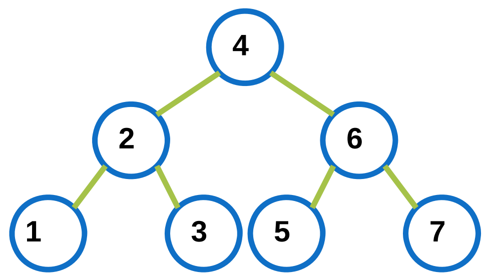
Advanced Topics – More Trees

Yi-Shin Chen
Institute of Information Systems and Applications
Department of Computer Science
National Tsing Hua University
yishin@gmail.com

Binary Search Tree

- All BST operations are O(*h*)
  - h = height of BST

- Worst case h=n
  - Insert keys 1, 2, … n
- Best case h=logn
  - Insert keys : 4, 2, 6, 1, 3, 5, 7



How to Keep a Balanced BST

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

Height Balanced Trees

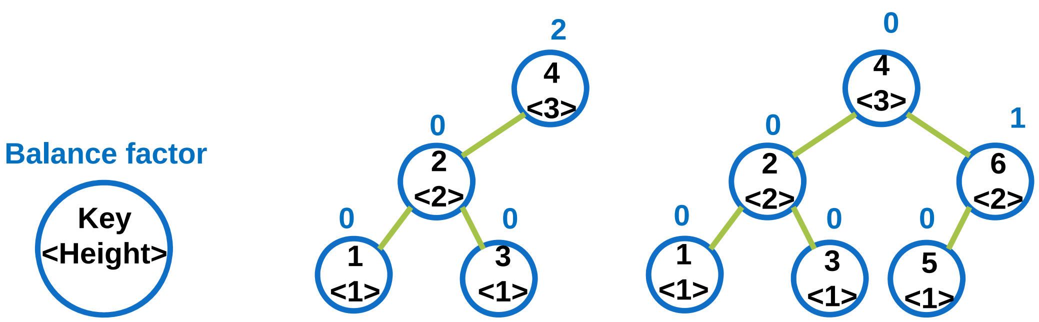
- An empty tree is height balanced.
- If T is a non-empty binary tree with T_L and T_R
  - As its left and right subtrees respectively
- Balance factor

$$bf(\boldsymbol{T}) = height(\boldsymbol{T_L}) - height(\boldsymbol{T_R})$$

- $\boldsymbol{T}$ is height balanced iff
  1) $\boldsymbol{T_L}$ and $\boldsymbol{T_R}$ are height balanced.
  2) $|bf(\boldsymbol{T})| \le 1$

AVL Trees

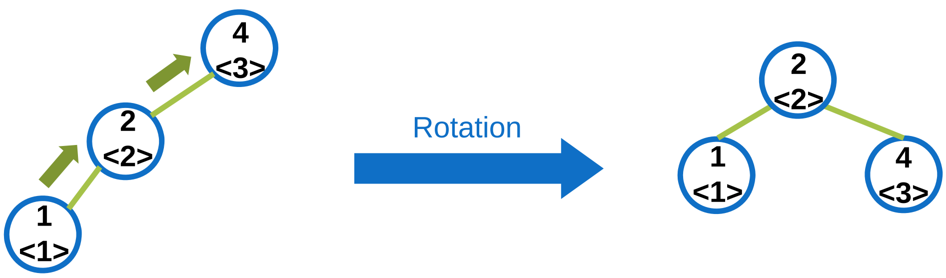
- AVL tree is a *height-balanced* binary search tree
- Each node in an AVL tree stores the current node height
  - For calculating the balance factor



Representation

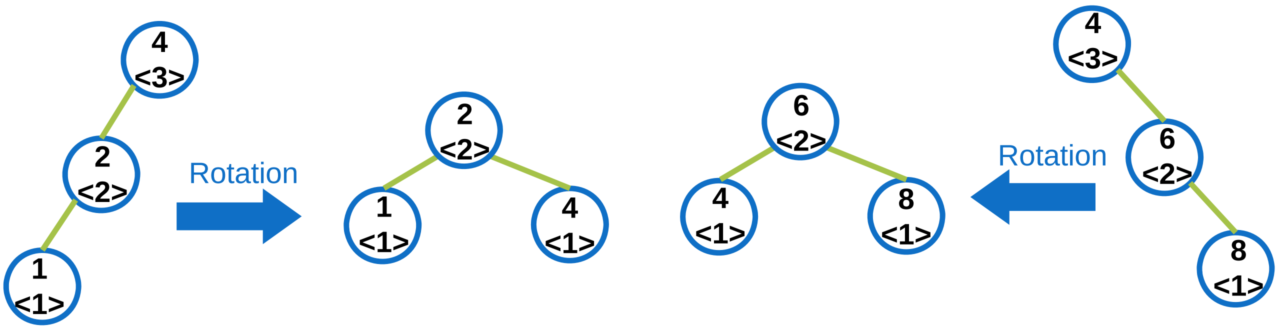
Rebalancing

- During BST insertion/deletion operations,
  - if balance factor >1 or <–1, activate rebalance process
- Rebalancing process
  - Fix unbalanced situations using “rotations”



