Discussion Session 8: AVL Trees

CS 225: Data Structures
& Software Principles

Agenda

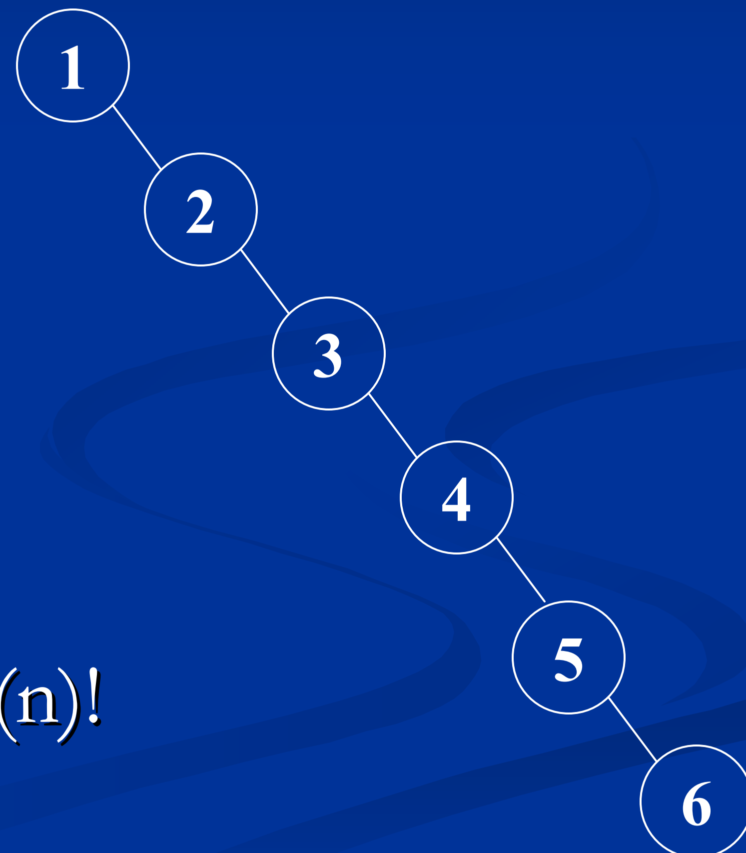
- **Unbalanced Trees**
- AVL Trees
- Balancing at each Node
- Rotations
- Operations on AVL Trees
  - Insert
  - Remove
- Implementation of AVL Trees

By the end of this class, you

- Need to
  - Understand why balanced trees are desirable
  - Be able to calculate balance at each node in an AVL tree
  - Understand the 4 different types of rotations and their effects
  - Be able to restructure a tree after insertion/deletion
- Ought to be able to implement an AVL Tree

Motivation: BST

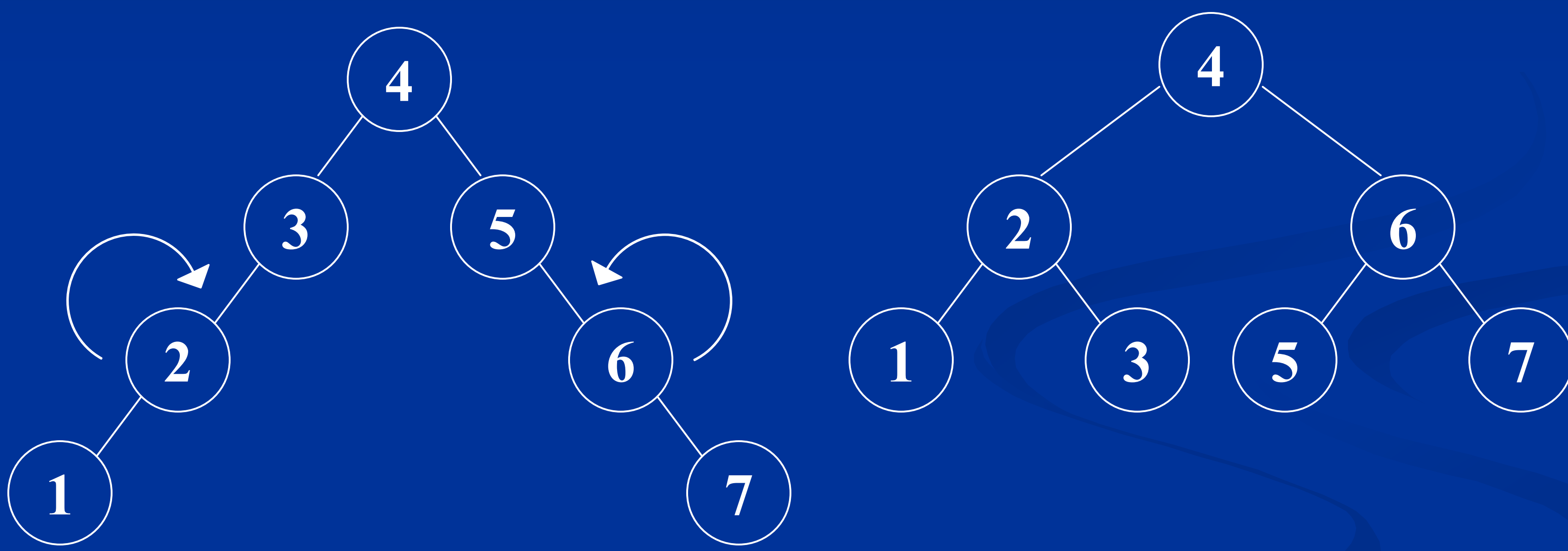
- Average case is O(log n)



- But worst case search is O(n)!

Rotation helps!

