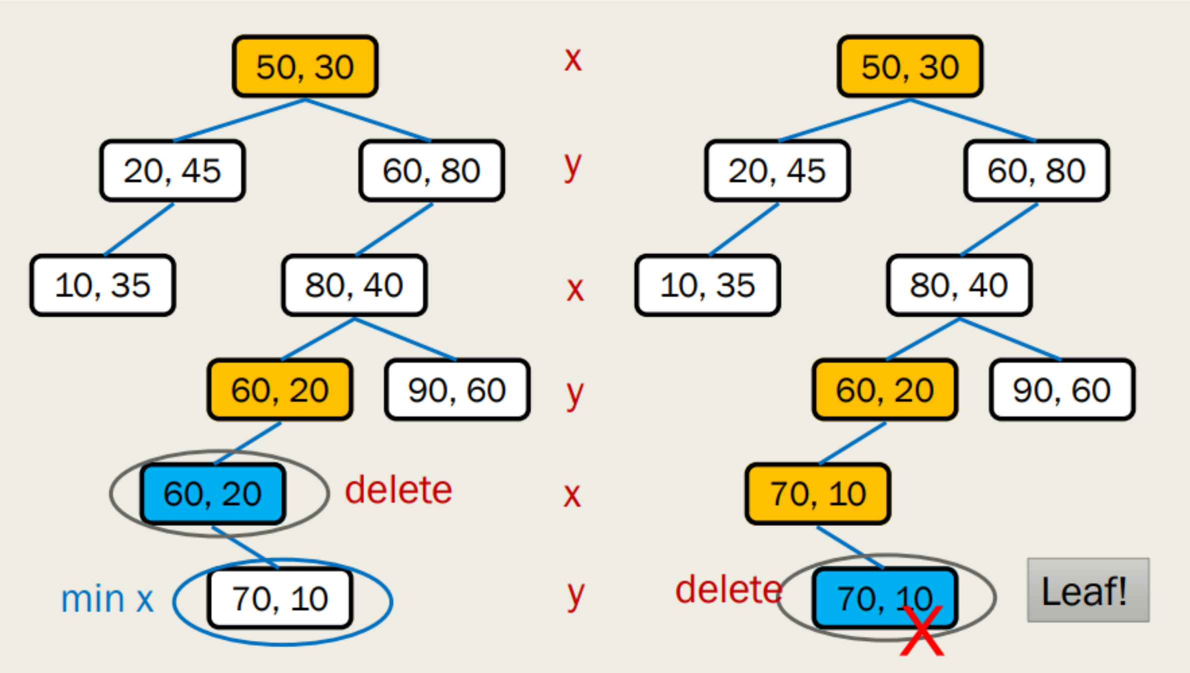
leaf: directly remove it

non-leaf:

- If the node R to be removed has **right** subtree, find the node M in **right** subtree with the **minimum** value of the current dimension
    - Replace the value of R with the value of M
    - Recurse on M until a leaf is reached. Then remove the leaf.
- Else, find the node M in **left** subtree with the **maximum** value of the current dimension. Then replace and recurse.







find minimum

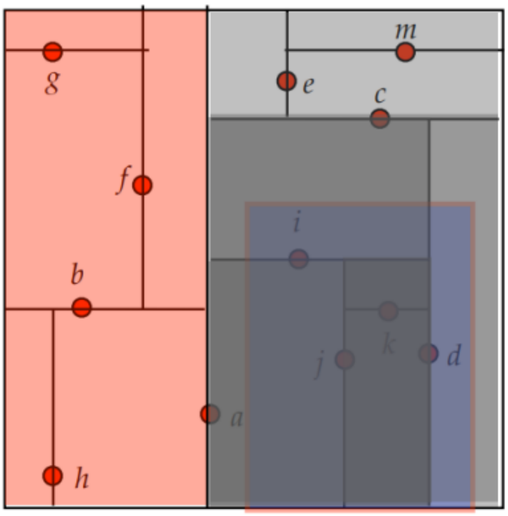
```
node *findMin(node *root, int dimCmp, int dim) {
// dimCmp: dimension for comparison
// dim: current dimension
    if(!root) return NULL;
    node *min = findMin(root->left, dimCmp, (dim+1)%numDim);
    if(dimCmp != dim) {
    // Then minimum might be in right subtree if the dimension doesn't match
        rightMin = findMin(root->right, dimCmp, (dim+1)%numDim);
        min = minNode(min, rightMin, dimCmp);
        // compare leftmin and rightmin
    }
    return minNode(min, root, dimCmp);
    // compare the minimum in subtrees and root, since root might not be in
comparison dimension
}
```

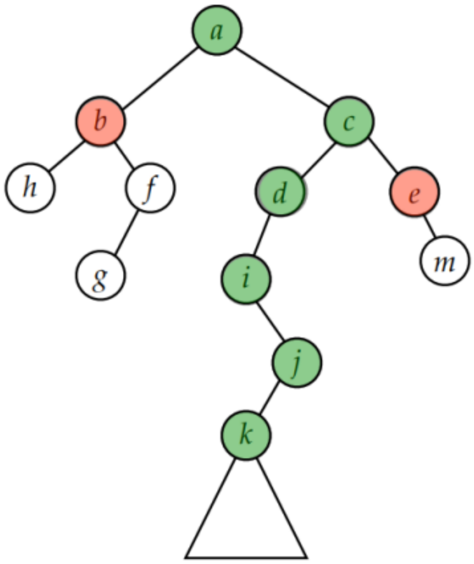
range search

```
void rangeSearch(
    node *root,
    int dim,
    Key searchRange[][2],
    Key treeRange[][2],List results
)
```

- `searchRange[][2]` holds two values (min, max) per dimension which define a hyper-cube
- `treeRange[][2]` holds lower bound and upper bound per dimension for the tree rooted at root.

Range Searching Example





If query box doesn't overlap bounding box, stop recursion

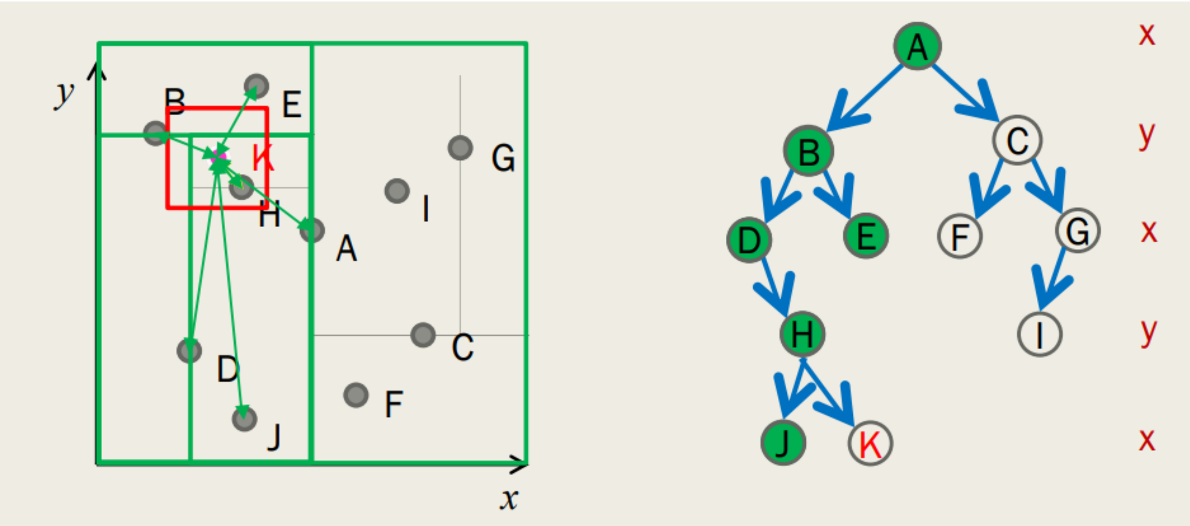
If bounding box is a subset of query box, report all the points in current subtree