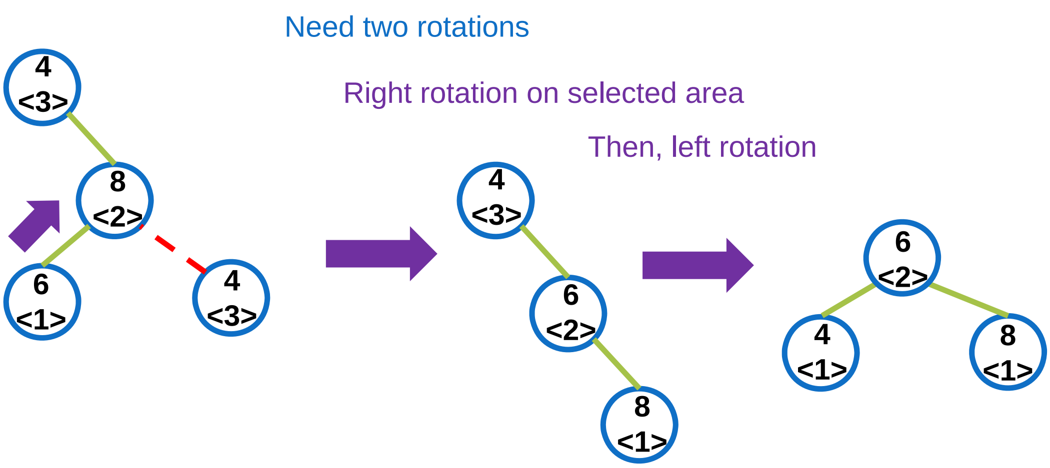
Rebalancing Operations

- Right rotation
- Left rotation



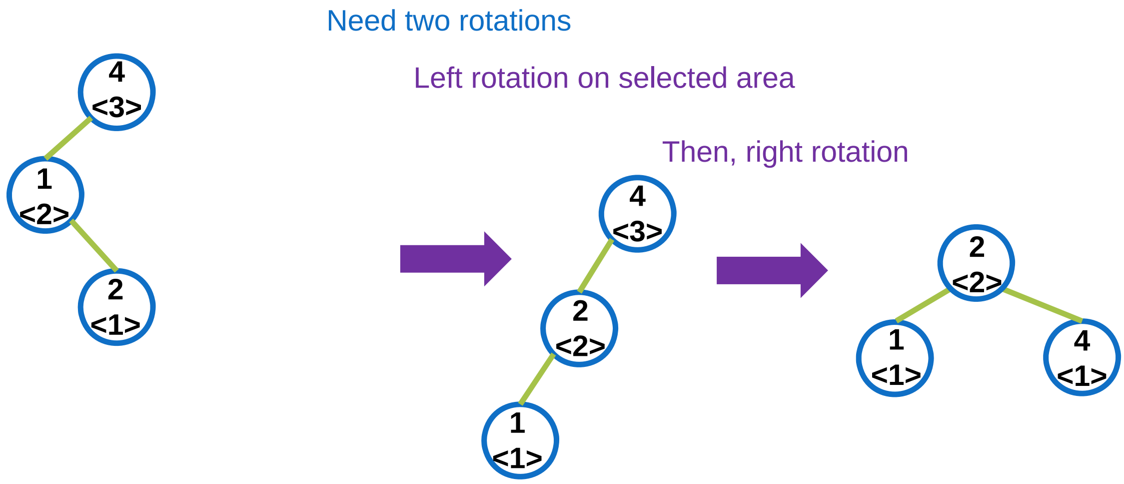
Rebalancing Operations

- Two rotations



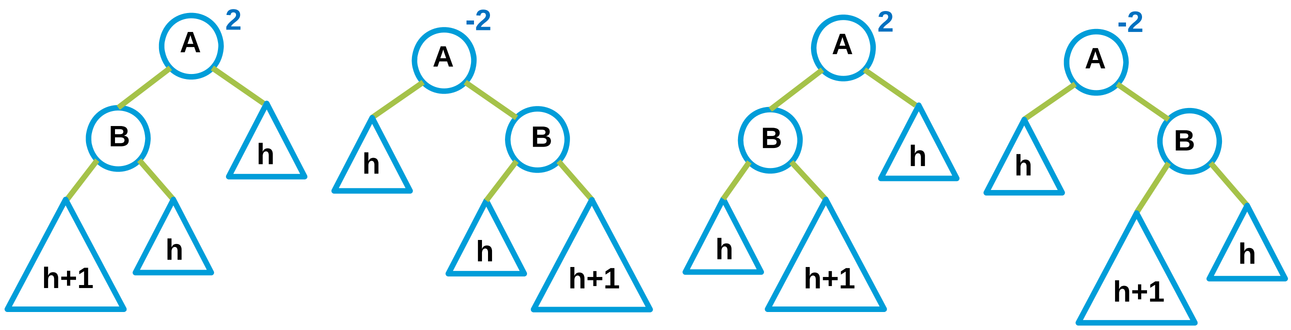
Rebalancing Operations

- Two rotations



Unbalanced Situations

- There are 4 kinds of unbalanced situations:
  - 2 outside cases: require single rotation (LL, RR)
  - 2 inside cases: require two rotations (LR, RL)

