Algorithms and Data Structures (CSci 115)

California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- AVL trees
  - Rotations
  - How to insert an element
  - How to remove an element

Introduction

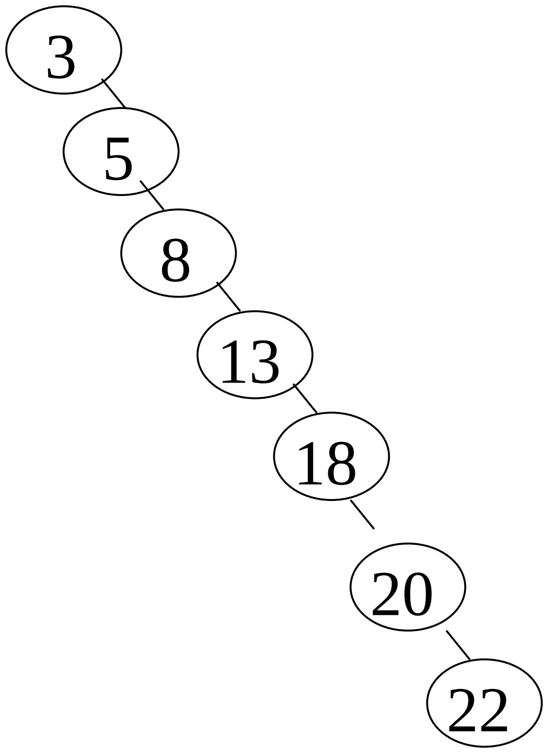
- Rationale
  - Most operations on a binary search tree (BST) take time directly proportional to the **height** of the tree
  - → desirable to keep the height small!
- Definition
  - Binary tree with height ***h*** can contain at most $2^0+2^1+\cdots+2^h = 2^{h+1}-1$ nodes
  - n = number of elements in the tree
  - → $n \le 2^{h+1}-1$ → $h \ge \text{floor}(\log 2(n))$

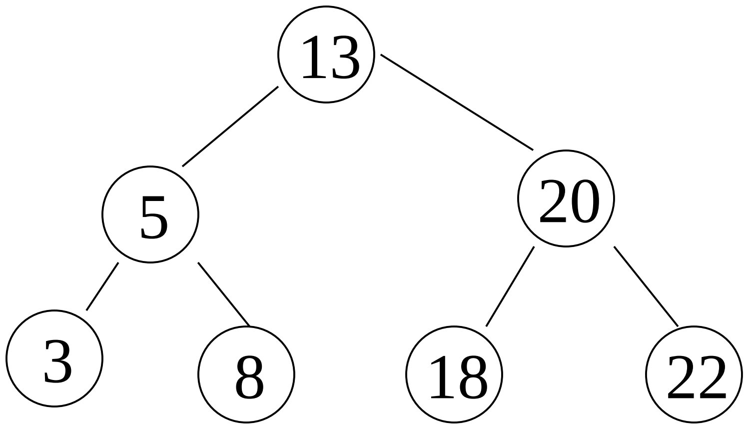
Introduction

- BST tree
  - Operations: search, max, min, insert, delete…
    - O(h) time where h is the height of the tree
  - Cost of the operations may become O(n)
    - for a skewed Binary tree !! (list)
- Goal:
  - If we make sure that height of the tree **remains** O(logn)
    - after every insertion and deletion
  - Then we can guarantee an **upper** bound of O(logn) for all these operations!
    - The height of an AVL tree is always O(logn)
      - With n being the number of nodes in the tree

Rationale

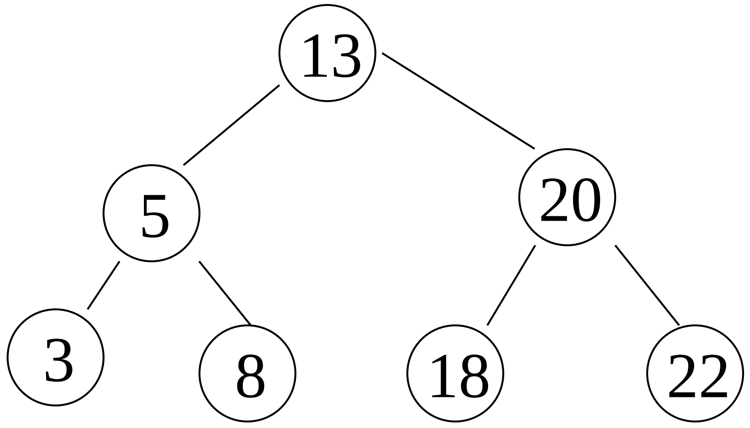
- When building a binary search tree (BST)
  - type of trees we would we like?
  - Example: 3, 5, 8, 20, 18, 13, 22

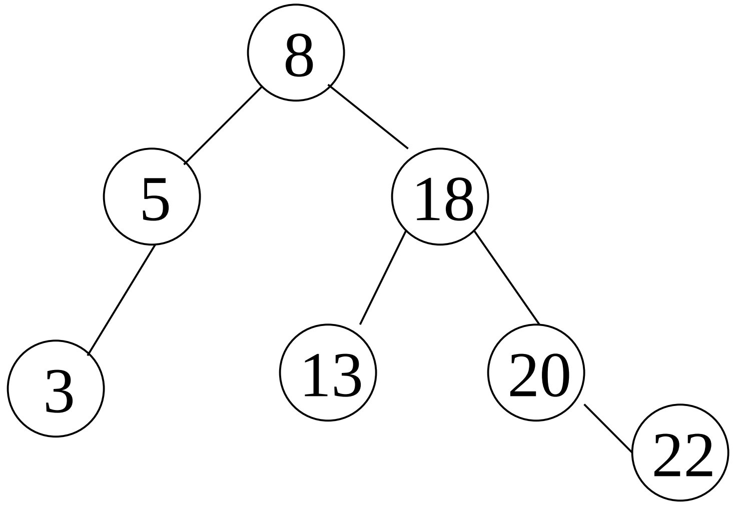




Rationale

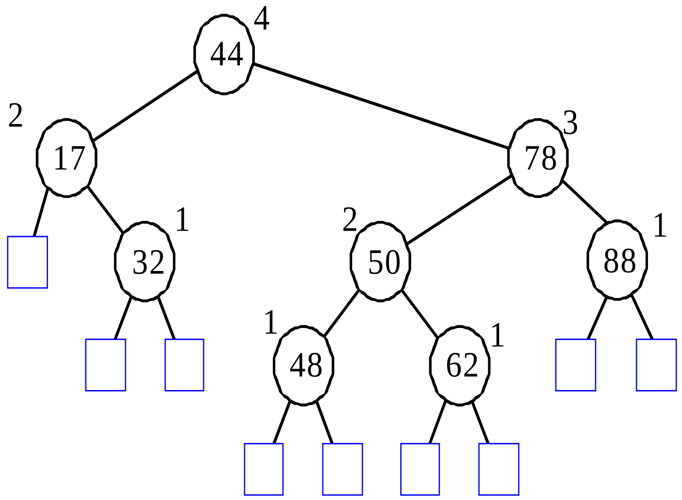
- Complete binary tree
  - Difficult to build when we allow **dynamic** insert and remove.
  - We want a tree that has the following properties
    - Tree height = O(log(N))
      - To allow dynamic insert and remove with O(log(N)) time complexity.
  - AVL tree is one of this type of trees.





AVL (Adelson-Velskii and Landis) Trees

- AVL Tree (1962)
  - **BST** such that for every internal node v of T
    - the heights of the children of v can differ by **at most 1**.



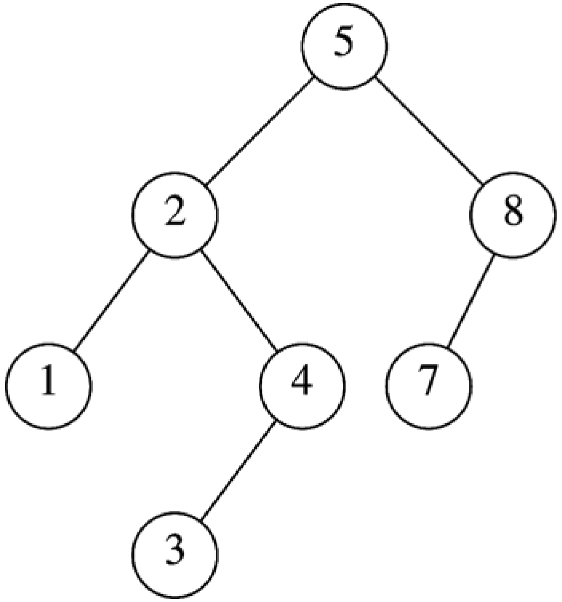
Example of an AVL tree where the heights are shown next to the nodes:

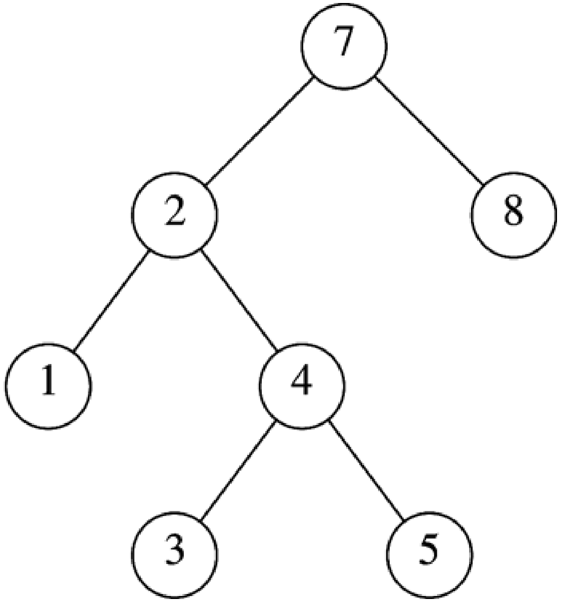
AVL (Adelson-Velskii and Landis) Trees

- AVL tree
  - A binary search tree (BST) with balance condition
    - To ensure depth of the tree is O(log(N))
    - And consequently, search/insert/remove complexity bound O(log(N))
- Balance condition
  - For every node in the tree: height of left and right subtree can differ by **at most 1**

AVL Tree

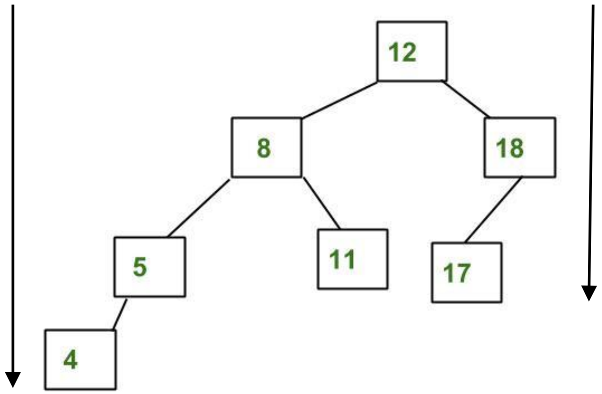
- Example:
  - ➢AVL tree or not?