If bounding box overlaps query box, recurse left and right.

nearest neighbor search

```
static void nearestNeighborSearch(node* currentNode, const Point2D& queryPoint,
int depth, Point2D& bestPoint, double& bestDist) {
        if (!currentNode) return;

        double d = distance(queryPoint, currentNode->key);
        if (d < bestDist) {
            bestDist = d;
            bestPoint = currentNode->key;
        }

        int axis = depth % 2;
        node* nextBranch = nullptr;
        node* oppositeBranch = nullptr;
        if ((axis == 0 && queryPoint.x < currentNode->key.x) || (axis == 1 &&
queryPoint.y < currentNode->key.y)) {
            nextBranch = currentNode->left_subtree;
            oppositeBranch = currentNode->right_subtree;
        } else {
            nextBranch = currentNode->right_subtree;
            oppositeBranch = currentNode->left_subtree;
        }

        // Search the side where the query point is
        nearestNeighborSearch(nextBranch, queryPoint, depth + 1, bestPoint,
bestDist);

        // Decide whether to search the opposite side
```

```
        if (oppositeBranch != nullptr && ((axis == 0 && abs(queryPoint.x -
currentNode->key.x) < bestDist) || (axis == 1 && abs(queryPoint.y - currentNode-
>key.y) < bestDist))) {
            nearestNeighborSearch(oppositeBranch, queryPoint, depth + 1,
bestPoint, bestDist);
        }
    }
```



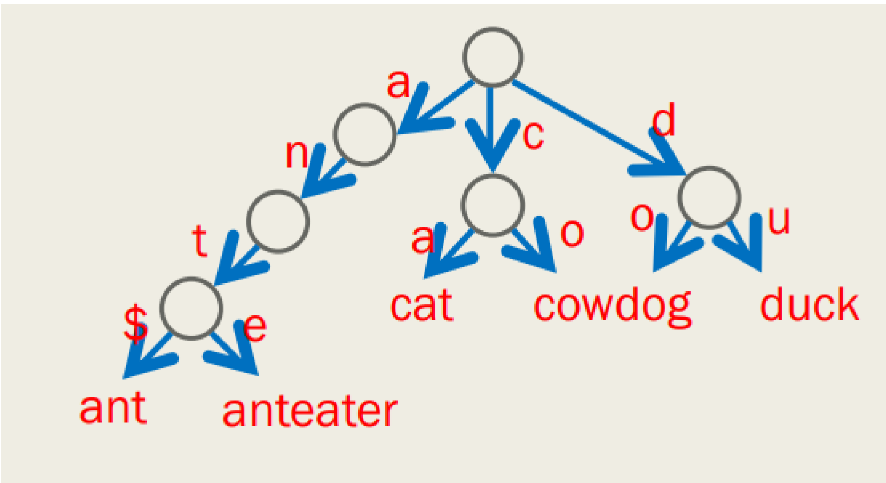
H and B here trigger the `oppositeBranch`

Time complexity

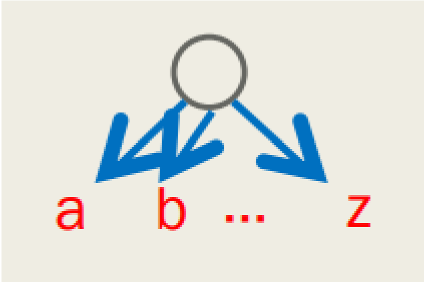
- insert: $O(\log n)$ (on average), $O(n)$ (worst).
- search: $O(\log n)$ (on average), $O(n)$ (worst)
- remove: $O(\log n)$ (on average), $O(n)$ (worst)

$O(\log n)$ is all you need.

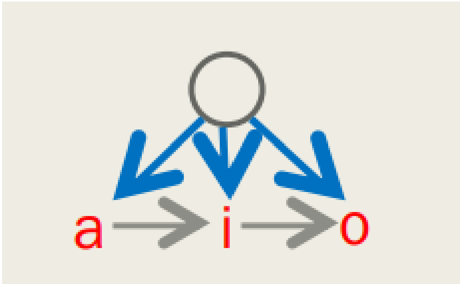
Trie / Prefix Tree



- We can keep an **array of pointers** in a node, which corresponds to all possible symbols in the alphabet.
  - For example, `ptr alphabet[26]` here



- or, a **linked list** of pointers to the child nodes, corresponding to a small fraction of the possible symbols in the alphabet.



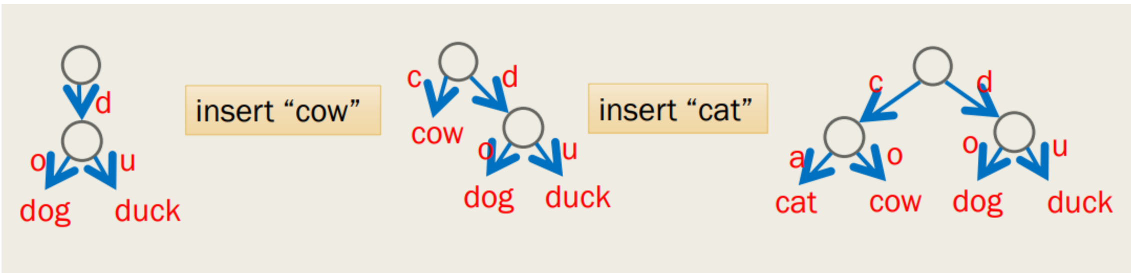
- prefix tree: Labels of edges on the path from the root to any leaf in the trie forms a prefix of a string in that leaf.
- We add a symbol to the alphabet to indicate the end of a string. For example, use "$" to **indicate the end**. This is important as sometimes a string may be a prefix of others. See `ant` in the figure above.

search

- Follow the search path, starting from the root.
- When there is **no branch**, return **false**.
- When the search leads to a leaf, further **compare** with the key at the leaf

insert

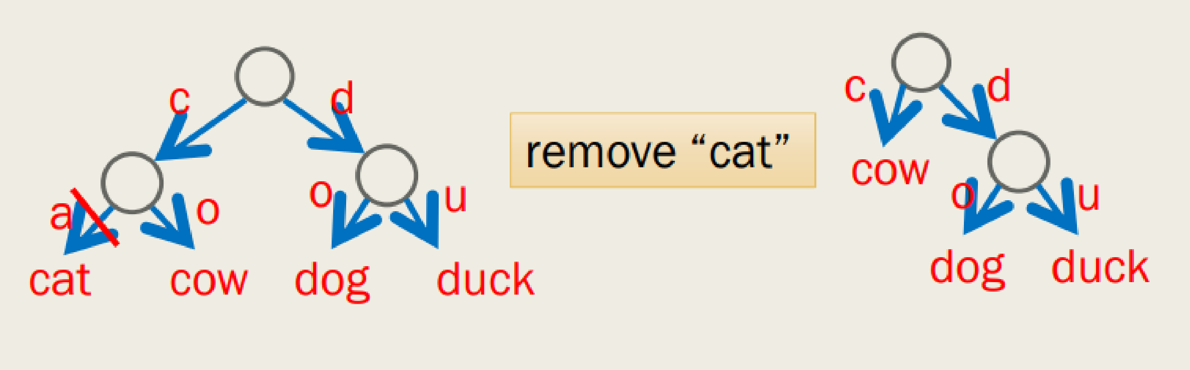
- Follow the search path, starting from the root.
- If a new branch is needed, add it.
- When the search leads to a leaf, a conflict occurs. We need to branch.
  - Use the next symbol in the key.
  - The originally-unique word must be moved to lower level



remove

- The key to be removed is always at the leaf
- After deleting the key, if the parent of that key now has only one child C, remove the parent node and move key C one level up

- If key C is the only child of its new parent, repeat the above procedure again.