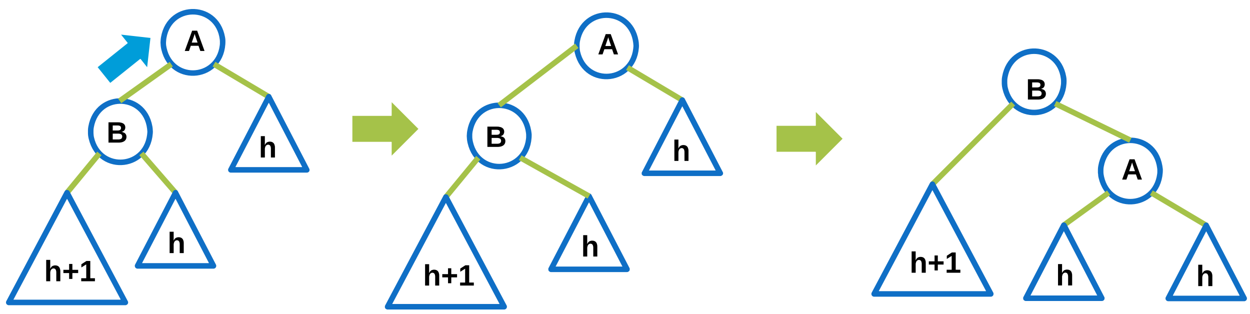


2 outside cases

2 inside cases

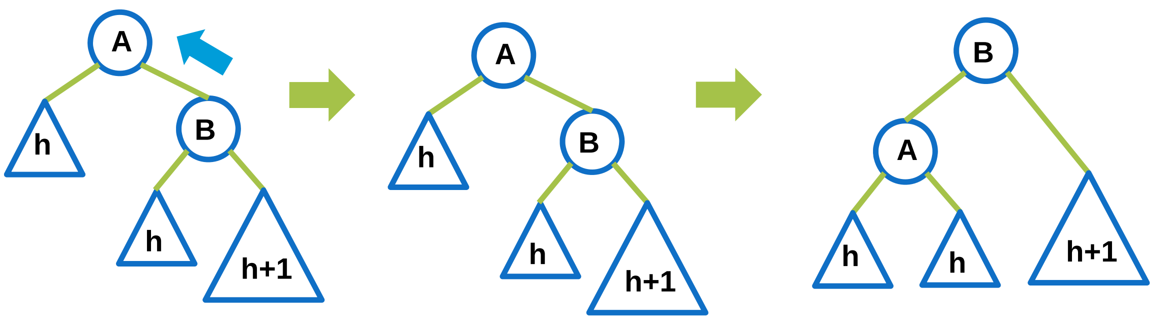
Outside Cases - LL Rotation

- Right rotation (LL Rotation)
  - The new node is inserted in the left subtree of the left subtree of A



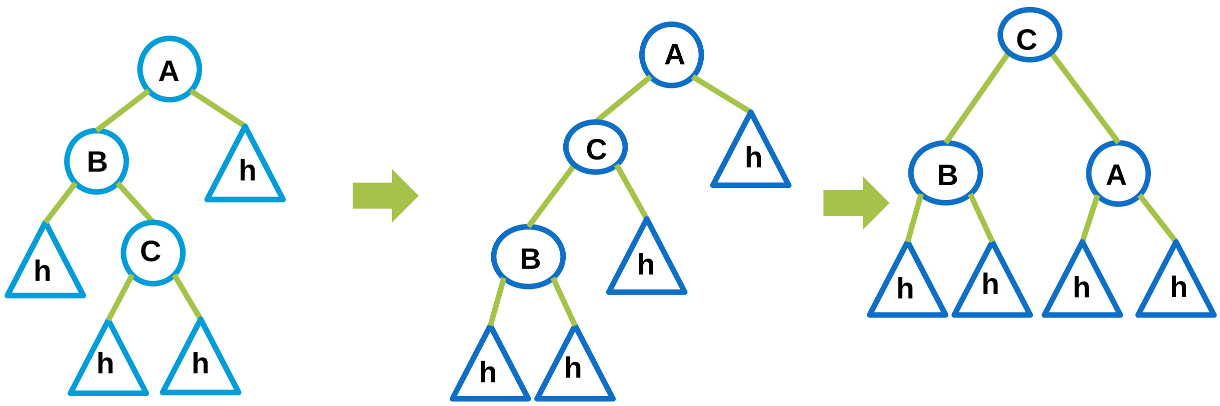
Outside Cases - RR Rotation

- Left rotation (RR Rotation)
  - RR Rotation: The new node is inserted in the right subtree of the right subtree of A



