Balanced Trees

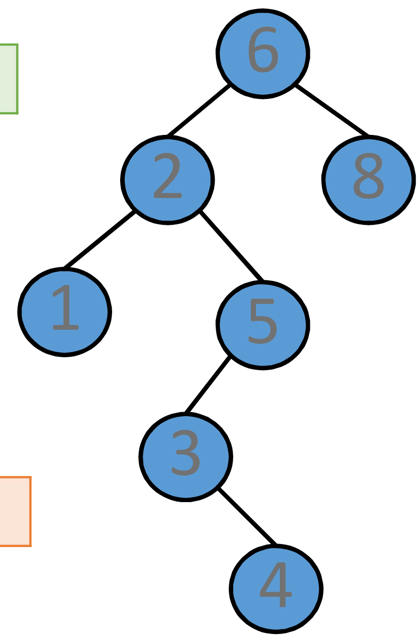
Tree Insertion Sorting

- Construct a binary search tree for all keys, an inorder traversal will visit the elements in sorted order.

6	2	8	5	1	3	4

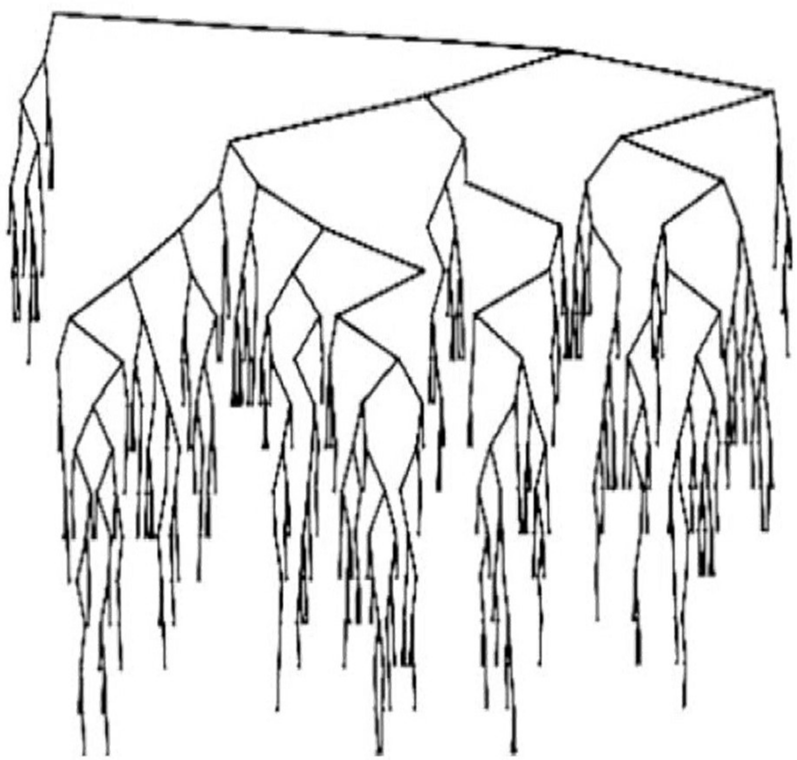
inorder traversal ⇩

1	2	3	4	5	6	8



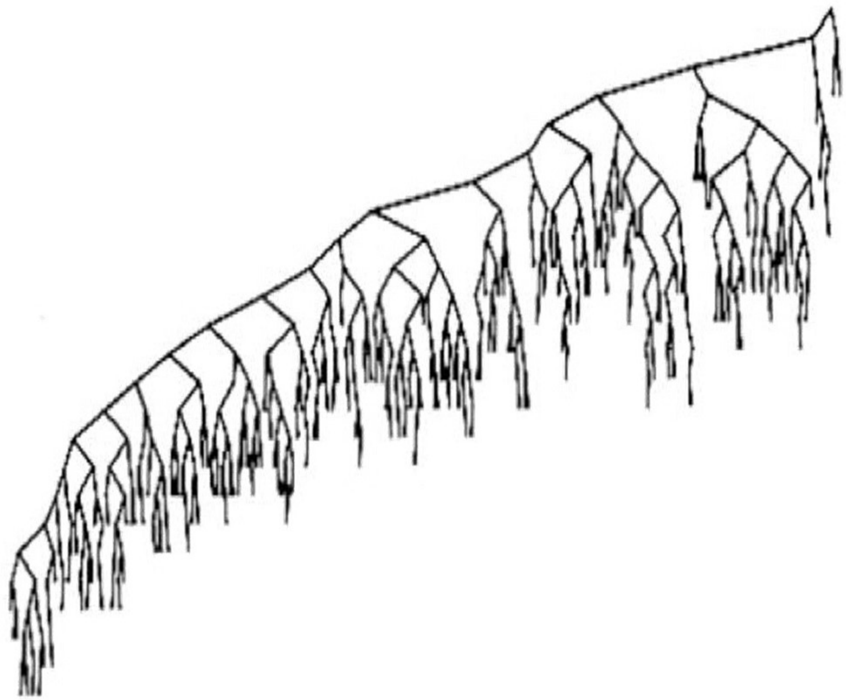
Normal Binary Search Tree

A randomly generated 500-node tree



An Unbalanced BST

- After many *insert* and *delete*, we could end up with an unbalanced BST.



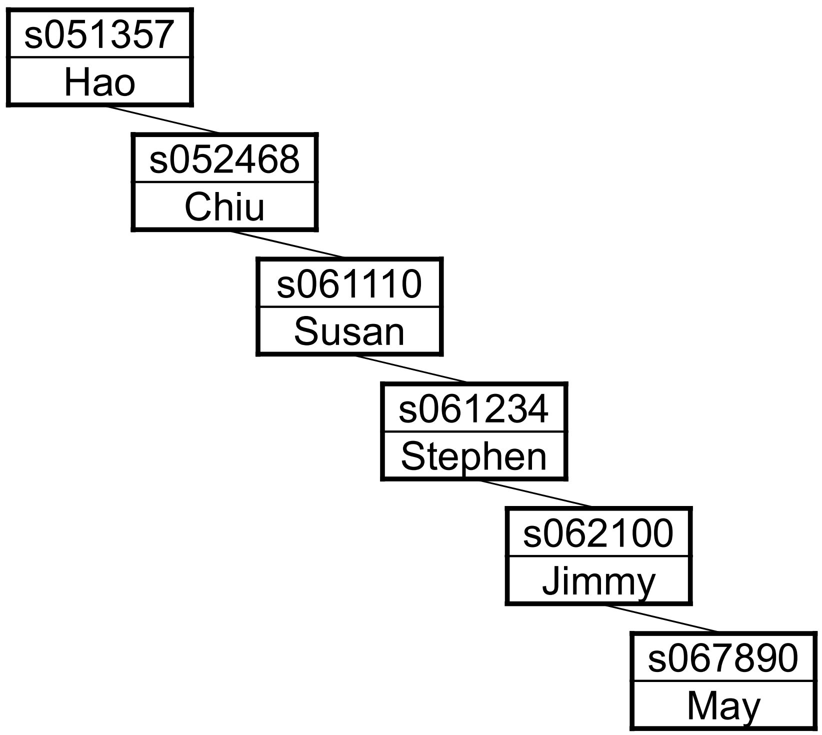
Can We Do Better?

- Balance is important to ensure that the tree does not degenerate into an unbalanced tree.

- Difficult as balancing a tree often requires the structure of the tree to be changed (take longer on average for updates).

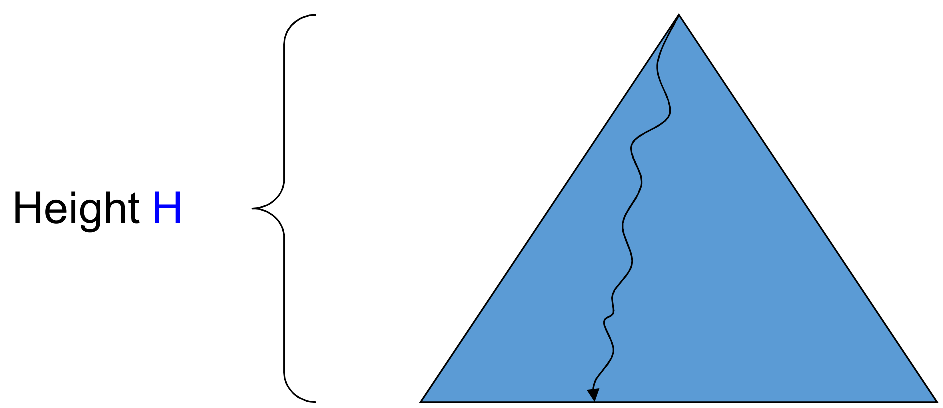
Skewed Binary Search Tree

- Recall that this is also a BST, although it is a *skewed* one.



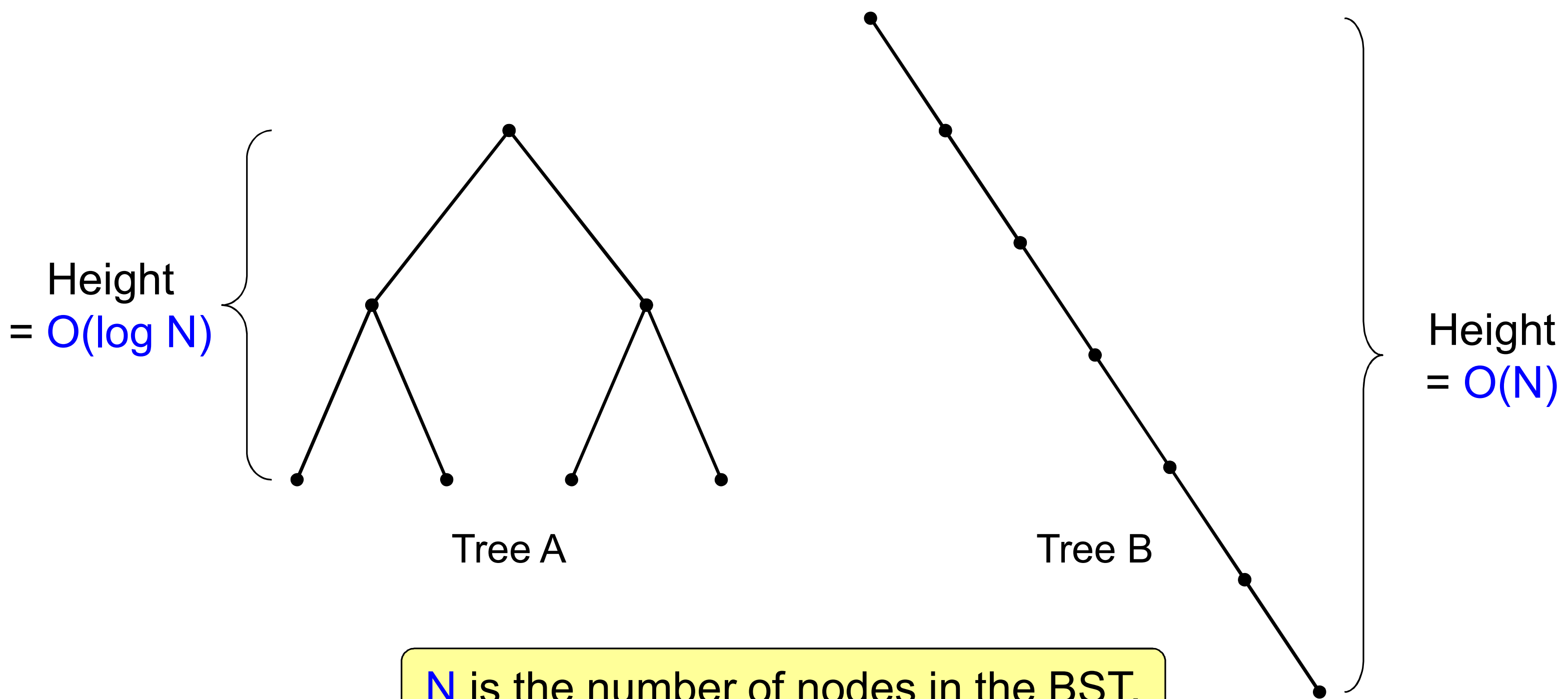
BST Operations

- In fact, the functions **`FindNode`**, **`InsertNode`**, and **`DeleteNode`** all run in O(H) time, where H is the height of the BST.



O(log N) vs O(N)

- Therefore, we like tree A more than tree B, although both trees have 7 nodes.

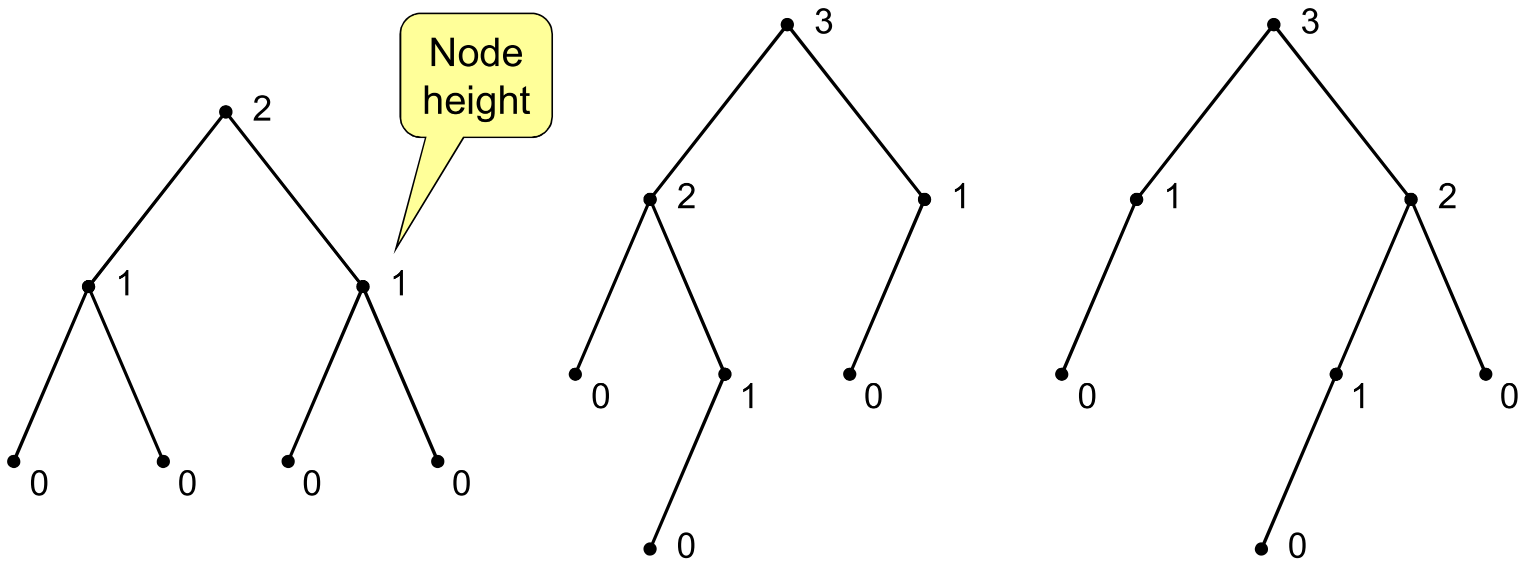


N is the number of nodes in the BST.

Balanced Binary Trees

- The ***height*** of n_i is the length of the longest path from n_i to a leaf.

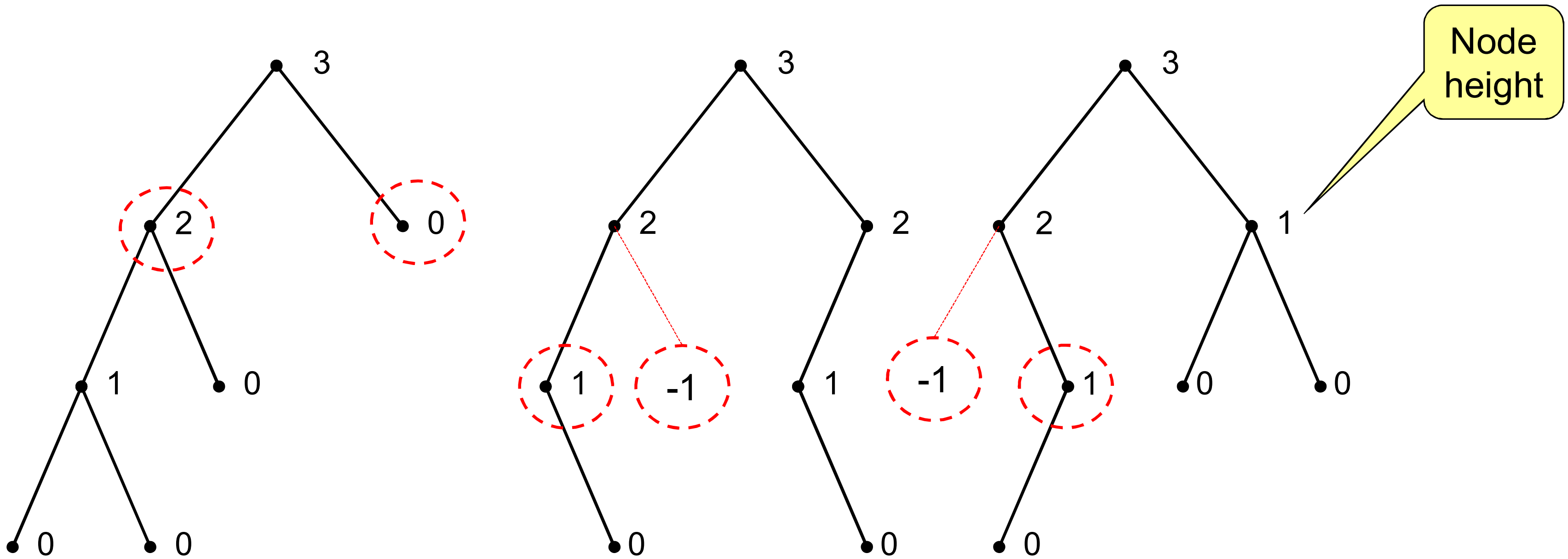
- A binary tree is said to be ***balanced*** if
  - at each node, the *height* of its *left* and *right* subtrees differ by *at most one*.
- Examples



Unbalanced Binary Trees

- An empty tree has ***height*** **-1**

- Examples of *unbalanced* binary trees

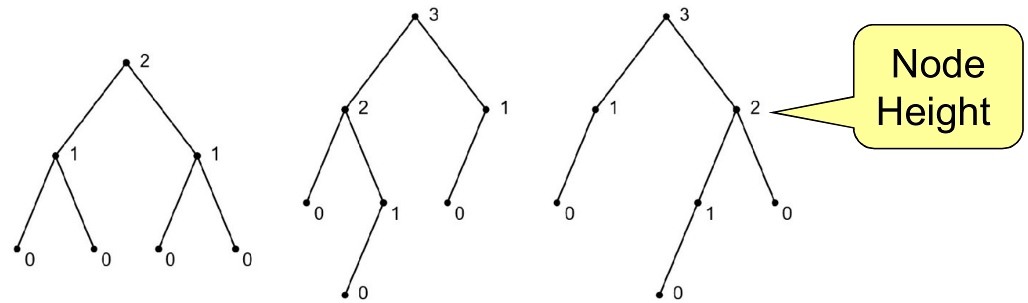


Tree-balancing Strategy

- We want to avoid the O(N) time complexity associated with unbalanced BSTs.

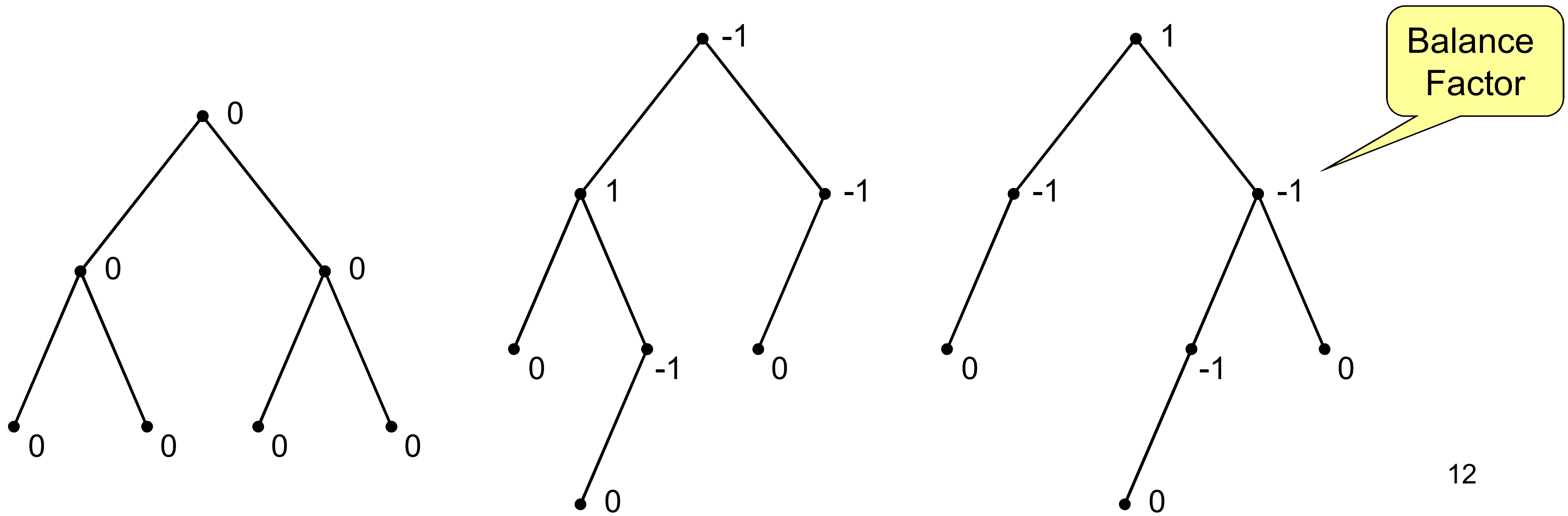
- The idea is to keep a BST balanced as we build it.
  - During insertion, *rearrange* the nodes in the BST whenever it becomes out of balance.