Time complexity

For `insert` and `search`:

- $O(k)$ (worst), where k is the length of the string.
- not depend on the number of keys n.
- not depend on the number of keys .
  - For example, in the previous example, we can find the word "duck" with just "du".

Balanced Search Tree

- Height of a tree of n nodes = $O(\log n)$
- Balance condition can be maintained **efficiently**: $O(\log n)$ time to **rebalance** a tree.

AVL Tree

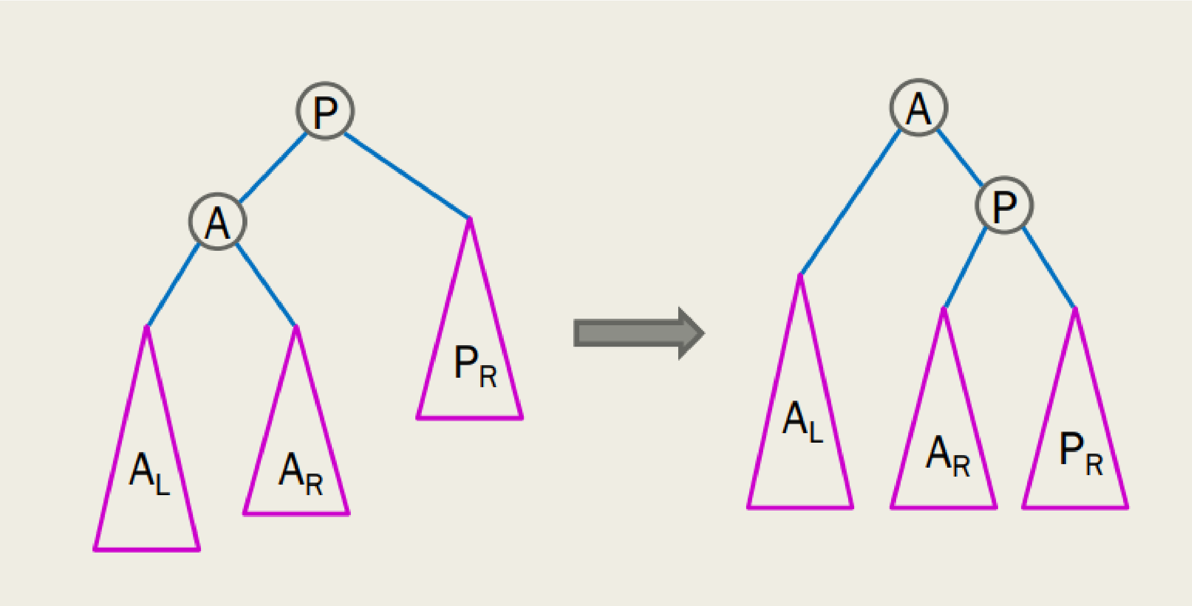
Balance condition

- An empty tree is AVL balanced
- A non-empty binary tree is AVL balanced if
  - Both its left and right subtrees are AVL balanced, and
  - The **height** of left and right subtrees differ by **at most 1**.

Re-Balance the Tree via Rotation

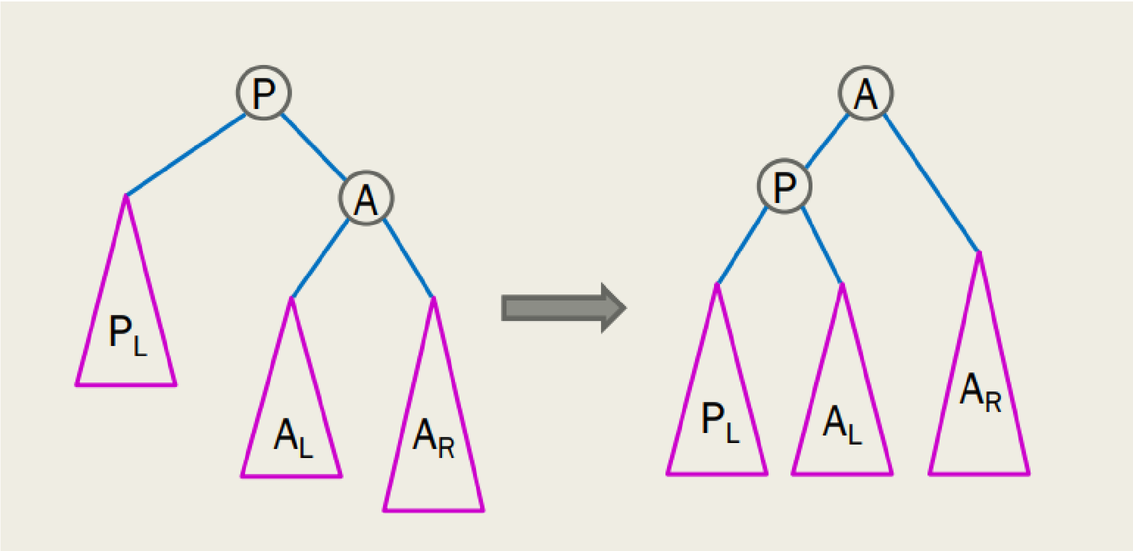
Right Rotation

1. The **right link** of the **left child** becomes the **left link** of the parent.
2. Parent becomes **right child** of the **old left child**.



Left Rotation

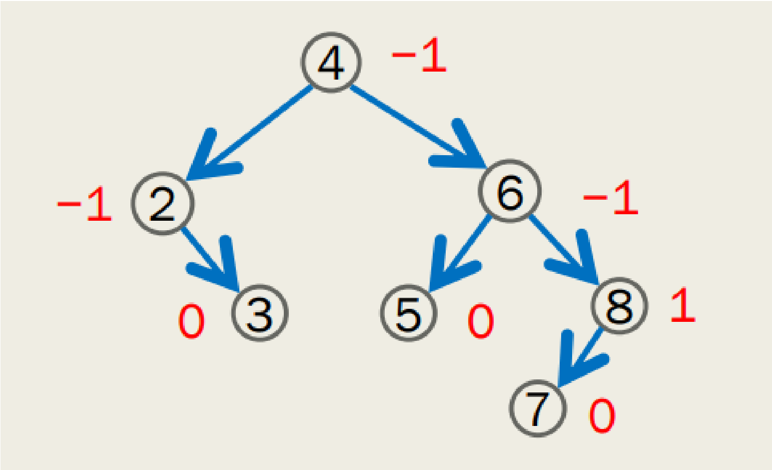
1. The **left link** of the **right child** becomes the **right link** of the parent.
2. Parent becomes **left child** of the **old right child**.