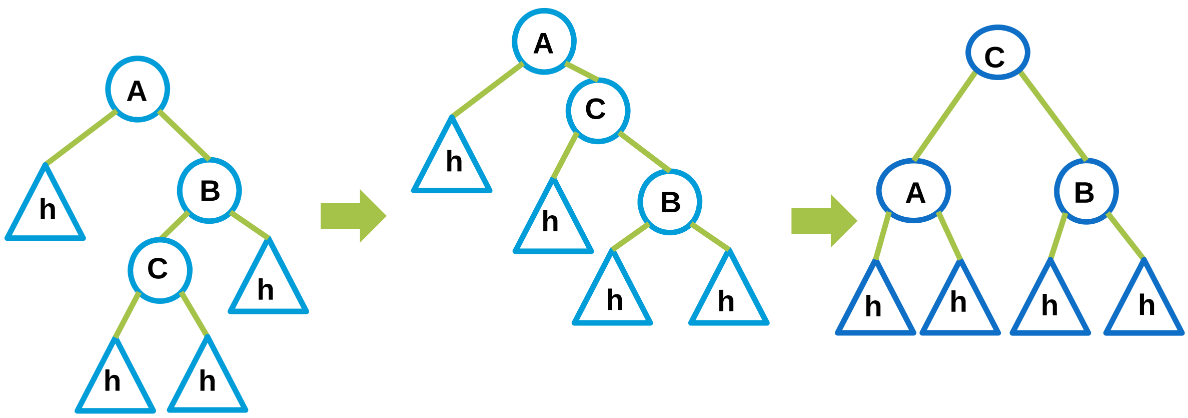
Inside Cases - LR Rotation

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

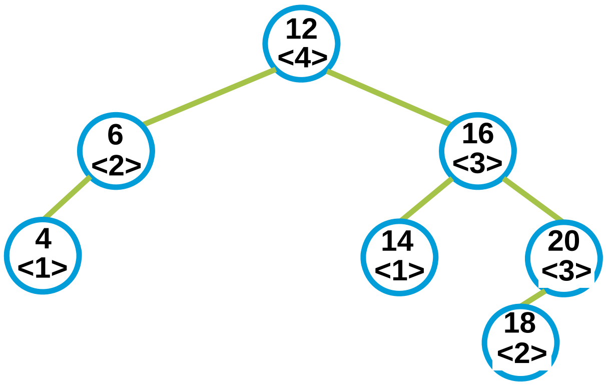


Inside Cases - RL Rotation

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

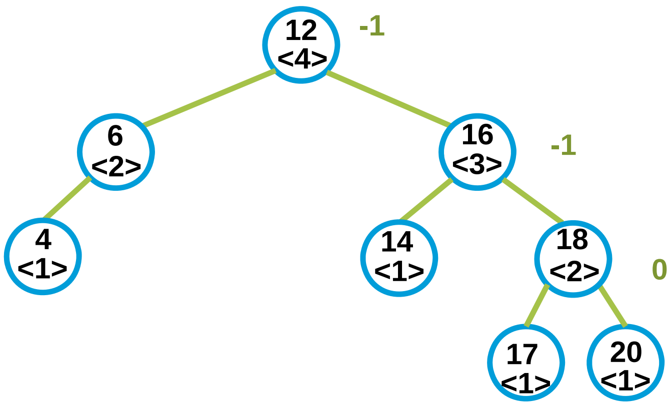


Example AVL Tree:
Insert 17



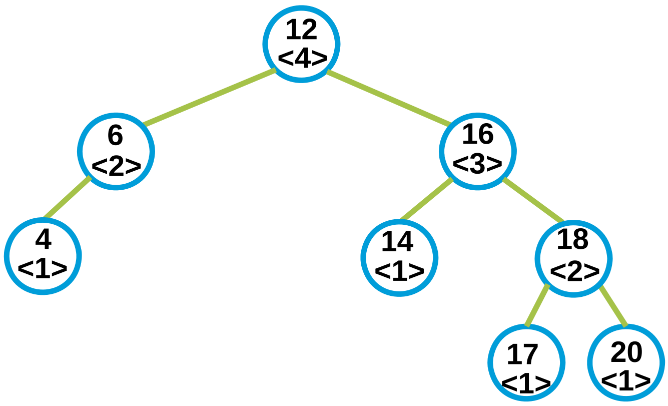
Example AVL Tree:
Insert 17

Finish checking



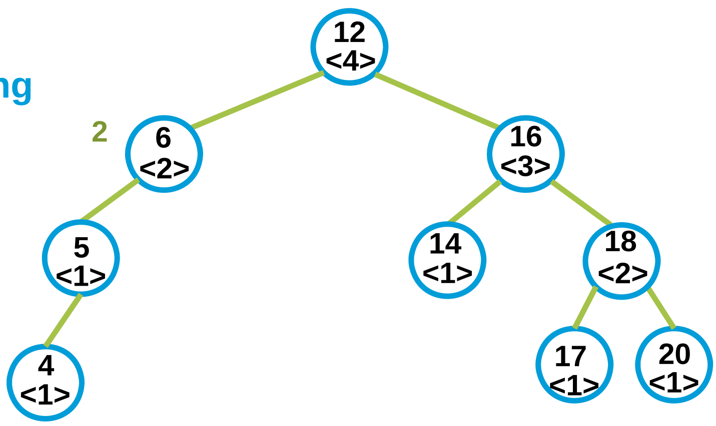
Example AVL Tree:
Insert 5

Use rebalancing LR rotation!

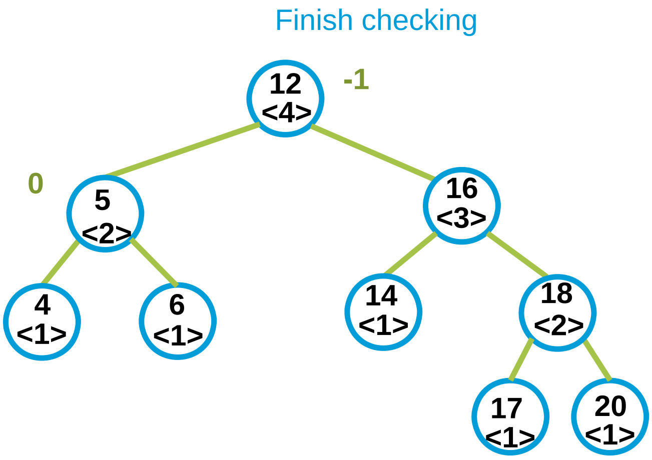


Example AVL Tree:
Insert 5

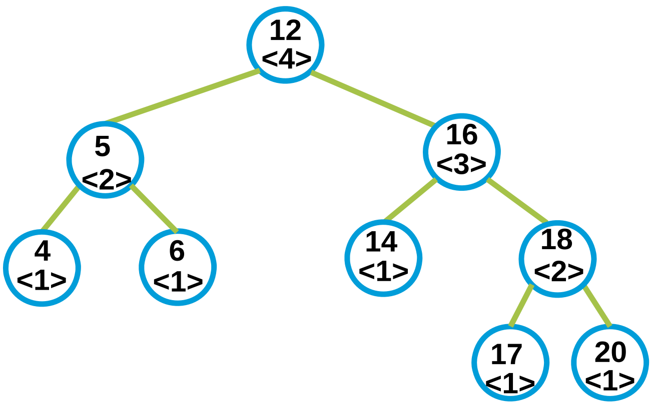
Use rebalancing LR rotation!



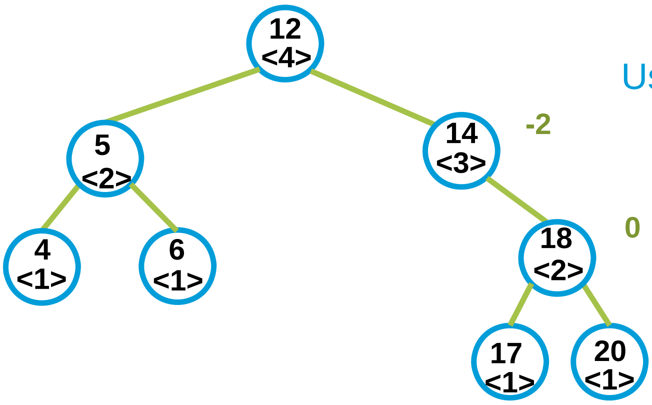
Example AVL Tree:
Insert 5



Example AVL Tree:
Delete 16



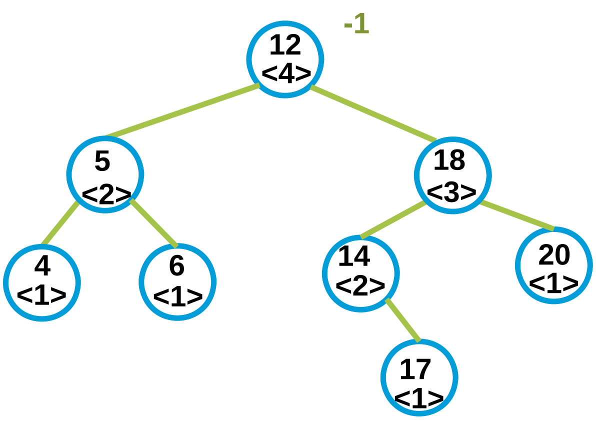
Example AVL Tree:
Delete 16



Use rebalancing
RR rotation!