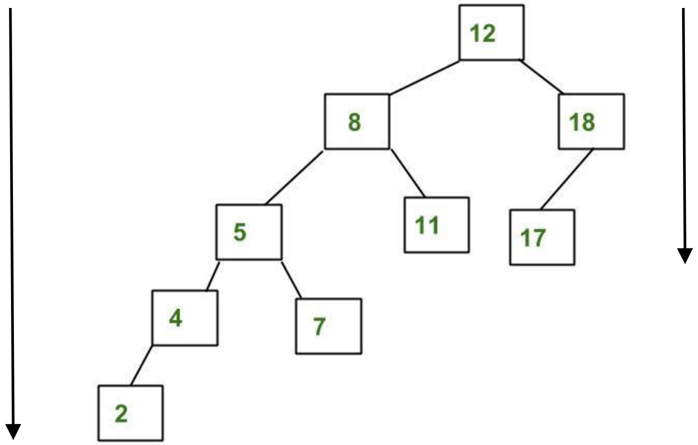


AVL Tree

- Example
  - AVL tree or not?



An AVL tree



Not an AVL tree

Height of an AVL tree

- **Theorem**:
  - The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**:
  - Let us bound **n(h)**
    - the minimum number of **internal** nodes of an AVL tree of height h.
  - We easily see that n(0) = 1 (just the root) and n(1) = 2 (left & right children)
  - For h > 2, an AVL tree of height h contains
    - the root node
    - one AVL subtree of height h-1 (can be left or right)
    - one AVL subtree of height h-2 (at worst) (can be left or right).
  - That is, n(h) >= 1 + n(h-1) + n(h-2)
  - Knowing n(h-1) > n(h-2) → we get n(h) > 2n(h-2). So →

    n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

    $n(h) > 2^i n(h-2i)$
  - Solving the base case we get: $n(h) > 2^{h/2-1}$
  - → log: h < 2log n(h) +2
  - Since n>=n(h),
    - → h < 2log(n)+2 and the height of an AVL tree is O(log n)

AVL Tree Insert and Remove

- **Step 1**
  - Do binary search tree insert and remove
- **Step 2**
  - The balance condition can be violated sometimes
    - Do something to fix it : **rotations**
    - After rotations, the balance of the whole tree is maintained

AVL

- C++ definition

```
// AVL node
template <class T>
class AVLnode {
public:
    T key;
    int balance;
    AVLnode *left, *right, *parent;

    AVLnode(T k, AVLnode *p) : key(k), balance(0), parent(p),
        left(NULL), right(NULL) {}

    ~AVLnode() {
        delete left;
        delete right;
    }
};
```

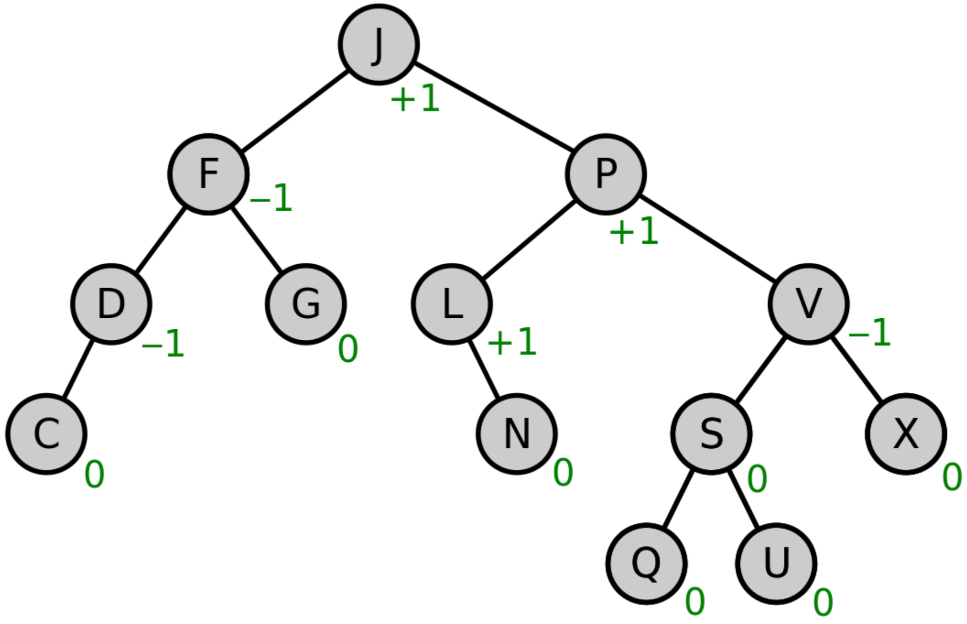
AVL

- What we need:
  - Height of a tree
    - Recursive function
  - SetBalance
    - Difference between right and left

```cpp
template <class T>
int AVLtree<T>::height(AVLnode<T> *n) {
    if (n == NULL)
        return -1;
    return 1 + std::max(height(n->left), height(n->right));
}

template <class T>
void AVLtree<T>::setBalance(AVLnode<T> *n) {
    n->balance = height(n->right) - height(n->left);
}
```

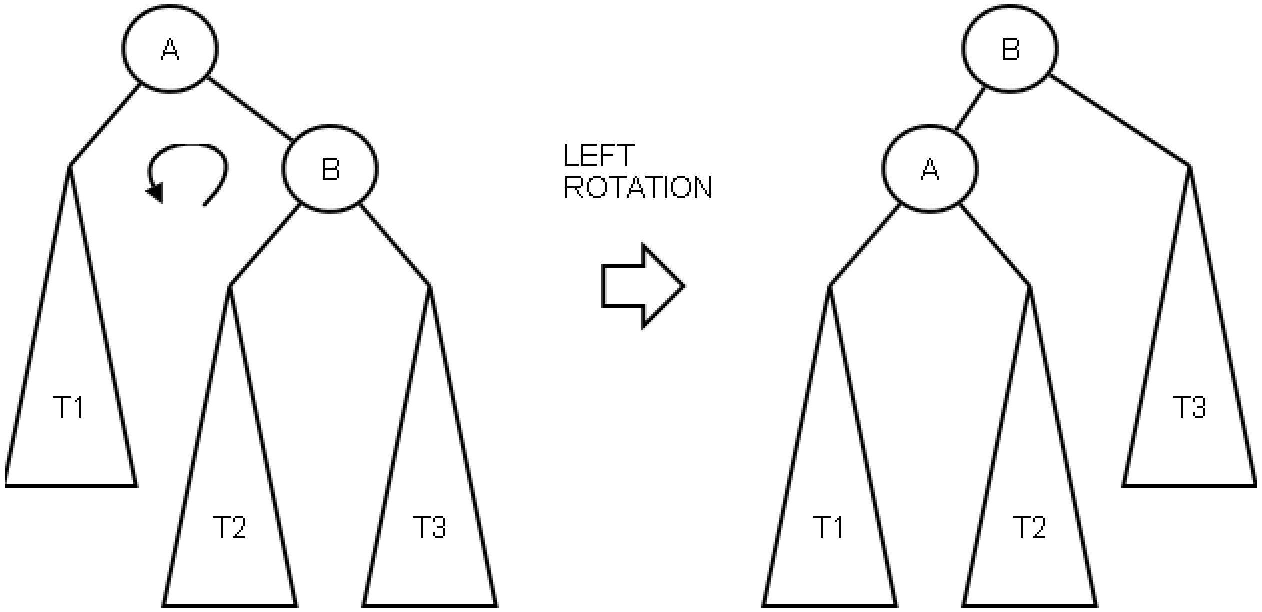
AVL

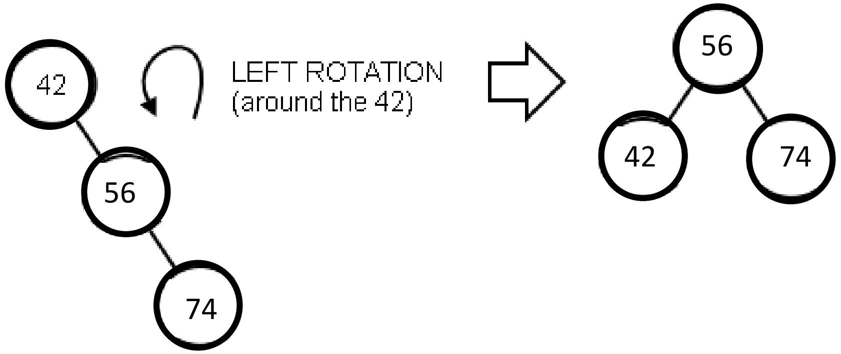
- SetBalance
  - Height(right)-Height(left)
  - Example



