CS 2213
Advanced Programming
Ch 13 – Trees 2
Balanced trees

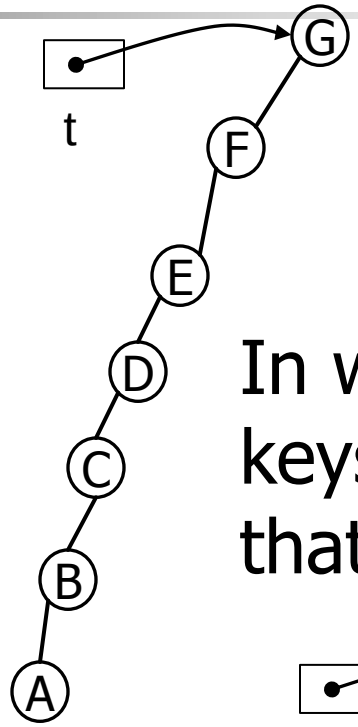
Turgay Korkmaz

Office: SB 4.01.13
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

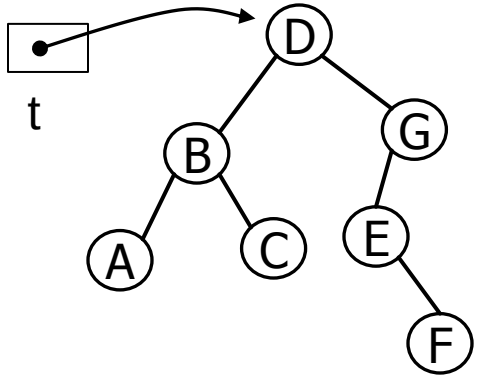
Thanks to Eric S. Roberts, the author of our textbook, for providing some slides/figures/programs.

Binary Search Tree: Balancing Problem

- nodeT *t=NULL;
- …
- InsertNode(&t, ‘G’);
- InsertNode(&t, ‘F’);
- InsertNode(&t, ‘E’);
- InsertNode(&t, ‘D’);
- InsertNode(&t, ‘C’);
- InsertNode(&t, ‘B’);
- InsertNode(&t, ‘A’);

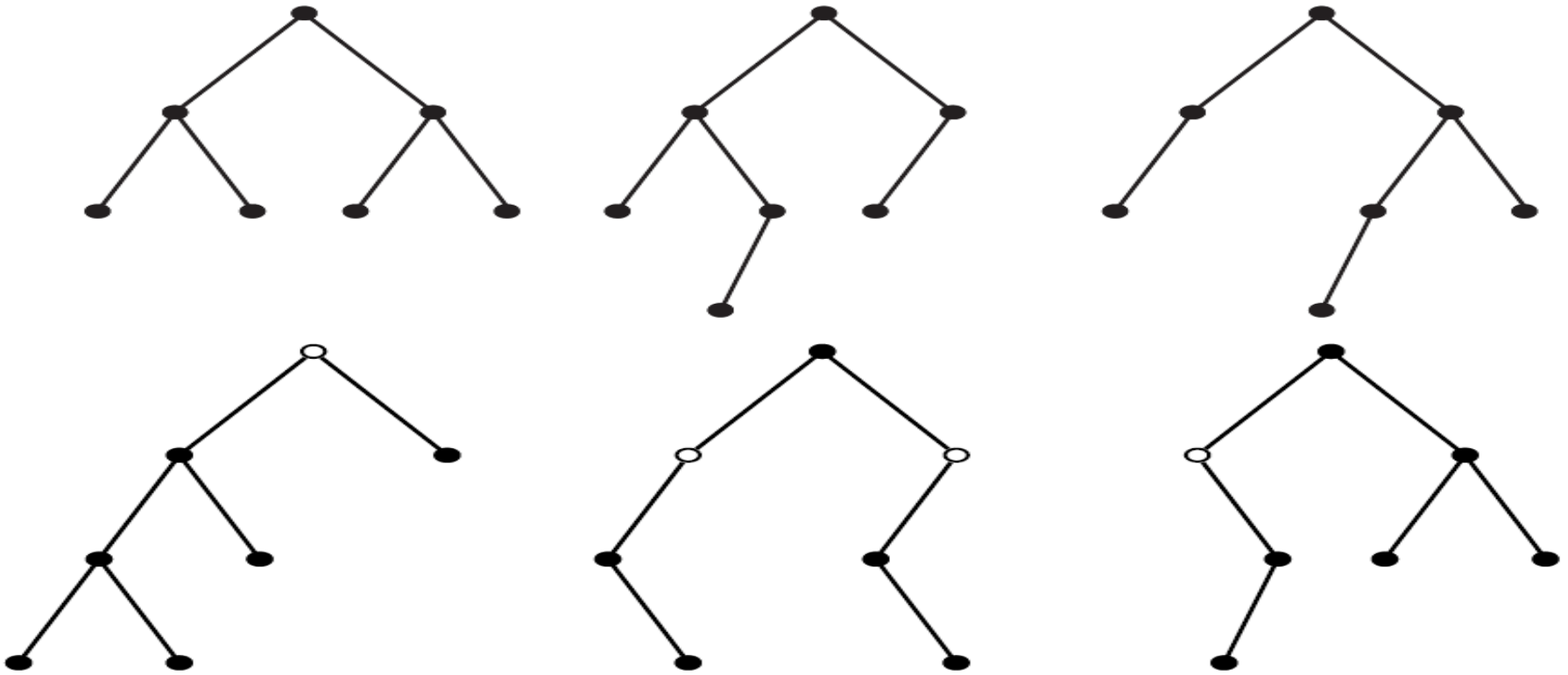


In which order the keys are inserted such that we got



Balanced Binary Search Tree

- A binary tree is balanced if, at each node, the height of the left and right subtrees differ by at most one.



the nodes at which the balanced-tree definition fails are shown as open circles

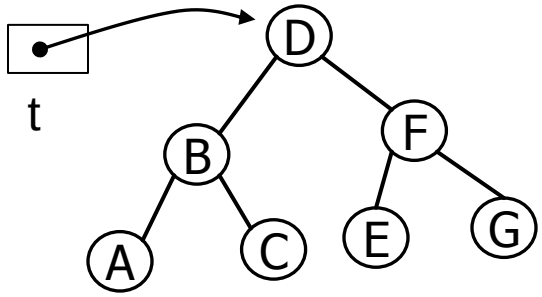
Exercise

- nodeT *t=NULL;
- …
- InsertNode(&t, ' ');
- InsertNode(&t, ' ');
- InsertNode(&t, ' ');
- InsertNode(&t, ' ');
- InsertNode(&t, ' ');
- InsertNode(&t, ' ');
- InsertNode(&t, ' ');

In which order should we insert the keys

A B C D E F G

so that we can get a fully balanced tree like this

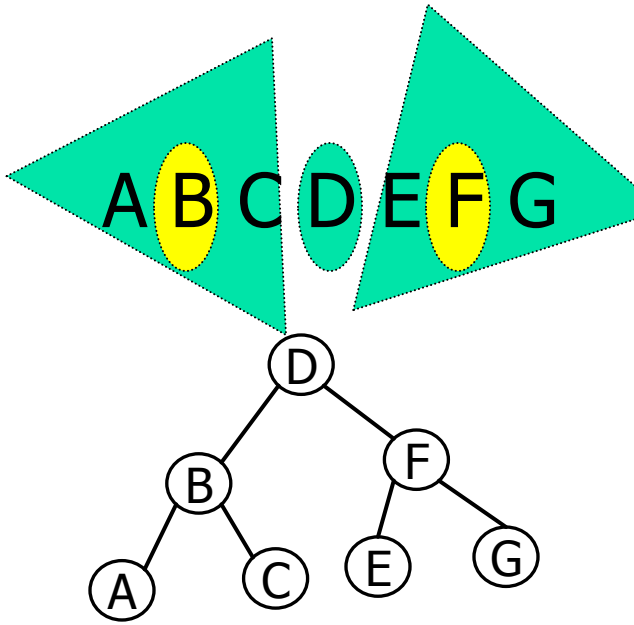


Exercise: Creating a balanced tree from sorted data

Suppose we have an array containing several values in a sorted manner. Can you generalize the idea in previous slide to create a balanced tree using the values in the sorted array?