Balance Factor

$$B_T = h_l - h_r$$

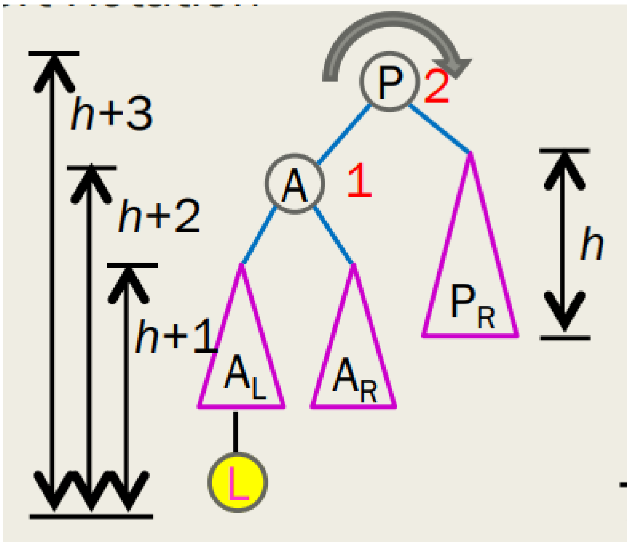
Rewrite the AVL tree's balance condition: for every node T in the tree $|B_T| \leq 1$.

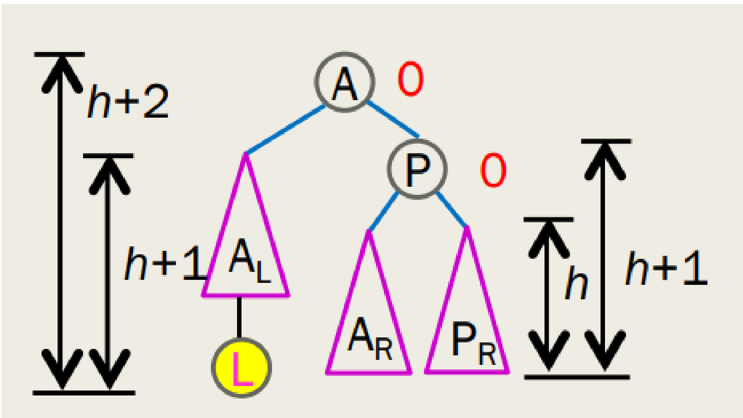


After Insertion

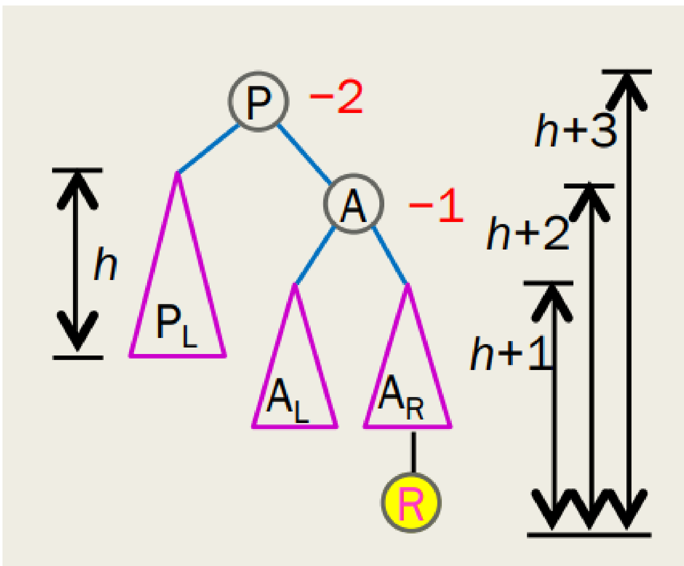
The heights of all the nodes along the access path, i.e., the **path from the root to that leaf** must be **recomputed** and the AVL balance condition must be checked.