Rotation - Left

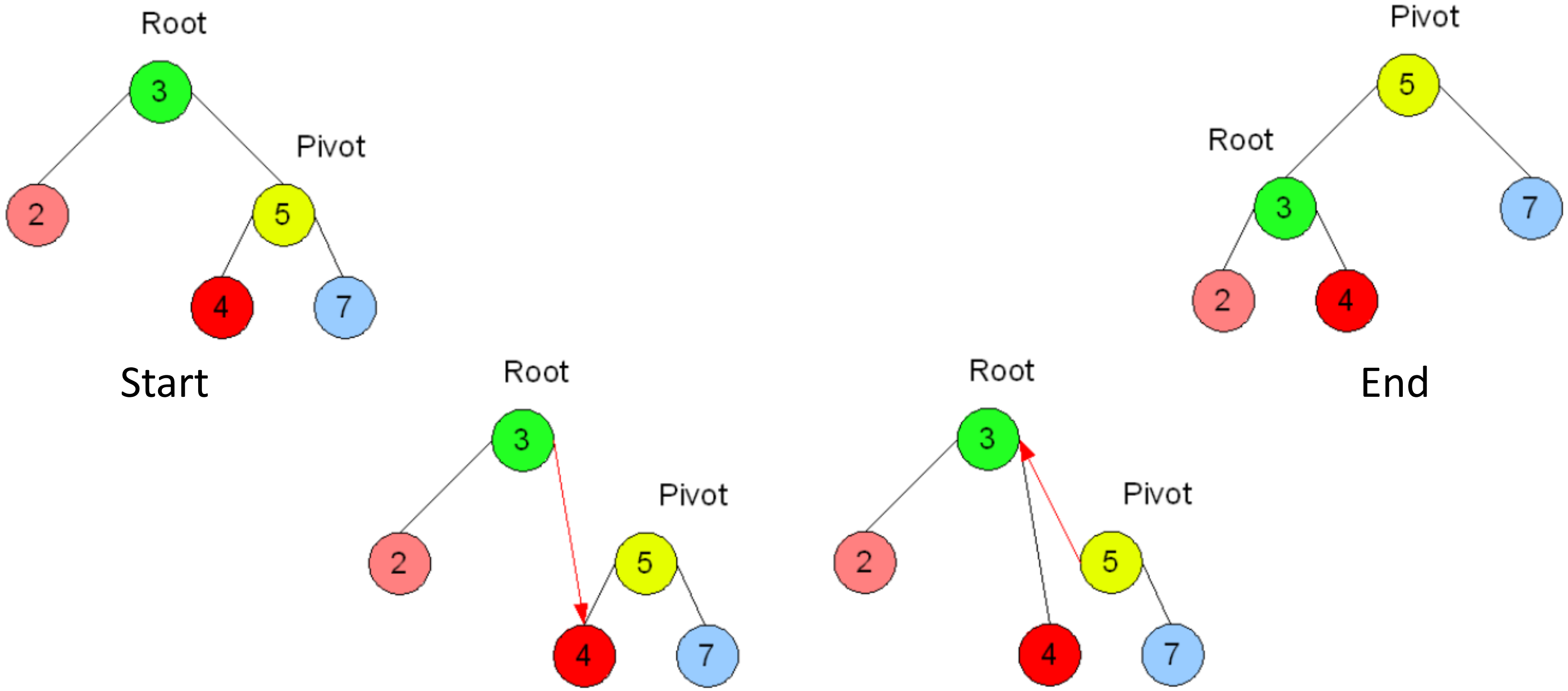
- Rotate left
  - ➢C++ code: To do





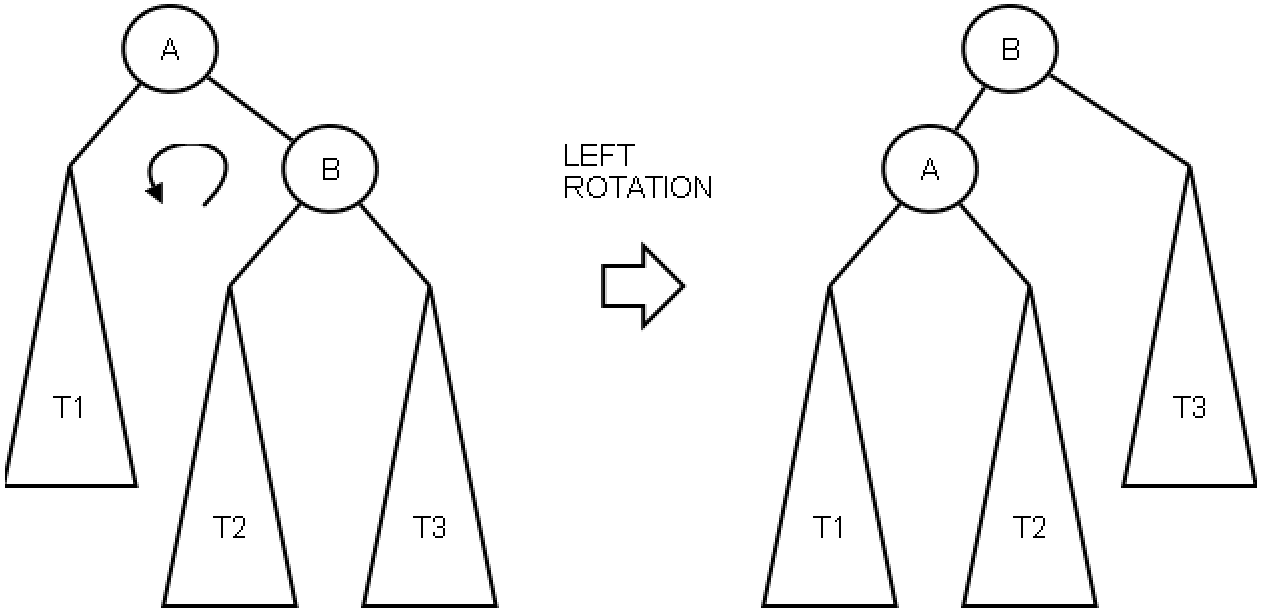
Rotation - Left

- What we want to do:



Rotation - Left

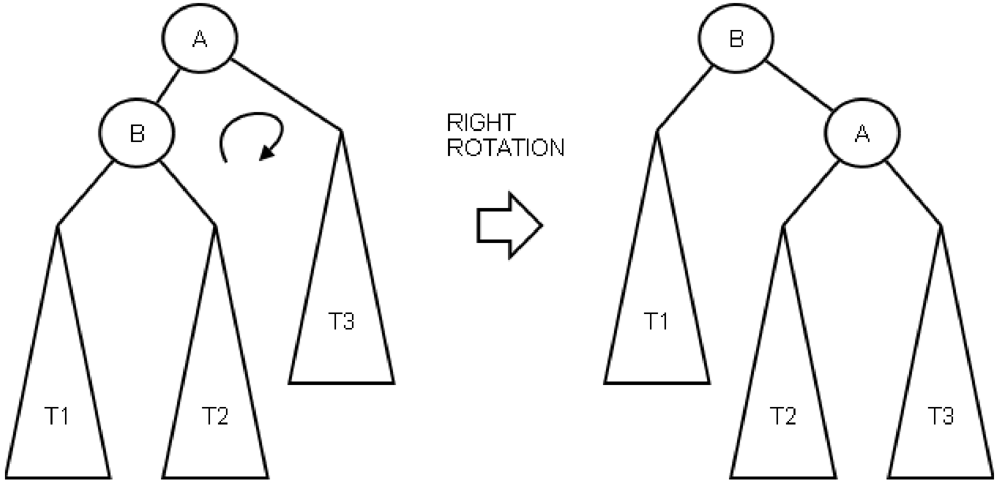
- Rotate left
  - C++ code: solution
    - Update links to children **and** parent

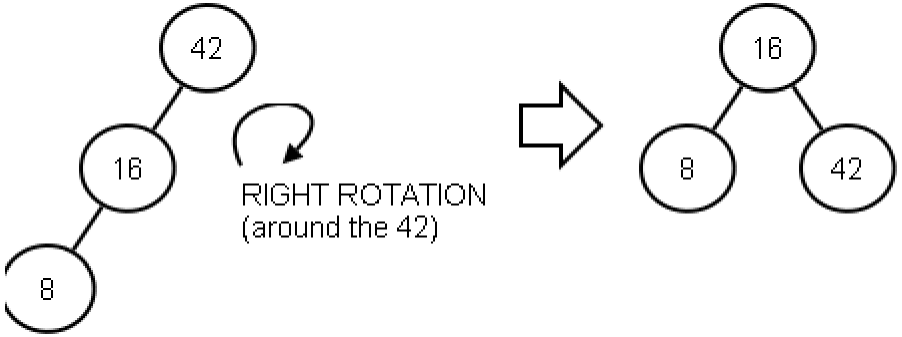



```
template <class T>
AVLnode<T>* AVLtree<T>::rotateLeft(AVLnode<T> *a) {
    AVLnode<T> *b = a->right;
    b->parent = a->parent;
    a->right = b->left;

    if (a->right != NULL)
        a->right->parent = a;

    b->left = a;
    a->parent = b;

    if (b->parent != NULL) {            // Where it was attached
        if (b->parent->right == a) {
            b->parent->right = b;
        }
        else {
            b->parent->left = b;
        }
    }

    setBalance(a);
    setBalance(b);
    return b;
}
```

Rotation - Right

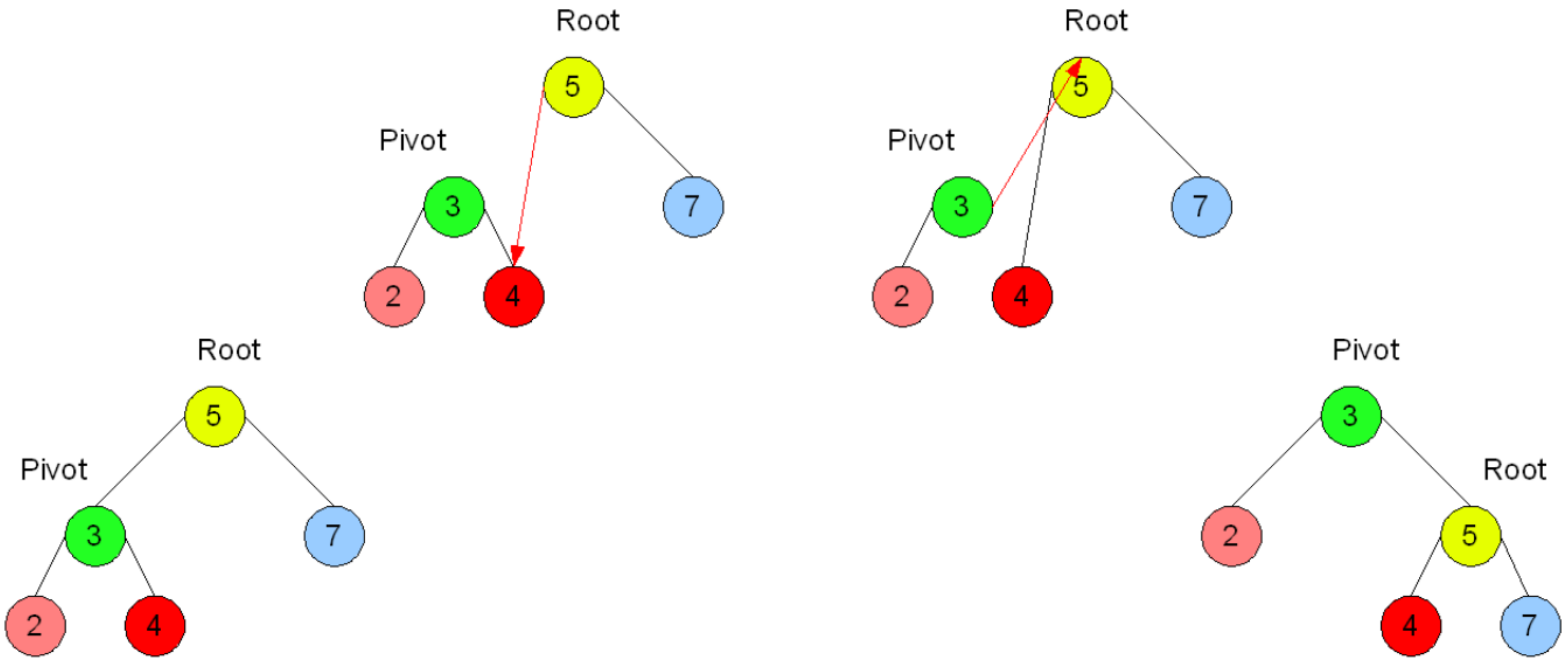
- Rotate right
  - ➢C++ code: To do





Rotation - Right

- What we want to do:

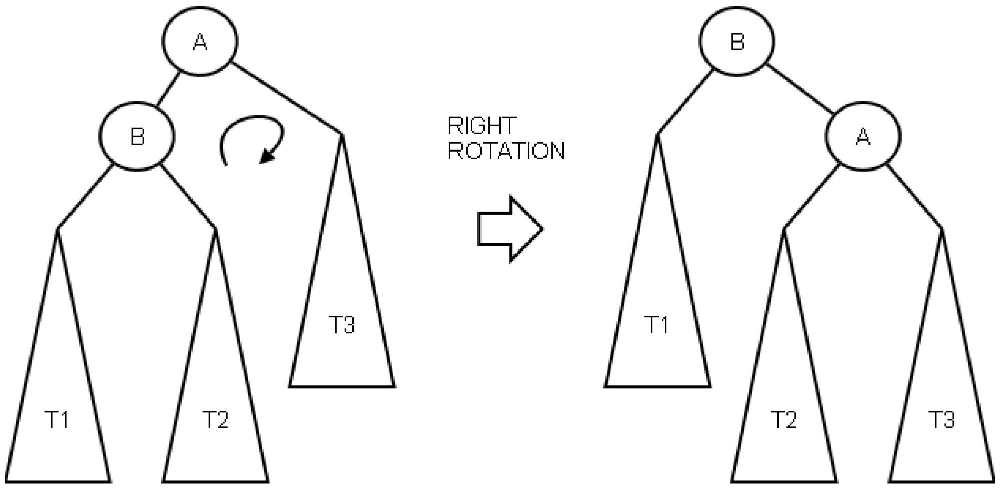


Start

End

Rotation - Right

- Rotate right
  - C++ code: solution
    - Update links to children **and** parent




```cpp
template <class T>
AVLnode<T>* AVLtree<T>::rotateRight(AVLnode<T> *a) {
    AVLnode<T> *b = a->left;
    b->parent = a->parent;
    a->left = b->right;

    if (a->left != NULL)
        a->left->parent = a;

    b->right = a;
    a->parent = b;

    if (b->parent != NULL) {                // Where it was attached
        if (b->parent->right == a) {
            b->parent->right = b;
        }
        else {
            b->parent->left = b;
        }
    }

    setBalance(a);
    setBalance(b);
    return b;
}
```

Rotation

- Combination
  - ➢ Left Then Right
  - ➢ Right Then Left

```
template <class T>
AVLnode<T>* AVLtree<T>::rotateLeftThenRight(AVLnode<T> *n) {
    n->left = rotateLeft(n->left);
    return rotateRight(n);
}

template <class T>
AVLnode<T>* AVLtree<T>::rotateRightThenLeft(AVLnode<T> *n) {
    n->right = rotateRight(n->right);
    return rotateLeft(n);
}
```

Balance Condition Violation

- If condition violated after a node insertion
  - Which nodes do we need to rotate?
  - Only nodes on path from insertion point to root may have their balance altered
- Rebalance the tree through rotation at the deepest node with balance violated
  - The entire tree will be rebalanced
- Violation cases at node k (deepest node)
  1. An insertion into **left** subtree of **left** child of k
  2. An insertion into **right** subtree of **left** child of k
  3. An insertion into **left** subtree of **right** child of k
  4. An insertion into **right** subtree of **right** child of k
  - **Cases 1 and 4** equivalent
    - Single rotation to rebalance
  - **Cases 2 and 3** equivalent
    - Double rotation to rebalance

Rebalance the tree

- Code C++

```cpp
template <class T>
void AVLtree<T>::rebalance(AVLnode<T> *n) {
    setBalance(n);

    if (n->balance == -2) {
        if (height(n->left->left) >= height(n->left->right))
            n = rotateRight(n);
        else
            n = rotateLeftThenRight(n);
    }
    else if (n->balance == 2) {
        if (height(n->right->right) >= height(n->right->left))
            n = rotateLeft(n);
        else
            n = rotateRightThenLeft(n);
    }

    if (n->parent != NULL) {
        rebalance(n->parent);
    }
    else {
        root = n;
    }
}
```

Case 1

Case 2

Case 4

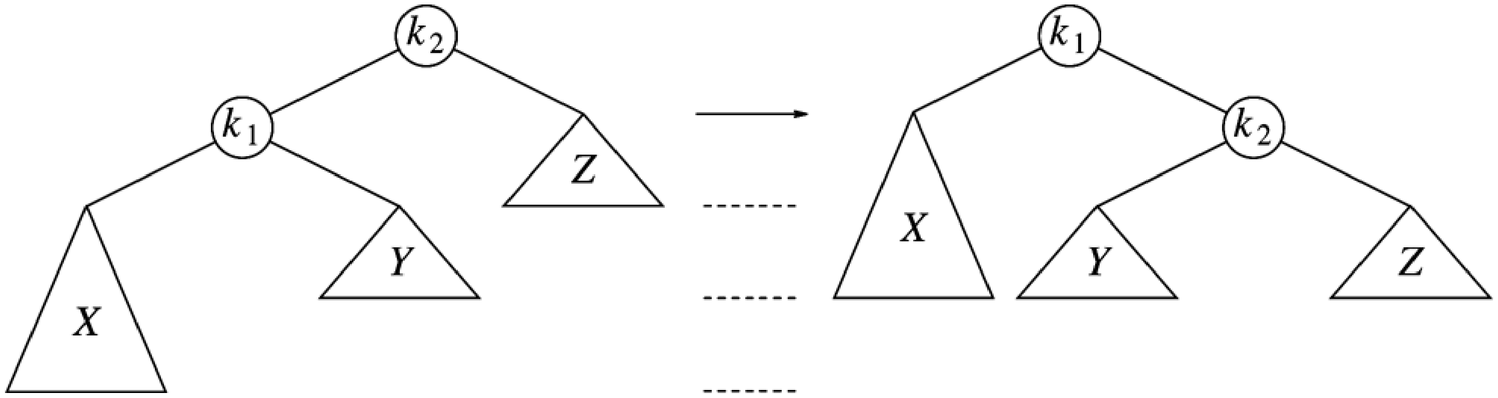
Case 3

AVL Trees Complexity

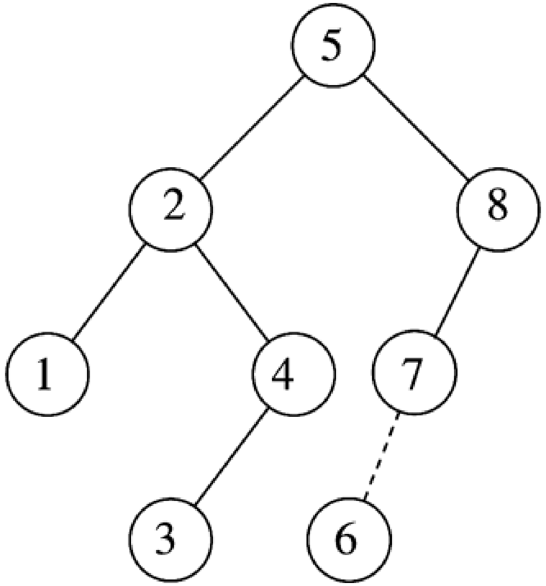
- Overhead
  - Extra space for maintaining height information at each node
  - Insertion and deletion become more complicated, but still O(log N)
- Advantage
  - Worst case O(log(N)) for insert, delete, and search

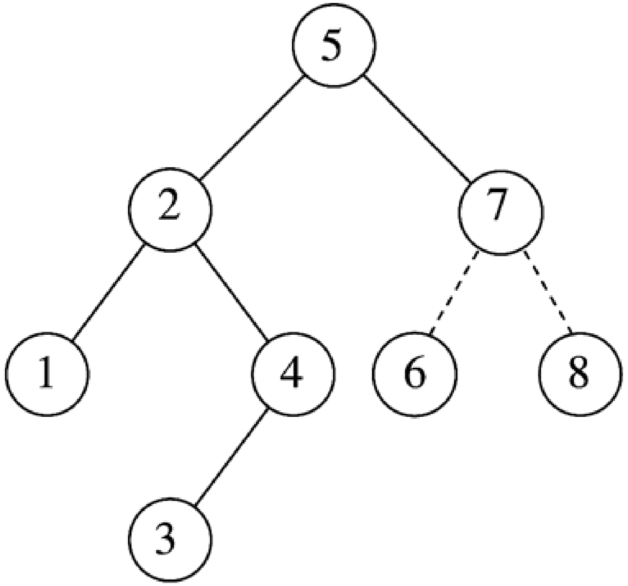
Single Rotation (Case 1)

- Replace node k_2 by node k_1
- Set node k_2 to be right child of node k_1
- Set subtree Y to be left child of node k_2
- Case 4 is similar



Example



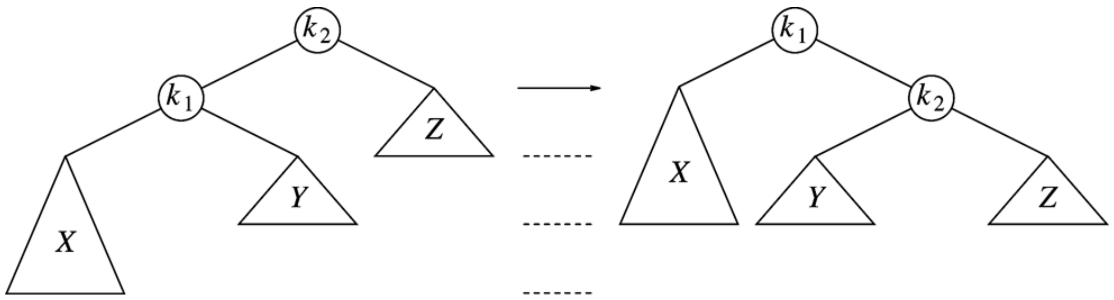


- After inserting 6
  - Balance condition at node 8 is violated

Single Rotation (Case 1)