Binary Search --- Binary Search based insert

```
void balance_ins(nodeT **t, char array[ ],
                 int first, int last)
{
  int mid;
  if (first <= last) {
     mid = (first + last)/2;
     InsertNode(t, array[mid]);
     balance_ins(t, array, first, mid-1);
     balance_ins(t, array, mid+1, last);
   }
}
```



```
int height(nodeT *t)
{
  if (t == NULL)
      return 0;
  else
     return (1 + max( height(t->left),
                      height(t->right)) );
}
```

Exercise

- Write a function to mark nodes as black or white as shown before in slide 3,
  - white means the balanced-tree definition fails
  - black means the balanced-tree definition holds
- Write a function to check if a given tree is balanced or not

```
typedef struct nodeT {
    char key;
    short color;  // 0: white  1: black
    struct nodeT *left, *right;
} nodeT, *treeT;
```

Home exercise

```
#define max(x,y) ((x)>(y) ? (x) : (y))

int height(nodeT *t)
{
   if (t == NULL)
         return 0;
   else
       return (1 + max( height(t->left),
                        height(t->right) ));
}
```

```
void MarkBW(nodeT *t) {
   int bf;
   if (t == NULL) return;
   bf = height(t->right) - height(t->left);
   if (bf>=-1 && bf <= 1)
      t->color = 1; /* black */
   else
      t->color = 0; /* white */
   MarkBW(t->right);
   MarkBW(t->left);
}
```

preorder

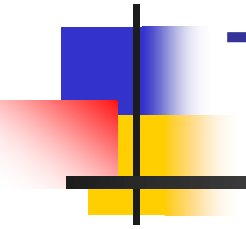
```
int impMarkBW(nodeT *t) {
   int bf, hr, hl;
   if (t == NULL)
         return 0;
   hr = impMarkBW(t->right);
   hl = impMarkBW(t->left);
   bf = hr – hl;
   if (bf>=-1 && bf <= 1)
      t->color = 1; /* black */
   else
      t->color = 0; /* white */
   return ( 1 + max(hr, hl ) );
}

void MarkBW(nodeT *t)
{
   (void) impMarkBW(t);
}
```

postorder

```
#define max(x,y) ((x)>(y) ? (x) : (y))

int height(nodeT *t)
{
   if (t == NULL)
         return 0;
   else
      return (1 + max(
                        height(t->left),
                        height(t->right)) );
}
```

```
int isBalanced(nodeT *t) {
   int bf;
   if (t == NULL) return 1; /* true */
   bf = height(t->right) - height(t->left);
   return (bf>=-1 && bf <= 1) &&
            isBalanced(t->left) &&
            isBalanced(t->right);
}
```

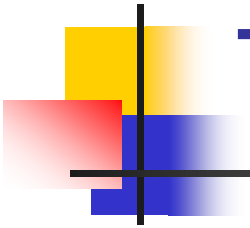
```
int impBalanced(nodeT *t, int *h) {
   int bf, hr, hl, br, bl;
   if (t == NULL) {
      *h=0;
      return 1; /* true */
   }
   br = impBalanced (t->right, &hr);
   bl = impBalanced (t->left, &hl);
  *h = 1+max(hr, hl);
   bf = hr – hl;
   if (br && bl && (bf>=-1 && bf <= 1) )
      return 1; /* true */
   else
      return 0; /* false */
}

int isBalances(nodeT *t) {
   int h;
   return impBalanced(t, &h);
}
```



Tree-balancing strategies

What if the input data values are given in a random order?

How can we keep the tree balanced?



Tree-balancing strategies

- The worst-case behavior of unbalanced trees
  - **FindNode** and **InsertNode** become O(N)
- We can keep trees balanced as we build them
  - Extend the implementation of **InsertNode**
    - it keeps track of whether the tree is balanced while inserting new nodes.
    - If the tree ever becomes out of balance, **InsertNode must rearrange the nodes in** the tree so that the balance is restored without disturbing the ordering relationships.
  - Assuming that it is possible to rearrange a tree in time proportional to its height, both **FindNode** and **InsertNode** can be implemented in O(log N) time. Why?
  - There are several algorithms, we will study AVL

AVL - Example

- Create a binary search tree in which the nodes contain the symbols for the chemical elements.
  - **H** (Hydrogen)
  - **He** (Helium)
  - **Li** (Lithium)
  - **Be** (Beryllium)
  - **B** (Boron)
  - **C** (Carbon**)**

```
typedef struct nodeT {
    string key;
    struct nodeT *left, *right;
} nodeT, *treeT;
```

- How will the tree look like if we insert them in that order? Can we keep the tree balanced?

AVL– basic idea

Published in 1962 by the Russian mathematicians Georgii Adel'son-Vel'skii and Evgenii Landis and has since been known by the initials AVL.

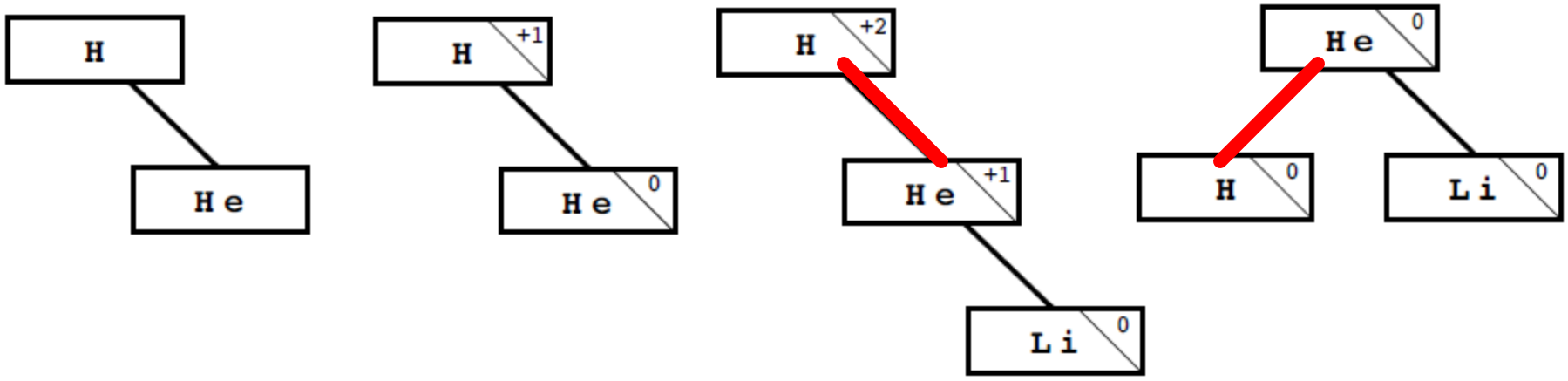
- Associate an integer with each node, called **balance factor** = height(right)-height(left)
  - Each node of an AVL tree has a **bf** 0, -1 or 1

    -1 : left-heavy (the height of the left sub-tree is 1 greater than the right sub-tree),

    0 : balanced (both sub-trees are the same height) or

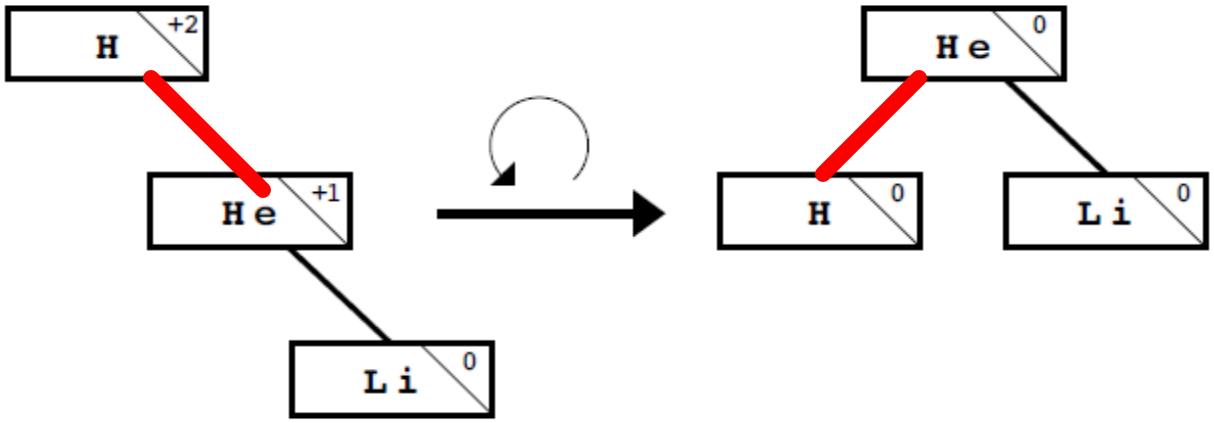
    +1: right-heavy (the height of the right sub-tree is 1 greater than the left sub-tree).
  - If **bf** for any node is 2 or -2 , the tree is not balanced.
- To fix the imbalance, restructure the tree

AVL - operation



How many different cases do we have to consider?