Single Left and Single Right Rotations



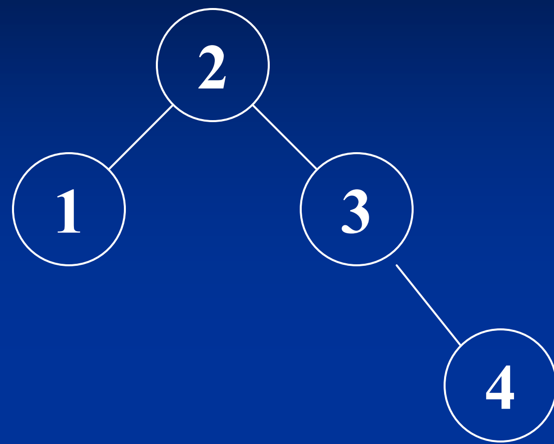
Balanced Trees

- By mechanisms similar to the rotations we've just seen, you can always prevent a tree from getting too unbalanced.
- Different kinds of balanced Trees
  - AVL Trees
  - Red-Black Trees
  - ***Splay Trees***

Agenda

- Unbalanced Trees
- **AVL Trees**
- Balancing at each Node
- Rotations
- Operations on AVL Trees
  - Insert
  - Remove
- Implementation of AVL Trees

AVL trees



- Named after creators: Adelson-Velskii and Landis
- Additional constraint to BST: balance
  - At each node, subtree heights differ by no more than 1.
- Every AVL Tree with n nodes has height less than 1.44 log n (i.e. O(log n))
- Worst case search, insert and remove is O(log n)

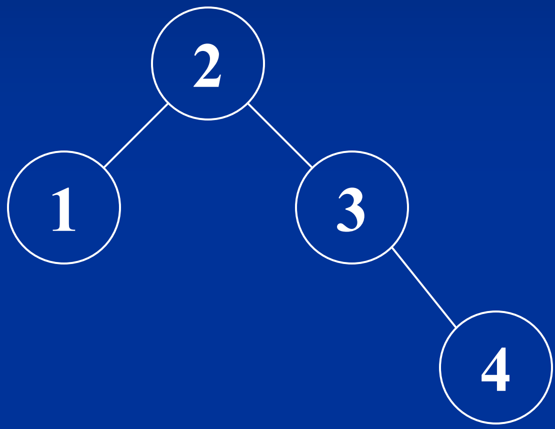
Agenda

- Unbalanced Trees
- AVL Trees
- **Balancing at each Node**
- Rotations
- Operations on AVL Trees
  - Insert
  - Remove
- Implementation of AVL Trees

AVL Tree Node Balance

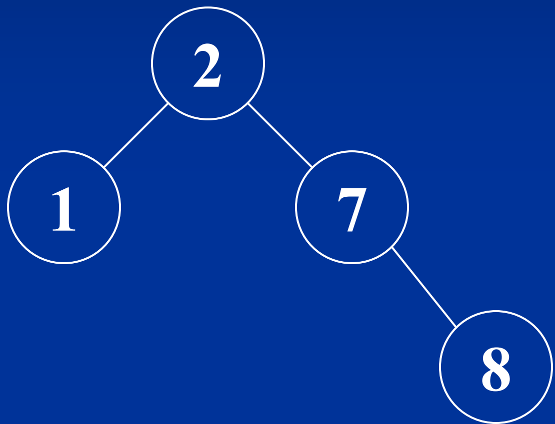
- LeftHeight(T) = ***0*** if LeftChild(T)==NULL
  1 + Height(T->Left) otherwise
- RightHeight(T) = ***0*** if RightChild(T)==NULL
  1 + Height(T->Right) otherwise
- Height(T) = max {LeftHeight ,RightHeight}
- Balance(T) = RightHeight(T) - LeftHeight(T)
- A node is only allowed to have a balance of:
  -1, 0, +1

Example AVL Trees



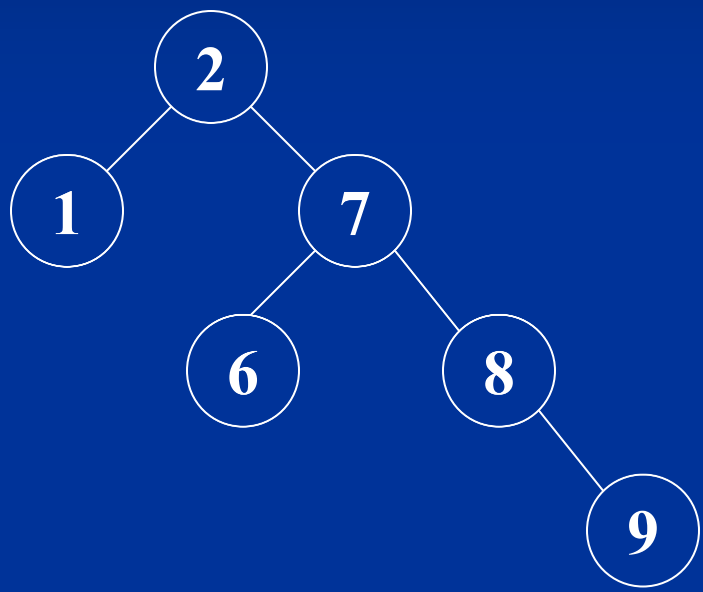
- 1
  - Height
  - Balance
- 4
  - Height
  - Balance
- 3
  - Height
  - Balance
- 2
  - Height
  - Balance

Example AVL Trees



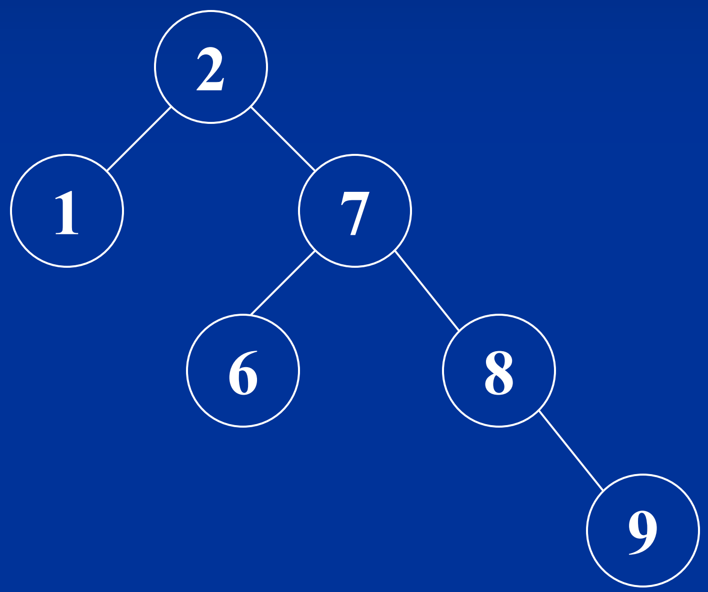
- 1
  - Height 0
  - Balance 0
- 8
  - Height 0
  - Balance 0
- 7
  - Height 1
  - Balance 1
- 2
  - Height 2
  - Balance 1