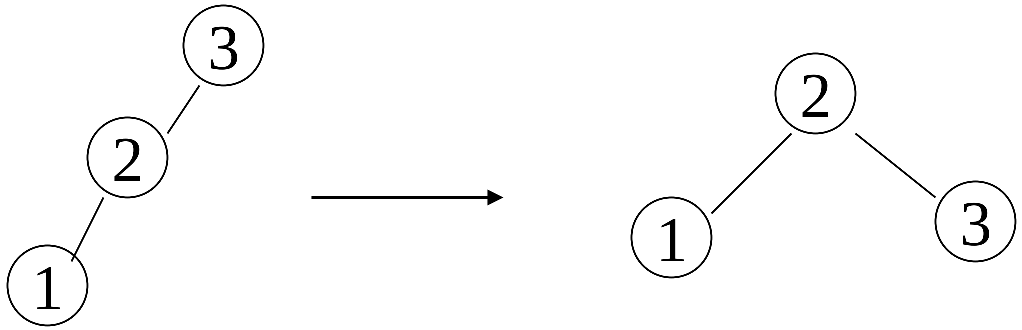
- Pseudo-code

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
void rotateWithLeftChild( AvlNode * & k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}
```



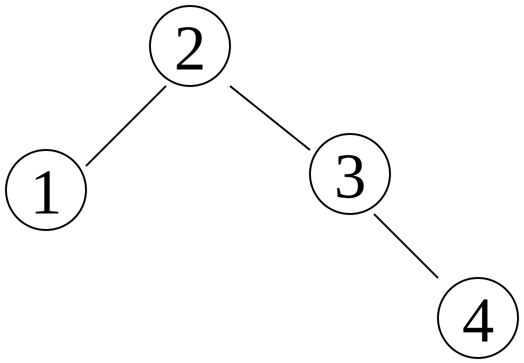
Example

- Inserting 3, 2, 1, and then 4 to 7 sequentially into empty AVL tree

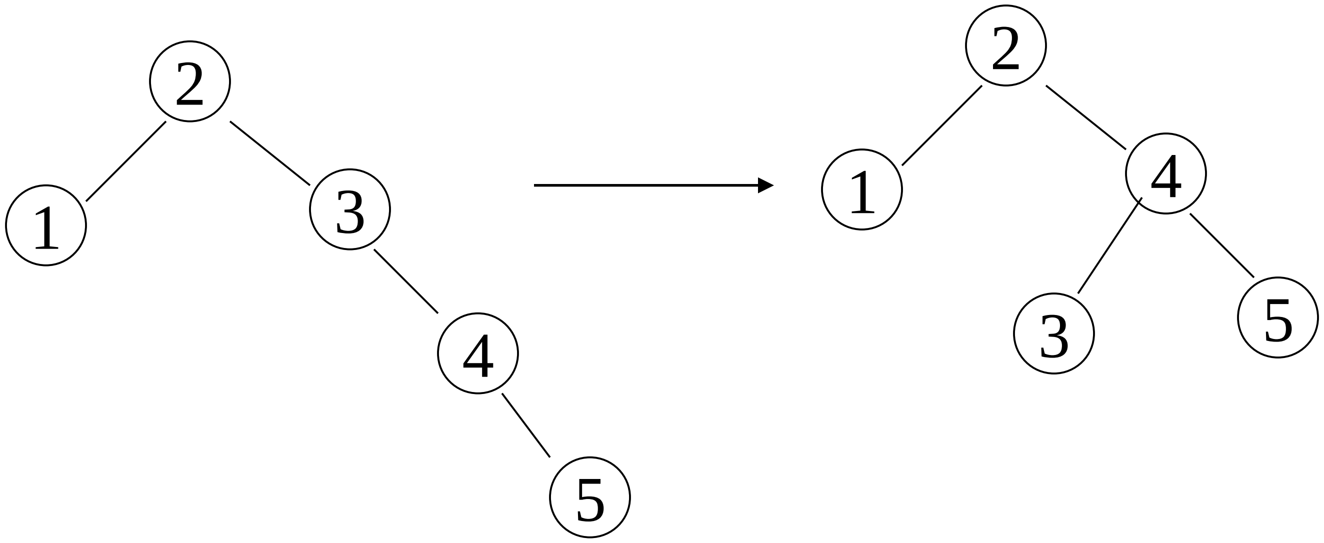


Example

- Inserting 4

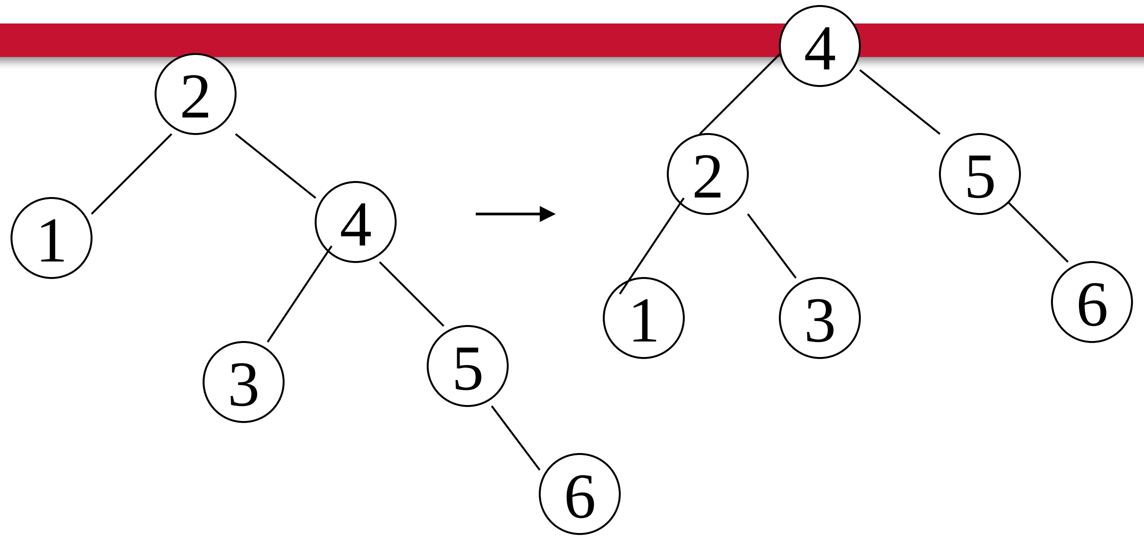


- Inserting 5

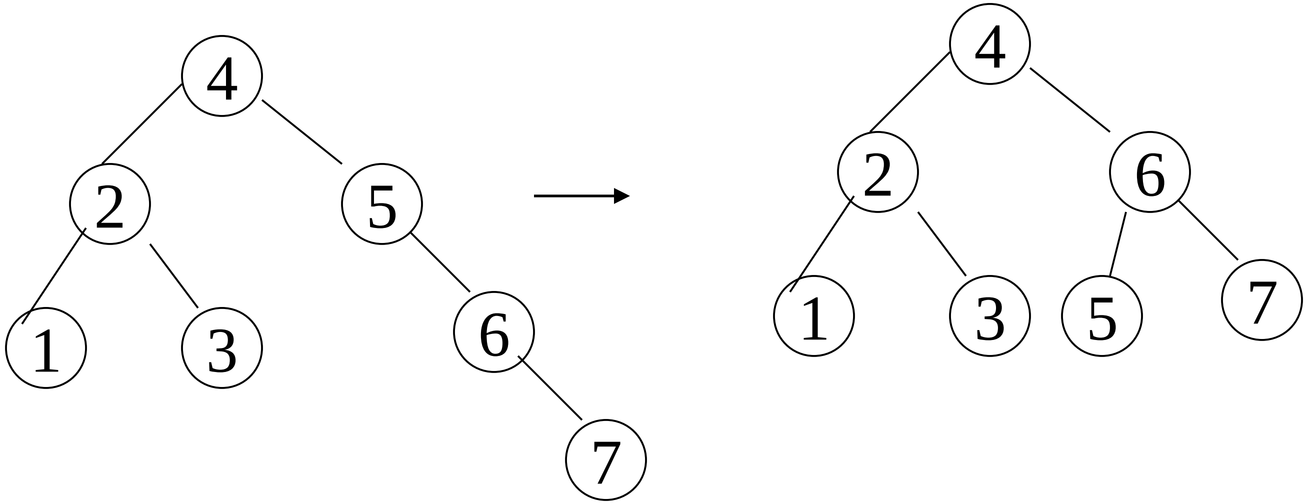


Example

- Inserting 6

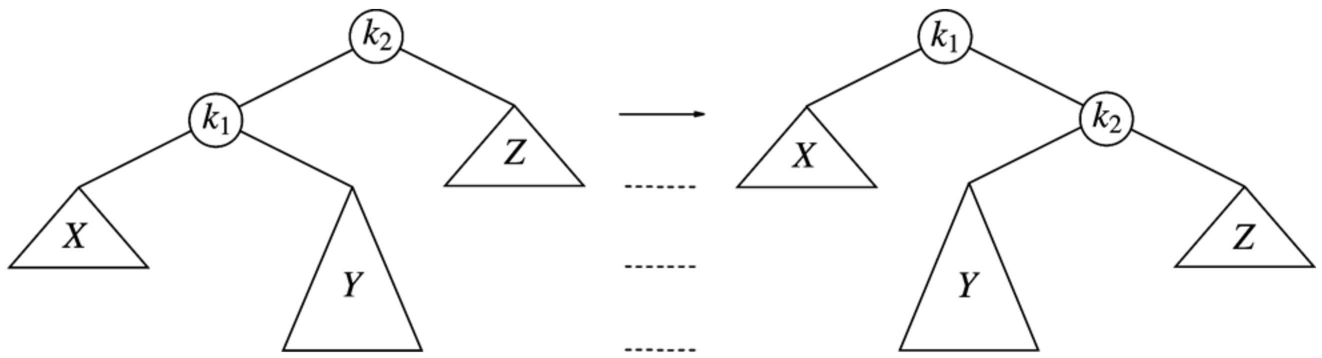


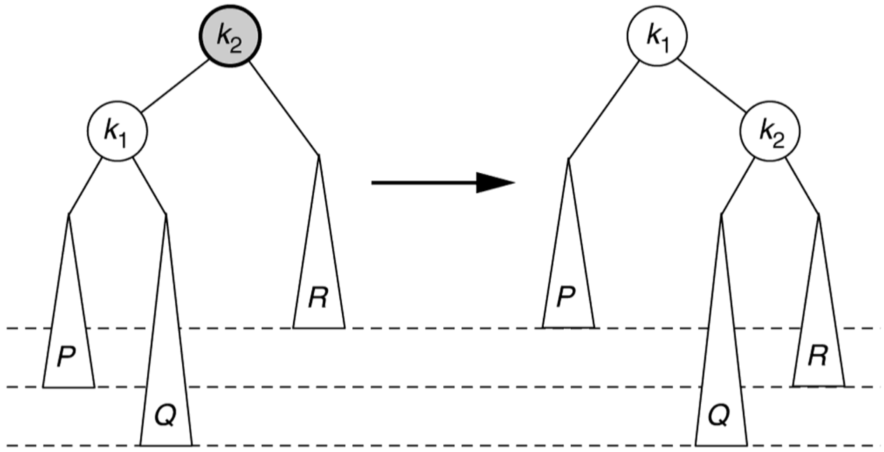
- Inserting 7



Single Rotation Will Not Work for the Other Case

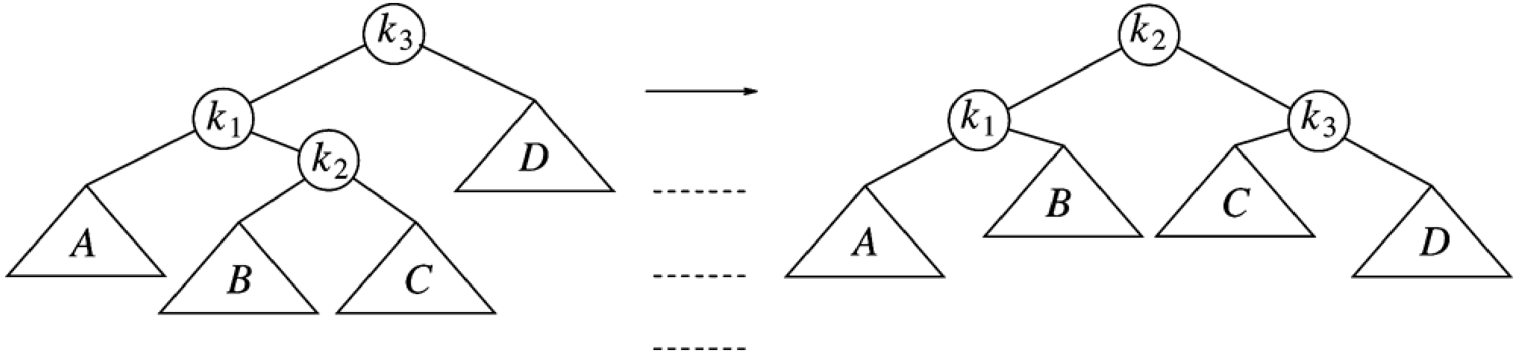
- For case 2
- After single rotation, k_1 still not balanced ☹
- **Double** rotations needed for case 2 and case 3





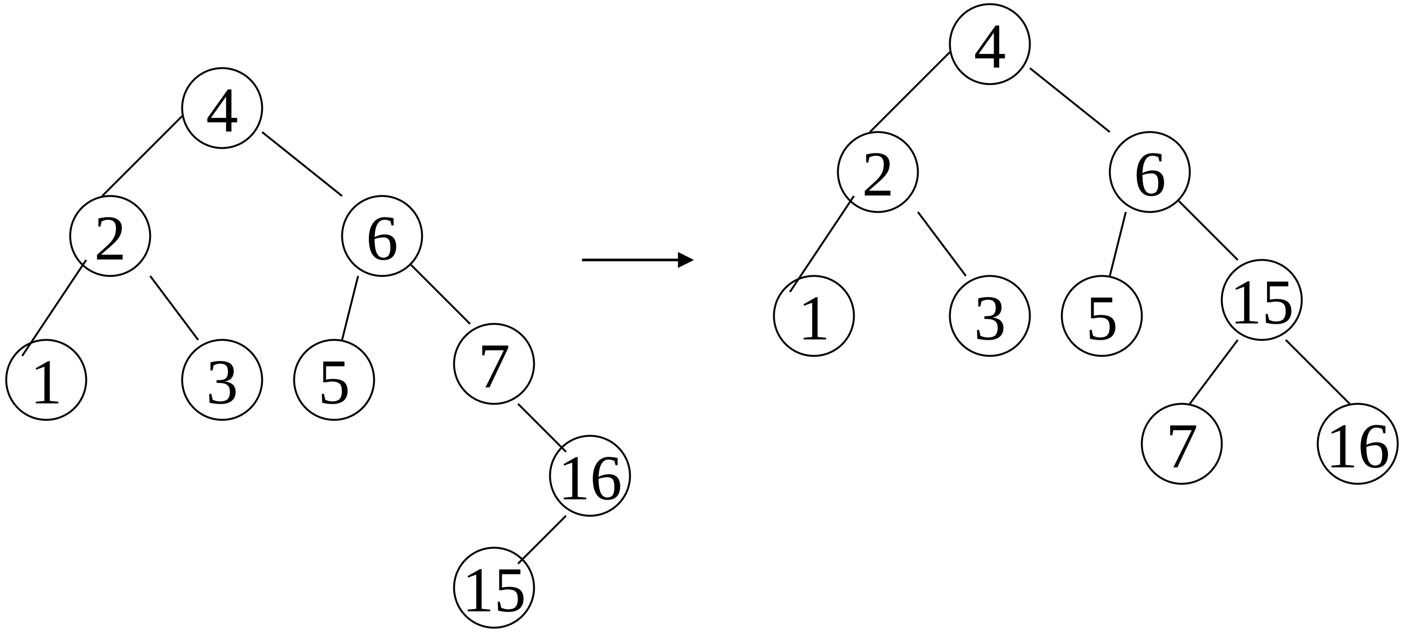
Double Rotation (Case 2)

- Method
  - ➢Left-right double rotation to fix case 2
  - ➢First rotate between $\mathbf{k_1}$ and $\mathbf{k_2}$
  - ➢Then rotate between $\mathbf{k_2}$ and $\mathbf{k_3}$
- Case 3 is similar



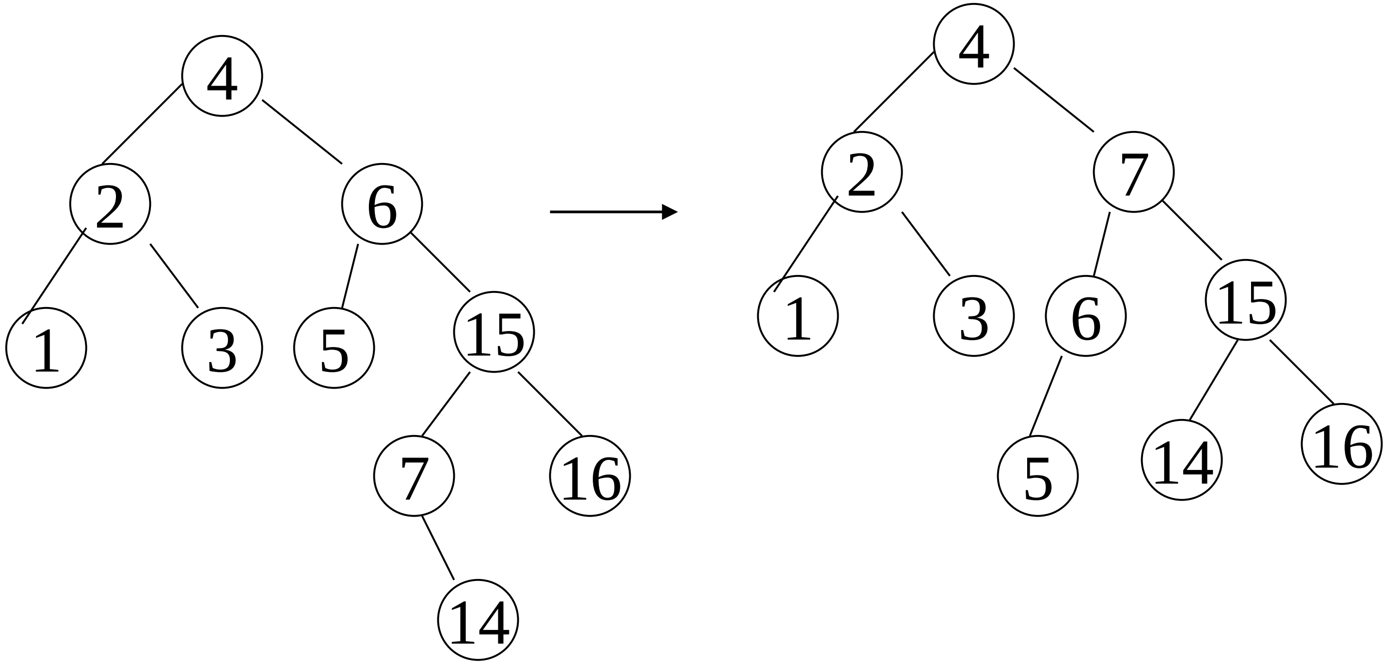
Example

- Continuing the previous example by inserting
  - 16 down to 10, and then 8 and 9
- Inserting 16 and 15



Example

- Inserting 14

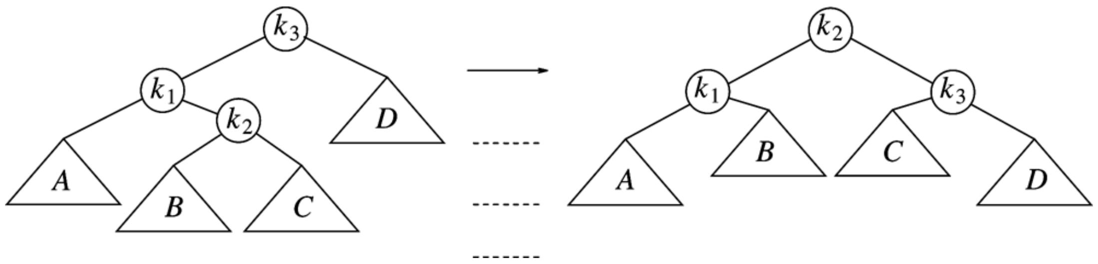


- ...

Double Rotation (Case 2)

- Pseudo code