Balance Factors

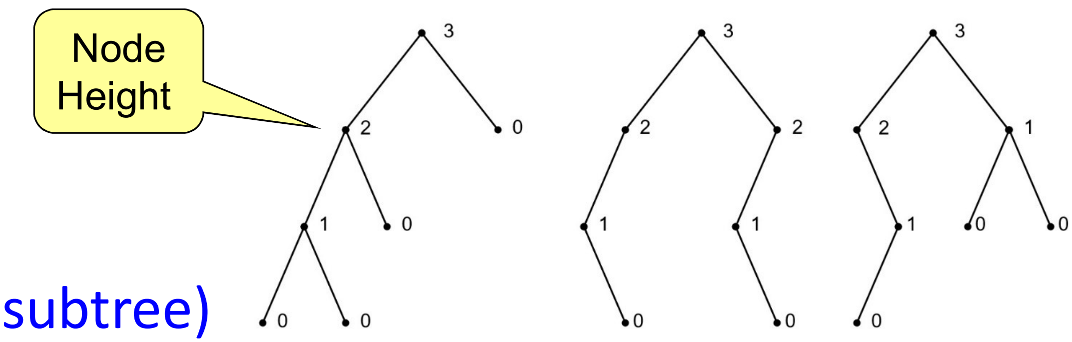


- To keep track of whether a BST is balanced, we associate with each node a ***balance factor***, which is

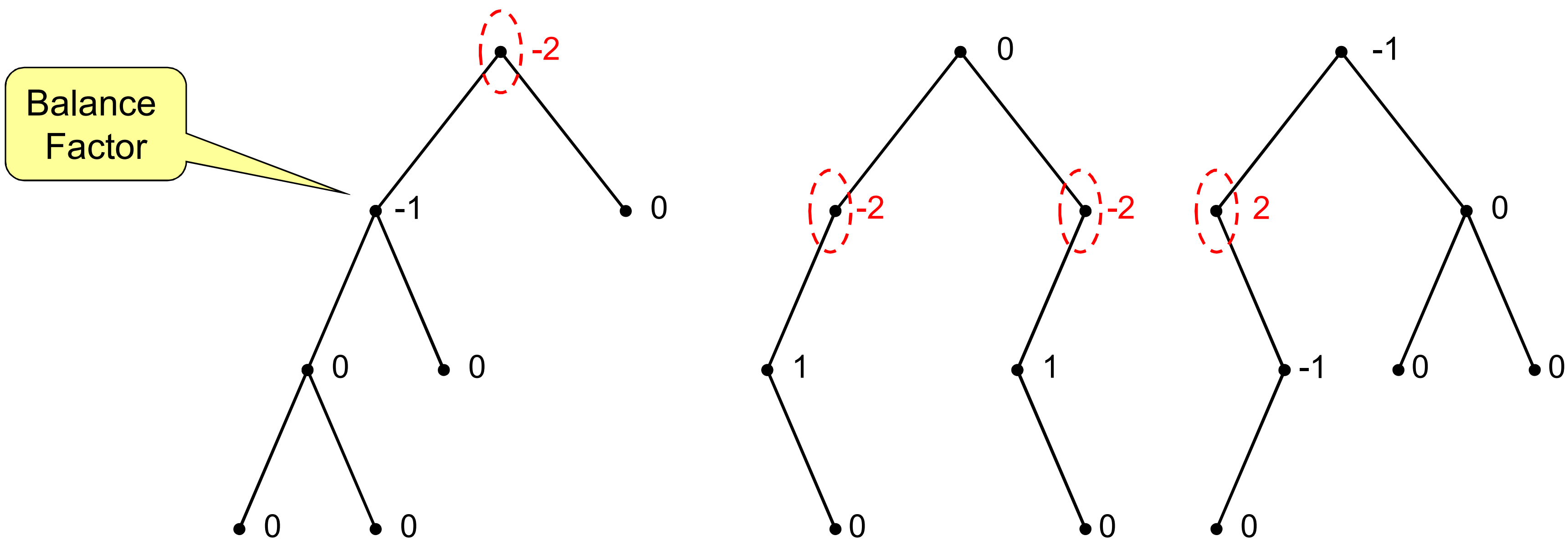
  Height(right subtree) - Height(left subtree)



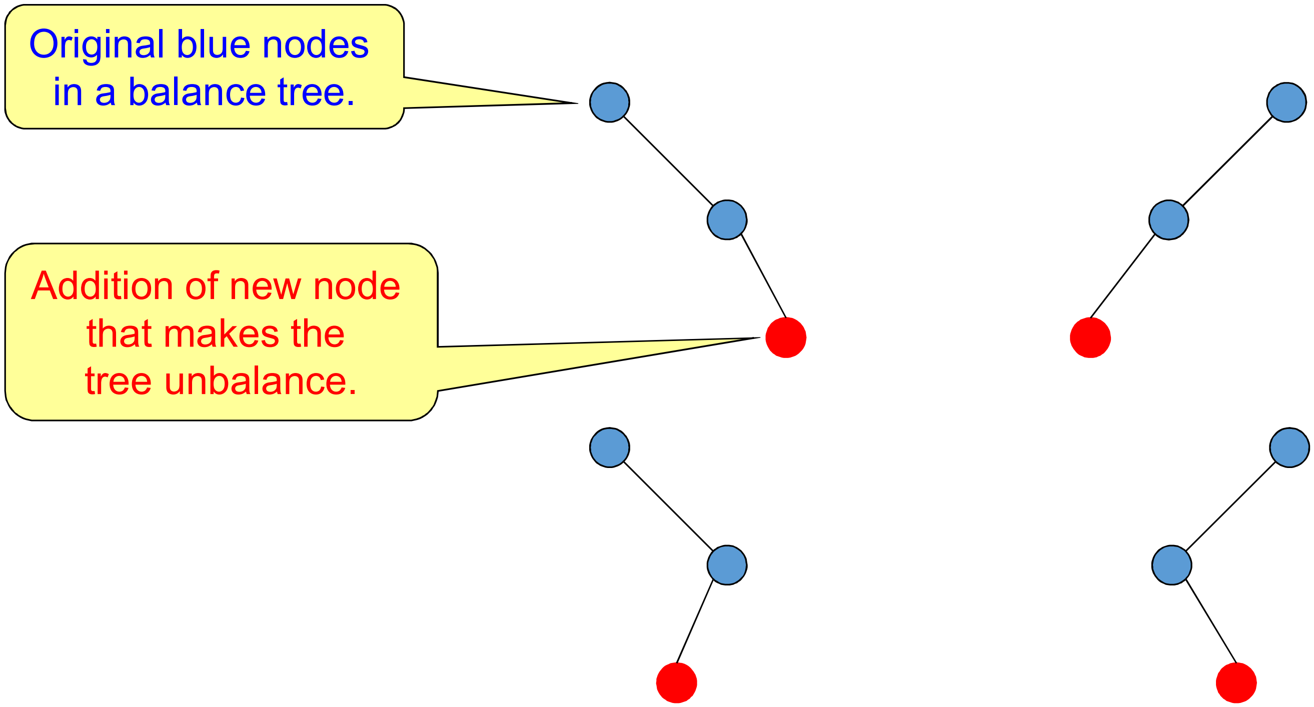
Balance Factors



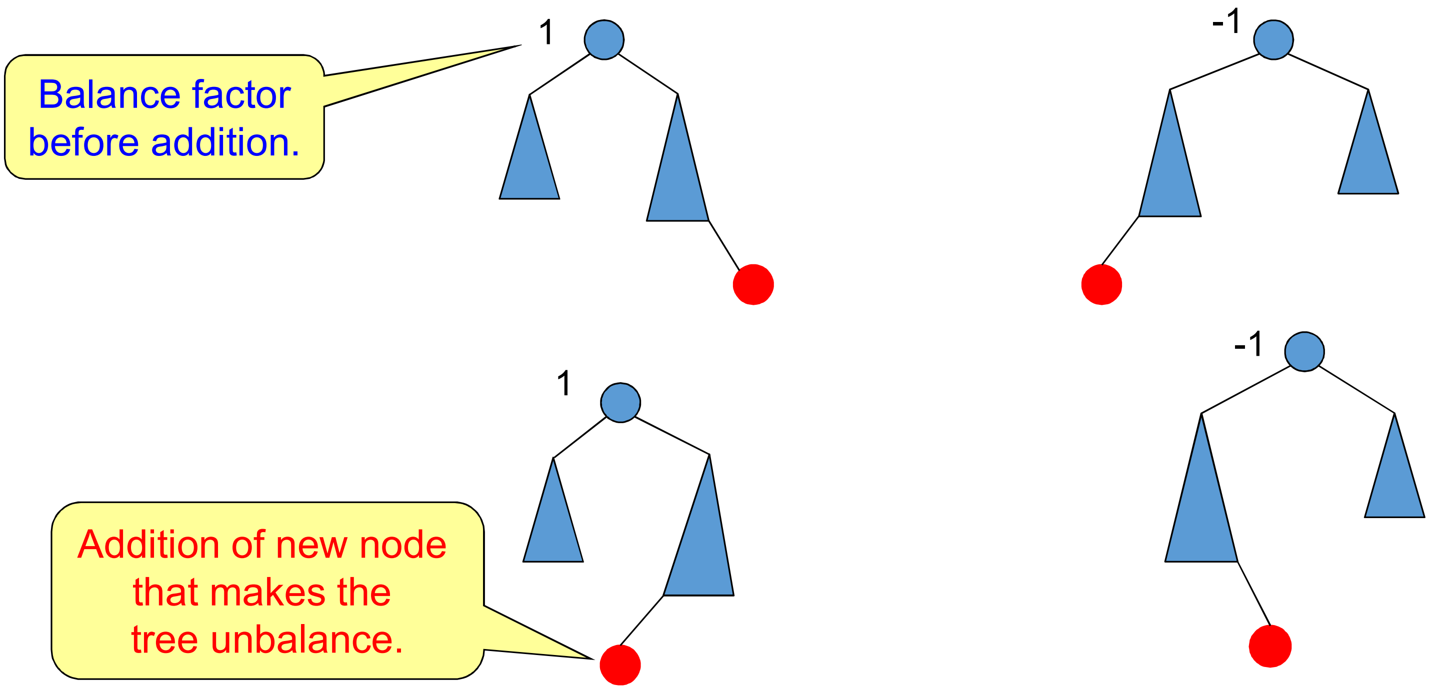
Height(right subtree) - Height(left subtree)



Simple cases

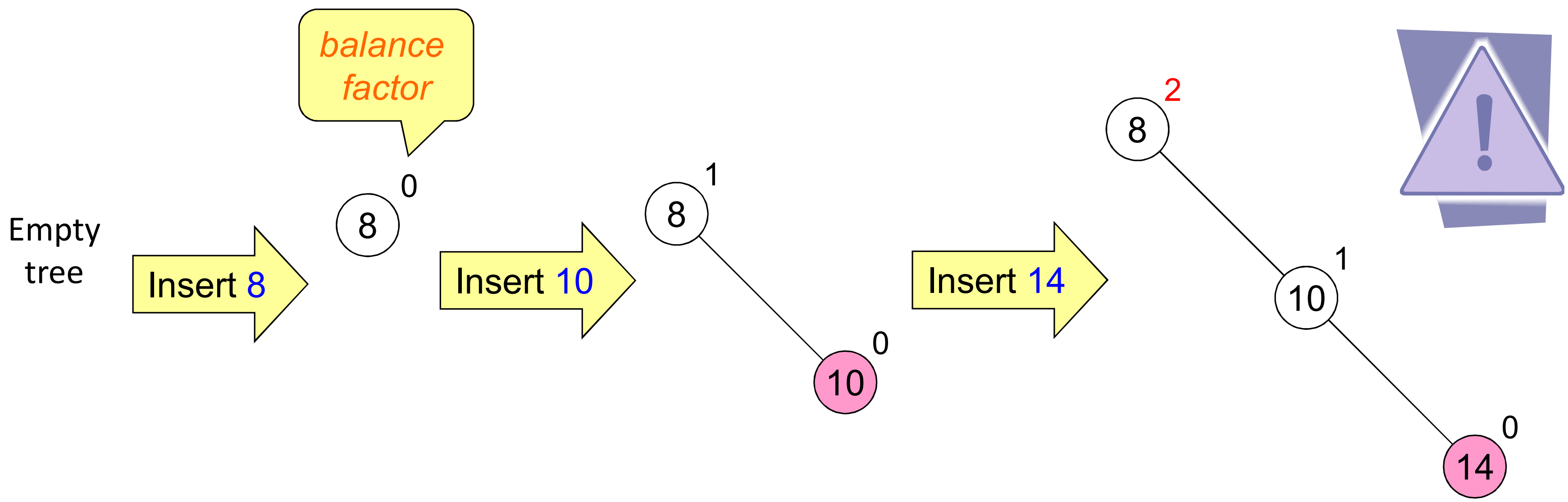


More general cases



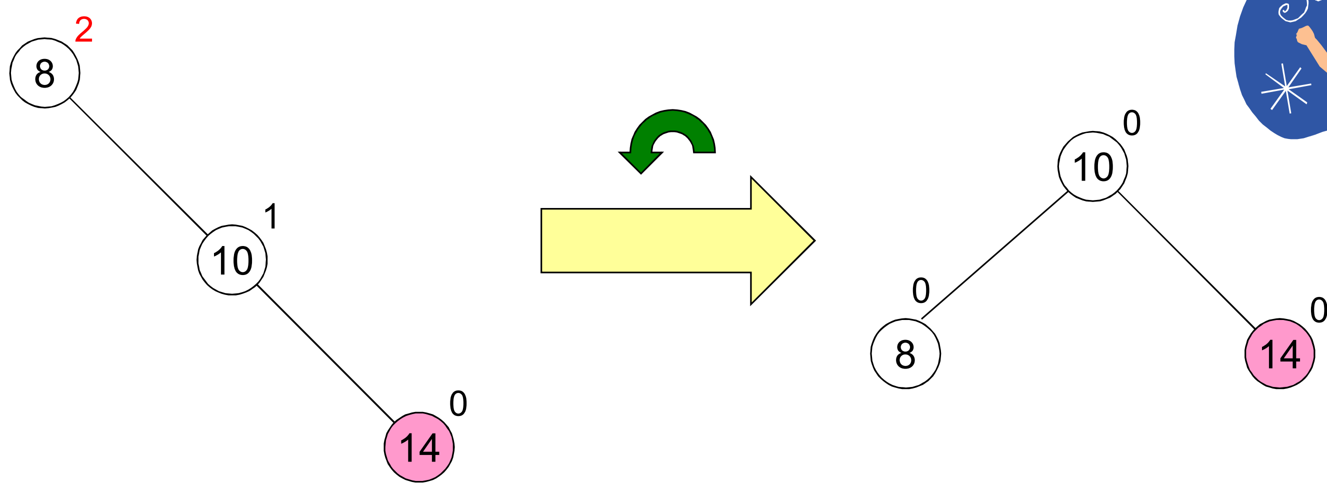
Example

- For simplicity, we use numbers as keys.
- From an empty BST, insert the keys 8, 10, and 14.



Fixing the Imbalance

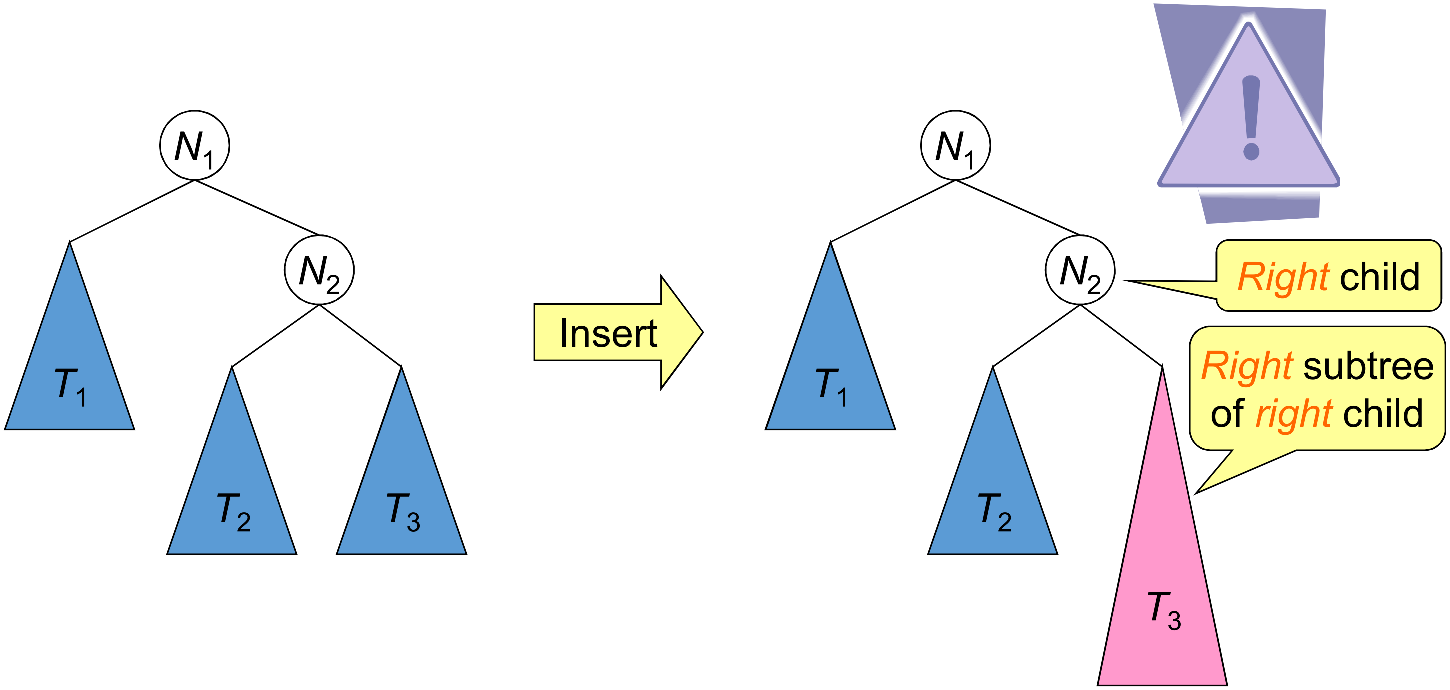
- To fix the imbalance, we *rearrange* the nodes.



1. Node 10 moves *up* to become the root.
2. Node 8 moves *down* to become the child of node 10.

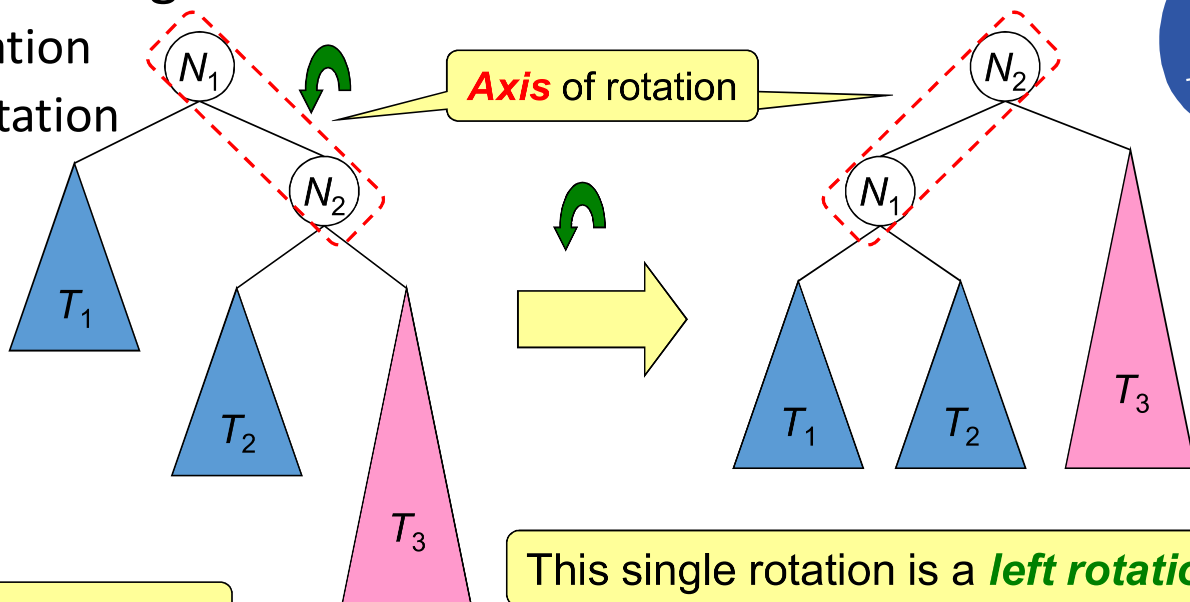
Insertion Causes Imbalance

- In general, such kind of imbalance occurs when a node is inserted into the *right* subtree of the *right* child.



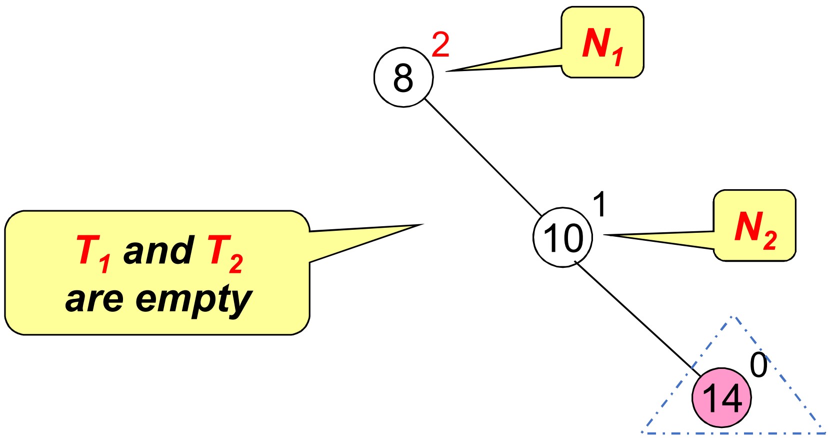
Single Rotation

- To fix such imbalance, we perform an operation called a ***single rotation***.
- Two types of single rotation
  - Left rotation
  - Right rotation

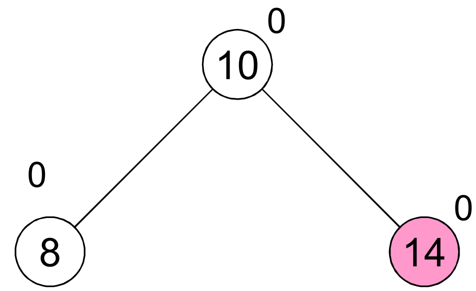


This single rotation is a ***left rotation***.

