CMPC-5014Y Data Structures and Algorithms

Geoff McKeown

Binary Search Trees

A Structure for Efficient Searching

Lecture Objectives

- To introduce a Binary Search Tree (BST) Abstract Data Type (ADT).
- To present a Java implementation.
- To demonstrate how operations called "rotations" can be used to keep a BST "balanced" when insertions/deletions are made to the tree.

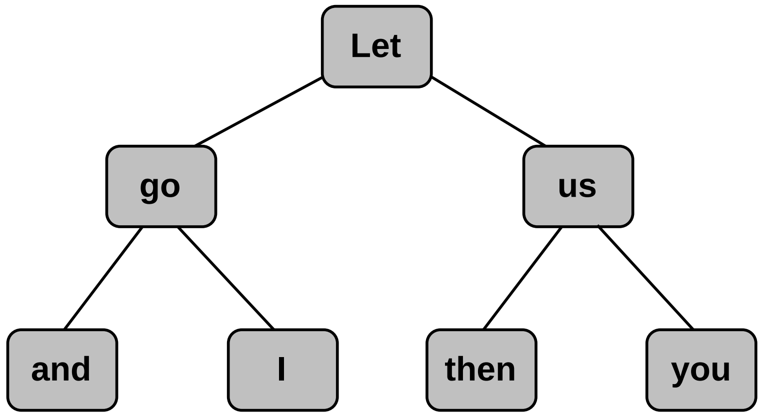
Binary Search Trees

Definition A *binary search tree (bst)*, t, is a binary tree; either it is empty or each node in the tree has an associated identifier, or *key*, from a totally ordered set of keys, such that:

(i) the keys of all nodes in the left sub-tree of t are less than the key of the root of t,

(ii) the keys of all nodes in the right sub-tree of t are greater than the key of the root of t,

(iii) the left and right sub-trees of t are themselves binary search trees.



Exercise
Prove that an inorder traversal of a bst results in an output in *lexicographic* order with respect to the keys.
[Hint: Use induction on the number of nodes in the bst.]

Java Implementation of Totally Ordered Sets

◇ The elements in a totally ordered set, K, are *comparable*

▷ if $a, b \in K$ and $a \neq b$, then

either $\quad a$ "is less than" b

or $\quad b$ "is less than" a.

◇ Java provides a standard interface `Comparable<E>`. This has the query method

```
public int compareTo( E rhs )
```

such that

`this` less than `rhs` implies returned `int` < 0

`this` equal `rhs` implies returned `int` $== 0$

`this` greater than `rhs` implies returned `int` > 0

Java Interface

```
package binarytree;

public interface ADT_BST
{
    public boolean isMember( Comparable k );

    public void insert( Comparable k );

    public void delete( Comparable k );

    public Comparable getMinElt( );
}
```

Implementation of `ADT_BST`

```
package binarytree;

public class BST extends BinaryTree implements ADT_BST{
// Constructor

    public BST( )
        { root = null; }

    public void insert( Comparable k )
        { root = insertTree( k, root ); }

    private TreeNode insertTree( Comparable k, TreeNode t )
        // Defined below

    public boolean isMember( Comparable k )
        { return isMemberTree( k, root ); }

    private boolean isMemberTree( Comparable k, TreeNode t )
        // Defined below

    public void delete( Comparable k )
        // Exercise:  Give a recursive definition }

    public Comparable getMinElt( )
        // Exercise:  Give a recursive definition }
}
```

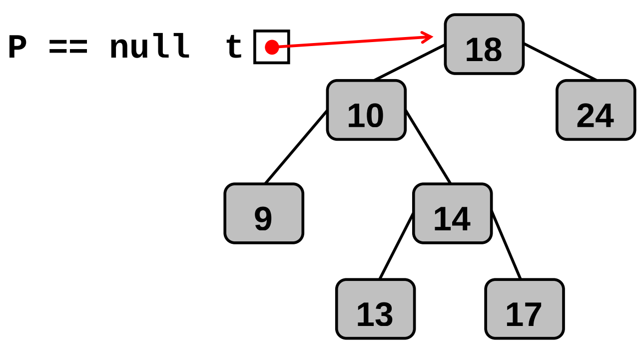
```
private TreeNode insertTree( Comparable k, TreeNode t )
{
    if ( t == null )
        { t = new TreeNode( k ); }
    else{
        int comp = k.compareTo( t.elt );
        if ( comp == 0 )
            throw new IllegalArgumentException
            ( "Given element is already in the tree." );
        else if (comp < 0) // go to the left
            t.left = insertTree( k, t.left );
        else // go to the right
            t.right = insertTree( k, t.right );
    return t;
}
```