Example AVL Tree:
Delete 16

Finish checking



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;
};
```


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 );
}
```

AVL Tree Rebalance

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


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;
}
```

B-Tree

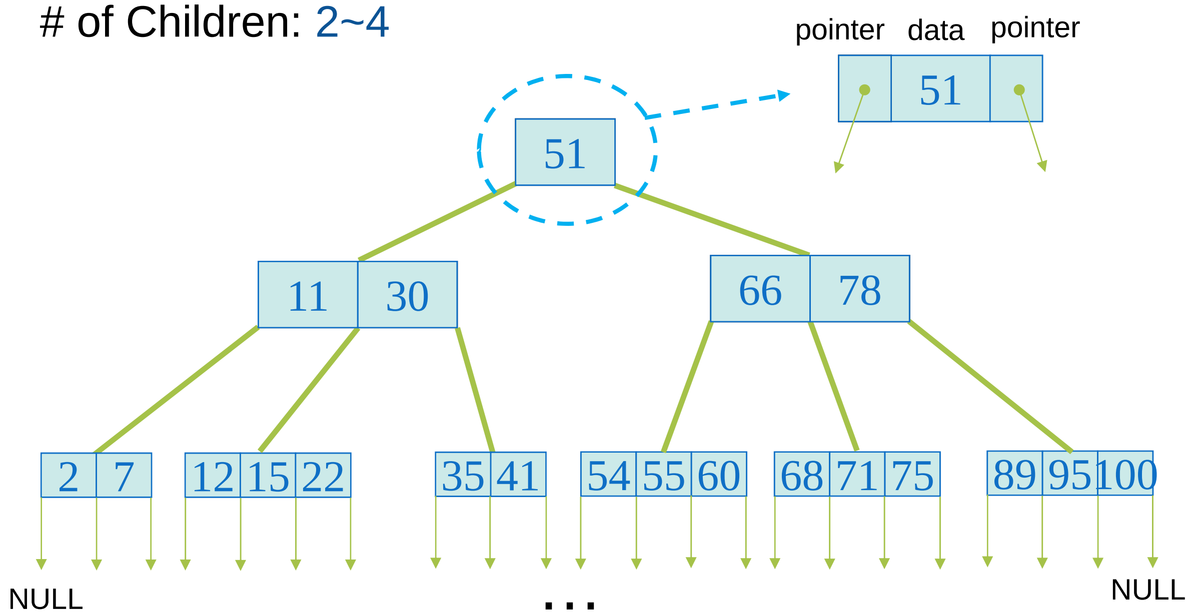
B-tree: Definition

- A B-tree of order *m* is a height-balanced tree , where each node may have up to *m* children, and in which:
  1. All leaves are on the same level
  2. No node can contain more than *m* children
  3. All nodes except the root have at least $\left\lceil \frac{m}{2} \right\rceil$ children