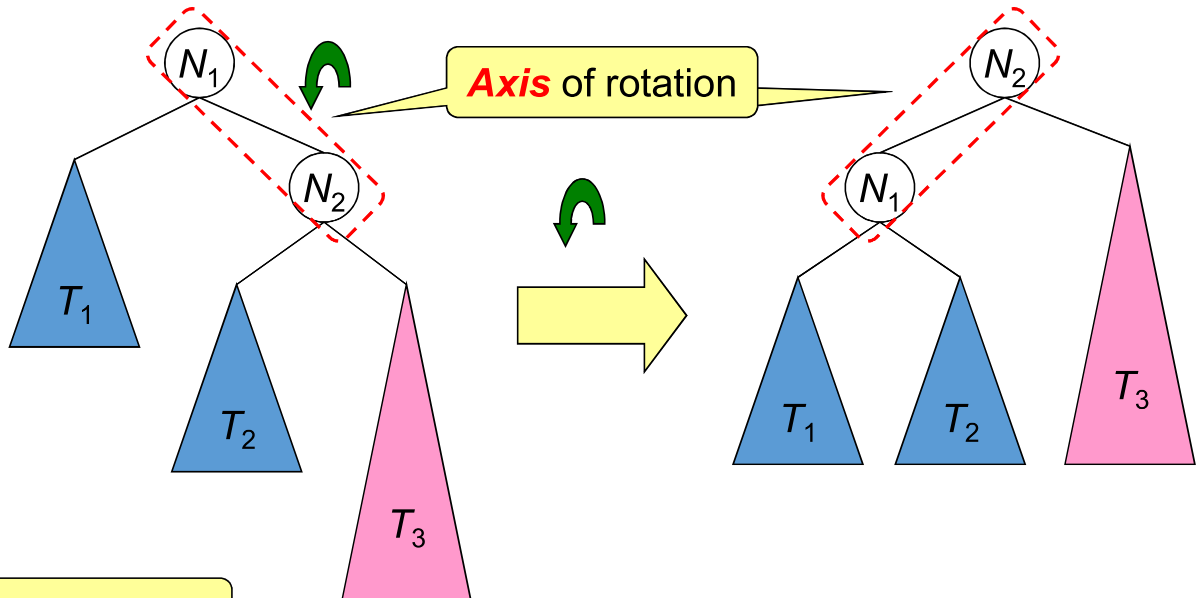
$T_1 < N_1 < T_2 < N_2 < T_3$



2
8
N1
1
10
N2
T_1 and T_2 are empty
0
14



0
10
0
8
0
14

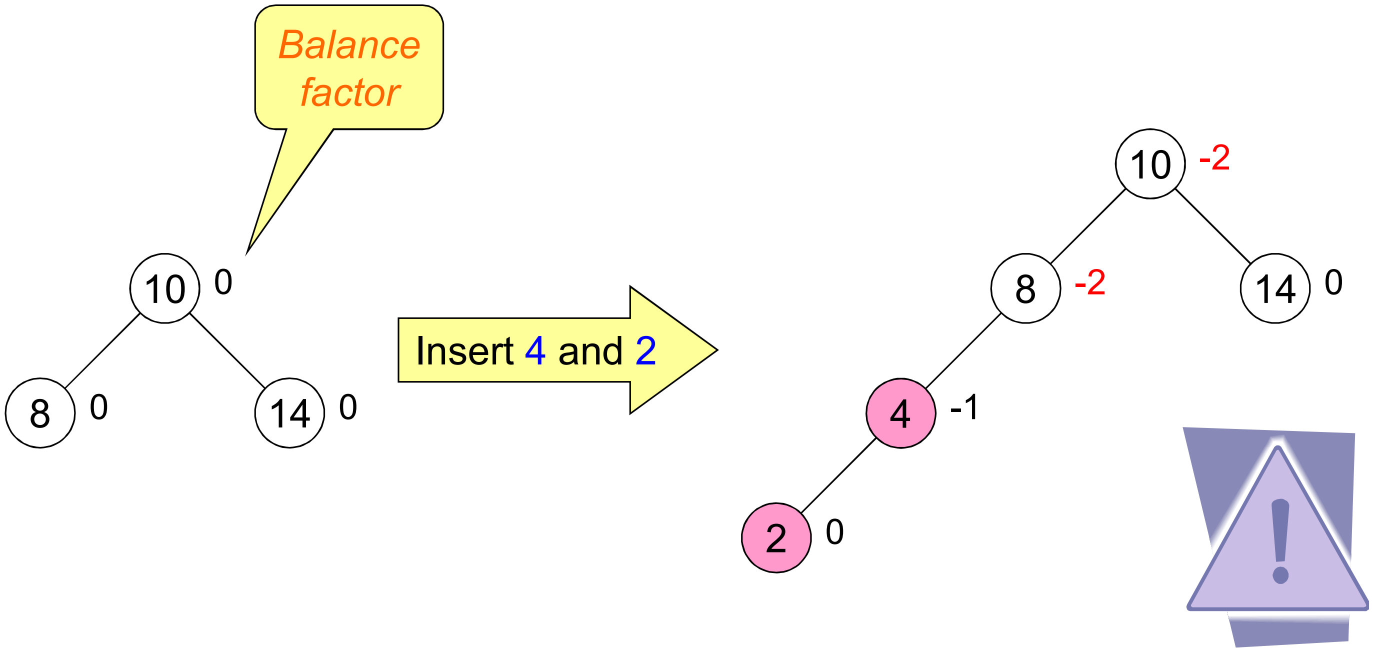


N1
N2
Axis of rotation
T1
T2
T3
N2
N1
T1
T2
T3

$T_1 < N_1 < T_2 < N_2 < T_3$

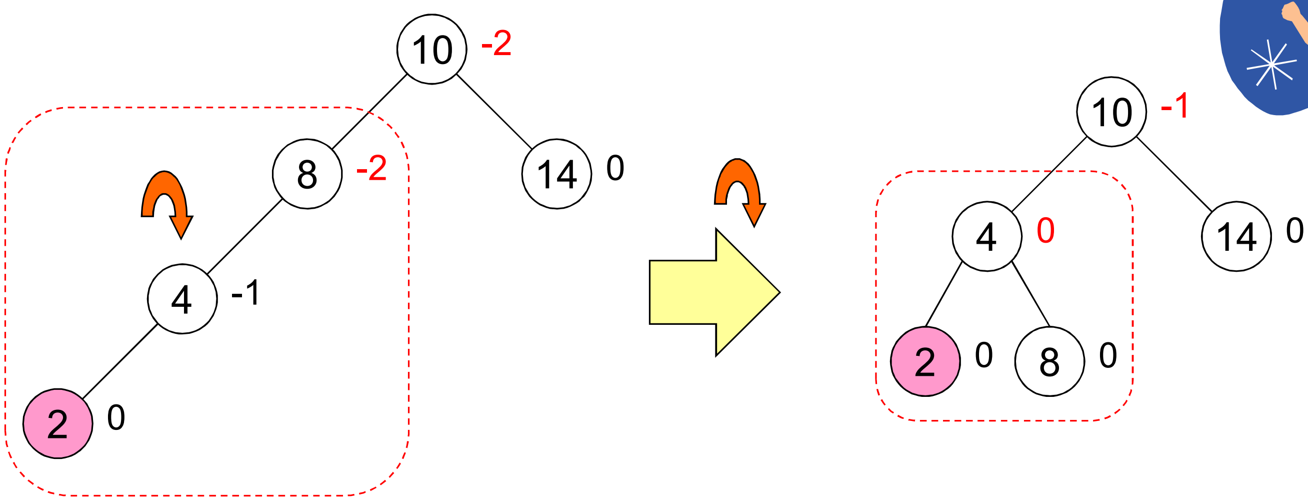
Example

- We continue to insert the keys 4 and 2 into our rotated BST.



Single Rotation

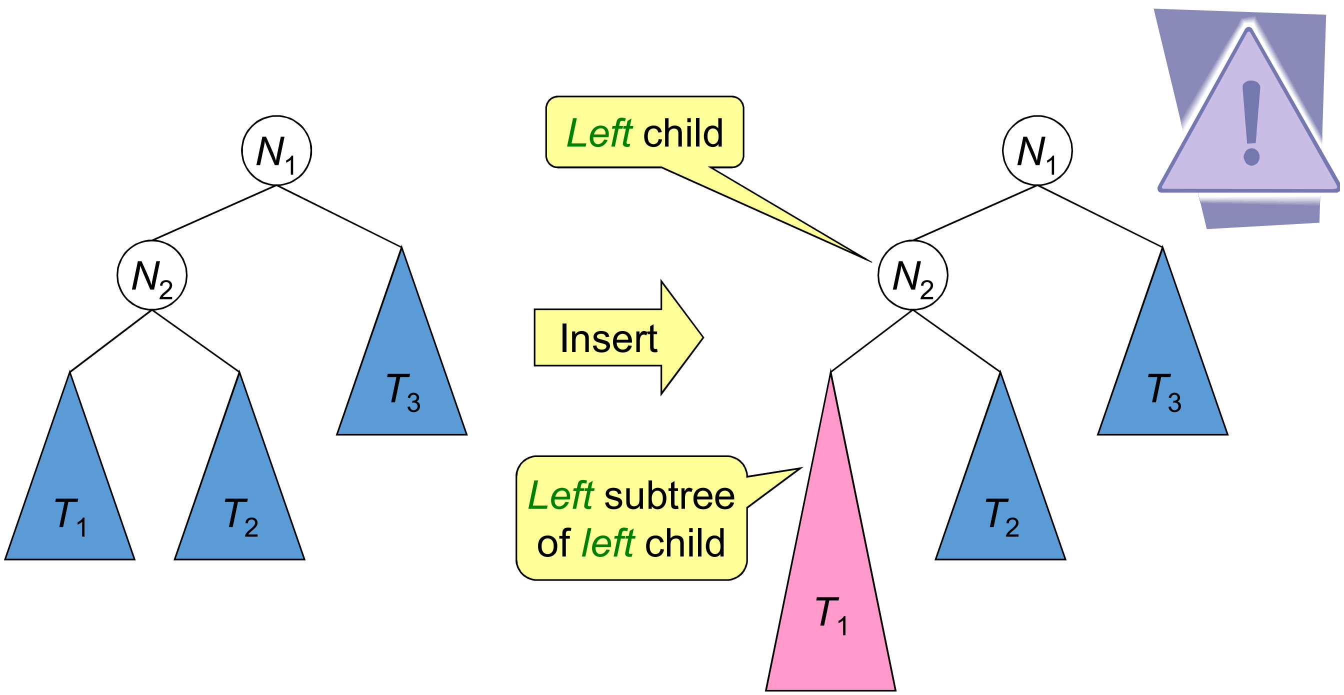
- To fix the imbalance, we perform a ***right rotation***.



1. Node 4 moves *up* to become the root of the subtree.
2. Node 8 moves *down* to become the child of node 4.

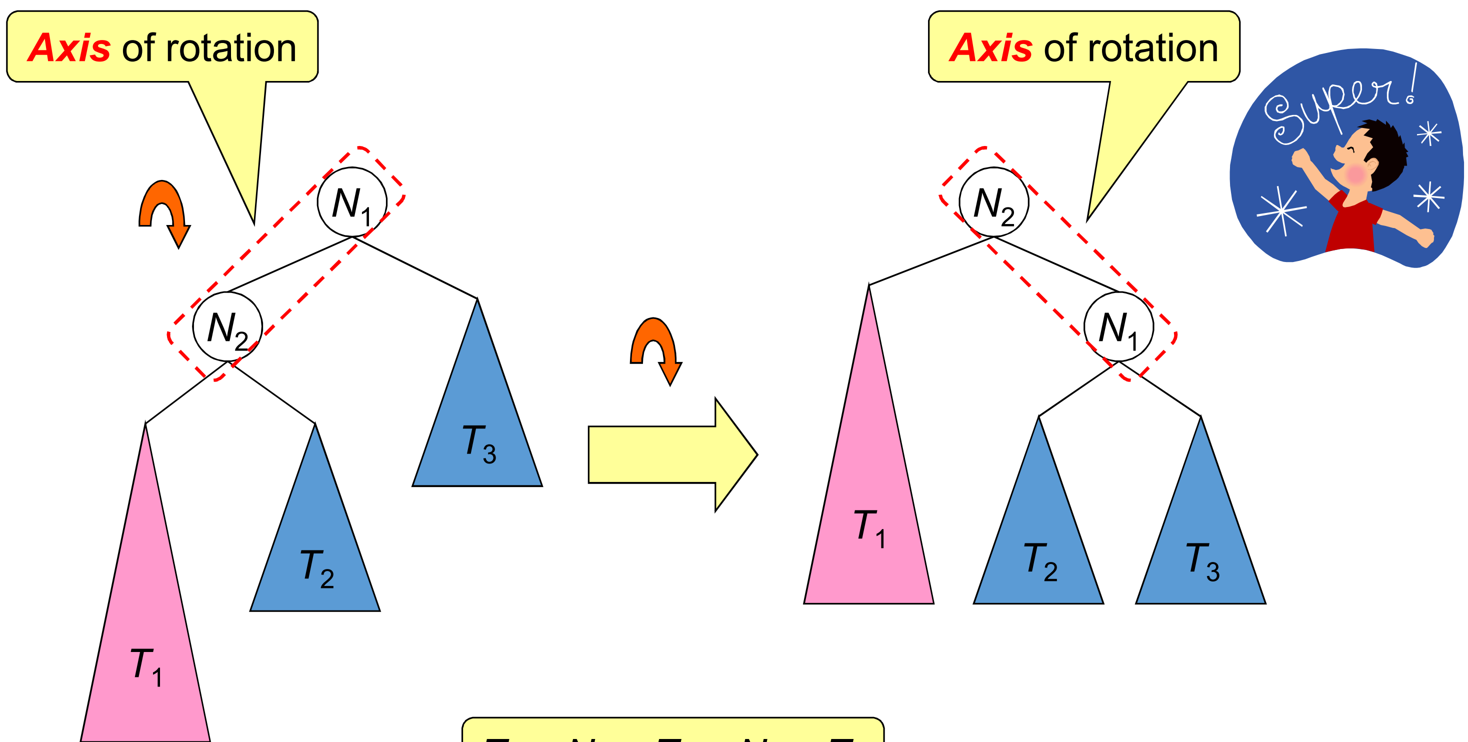
Insertion Causes Imbalance

- In general, such kind of imbalance occurs when a node is inserted into the *left* subtree of the *left* child.

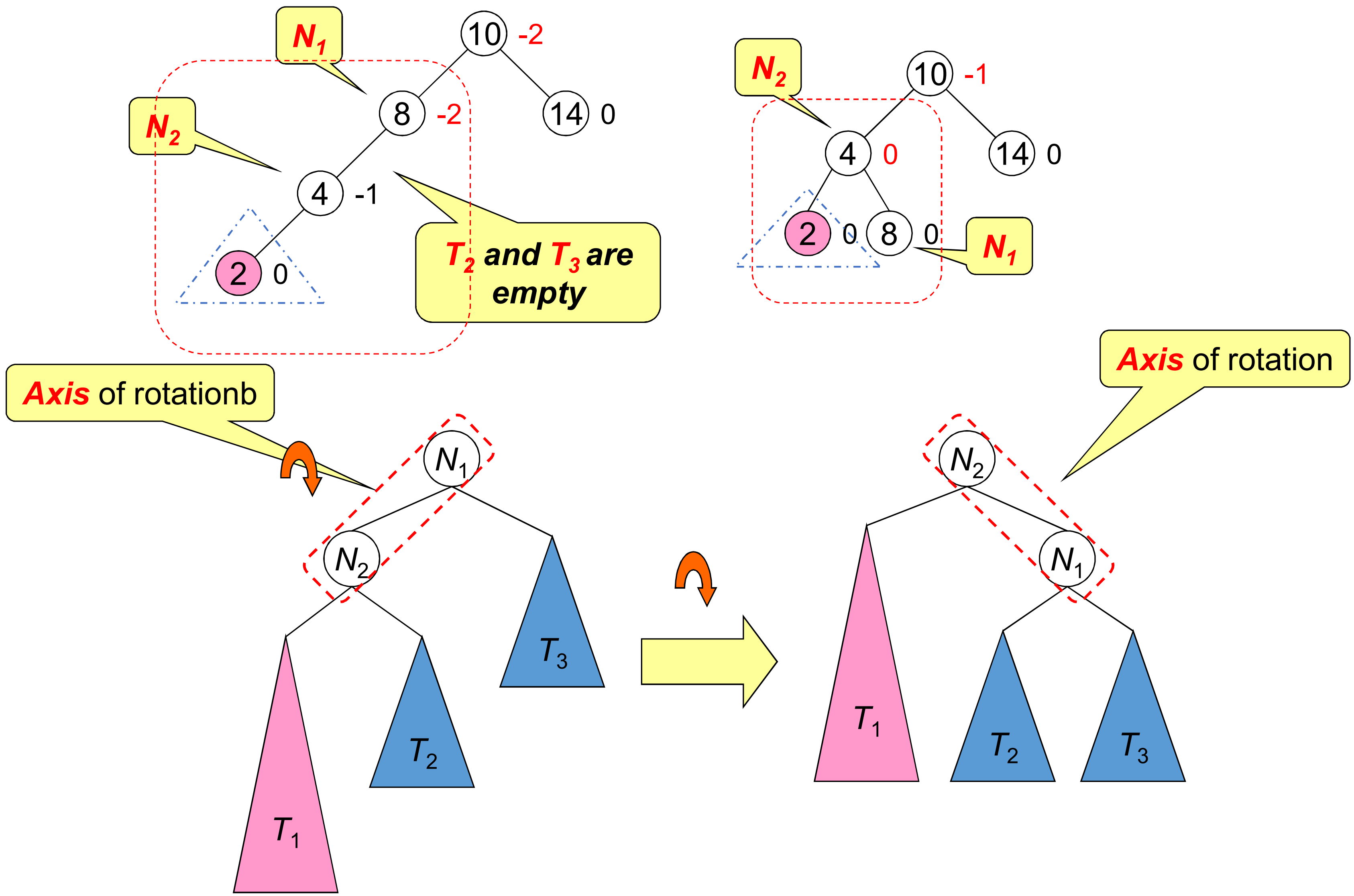


Single Rotation

- A ***right rotation*** can fix such imbalance.

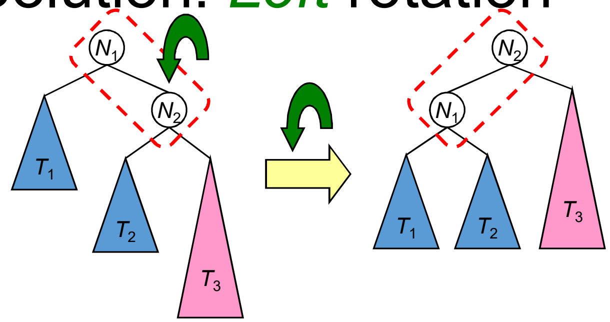
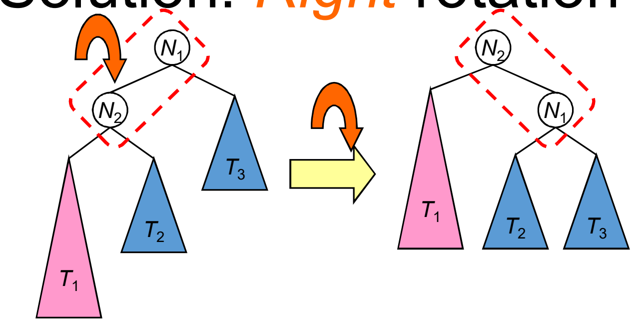


$T_1 < N_2 < T_2 < N_1 < T_3$



N1
10 -2
8 -2
14 0
N2
4 -1
2 0
T2 and T3 are empty
N2
10 -1
4 0
14 0
2 0
8 0
N1
Axis of rotationb
N1
N2
T3
T2
T1
Axis of rotation
N2
N1
T1
T2
T3

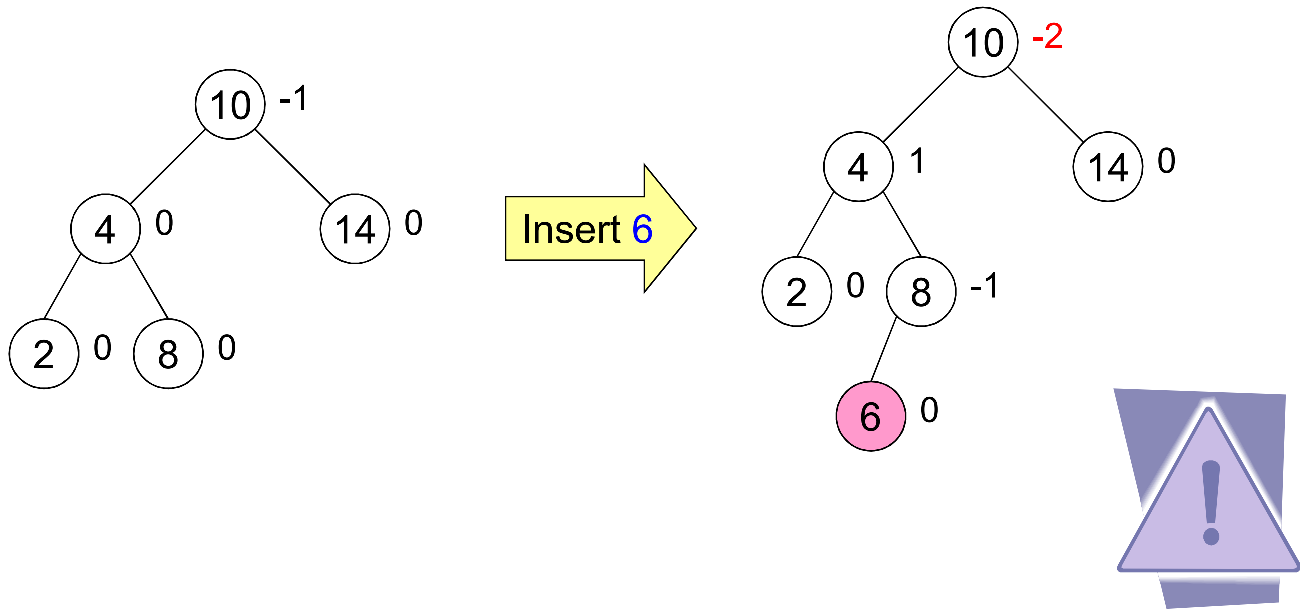
Summary So Far (Single Rotations)

<table>
<tr>
<td>Case 1:
Imbalance occurs when a node is inserted in the right substree of the right child.</td>
<td>Case 2:
Imbalance occurs when a node is inserted in the left subtree of the left child.</td>
</tr>
<tr>
<td>Solution: Left rotation

</td>
<td>Solution: Right rotation

</td>
</tr>
</table>

Both *left* and *right* rotations are single rotations.