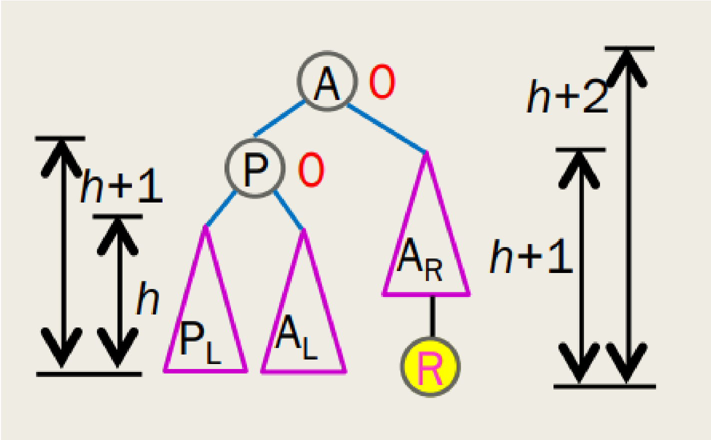
LL rotation



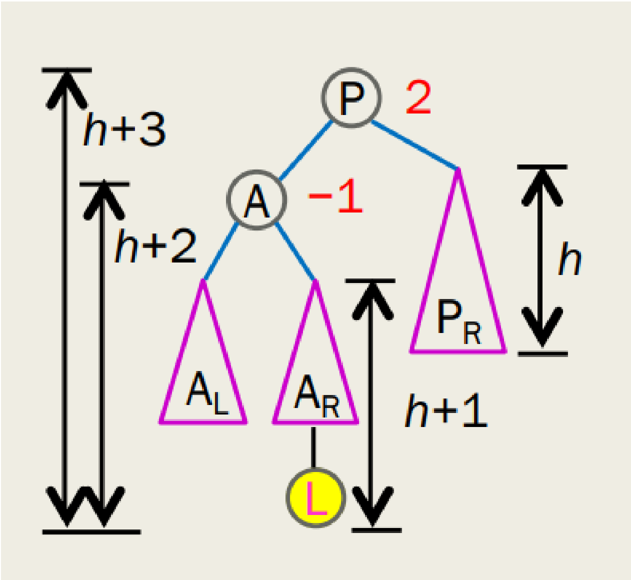


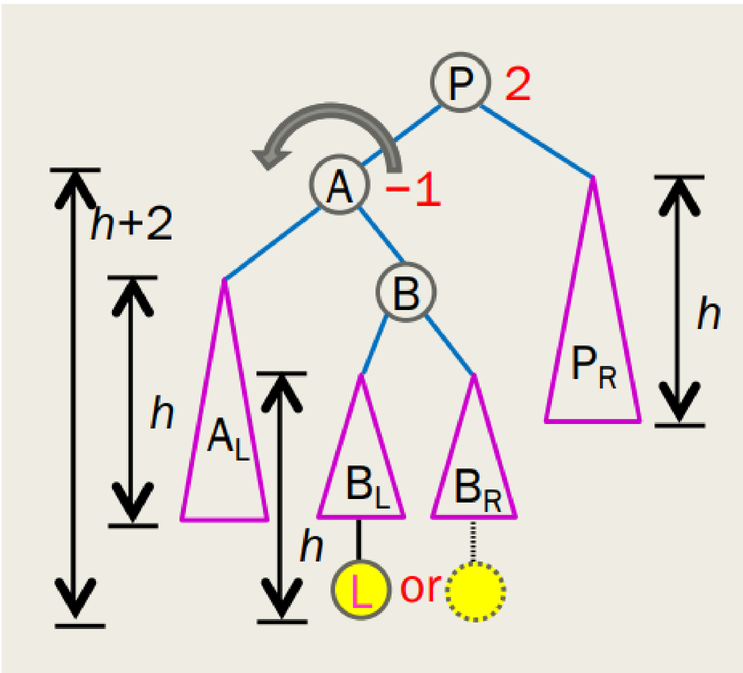
RR rotation

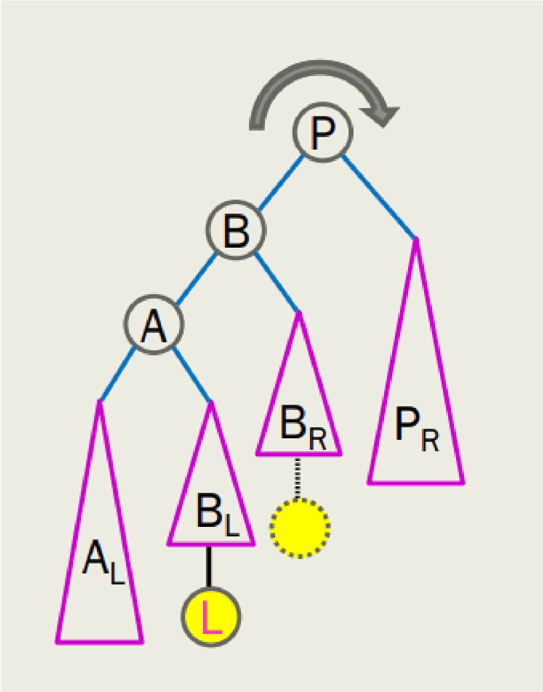


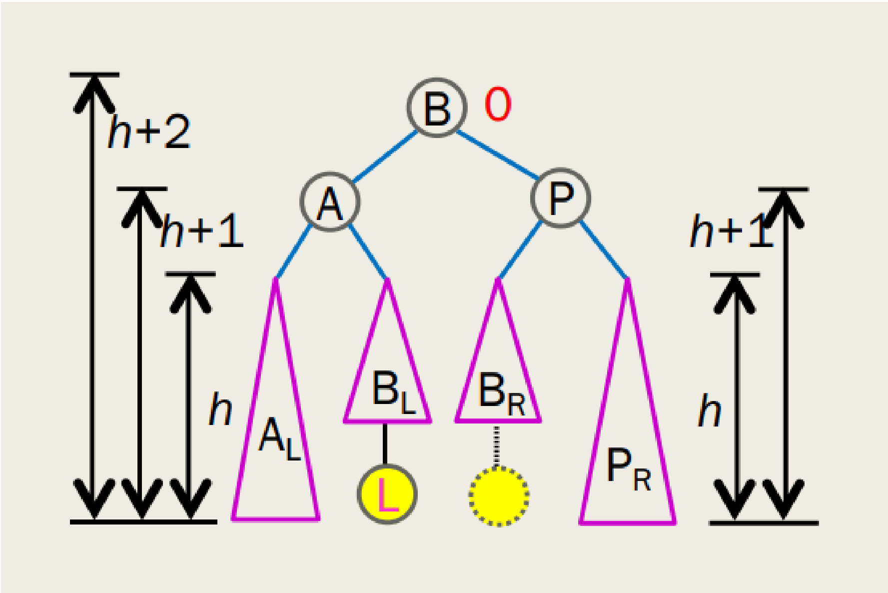

A
0
P
0
h+2
h+1
h
P_L
A_L
A_R
R
h+1

LR rotation

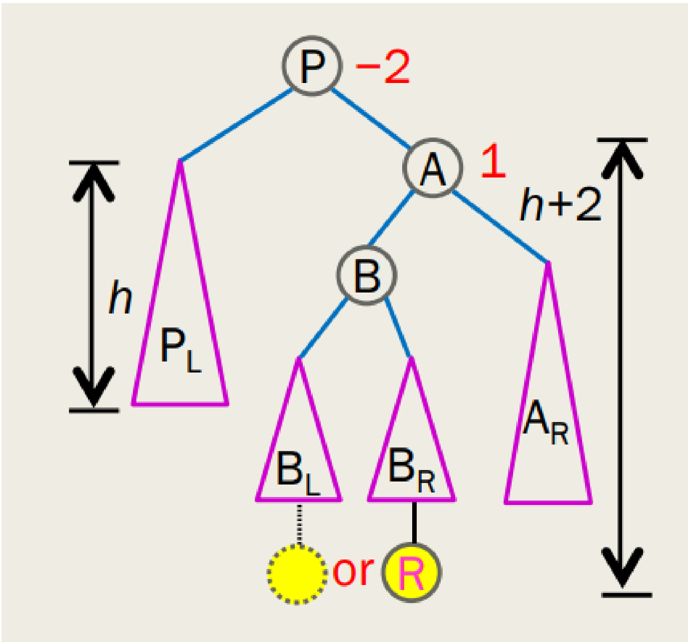

P
2
h+3
A
−1
h+2
h
P_R
A_L
A_R
h+1
L

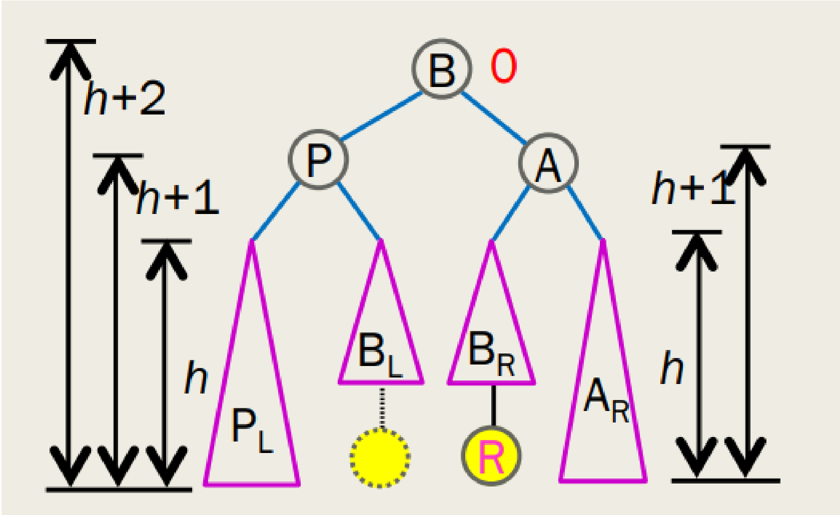

P
2
A
−1
h+2
B
h
P_R
h
A_L
B_L
B_R
h
L
or


P
B
A
B_R
P_R
B_L
A_L
L


h+2
h+1
h
B
0
A
P
h+1
h
B_L
B_R
A_L
P_R
L

RL rotation


P
−2
A
1
h+2
h
B
P_L
A_R
B_L
B_R
or
R



Summary

- We fix the **first unbalanced node** in the access path **from the leaf**.
- When an AVL tree becomes unbalanced after an insertion, **exactly one single or double rotation** is required to balance the tree.