  4. The root is either a leaf node, or it has from 2 to m children

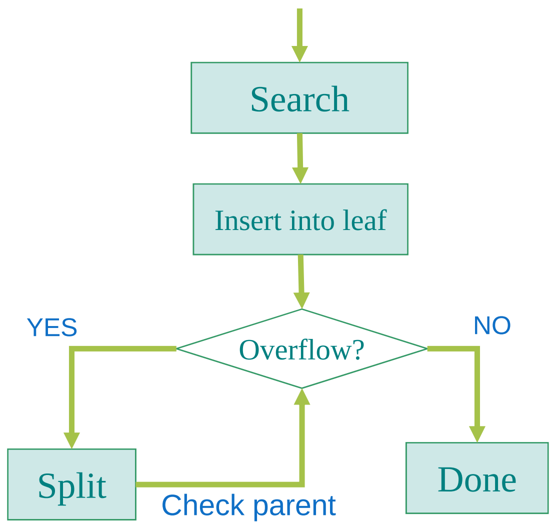
B-tree: Example

m = 4

# of Children: 2~4



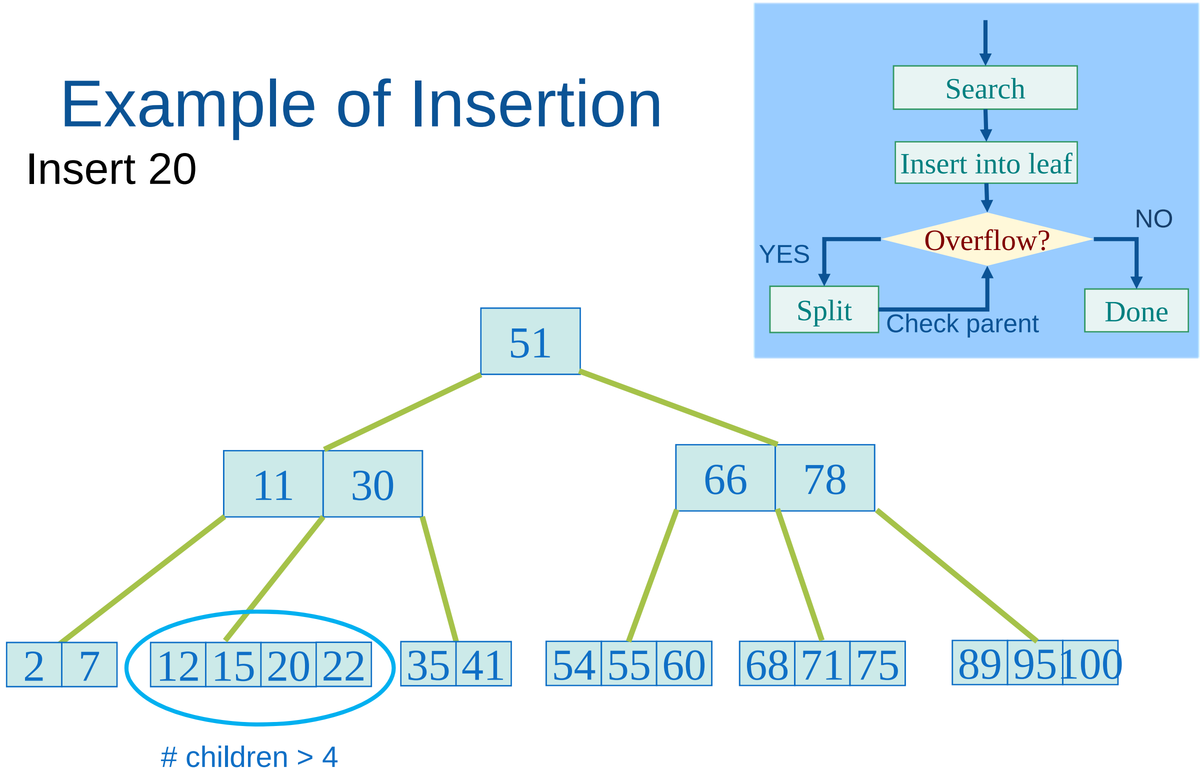
B-tree: Insert



- Search
- Insert the new key into a leaf
- If the leaf overflows
  - Split the leaf into two and push up the middle key to the leaf's parent
  - If the parent overflows
    - Split the parent into two and push up the middle key again
  - This strategy might have to be repeated all the way until arriving the root
  - If necessary, the root is split in two and the middle key is pushed up to a new root, making the tree one level higher

