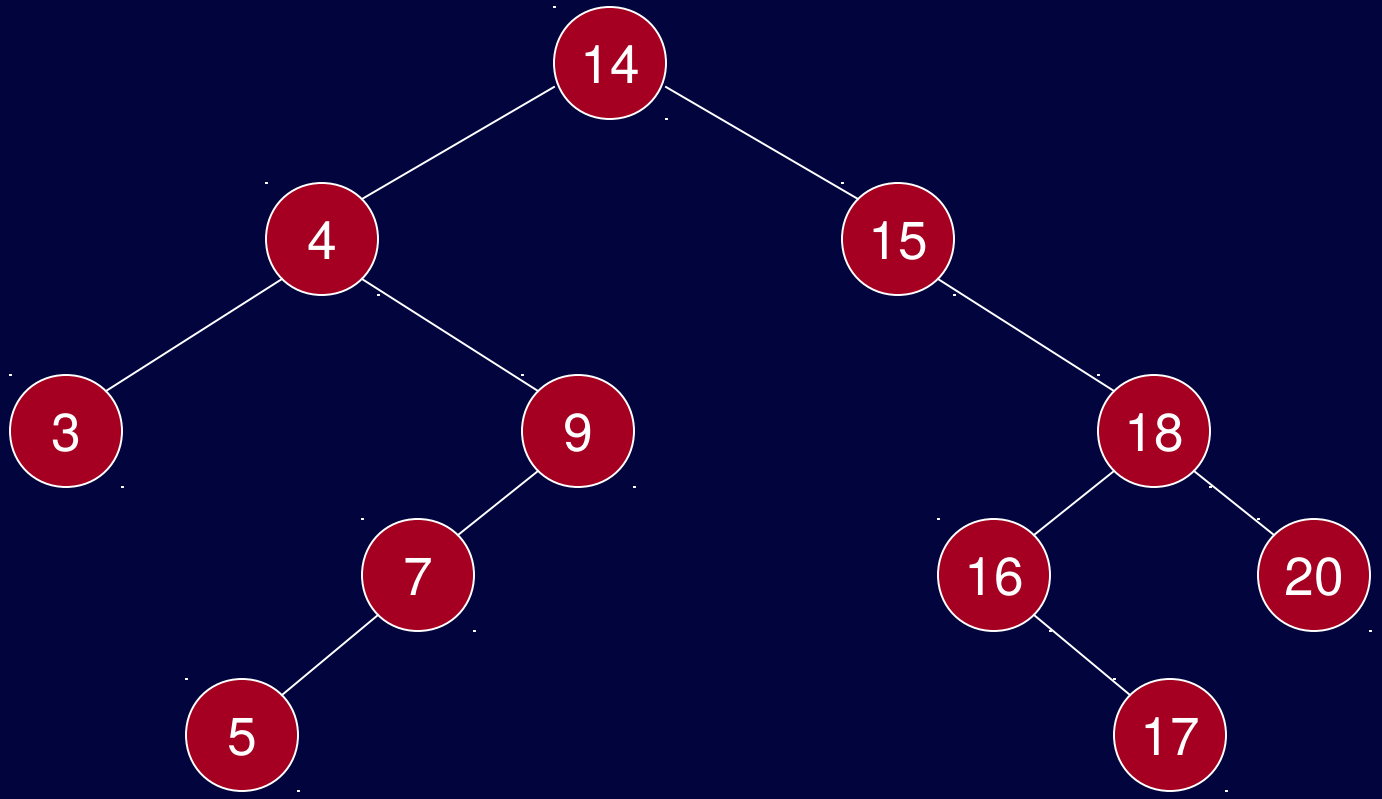
Lecture No.16
AVL Trees

CC-213 Data Structures
Department of Computer Science
University of the Punjab

Slides modified very slightly from the late Dr. Sohail Aslam's lectures at VU

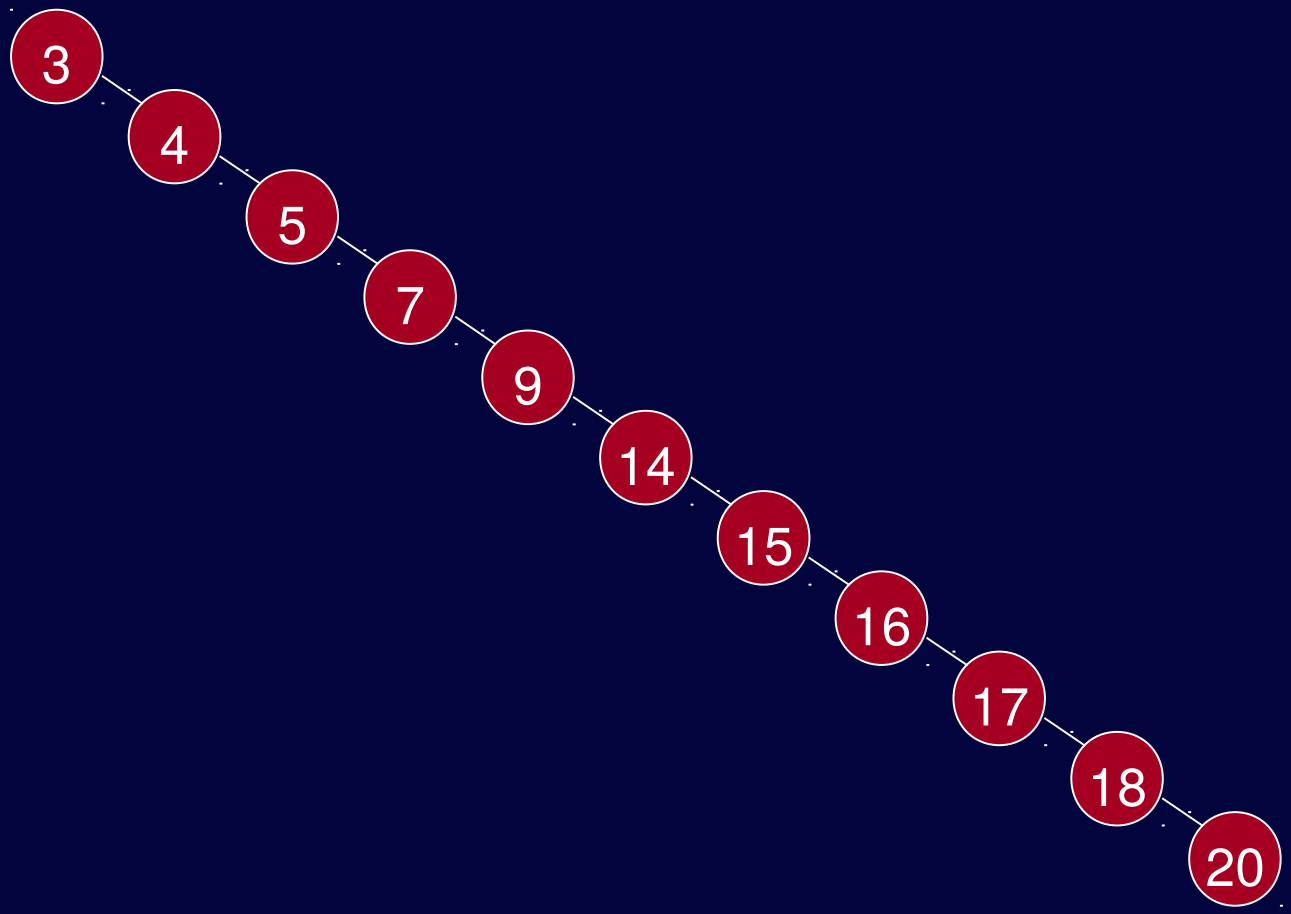
Degenerate Binary Search Tree

BST for 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17



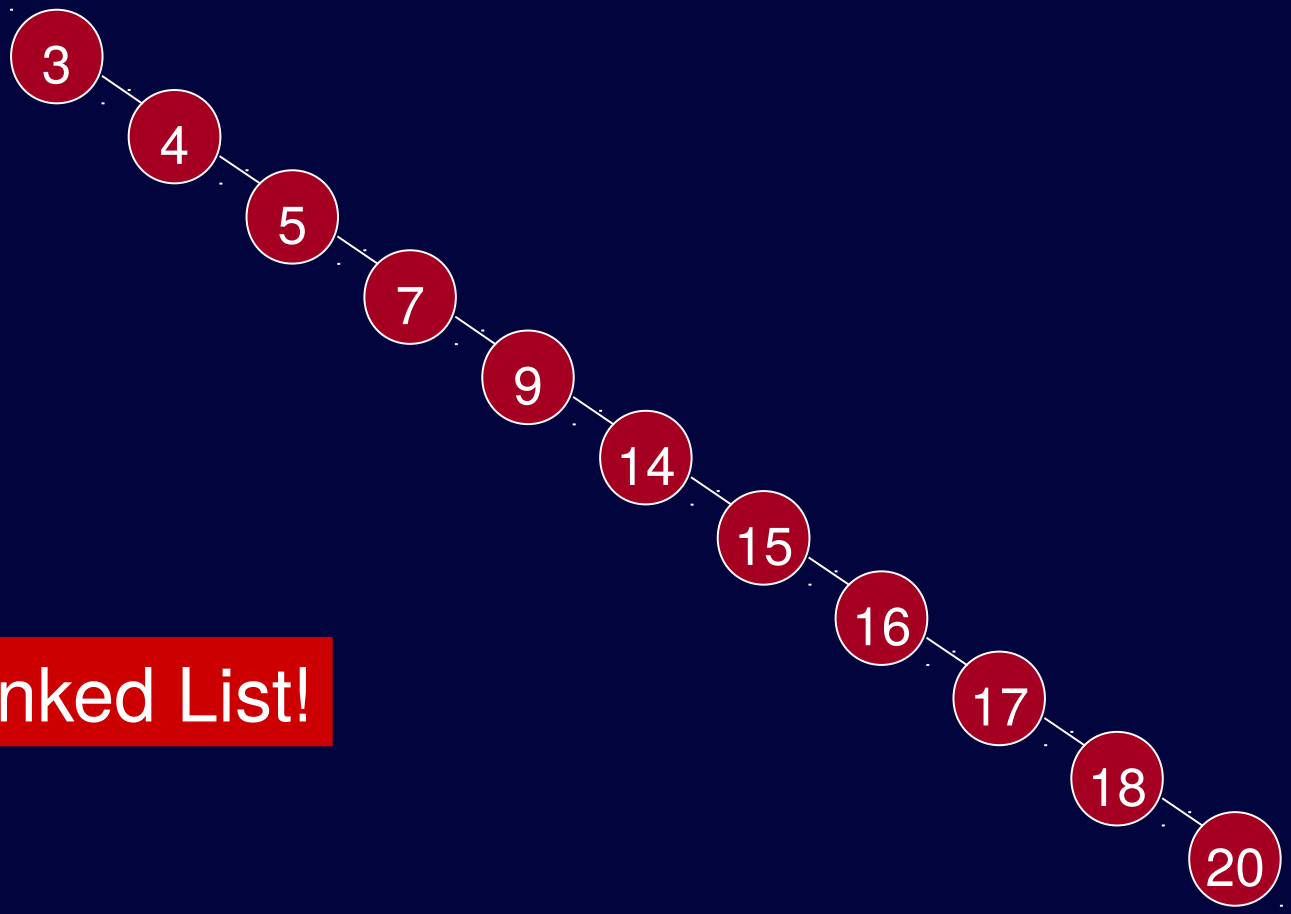
Degenerate Binary Search Tree

BST for 3 4 5 7 9 14 15 16 17 18 20



Degenerate Binary Search Tree

BST for 3 4 5 7 9 14 15 16 17 18 20

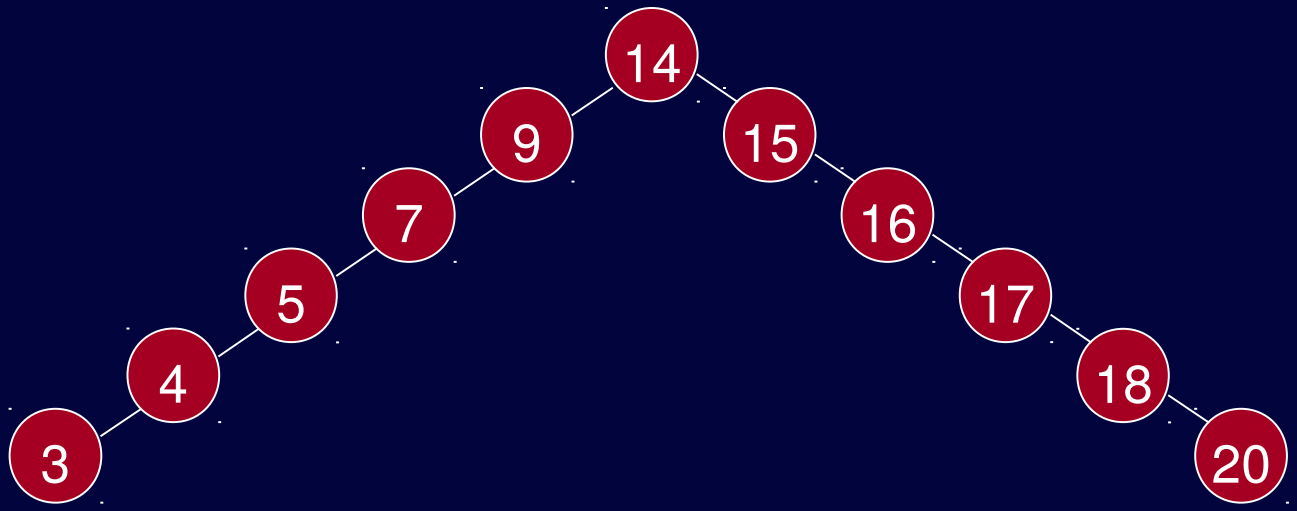


Linked List!

Balanced BST

- We should keep the tree *balanced*.
- One idea would be to have the left and right subtrees have the same height

Balanced BST



Does not force the tree to be *shallow*.

Balanced BST

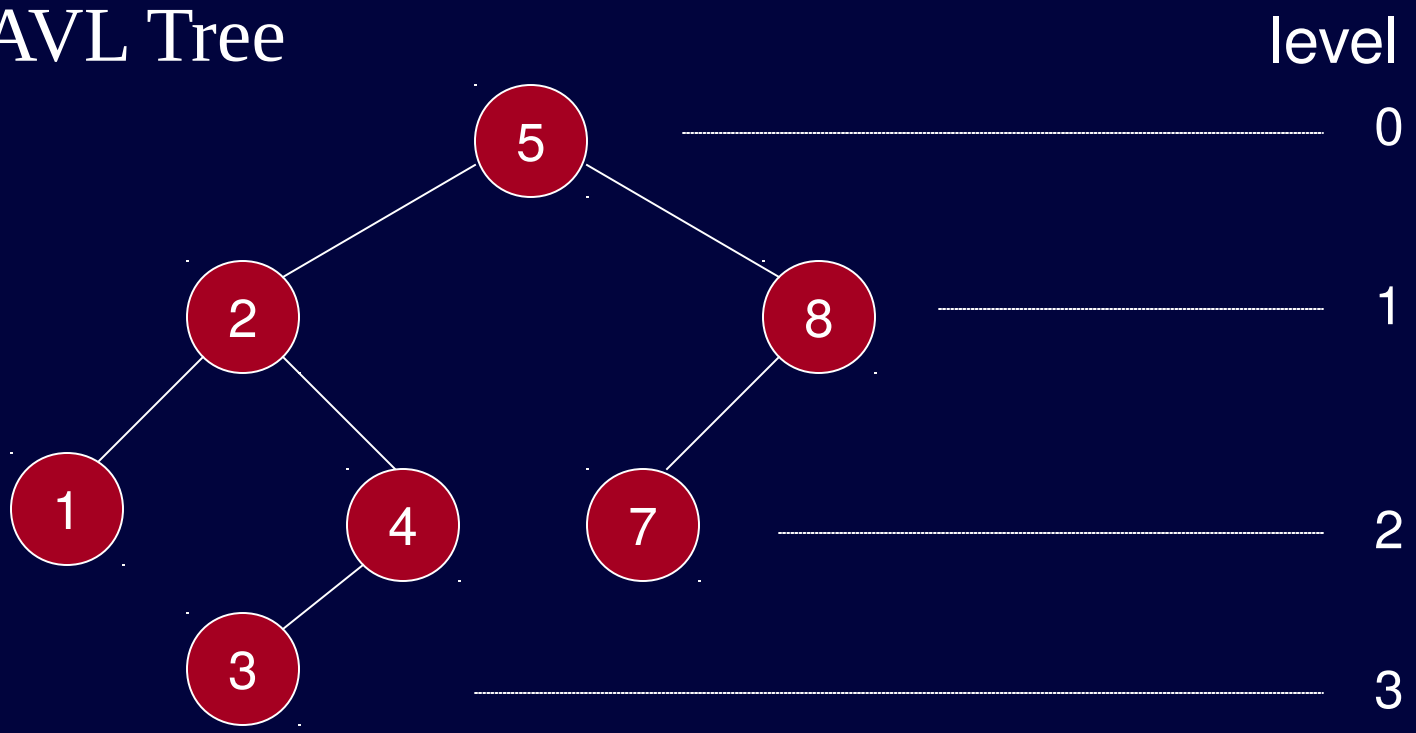
- We could insist that every node must have left and right subtrees of same height.
- But this requires that the tree be a complete binary tree
- To do this, there must have $(2^{d+1} - 1)$ data items, where d is the depth of the tree.
- This is too rigid a condition.

AVL Tree

- AVL (Adelson-Velskii and Landis) tree.
- An AVL tree is identical to a BST except
  - height of the left and right subtrees can differ by at most 1.
  - height of an empty tree is defined to be (–1).

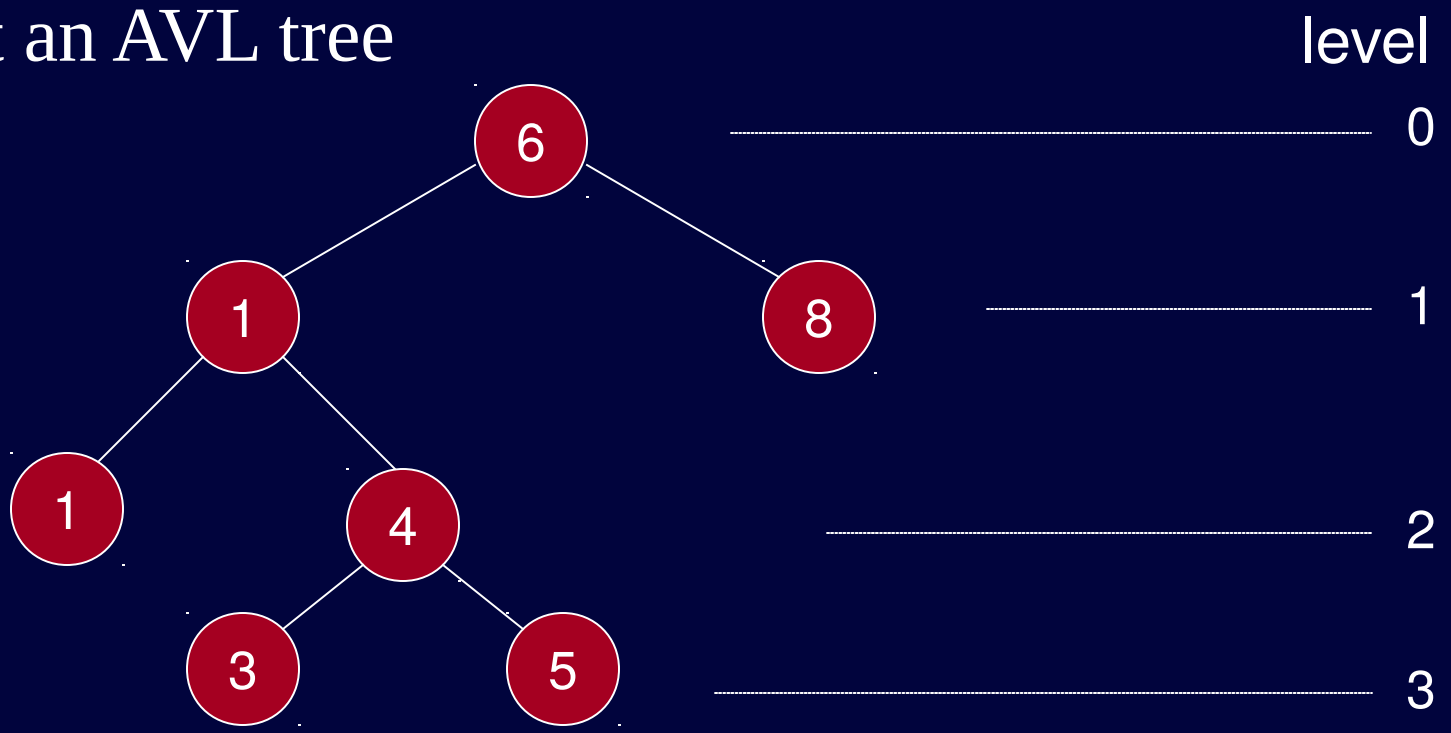
AVL Tree

- An AVL Tree



AVL Tree

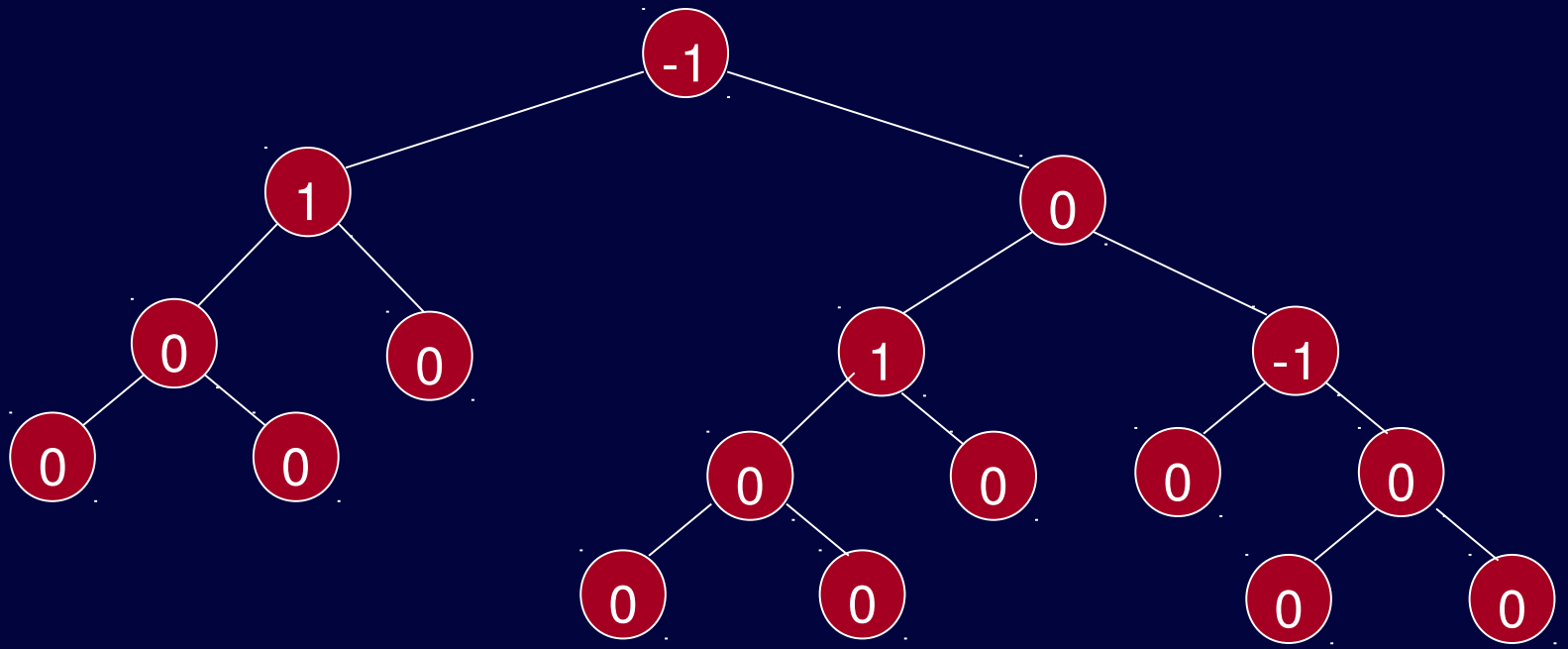
- Not an AVL tree



Balanced Binary Tree

- The *height* of a binary tree is the maximum level of its leaves (also called the depth).
- The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.
- Here, for example, is a balanced tree. Each node has an indicated balance of 1, 0, or –1.