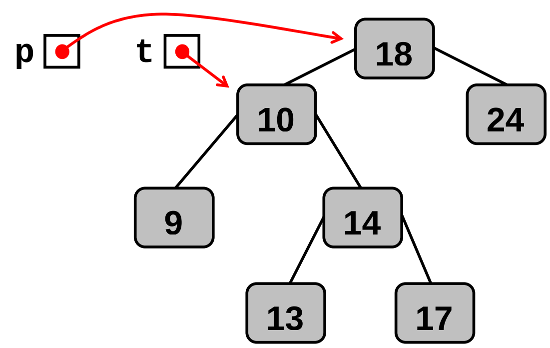
```java
private boolean isMemberTree( Comparable k, TreeNode t )
{
    if ( t == null )
        return false;
    else{
        int comp = k.compareTo( t.elt );
        if ( comp == 0 )
            return true;
        else if (comp < 0)
            return isMemberTree( k, t.left );
        else
            return isMemberTree( k, t.right );
}
```

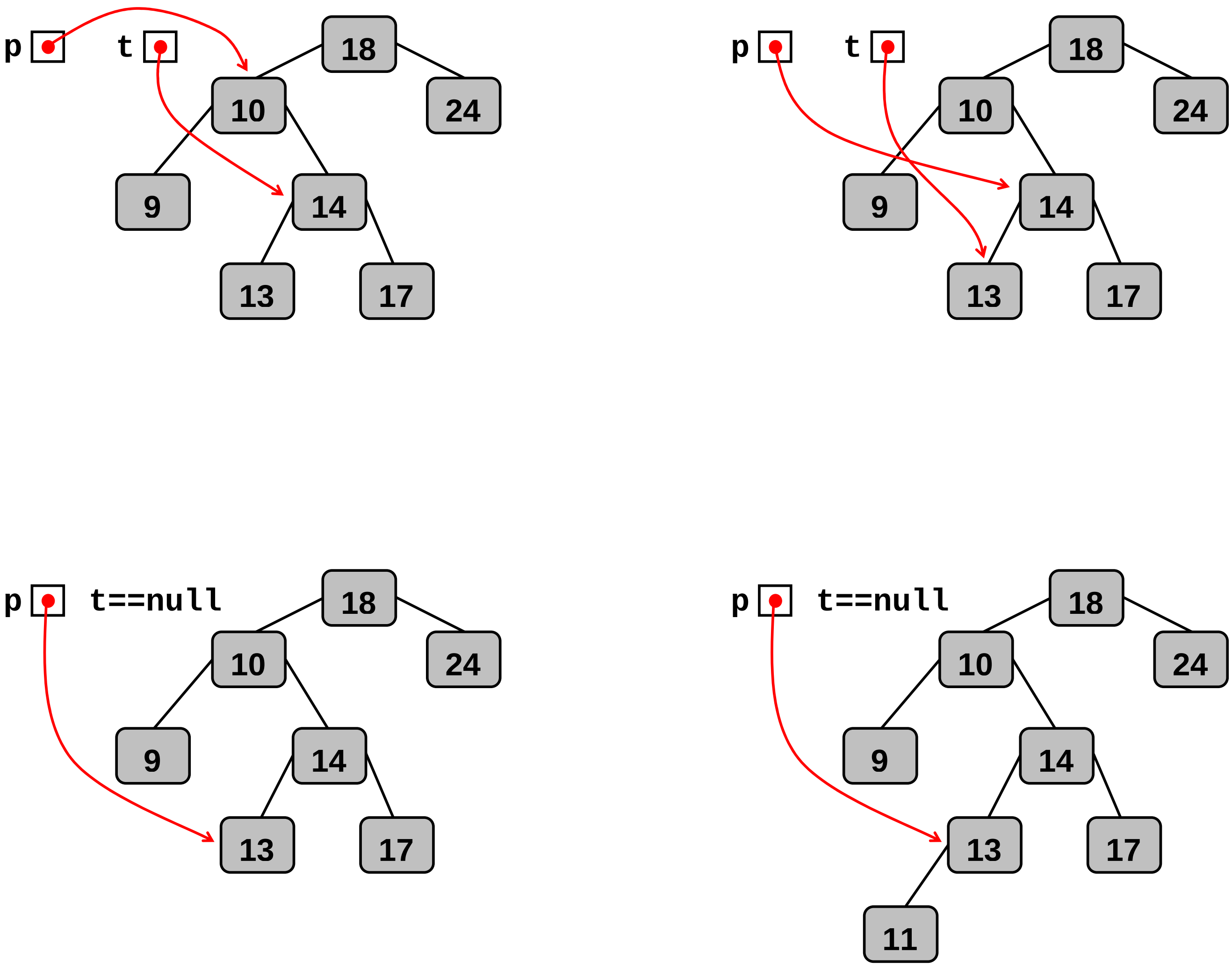
Non-recursive implementation of `insert`

