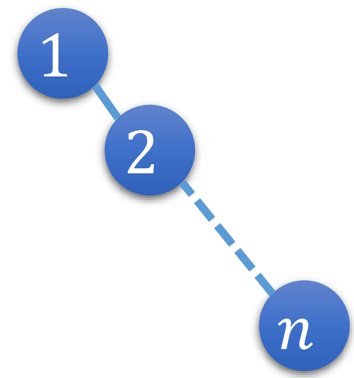
EECS 204002
Data Structures 資料結構
Prof. REN-SONG TSAY 蔡仁松 教授
NTHU

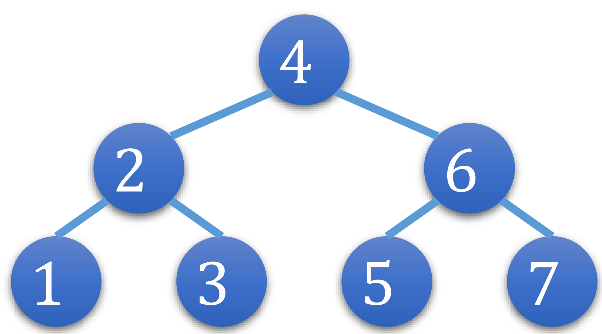
CH. 10 EFFICIENT BINARY SEARCH TREES

Binary Search Trees

- All BST operations complexity = $O(h)$
  - h = height of the BST
- Worst case: $h = n$
  - Ex: insert keys $1, 2, \dots, n$

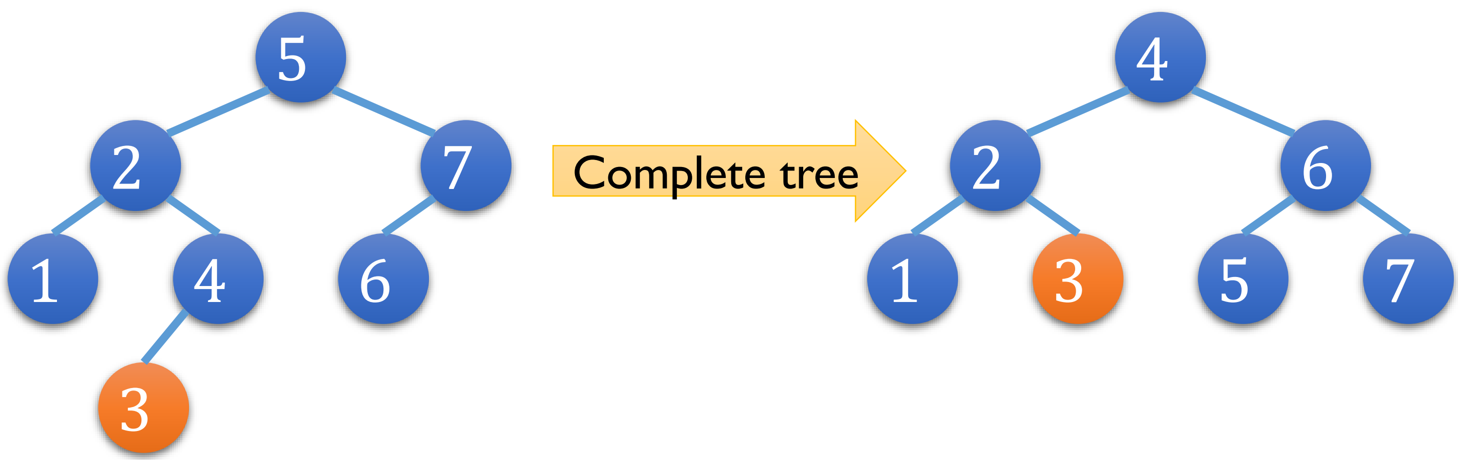


- Best case: $h = \log n$
  - Ex: insert keys $4, 2, 6, 1, 3, 5, 7$



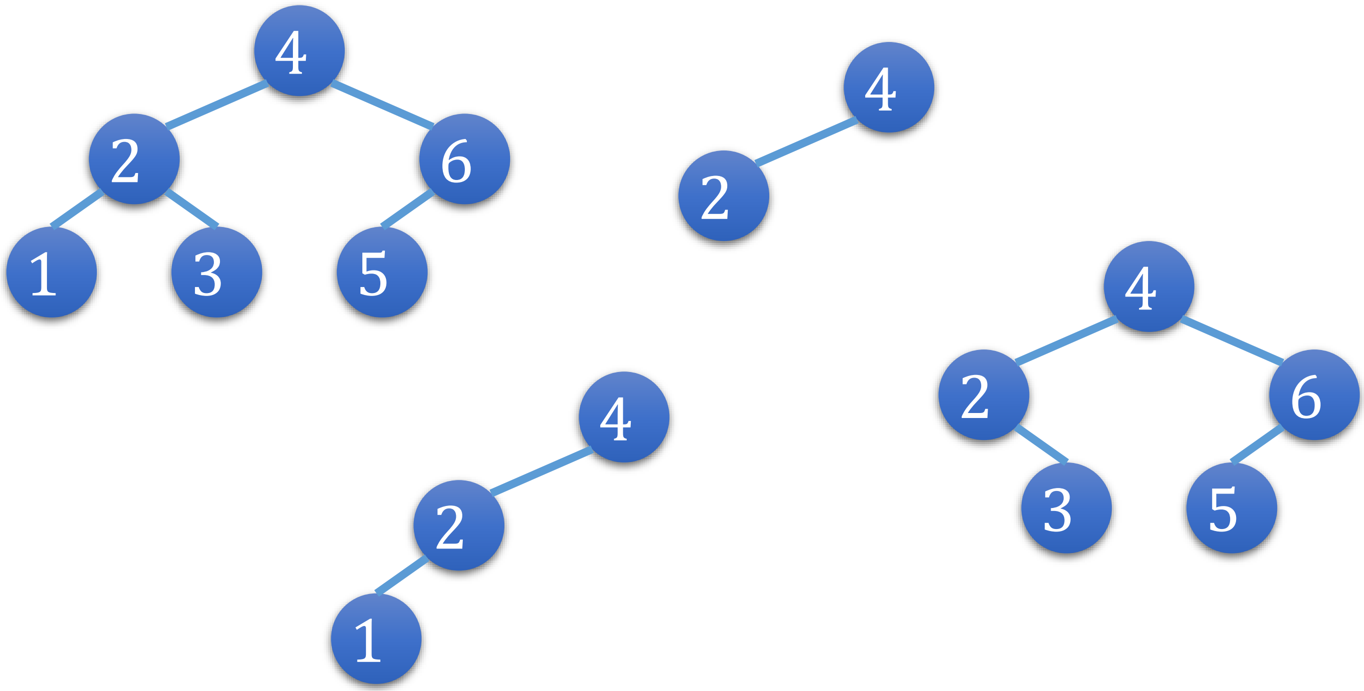
What is the Best Case?

- If BST retains a complete tree
- But expensive to retain a complete tree
  - Ex: insert 3 into the tree on the left



A Compromise

- Fairly, but not perfectly, balanced tree
  - Depths of the left and right subtrees $\Rightarrow \pm 1$
- Which one is “balanced”?



How to Keep a Balanced BST ?

- AVL Trees
- Red-black Trees (self-study)
- Splay trees (self-study)
  - Self adjusting trees
- B-trees
  - Multiway search trees

10.2

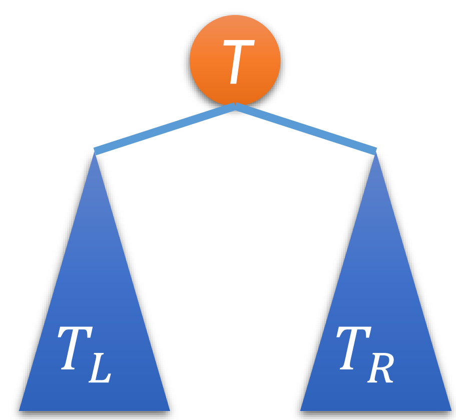
AVL Trees

Height Balanced Trees

- An empty tree is height balanced.
- If T is a non-empty binary tree with T_L and T_R
  - As its left and right subtrees respectively
- Balance factor

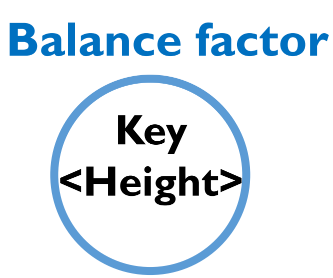
$$bf(\boldsymbol{T}) = height(\boldsymbol{T}_{\boldsymbol{L}}) - height(\boldsymbol{T}_{\boldsymbol{R}})$$

- $\boldsymbol{T}$ is height balanced iff
  1) $\boldsymbol{T}_{\boldsymbol{L}}$ and $\boldsymbol{T}_{\boldsymbol{R}}$ are height balanced.
  2) $|bf(\boldsymbol{T})| \leq 1$

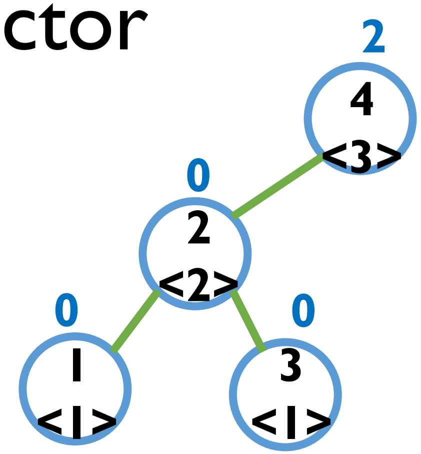


Definition of AVL Trees

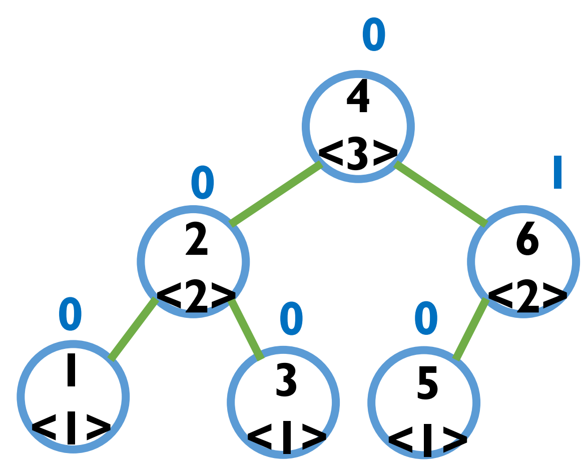
- AVL tree is a *height-balanced* binary search tree. (**A**delson, **V**elskii, **L**andis)
- Each node in an AVL tree stores the current node height for calculating the balance factor