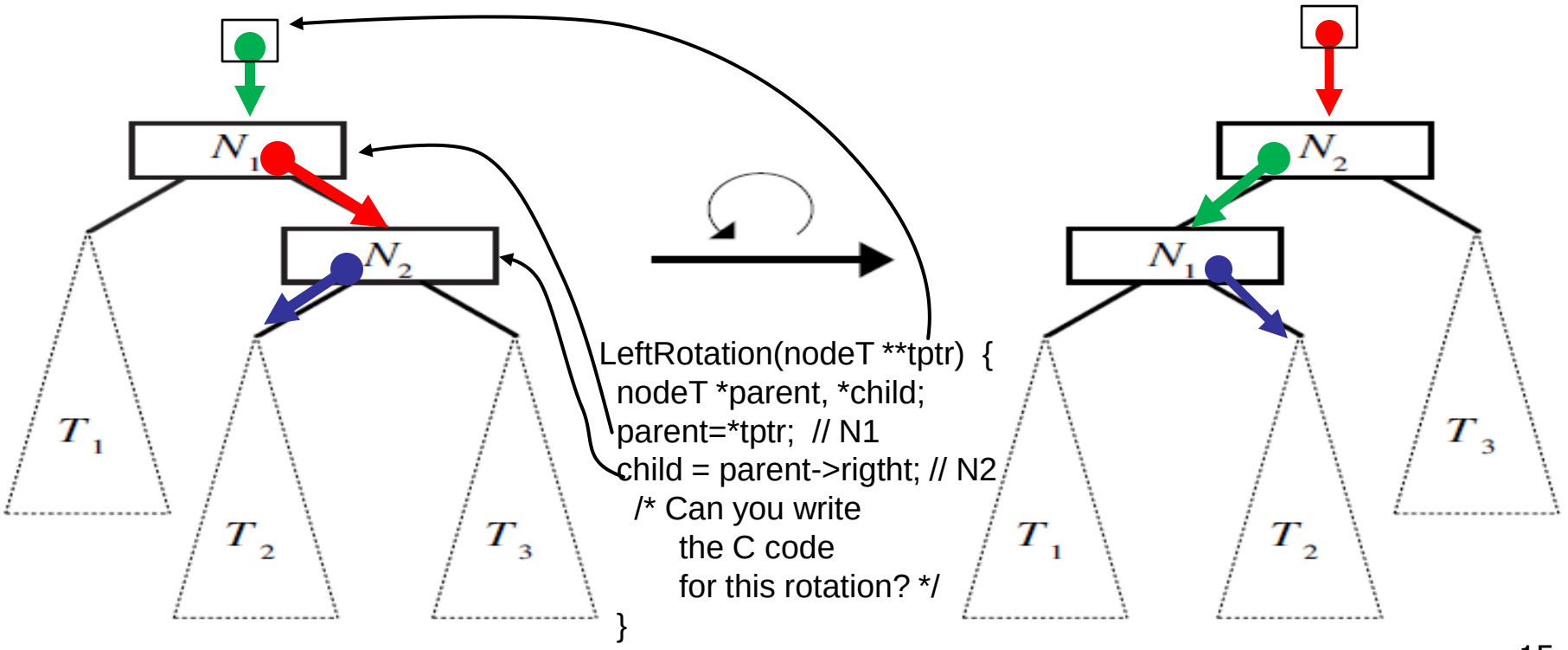
AVL -- Single Rotation



H-He axis
Left rotation

AVL -- Single Rotation Generalization

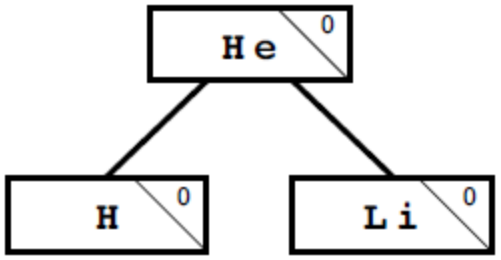
In general, you can always perform this type of rotation operation on any two nodes in a binary search tree without invalidating the relative ordering of the nodes, even if the nodes have subtrees descending from them. In a tree with additional nodes, the basic operation looks like this:

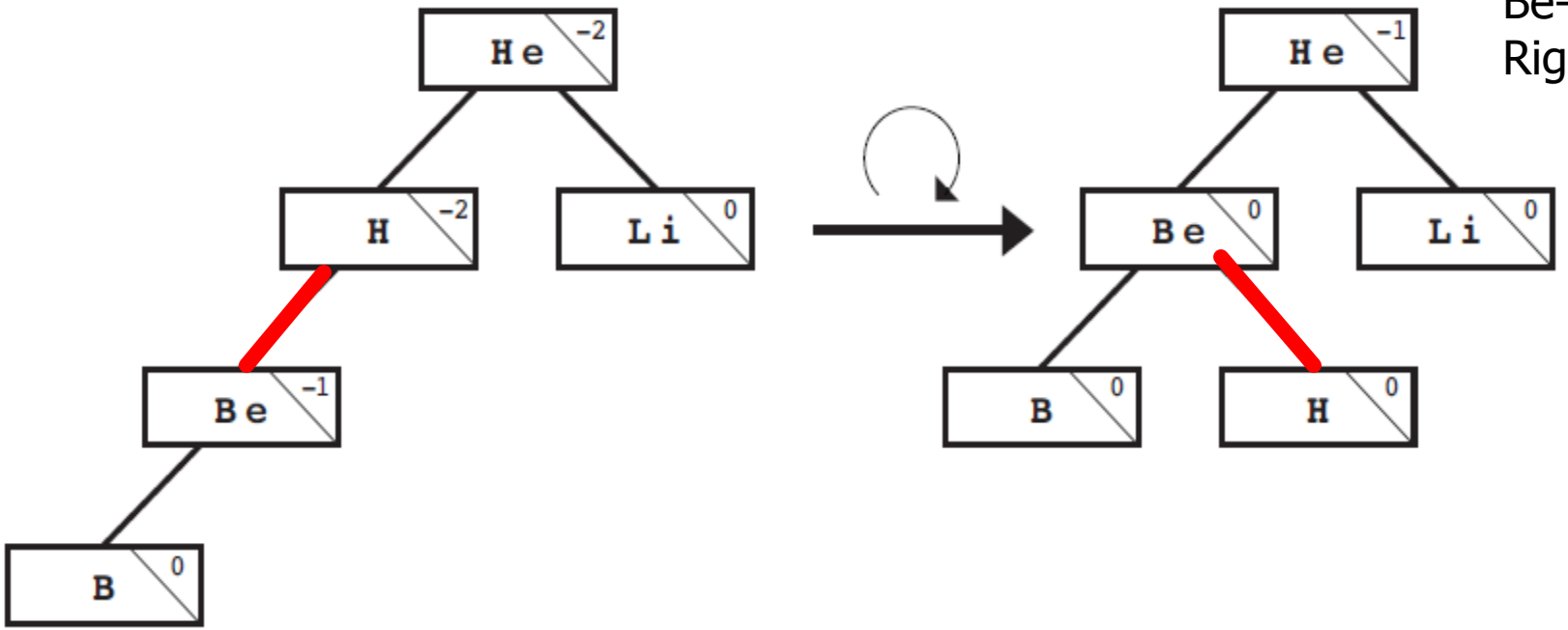



```
LeftRotation(nodeT **tptr)  {
  nodeT *parent, *child;
  parent=*tptr;  // N1
  child = parent->rigtht; // N2
    /* Can you write
         the C code
         for this rotation? */
}
```