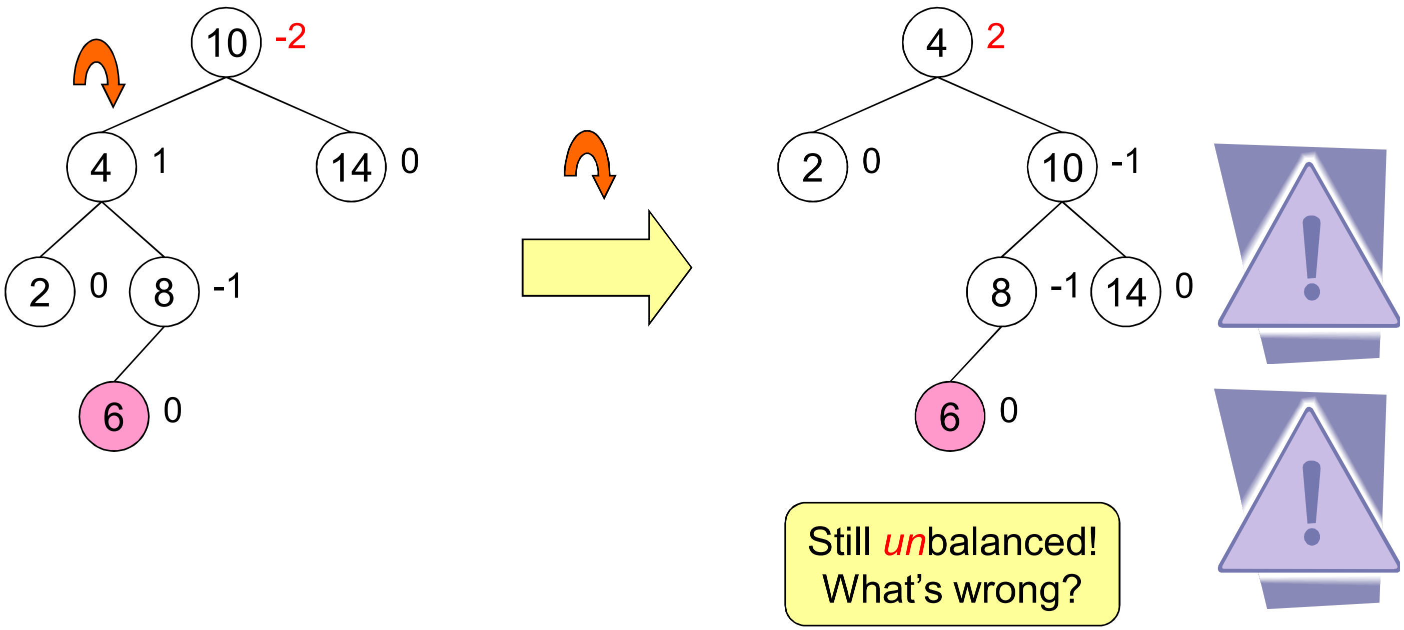
Example

- Next, we insert the key 6.



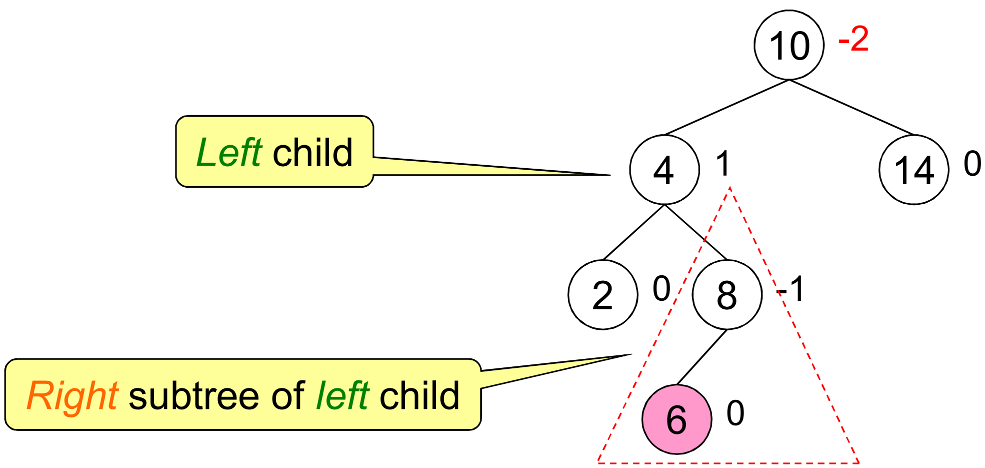
Single Rotation Not Working

- Let's do a *right* rotation along the 10-4 axis.



Single Rotation Not Working

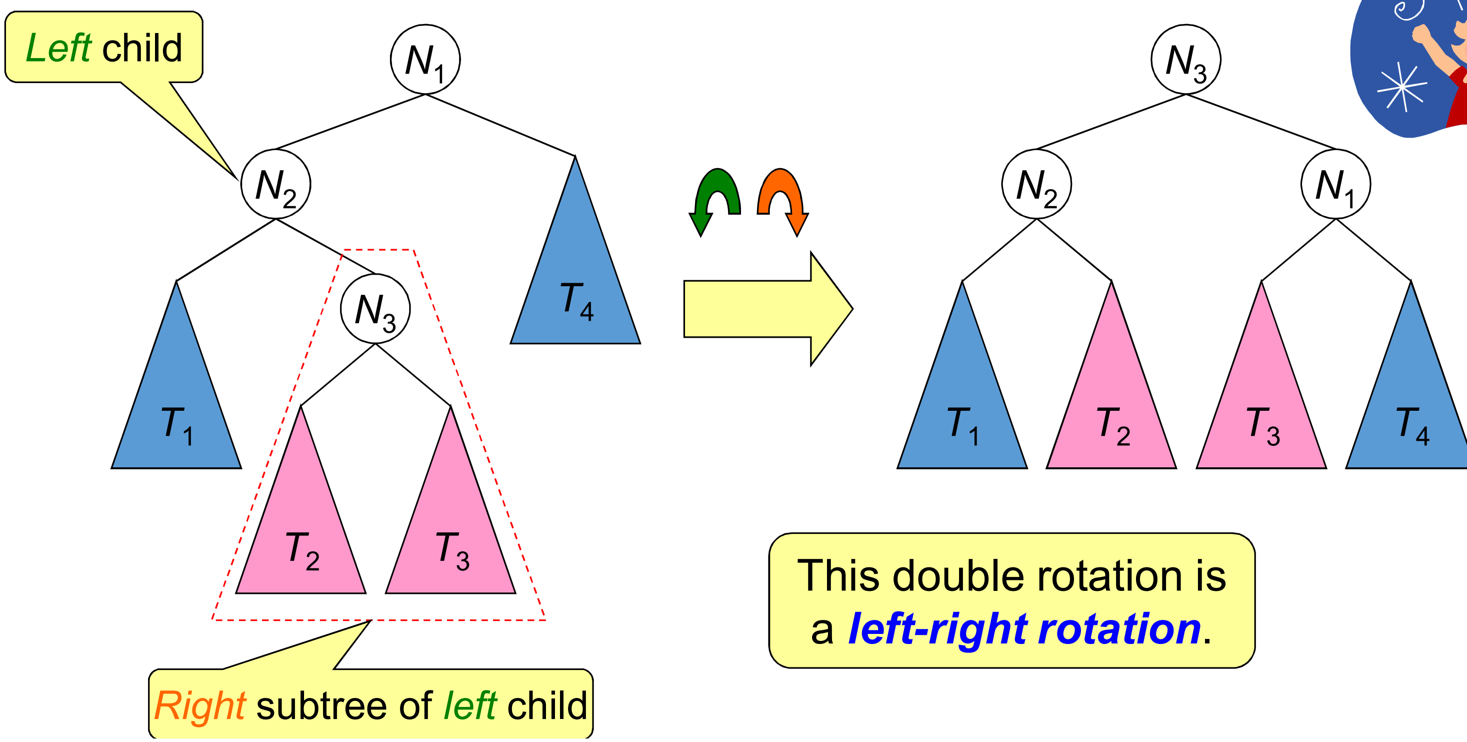
- The problem is, this imbalance occurs when a node is inserted into the *right* subtree of the *left* child.



- Single rotations *cannot* fix such imbalance.

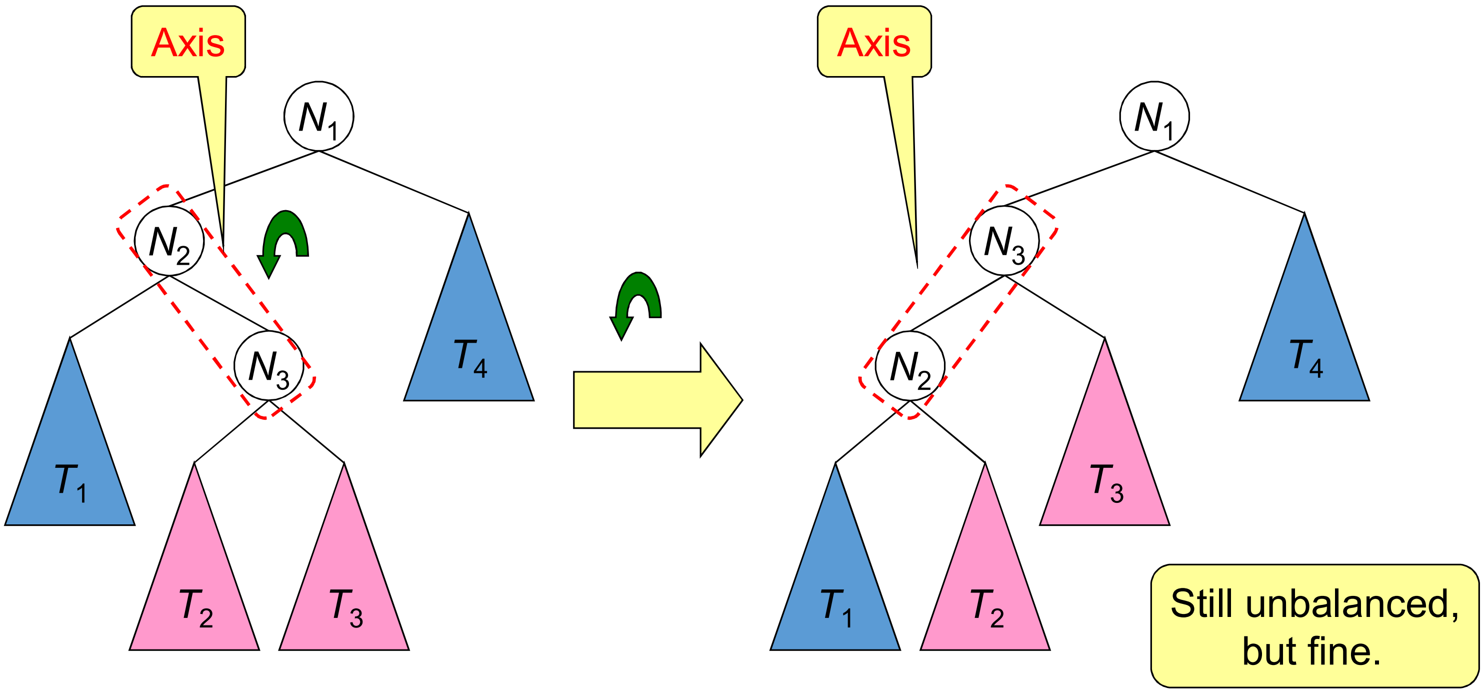
Double Rotation

- We need a ***double rotation*** in this case.



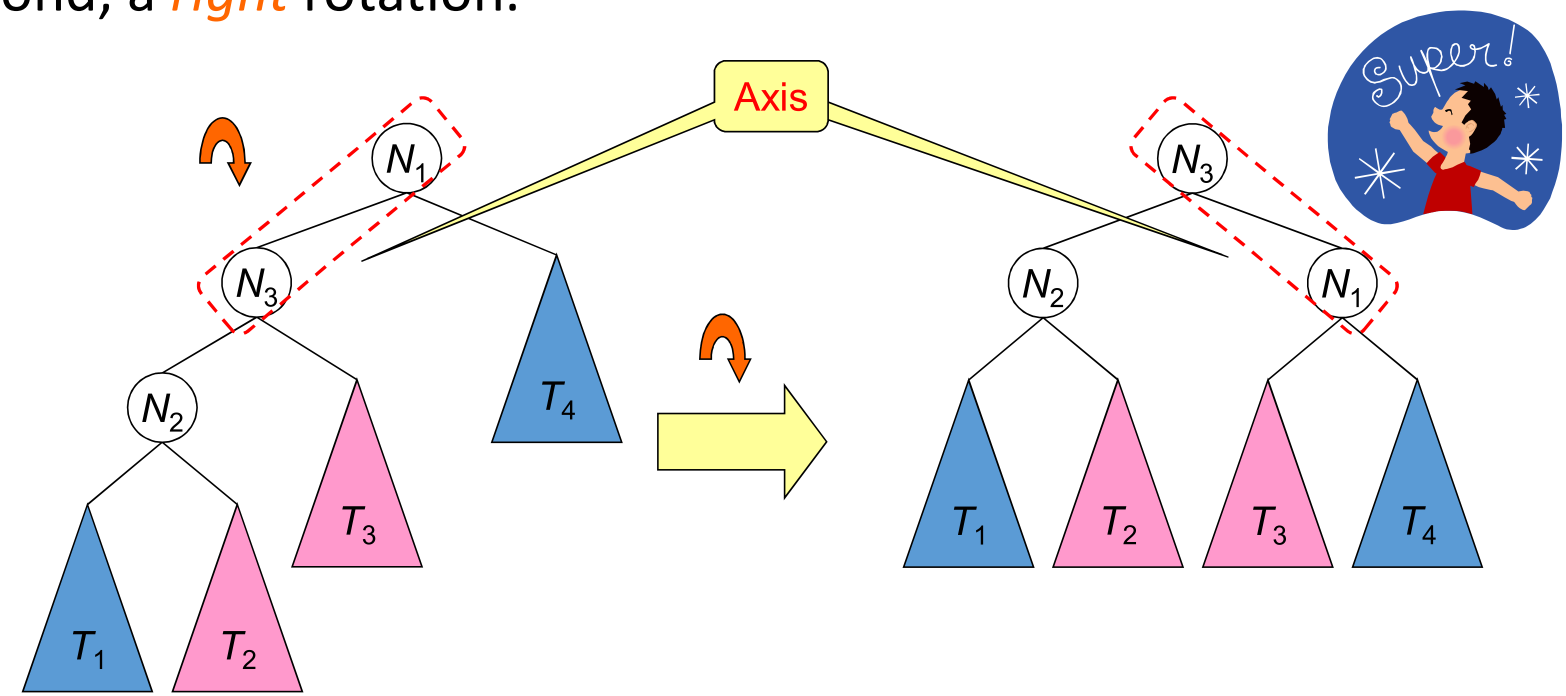
Double Rotation: Step 1

- First, a *left* rotation on the *left* subtree.



Double Rotation: Step 2

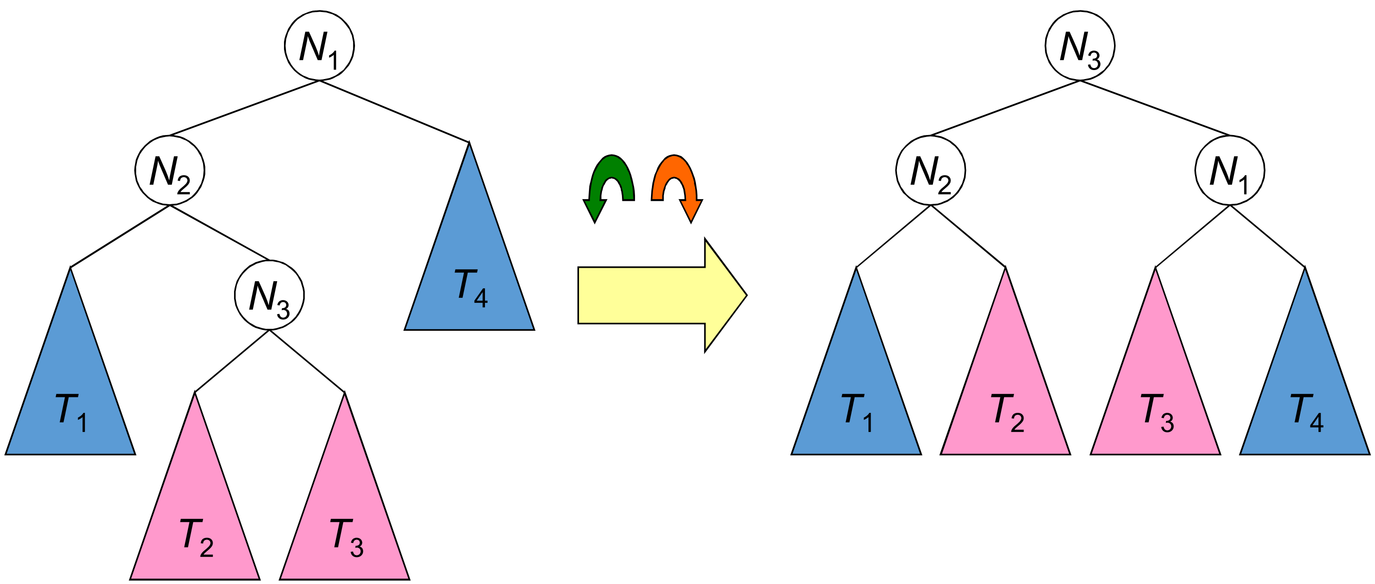
- Second, a *right* rotation.



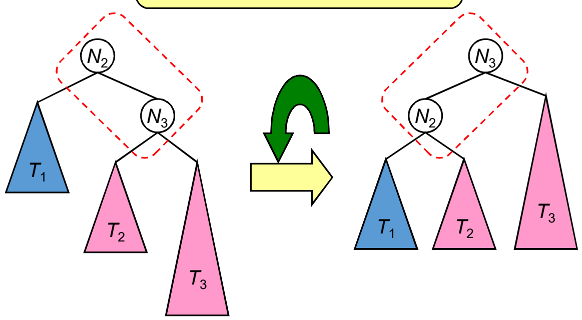
Double Rotation

- A ***left-right rotation*** is simply a *left* rotation (on the *left* subtree) followed by a *right* rotation.

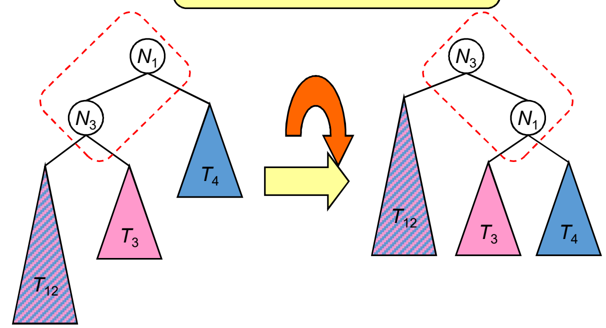
A *left*-*right* rotation fixes the imbalance caused by an insertion into the *right* subtree of the *left* child.