More Example Trees



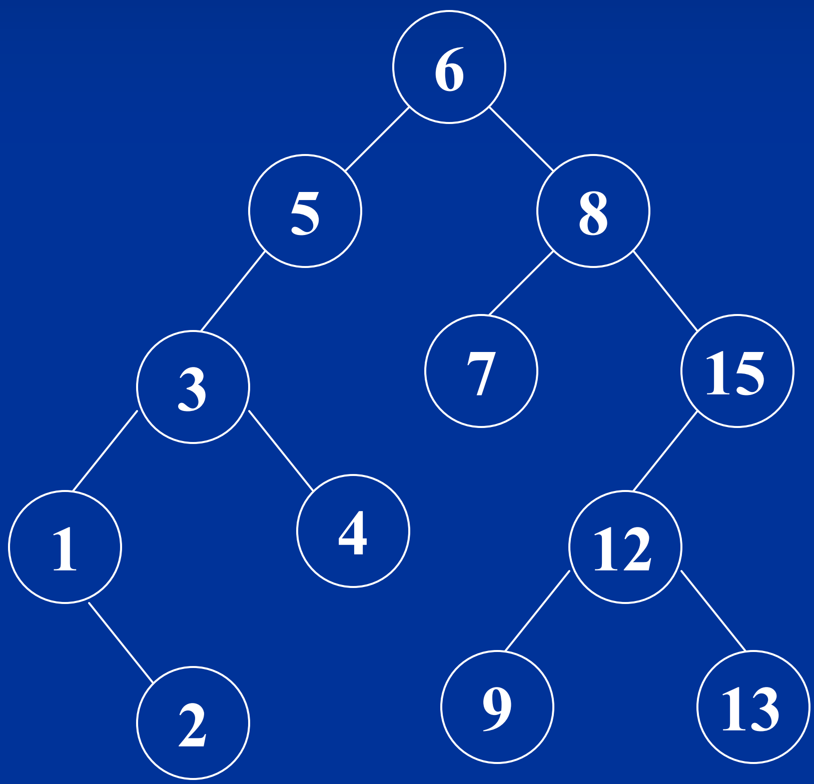
- 1 = ?, ?
- 6 = ?, ?
- 9 = ?, ?
- 8 = ?, ?
- 7 = ?, ?
- 2 = ?, ?

More Example Trees



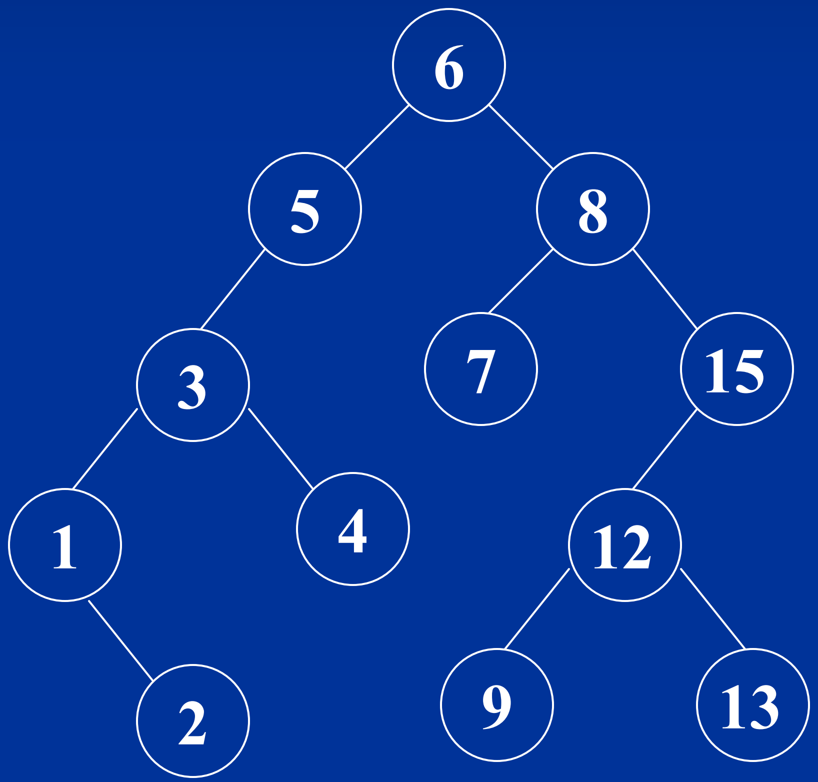
- 1 = 0, 0
- 6 = 0, 0
- 9 = 0, 0
- 8 = 1, 1
- 7 = 2, 1
- 2 = 3, **2**

More Example Trees



- 2 = ?, ?
- 1 = ?, ?
- 4 = ?, ?
- 3 = ?, ?
- 5 = ?, ?
- 7 = ?, ?
- 9 = ?, ?
- 13 = ?, ?
- 12 = ?, ?
- 15 = ?, ?
- 8 = ?, ?
- 6 = ?, ?

More Example Trees



- 2 = 0, 0
- 1 = 1, 1
- 4 = 0, 0
- 3 = 2, -1
- 5 = 3, -3
- 7 = 0, 0
- 9 = 0, 0
- 13 = 0, 0
- 12 = 1, 0
- 15 = 2, -2
- 8 = 3, 2
- 6 = 4, 0

AVL Trees

- Binary Search Tree
- Operations similar to those in BSTs except
  - Remove & Insert
- Remove and Insert must make sure the tree stays balanced!
- Require re-balancing (Rotation) in some cases

Agenda

- Unbalanced Trees
- AVL Trees
- Balancing at each Node
- **Rotations**
- Operations on AVL Trees
  - Insert
  - Remove
- Implementation of AVL Trees

Rotations

- After running a BST insert/remove, must rebalance using rotations (rotate when the balance becomes –2 or +2).