Basic idea:

- Search the bst for the given key:
  - use two reference variables, `t` and `p`, say:
  - `t` references the current `TreeNode`;
  - `p` references the parent node of `t`).
- If the given key is found, do nothing;
- otherwise, `p` references the `TreeNode` which will be the parent of the incoming node:
  - update `p.left` or `p.right` as appropriate.







Non-recursive implementation of `delete`

Basic idea:

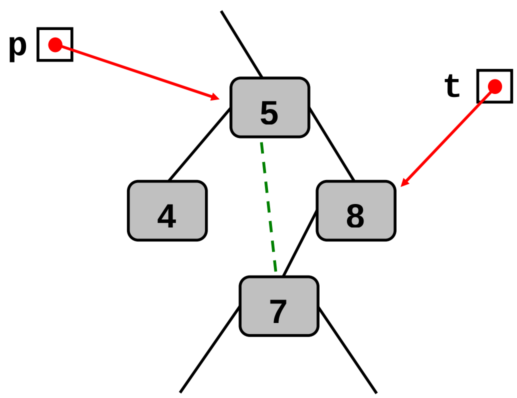
- As for `insert`, first locate node, `t`, in tree, if present, and its parent node, `p`.
- If the key to be deleted is present in the bst, there are 3 cases to consider:

Node containing key has no children:

- ▷ just set the `left` or `right` reference field, as appropriate, of parent node, `p`, to `null`

Node to be deleted has a single child:

- ◇ set the `left` or `right` reference field, as appropriate, of parent node, `p`, to either `t.left` or `t.right` as the case may be, e.g.



`p.right = t.left`

Node to be deleted has two children:

◇ the inorder successor of `t` must take its place;

◇ the inorder successor, `s`, cannot have a left child (*WHY?*).

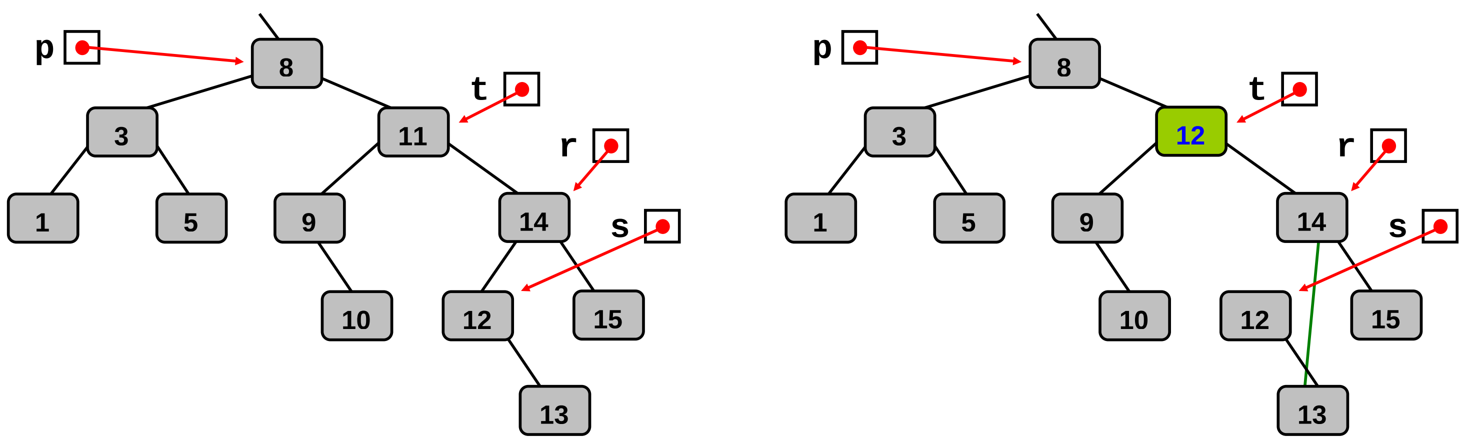
◇ right child of `s` can be moved up to take its place.