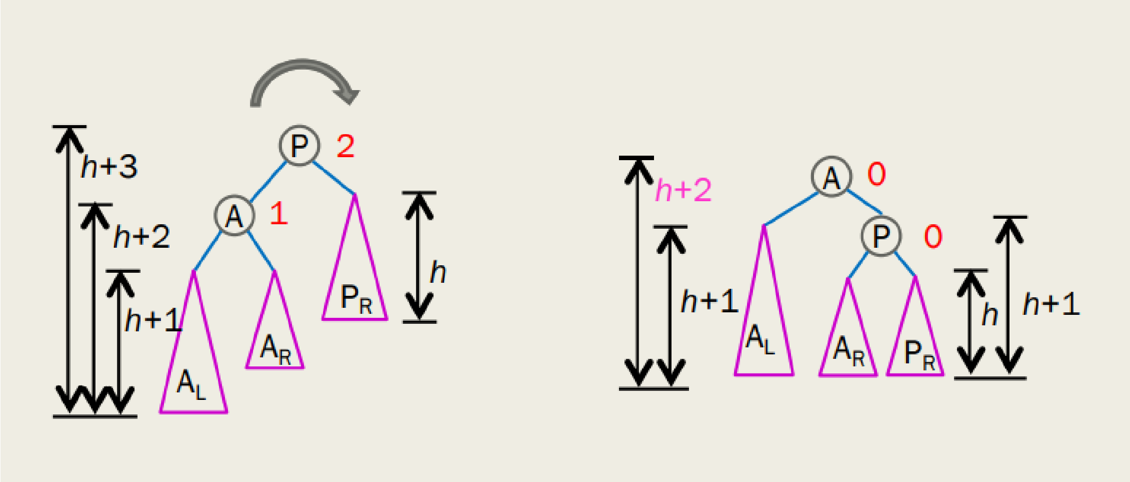
Solution	Situation
LL	Node P becomes unbalanced with a **positive** balance factor and the **left subtree** of the node also has a **positive** balance factor.
RR	Node P becomes unbalanced with a **negative** balance factor and the **right subtree** of the node also has a **negative** balance factor.
LR	Node P becomes unbalanced with a **positive** balance factor but the **left subtree** of the node has a **negative** balance factor.
RL	Node P becomes unbalanced with a **negative** balance factor but the **right subtree** of the node has a **positive** balance factor.

After Removal

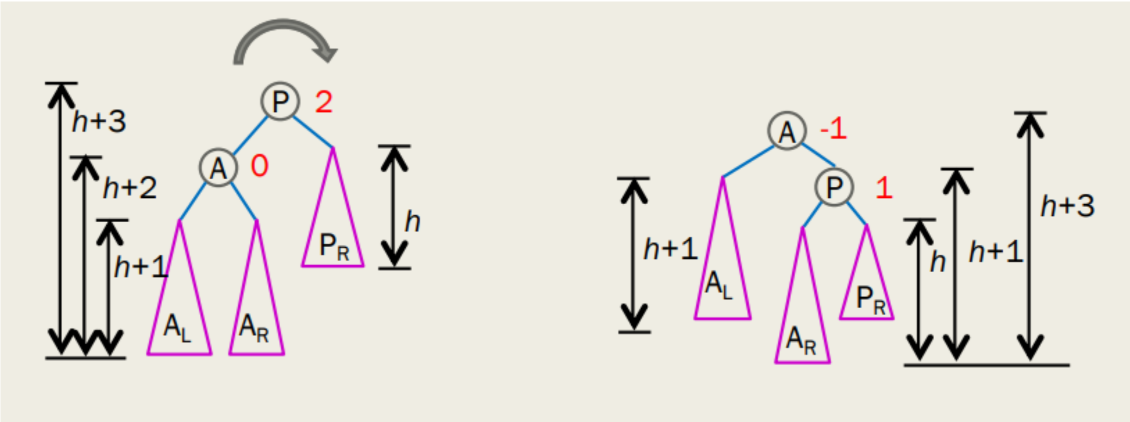
1. First remove node as with BST
2. Then update the balance factors of those ancestors in the access path and rebalance as needed. (almost the same as the operations above).

Difference from insertion: a single rotation might not completely fix all AVL imbalance (rebalancing may be applied to the ancestor).

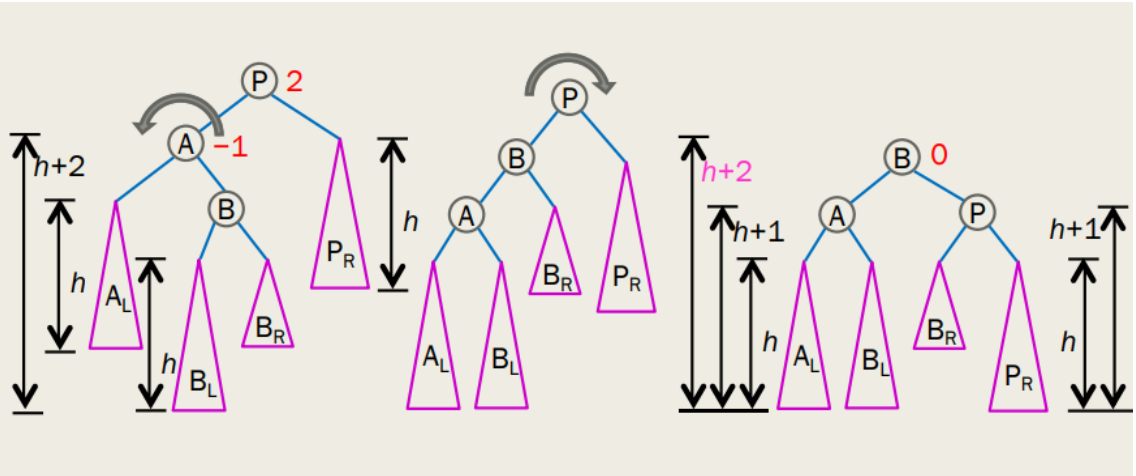
- case 1:



- case 2:



- case 3:



Time complexity

- search: $O(\log n)$
- insert: $O(\log n)$
- delete: $O(\log n)$

Red-Black Tree

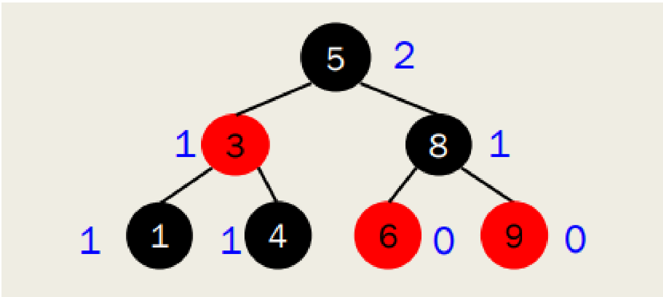
Properties

- a binary search tree

- Every node is either red or black.
- Root rule: The root is black.
- Red rule: Red node can only have black children.
- Path rule: Every path from a node x to NULL must have the **same** number of black nodes (including x itself).

Black height

Black height of a node x is the number of black nodes on the path from x to NULL, including x itself.



Implication of the Rules

- If a **red** node has **at least one child**, it must have **two children** and they must be **black**.
- If a **black** node has **only one child**, that child must be a **red leaf**.