Balanced Binary Tree

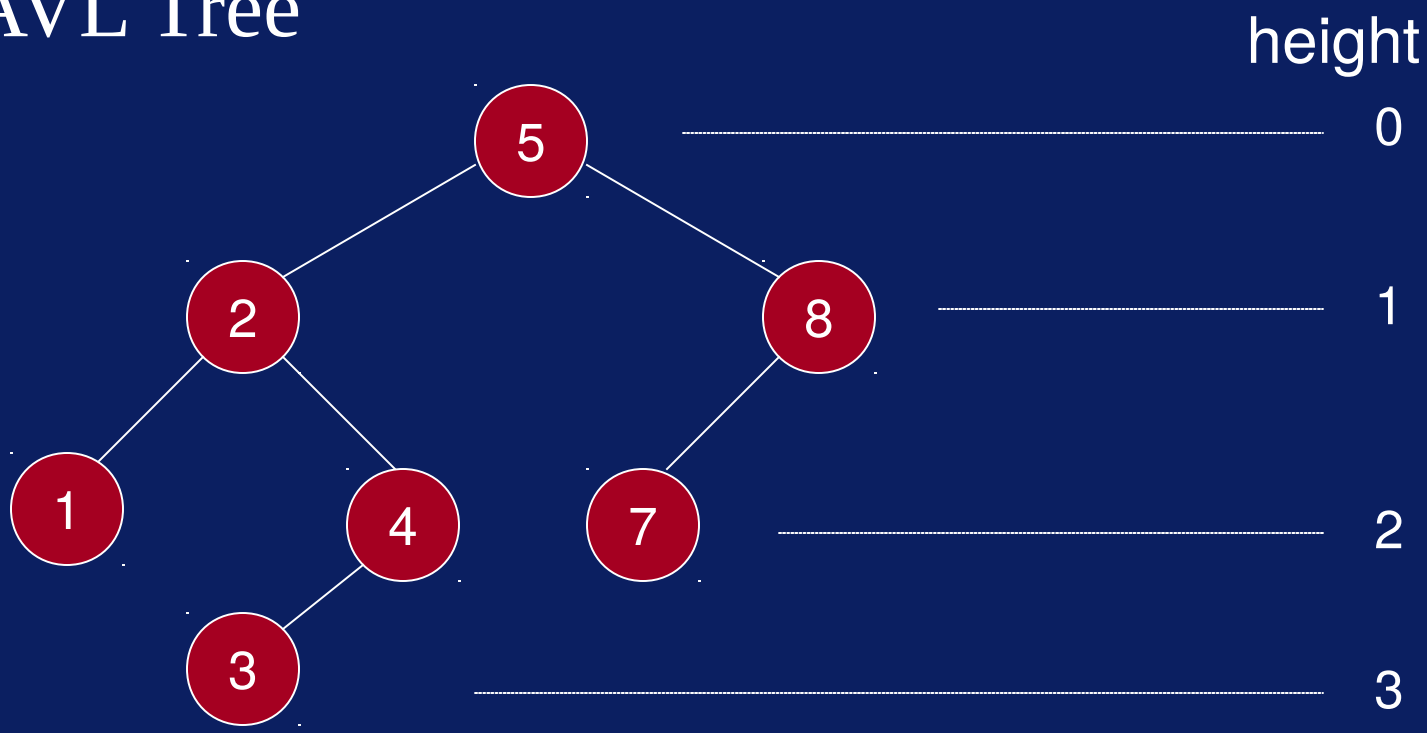


AVL Tree

- AVL (Adelson-Velskii and Landis) tree.
- An AVL tree is identical to a BST except
  - height of the left and right subtrees can differ by at most 1.
  - height of an empty tree is defined to be (–1).

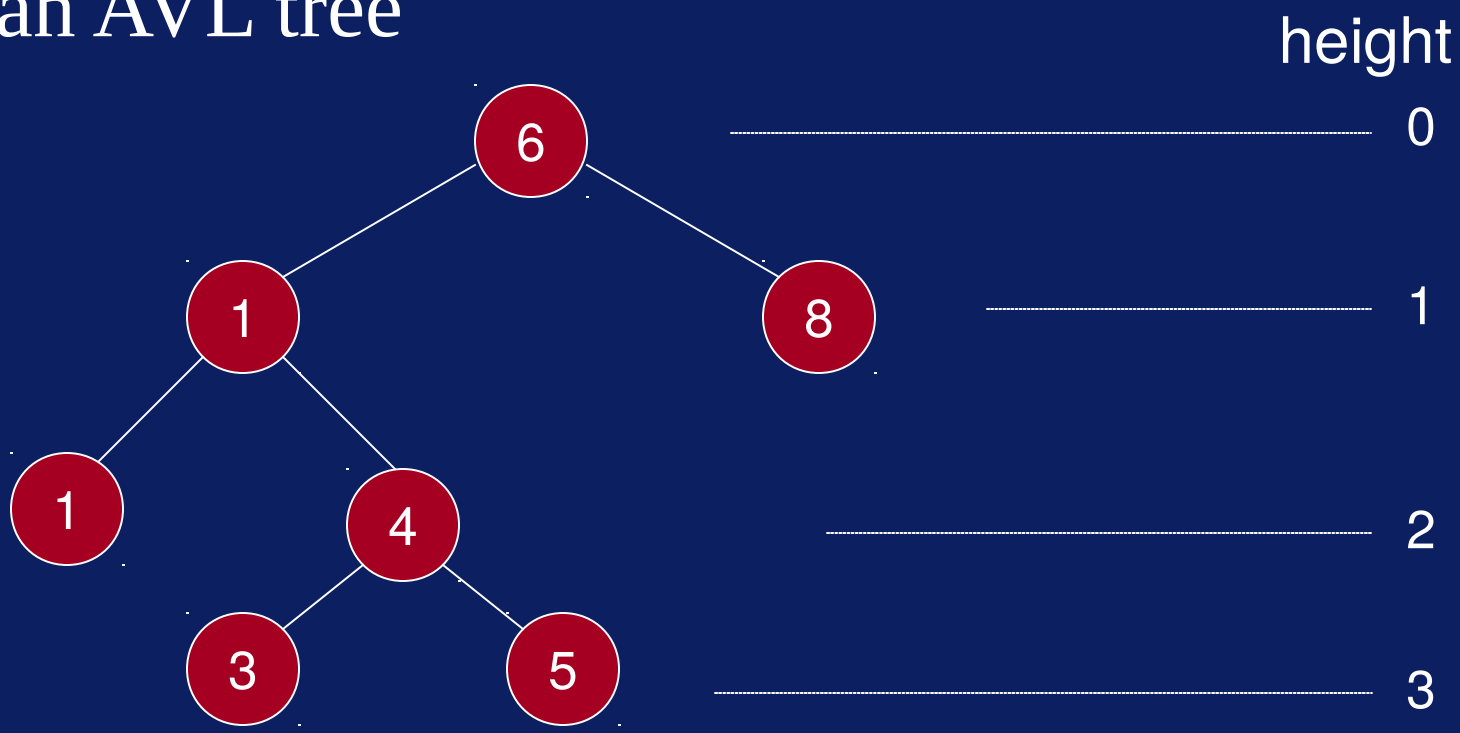
AVL Tree

- An AVL Tree



AVL Tree

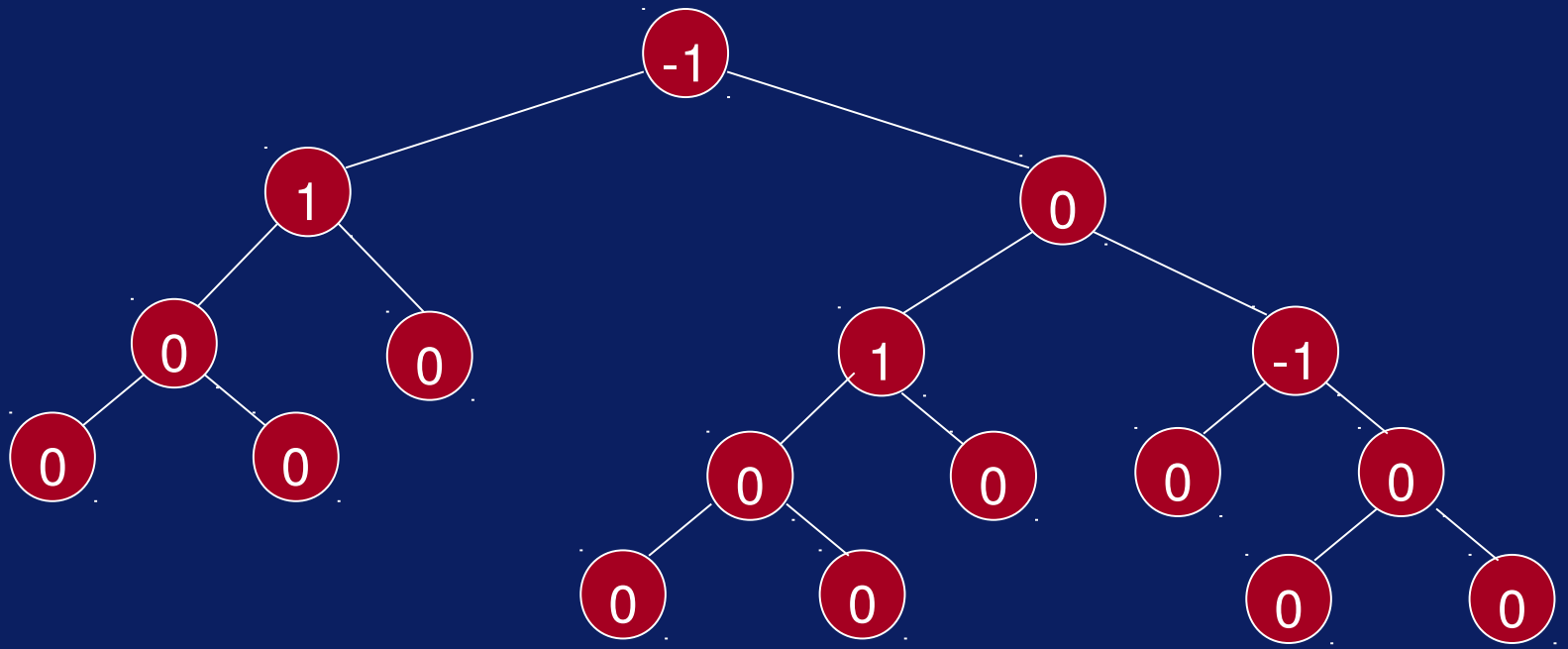
- Not an AVL tree



Balanced Binary Tree

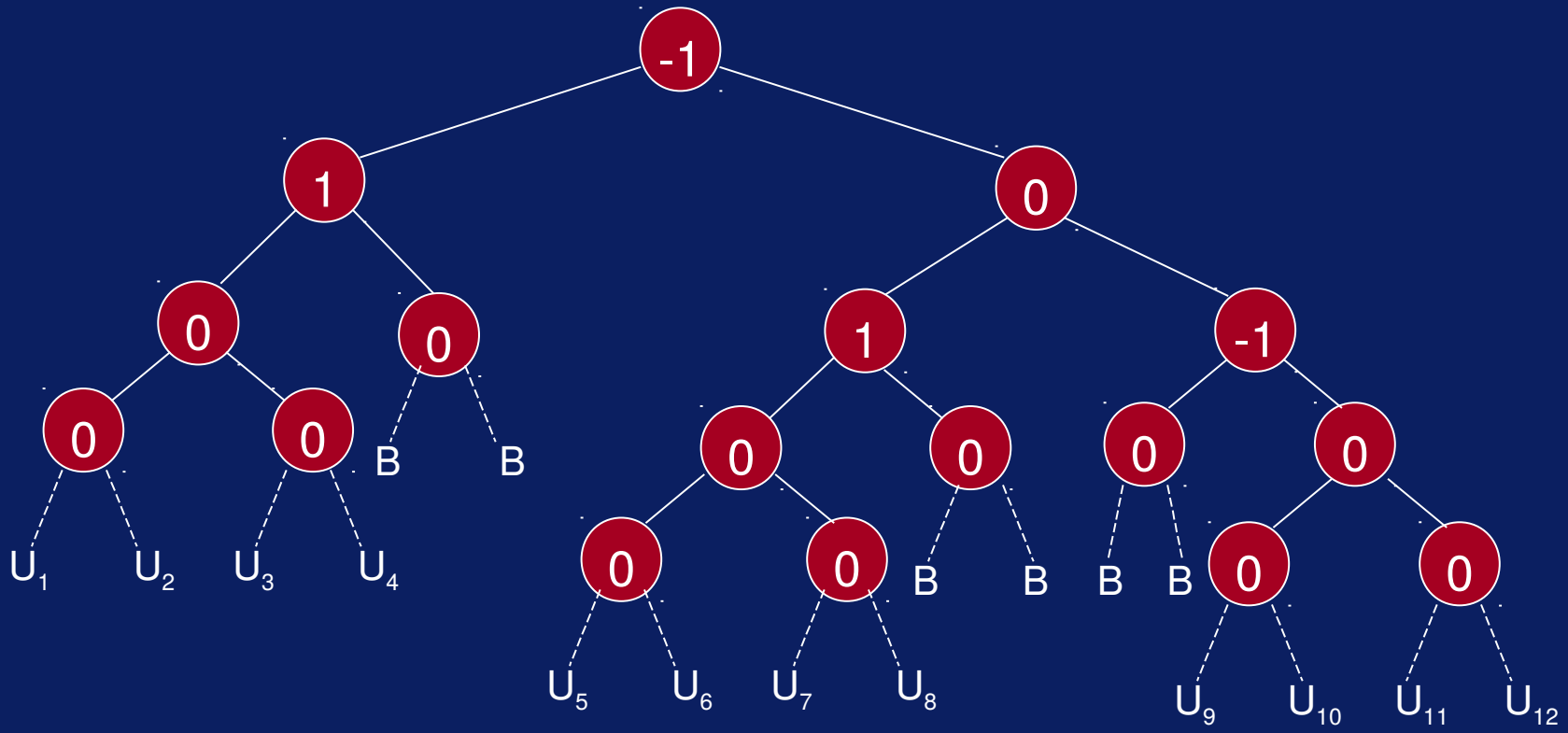
- The *height* of a binary tree is the maximum level of its leaves (also called the depth).
- The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.
- Here, for example, is a balanced tree. Each node has an indicated balance of 1, 0, or –1.

Balanced Binary Tree



Balanced Binary Tree

Insertions and effect on balance

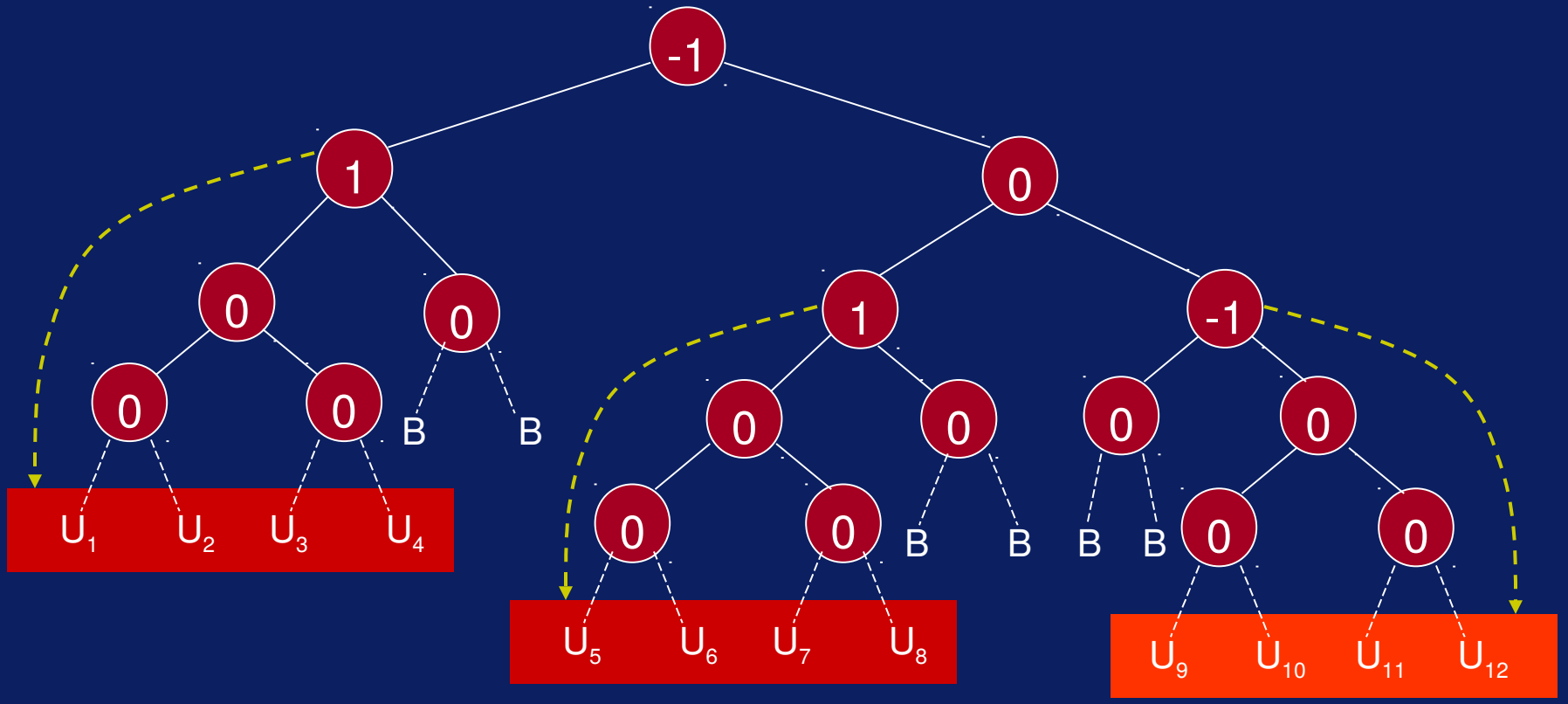


Balanced Binary Tree

- Tree becomes unbalanced only if the newly inserted node
  - is a left descendant of a node that previously had a balance of 1 (U_1 to U_8),
  - or is a descendant of a node that previously had a balance of –1 (U_9 to U_{12})