Representation



An unbalanced BST



An AVL Tree

10.2

Prove $N \geq 2^{h/2}$

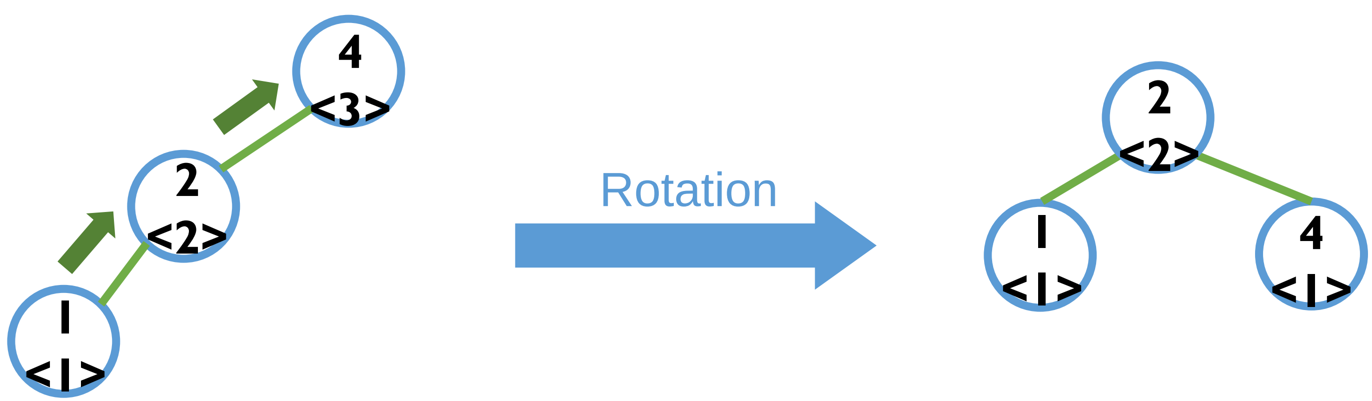
for an N-node AVL tree of height h

- Idea: to prove by showing $N \geq N(h) \geq 2^{h/2}$, *where* $N(h)$ = minimum # of nodes of an h−height AVL tree.
- Induction
  - $N(1) = 1, N(2) = 2$
  - $N(h) = N(h-1) + N(h-2) + 1$
    - $N(h) \geq 2 \cdot N(h-2)$
- Solution
  - $N(h) \geq 2N(h-2) \geq 2(2N(h-4))$
    $\geq 2^i N(h-2i) \approx 2^{h/2}$
  - Or $h = O(\log_2 N)$

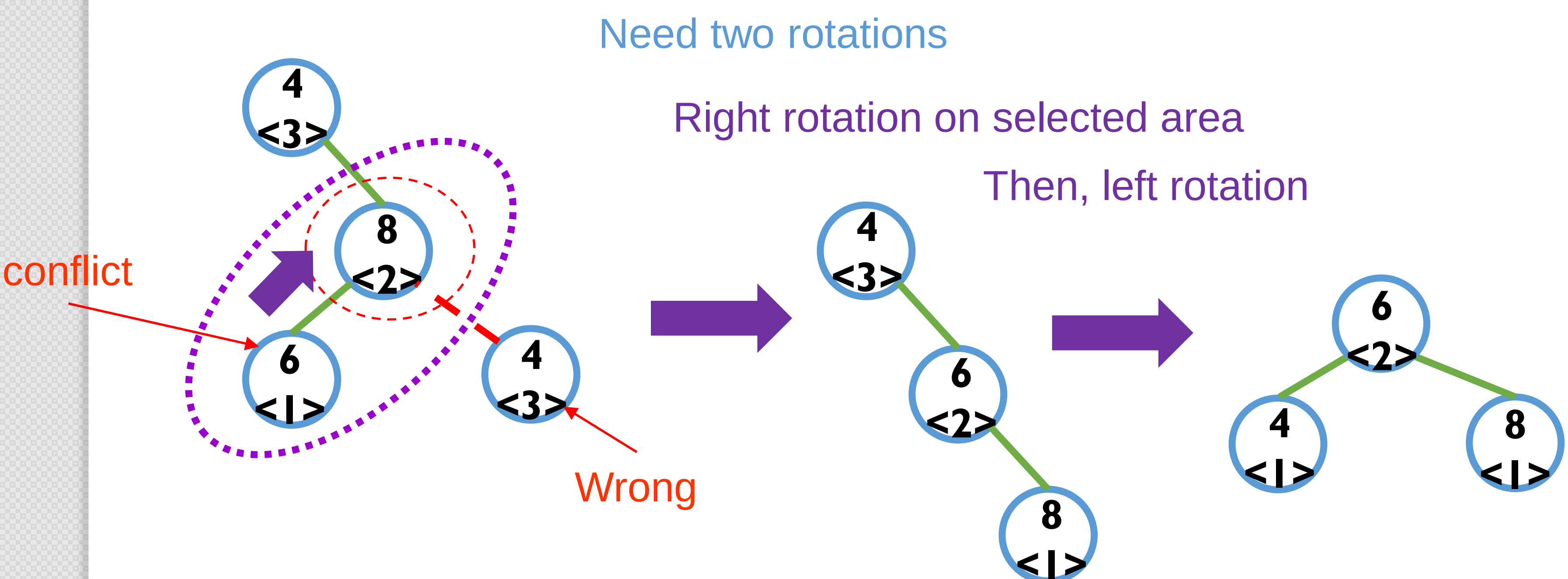
10.2

Rebalancing Process

- BST insertion/deletion operation may cause nodes with balance factor > 1 or < -1.
- Rebalancing process
  - Update the heights (balance factors) from the inserted/deleted node up to the root.
  - Fix unbalanced situations by rotations.