Height Guarantee

- Claim: every red-black tree with nodes has height $\leq 2\log_2(n+1)$.
- Proof:
    1. In a binary tree with nodes, there is a root-NULL path with **at most** $\log_2(n+1)$ nodes.
    2. # black nodes on that path $\leq \log_2(n+1)$.
    3. By **path rule**: every root-NULL path has $\leq \log_2(n+1)$ **black** nodes.
    4. By **red rule**: every root-NULL path has $2 \leq \log_2(n+1)$ **total** nodes.

Insertion

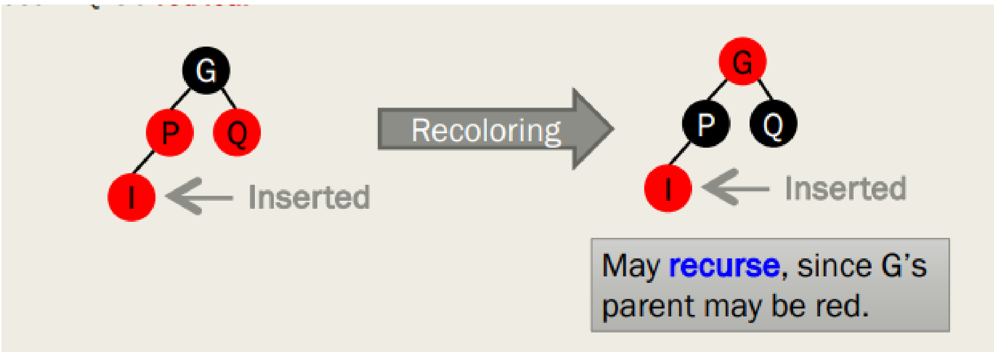
new node is always a **red leaf**.

- parent is black, done.
- parent is red, violate the **red rule**. Need to fix by recoloring/rotation.
    - moving the violation up the tree -> the root may become red-> set root to be black

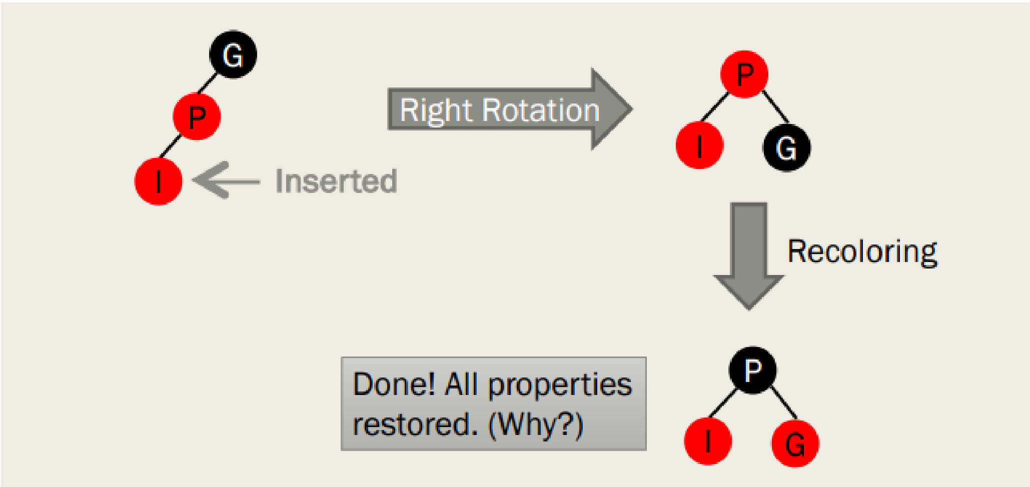
Following cases assume that the parent "P" is the **left child**. The **"right case"** is symmetric.

Violation at Leaf

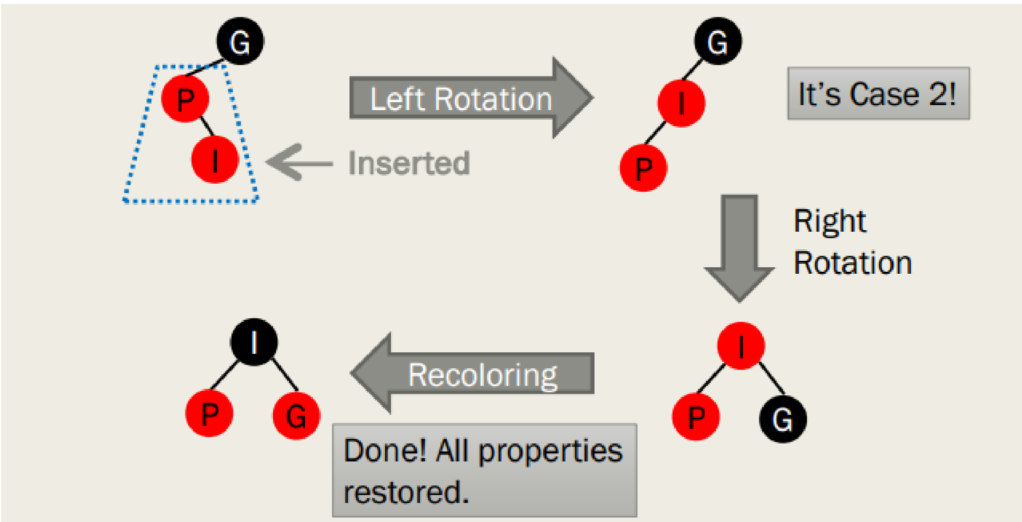
- case 1:



- case 2:

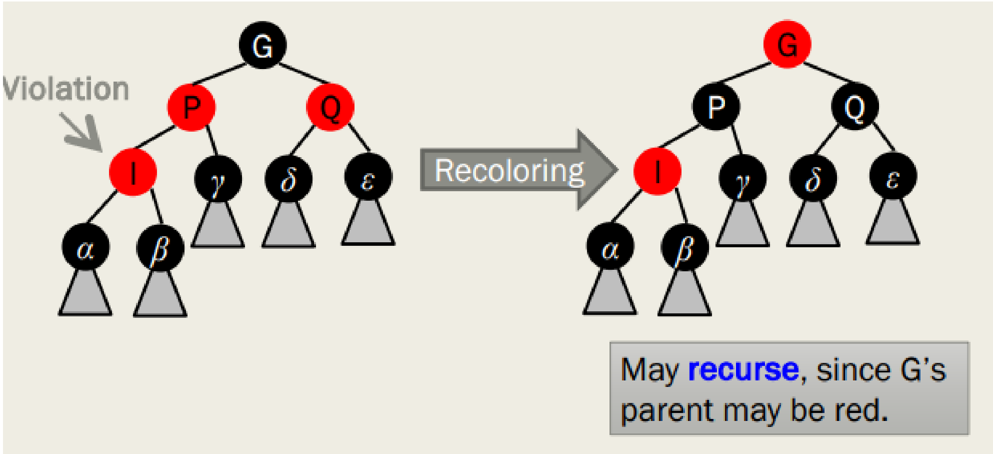


- case 3:

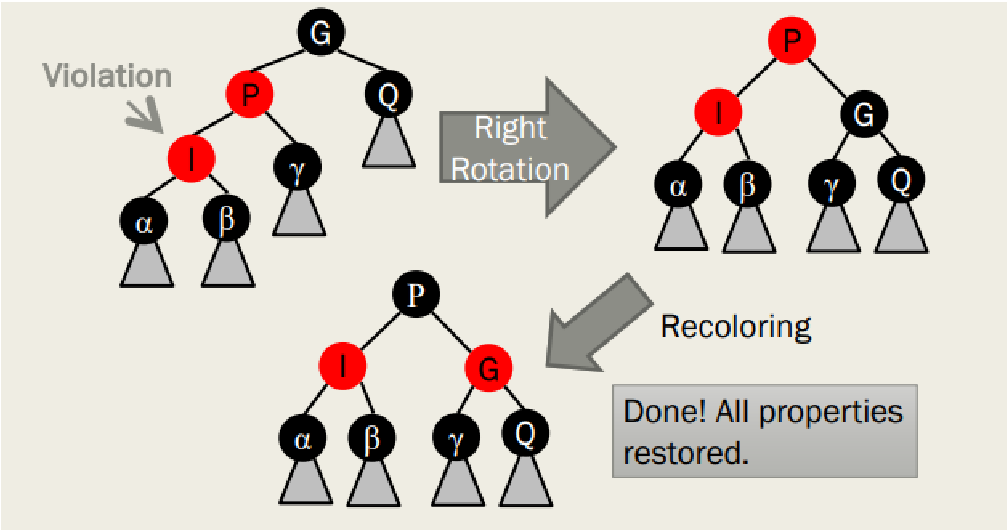


Violation at Internal Nodes

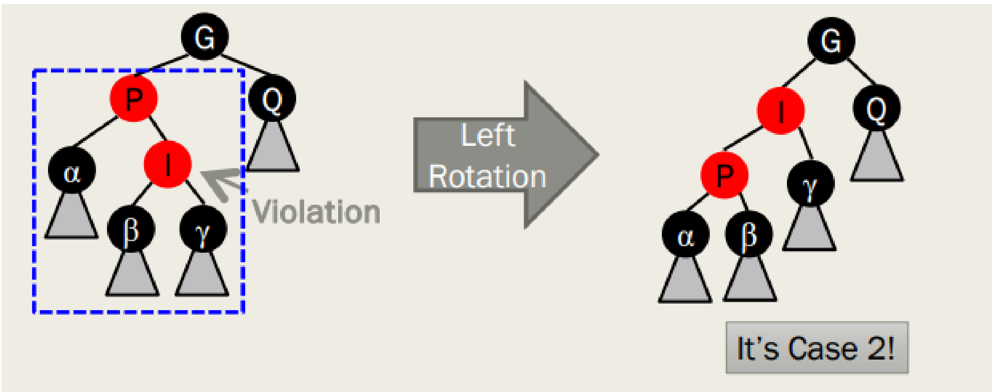
- case 1:



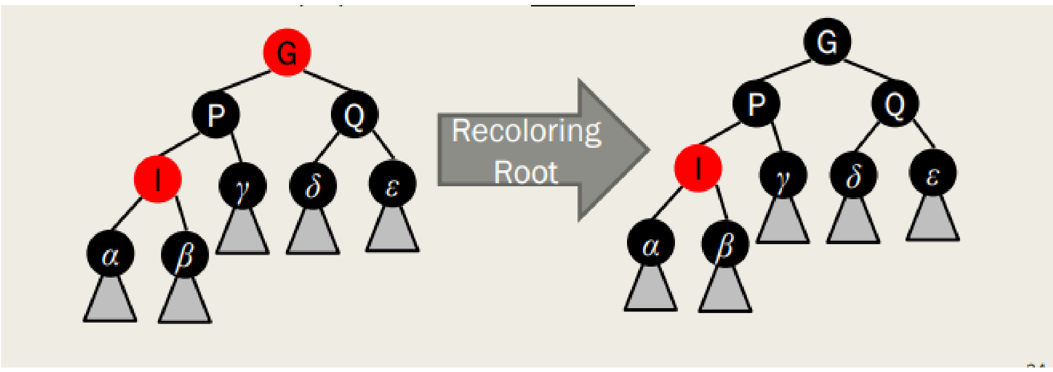
- case 2:



- case 3:



- final step:



Runtime Complexity

violations $\leq O(\log n)$.

Deletion

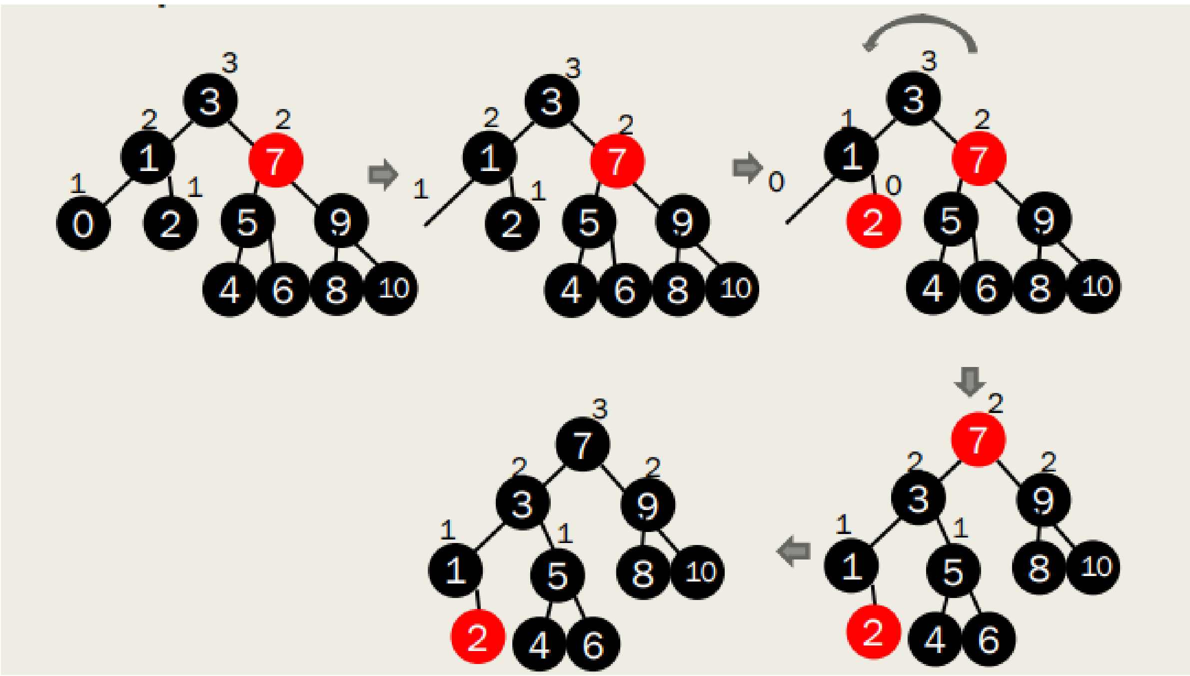
Deleting a red node

simple

Deleting a black node

complicated

What's wrong??



Compared Against AVL Tree

- Red-black tree is less balanced:
  - bad for search
  - good for insertion/deletion
- Example
  - AVL tree for database (lots of lookups, fewer modifications)
  - Red-black tree for stock market transactions (lots of modifications)

Time complexity

- search: $O(\log n)$
- insert: $O(\log n)$
- delete: $O(\log n)$