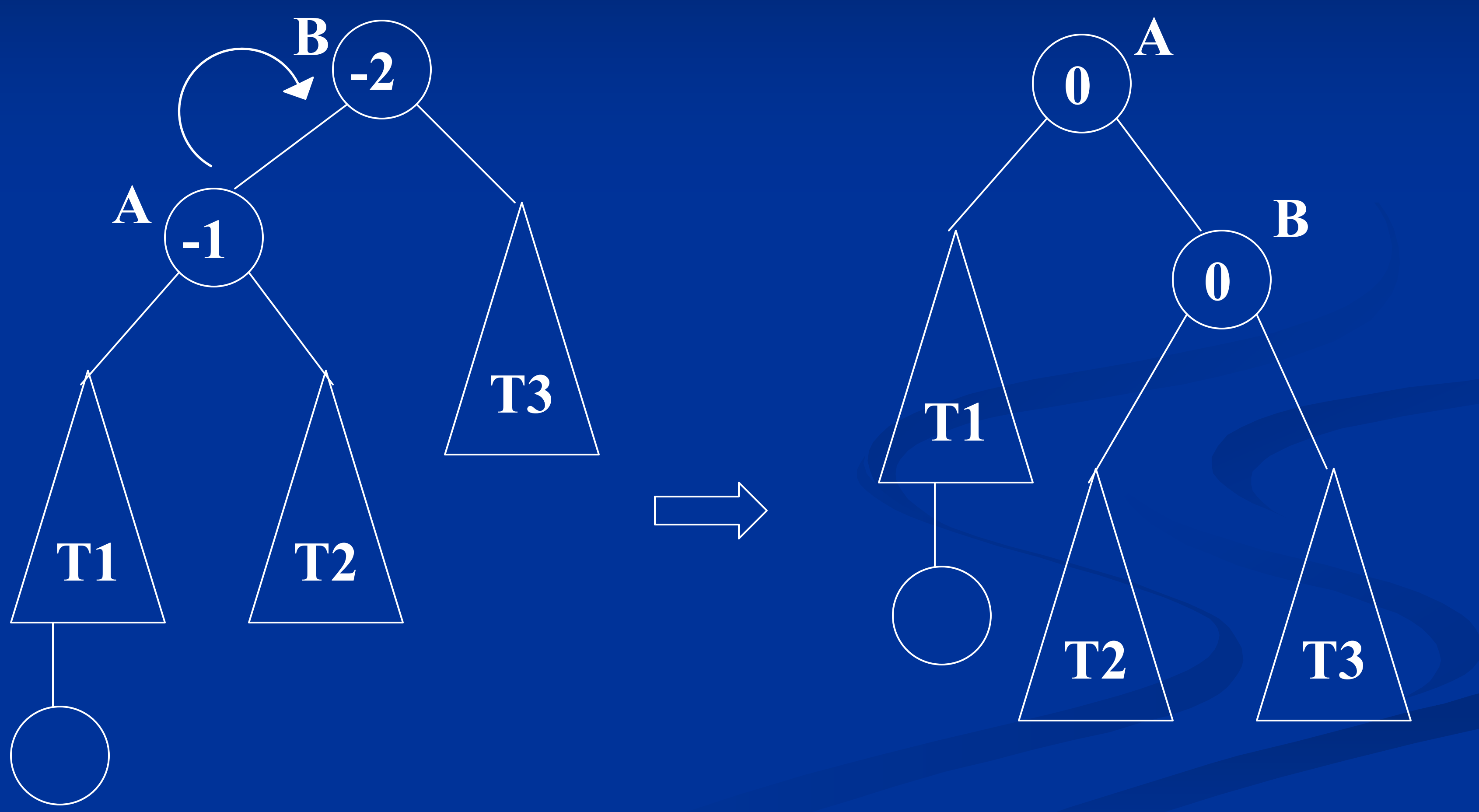
- Must know:
  - ***HOW?***: how to apply a rotation
  - ***WHICH?***: which type of rotation to use
  - ***WHERE?***: where to apply the rotation

How?

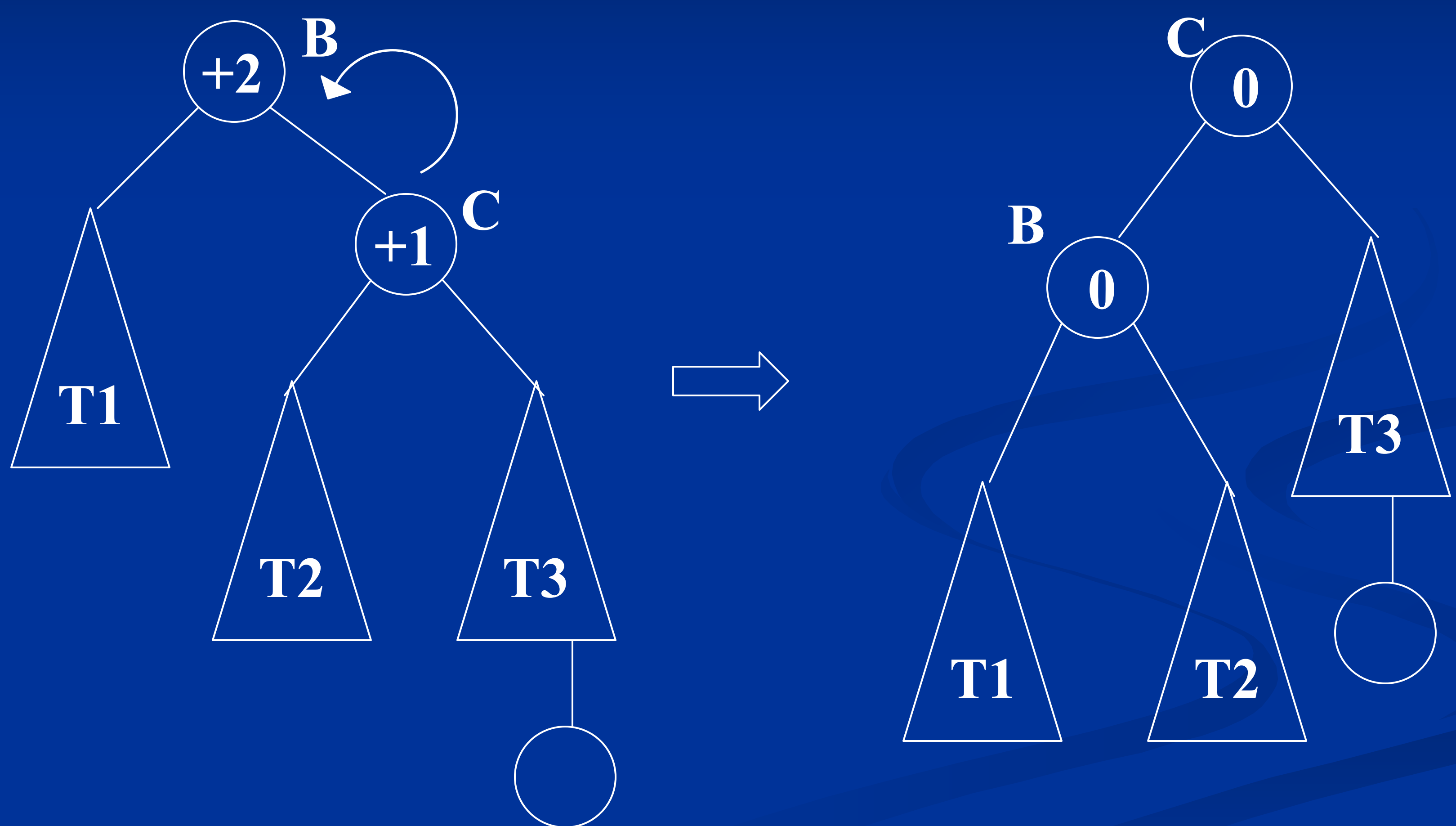
- There are 4 different rotations:
  - single right
  - single left
  - double rotation LR
  - double rotation RL

- If you know single right, you can derive the other three!