Rebalancing Operations

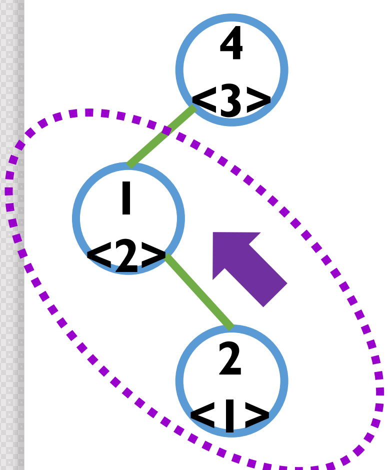


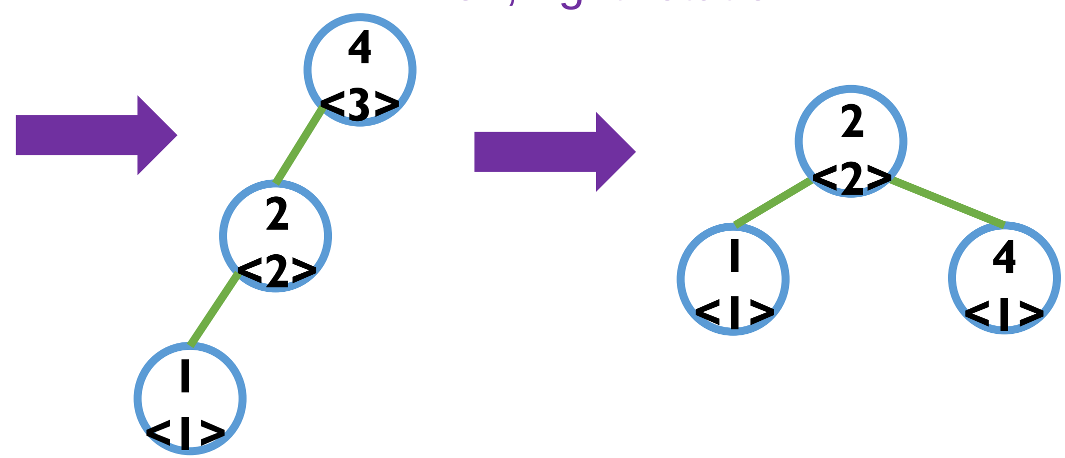
Rebalancing Operations

Need two rotations

Left rotation on selected area

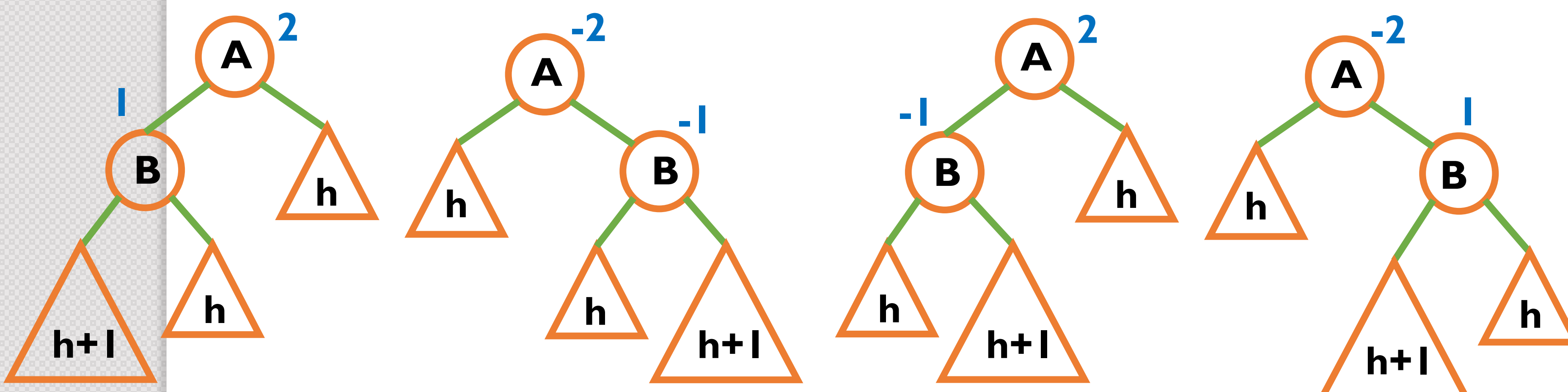
Then, right rotation





4 Unbalanced Situations

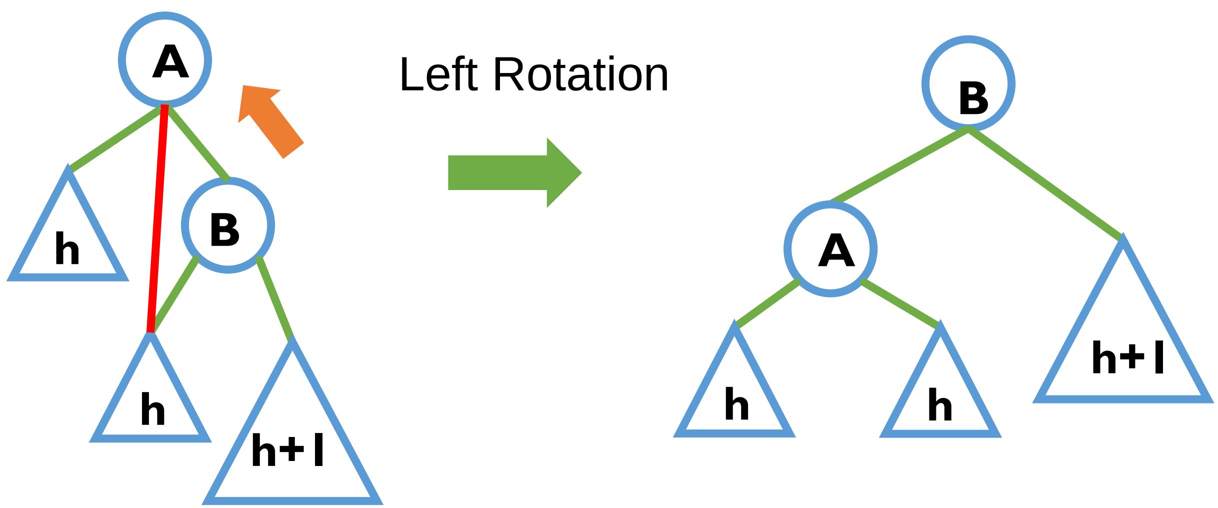
- 2 outside cases: require single rotation (LL, RR)
- 2 inside cases: require double rotation (LR, RL)



2 outside cases　　　　2 inside cases

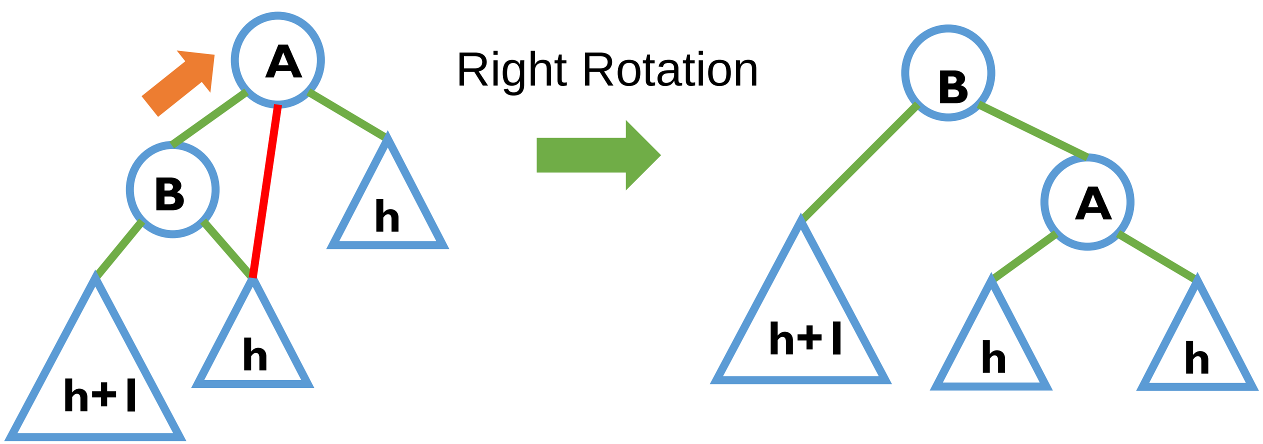
Outside RR Case - Left Rotation

- The new node is inserted in the right subtree of the right subtree of A



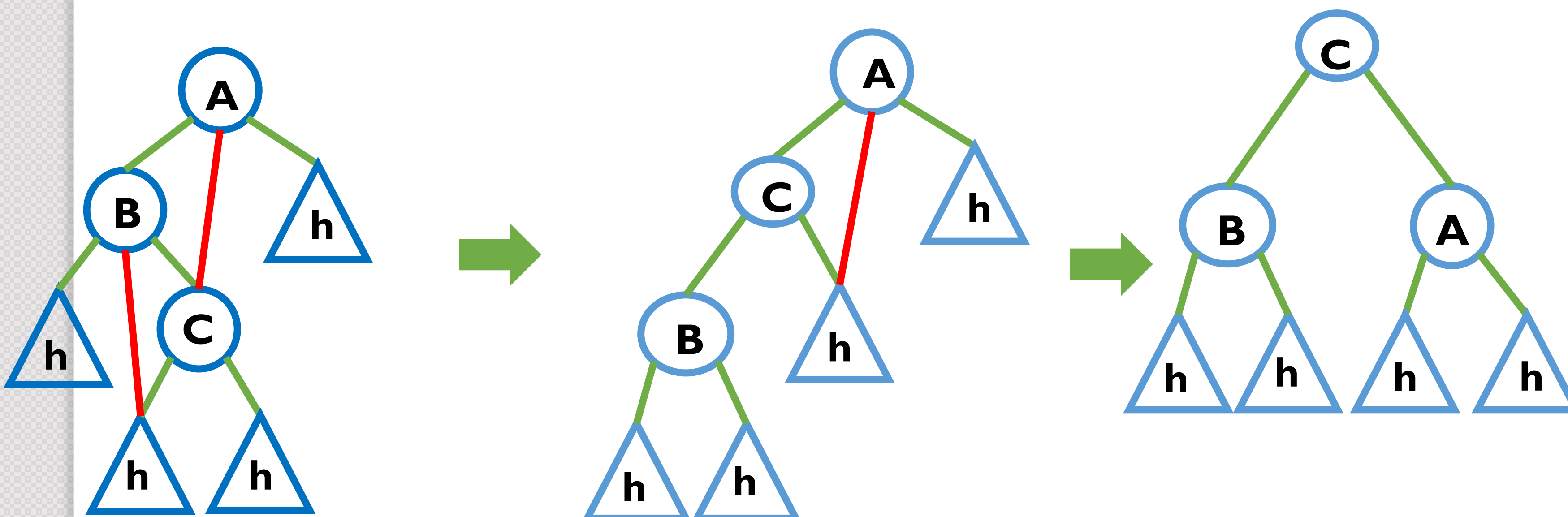
Outside LL Case - Right Rotation

- The new node is inserted in the left subtree of the left subtree of A



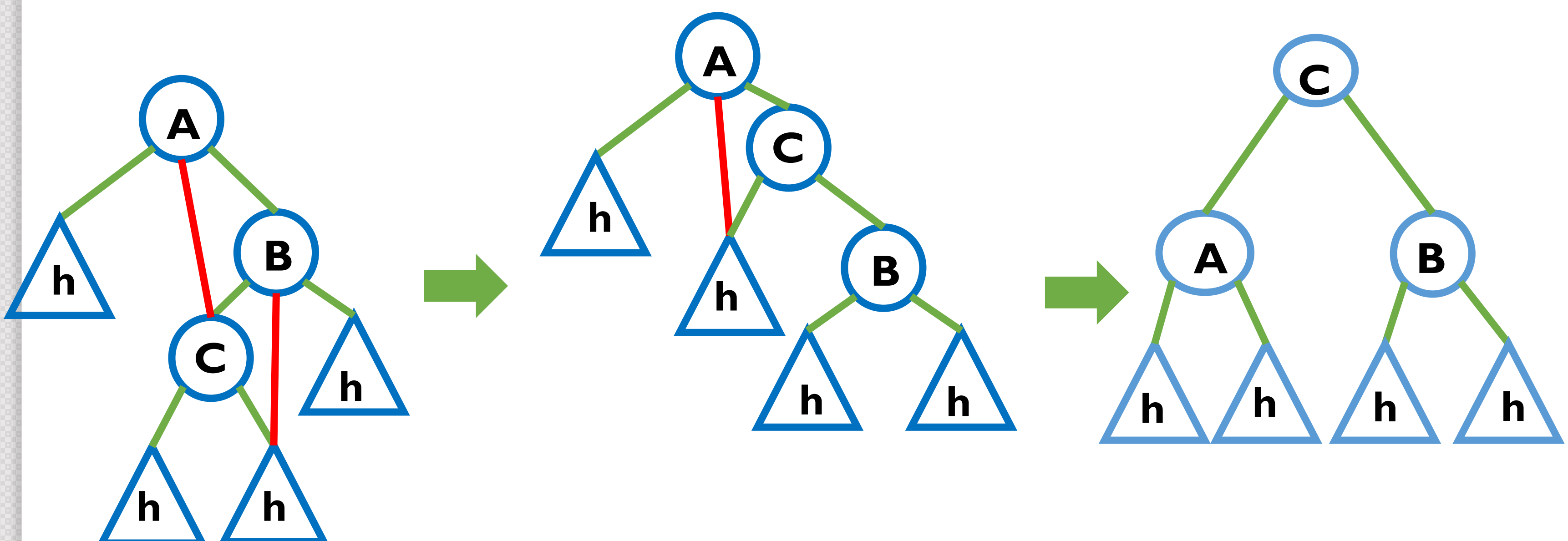
Inside RL Case - LR Rotation

- The new node is inserted in the right subtree of the left subtree of A
- Left rotation + Right rotation