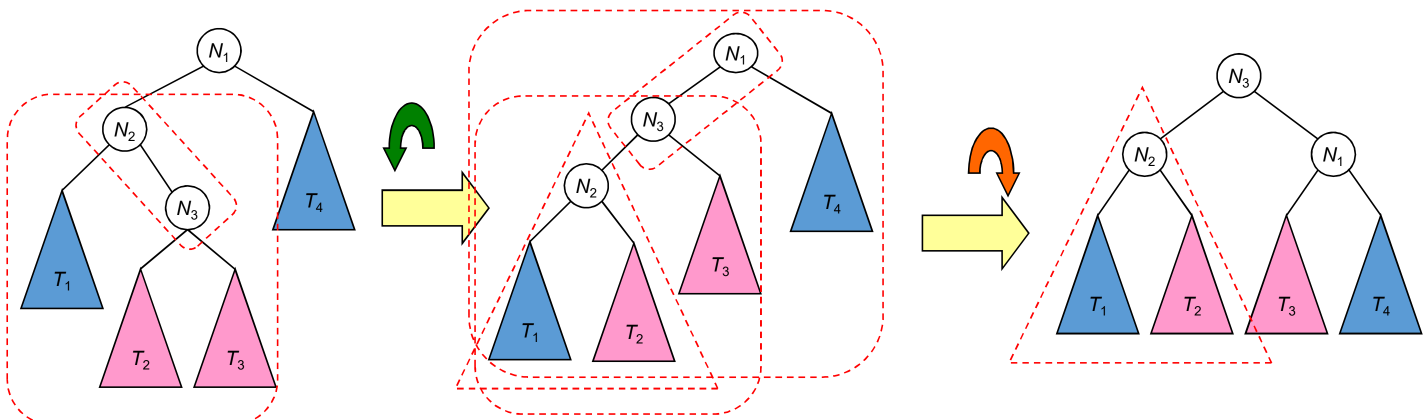


left rotation



right rotation

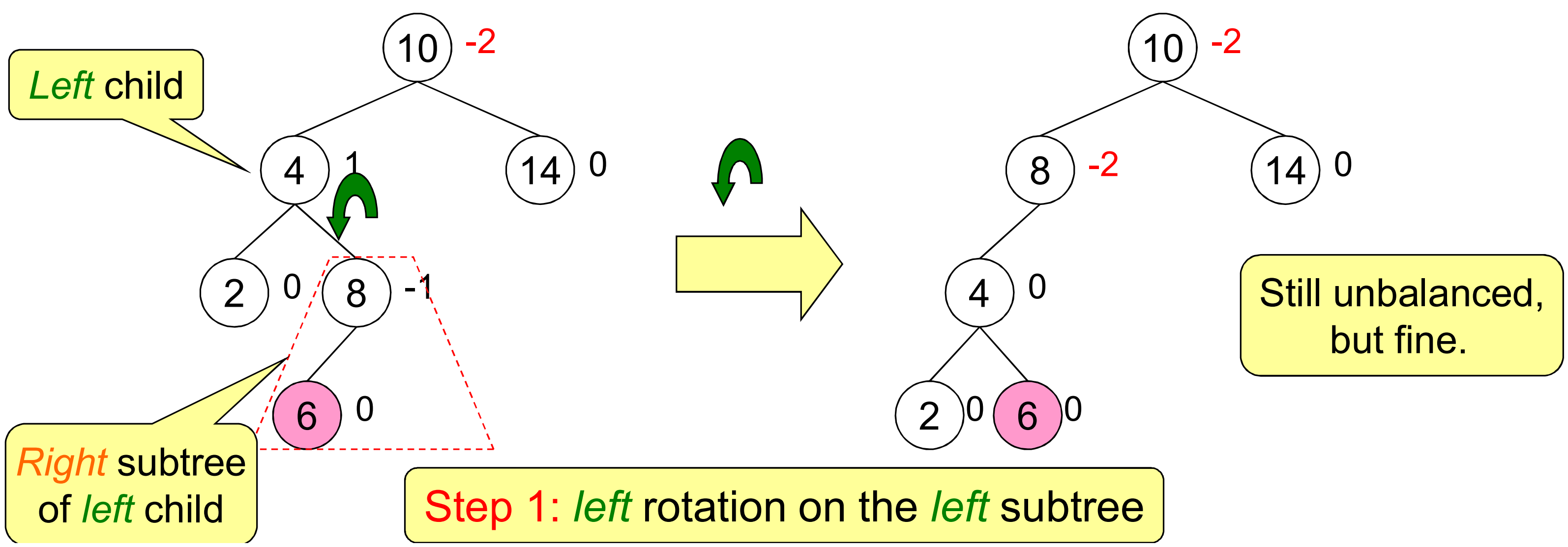




This double rotation is a ***left-right rotation***.

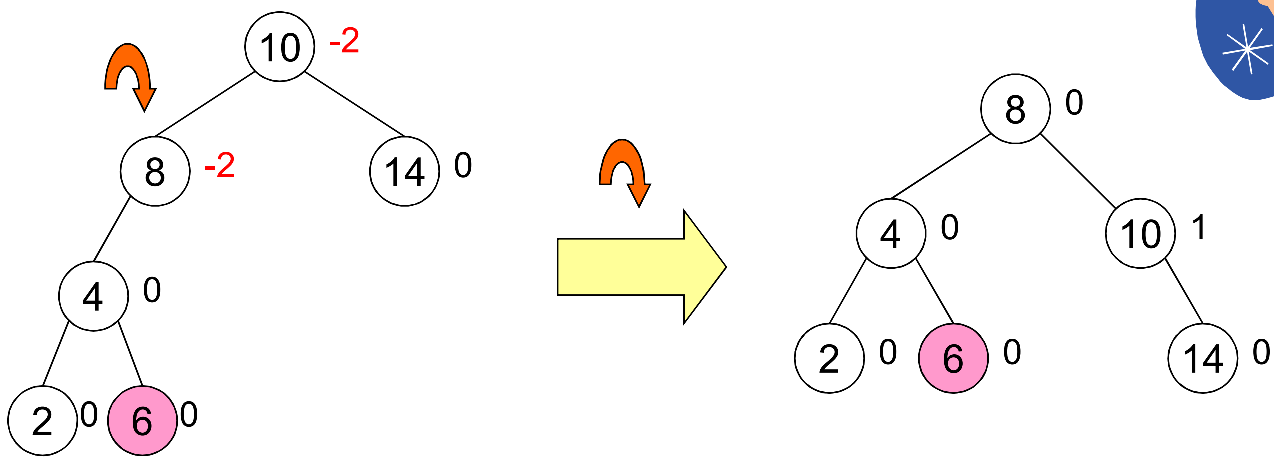
Example

- Let's go back to our example where key 6 is just inserted into the *right* subtree of the *left* child.
- We need to perform a *left*-*right* rotation.

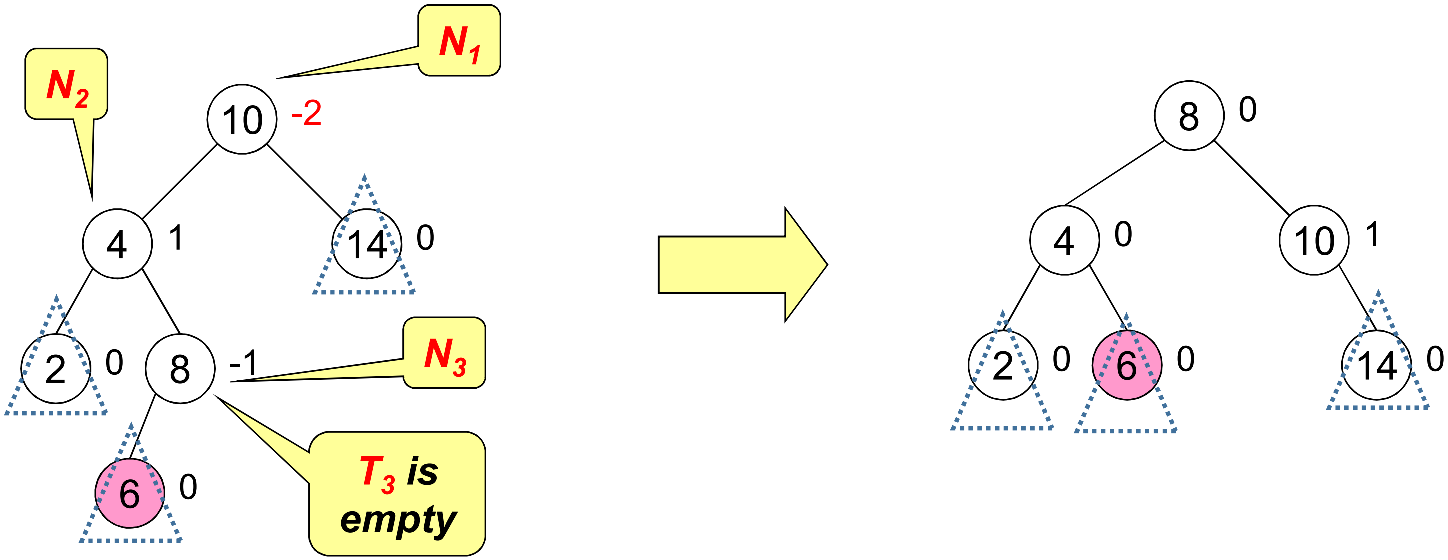


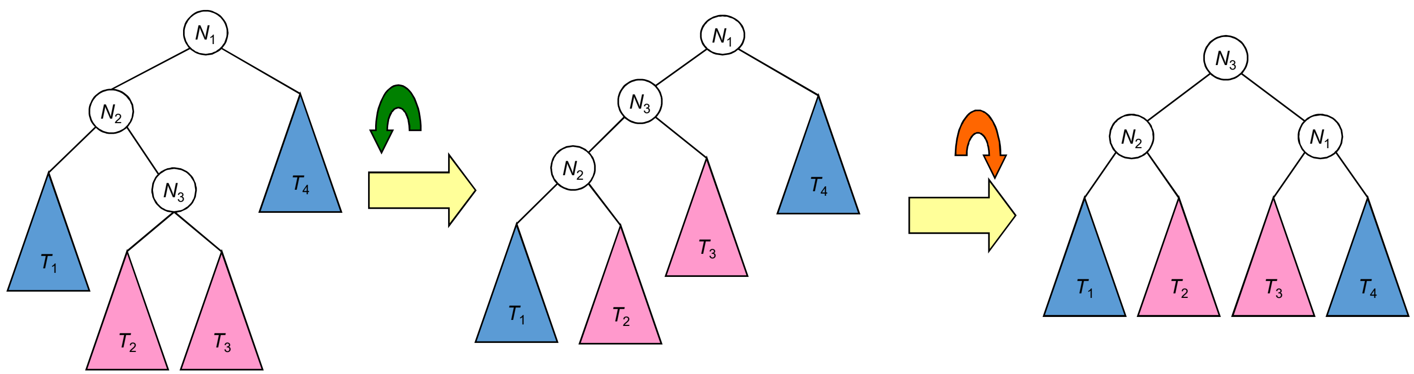
Example

- Continue with the *left*-*right* rotation.



Step 2: *right* rotation

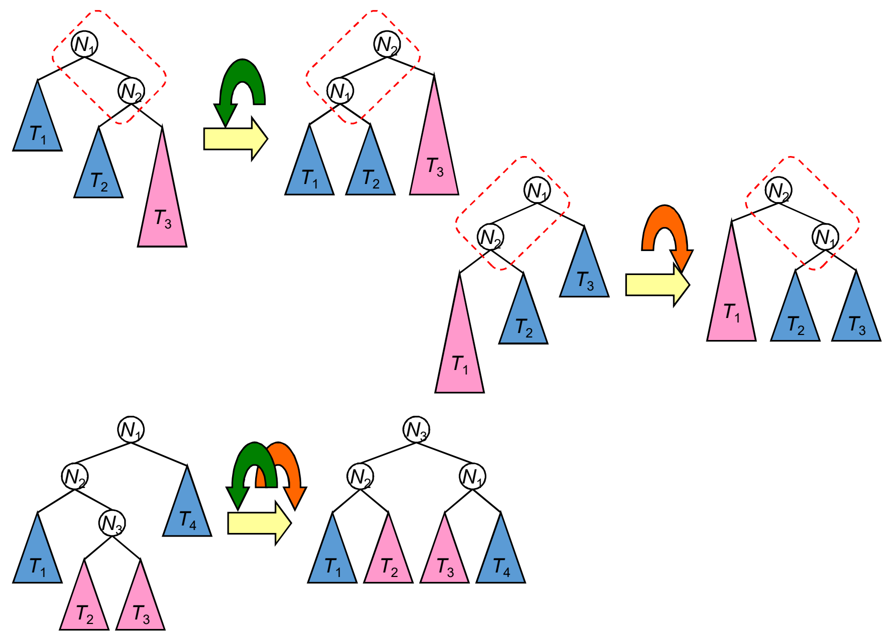




This double rotation is a ***left-right rotation***.

Summary So Far

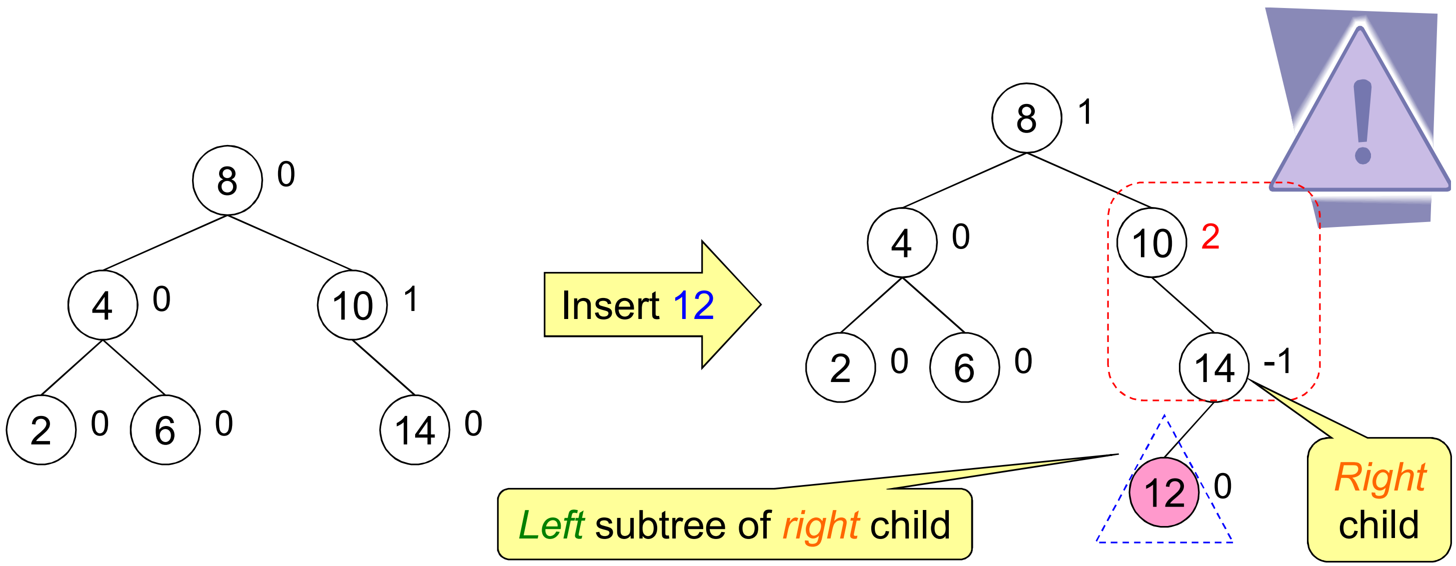
- Single rotations
  - *Left* rotations
  - *Right* rotations
- Double rotations
  - *Left*-*right* rotations



- As expected, there is a kind of double rotation called *right*-*left* rotation.

Example

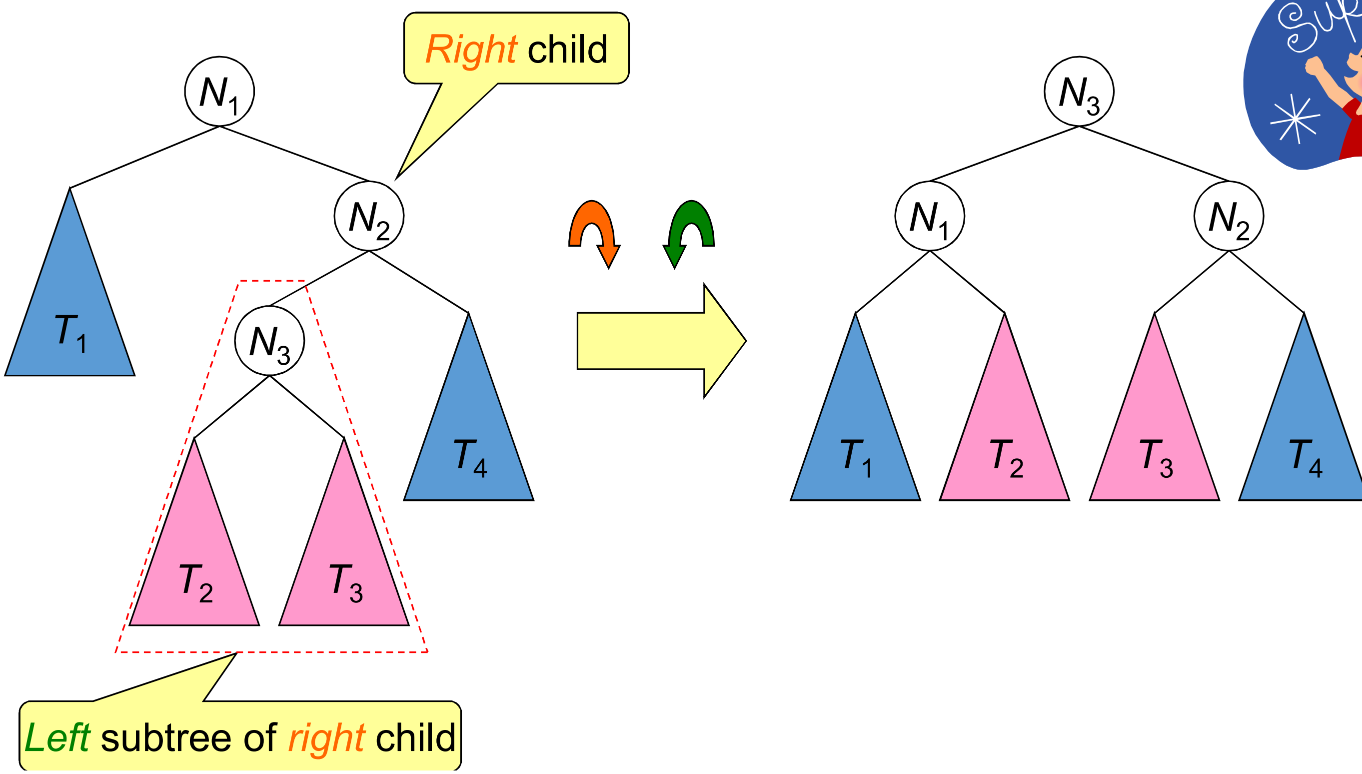
- To illustrate the idea, we insert the key 12.



Such imbalance occurs when a node is inserted into the *left* subtree of the *right* child.

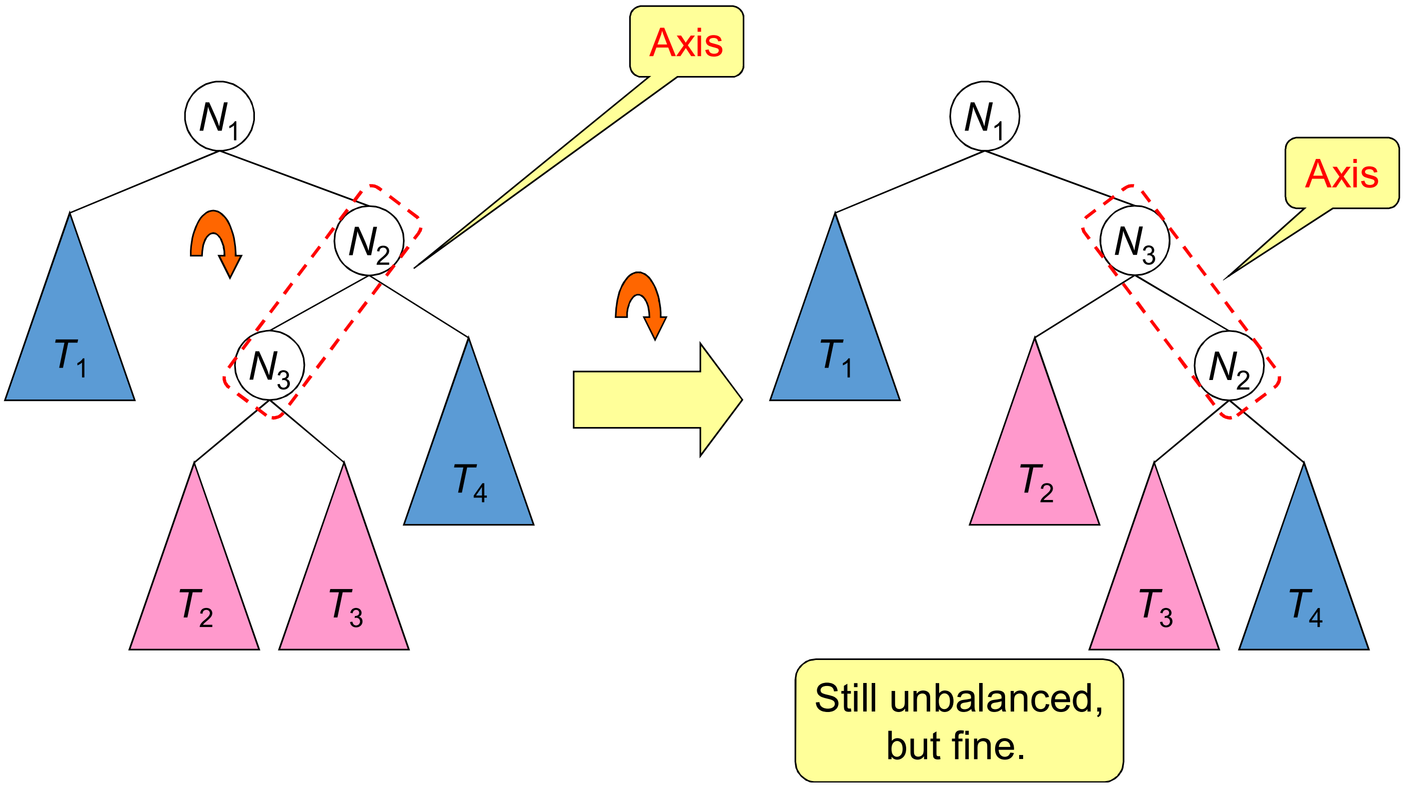
Double Rotation

- We need a ***right-left rotation*** in this case.



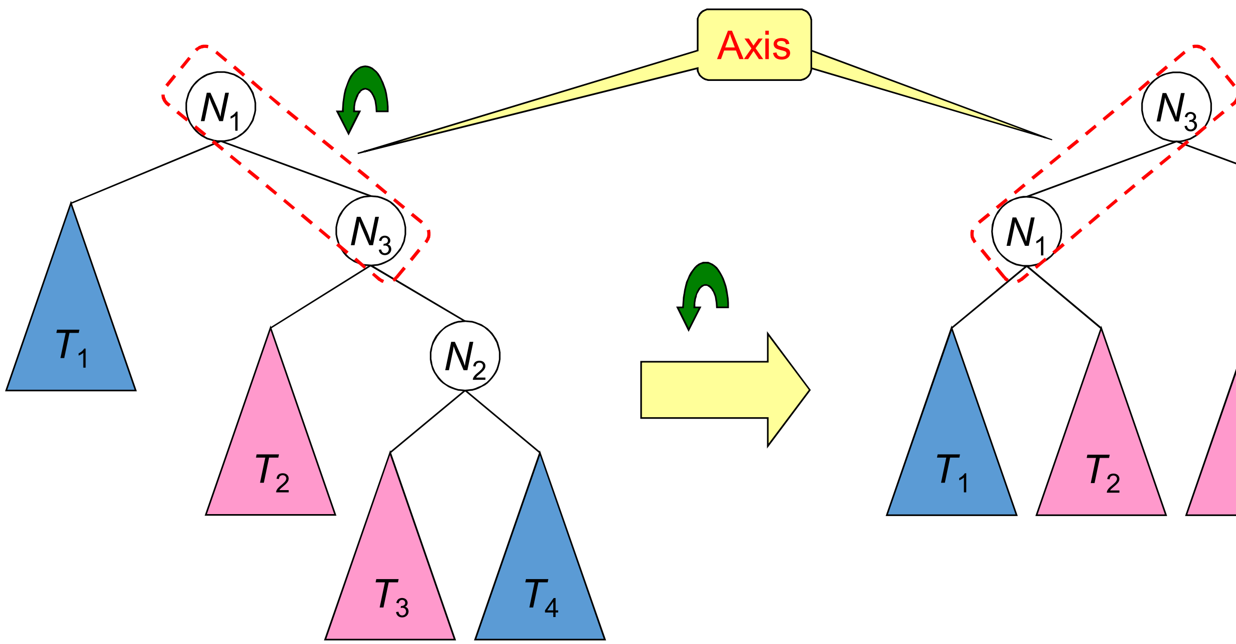
Double Rotation: Step 1

- First, a *right* rotation on the *right* subtree.