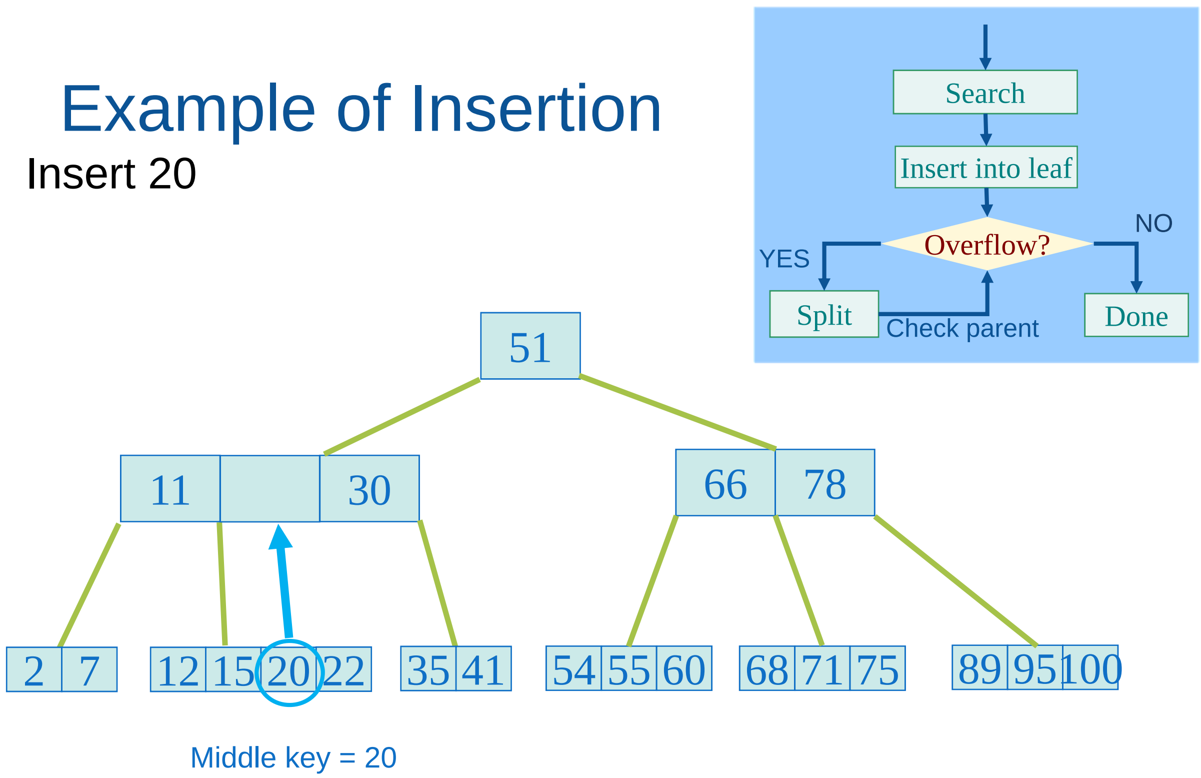
Example of Insertion

Insert 20



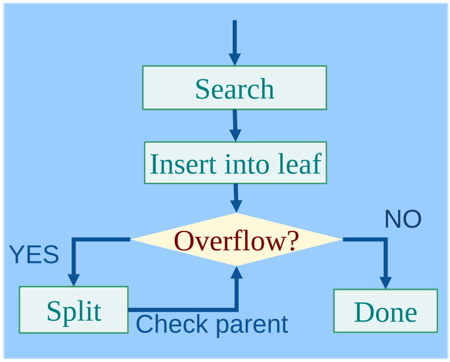
Example of Insertion

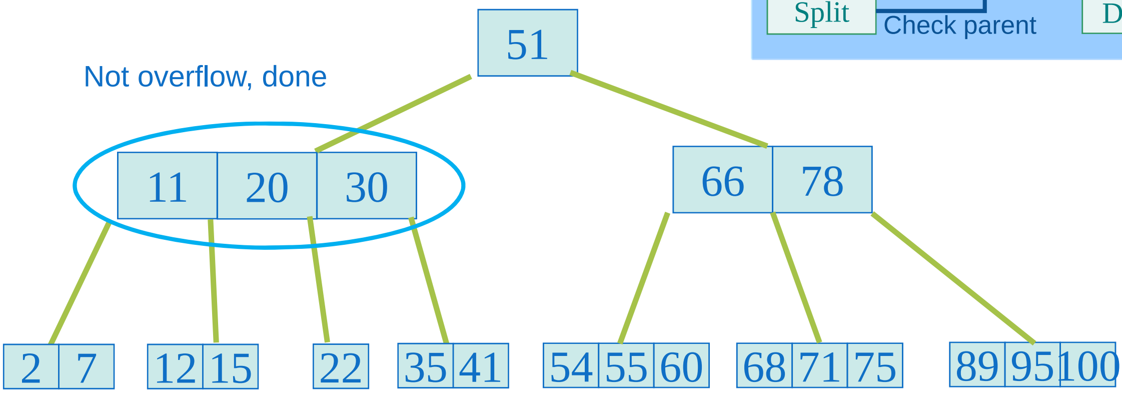
Insert 20



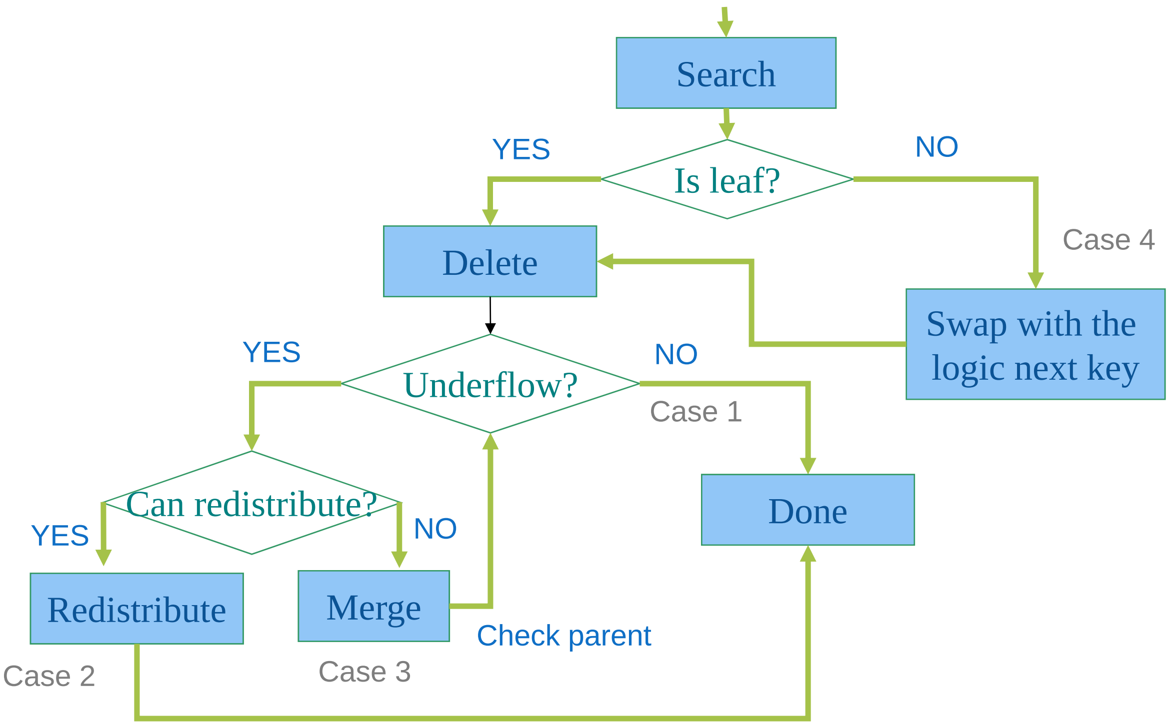
Example of Insertion

Insert 20



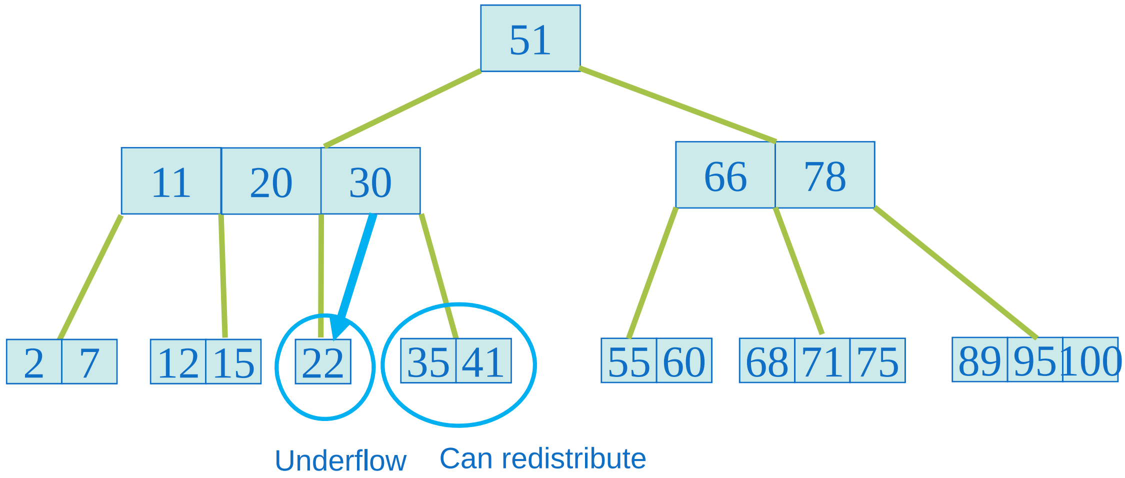


B-tree: Flow Chart of Deletion



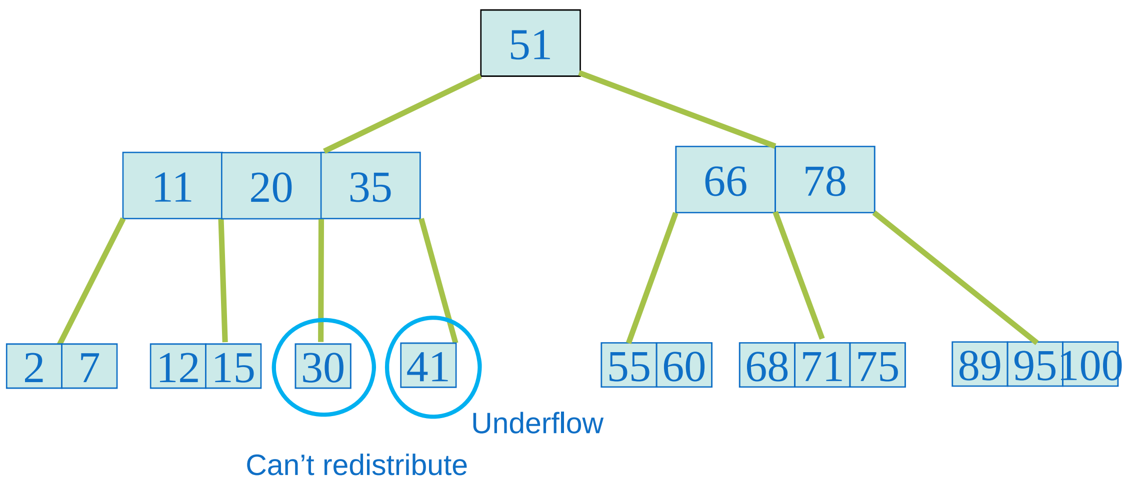
Example of Deleting A Leaf

Case 2 (Redistribute): delete 22



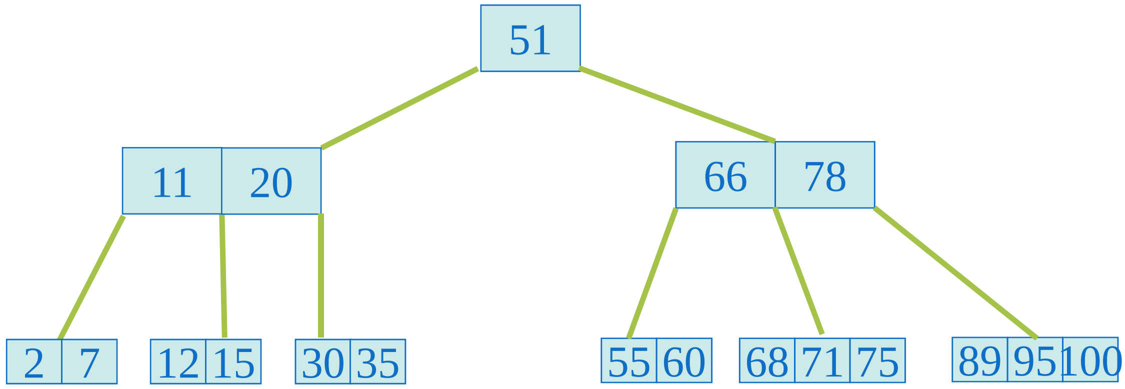
Example of Deleting A Leaf

Case 3 (Merge): delete 41



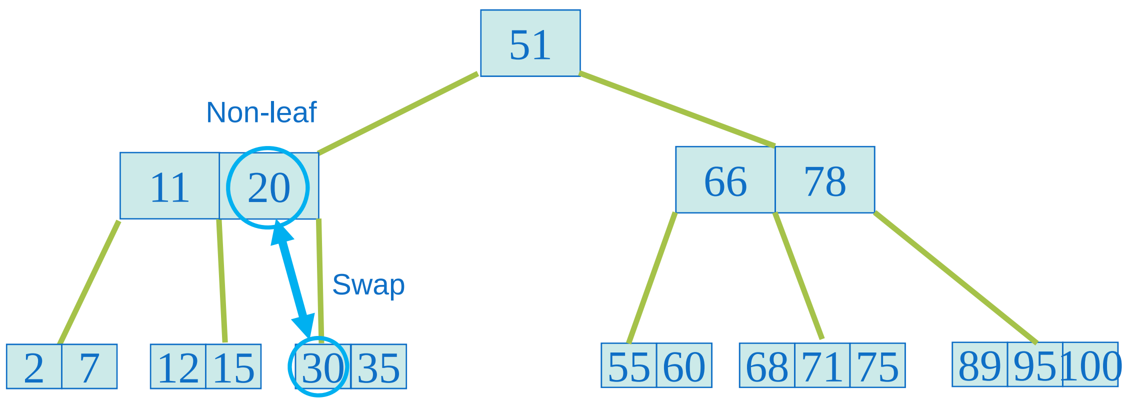
Example of Deleting Leaf

Case 3 (Merge): delete 41



Example of Deleting An Non-Leaf

Case 4 (non-leaf): delete 20



The key in the leftmost leaf of the right subtree

B-tree: Deleting Leaf

■Leaf

Delete the key
If the number of keys is valid after deletion
    O.K.
else   //underflow
    If any sibling node has keys more than $\left\lceil \frac{m}{2} \right\rceil$ -1
        Redistribute key from siblings
    else
        Merge nodes into one node
        Check if parent is underflow

B-tree: Deleting Non-leaf

■Non-leaf

Swap the key with the logic next key (i.e. the first key in the leftmost leaf of the right subtree)
Call Delete leaf

P.S. Alternatively, we can choose the logic previous key (i.e. the last key in the rightmost leaf of the left subtree) to swap