AVL -- Single Rotation

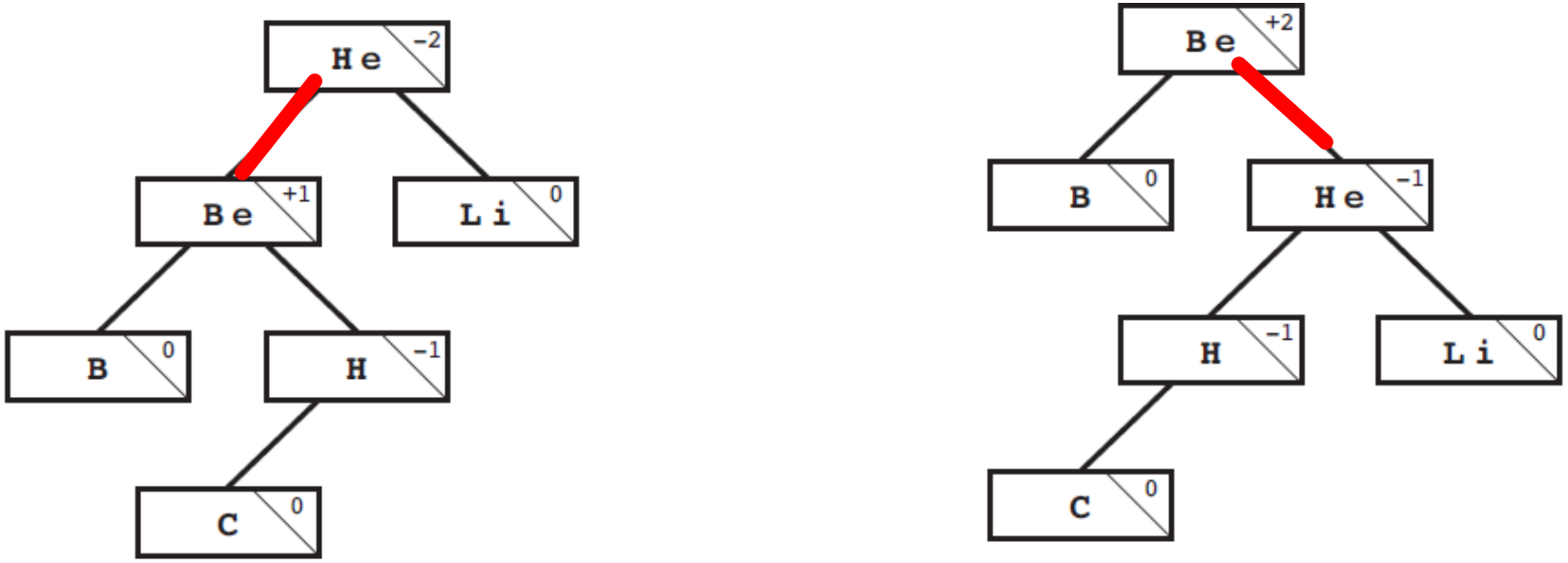
insert **Be** and **B**





Be-H axis
Right rotation

AVL -- Single Rotation
is not always sufficient: insert C

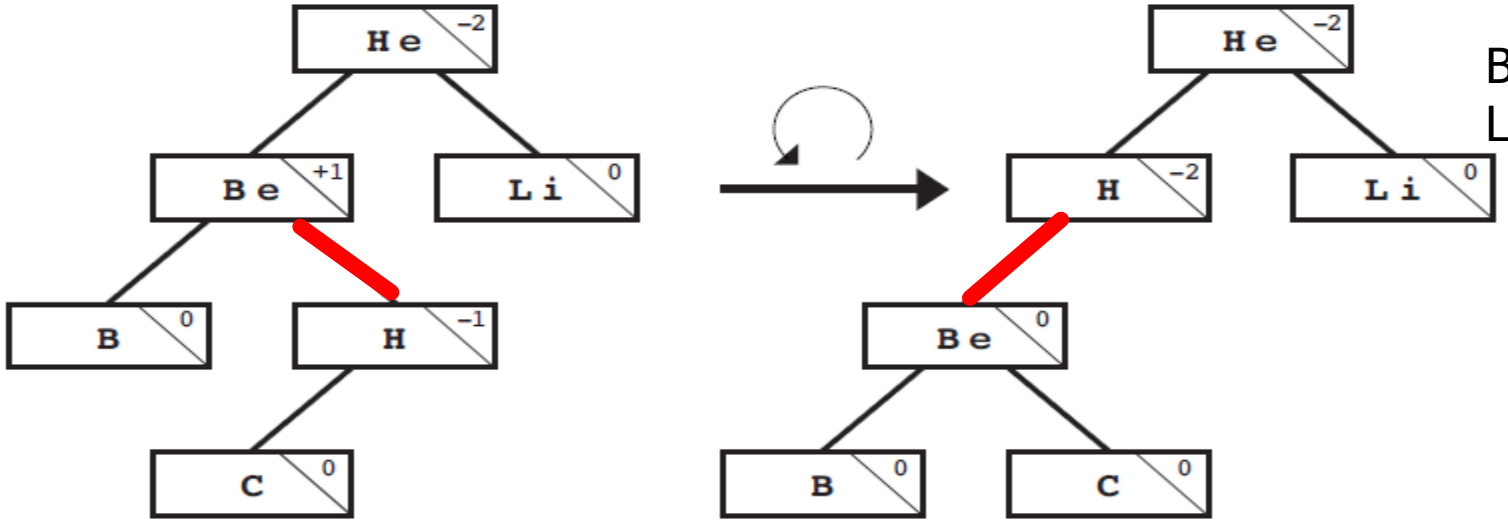


When the nodes involved in the rotation have balance factors with **opposite signs**, a single rotation is not enough. To fix the problem, we need to make two rotations.

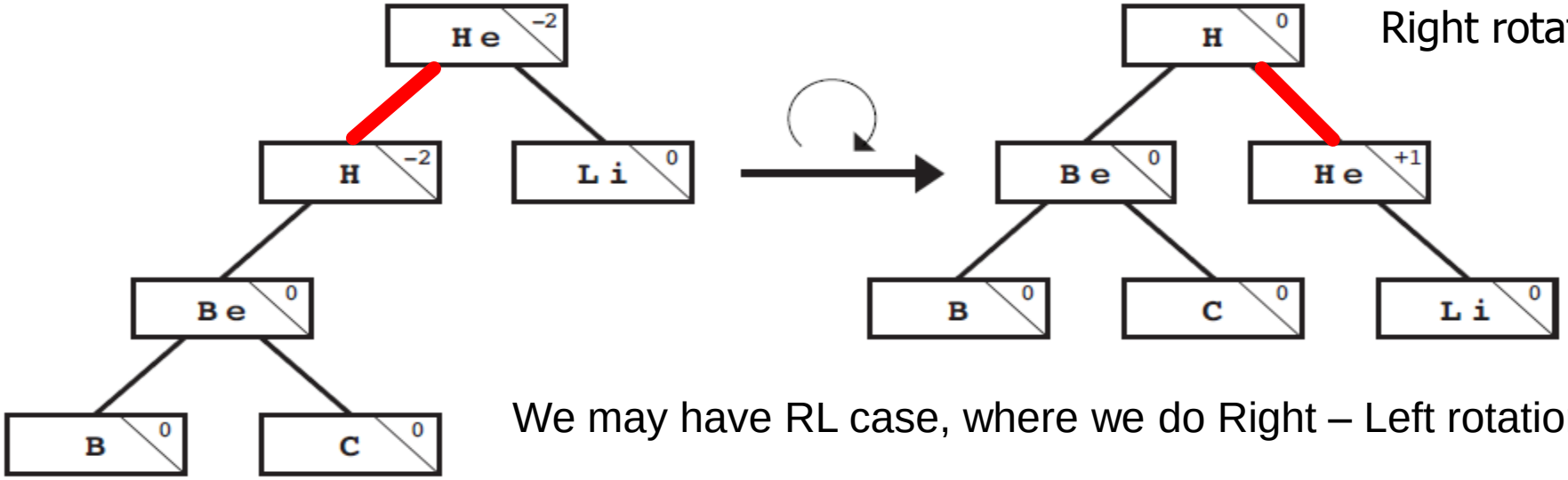
1. Before rotating the out-of balance node, we rotate its child in the opposite direction. Rotating the child gives the **balance factors in the parent and child the same sign,** which means that the following rotation will succeed.
2. We then do the required rotation.

This pair of operations is called a **double rotation**

AVL – Double Rotation (LR)



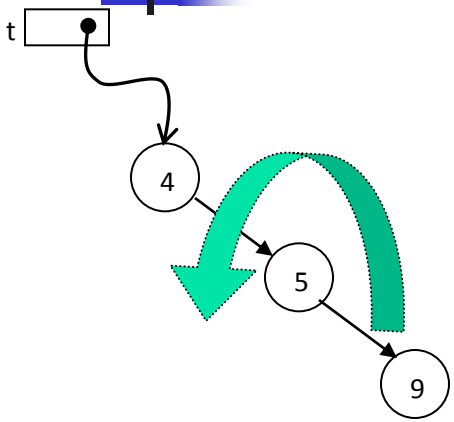
Be-H axis
Left rotation

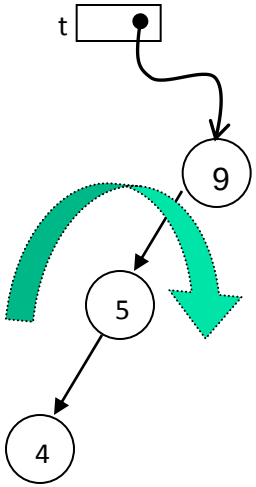


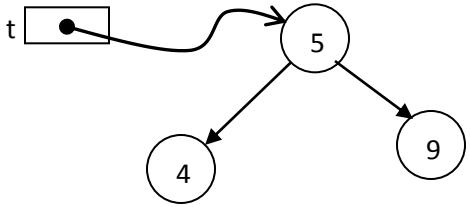
H-He axis
Right rotation

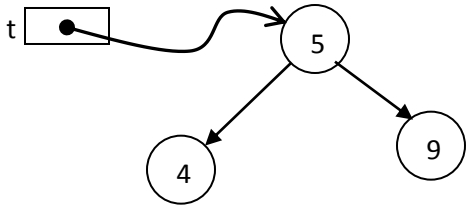
We may have RL case, where we do Right – Left rotations