Inside LR Case - RL Rotation

- The new node is inserted in the left subtree of the right subtree of A
- Right rotation + left rotation



10.2

ADT: AVL Tree

```
template < class T > class AVLTree;

template < class T >
Class TreeNode {
friend class AVLTree <T>;
private:
    T data;
    int height;
    void updateHeight();
    int bf();
    TreeNode<T>* left, right;
};

template <class T>
Class AVLTree{
public:
       // Constructor
       AVLTree(void) {root=NULL;}

       // Tree operations here…

private:
      TreeNode<T> *root;
};
```

10.2

AVL Tree Insert/Delete

```
template < class T >
TreeNode<T>* AVLTree<T>::insert(TreeNode<T> *node, T data)
{
    // BST Insert
    // ...

    // rebalance from node to root
    node->updateHeight();
    return rebalance( node );
}

template < class T >
TreeNode<T>* AVLTree<T>::delete(TreeNode<T> *node, T data)
{
    // BST Delete
    // ...

    // rebalance from node to root
    node->updateHeight();
    return rebalance( node );
}
```

10.2

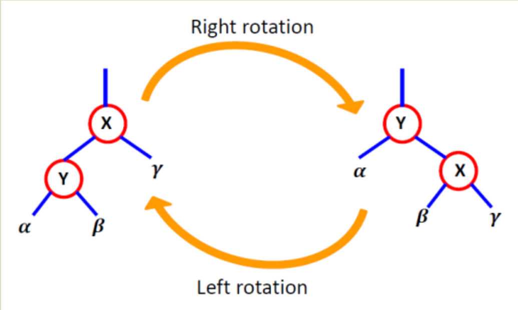
AVL Tree Rebalance

```
template < class T >
TreeNode<T>* AVLTree<T>::rebalance(TreeNode<T> *node){
    //    LL Case
    if ( node->bf()>1 && node->left->bf()>=0 ){
        return rightRotate( node );
    }
    //    RR Case
    if ( node->bf()<-1 && node->right->bf()<=0 ){
        return leftRotate( node );
    }
    //    RL Case
    if ( node->bf()>1 && node->left->bf()<0 ){
        node->left = leftRotate( node->left );
        return rightRotate( node );
    }
    //    LR Case
    if ( node->bf()<-1 && node->right->bf()>0 ){
        node->right = rightRotate( node->right );
        return leftRotate( node );
    }
}
```

10.2

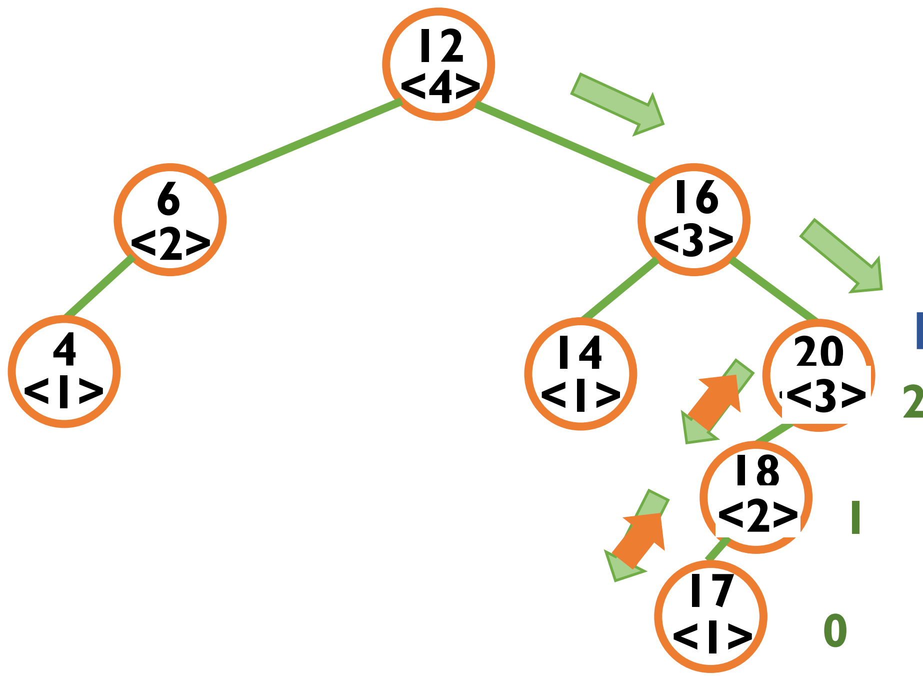
AVL Tree Left/Right Rotation

```
template < class T >
TreeNode<T>* AVLTree<T>::leftRotate(TreeNode<T> *node)
{
    TreeNode<T>* node_r = node->right;
    TreeNode<T>* node_rl = node_r->left;
    node_r->left = node;
    node->right = node_rl;
    node->UpdateHeight();
    node_r->UpdateHeight();
    return node_r;
}
```




```
template < class T >
TreeNode<T>* AVLTree<T>::rightRotate(TreeNode<T> *node)
{
    TreeNode<T>* node_l = node->left;
    TreeNode<T>* node_lr = node_l->right;
    node_l->right = node;
    node->left = node_lr;
    node->UpdateHeight();
    node_l->UpdateHeight();
    return node_l;
}
```

AVL Tree: Example: Insert 17



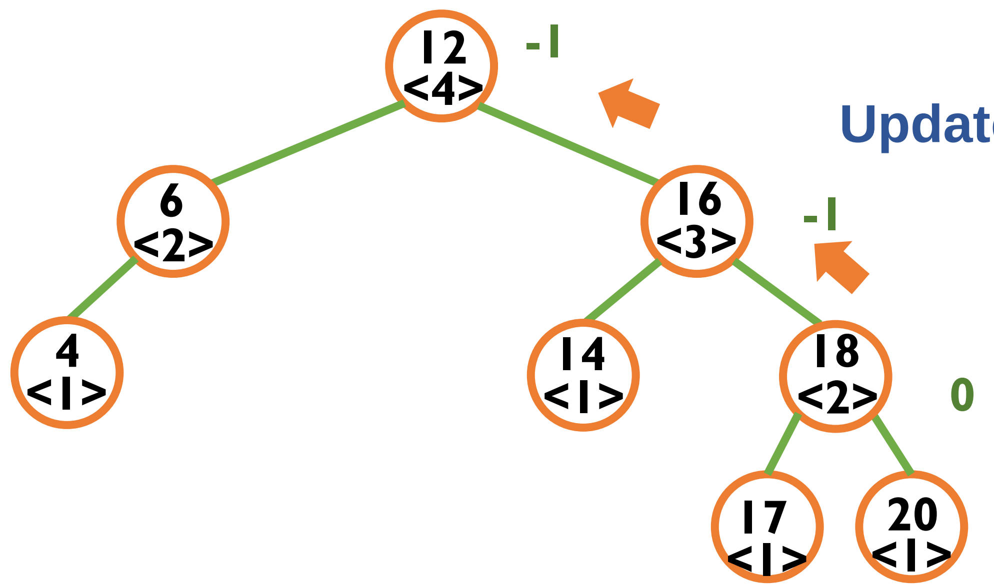
LL Case Rebalancing
Right rotation!

Update height and bf

AVL Tree: Example: Insert 17

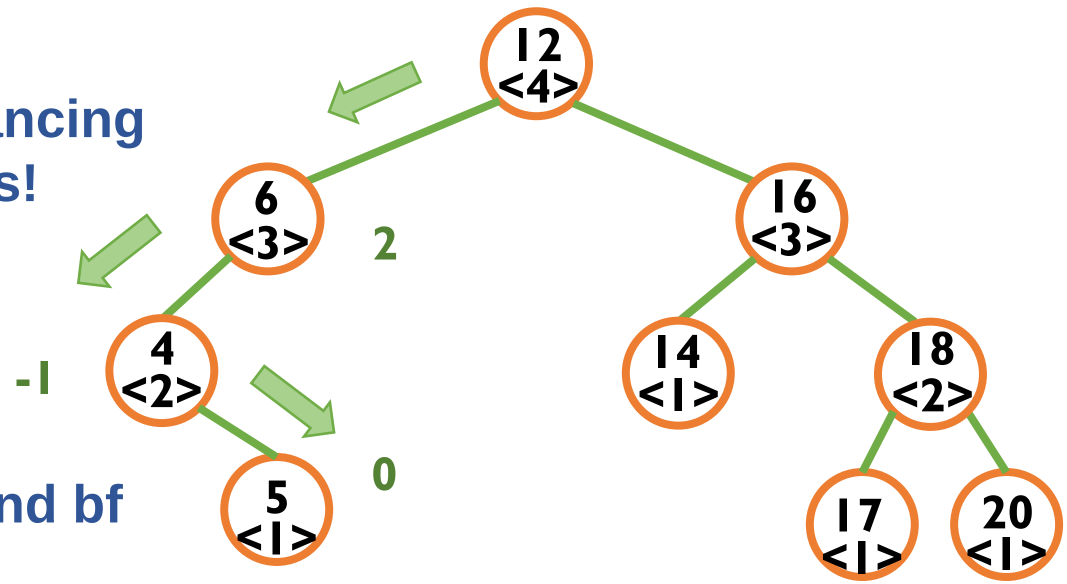
Finish checking

Update height and bf



AVL Tree: Example: Insert 5

RL Case Rebalancing
LR rotations!