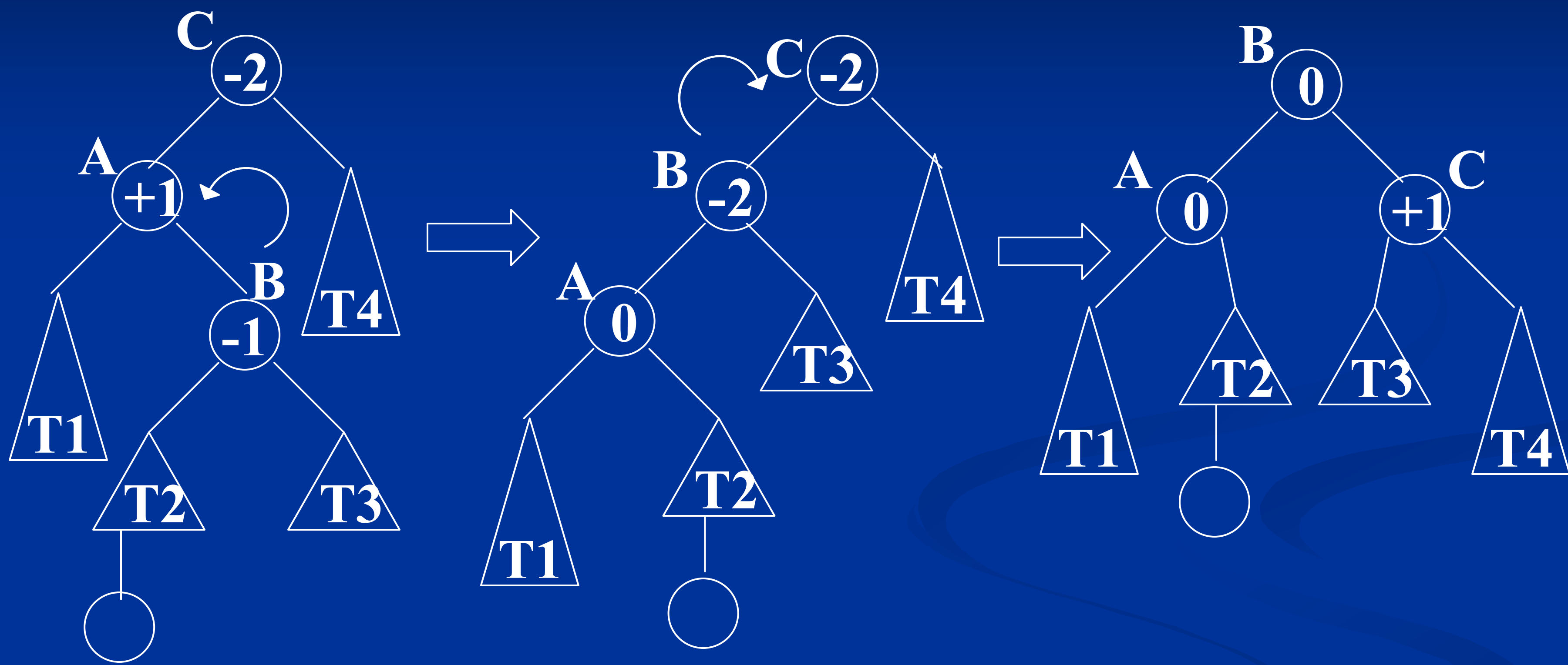
Single Right Rotation



Single Left Rotation

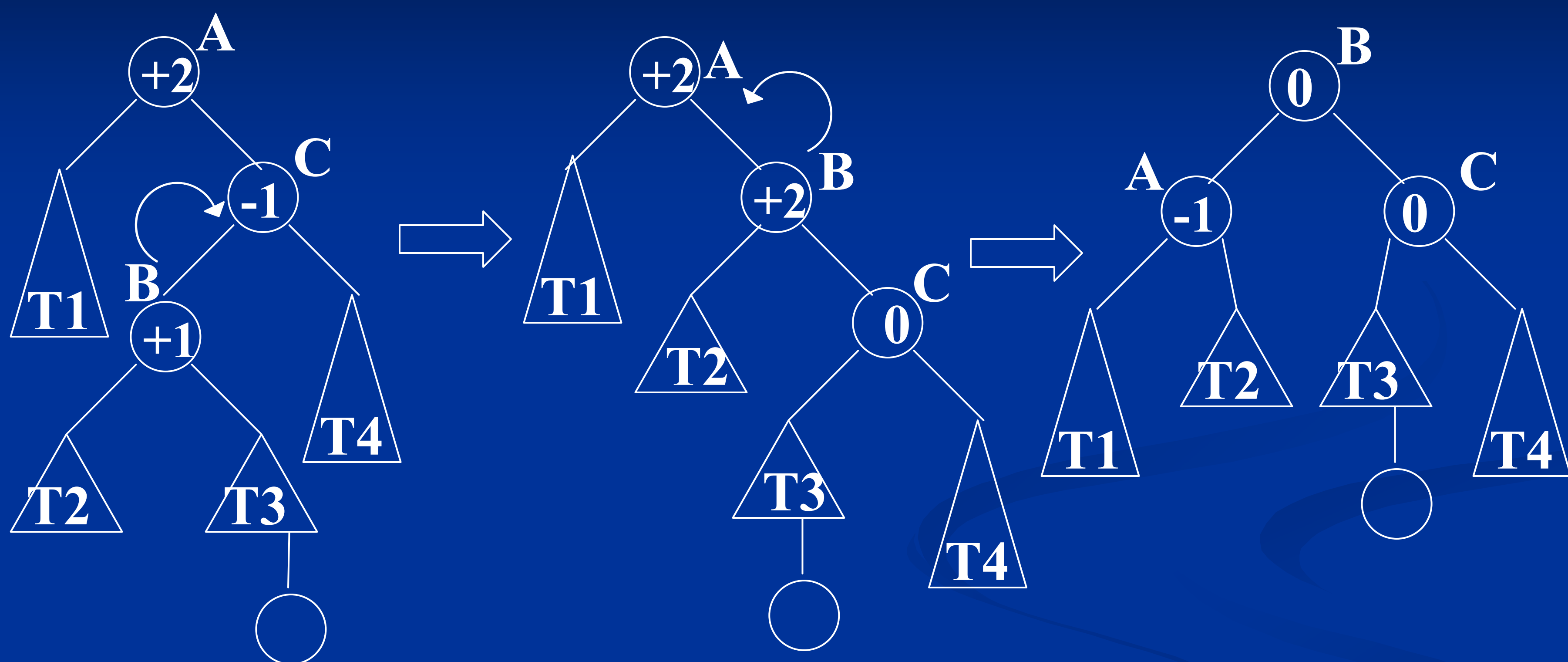


Double Rotation (LR)