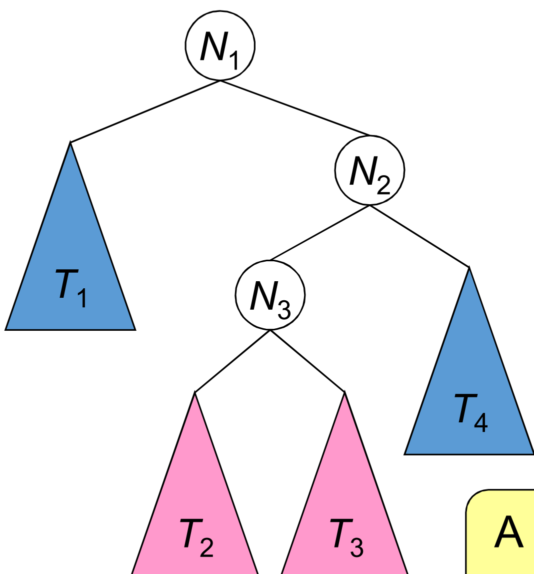
Double Rotation: Step 2

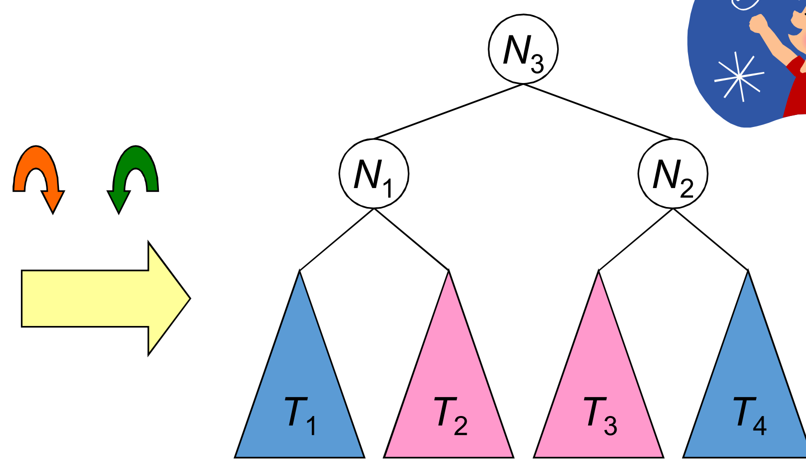
- Second, a *left* rotation.



Double Rotation

- A ***right-left rotation*** is simply a *right* rotation (on the *right* subtree) followed by a *left* rotation.

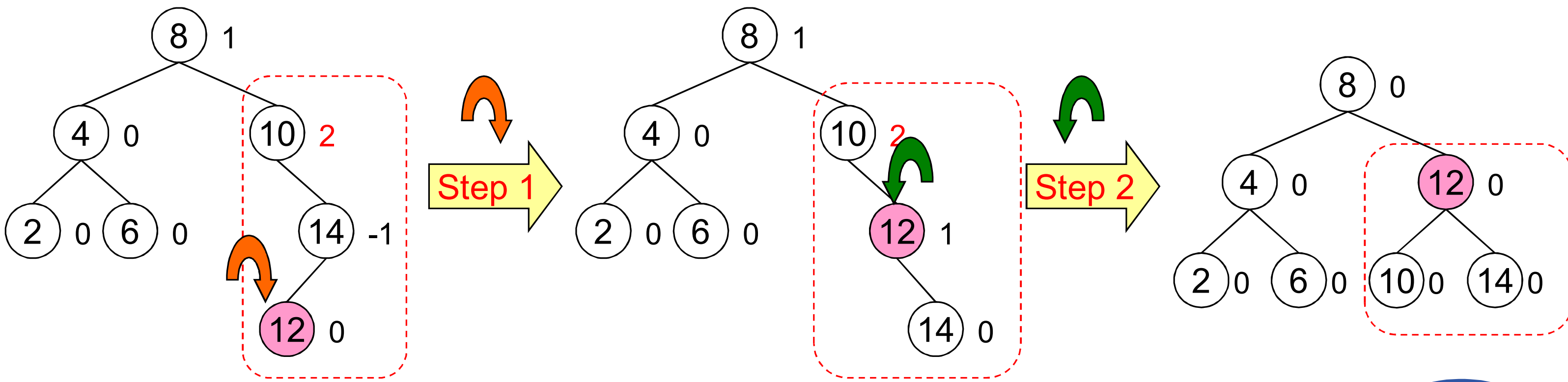




A *right*-*left* rotation fixes the imbalance caused by an insertion into the *left* subtree of the *right* child.

Example

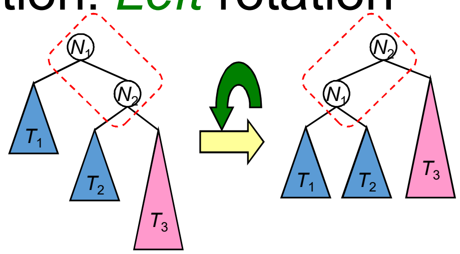
- Do a *right*-*left* rotation after inserting key 12.



Summary of Imbalance

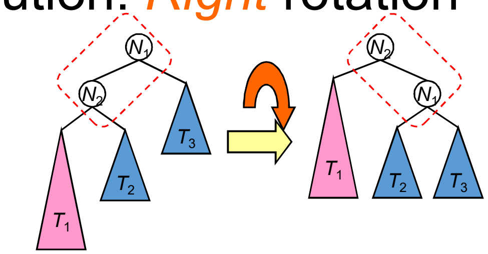
Case 1: insertion to *right* subtree of *right* child

Solution: *Left* rotation



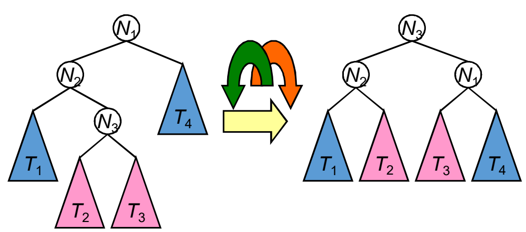
Case 2: insertion to *left* subtree of *left* child

Solution: *Right* rotation



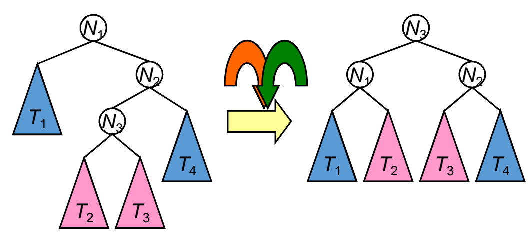
Case 3: insertion to *right* subtree of *left* child

Solution: *Left*-*right* rotation



Case 4: insertion to *left* subtree of *right* child

Solution: *Right*-*left* rotation



Steps of Fixing Imbalance

- Find the lowest node that is imbalanced, i.e., the lowest node that has a balanced factor <= -2 or >= 2, after an insertion
- Determine which case
- Perform the rotation accordingly

AVL Trees

Observaton

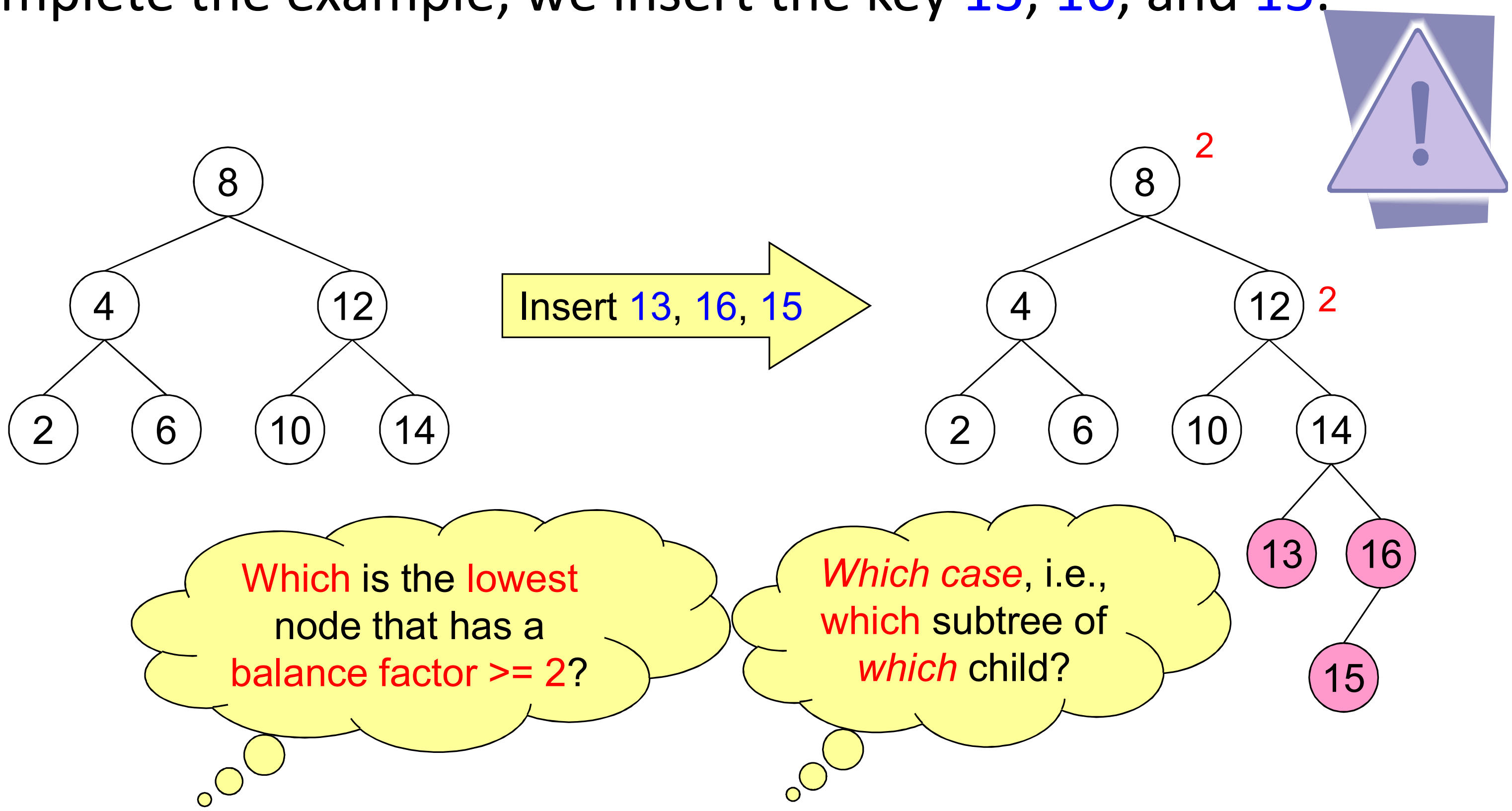
An imbalance caused by insertion can *always* be fixed by performing *one* operation, either a *single* or *double* rotation.

- A balanced tree maintained by single or double rotations is called an ***AVL tree***.

AVL stands for Adel'son-Vel'skii and Landis, the two Russian mathematicans who invented this algorithm.

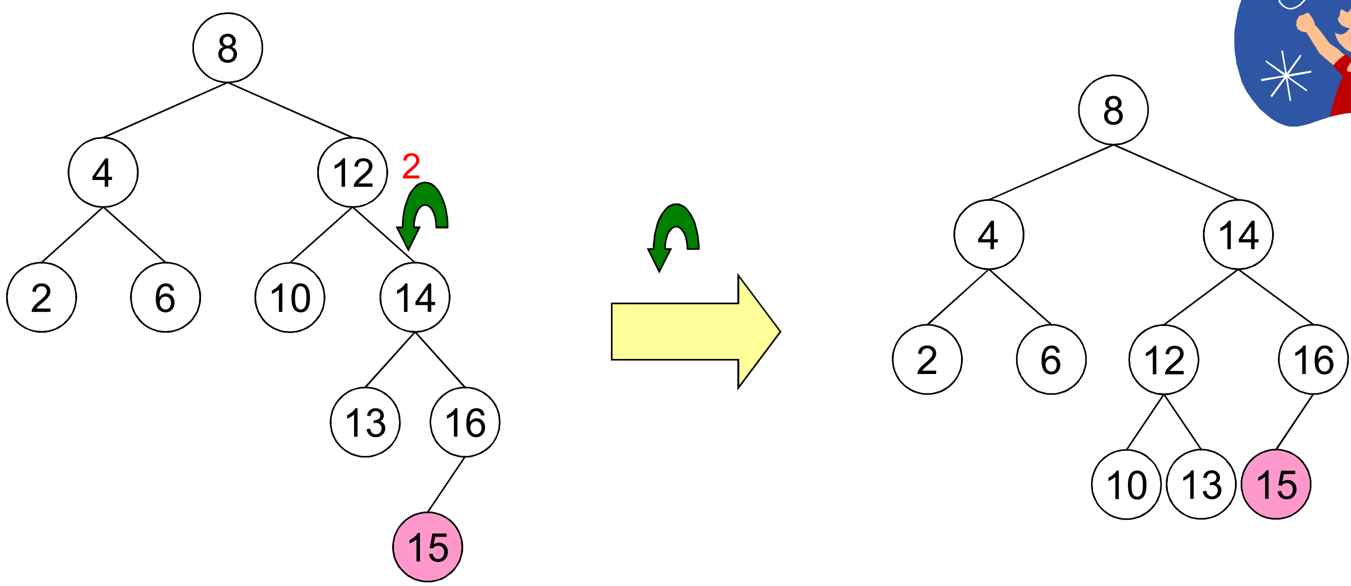
Example

- To complete the example, we insert the key 13, 16, and 15.

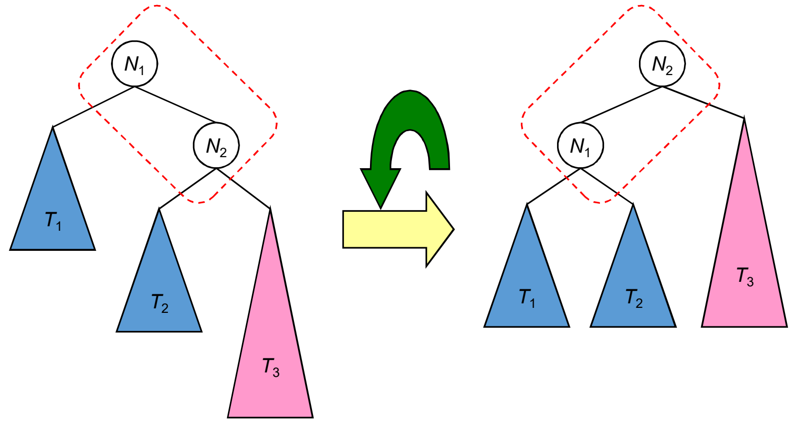


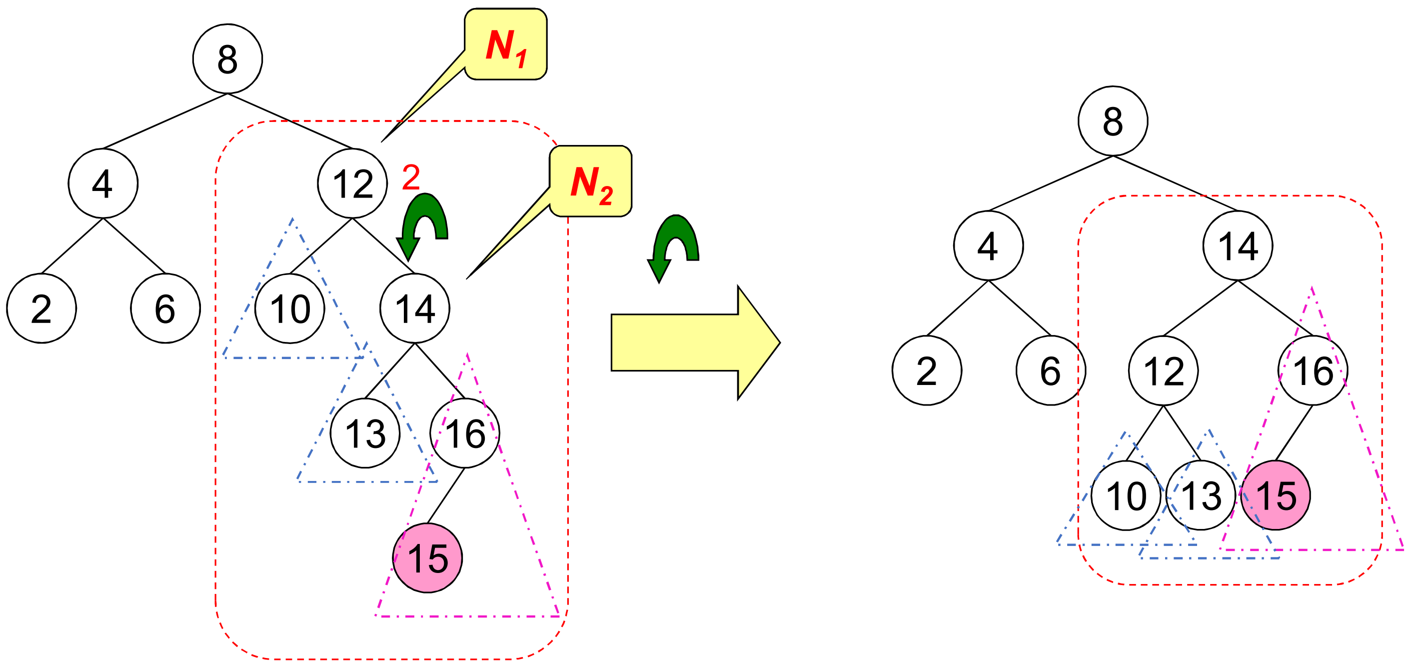
Left Rotation

- Key 15 is inserted to the *right* subtree of the *right* child (of node 12). A *left* rotation is required. (Case 1)



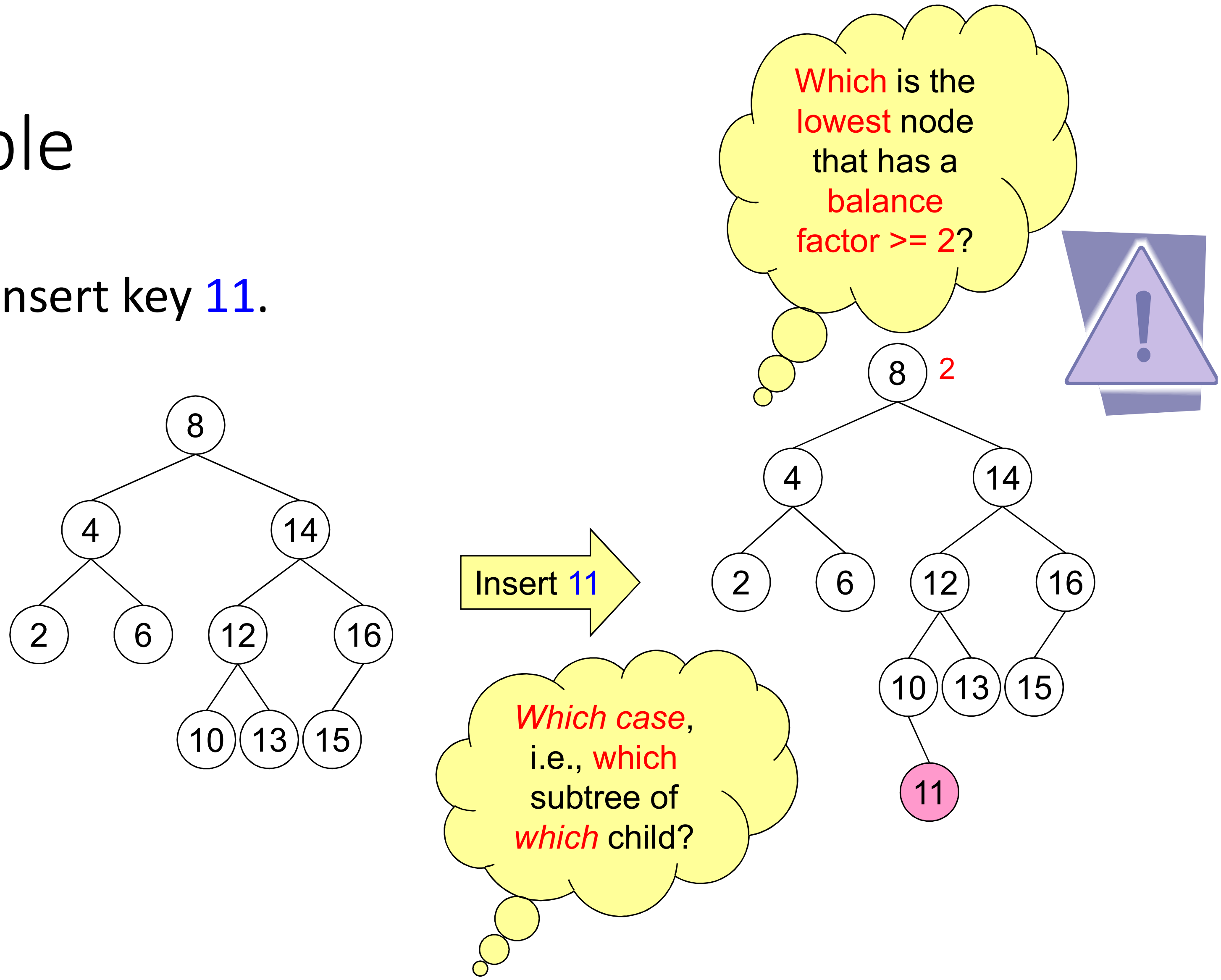
Left Rotation





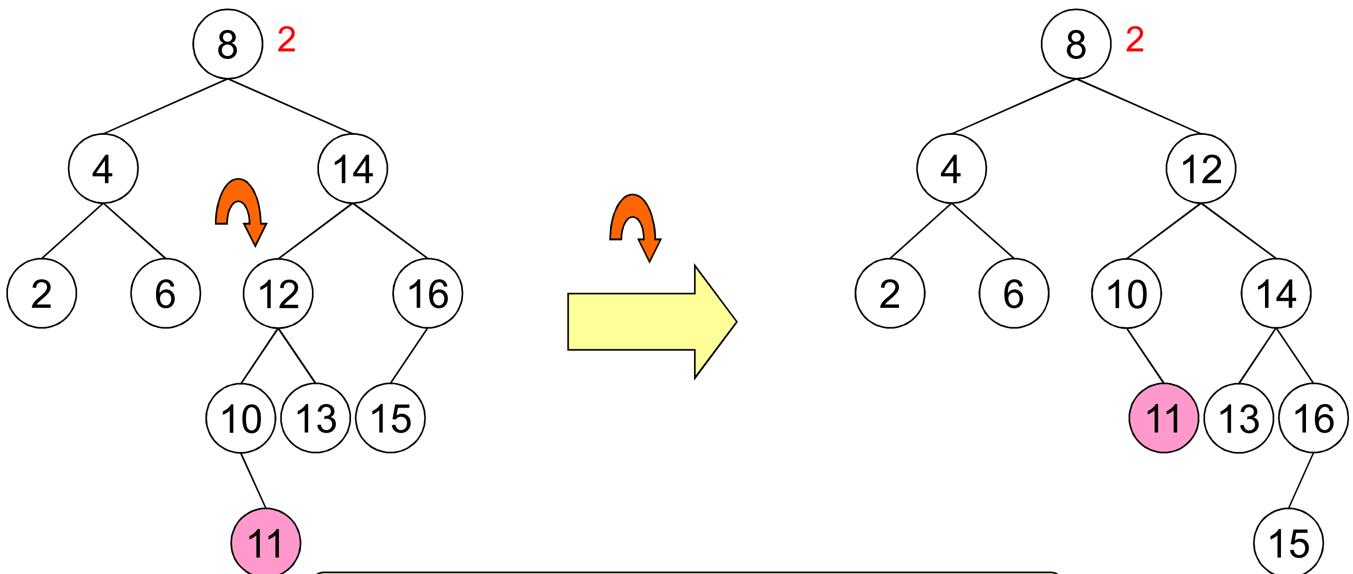
Example

- Finally, insert key 11.