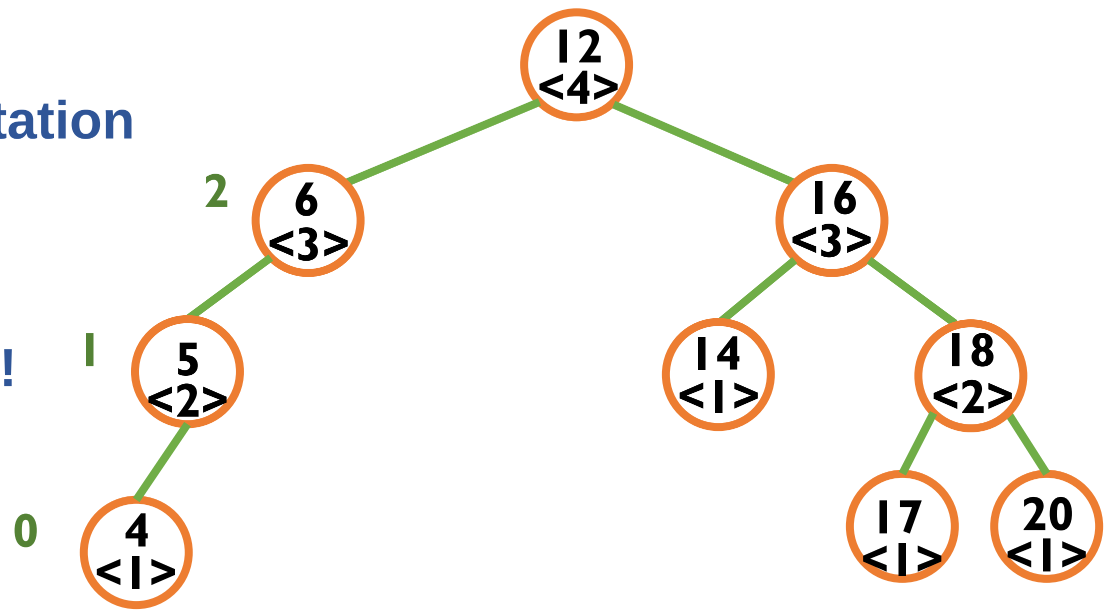


Update height and bf

AVL Tree: Example: Insert 5

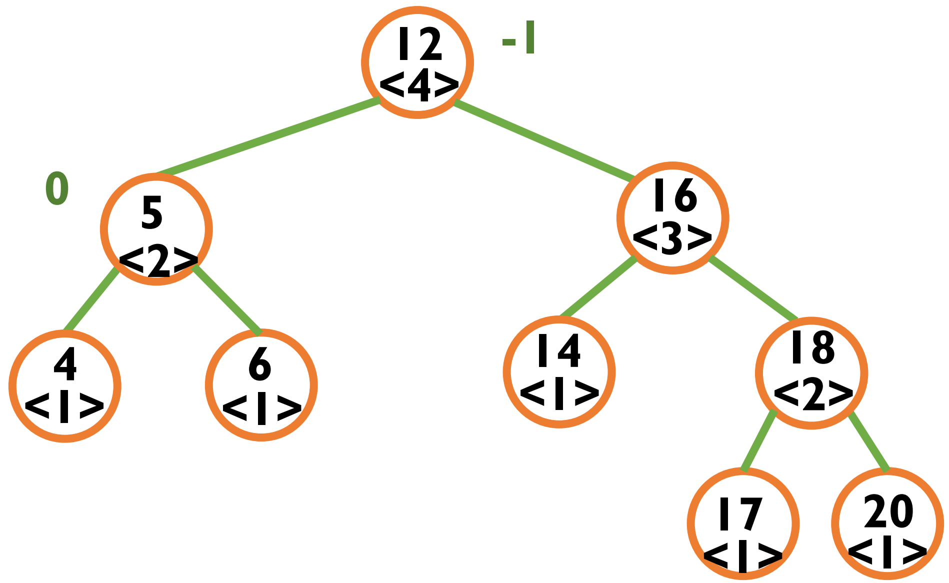
After left rotation

continue rebalancing, right rotation!

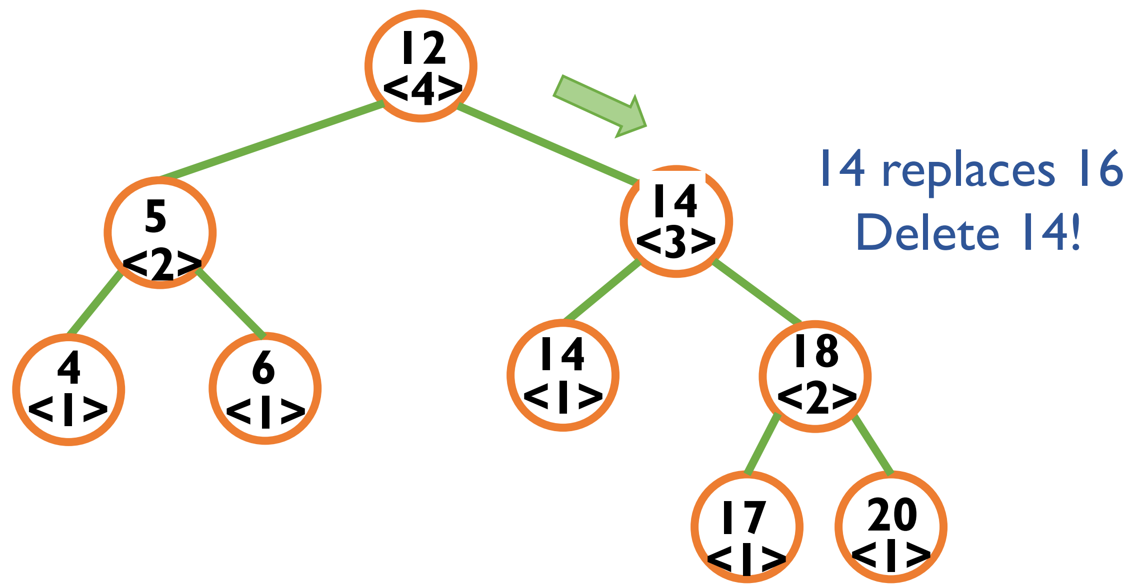


AVL Tree: Example: Insert 5

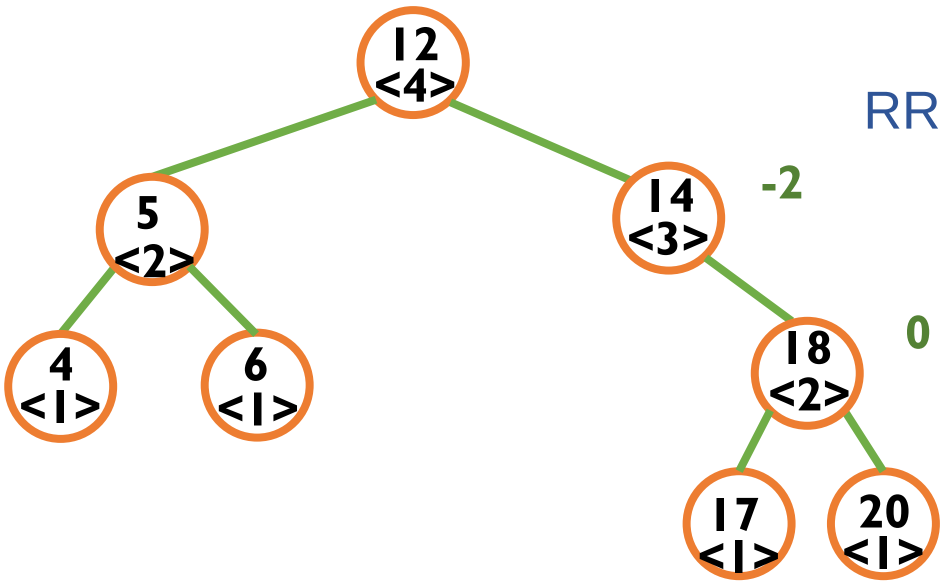
Finish checking



AVL Tree: Example: Delete 16

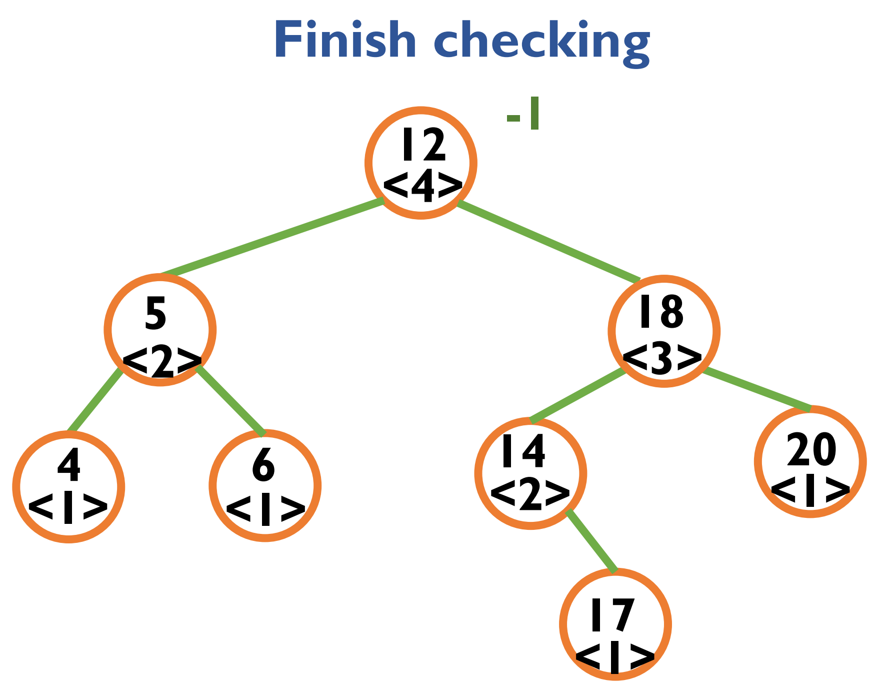


AVL Tree: Example: Delete 16



RR Case Rebalancing
Left rotation!