```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```



Summary

- Different cases
  - Violation cases at node k (deepest node)
    1. An insertion into **left** subtree of **left** child of k
    2. An insertion into **right** subtree of **left** child of k
    3. An insertion into **left** subtree of **right** child of k
    4. An insertion into **right** subtree of **right** child of k

Implementation of AVL Tree

- Pseudo code

```
struct AvlNode
{
    Comparable element;
    AvlNode   *left;
    AvlNode   *right;
    int       height;

    AvlNode( const Comparable & theElement, AvlNode *lt,
                                            AvlNode *rt, int h = 0 )
      : element( theElement ), left( lt ), right( rt ), height( h ) { }
};
```

```
/**
 * Return the height of node t or -1 if NULL.
 */
int height( AvlNode *t ) const
{
    return t == NULL ? -1 : t->height;
}
```

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, AvlNode * & t )
{
    if( t == NULL )
        t = new AvlNode( x, NULL, NULL );
    else if( x < t->element )
    {
        insert( x, t->left );
        if( height( t->left ) - height( t->right ) == 2 )
            if( x < t->left->element )
                rotateWithLeftChild( t );          Case 1
            else
                doubleWithLeftChild( t );          Case 2
    }
    else if( t->element < x )
    {
        insert( x, t->right );
        if( height( t->right ) - height( t->left ) == 2 )
            if( t->right->element < x )
                rotateWithRightChild( t );         Case 4
            else
                doubleWithRightChild( t );         Case 3
    }
    else
        ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

Implementation of AVL Tree

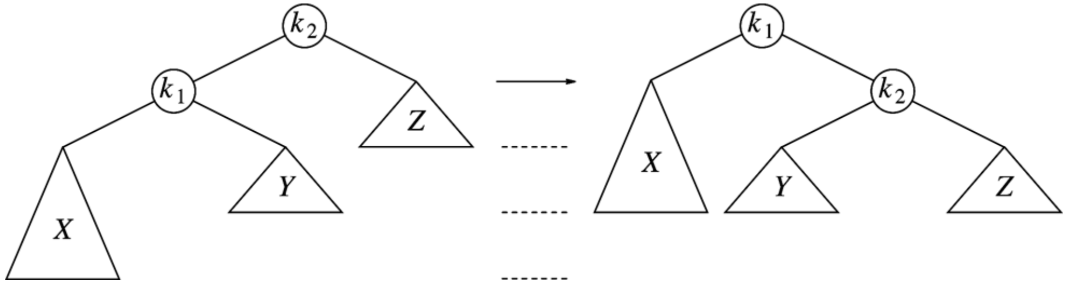
- C++

```
template <class T>
bool AVLtree<T>::insert(T key) {
    if (root == NULL) {
        root = new AVLnode<T>(key, NULL);
    }
    else {
        AVLnode<T>
            *n = root,
            *parent;
        while (true) {
            if (n->key == key)
                return false;
            parent = n;
            bool goLeft = n->key > key;
            n = goLeft ? n->left : n->right;
            if (n == NULL) {
                if (goLeft) {
                    parent->left = new AVLnode<T>(key, parent);
                }
                else {
                    parent->right = new AVLnode<T>(key, parent);
                }
                rebalance(parent);
                break;
            }
        }
    }
    return true;
}
```