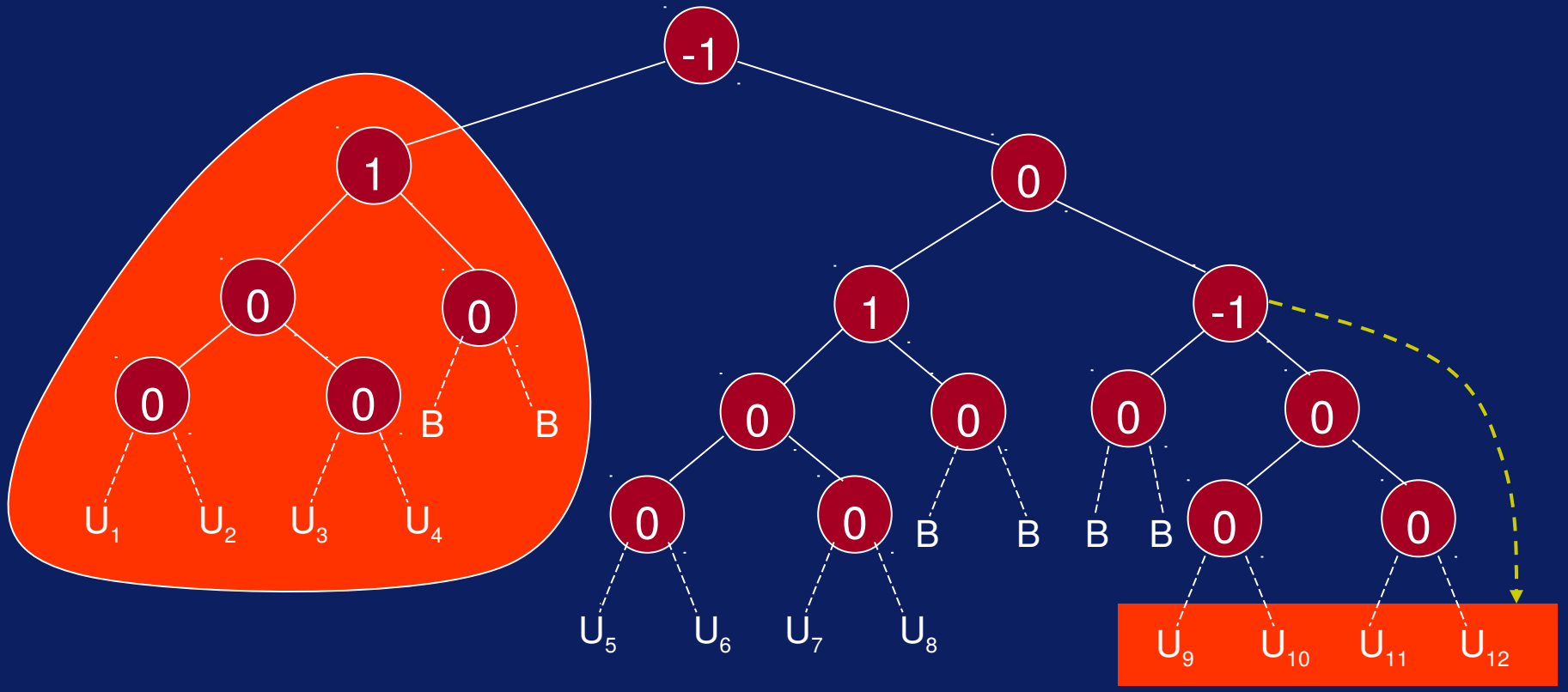
Balanced Binary Tree

Insertions and effect on balance

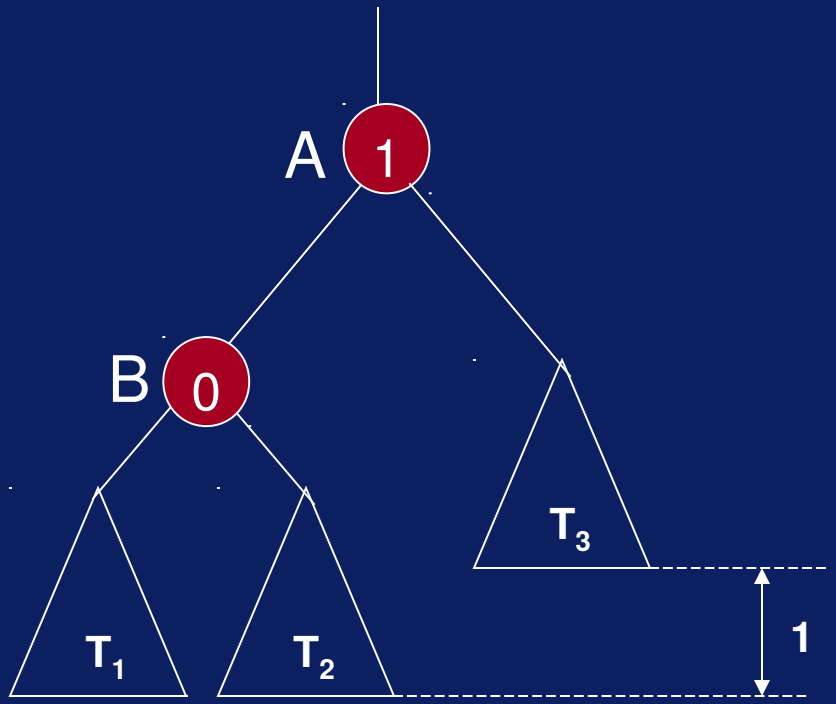


Balanced Binary Tree

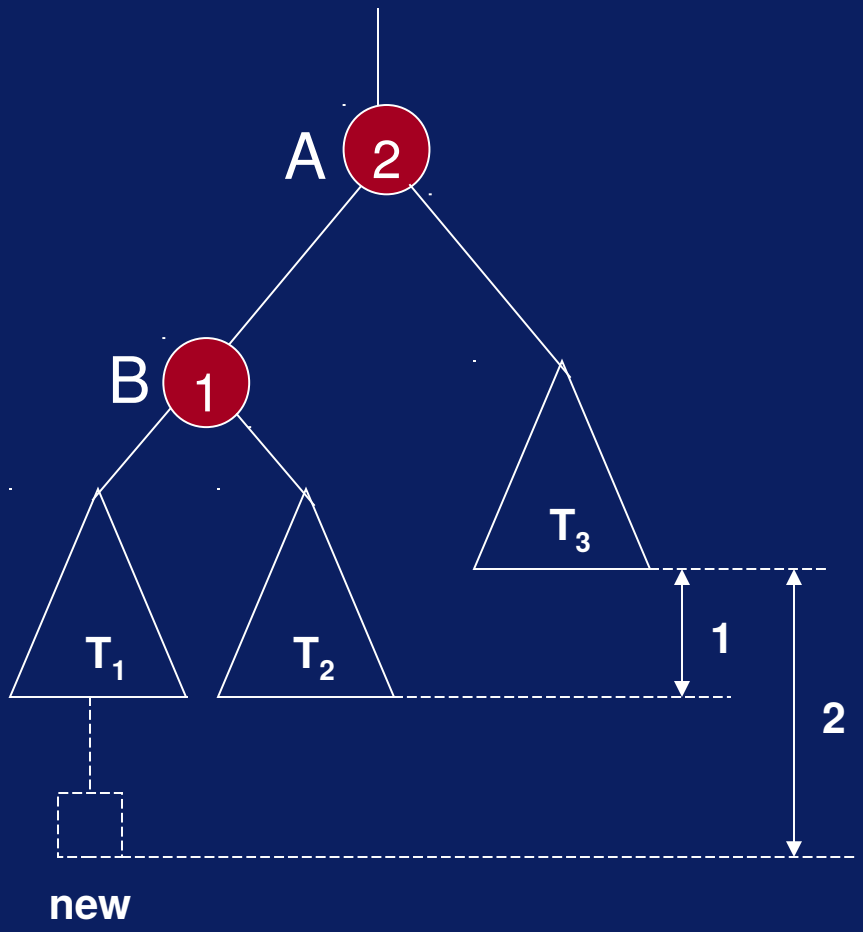
Consider the case of node that was previously 1



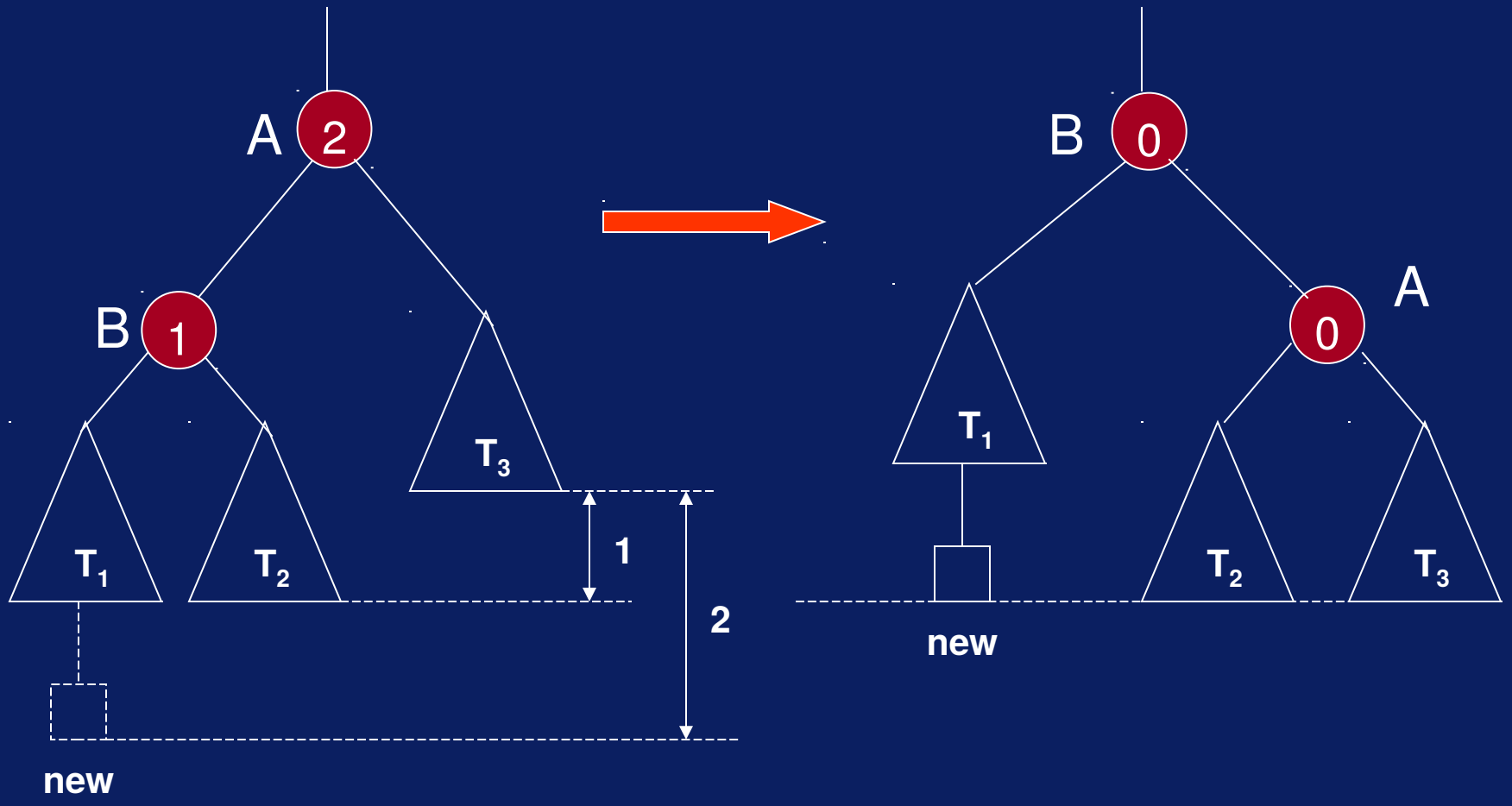
Inserting New Node in AVL Tree



Inserting New Node in AVL Tree



Inserting New Node in AVL Tree



Inorder: T_1 B T_2 A T_3

Inorder: T_1 B T_2 A T_3

AVL Tree Building Example

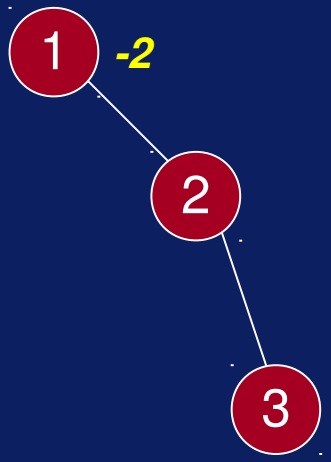
- Let us work through an example that inserts numbers in a balanced search tree.
- We will check the balance after each insert and rebalance if necessary using rotations.

AVL Tree Building Example

Insert(1)

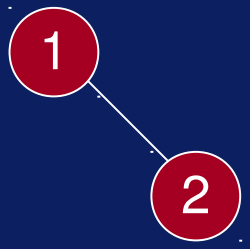
AVL Tree Building Example

Insert(3) single left rotation



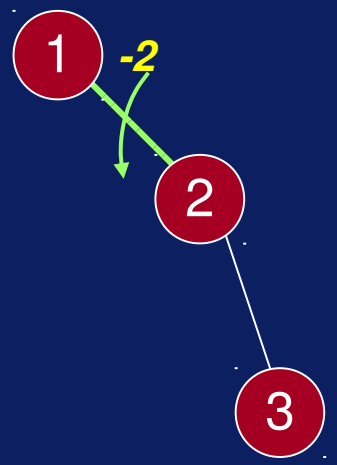
AVL Tree Building Example

Insert(2)



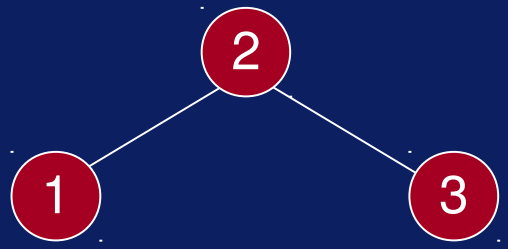
AVL Tree Building Example

Insert(3) single left rotation



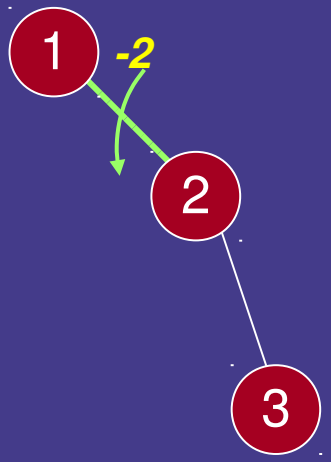
AVL Tree Building Example

Insert(3)



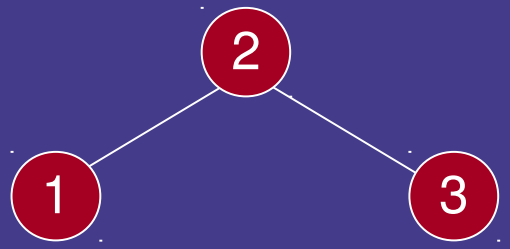
AVL Tree Building Example

Insert(3) single left rotation



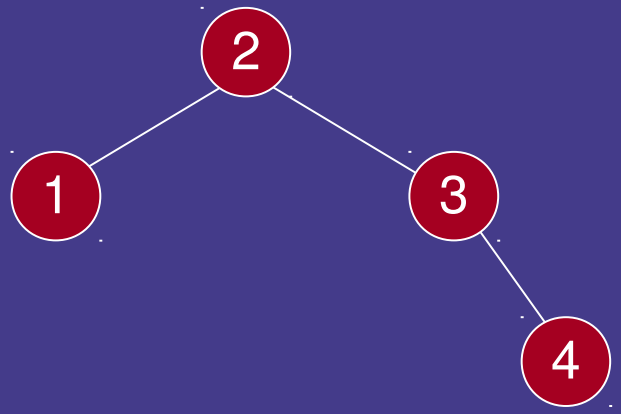
AVL Tree Building Example

Insert(3)



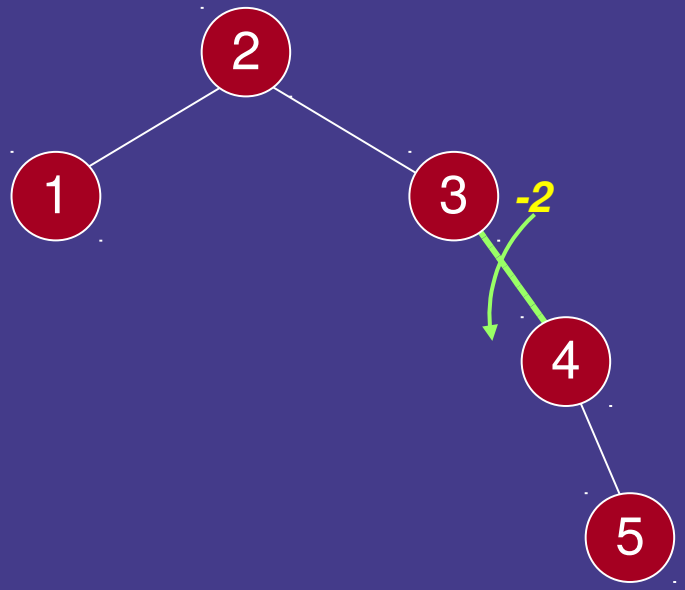
AVL Tree Building Example

Insert(4)



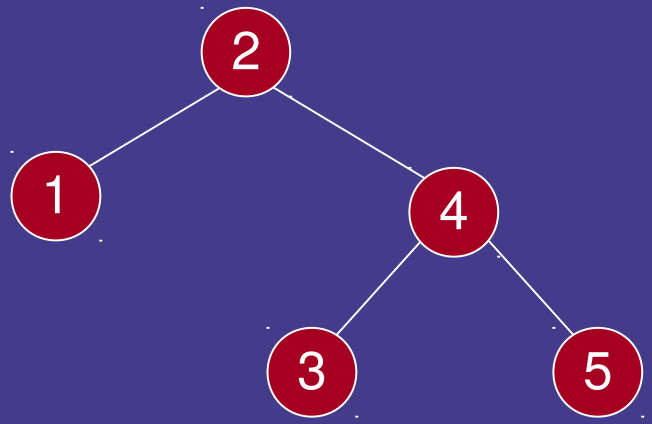
AVL Tree Building Example

Insert(5)



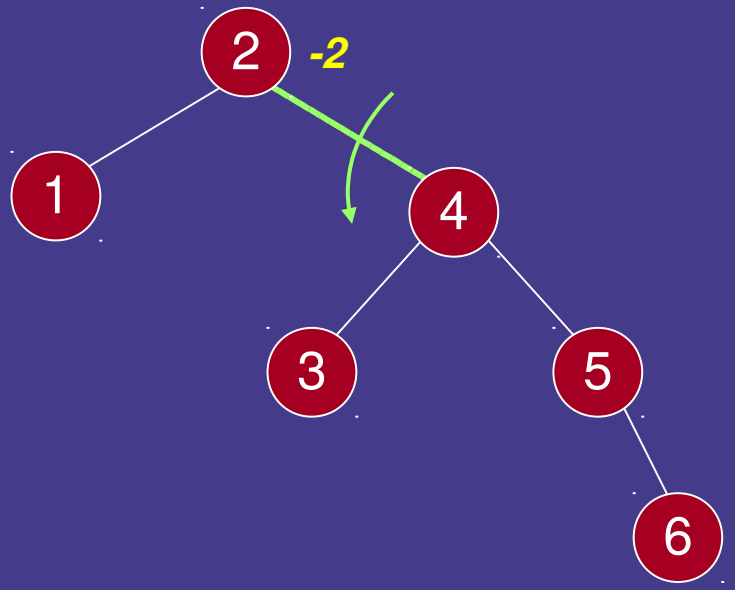
AVL Tree Building Example

Insert(5)



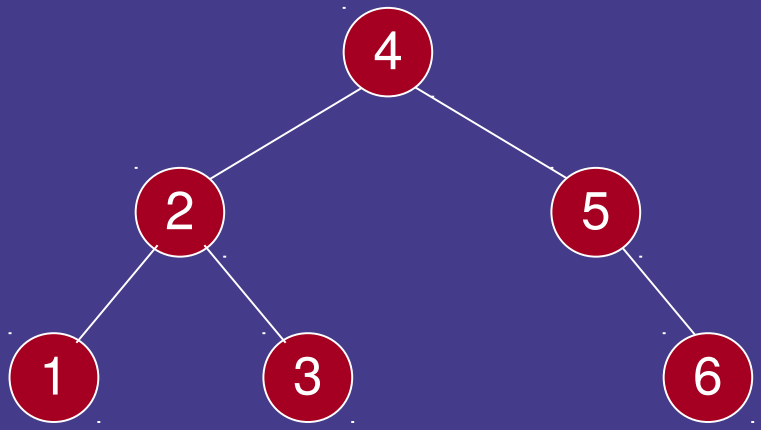
AVL Tree Building Example

Insert(6)



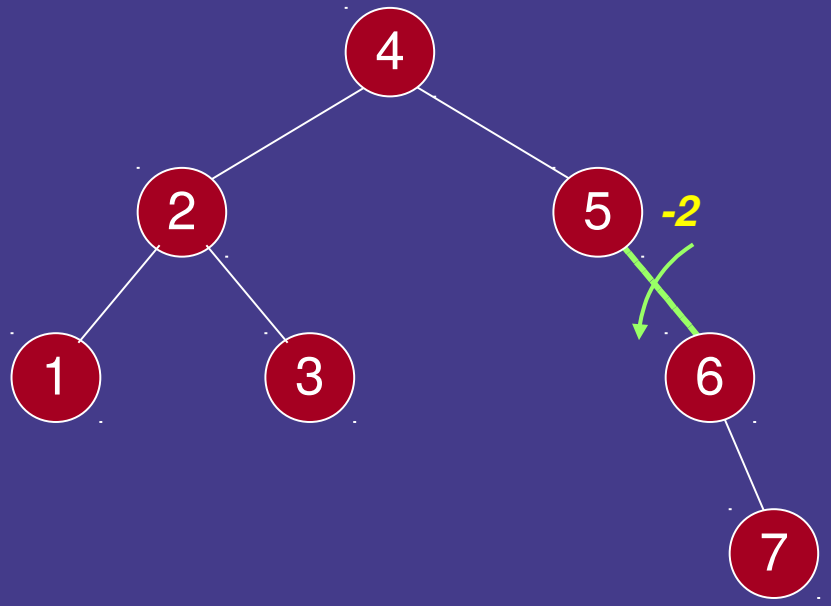
AVL Tree Building Example

Insert(6)



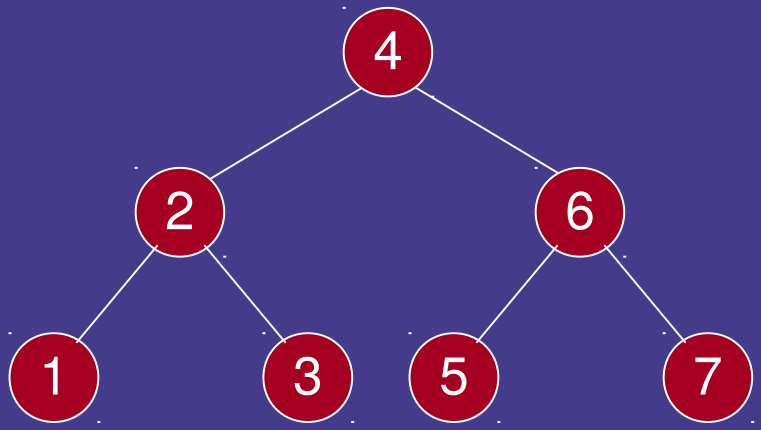
AVL Tree Building Example

Insert(7)