This is an LR Double rotation : requires a single Left followed by a single Right rotation

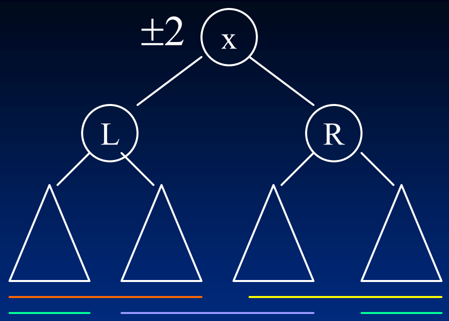
Double Rotation (RL)



This is an RL Double rotation : requires a single Right followed by a single Left rotation

Which?

- Find the new balance for the critical node and its child on the path to the inserted/deleted node.
- If these balance values are of the same sign, then you need only a single rotation. Otherwise, you need to do a double rotation.
- In case of a single rotation, additionally, if the sign on the critical node is +, do a left rotation and if it is -, do a right rotation.
- In case of a double rotation, additionally, if the sign on the critical node is +, do an RL and if it is -, do an LR.



Which?

- Type: Find the new balance for the critical node and child on the path to the inserted/deleted node.
  - If balance values are the *same sign*, then you need only a single rotation.
  - Else, you need to do a double rotation.
- Direction of Single Rotation:
  - if the sign on the critical node is +, do a left rotation
  - else if it is -, do a right rotation
- Direction of Double Rotation:
  - if the sign on the critical node is +, do an RL (double left)
  - else if it is -, do an LR (double right)

Where?

- When you insert or remove a node, height *may* change and balance *will* change.
- As you move back up the tree, must adjust heights and calculate a new balance.
- If new balance is +2 or -2, *this* is the node around which to perform a rotation.

Rotation Summary

- Memorize single right rotation
- Single left is mirror image of single right
- Double rotations are a combination of two single rotations:
  - 1st on child in first direction
  - 2nd on **critical node** in second direction

Rotation Effects

- Reduce balance of critical node from +2 or -2 to 0
- Reduce height of critical node by 1

Agenda

- Unbalanced Trees
- AVL Trees
- Balancing at each Node
- Rotations
- **Operations on AVL Trees**
  - Insert
  - Remove
- Implementation of AVL Trees

Insert

- Perform BST Insert
- New node is a leaf and has height 0, balance 0
- Work way up tree and compute new balance (starting with new node's parent)
  - if balance goes from ± 1 to 0, stop.
  - if balance goes from 0 to ± 1, then height has also changed. Proceed up.
  - if balance goes from +1 to +2 or -1 to -2, do a rotation on this node! … then stop.

Remove

- Perform BST remove (in-order successor or predecessor replaces removed node).
- Compute new balance of parent:
  - if balance goes from 0 to ± 1…done
  - if balance goes from ±1 to 0, height has also changed. Proceed up.
  - If balance goes from +1 to +2 or -1 to -2, do a rotation. Height has also changed, proceed up.

Operation Summary

- Insert has at most one single or one double rotation.
- Remove can have a rotation at every level.

Agenda

- Unbalanced Trees
- AVL Trees
- Balancing at each Node
- Rotations
- Operations on AVL Trees
  - Insert
  - Remove
- **Implementation of AVL Trees**

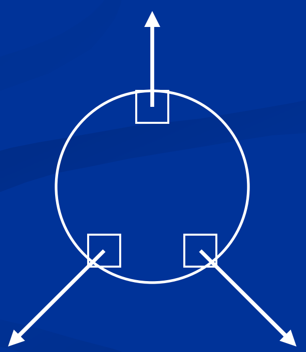
Source

- AVL Tree Code found at:
  ~cs225/src/library/08-avl/
  - Designed in the spirit of C++ STL interfaces
  - typedefs, 4 types of iterators, functions to return iterators, regular operations
  - Stores key-element pairs
    (template <class Ktype, class Etype>
  - Code that follows may not contain all the gory details!

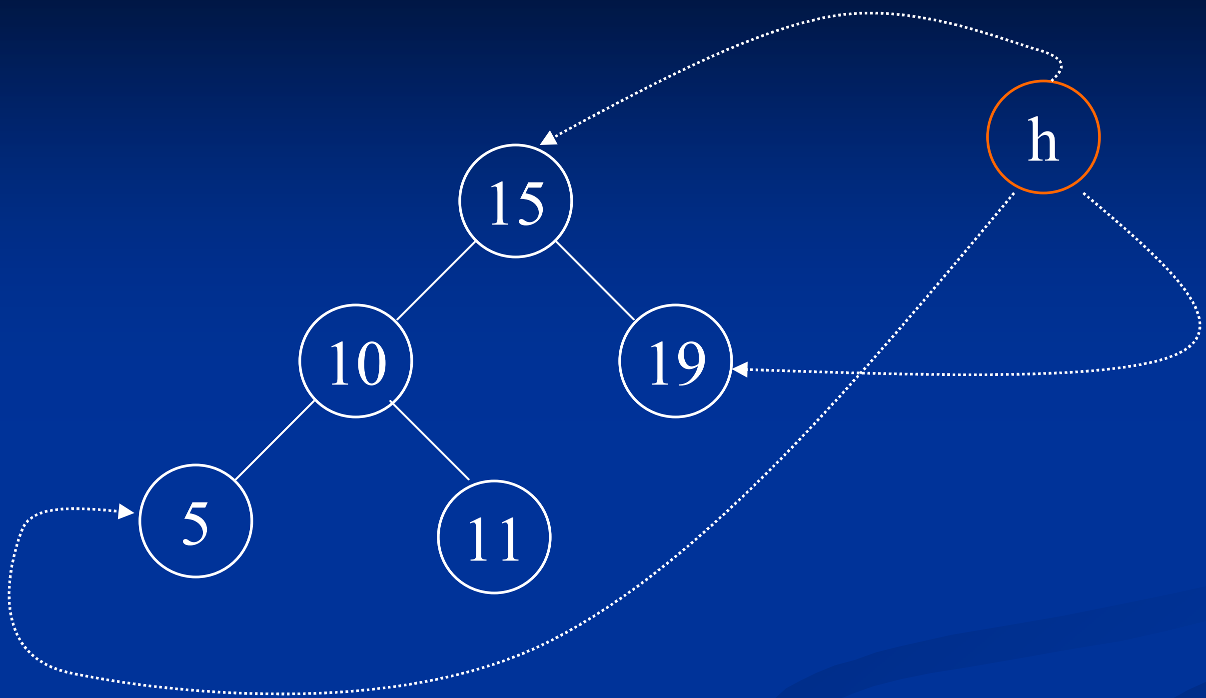
- The (not very good) applet can be found at
  http://www.binarytreesome.com/applets/latest/applet.html