Single Rotation (Case 1)

- Pseudo code

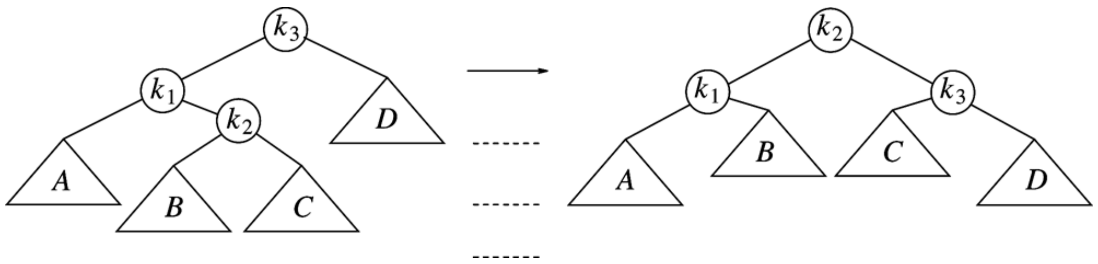
```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
void rotateWithLeftChild( AvlNode * & k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}
```



Double Rotation (Case 2)

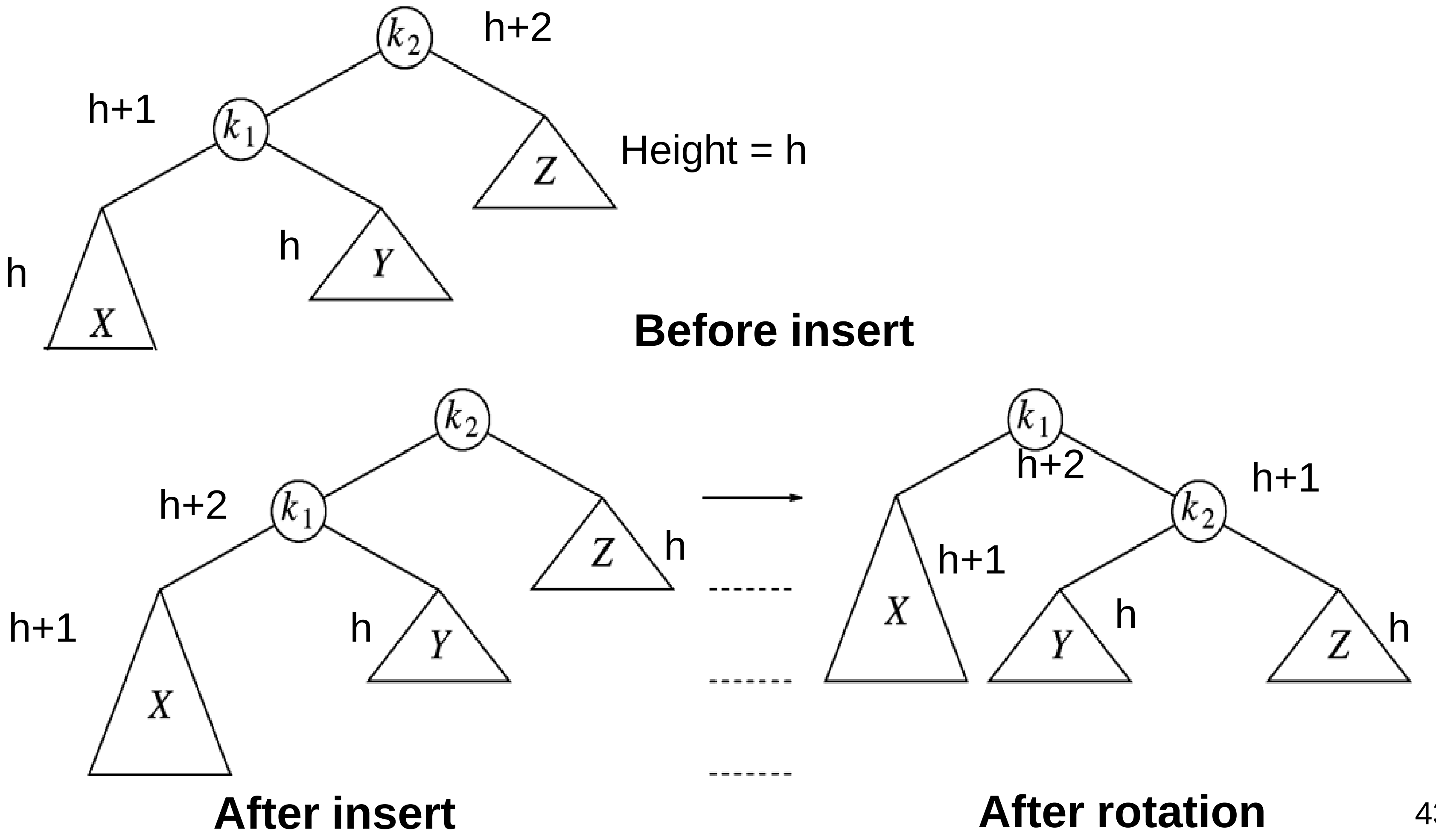
- Pseudo code

```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```



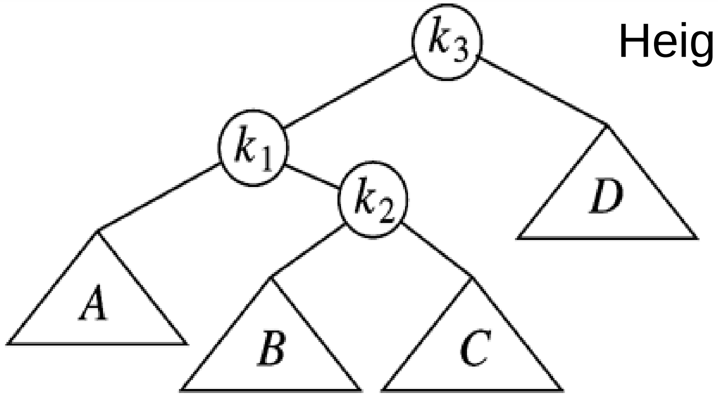
Insertion: Case 1

- .



Insertion: Case 2

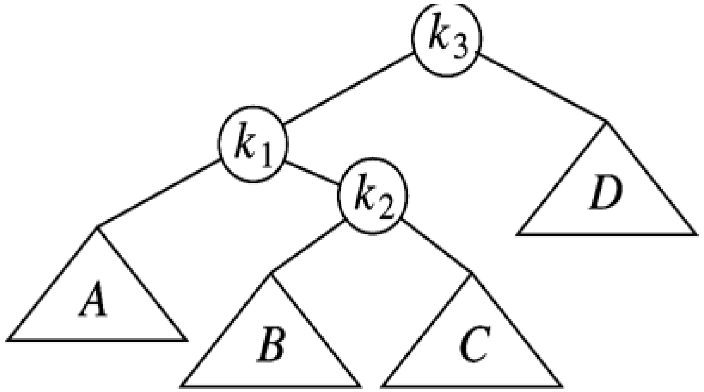
- .

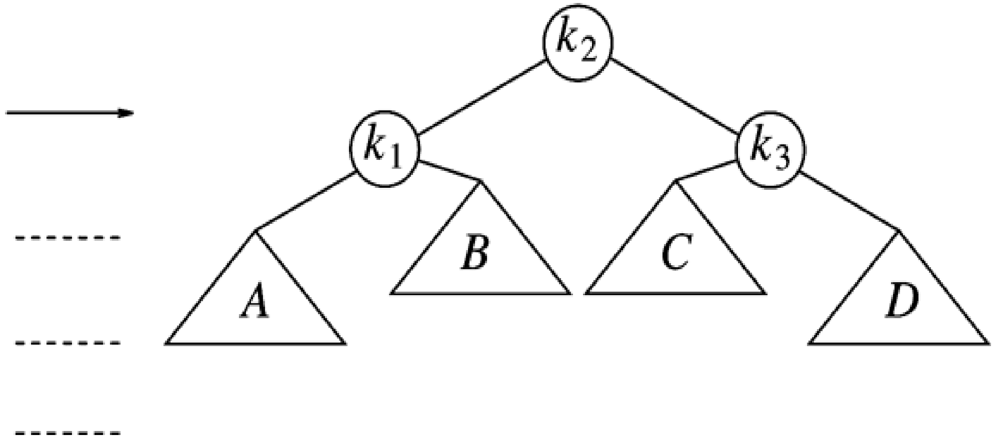


Height = h

Determine all heights

Before insert



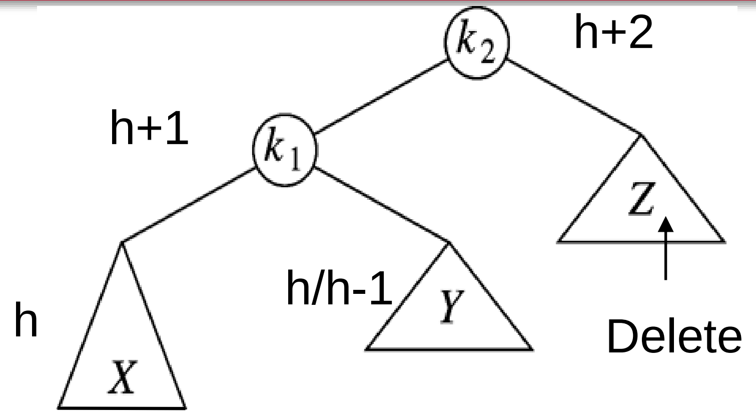


After insert

After double rotation

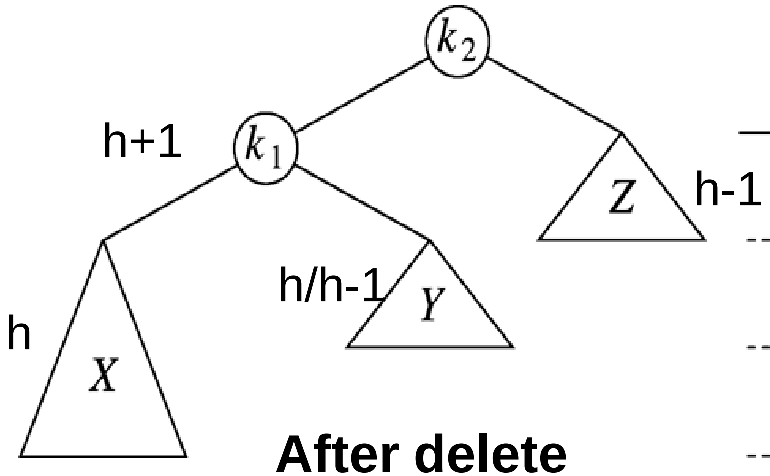
Delete: Case 1

- Consider deepest unbalanced node
  - Case 1:
    - Left child’s left side is too high
  - Case 4:
    - Right child’s right side is too high
  - The parents may need to be recursively rotated

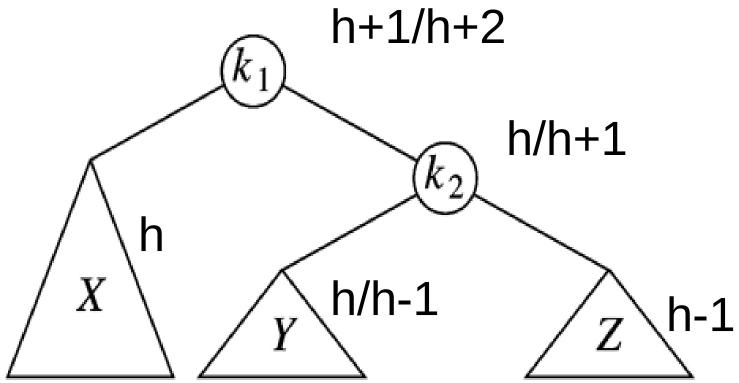


Height = h

Before Deletion



After delete

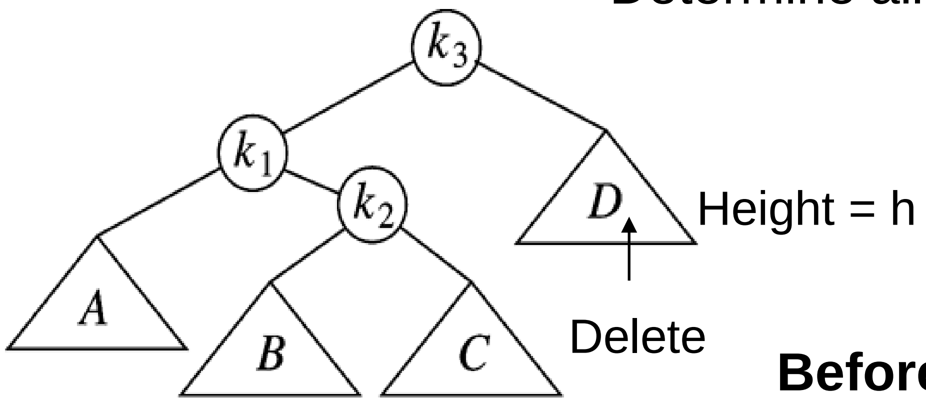


After single rotation

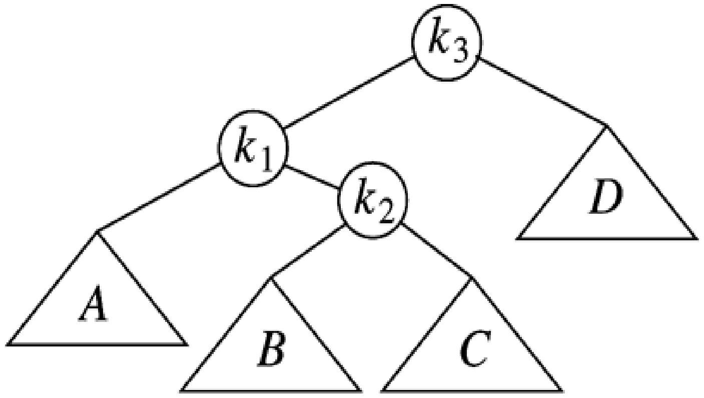
Delete: Case 2

- Consider deepest unbalanced node
  - Case 2:
    - Left child’s right side is too high
  - Case 3:
    - Right child’s left side is too high
  - The parents may need to be recursively rotated

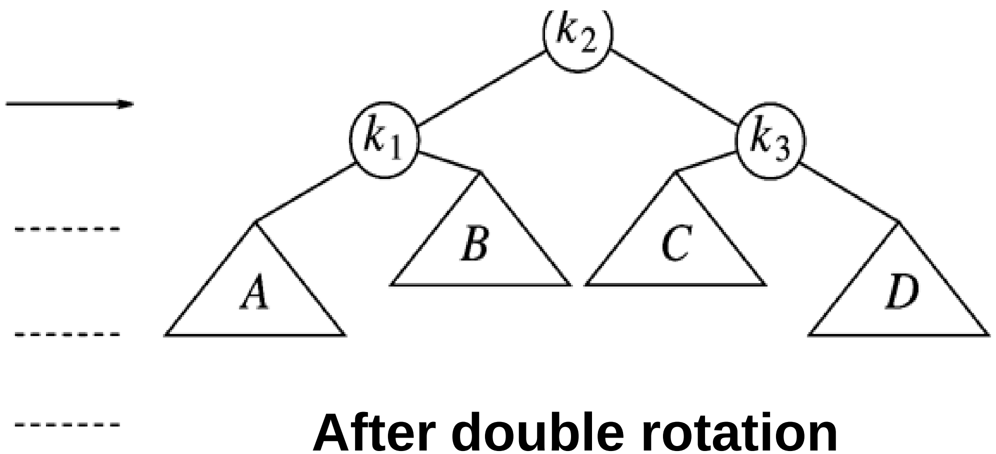
Determine all heights



Before Deletion



After Delete



After double rotation

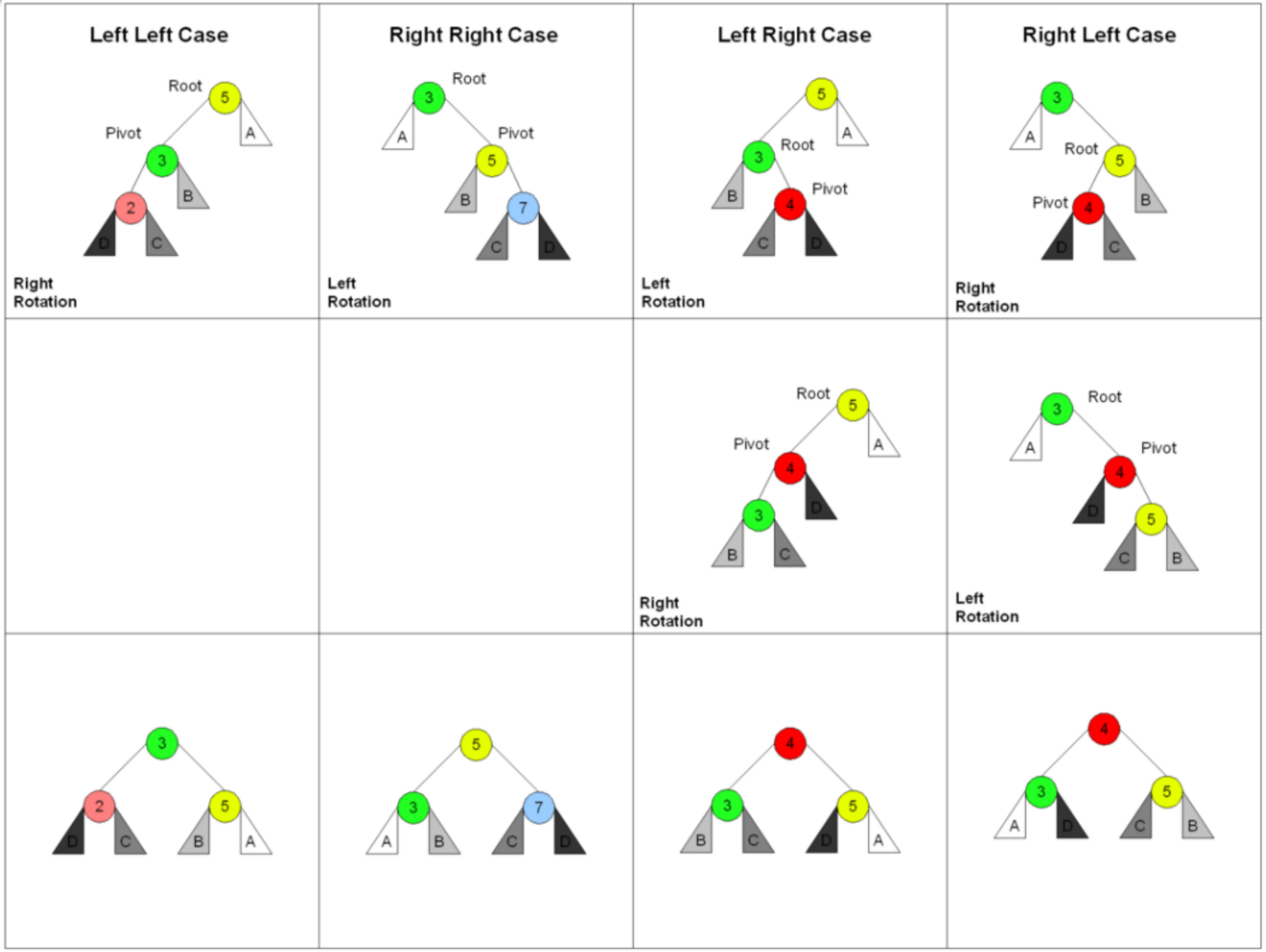
Delete

- Code C++