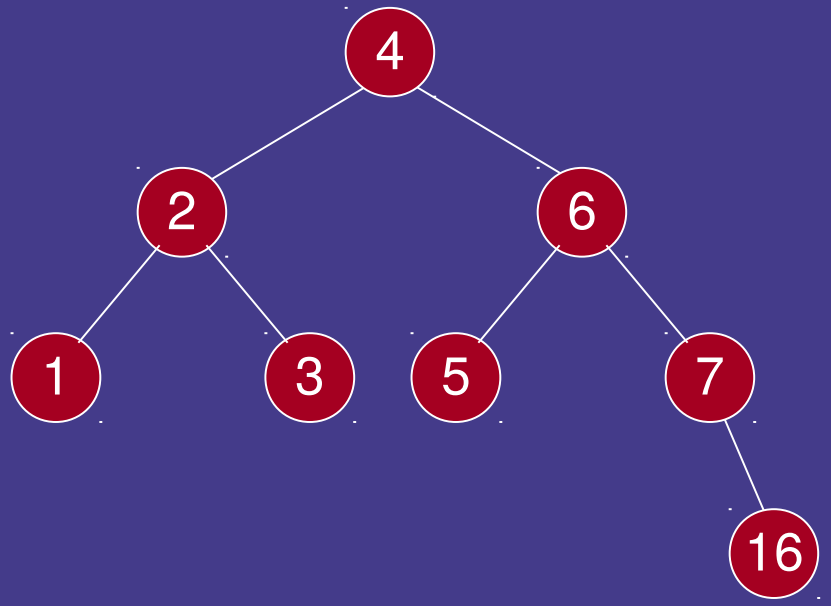
AVL Tree Building Example

Insert(7)



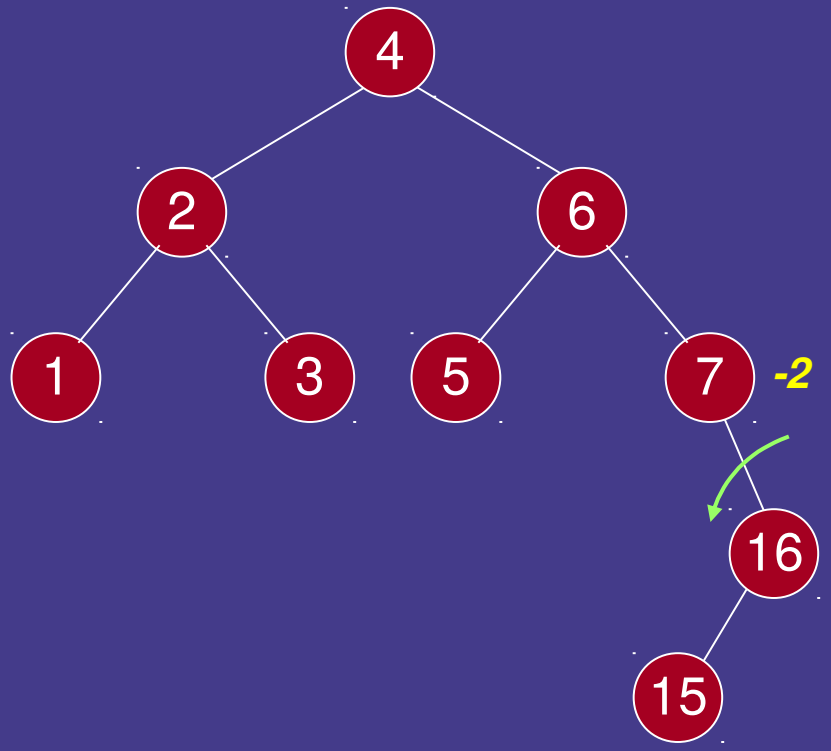
AVL Tree Building Example

Insert(16)



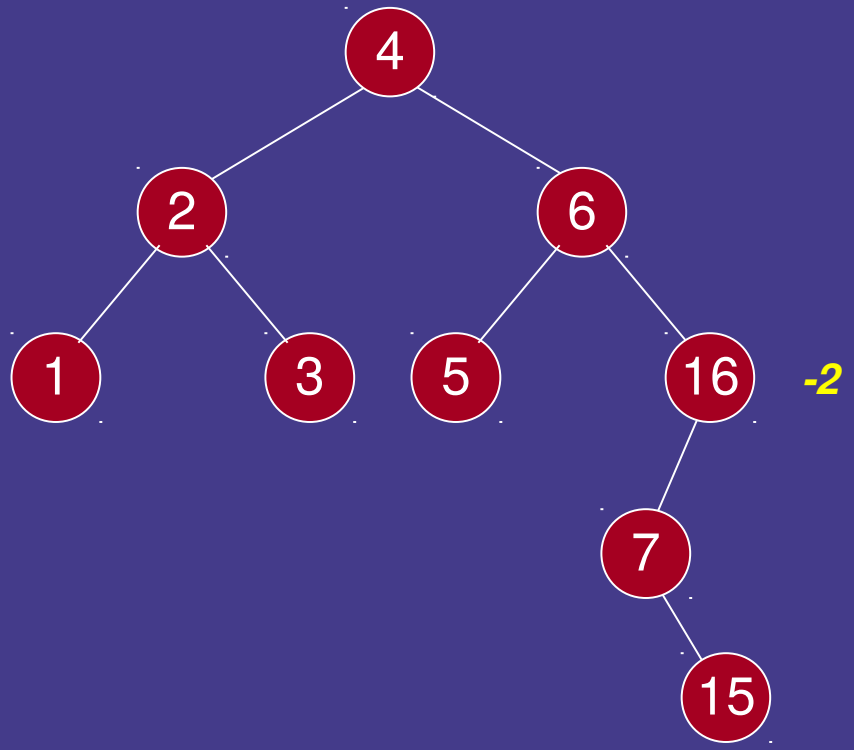
AVL Tree Building Example

Insert(15)



AVL Tree Building Example

Insert(15)



Cases for Rotation

- Single rotation does not seem to restore the balance.
- The problem is the node 15 is in an inner subtree that is too deep.
- Let us revisit the rotations.

Cases for Rotation

- Let us call the node that must be rebalanced α.
- Since any node has at most two children, and a height imbalance requires that α's two subtrees differ by two (or –2), the violation will occur in four cases:

Cases for Rotation

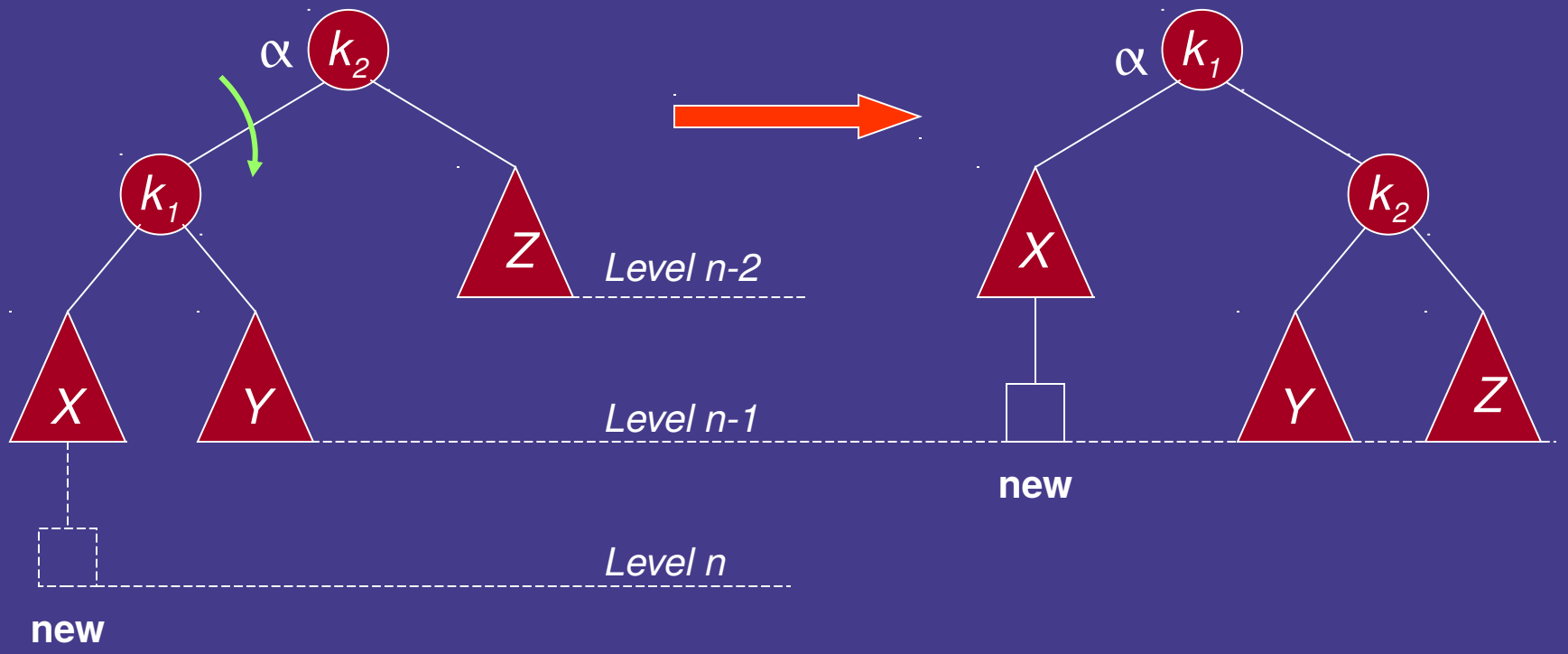
1. An insertion into left subtree of the left child of α.
2. An insertion into right subtree of the left child of α.
3. An insertion into left subtree of the right child of α.
4. An insertion into right subtree of the right child of α.

Cases for Rotation

- The insertion occurs on the “outside” (i.e., left-left or right-right) in cases 1 and 4
- Single rotation can fix the balance in cases 1 and 4.
- Insertion occurs on the “inside” in cases 2 and 3 which single rotation cannot fix.

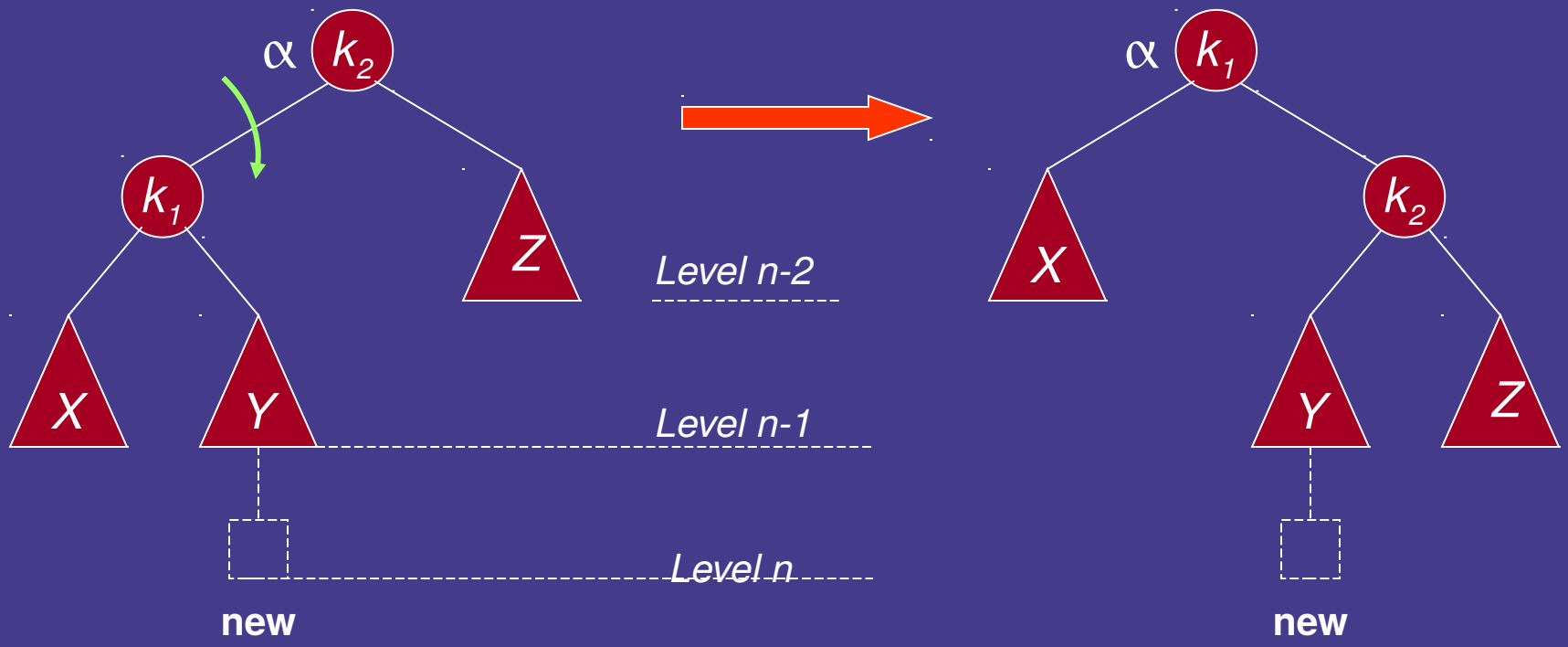
Cases for Rotation

- Single right rotation to fix case 1.

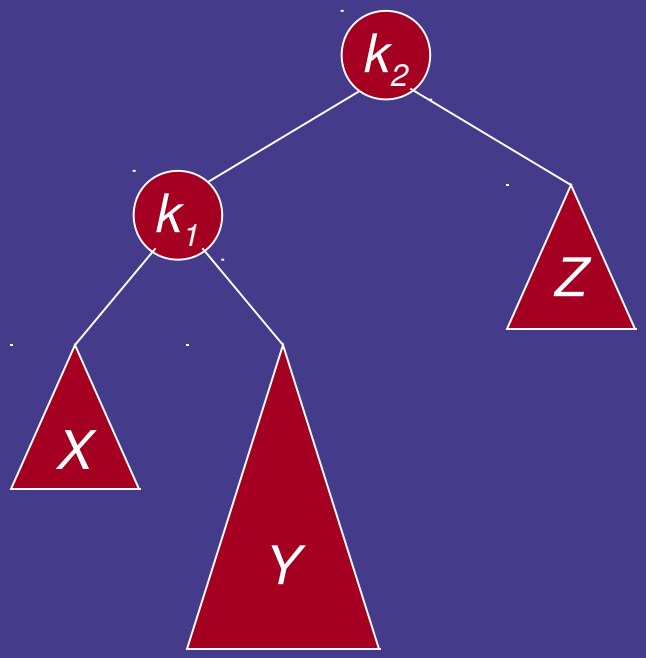


Cases for Rotation

- Single right rotation *fails* to fix case 2.

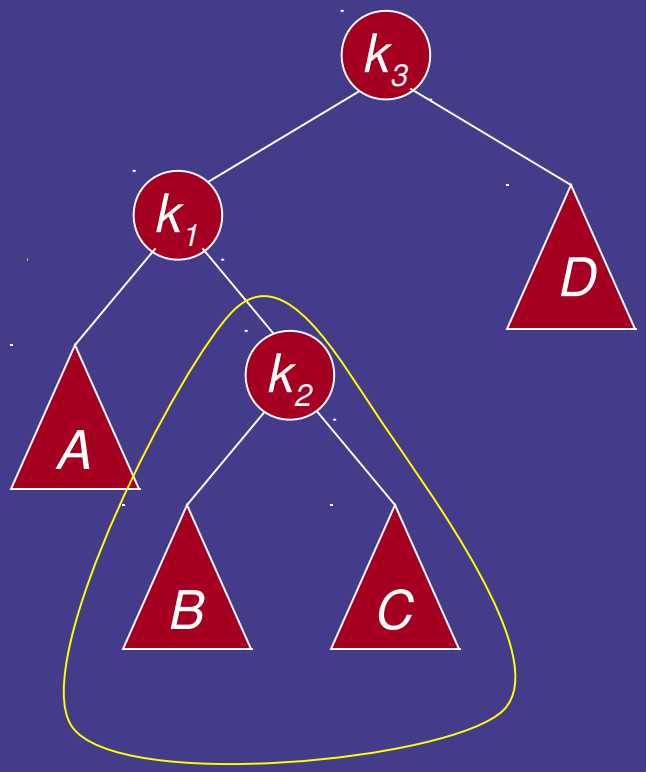


Cases for Rotation



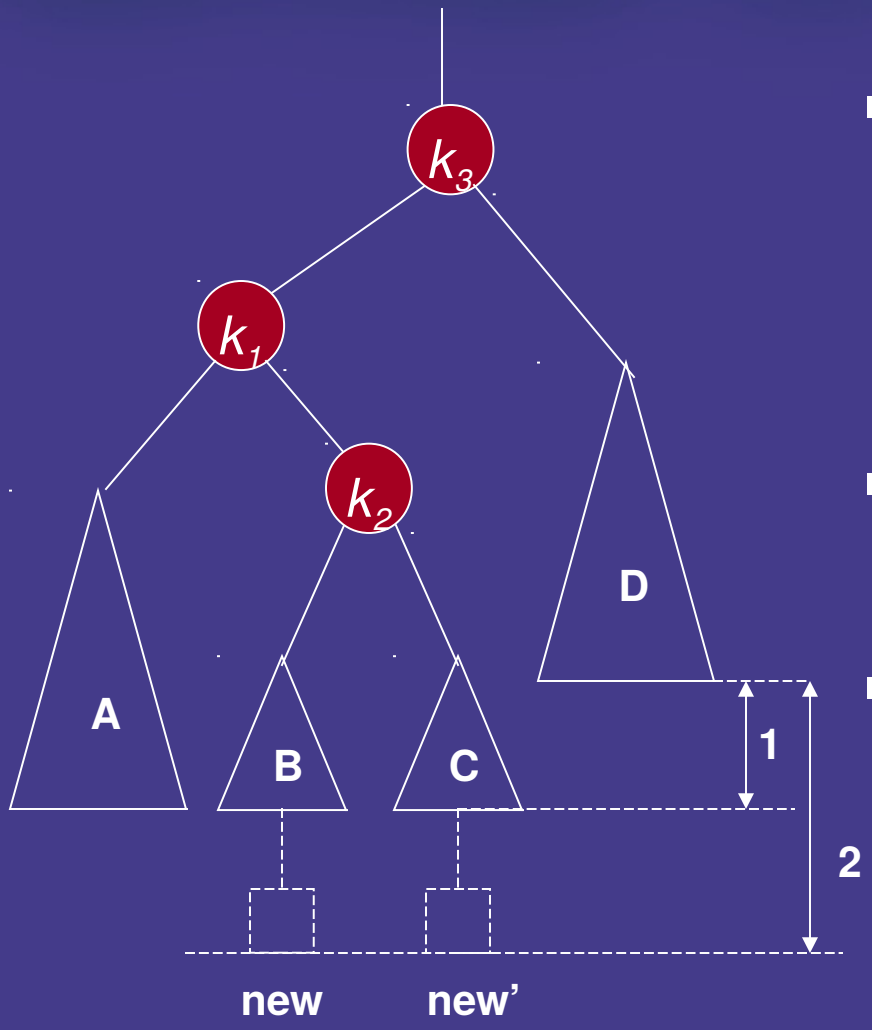
- Y is non-empty because the new node was inserted in Y.
- Thus Y has a root and two subtrees.
- View the entire tree with four subtrees connected with 3 nodes.

Cases for Rotation



- Y is non-empty because the new node was inserted in Y.
- Thus Y has a root and two subtrees.
- View the entire tree with four subtrees connected with 3 nodes.

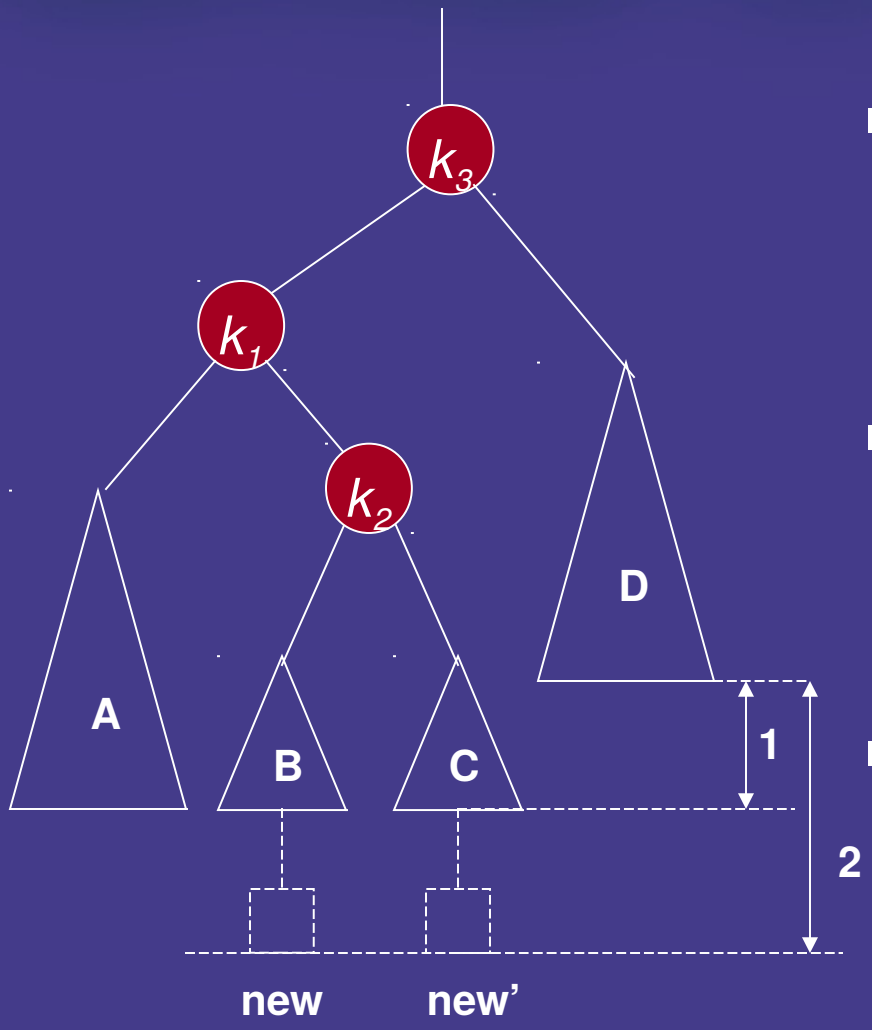
Cases for Rotation



New node inserted a either of the two spots

- Exactly one of tree *B* or *C* is two levels deeper than *D*; we are not sure which one.
- Good thing: it does not matter.
- To rebalance, k_3 cannot be left as the root.