AVL Tree: Example: Delete 16



EECS 204002
Data Structures 資料結構
Prof. REN-SONG TSAY 蔡仁松 教授
NTHU

CH. 11 MULTIWAY SEARCH TREES

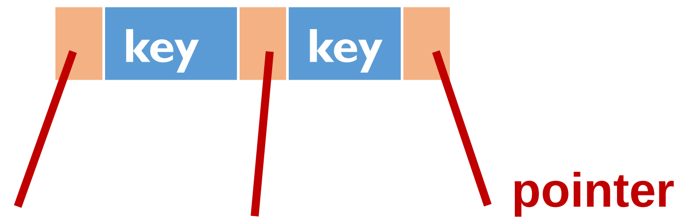
11.2

B Trees

B-tree: Definition

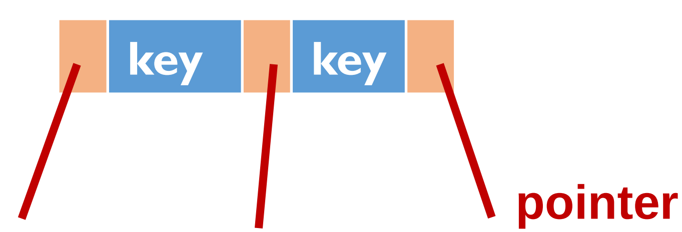
A B-tree of order m is a height-balanced m-way search tree, where each node may have up to m children, and in which:

1. Each internal node contains no more than $m - 1$ keys
2. All leaves are on the same level
3. All nodes except the root have $\lceil m/2 \rceil$ to m children
4. The root is either a leaf node, or it has 2 to m children
5. m usually is odd.

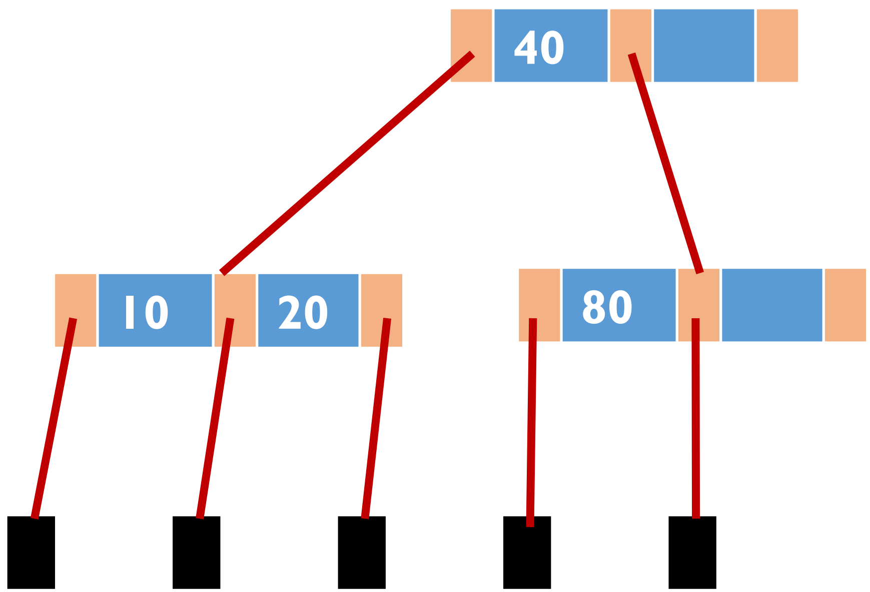


2-3 Trees

- A B-Tree of order 3 is called a 2-3 Tree.
  - 2 to 3 pointers
- In a 2-3 tree, each internal node has either 2 or 3 children.
- Most practical applications adopt larger order (e.g., $m = 128$) B-Trees.



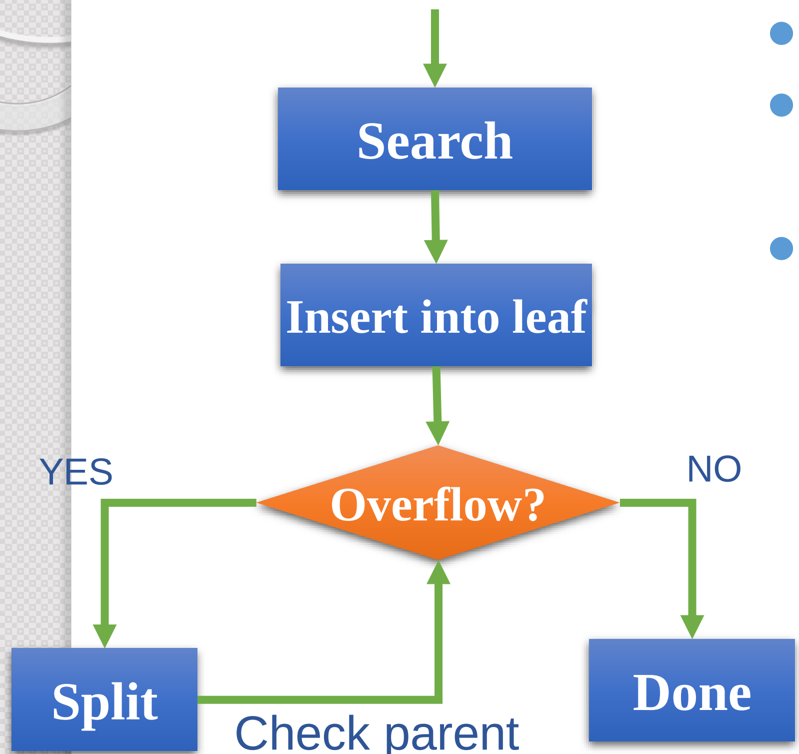
Example of a 2-3 Tree



m = 3
# of Children: 2~3

Insert 70?

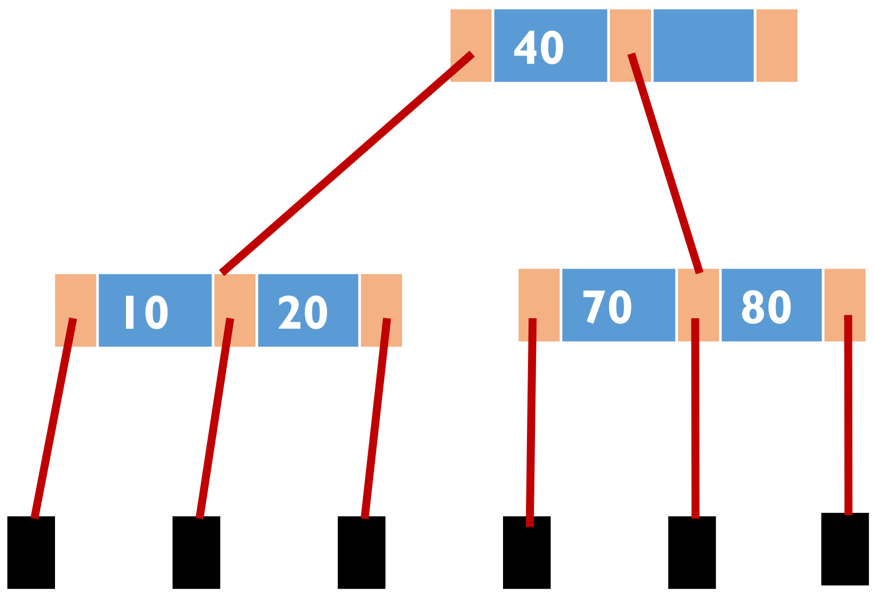
B-tree: Insert



- Search
- Insert the new key into a leaf, which is the node under work
- If the node overflows
  - If it is root, create a new root as its parent
  - Split the node into two and push up the middle key to the node's parent
  - Let the parent node be the node under work and repeat the overflow checking and split process.
- Done, if no overflow.

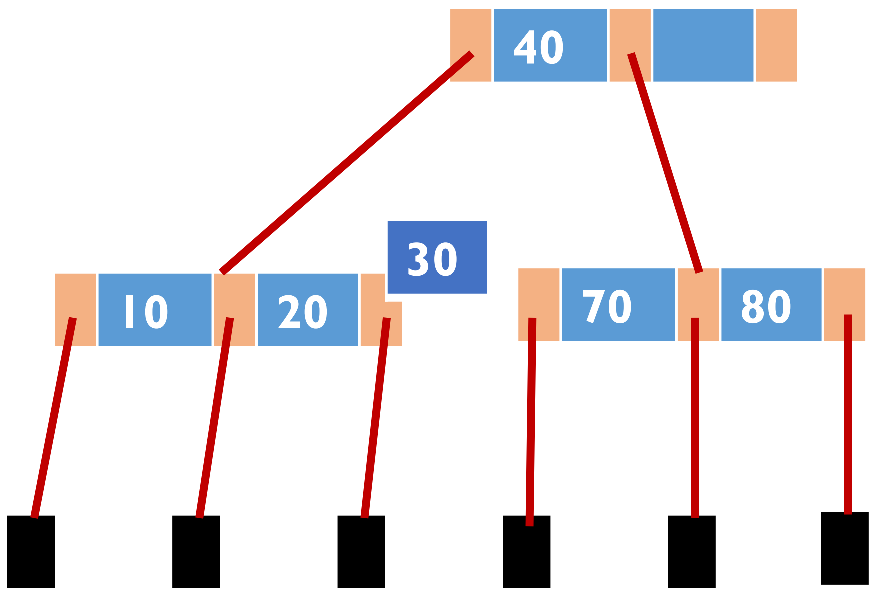
Example of a 2-3 Tree

After 70 Inserted



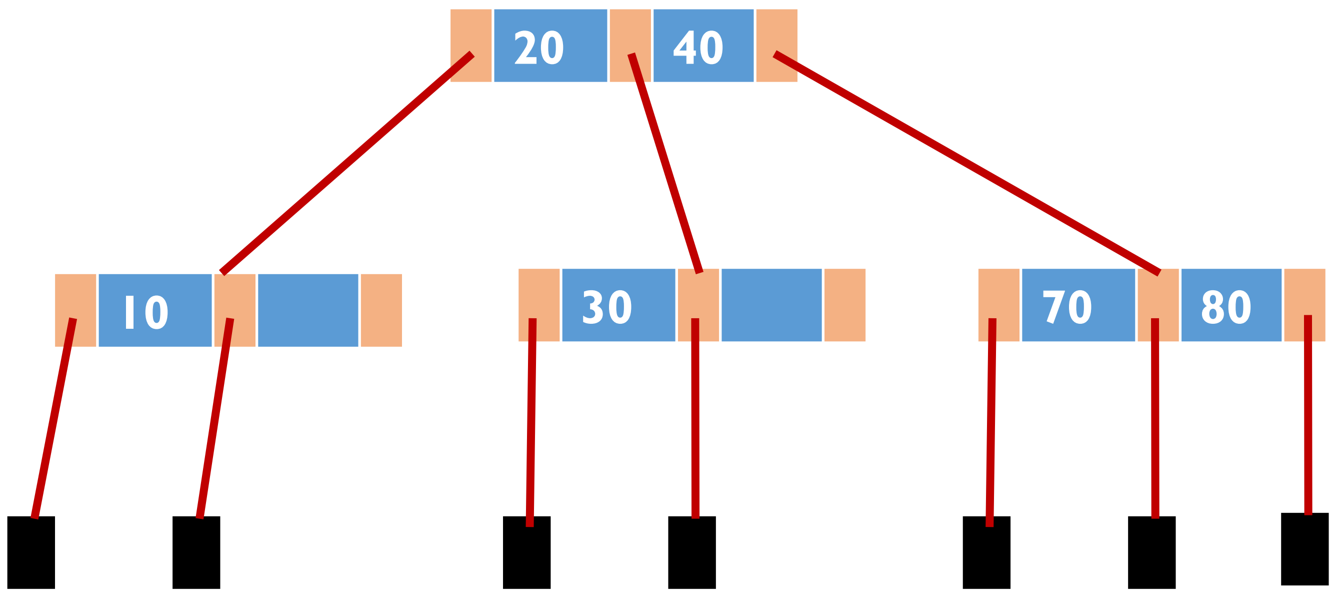
Insert 30?