◇ Suppose the element 11 is to be deleted from the following bst:

▷ t denotes the node (i.e. references the node) containing 11

▷ p denotes the parent of t

◇ r denotes the parent of the node, s, containing the inorder successor of the element to be deleted.




```
r.left = s.right; t.elt = s.elt;
```

```
r = t;
s = t.right;
q = s.left;
// find inorder successor of the node to be deleted
while( q != null ){
   r = s;
   s = q;
   q = q.left;
}
// update the tree
if ( r != t )
   r.left = s.right;
else
   r.right = s.right;
t.elt = s.elt;
```

Balanced Binary Search Trees

- ◇ Searching in a BST is $O(\log n)$, where n is the number of nodes, provided the tree is "balanced".
- ◇ In the worst-case, searching in a BST is $\Omega(n)$.
- ◇ Recall: height of a binary tree is the maximum level of its leaves
  - ▷ we define the height of an empty tree to be -1.

Definition A binary tree, t, is said to be *height-balanced* if

either

(i) $t = \emptyset$,

or

(ii) $t \neq \emptyset$ and:

(a) Left(t) and Right(t) are height-balanced;

(b) $|$ height(Left(t)) - height(Right(t)) $| \leq 1$.

AVL Trees

(Adelson-Velskii and Landis)

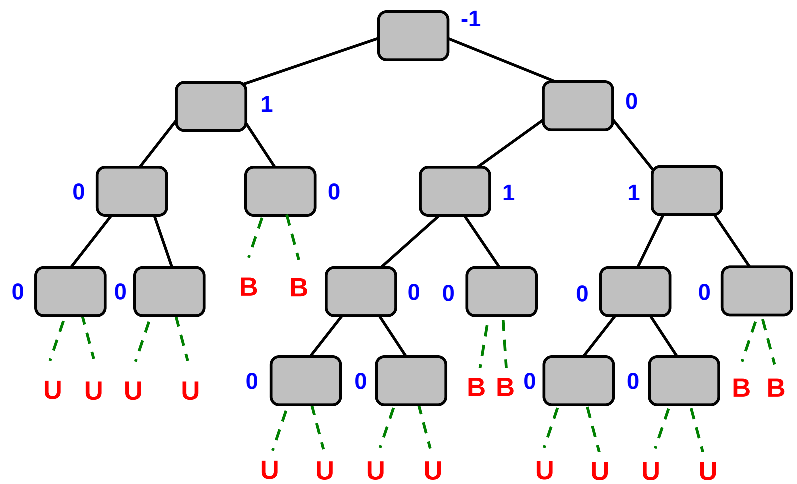
These are height-balanced binary search trees.

Definition balance(t) = height(Left(t)) - height(Right(t)).

- ◇ All of the nodes in the following tree are height-balanced:
  - ▷ using the above definition of balance, we obtain the node balances indicated next to the nodes.
- ◇ The dashed lines indicate the possible insertion points for a new node:

▷ an insertion that would leave the tree balanced is indicated by a 'B'

▷ an insertion that would unbalance the tree is indicated by a 'U'.



CASE: a $+1$ node becomes unbalanced.

◇ Let the balance of a node, a, in a bst be $+1$.

◇ Suppose a is the youngest ancestor to become unbalanced when a new node is inserted in the bst.

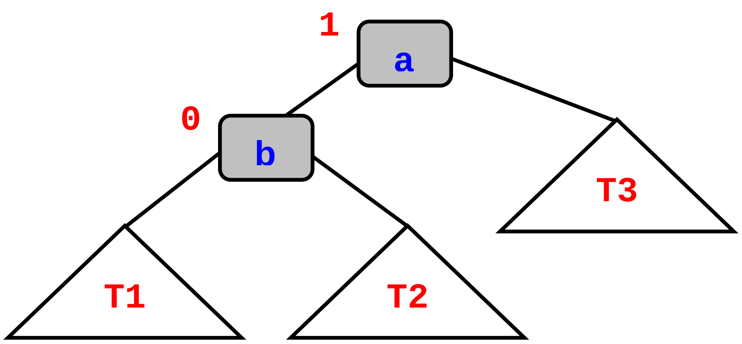
$$\text{balance}(a) = 1 \implies \text{Left}(a) \neq \emptyset.$$

◇ Let b be the left child of a.