Cases for Rotation

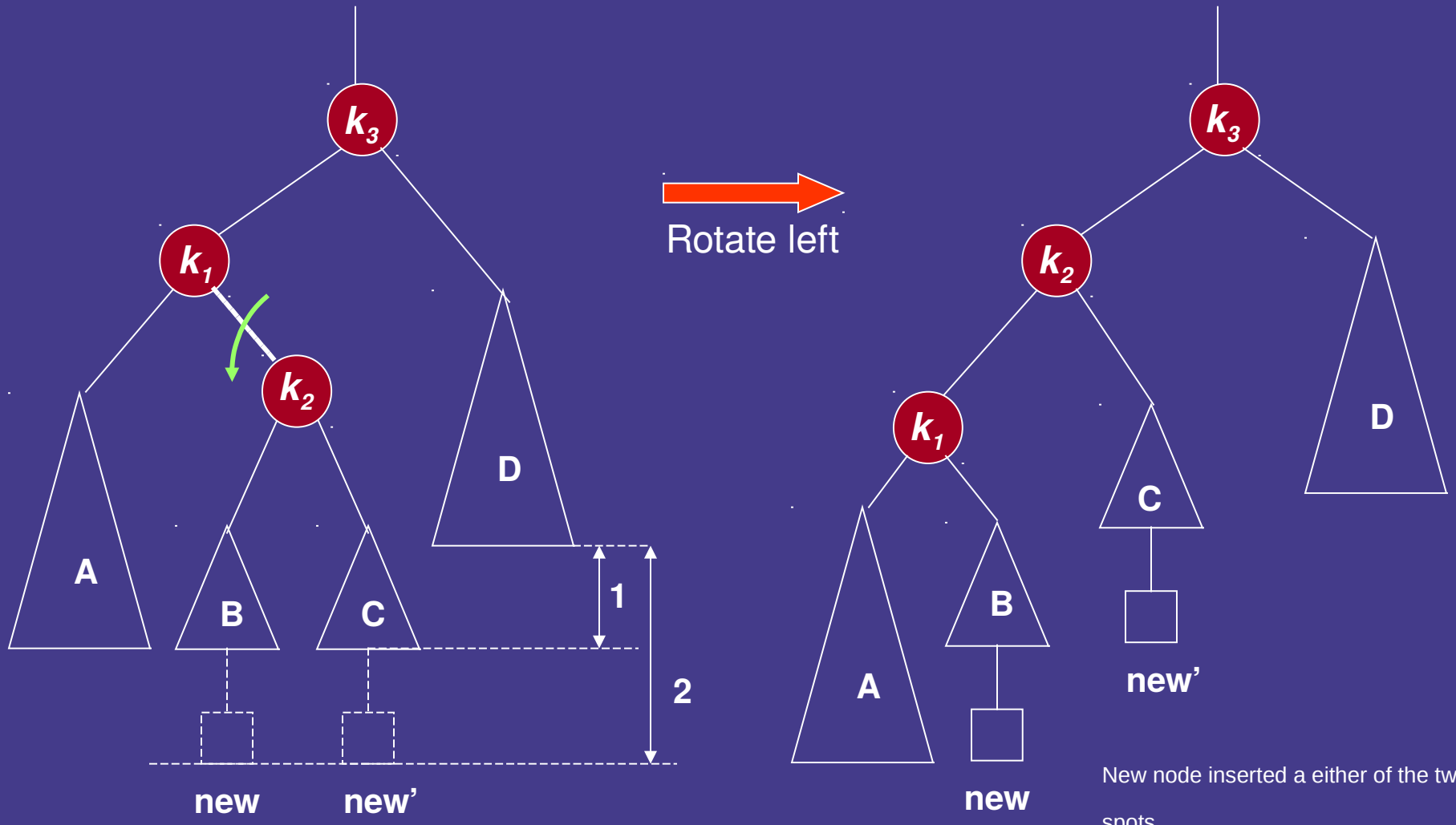


New node inserted a either of the two spots

- A rotation between k_3 and k_1 (k_3 was k_2 then) was shown to not work.
- The only alternative is to place k_2 as the new root.
- This forces k_1 to be k_2's left child and k_3 to be its right child.

Cases for Rotation

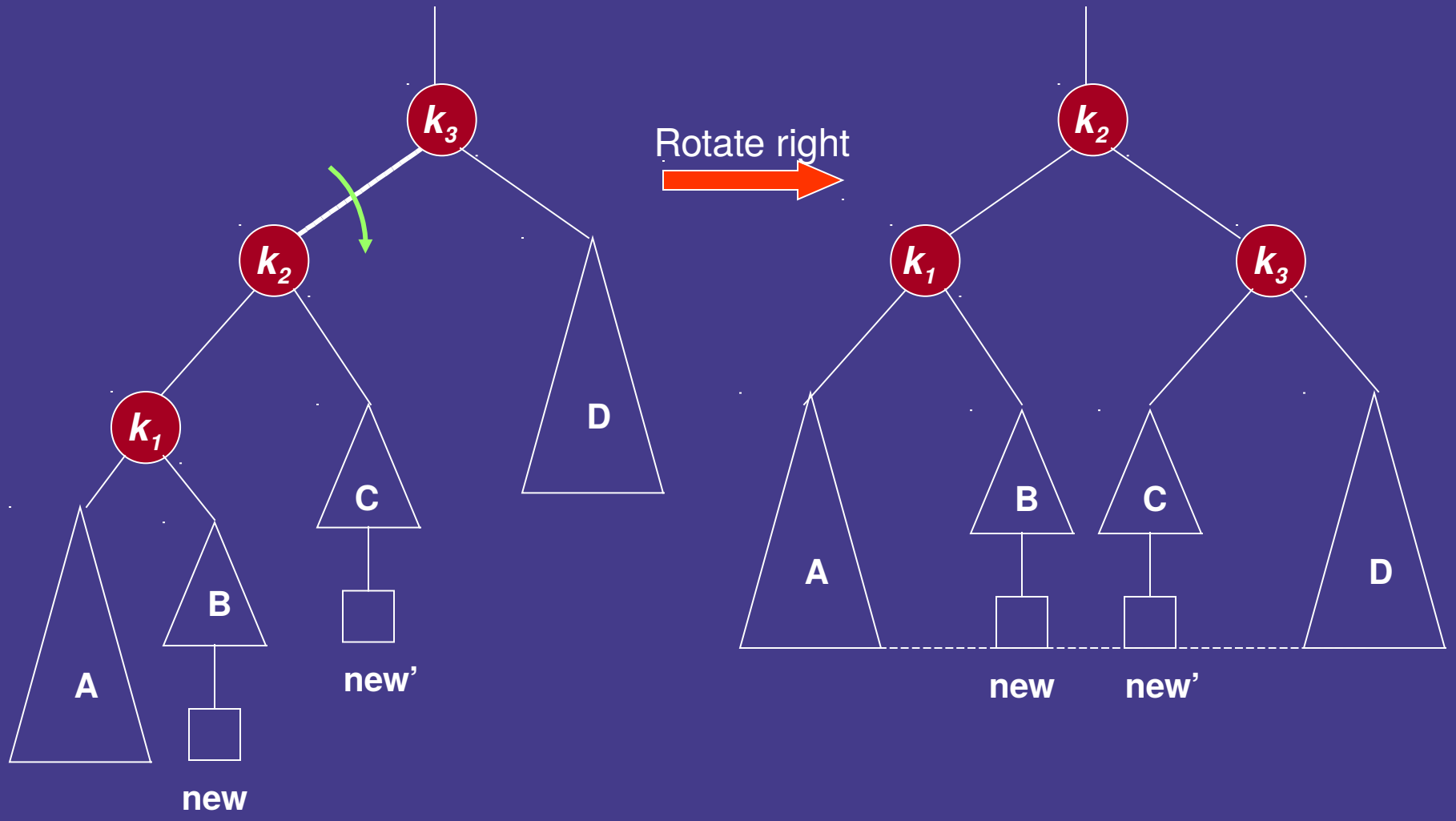
- Left-right *double* rotation to fix case 2.



New node inserted a either of the two spots

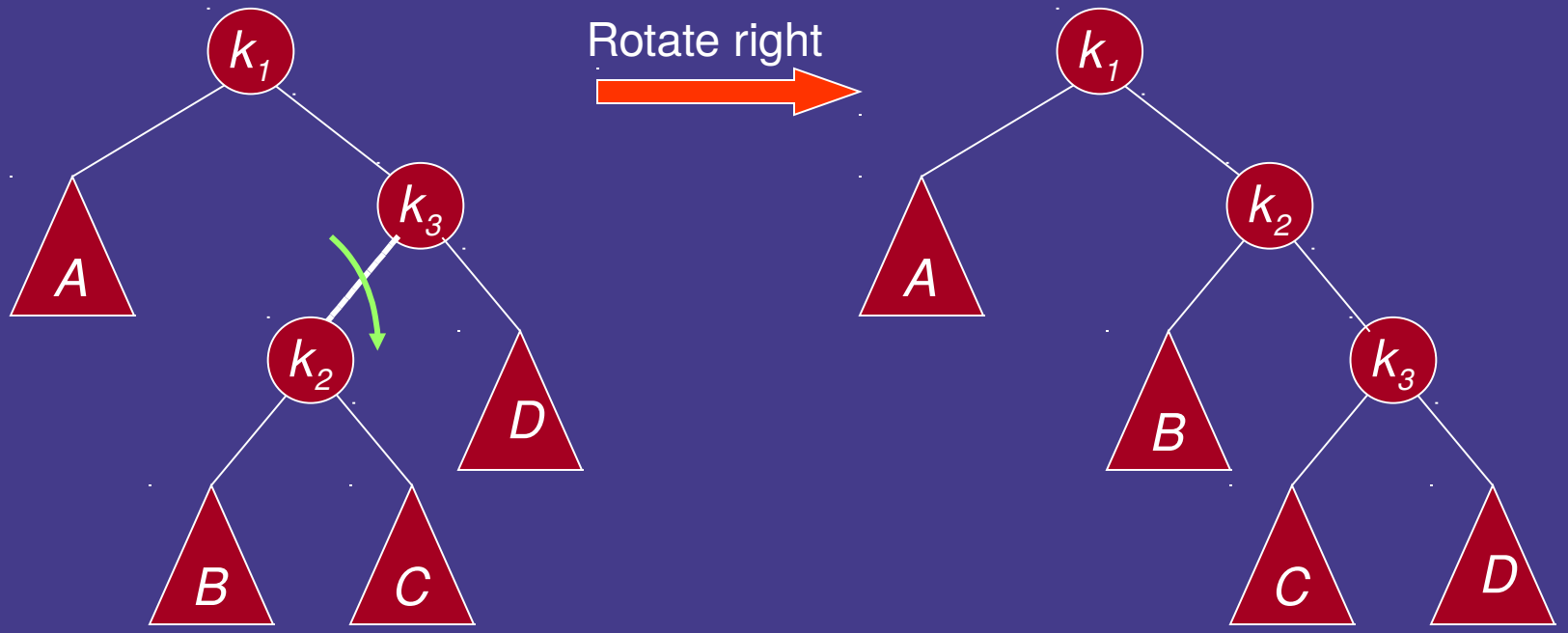
Cases for Rotation

- Left-right *double* rotation to fix case 2.



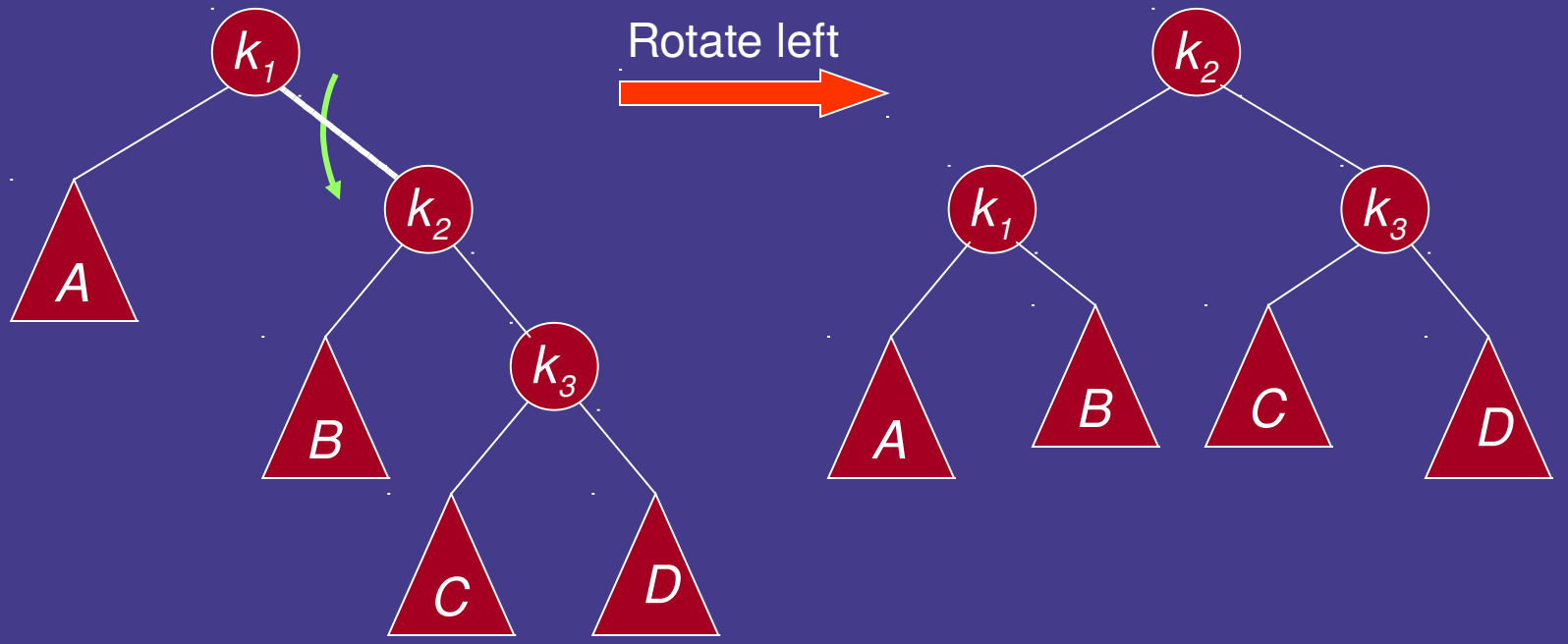
Cases for Rotation

- Right-left *double* rotation to fix case 3.



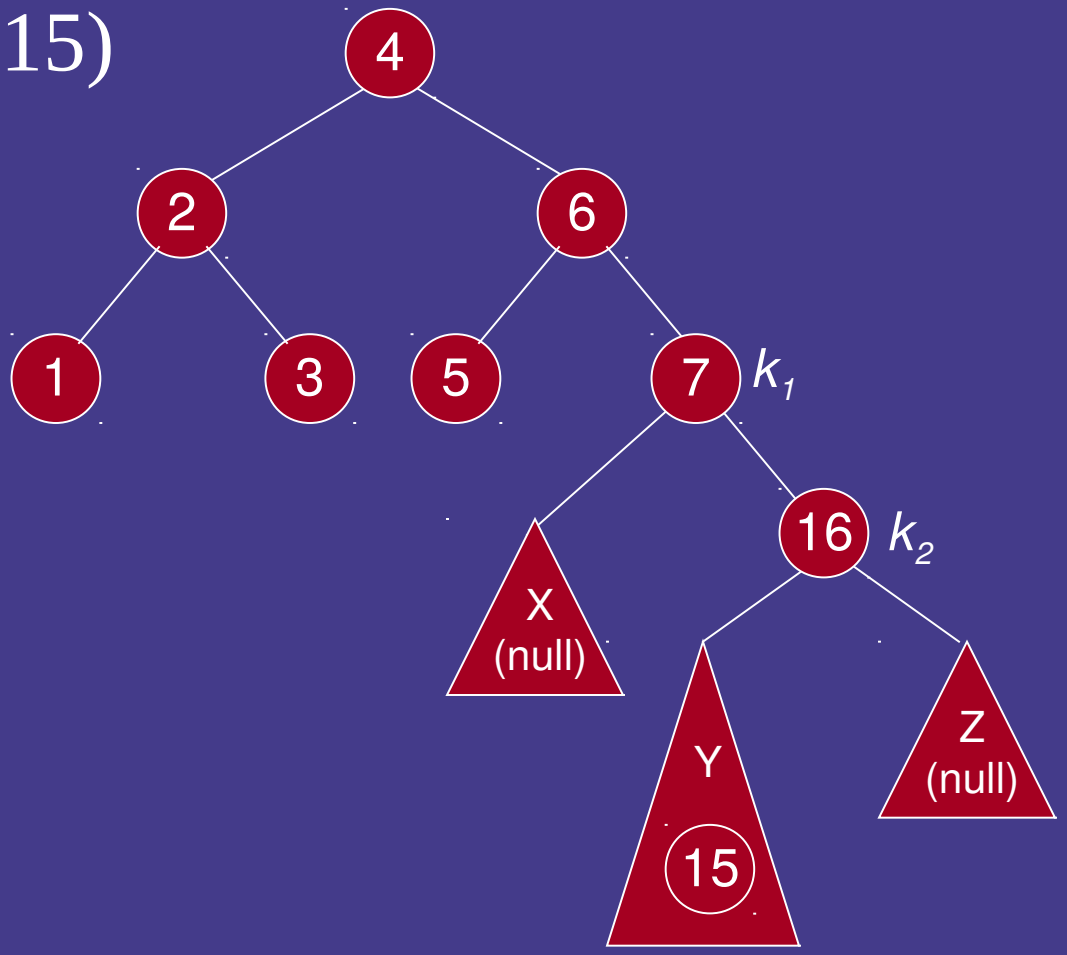
Cases for Rotation

- Right-left *double* rotation to fix case 3.