Example of a 2-3 Tree

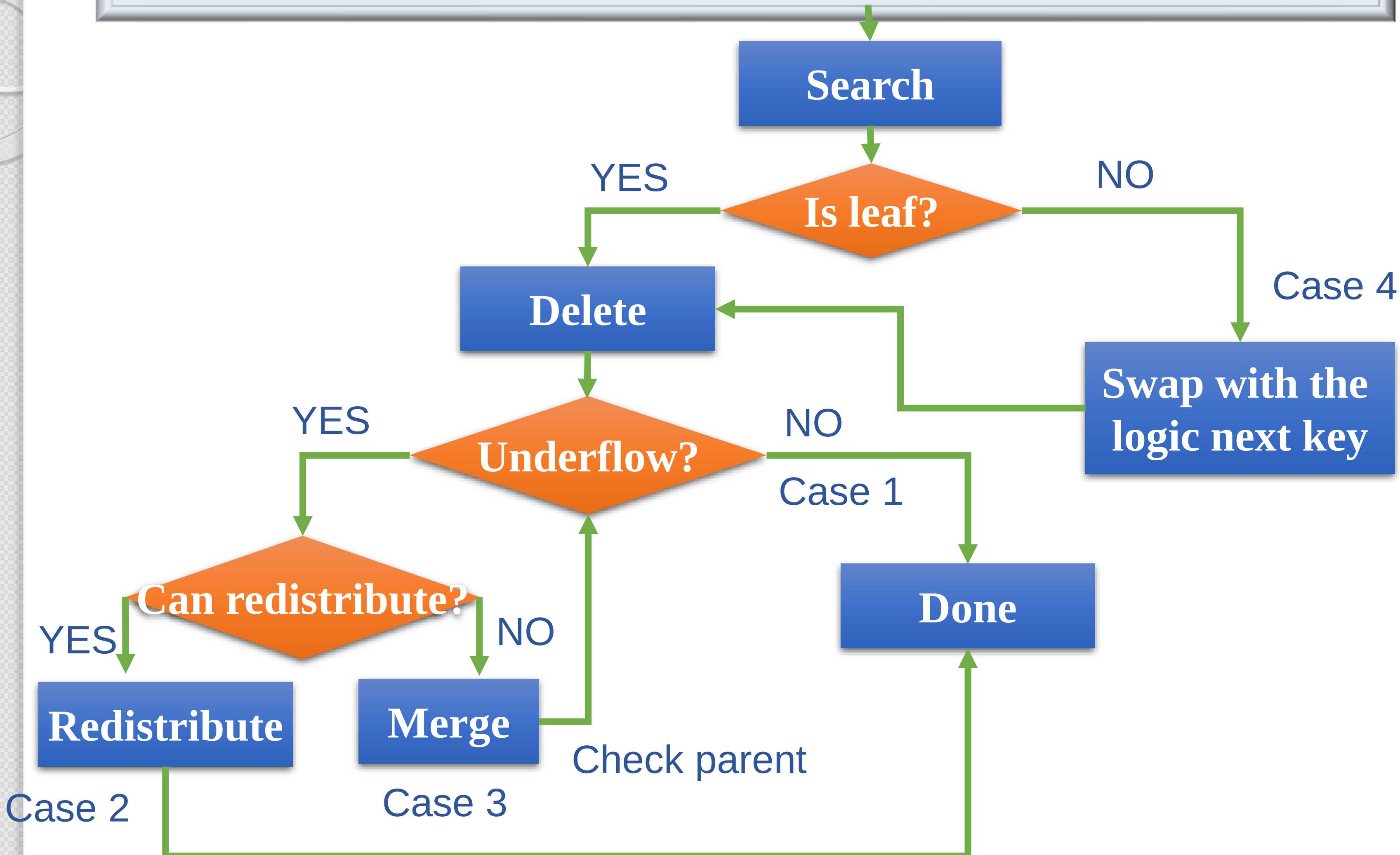


Example of a 2-3 Tree

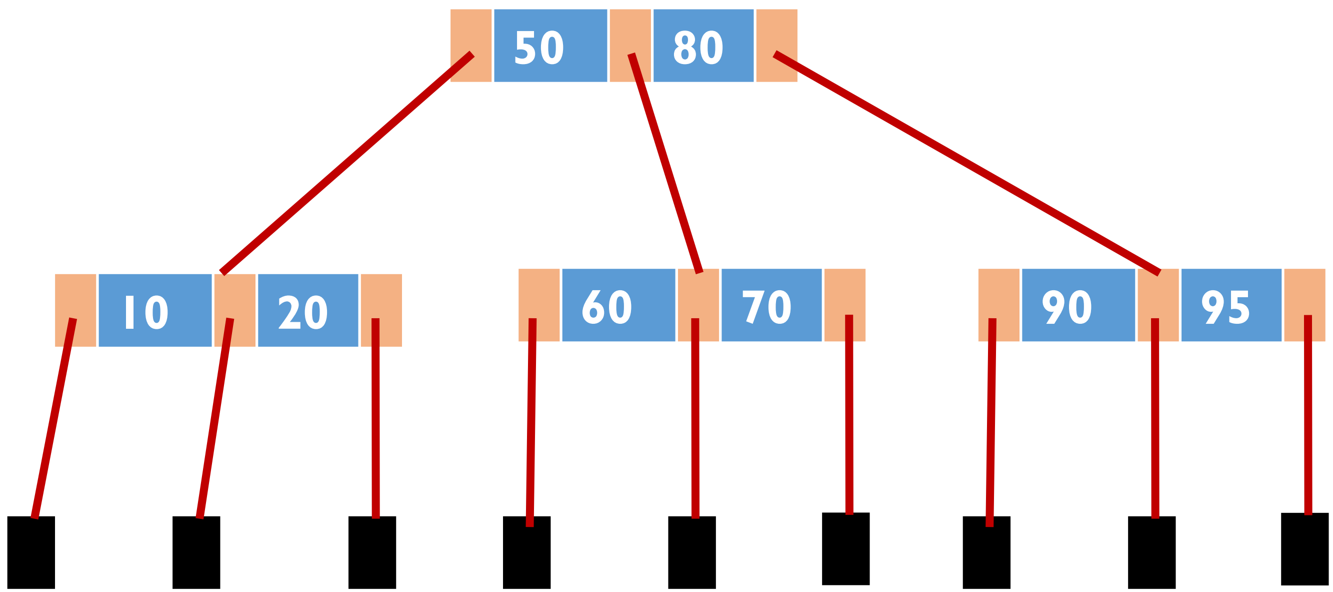
After 30 Inserted



Flow Chart of B-tree Deletion



Deletion from a 2-3 Tree

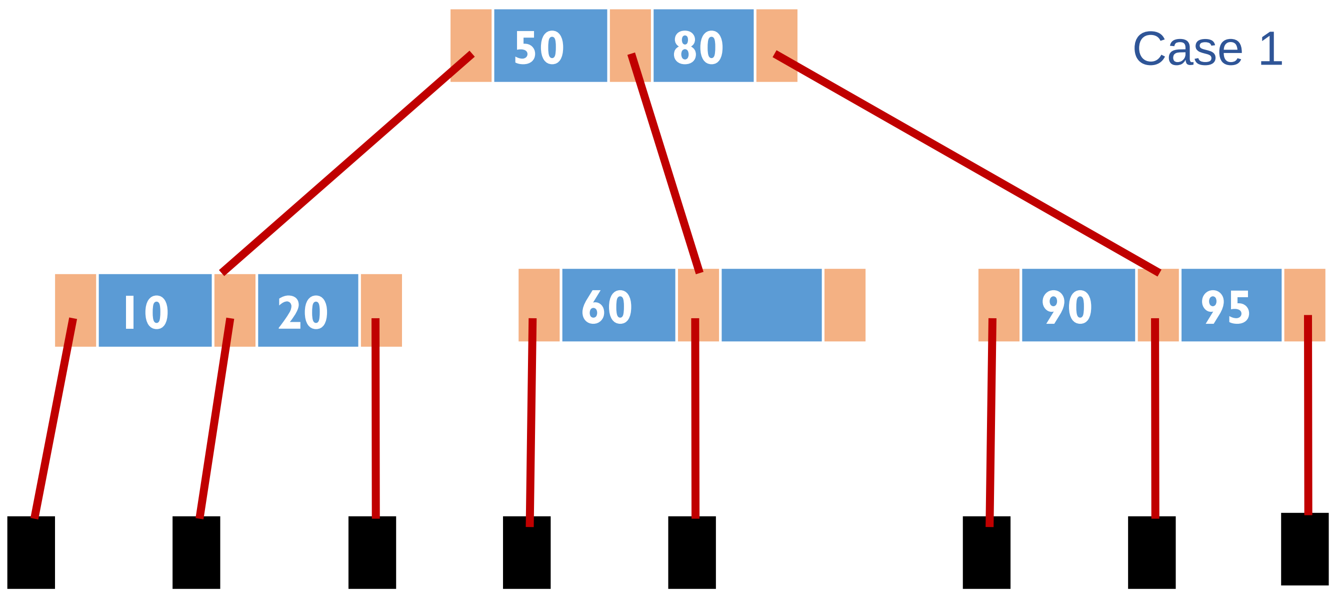


Delete 70?