L and R rotations

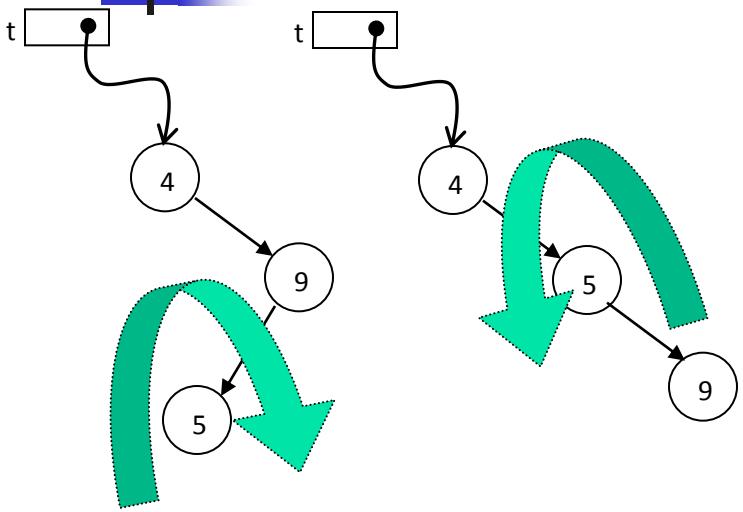


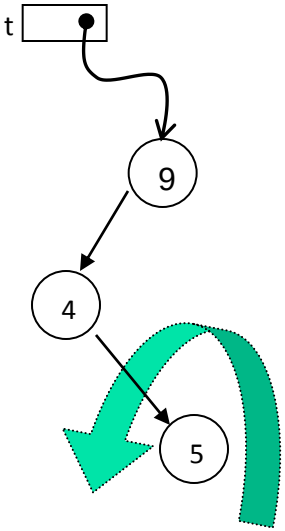


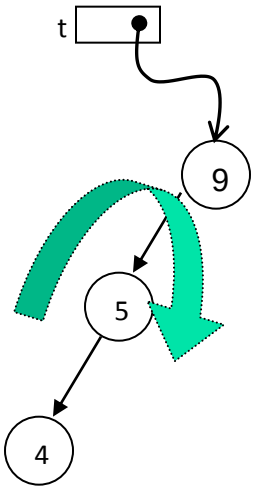


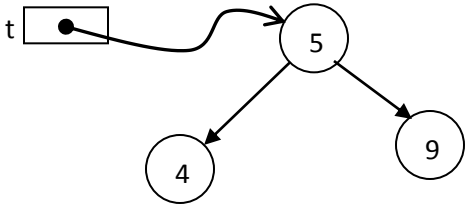


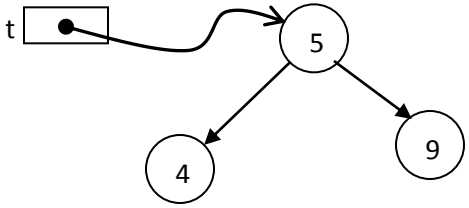
RL and LR rotations





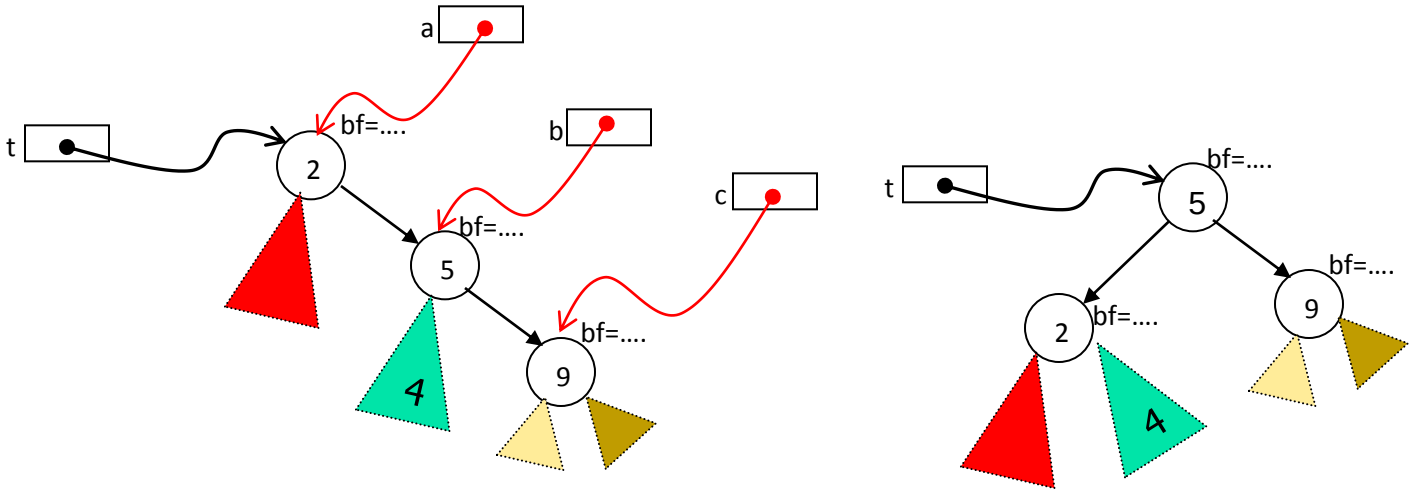






Exercise: Code for left rotation

Suppose you have pointers t and then a, b, c for nodes 4, 5, 9.
Write the necessary statements to achieve left rotation.

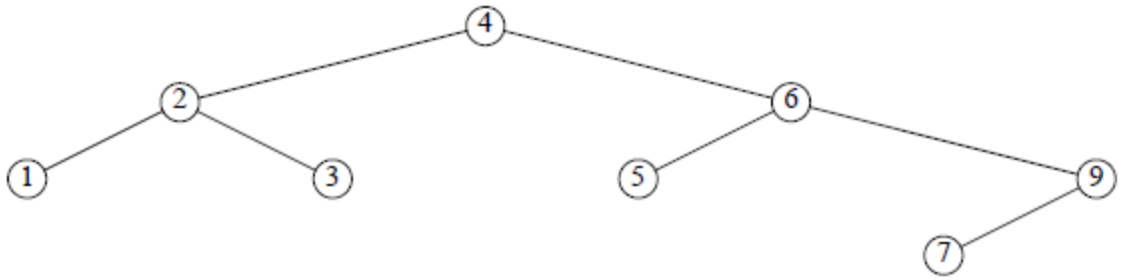


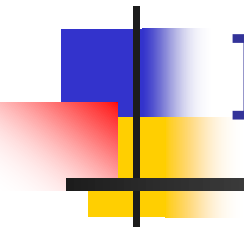

```
parent->right = child->left;
child->left = parent;
(*tptr) = child;
```

```
a->right = b->left;
b->left = a;
t = b;
```

Exercise: AVL tree

Show the results of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an AVL tree, which is currently empty.
You don't need to re-draw the tree after each insertion. But after a rotation, you must re-draw the tree. Also Don't forget to include balance factor (**bf**) for each node and show how it changes as you insert new nodes.





Insert

Implementing **AVL** algorithm InsertNode → InsertAVL

```
typedef struct nodeT {
    char key;
    struct nodeT *left, *right;
} nodeT, *treeT;
```

```
typedef struct nodeT {
    string key;
    struct nodeT *left, *right;
    int bf;
} nodeT, *treeT;
```

```
void InsertNode(treeT  *tptr, string key)
{
  (void) InsertAVL(tprt, key);
}
```

```
void InsertNode(nodeT **tptr, char key)
{
    nodeT  *tmp;
    tmp=*tptr;
    if (tmp == NULL) {
        tmp=New(nodeT *);
        tmp->key = key;
        tmp->left=tmp->right=NULL;
        *tptr=tmp;
        return;
    }
    if (key < tmp->key) {
            InsertNode(&tmp->left, key);
    } else {
            InsertNode(&tmp->right, key);
    }
}
```

int InsertAVL(treeT *tprt, string key) vs. void InsertNode(treeT *tprt, string key)

- **InsertAVL** seems to have the same prototype of **InsertNode**, which is implemented as a wrapper to **InsertAVL** (why).
- The parameters are indeed the same.
- But, **InsertAVL** returns an integer value (why)
  - that represents *the change in the height of the tree* after inserting the node.
- This return value, which will always be 0 or 1, makes it easy to **fix the structure of the tree** as the code makes its way back through the level of recursive calls.

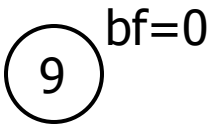
MarkBW vs. impMarkBW

```
static int InsertAVL(treeT *tptr, string key)
{
    treeT t;
    int sign, delta;

    t = *tptr;
    if (t == NULL) {
        t = New(treeT);
        t->key = CopyString(key);
        t->bf = 0;
        t->left = t->right = NULL;
        *tptr = t;
        return (+1);
    }
    sign = StringCompare(key, t->key);
    if (sign == 0) return (0);

}
```

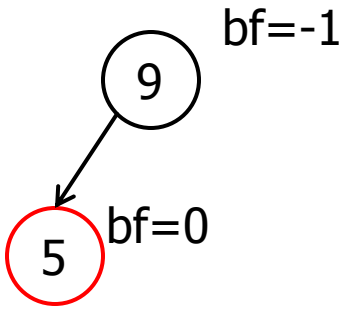
```
if (sign < 0) {
        delta = InsertAVL(&t->left, key);
        if (delta == 0) return (0);
        switch (t->bf) {
          case +1: t->bf =  0; return (0);
          case  0: t->bf = -1; return (+1);
          case -1: FixLeftImbalance(tptr);
                   return (0);
        }
    } else {
        delta = InsertAVL(&t->right, key);
        if (delta == 0) return (0);
        switch (t->bf) {
          case -1: t->bf =  0; return (0);
          case  0: t->bf = +1; return (+1);
          case +1: FixRightImbalance(tptr);
                   return (0);
        }
    }
```