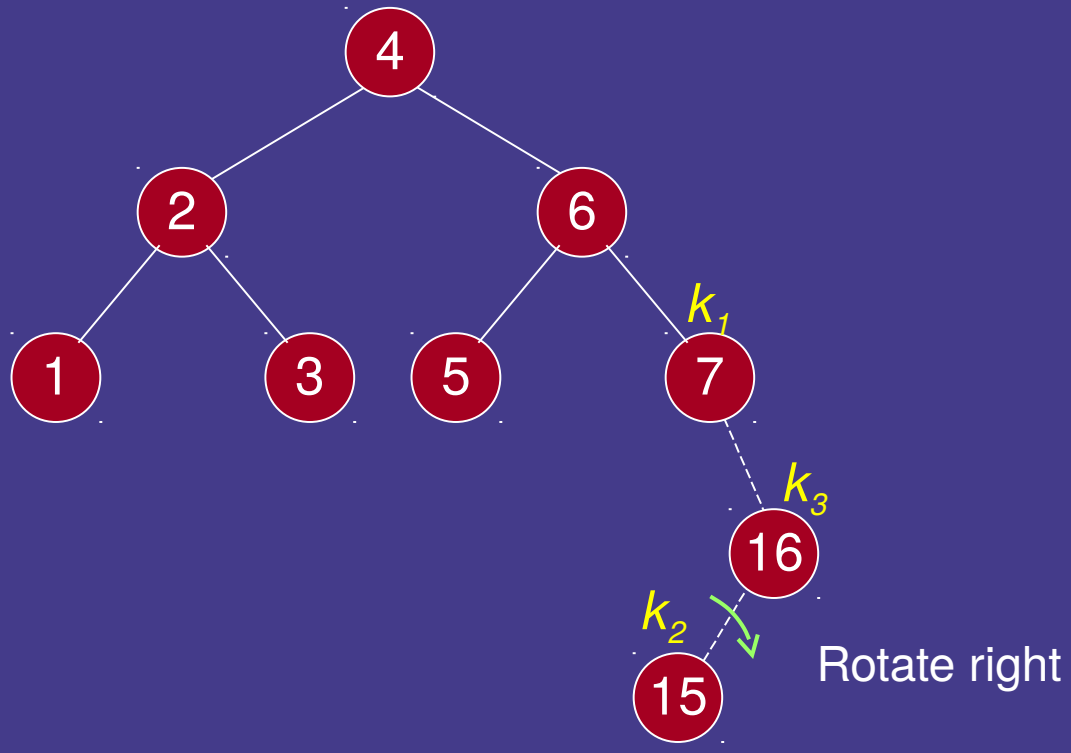
AVL Tree Building Example

Insert(15)



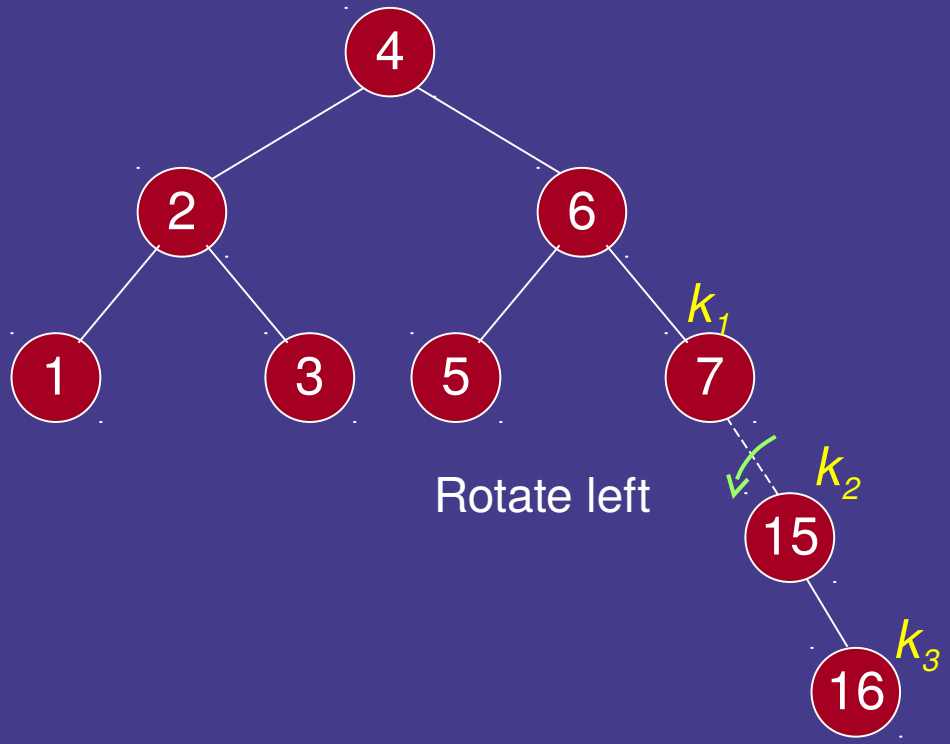
AVL Tree Building Example

Insert(15) right-left double rotation



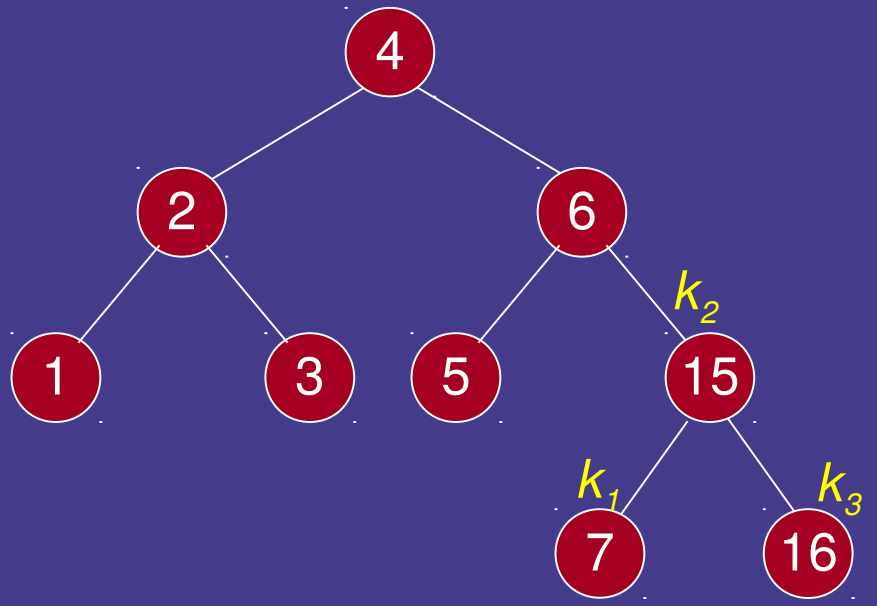
AVL Tree Building Example

Insert(15) right-left double rotation



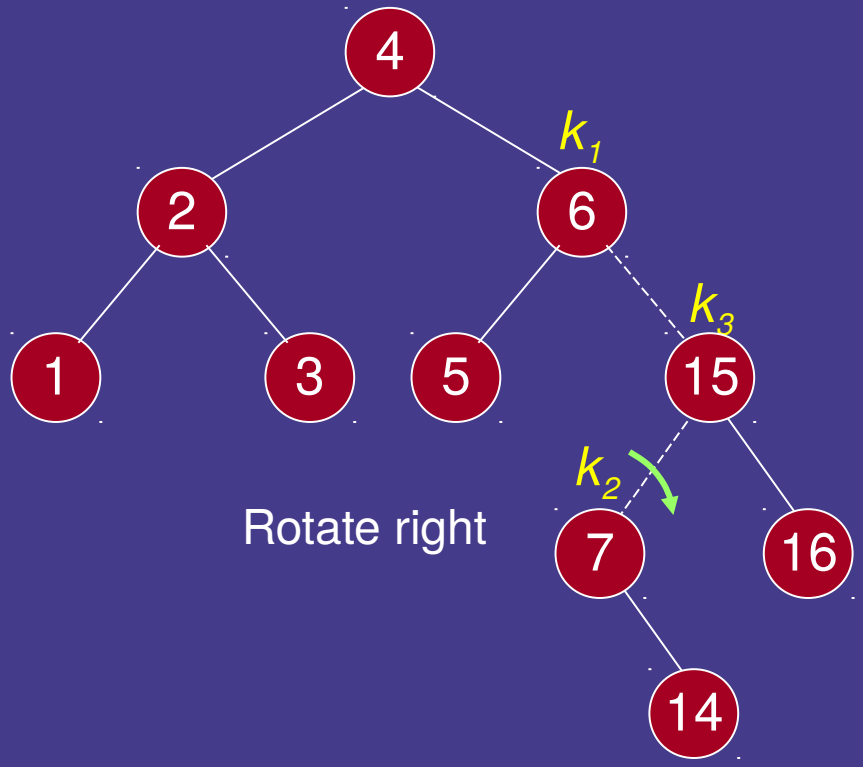
AVL Tree Building Example

Insert(15) right-left double rotation



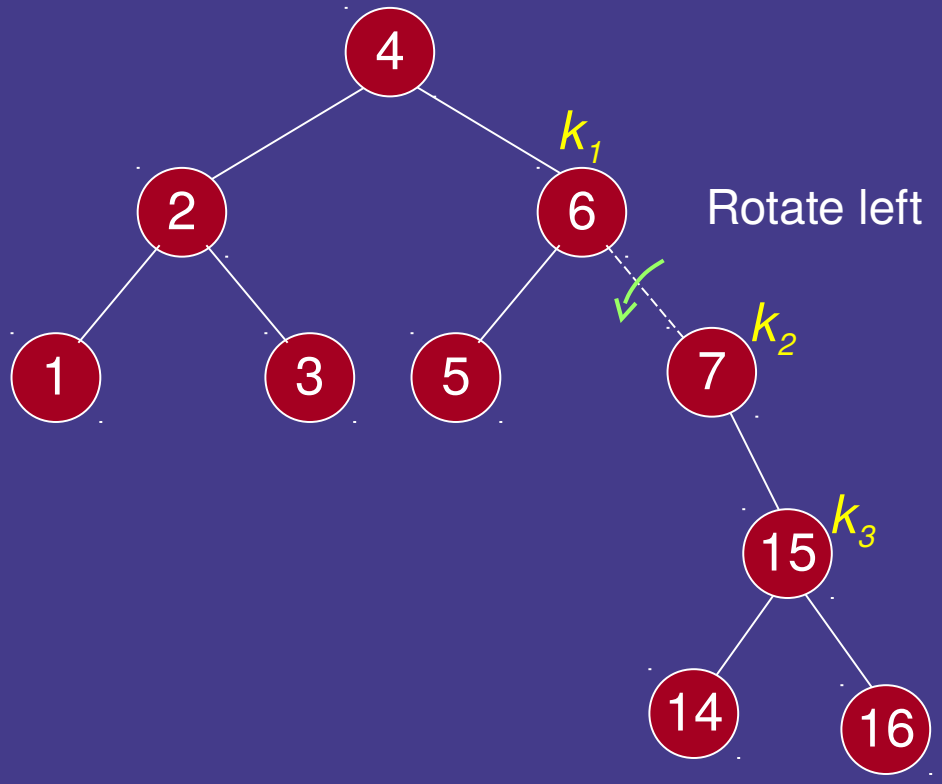
AVL Tree Building Example

Insert(14): right-left double rotation



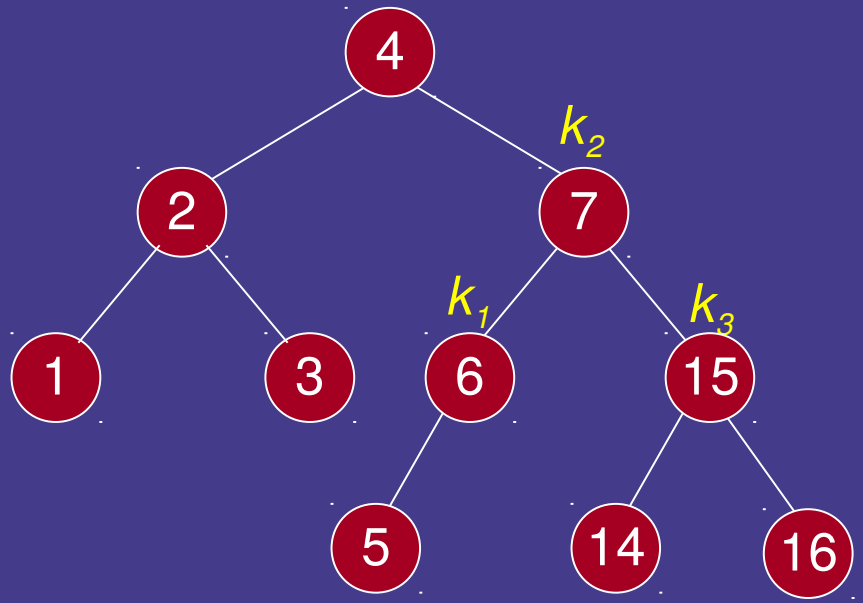
AVL Tree Building Example

Insert(14): right-left double rotation



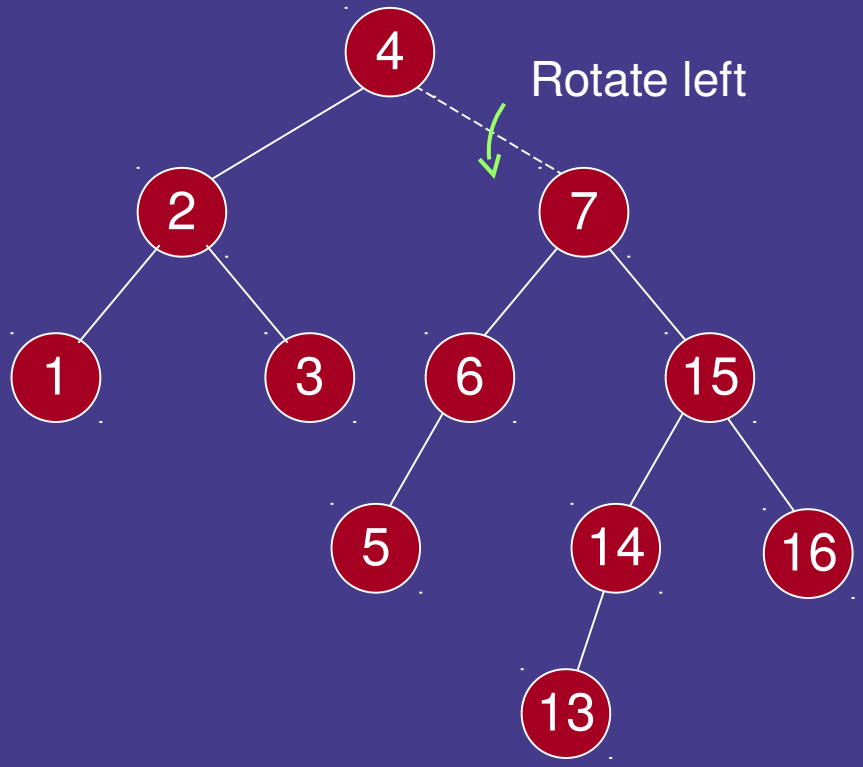
AVL Tree Building Example

Insert(14): right-left double rotation



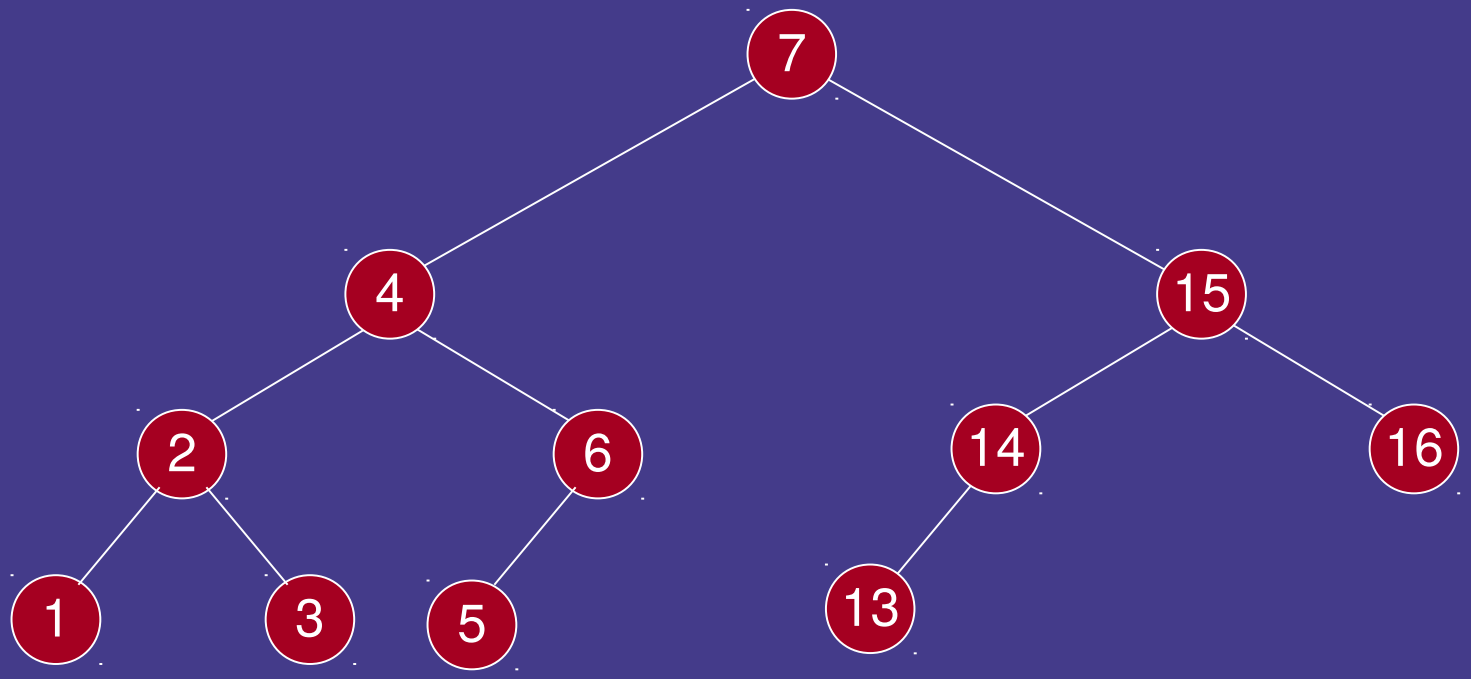
AVL Tree Building Example

Insert(13): single rotation



AVL Tree Building Example

Insert(13): single rotation



C++ code for insert

```
TreeNode<int>*
avlInsert(TreeNode<int>* root, int info)
{
    if( info < root->getInfo() ){
        root->se tLeft(avlInsert(root->getLeft(), info));
        int htdiff = height(root->getLeft()) –
                     height(root->getRight());
        if( htdiff == 2 )
            if( info < root->getLeft()->getInfo() )
              root = singleRightRotation( root );
            else
              root = doubleLeftRightRotation( root );
  }
```

C++ code for insert

```
TreeNode<int>*
avlInsert(TreeNode<int>* root, int info)
{
    if( info < root->getInfo() ){
        root->setLeft(avlInsert(root->getLeft(), info));
        int htdiff = height(root->getLeft()) –
                     height(root->getRight());
        if( htdiff == 2 )
            if( info < root->getLeft()->getInfo() )
              root = singleRightRotation( root );
            else
              root = doubleLeftRightRotation( root );
  }
```

C++ code for insert

```
TreeNode<int>*
avlInsert(TreeNode<int>* root, int info)
{
    if( info < root->getInfo() ){
        root->setLeft(avlInsert(root->getLeft(), info));
        int htdiff = height(root->getLeft()) –
                     height(root->getRight());
        if( htdiff == 2 )
            if( info < root->getLeft()->getInfo() )
              root = singleRightRotation( root );
            else
              root = doubleLeftRightRotation( root );
  }
```

Root of left subtree may change

C++ code for insert

```
TreeNode<int>*
avlInsert(TreeNode<int>* root, int info)
{
    if( info < root->getInfo() ){
        root->setLeft(avlInsert(root->getLeft(), info));
        int htdiff = height(root->getLeft()) –
                     height(root->getRight());
        if( htdiff == 2 )
            if( info < root->getLeft()->getInfo() )
              root = singleRightRotation( root );
            else
              root = doubleLeftRightRotation( root );
  }
```

C++ code for insert

```
TreeNode<int>*
avlInsert(TreeNode<int>* root, int info)
{
    if( info < root->getInfo() ){
        root->setLeft(avlInsert(root->getLeft(), info));
        int htdiff = height(root->getLeft()) –
                     height(root->getRight());
        if( htdiff == 2 )
            if( info < root->getLeft()->getInfo() )
              root = singleRightRotation( root );
            else
              root = doubleLeftRightRotation( root );
  }
```

C++ code for insert

```
TreeNode<int>*
avlInsert(TreeNode<int>* root, int info)
{
    if( info < root->getInfo() ){
        root->setLeft(avlInsert(root->getLeft(), info));
        int htdiff = height(root->getLeft()) –
                     height(root->getRight());
        if( htdiff == 2 )
            if( info < root->getLeft()->getInfo() )
              root = singleRightRotation( root );
            else
              root = doubleLeftRightRotation( root );
  }
```

Outside case

inside case

C++ code for insert

```
    else if(info > root->getInfo() ) {
        root->setRight(avlInsert(root->getRight(),info));
        int htdiff = height(root->getRight()) –
                     height(root->getLeft());
        if( htdiff == 2 )
            if( info > root->getRight()->getInfo() )
                root = singleLeftRotation( root );
            else
                root = doubleRightLeftRotation( root );
    }
    // else a node with info is already in the tree. In
    // case, reset the height of this root node.
    int ht = Max(height(root->getLeft()),
                 height(root->getRight()));
    root->setHeight( ht+1 ); // new height for root.
    return root;
}
```

C++ code for insert

```
    else if(info > root->getInfo() ) {
        root->setRight(avlInsert(root->getRight(),info));
        int htdiff = height(root->getRight()) –
                     height(root->getLeft());
        if( htdiff == 2 )
            if( info > root->getRight()->getInfo() )
                root = singleLeftRotation( root );
            else
                root = doubleRightLeftRotation( root );
    }
    // else a node with info is already in the tree. In
    // case, reset the height of this root node.
    int ht = Max(height(root->getLeft()),
                 height(root->getRight()));
    root->setHeight( ht+1 ); // new height for root.
    return root;
}
```

C++ code for insert

```
    else if(info > root->getInfo() ) {
        root->setRight(avlInsert(root->getRight(),info));
        int htdiff = height(root->getRight()) –
                     height(root->getLeft());
        if( htdiff == 2 )
            if( info > root->getRight()->getInfo() )
                root = singleLeftRotation( root );
            else
                root = doubleRightLeftRotation( root );
    }
    // else a node with info is already in the tree. In
    // case, reset the height of this root node.
    int ht = Max(height(root->getLeft()),
                 height(root->getRight()));
    root->setHeight( ht+1 ); // new height for root.
    return root;
}
```

C++ code for insert

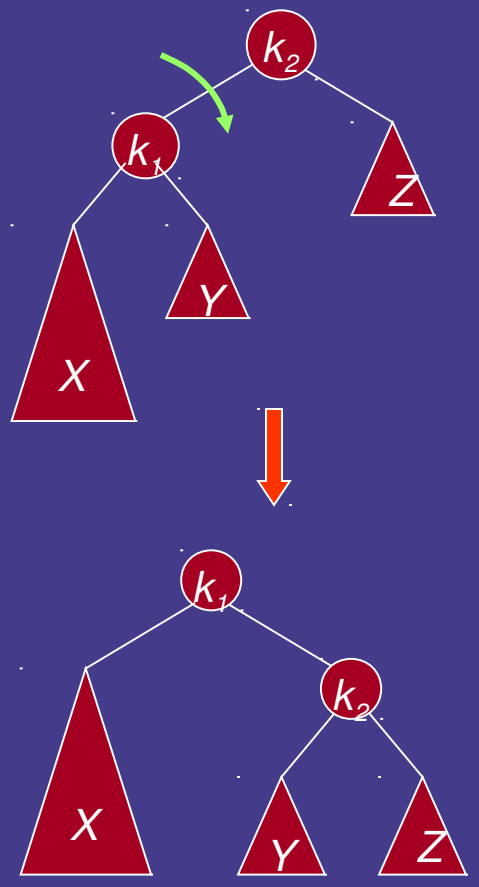
```
    else if(info > root->getInfo() ) {
        root->setRight(avlInsert(root->getRight(),info));
        int htdiff = height(root->getRight()) –
                     height(root->getLeft());
        if( htdiff == 2 )
            if( info > root->getRight()->getInfo() )
                root = singleLeftRotation( root );
            else
                root = doubleRightLeftRotation( root );
    }
    // else a node with info is already in the tree. In
    // case, reset the height of this root node.
    int ht = Max(height(root->getLeft()),
                 height(root->getRight()));
    root->setHeight( ht+1 ); // new height for root.
    return root;
}
```

C++ code for insert

```
    else if(info > root->getInfo() ) {
        root->setRight(avlInsert(root->getRight(),info));
        int htdiff = height(root->getRight()) –
                     height(root->getLeft());
        if( htdiff == 2 )
            if( info > root->getRight()->getInfo() )
                root = singleLeftRotation( root );
            else
                root = doubleRightLeftRotation( root );
    }
    // else a node with info is already in the tree. In
    // case, reset the height of this root node.
    int ht = Max(height(root->getLeft()),
                 height(root->getRight()));
    root->setHeight( ht+1 ); // new height for root.
    return root;
}
```