Right-Left Rotation

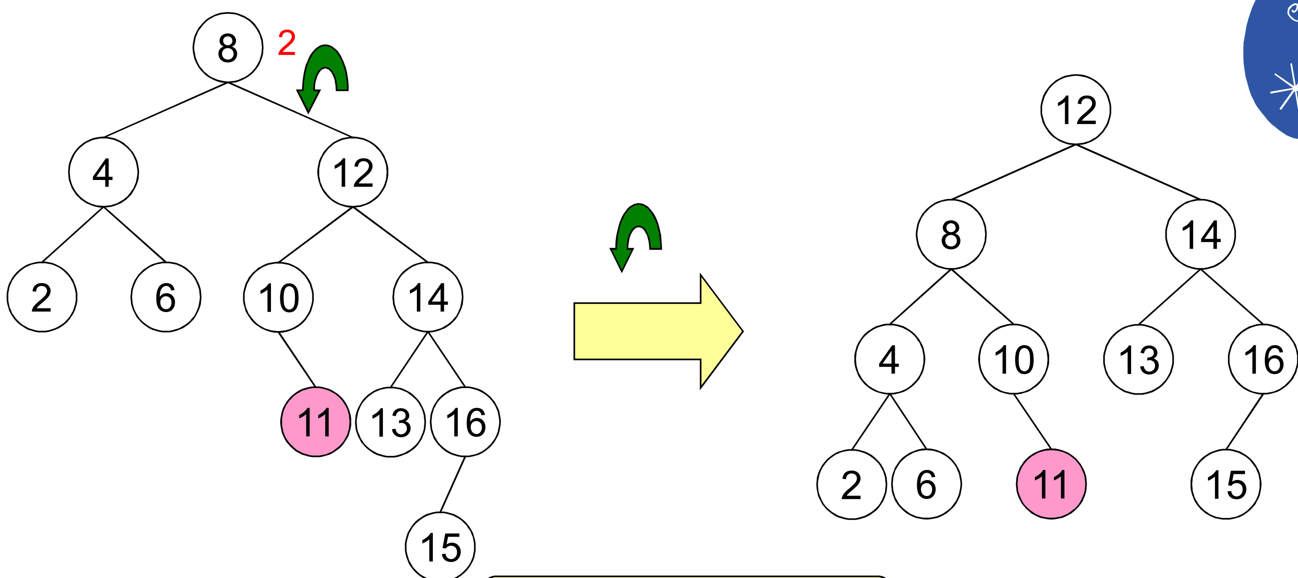
- Key 11 is inserted to the *left* subtree of the *right* child. A *right*-*left* rotation is required. (Case 4)



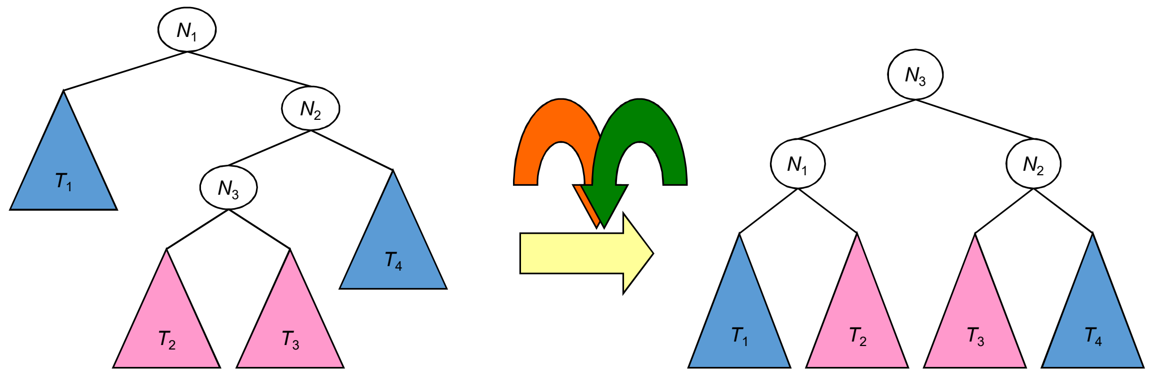
Step 1: *right* rotation on *right* subtree

Right-Left Rotation

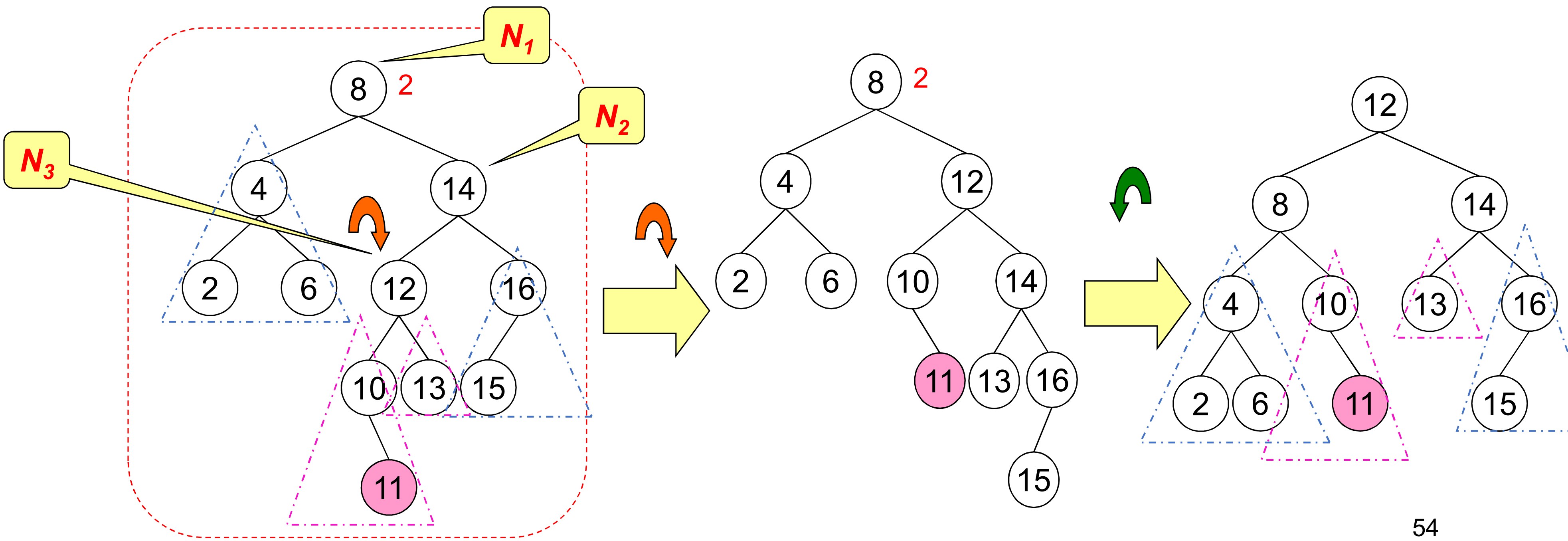
- Key 11 is inserted to the *left* subtree of the *right* child. A *right*-*left* rotation is required. (Case 4)



Step 2: *left* rotation

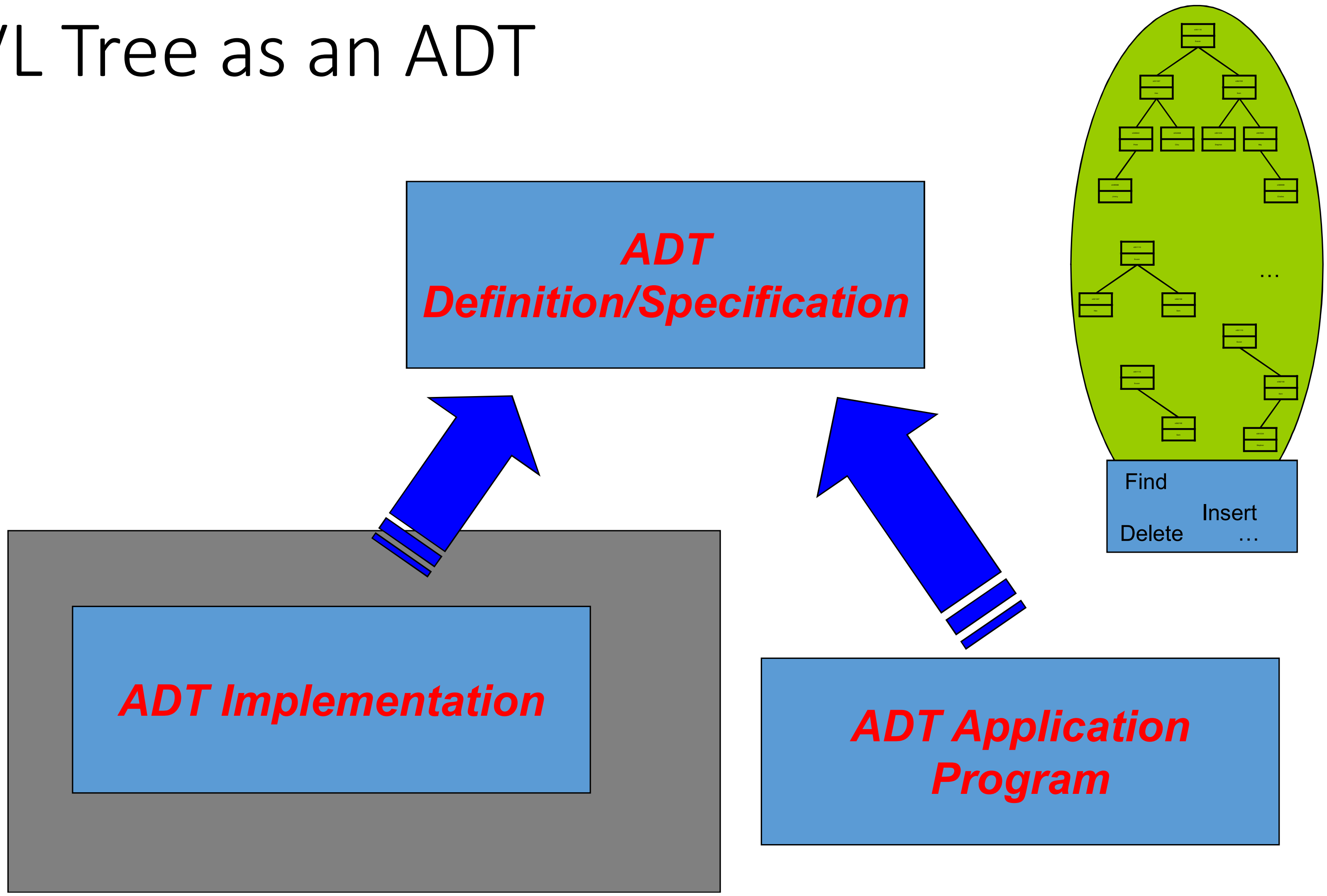


N_1
N_2
N_3
T_1
T_2
T_3
T_4



N_1
N_2
N_3
8
2
4
14
2
6
12
16
10
13
15
11

AVL Tree as an ADT



Reusing the BST ADT Interface

- AVL Trees can be seen as an extension to BSTs. We can *reuse* the interface of the BST ADT.

bst.h

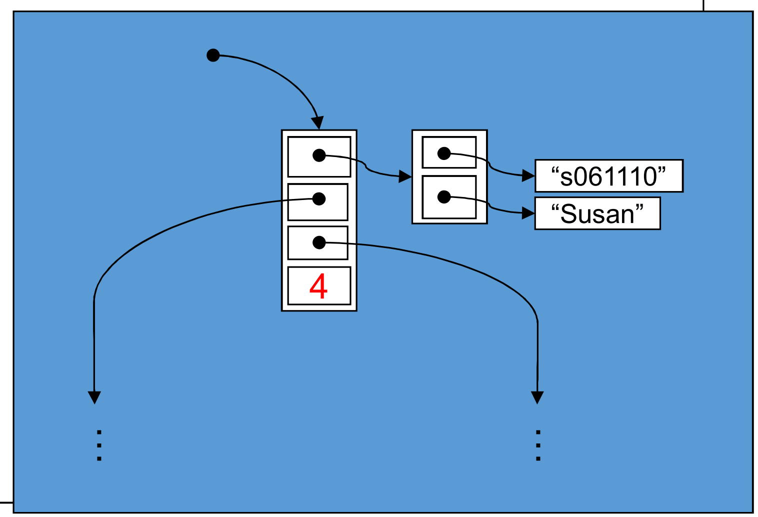
```
#include "treenode.h"

typedef struct bstCDT *bstADT;

bstADT EmptyBST();
int BSTIsEmpty(bstADT t);

bstADT MakeBST(treeNodeADT root, bstADT left, bstADT right);

treeNodeADT Root(bstADT t);
bstADT LeftSubtree(bstADT t);
bstADT RightSubtree(bstADT t);

treeNodeADT FindNode(bstADT t, char *key);
bstADT InsertNode(bstADT t, treeNodeADT n);
bstADT DeleteNode(bstADT t, char *key);
```

Implementation of AVL Trees

- We modify the representation to explicitly store the *height* of a BST.

avltree.c

```
#include <stdio.h>
#include <stdlib.h>
#indlude <string.h>
#include "bst.h"

struct bstCDT {
   treeNodeADT root;
   bstADT left;
   bstADT right;
   int height;
};
```



avltree.c (continue)

```
int Height(bstADT t) {   // a private function
   if (BSTIsEmpty(t))
      return -1;
   else
      return t->height;
}

bstADT MakeBST(treeNodeADT root, bstADT left, bstADT right) {
   bstADT t;
   int lh, rh;
   t = (bstADT)malloc(sizeof(struct bstCDT));
   t->root = root;
   t->left = left;
   t->right = right;
   lh = Height(left);
   rh = Height(right);
   if (lh < rh)
      t->height = rh + 1;
   else
      t->height = lh + 1;
   return t;
}
```

Implementation of AVL Trees

- The functions **`EmptyBST`**, **`BSTIsEmpty`**, **`Root`**, **`LeftSubtree`**, **`RightSubtree`**, and **`FindNode`** are exactly the *same* as those in bst.c.

`avltree.c` (continue)

```
bstADT EmptyBST() { … }

int BSTIsEmpty(bstADT t) { … }

treeNodeADT Root(bstADT t) { … }

bstADT LeftSubtree(bstADT t) { … }

bstADT RightSubtree(bstADT t) { … }

treeNodeADT FindNode(bstADT t, char *key) { … }
```

Rotation Functions

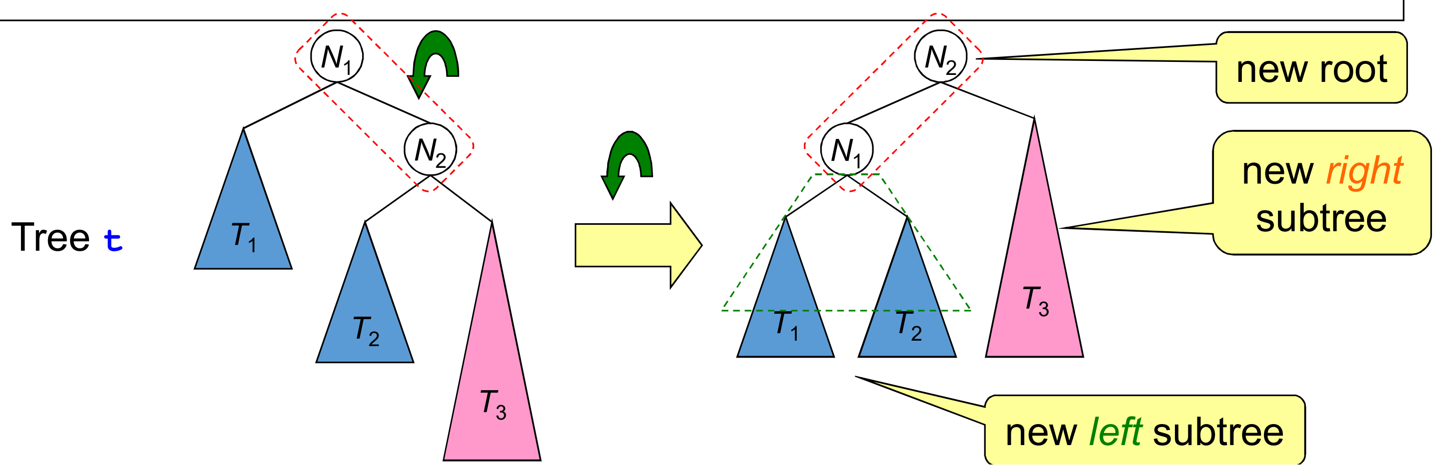
- Next, we have to write (private) functions for the rotation operations.
  - Single Rotations
    - `bstADT RotateLeft(bstADT t)`
    - `bstADT RotateRight(bstADT t)`
  - Double Rotations
    - `bstADT RotateLeftRight(bstADT t)`
    - `bstADT RotateRightLeft(bstADT t)`

Single Rotation: **RotateLeft**

avltree.c (continue)

```
bstADT RotateLeft(bstADT t) {
   treeNodeADT newRoot;
   bstADT newLeft, newRight;
   newRoot = Root(RightSubtree(t));
   newLeft = MakeBST(Root(t), LeftSubtree(t),
                     LeftSubtree(RightSubtree(t)));
   newRight = RightSubtree(RightSubtree(t));
   return MakeBST(newRoot, newLeft, newRight);
}
```

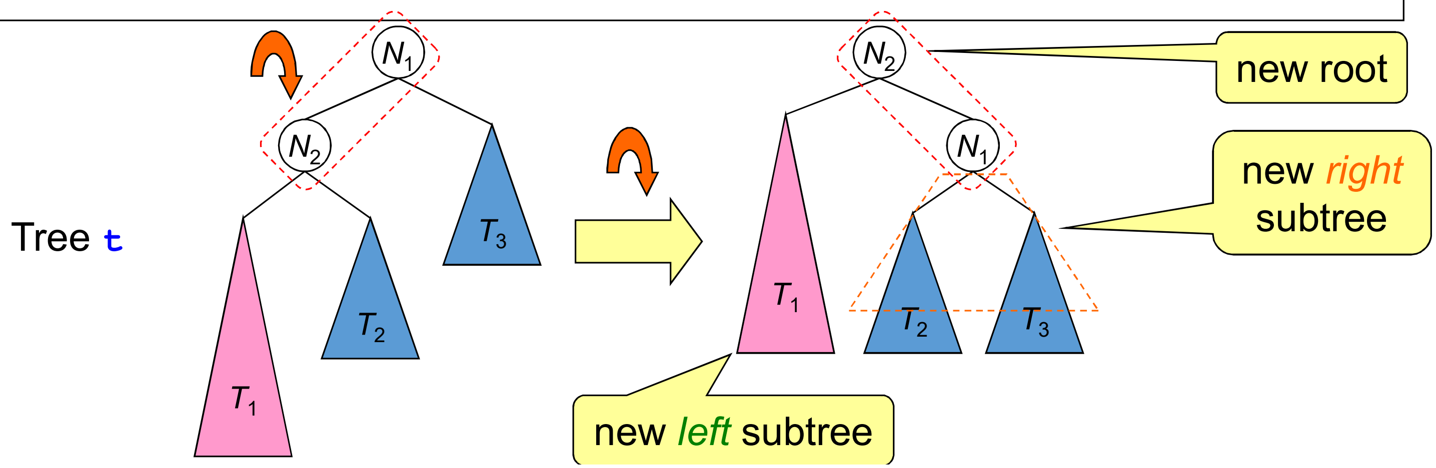
N_2 T_1 T_2 N_1 T_3



Single Rotation: **RotateRight**

avltree.c (continue)