Insert left left

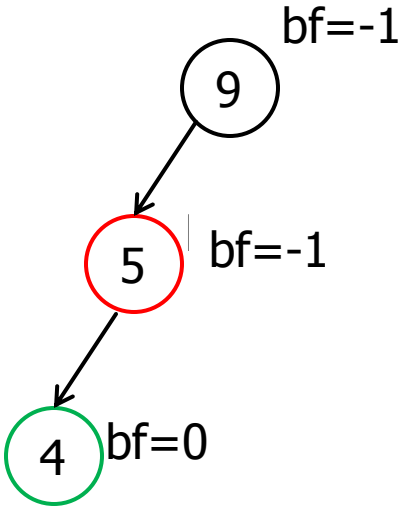
Insert 9



Insert 5

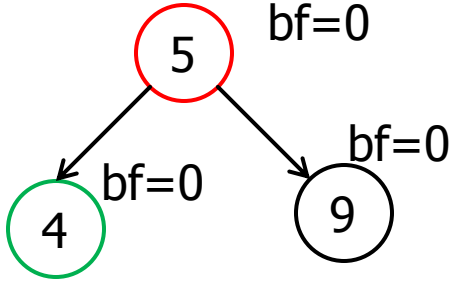


Insert 4

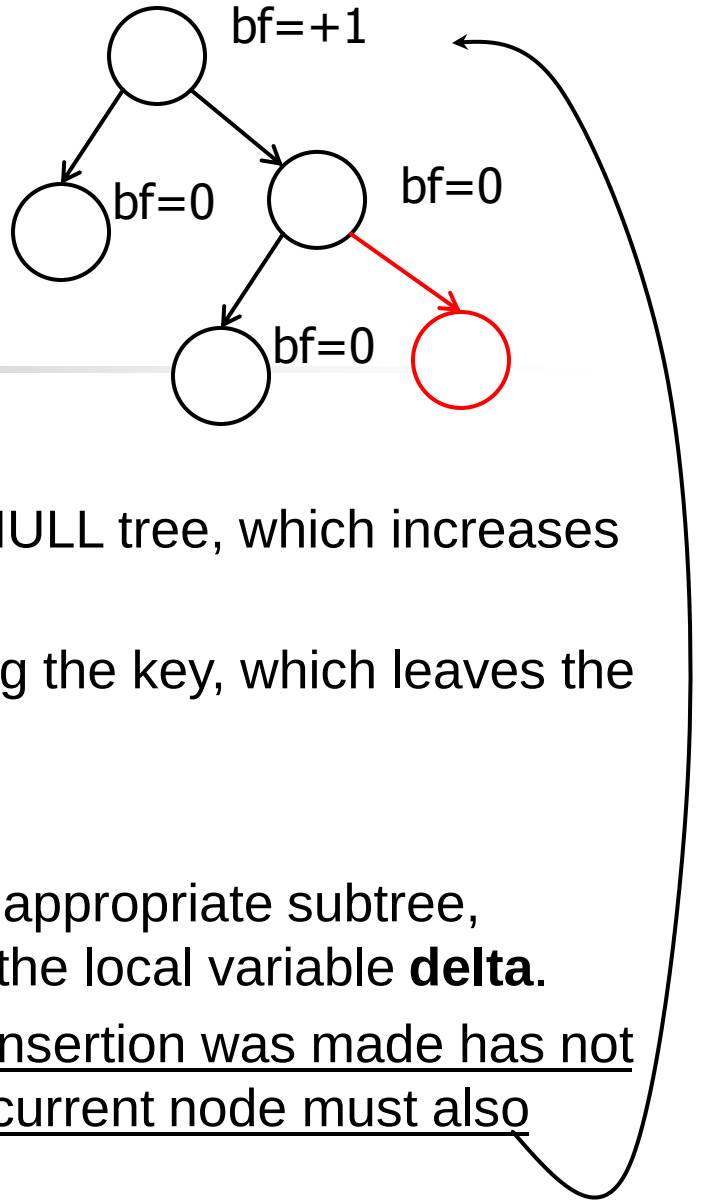


Initial case
return 1

```
delta = InsertAVL(&t->left, key);
if (delta == 0) return (0);
switch (t->bf) {
    case +1: t->bf =  0; return (0);
    case  0: t->bf = -1; return (+1);
    case -1: FixLeftImbalance(tptr);
                  return (0);
}
```

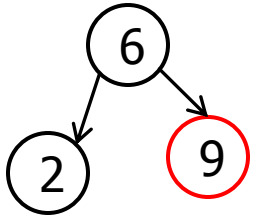


Recursive calls to InsertAVL(…)

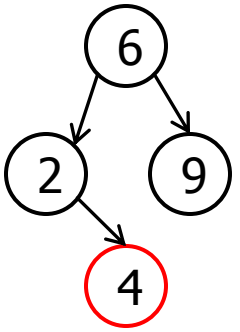


- The simple cases are
  1. Adding a node in place of a previously NULL tree, which increases the height by one
  2. Encountering an existing node containing the key, which leaves the height unchanged
- In the recursive cases,
  1. The code first adds the new node to the appropriate subtree, keeping track of the change in height in the local variable **delta**.
  2. If the height of the subtree to which the insertion was made has not changed, then the balance factor in the current node must also remain the same.
  3. **If, however, the subtree increased in height, there are three possibilities:** (see next slide)

1. *That subtree was previously shorter than the other subtree in this node.* In this case, inserting the new node actually makes the tree more balanced than it was previously. The balance factor of the current node becomes 0, and the height of the subtree rooted there remains the same as before.

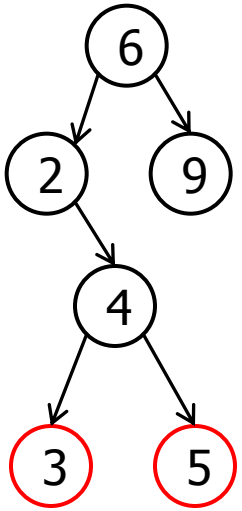


2. The two subtrees in the current node were previously the same size. In this case, increasing the size of one of the subtrees makes the current node slightly out of balance, but not to the point that any corrective action is required. The balance factor becomes –1 or +1, as appropriate, and the function returns 1 to show that the height of the subtree rooted at this node has increased.

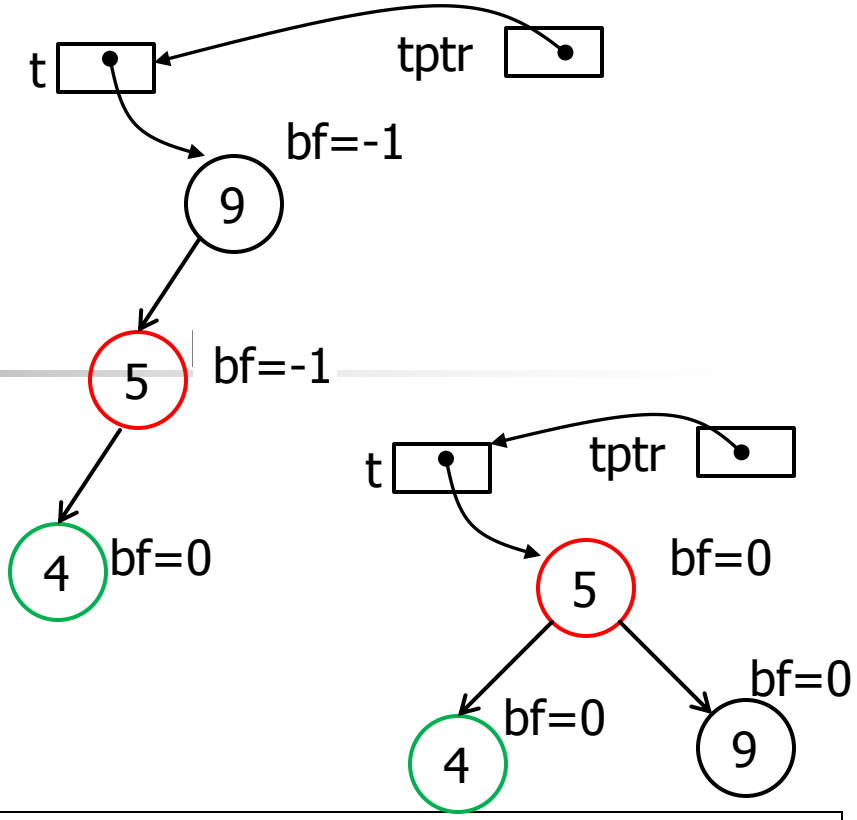


3. The subtree that grew taller was already taller than the other subtree. When this situation occurs, the tree has become **seriously out of balance**, because one subtree is now two nodes higher than the other. At this point, the code must execute the appropriate rotation operations to correct the imbalance.

- If the balance factors in the current node and the root of the subtree that expanded have **the same sign**, a single rotation is sufficient.
- **If not**, the code must perform a double rotation. After performing the rotations, the code must correct the balance factors in the nodes whose positions have changed.