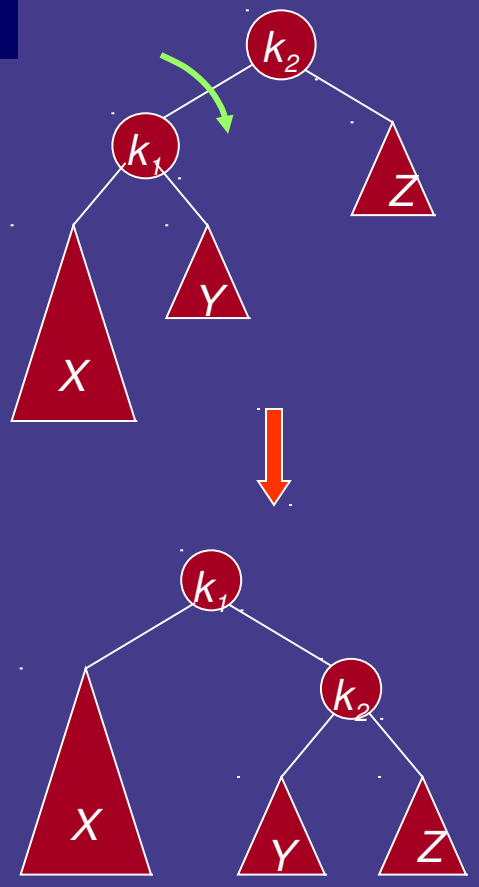
singleRightRotation

```
TreeNode<int>*  singleRightRotation(TreeNode<int>* k2)
{
    if( k2 == NULL ) return NULL;
    // k1 (first node in k2's left subtree)
    // will be the new root
    TreeNode<int>* k1 = k2->getLeft();
    // Y moves from k1's right to k2's left
    k2->setLeft( k1->getRight() );
    k1->setRight(k2);

    // reassign heights. First k2
    int h = Max(height(k2->getLeft()),
                height(k2->getRight()));
    k2->setHeight( h+1 );
    // k2 is now k1's right subtree
    h = Max( height(k1->getLeft()),
             k2->getHeight());
    k1->setHeight( h+1 );

    return k1;
}
```



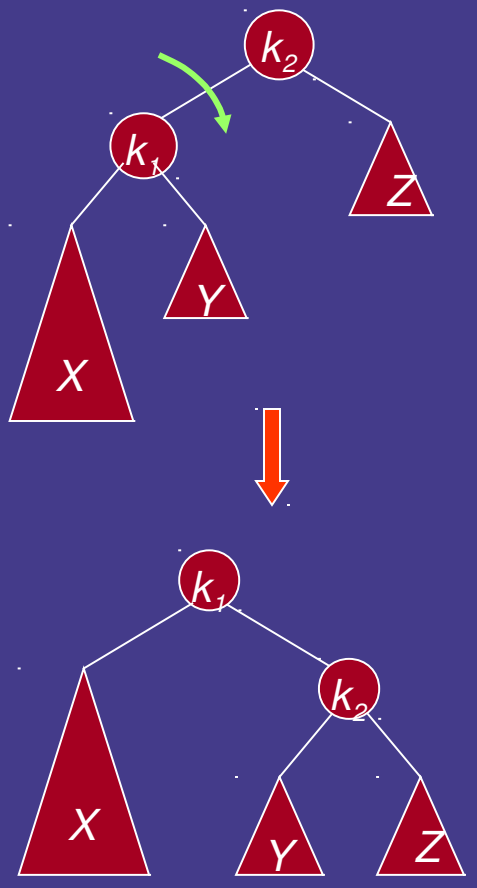
singleRightRotation

```
TreeNode<int>*  singleRightRotation(TreeNode<int>* k2)
{
    if( k2 == NULL ) return NULL;
    // k1 (first node in k2's left subtree)
    // will be the new root
    TreeNode<int>* k1 = k2->getLeft();
    // Y moves from k1's right to k2's left
    k2->setLeft( k1->getRight() );
    k1->setRight(k2);

    // reassign heights. First k2
    int h = Max(height(k2->getLeft()),
                height(k2->getRight()));
    k2->setHeight( h+1 );
    // k2 is now k1's right subtree
    h = Max( height(k1->getLeft()),
             k2->getHeight());
    k1->setHeight( h+1 );

    return k1;
}
```



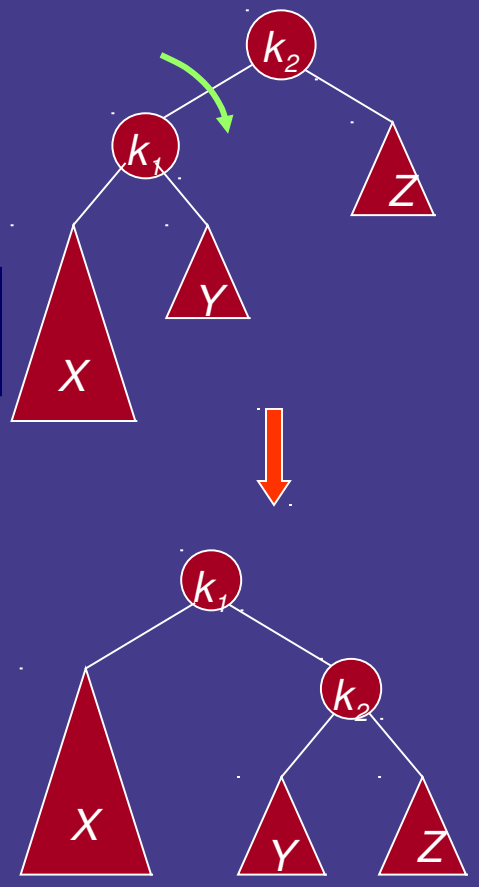
singleRightRotation

```
TreeNode<int>*  singleRightRotation(TreeNode<int>* k2)
{
    if( k2 == NULL ) return NULL;
    // k1 (first node in k2's left subtree)
    // will be the new root
    TreeNode<int>* k1 = k2->getLeft();
    // Y moves from k1's right to k2's left
    k2->setLeft( k1->getRight() );
    k1->setRight(k2);

    // reassign heights. First k2
    int h = Max(height(k2->getLeft()),
                height(k2->getRight()));
    k2->setHeight( h+1 );
    // k2 is now k1's right subtree
    h = Max( height(k1->getLeft()),
             k2->getHeight());
    k1->setHeight( h+1 );

    return k1;
}
```



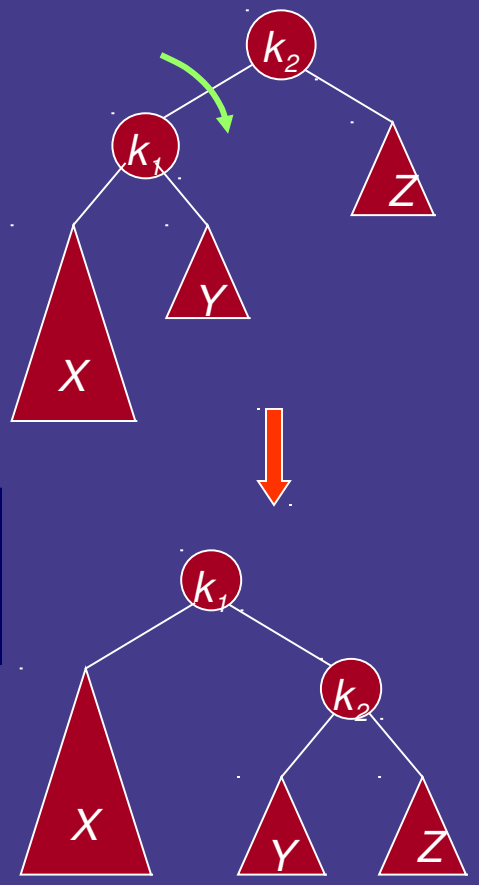
singleRightRotation

```
TreeNode<int>*  singleRightRotation(TreeNode<int>* k2)
{
    if( k2 == NULL ) return NULL;
    // k1 (first node in k2's left subtree)
    // will be the new root
    TreeNode<int>* k1 = k2->getLeft();
    // Y moves from k1's right to k2's left
    k2->setLeft( k1->getRight() );
    k1->setRight(k2);

    // reassign heights. First k2
    int h = Max(height(k2->getLeft()),
                height(k2->getRight()));
    k2->setHeight( h+1 );
    // k2 is now k1's right subtree
    h = Max( height(k1->getLeft()),
             k2->getHeight());
    k1->setHeight( h+1 );

    return k1;
}
```



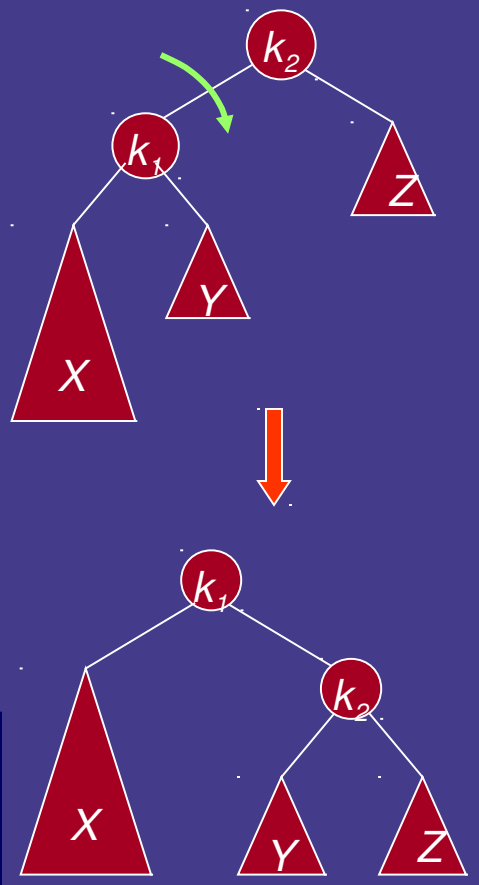
singleRightRotation

```
TreeNode<int>*  singleRightRotation(TreeNode<int>* k2)
{
    if( k2 == NULL ) return NULL;
    // k1 (first node in k2's left subtree)
    // will be the new root
    TreeNode<int>* k1 = k2->getLeft();
    // Y moves from k1's right to k2's left
    k2->setLeft( k1->getRight() );
    k1->setRight(k2);

    // reassign heights. First k2
    int h = Max(height(k2->getLeft()),
                height(k2->getRight()));
    k2->setHeight( h+1 );
    // k2 is now k1's right subtree
    h = Max( height(k1->getLeft()),
             k2->getHeight());
    k1->setHeight( h+1 );

    return k1;
}
```



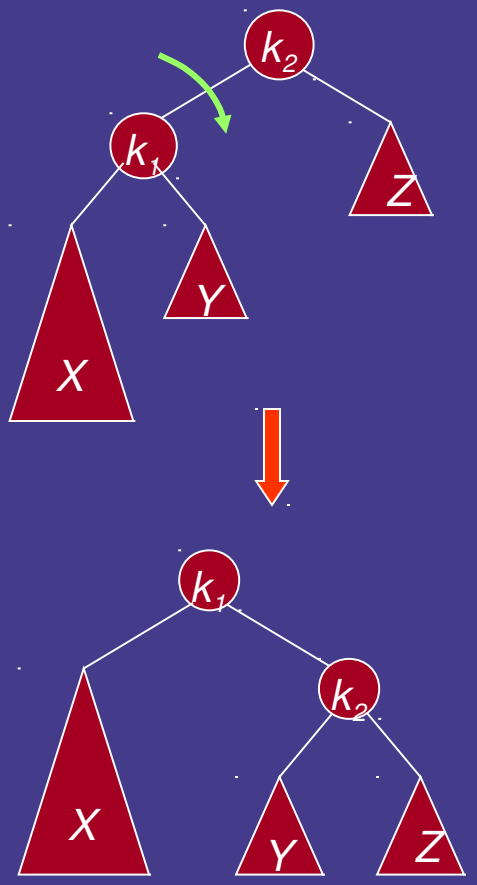
singleRightRotation

```
TreeNode<int>*  singleRightRotation(TreeNode<int>* k2)
{
    if( k2 == NULL ) return NULL;
    // k1 (first node in k2's left subtree)
    // will be the new root
    TreeNode<int>* k1 = k2->getLeft();
    // Y moves from k1's right to k2's left
    k2->setLeft( k1->getRight() );
    k1->setRight(k2);

    // reassign heights. First k2
    int h = Max(height(k2->getLeft()),
                height(k2->getRight()));
    k2->setHeight( h+1 );
    // k2 is now k1's right subtree
    h = Max( height(k1->getLeft()),
             k2->getHeight());
    k1->setHeight( h+1 );

    return k1;
}
```



singleRightRotation

```
TreeNode<int>*  singleRightRotation(TreeNode<int>* k2)
{
    if( k2 == NULL ) return NULL;
    // k1 (first node in k2's left subtree)
    // will be the new root
    TreeNode<int>* k1 = k2->getLeft();
    // Y moves from k1's right to k2's left
    k2->setLeft( k1->getRight() );
    k1->setRight(k2);

    // reassign heights. First k2
    int h = Max(height(k2->getLeft()),
                height(k2->getRight()));
    k2->setHeight( h+1 );
    // k2 is now k1's right subtree
    h = Max( height(k1->getLeft()),
             k2->getHeight());
    k1->setHeight( h+1 );

    return k1;
}
```



height

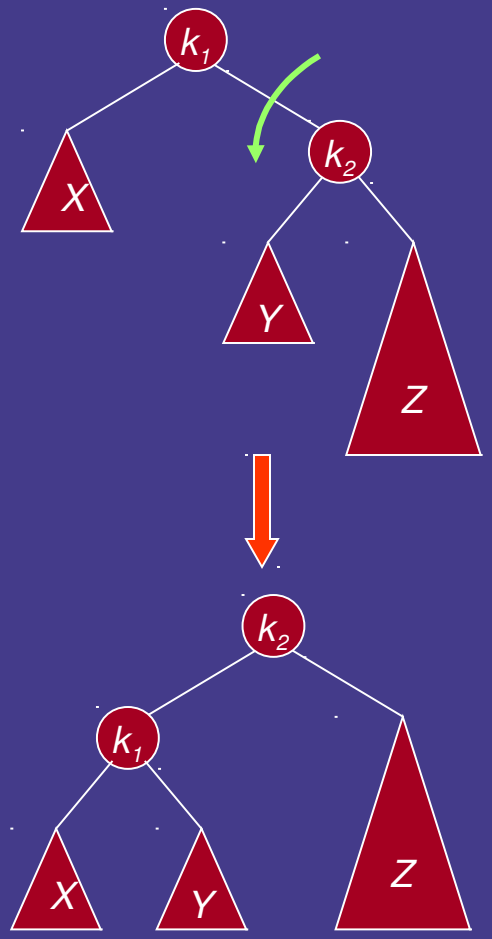
```
int height( TreeNode<int>* node )
{
    if( node != NULL ) return node->getHeight();
    return -1;
}
```

height

```
int height( TreeNode<int>* node )
{
    if( node != NULL ) return node->getHeight();
    return -1;
}
```

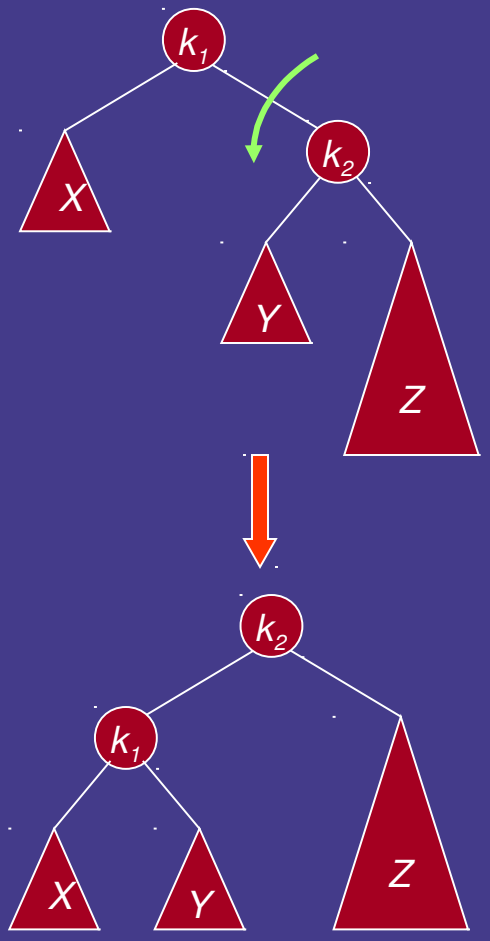
singleLeftRotation

```
TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )
{
    if( k1 == NULL ) return NULL;
    // k2 is now the new root
    TreeNode<int>* k2 = k1->getRight();
    k1->setRight( k2->getLeft() ); // Y
    k2->setLeft( k1 );
    // reassign heights. First k1 (demoted)
    int h = Max(height(k1->getLeft()),
                height(k1->getRight()));
    k1->setHeight( h+1 );
    // k1 is now k2's left subtree
    h = Max( height(k2->getRight()),
             k1->getHeight());
    k2->setHeight( h+1 );

    return k2;
}
```



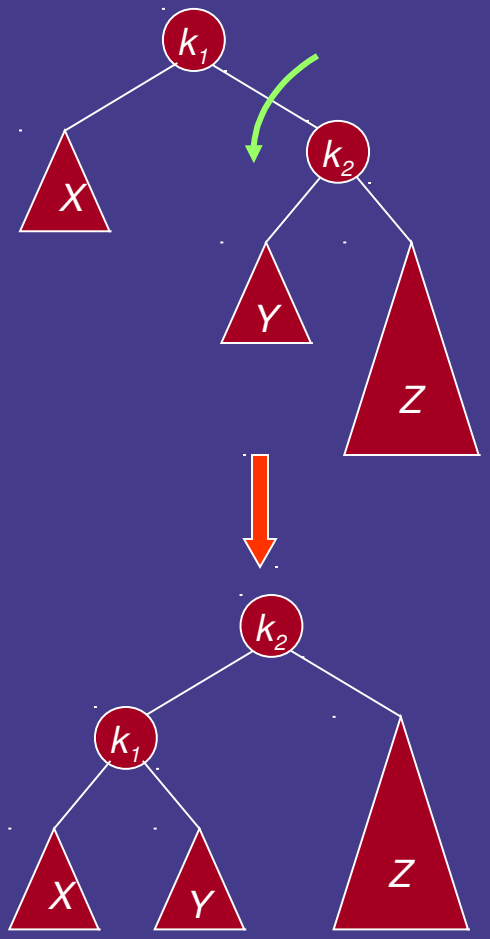
singleLeftRotation

```
TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )
{
    if( k1 == NULL ) return NULL;
    // k2 is now the new root
    TreeNode<int>* k2 = k1->getRight();
    k1->setRight( k2->getLeft() ); // Y
    k2->setLeft( k1 );
    // reassign heights. First k1 (demoted)
    int h = Max(height(k1->getLeft()),
                height(k1->getRight()));
    k1->setHeight( h+1 );
    // k1 is now k2's left subtree
    h = Max( height(k2->getRight()),
             k1->getHeight());
    k2->setHeight( h+1 );

    return k2;
}
```



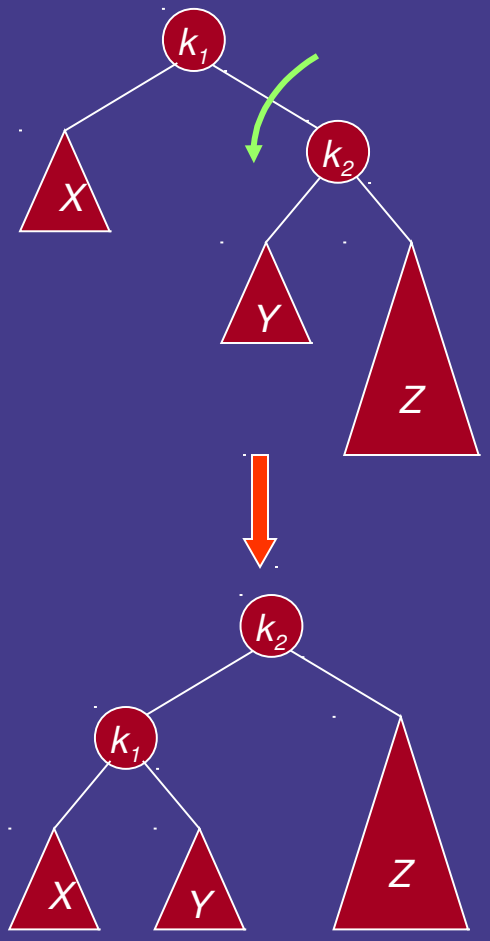
singleLeftRotation

```
TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )
{
    if( k1 == NULL ) return NULL;
    // k2 is now the new root
    TreeNode<int>* k2 = k1->getRight();
    k1->setRight( k2->getLeft() ); // Y
    k2->setLeft( k1 );
    // reassign heights. First k1 (demoted)
    int h = Max(height(k1->getLeft()),
                height(k1->getRight()));
    k1->setHeight( h+1 );
    // k1 is now k2's left subtree
    h = Max( height(k2->getRight()),
             k1->getHeight());
    k2->setHeight( h+1 );

    return k2;
}
```