class avl_tree Private Data

```
class avl_tree_node {
  public:
    avl_tree_node( ) : element( ), left(NULL), right(NULL), parent(NULL),
            height(0) {}
    avl_tree_node(pair<Ktype, Etype> elmt,
            avl_tree_node* leftPtr = NULL, avl_tree_node* rightPtr = NULL,
            avl_tree_node* parentPtr = NULL, int hgt = 0) ;

    pair<Ktype, Etype> element;          // value element of node
    avl_tree_node* left;                 // pointer to left subtree
    avl_tree_node* right;                // pointer to right subtree
    avl_tree_node* parent;               // pointer to parent
    int height;                          // height of node
};
avl_tree_node* headerNode;               // pointer to header node [ end() ]
int treeSize;                            // number of nodes in tree
```



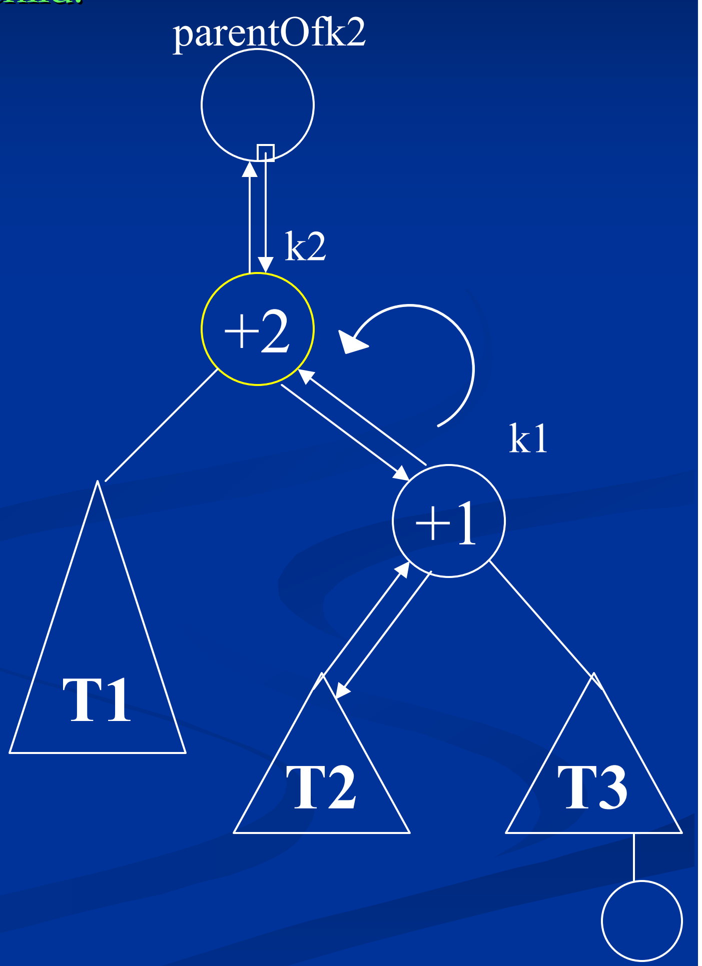
headerNode



- headerNode->parent = root node [root( )]
- headerNode->left = leftmost key in tree [leftmost( )]
- headerNode->right = rightmost key in tree [rightmost( )]
- Useful for working with iterators

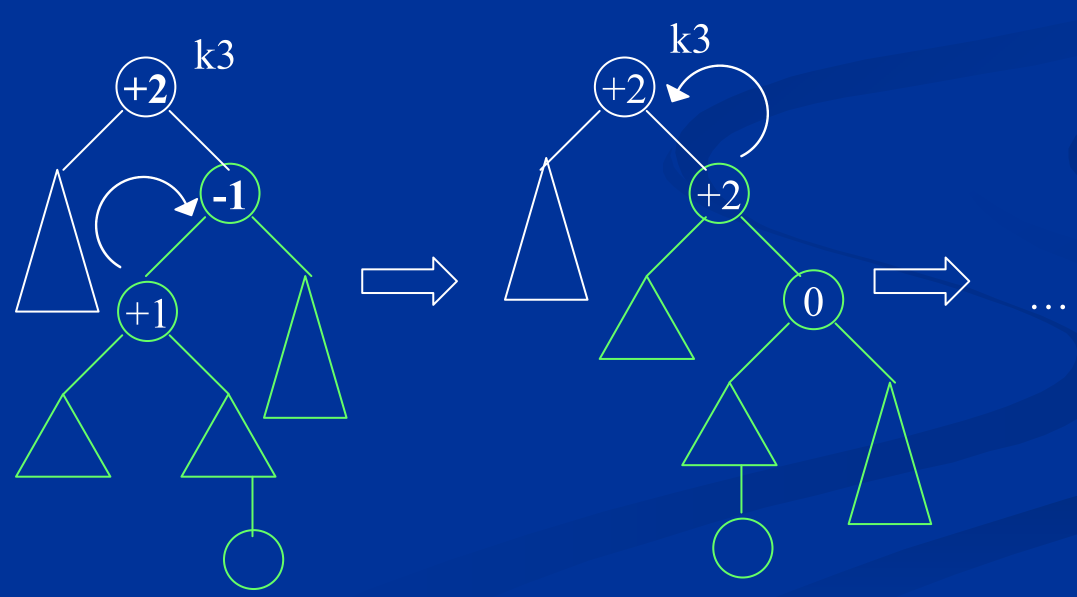
single_left_rotation()

```
//   Performs a rotation between a node (k2) and its right child, updating the
//  heights of both.  Can be called only if k2 has a right child.
void single_left_rotation( avl_tree_node* & k2 ) {
    avl_tree_node *k1 = k2->right;
    avl_tree_node *parentOfk2 = k2->parent;

    // rotate k1 up to "root" and k2 down to left
    k2->right = k1->left;
    if (k2->right != NULL)     k2->right->parent = k2;
    k1->left = k2;
    k2->parent = k1;

    k1->parent = parentOfk2;

    calculate_height(k2);
    calculate_height(k1);
    // reset the root
    k2 = k1;
}
```



double_left_rotation()

```
//   Performs a right-left rotation, also known as a double left rotation.  Can
//   only be called if k3 has a right child and k3's right child has a left child.
void double_left_rotation( avl_tree_node* & k3 ) {
    single_right_rotation(k3->right);
    single_left_rotation(k3);
}
```