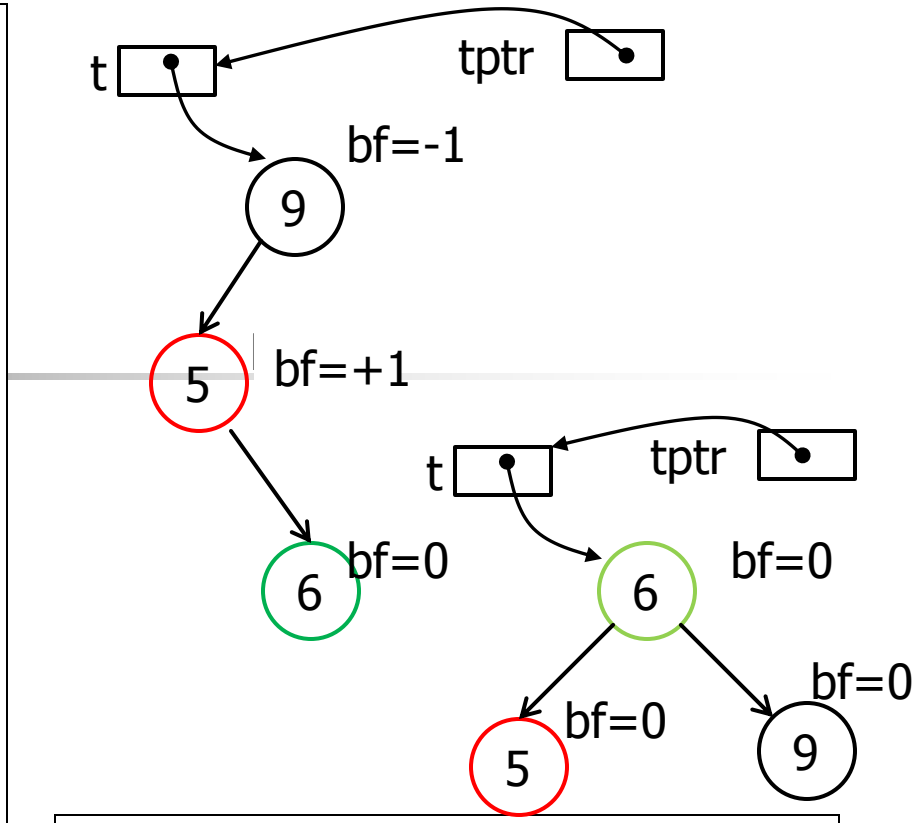


```
static void FixLeftImbalance(treeT *tptr)
{
    treeT t, parent, child, *cptr;
    int oldBF;

    parent = *tptr;
    cptr = &parent->left;
    child = *cptr;
    if (child->bf != parent->bf) { // check signs
        oldBF = child->right->bf;
        RotateLeft(cptr);
        RotateRight(tptr);
        t = *tptr;
        t->bf = 0;
        switch (oldBF) {
          case -1: t->left->bf = 0;
                   t->right->bf = +1; break;
          case  0: t->left->bf = t->right->bf = 0;
                   break;
          case +1: t->left->bf = -1;
                   t->right->bf = 0; break;
        }
    } else {
        RotateRight(tptr);
        t = *tptr;
        t->right->bf = t->bf = 0;
    }
}
```




```
delta = InsertAVL(&t->left, key);
if (delta == 0) return (0);
switch (t->bf) {
    case +1: t->bf =  0; return (0);
    case  0: t->bf = -1; return (+1);
    case -1: FixLeftImbalance(tptr);
             return (0);
}
```

```
static void FixLeftImbalance(treeT *tptr)
{
    treeT t, parent, child, *cptr;
    int oldBF;

    parent = *tptr;
    cptr = &parent->left;
    child = *cptr;
    if (child->bf != parent->bf) {
        oldBF = child->right->bf;
        RotateLeft(cptr);
        RotateRight(tptr);
        t = *tptr;
        t->bf = 0;
        switch (oldBF) {
          case -1: t->left->bf = 0;
                   t->right->bf = +1; break;
          case  0: t->left->bf = t->right->bf = 0;
                   break;
          case +1: t->left->bf = -1;
                   t->right->bf = 0; break;
        }
    } else {
        RotateRight(tptr);
        t = *tptr;
        t->right->bf = t->bf = 0;
    }
}
```




```
delta = InsertAVL(&t->left, key);
if (delta == 0) return (0);
switch (t->bf) {
    case +1: t->bf =  0; return (0);
    case  0: t->bf = -1; return (+1);
    case -1: FixLeftImbalance(tptr);
                 return (0);
}
```

Exercise

FixRightImbalance()

```
static void FixRightImbalance(treeT *tptr)
{
    treeT t, parent, child, *cptr;
    int oldBF;

    parent = *tptr;
    cptr = &parent->right;
    child = *cptr;
    if (child->bf != parent->bf) {
        oldBF = child->right->bf;
        RotateRight(cptr);
        RotateLeft(tptr);
        t = *tptr;
        t->bf = 0;
        switch (oldBF) {
          case -1: t->left->bf = 0;
                   t->right->bf = +1; break;
          case  0: t->left->bf = t->right->bf = 0;
                   break;
          case +1: t->left->bf = -1;
                     t->right->bf = 0; break;
        }
    } else {
        RotateLeft(tptr);
        t = *tptr;
        t->left->bf = t->bf = 0;
    }
}
```

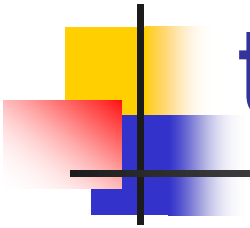
AVL Rotations

```
void RotateLeft(treeT *tptr)
{
    treeT parent, child;

    parent = *tptr;
    child = parent->right;
    parent->right = child->left;
    child->left = parent;
    (*tptr) = child;
}
```

```
void RotateRight(treeT *tptr)
{
    treeT parent, child;

    parent = *tptr;
    child = parent->left;
    parent->left = child->right;
    child->right = parent;
    (*tptr) = child;
}
```

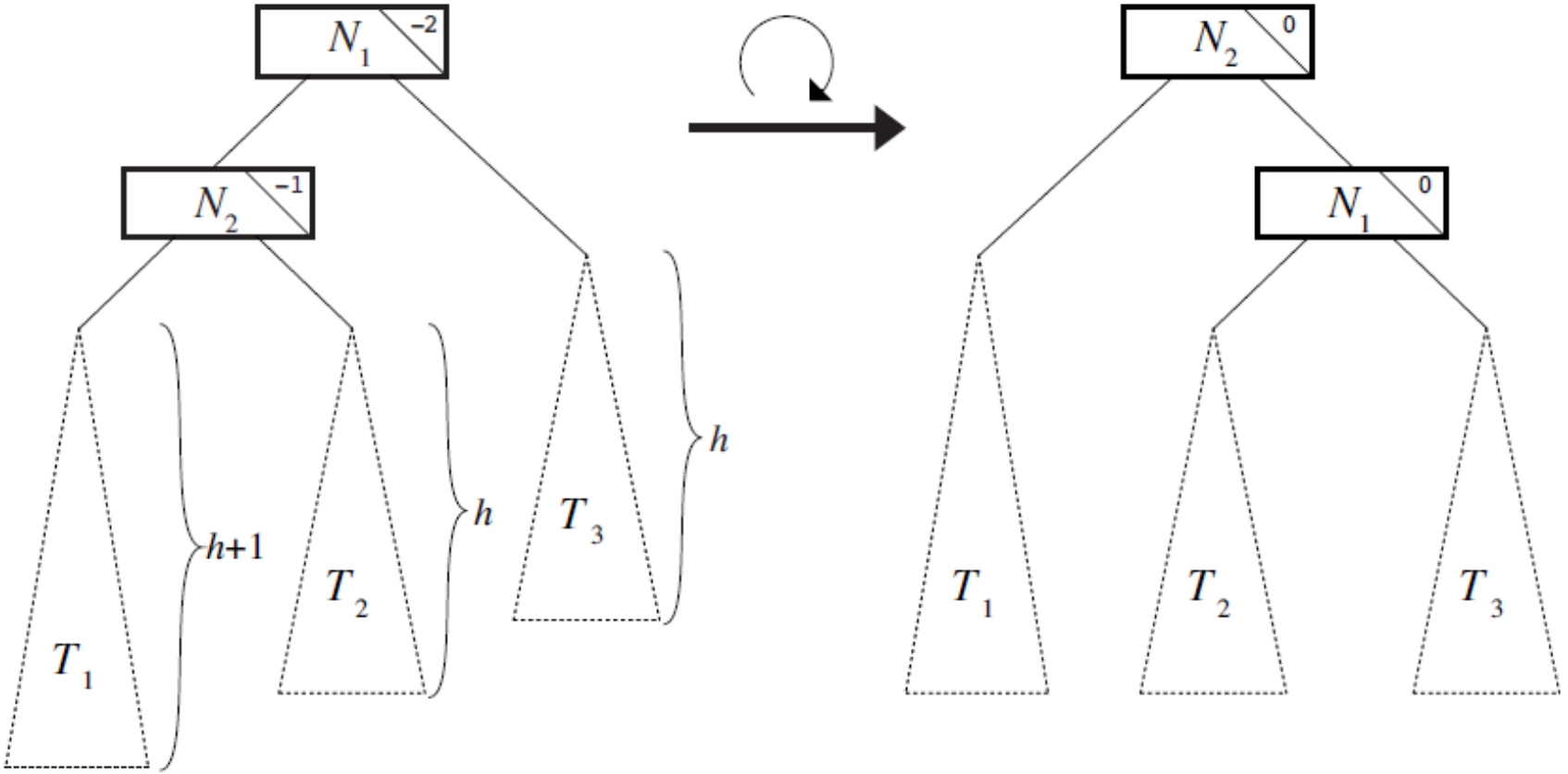
```
static void DisplayTree(treeT t)
{
   if (t != NULL) {
      DisplayTree(t->left);
      printf("%s\n", t->key);
      DisplayTree(t->right);
   }
}

static void DisplayStructure(treeT t)
{
   RecDisplayStructure(t, 0, NULL);
}

static void RecDisplayStructure(treeT t, int depth, string label)
{
   if (t == NULL) return;
   printf("%*s", 3 * depth, "");
   if (label != NULL) printf("%s: ", label);
   printf("%s (%s%d)\n", t->key, (t->bf > 0) ? "+" : "", t->bf);
   RecDisplayStructure(t->left, depth + 1, "L");
   RecDisplayStructure(t->right, depth + 1, "R");
}
```