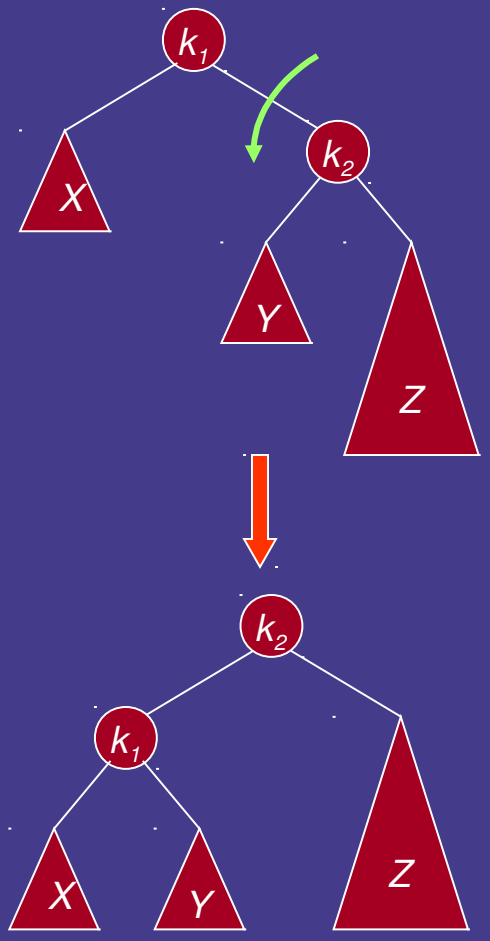
singleLeftRotation

```
TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )
{
    if( k1 == NULL ) return NULL;
    // k2 is now the new root
    TreeNode<int>* k2 = k1->getRight();
    k1->setRight( k2->getLeft() ); // Y
    k2->setLeft( k1 );
    // reassign heights. First k1 (demoted)
    int h = Max(height(k1->getLeft()),
                height(k1->getRight()));
    k1->setHeight( h+1 );
    // k1 is now k2's left subtree
    h = Max( height(k2->getRight()),
             k1->getHeight());
    k2->setHeight( h+1 );

    return k2;
}
```



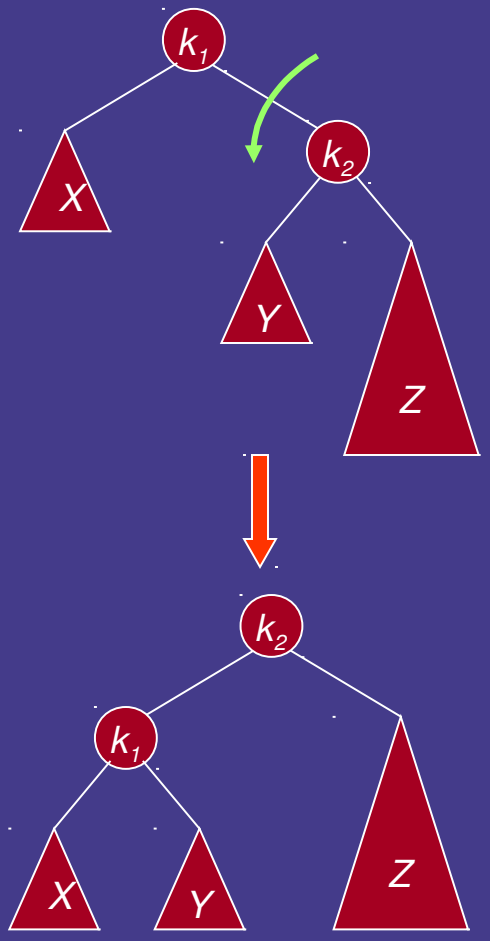
singleLeftRotation

```
TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )
{
    if( k1 == NULL ) return NULL;
    // k2 is now the new root
    TreeNode<int>* k2 = k1->getRight();
    k1->setRight( k2->getLeft() ); // Y
    k2->setLeft( k1 );
    // reassign heights. First k1 (demoted)
    int h = Max(height(k1->getLeft()),
                height(k1->getRight()));
    k1->setHeight( h+1 );
    // k1 is now k2's left subtree
    h = Max( height(k2->getRight()),
             k1->getHeight());
    k2->setHeight( h+1 );

    return k2;
}
```



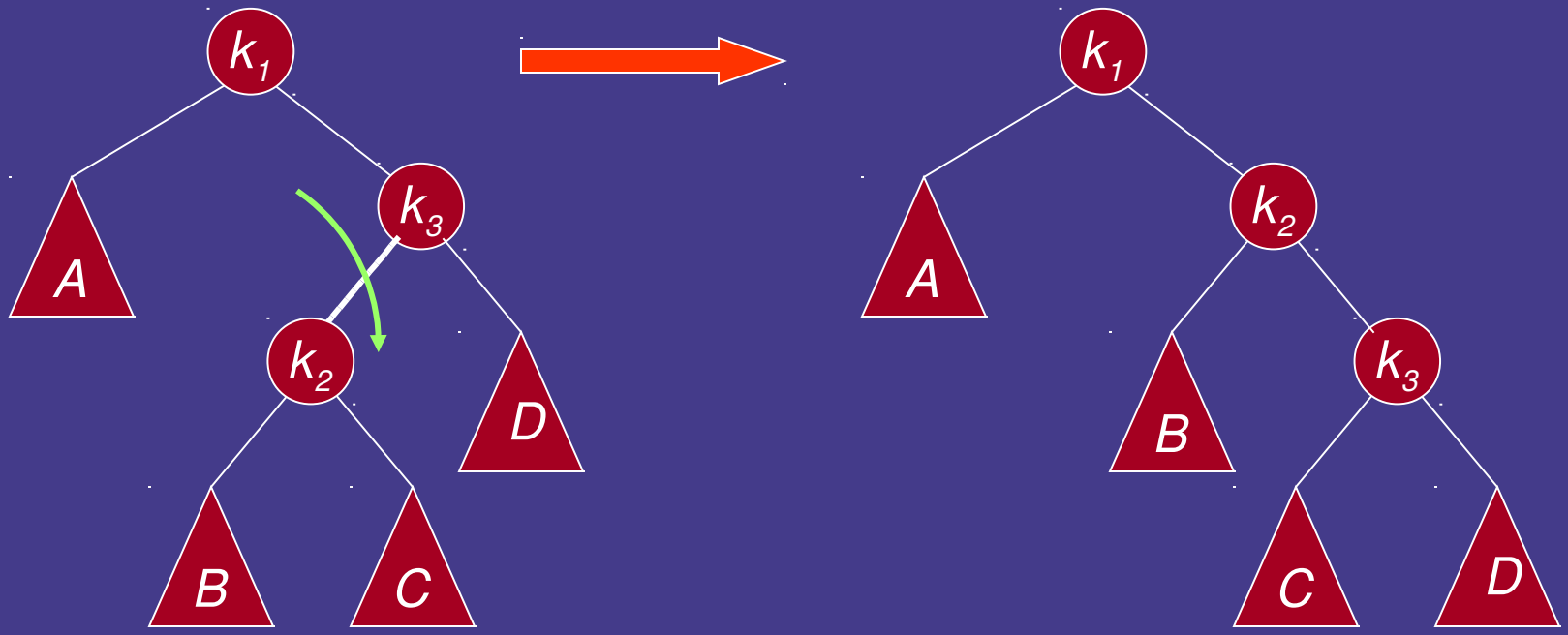
singleLeftRotation

```
TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )
{
    if( k1 == NULL ) return NULL;
    // k2 is now the new root
    TreeNode<int>* k2 = k1->getRight();
    k1->setRight( k2->getLeft() ); // Y
    k2->setLeft( k1 );
    // reassign heights. First k1 (demoted)
    int h = Max(height(k1->getLeft()),
                height(k1->getRight()));
    k1->setHeight( h+1 );
    // k1 is now k2's left subtree
    h = Max( height(k2->getRight()),
             k1->getHeight());
    k2->setHeight( h+1 );

    return k2;
}
```



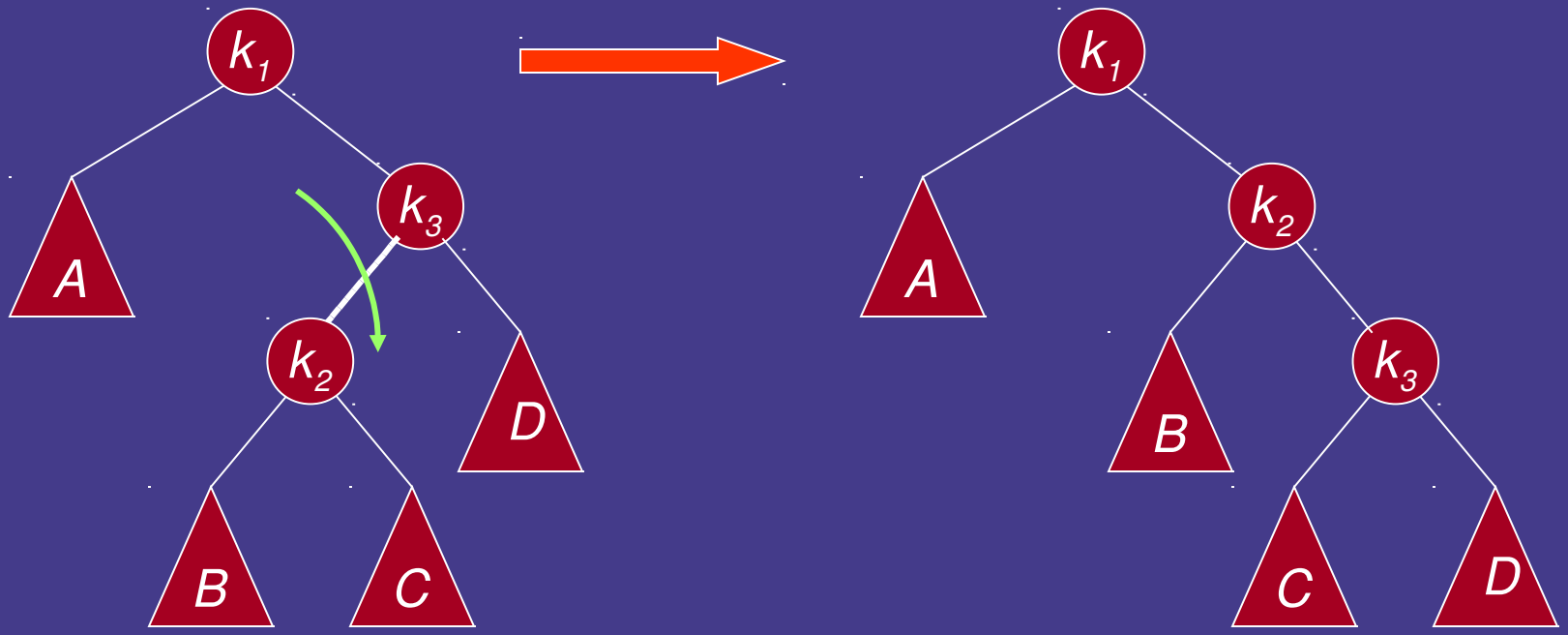
doubleRightLeftRotation

```
TreeNode<int>* doubleRightLeftRotation(TreeNode<int>* k1)
{
    if( k1 == NULL ) return NULL;
    // single right rotate with k3 (k1's right child)
    k1->setRight( singleRightRotation(k1->getRight()));
    // now single left rotate with k1 as the root
    return singleLeftRotation(k1);
}
```



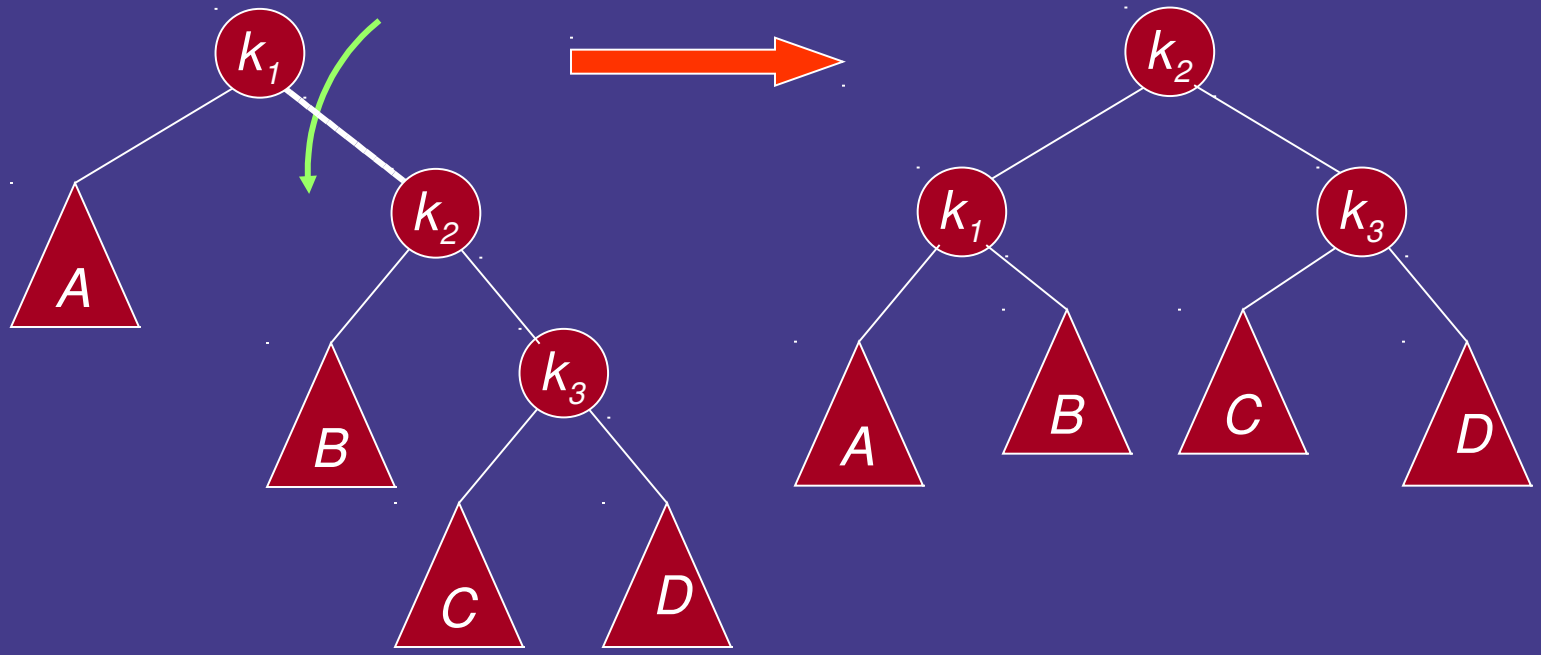
doubleRightLeftRotation

```
TreeNode<int>* doubleRightLeftRotation(TreeNode<int>* k1)
{
    if( k1 == NULL ) return NULL;
    // single right rotate with k3 (k1's right child)
    k1->setRight( singleRightRotation(k1->getRight()));
    // now single left rotate with k1 as the root
    return singleLeftRotation(k1);
}
```



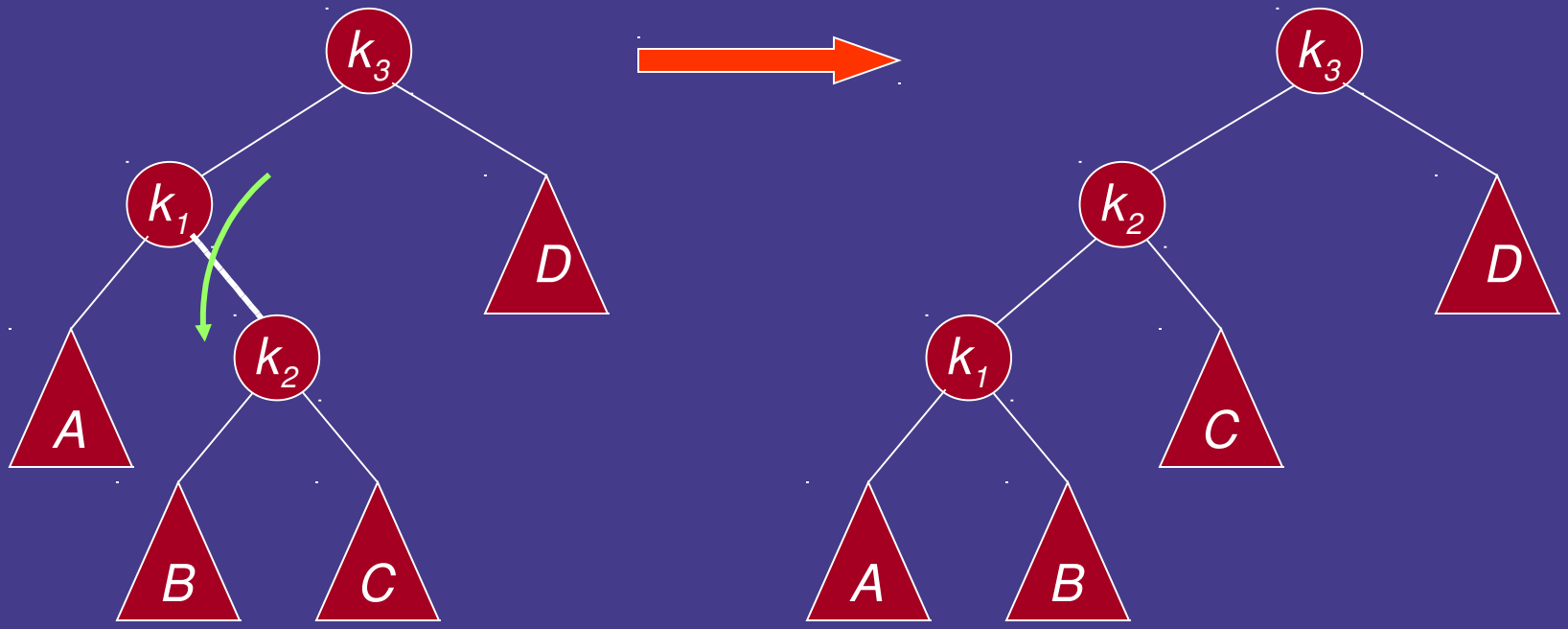
doubleRightLeftRotation

```
TreeNode<int>* doubleRightLeftRotation(TreeNode<int>* k1)
{
    if( k1 == NULL ) return NULL;
    // single right rotate with k3 (k1's right child)
    k1->setRight( singleRightRotation(k1->getRight()));
    // now single left rotate with k1 as the root
    return singleLeftRotation(k1);
}
```



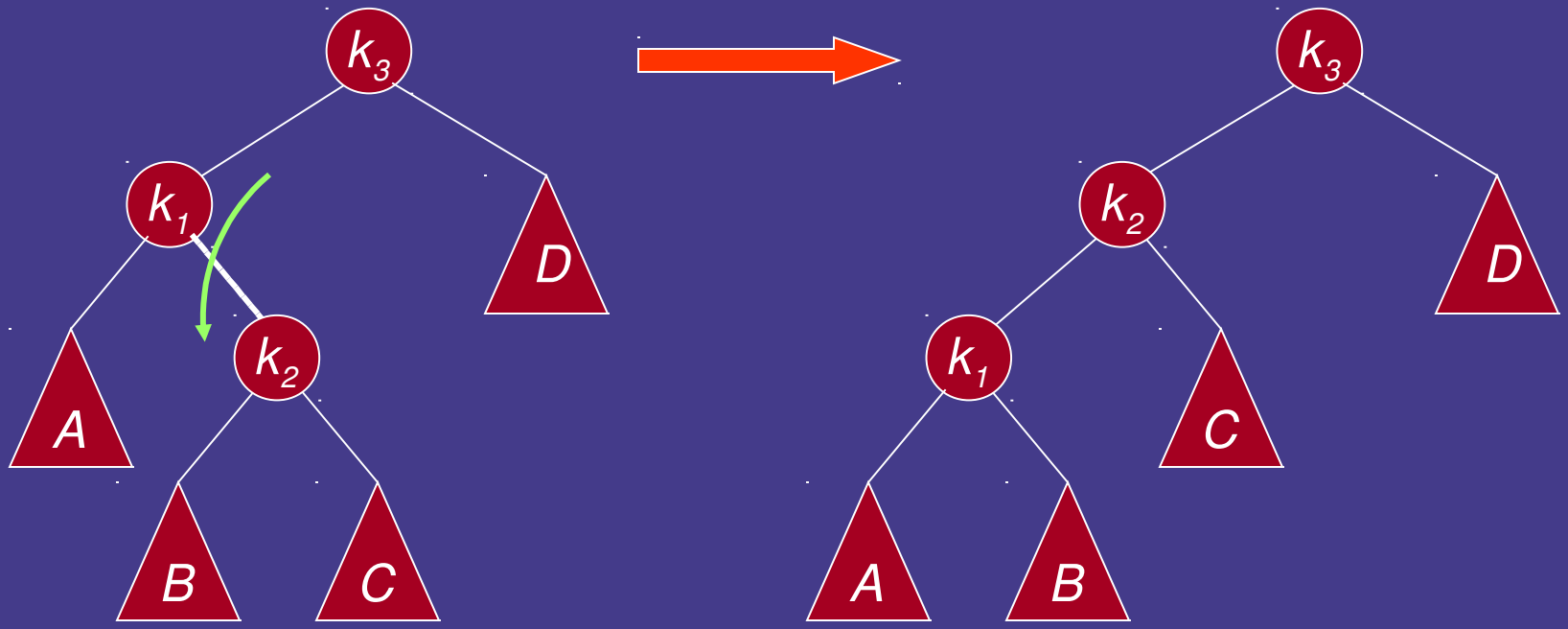
doubleRightLeftRotation

```
TreeNode<int>* doubleLeftRightRotation(TreeNode<int>* k3)
{
    if( k3 == NULL ) return NULL;
    // single left rotate with k1 (k3's left child)
    k3->setLeft( singleLeftRotation(k3->getLeft()));
    // now single right rotate with k3 as the root
    return singleRightRotation(k3);
}
```



doubleRightLeftRotation

```
TreeNode<int>* doubleLeftRightRotation(TreeNode<int>* k3)
{
    if( k3 == NULL ) return NULL;
    // single left rotate with k1 (k3's left child)
    k3->setLeft( singleLeftRotation(k3->getLeft()));
    // now single right rotate with k3 as the root
    return singleRightRotation(k3);
}
```



doubleRightLeftRotation

```
TreeNode<int>* doubleLeftRightRotation(TreeNode<int>* k3)
{
    if( k3 == NULL ) return NULL;
    // single left rotate with k1 (k3's left child)
    k3->setLeft( singleLeftRotation(k3->getLeft()));
    // now single right rotate with k3 as the root
    return singleRightRotation(k3);
}
```