Public insert( ) Interface

```
//  - return value : a pair that contains an iterator to the inserted
// element and a boolean integer telling us if the insertion was
// performed (1) or if we instead were trying to insert a duplicate
// key and thus stopped (0).
pair<iterator, int> insert(const pair<Ktype, Etype>& insElem) {
   if (count(key(insElem)) == 1)        // this key already appears
         // note the constructor call
         return pair<iterator, int>( find(key(insElem)), 0);
   else
         return pair<iterator, int>(
                         insert(insElem, root( ), headerNode), 1);
}
```

private insert( )

```
iterator insert(const pair& insElem, avl_tree_node*& TN, avl_tree_node* parentOfTN) {
    // we've found the spot to insert at
    if (TN == NULL) {
        // Create and insert the node with value insElem
        TN = new avl_tree_node();
        TN->element = insElem;
        TN->height = 0;
        TN->parent = parentOfTN;

        // new code
        if ( (treeSize==0) || (key(insElem) < key(headerNode->left->element)) )
                headerNode->left = TN;
        if ( (treeSize==0) || (key(insElem) > key(headerNode->right->element)) )
                headerNode->right = TN;

        treeSize++;
        return iterator(TN);
    }
```

…private insert continued…

```
else if (key(insElem) < key(TN->element) ) {
    // insert in the left subtree
    iterator tempIt = insert(insElem, TN->left, TN);
    // check balance condition
    if ( node_height(TN->left) - node_height(TN->right) == 2) {
        // if unbalanced, determine type of rotation
        if (key(insElem) < key(TN->left->element) ) {
            // if inserted in left-most subtree, do single
            single_right_rotation(TN);
        } else {
            // if inserted in interior subtree, do double
            double_right_rotation(TN);
        }
    } else {
        // if balanced, recalculate height (due to an insertion further down the tree)
        calculate_height(TN);
    }
    return tempIt;
}
```

…private insert( ) finished.

```
else if (key(insElem) > key(TN->element) ) {
    // insert into right subtree…

    // …same as previous slide, but anywhere there's a "left" replace with
    // "right" and vice versa.
}
// else insElem is in the tree already--do nothing
else
    return end( );
```

private remove( ) components

```
void remove(pair & remElem, avl_tree_node* & TN, avl_tree_node* parentOfTN) {
    avl_tree_node *tmp_cell, *tmpCellParent, *newCell;      // used in 2 child remove case
    if (TN == NULL)
        return;
    else if ( key(remElem) < key(TN->element) ) {
        // remove from the left subtree
        remove(remElem, TN->left, TN);
        // check balance condition, perform left rotation if necessary, update heights
        // …
    }
    else if (key(remElem) > key(TN->element))  {
        // remove from the right subtree
        remove(remElem, TN->right, TN);
        // check balance condition, perform right rotation if necessary, update heights
        // …
    }
    else { // remElem == TN->element
        // 2 child case, 1 child case, 0 child case, fix headerNode if necessary
        // see next slides…
```

…private remove() 2-child case…

```
// 2 child case -- replace TN (node we want to delete) with its inorder successor node
if ((TN->left != NULL) && (TN->right != NULL) ) {
        // find inorder successor
        tmp_cell = TN->right;  // we just checked, TN->right isn't NULL
        while (tmp_cell->left != NULL)
                tmp_cell = tmp_cell->left;

        // Put IOS node in TN's place, and put a new node in IOS's place.  We do this to:
        // 1) make sure any iterators pointing to IOS are still valid
        // 2) allow recursive calls to remove( ) to delete the new IOS and balance the tree
        tmpCellParent = tmp_cell->parent;
        newCell = new avl_tree_node();

        // make newCell a copy of tmp_cell
        newCell->right = tmp_cell->right;
        newCell->left = tmp_cell->left;
        newCell->element = tmp_cell->element;
        newCell->height = tmp_cell->height;
        newCell->parent = tmp_cell->parent;
```

…remove( ) 2-child case finished…

```
    // if inorder successor is left child of its parent
    if (tmpCellParent->left == tmp_cell)            tmpCellParent->left = newCell;
    // else if inorder successor is right child of its parent
    else if (tmpCellParent->right == tmp_cell)      tmpCellParent->right = newCell;
    // now, newCell has completely taken the place of tmp_cell.

    // make tmp_cell a copy of TN, except TN->element – we want to remove that!
    tmp_cell->right = TN->right;
    tmp_cell->parent = TN->parent;
    tmp_cell->left = TN->left;
    tmp_cell->height = TN->height;
    delete TN;                          // okay, since tmp_cell holds all data

    TN = tmp_cell;
    TN->left->parent = TN;
    TN->right->parent = TN;
     // now, finally, delete the just added new tmp_cell
    remove(TN->element, TN->right, TN);
    // check balance condition, perform rotation if necessary, update heights …
} // end 2 child case
```

…remove( ) 1-child & 0-child cases

```
        else {
                // X has 0 children or 1 child
                tmp_cell =  TN;
                if ((TN->left == NULL) && (TN->right == NULL)) {        // no children
                        TN = NULL;
                }
                else if ((TN->left == NULL) && (TN->right != NULL)) {   // no left child
                        TN = TN->right;
                        TN->parent = parentOfTN;
                }
                else if ((TN->left != NULL) && (TN->right == NULL)) {   // no right child
                        TN = TN->left;
                        TN->parent = parentOfTN;
                }
                delete tmp_cell;
                // reassign leftmost/rightmost in headerNode if need be …
                treeSize--;
        } // end 1-child & 0-child case
    } // remElem == TN->element
} // end remove( )
```

Summary

- Unbalanced Trees
- AVL Trees
- Balancing at each Node
- Rotations
- Operations on AVL Trees
  - Insert
  - Remove
- Implementation of AVL Trees