Properties of AVL tree-balancing algorithm

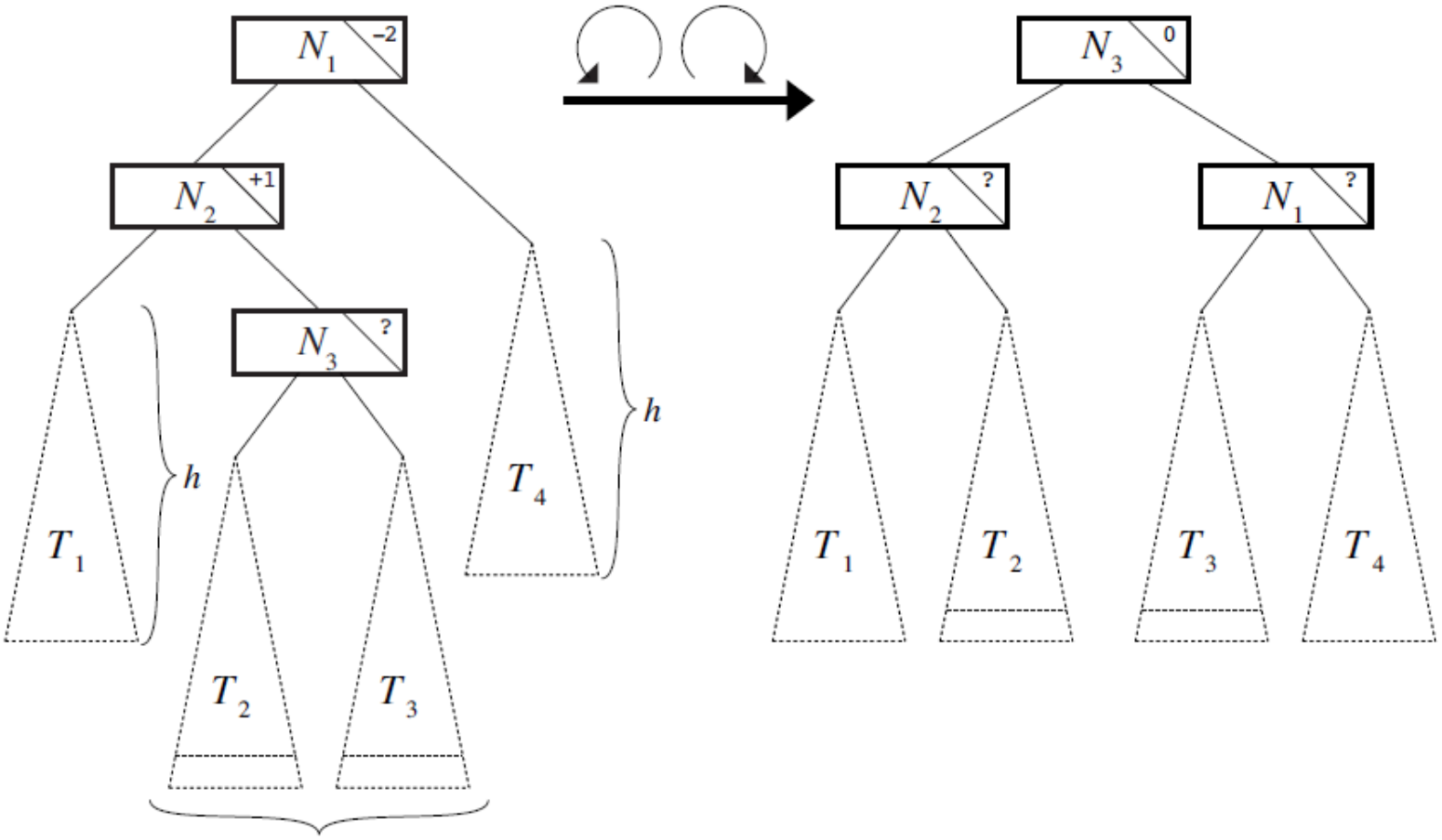


- If you insert a new node into an AVL tree, you can always restore its balance by performing at most one operation, which is either a single or a double rotation.
- After you complete the rotation operation, the height of the subtree at the axis of rotation is always the same as it was before inserting the new node.
- This property ensures that none of the balance factors change at any higher levels of the tree.

The effect of **single** rotation operation on balance factors

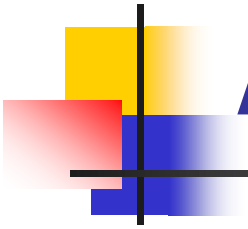


The effect of **double** rotation operations on balance factors



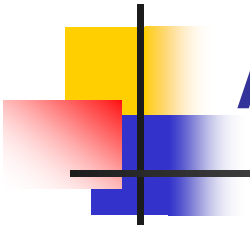
Unless both T_2 and T_3 are empty ($h = 0$), one will have height h and the other height h–1

The new balance factors in the N_1 and N_2 nodes depend on the relative heights of the subtrees T_2 and T_3



AVL Tree: Running Times

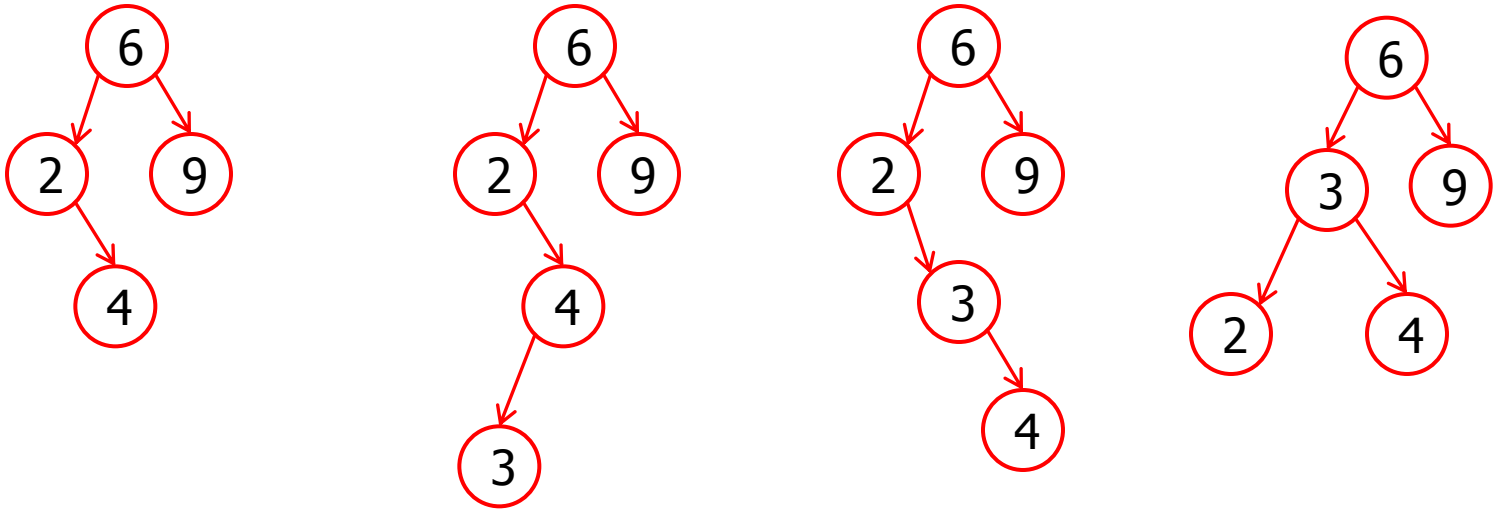
- Find takes O(log *n*), Why?
  - height of the tree is O(log *n*).
- Insert takes O(log *n*), Why?
  - We find the node (O(log *n*) time), and then we may have to visit every node on the path back to the root, performing up to 2 single rotations (O(1) time each) to fix the tree.
- Remove: O(log *n*). Why?

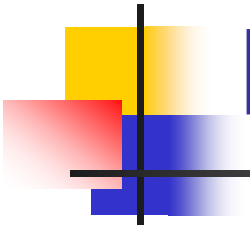


AVL Tree: Review

- Remember the definitions.
  - What is the purpose of AVL tree?
  - What is the balance factor (**bf**)?
- Insertion in AVL Trees (4 balancing cases):
  - L, R, LR and RL (single and double rotation).
  - Be able to give example
- Complexity of operations on AVL trees.

Which key insertion will cause single and double rotations





Exercise: AVL Delete

- Rebalancing after removal is quite similar to that for insertion.
  - Removing a node either may have no effect on the height of a tree or may shorten it by one.
  - If a tree gets shorter, the balance factor in its parent node changes.
  - If the parent node becomes out of balance, it is possible to rebalance the tree at that point by performing either a single or a double rotation.