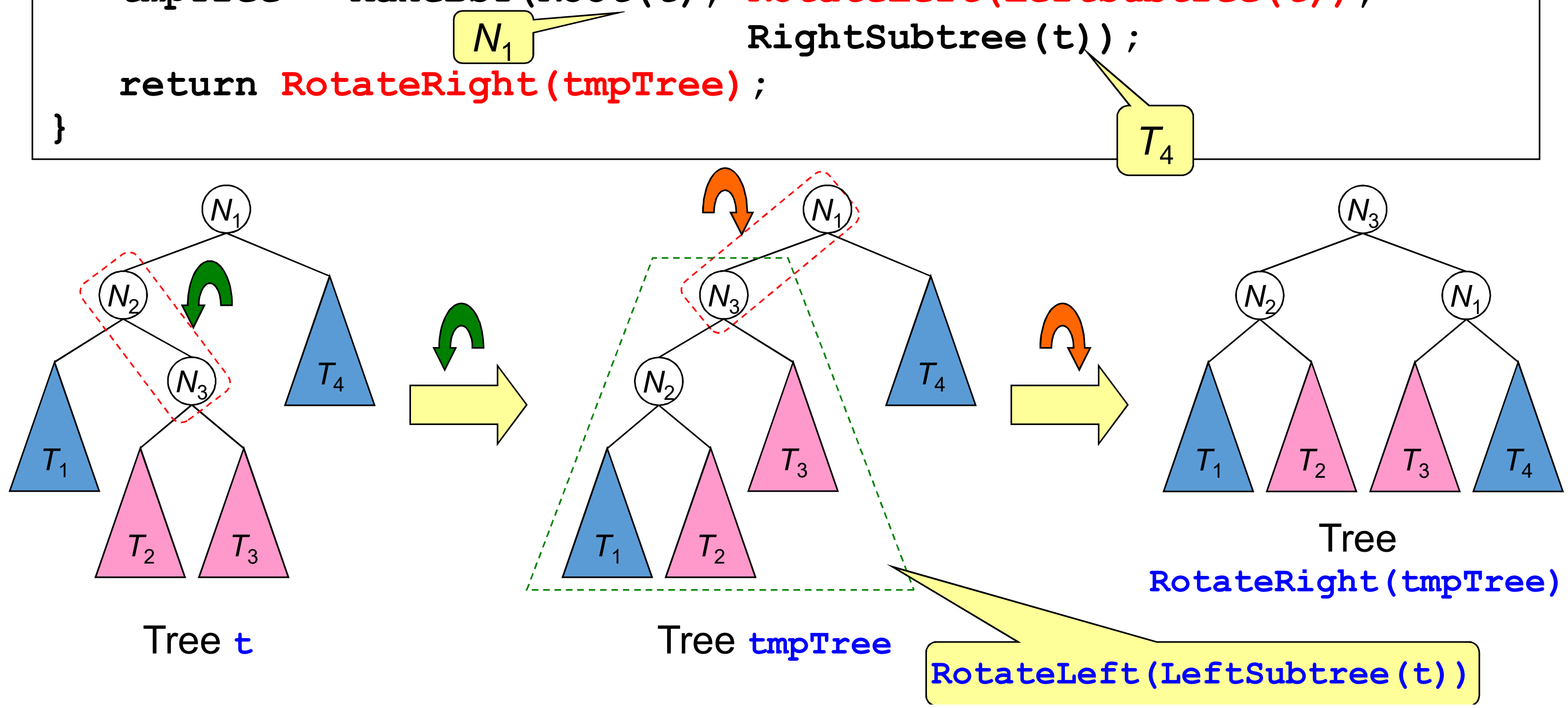
```
bstADT RotateRight(bstADT t) {

        (Entirely symmetrical to RotateLeft.)
        (Leave as an exercise.)

}
```



Double Rotation: **RotateLeftRight**

avltree.c (continue)

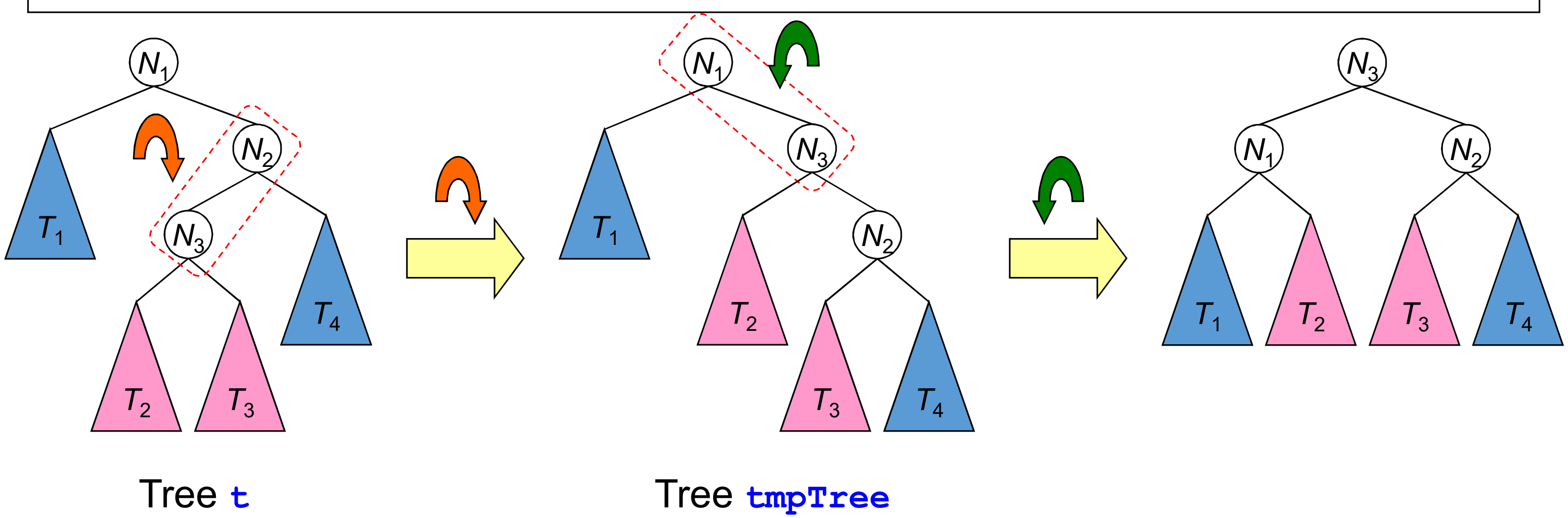
```
bstADT RotateLeftRight(bstADT t) {
   bstADT tmpTree;
   tmpTree = MakeBST(Root(t), RotateLeft(LeftSubtree(t)),
                              RightSubtree(t));
   return RotateRight(tmpTree);
}
```



Double Rotation: `RotateRightLeft`

`avltree.c` (continue)

```
bstADT RotateRightLeft(bstADT t) {

    (Entirely symmetrical to RotateLeftRight.)
    (Leave as an exercise.)

}
```



AVL Node Insertion

- All the rotation functions are very easy and logical to understand.
- They all execute in O(1) time.

- With the rotation functions, we are now ready to implement the **`InsertNode`** function, which is itself also very easy and logical.

AVL Node Insertion

avltree.c (continue)

```
bstADT InsertNode(bstADT t, treeNodeADT n) {
   int sign;
   bstADT tmpTree;
   if (BSTIsEmpty(t))
      return MakeBST(n, EmptyBST(), EmptyBST());
   sign = strcmp(GetNodeKey(n), GetNodeKey(Root(t)));
   if (sign == 0)
      return MakeBST(n, LeftSubtree(t), RightSubtree(t));
   else if (sign < 0) {   // insert to left subtree

                  (See page 67)

   } else {   // insert to right subtree

                  (See page 70)

   }
}
```

Same cases as in bst.c.

avltree.c (continue)

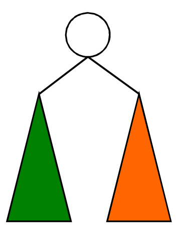
```
…
if (sign == 0)
   …
else if (sign < 0) {    // insert to left subtree
   tmpTree = MakeBST(Root(t), InsertNode(LeftSubtree(t), n),
                              RightSubtree(t));
   if (Height(LeftSubtree(tmpTree))
       - Height(RightSubtree(tmpTree)) == 2) {
      if (strcmp(GetNodeKey(n),
                 GetNodeKey(Root(LeftSubtree(tmpTree)))) < 0)
         return RotateRight(tmpTree);    // case 2
      else
         return RotateLeftRight(tmpTree);    // case 3
   } else
      return tmpTree;
} else {   // insert to right subtree
   …
}
```

Check if the insertion causes imbalance.

tmpTree is balanced, simply return it.

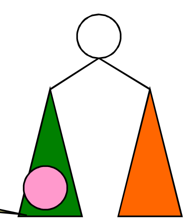
If **tmpTree** is unbalanced, then determine whether it is case 2 or case 3.

Tree **t**

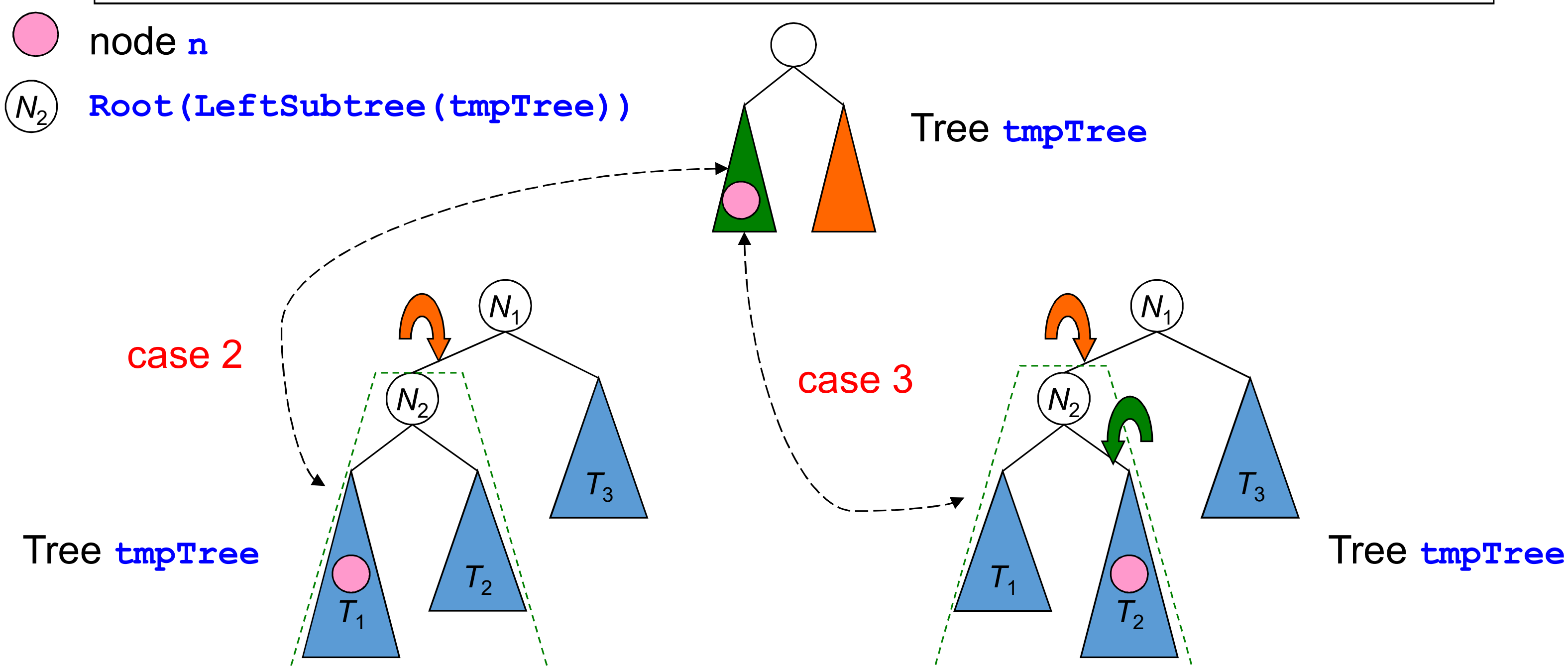


Tree **tmpTree**

node **n**



```
if (Height(LeftSubtree(tmpTree))
     - Height(RightSubtree(tmpTree)) == 2) {
   if (strcmp(GetNodeKey(n),
              GetNodeKey(Root(LeftSubtree(tmpTree)))) < 0)
      return RotateRight(tmpTree);    // case 2
   else
      return RotateLeftRight(tmpTree);    // case 3
} else
   return tmpTree;
```



AVL Node Insertion

avltree.c (continue)

```
bstADT InsertNode(bstADT t, treeNodeADT n) {
   int sign;
   bstADT tmpTree;
   if (BSTIsEmpty(t))
      return MakeBST(n, EmptyBST(), EmptyBST());
   sign = strcmp(GetNodeKey(n), GetNodeKey(Root(t)));
   if (sign == 0)
      return MakeBST(n, LeftSubtree(t), RightSubtree(t));
   else if (sign < 0) {   // insert to left subtree
```

(Shown in page 67)

```
   } else {   // insert to right subtree
```

(Shown in page 70)

```
   }
}
```

avltree.c (continue)

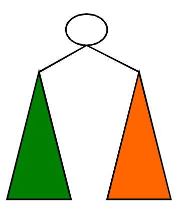
```
…
if (sign == 0)
   …
else if (sign < 0) {    // insert to left subtree
   …
} else {    // insert to right subtree
   tmpTree = MakeBST(Root(t), LeftSubtree(t),
                              InsertNode(RightSubtree(t), n));
   if (Height(RightSubtree(tmpTree))
         - Height(LeftSubtree(tmpTree)) == 2) {
      if (strcmp(GetNodeKey(n),
                  GetNodeKey(Root(RightSubtree(tmpTree)))) > 0)
         return RotateLeft(tmpTree);    // case 1
      else
         return RotateRightLeft(tmpTree);    // case 4
   } else
      return tmpTree;
}
```

Check if the insertion causes imbalance.

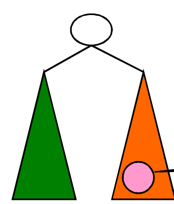
tmpTree is balanced, simply return it.

If tmpTree is unbalanced, then determine whether it is case 1 or case 4.

Tree t

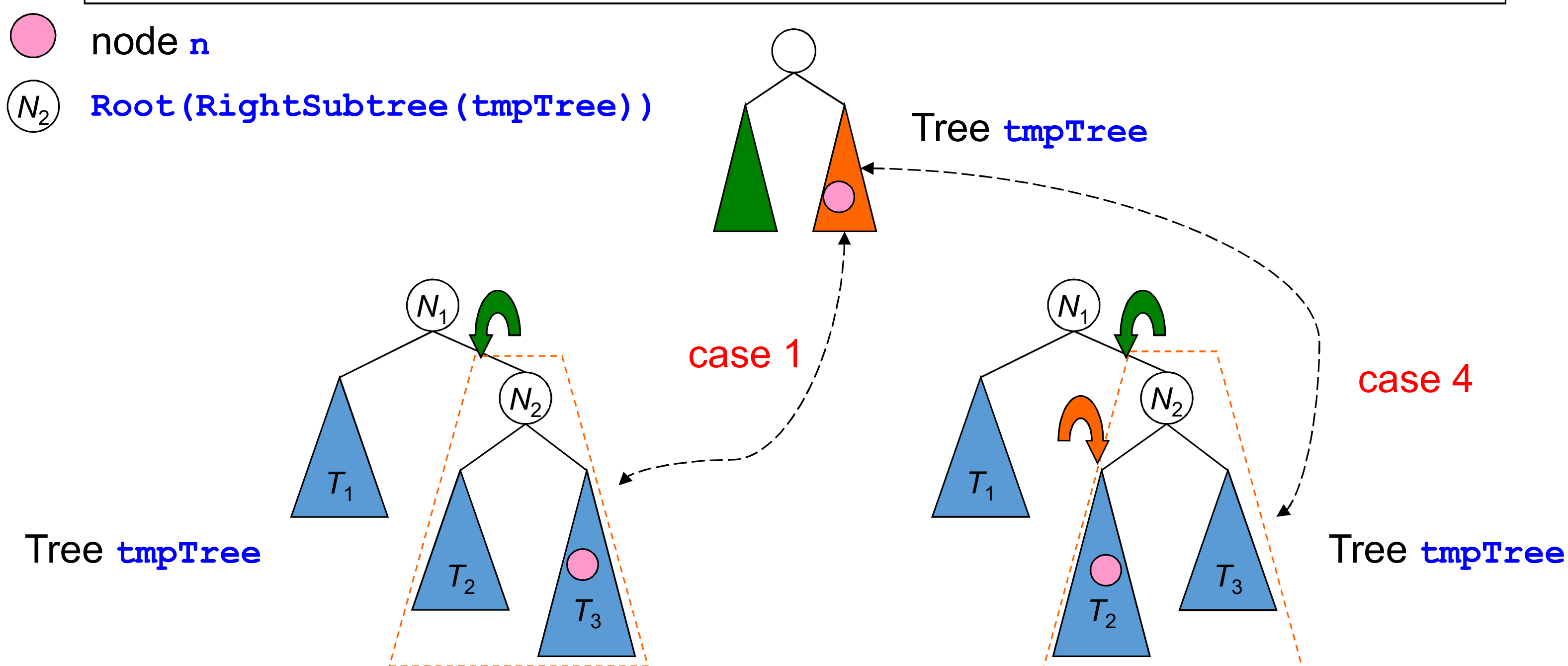


Tree tmpTree



node n

```
if (Height(RightSubtree(tmpTree))
      - Height(LeftSubtree(tmpTree)) == 2) {
   if (strcmp(GetNodeKey(n),
                 GetNodeKey(Root(RightSubtree(tmpTree)))) > 0)
      return RotateLeft(tmpTree);    // case 1
   else
      return RotateRightLeft(tmpTree);    // case 4
} else
   return tmpTree;
```

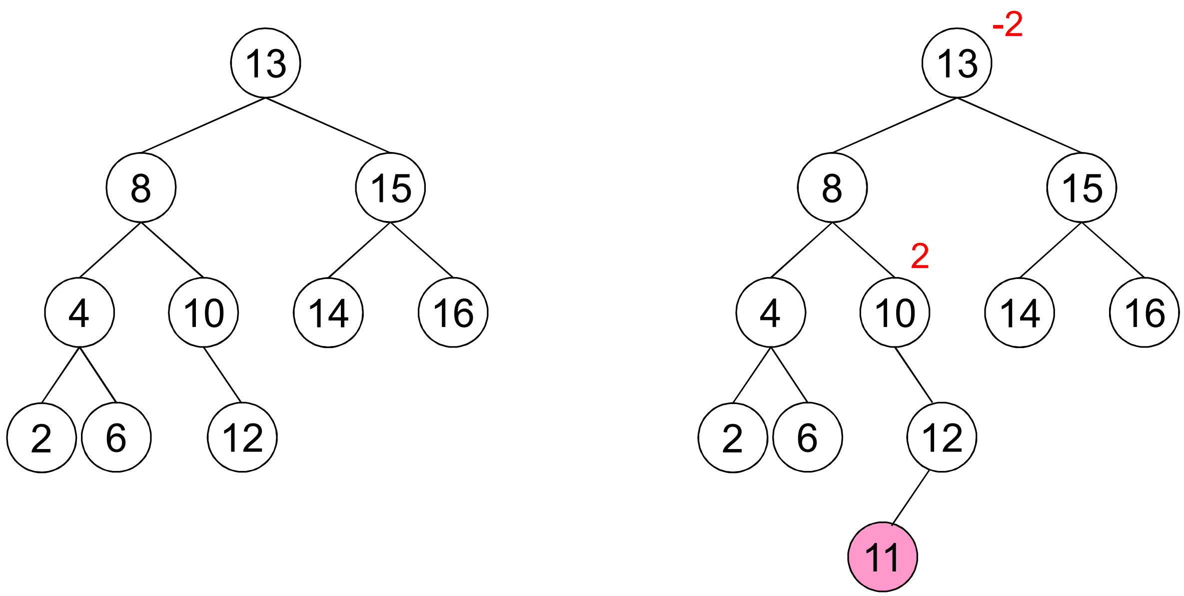


Why lowest node?

- Find the lowest node that is imbalanced, i.e., the lowest node that has a balanced factor <= -2 or >= 2, after an insertion
- Can we perform the rotation at any node that has a balanced factor <= -2 or >= 2, after an insertion?

Consider this case

- Let's insert 11, and perform the rotation at node 13 and 10, respectively
- Do we get a balanced tree for both cases?

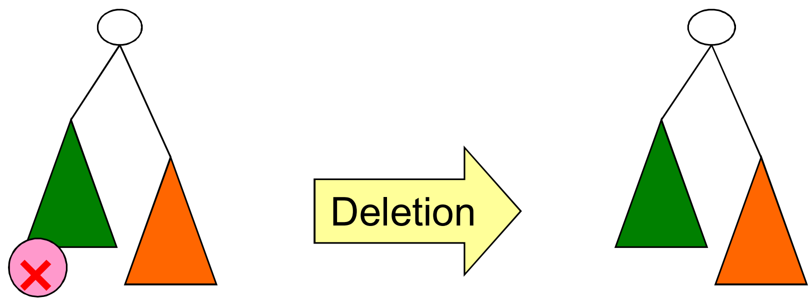


Correctness of **`InsertNode`**

- Does **`InsertNode`** find the lowest node that is imbalanced, i.e., the lowest node that has a balanced factor <= -2 or >= 2, after an insertion?

Deletion in AVL Trees

- Deleting a node from an AVL tree can also cause imbalance, which has to be fixed also.



- The deletion algorithm will not be covered here.