```cpp
template <class T>
void AVLtree<T>::deleteKey(const T delKey) {
    if (root == NULL)
        return;
    AVLnode<T>  *n = root, *parent = root, *delNode = NULL, *child = root;
    while (child != NULL) {
        parent = n;
        n = child;
        child = delKey >= n->key ? n->right : n->left;
        if (delKey == n->key)
            delNode = n;
    }
    if (delNode != NULL) {
        delNode->key = n->key;
        child = n->left != NULL ? n->left : n->right;
        if (root->key == delKey) {
            root = child;
        }
        else {
            if (parent->left == n)
                parent->left = child;
            else
                parent->right = child;
            rebalance(parent);
        }
    }
}
```

Rotations in a single slide

- .



Conclusion

- AVL Trees
  - A way to recover from unbalanced binary tree !
- Complexity
  - AVL tree vs. BST vs. Skip list

AVL tree

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	$\boxed{\text{O}(\log n)}$
Insert	O(log n)	$\boxed{\text{O}(\log n)}$
Delete	O(log n)	$\boxed{\text{O}(\log n)}$

BST

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Skip list

Because of the size of the nodes

Algorithm	Average	Worst case
Space	O(n)	O(n log n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

  - Advantage for the worst case !!!
- Question
  - Does rotations change in-order traversal?

- TO DO
  - Implement AVL trees at home

Questions ?

- Reading
  - On Canvas: Csci 115 book – Section 7.2
  - Introduction to Algorithms, 3rd Edition.