Deletion from a 2-3 Tree

After 70 deleted

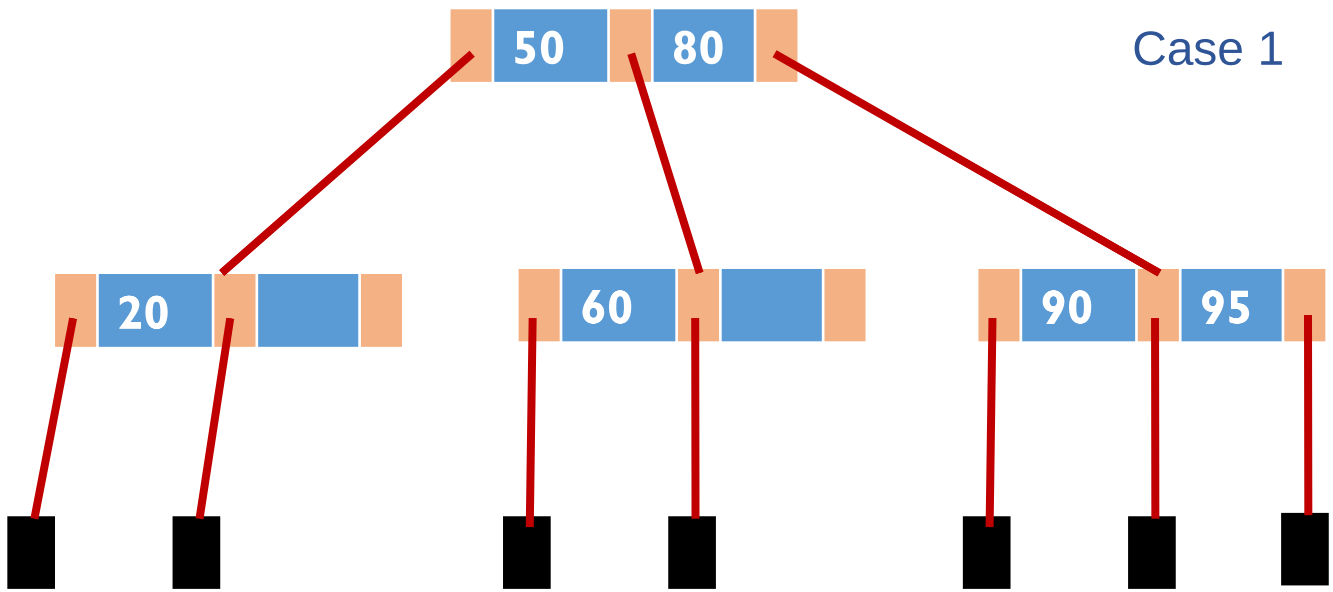
Case 1



Delete 10?

Deletion from a 2-3 Tree

After 10 deleted

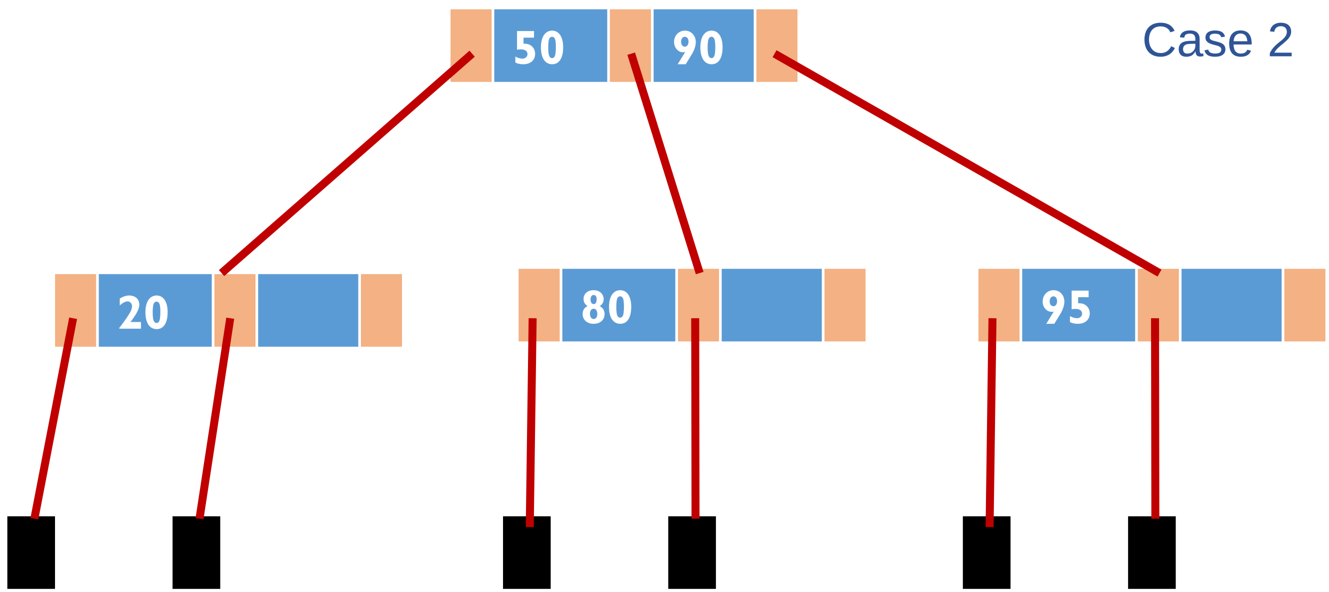


Delete 60?

Deletion from a 2-3 Tree

After 60 deleted

Case 2

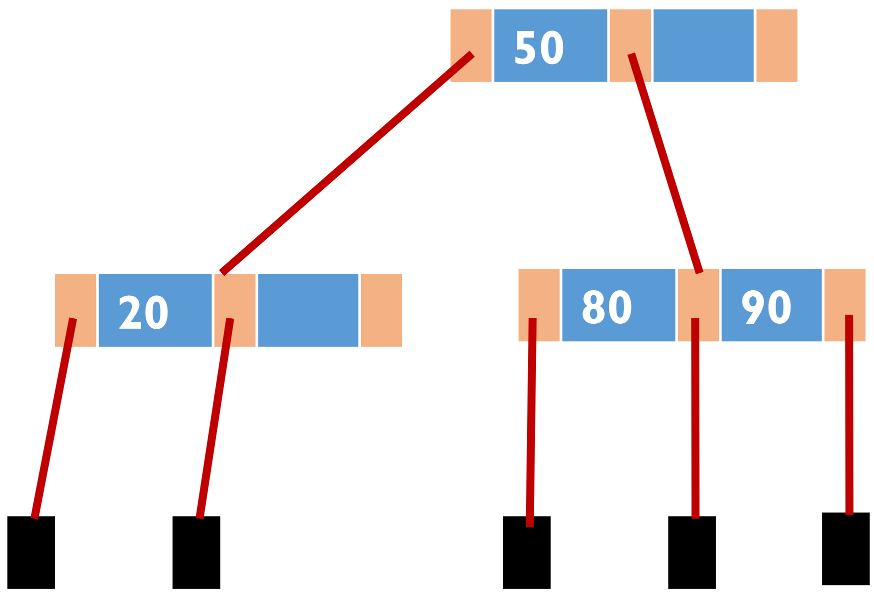


Delete 95?

Deletion from a 2-3 Tree

After 95 deleted

Case 3

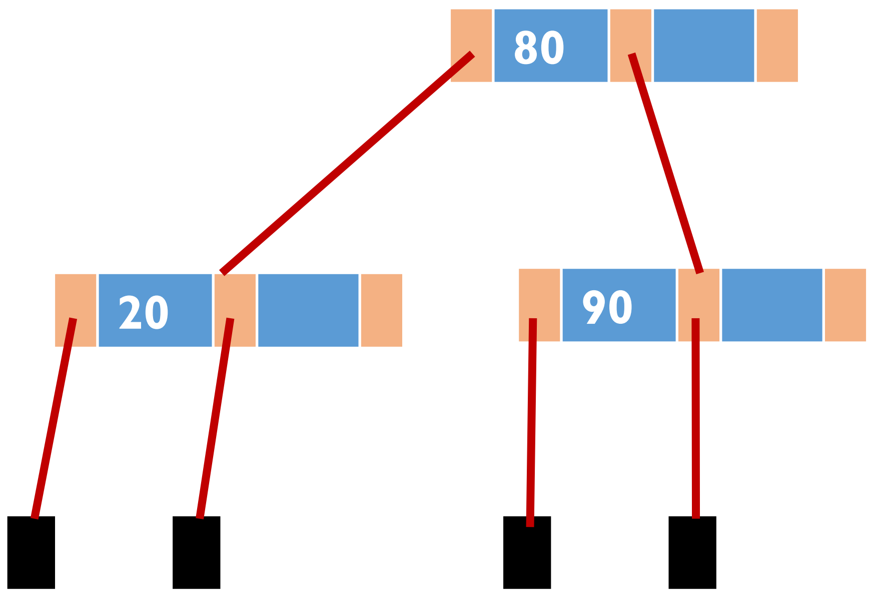


Delete 50?

Deletion from a 2-3 Tree

After 50 deleted

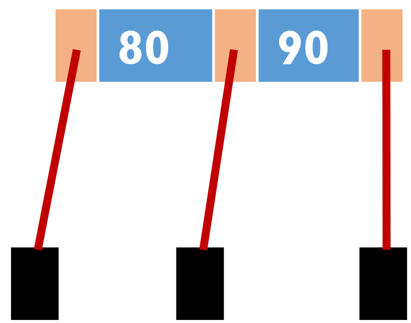
Case 1



Delete 20?

Deletion from a 2-3 Tree

After 20 deleted



B-Tree Exercise

Insert the following keys to a 5-way B-tree:

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Then delete all nodes subsequently in the reverse order of the insertion.