Exercise: Deduce that $\text{balance}(b) = 0$.

$$\text{balance}(b) = 0 \implies \text{height}(\text{Left}(b)) = \text{height}(\text{Right}(b))$$
$$= n, \text{ say, before insertion.}$$

$$(\text{height}(b) = n + 1) \wedge (\text{balance}(a) = 1) \implies \text{height}(\text{Right}(a)) = n$$

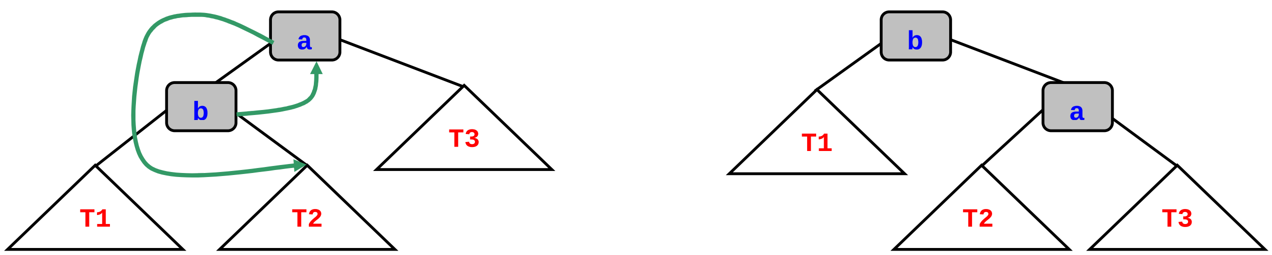


T1, T2 and T3 are each trees of height n

AIM: To transform a bst made unbalanced by an insertion into a balanced bst.

RIGHT ROTATION

The root of the tree swings to the right:

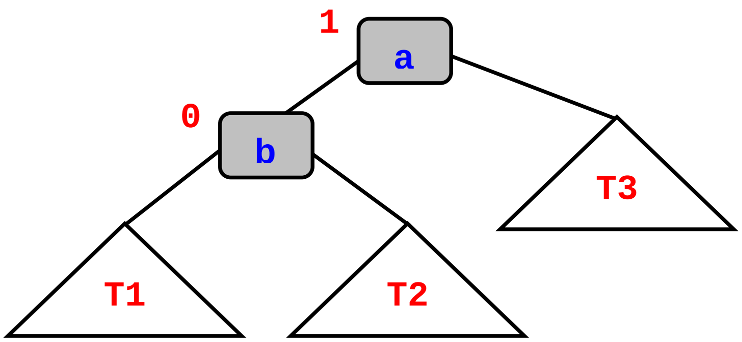


LEFT ROTATION

The root of the tree swings to the left:

Exercise: Deduce that both right and left rotations are bst-preserving.

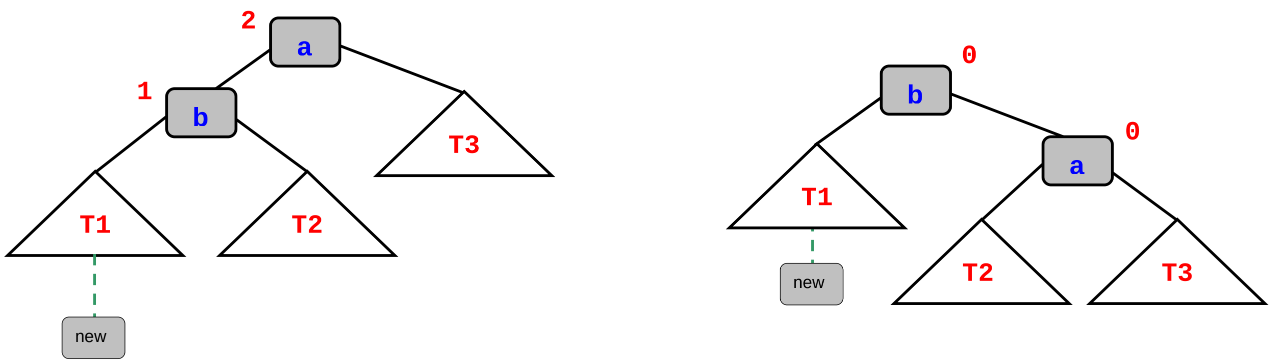
We now return to the case where a +1 node before an insertion becomes unbalanced after the insertion.



T1, T2 and T3 are each trees of height n.
Height of tree rooted at a is n + 2.
Node a is first ancestor to become unbalanced when a node is inserted in its left sub-tree.

(i) New node is inserted into the left sub-tree of b

A single right rotation re-balances the tree:



(ii) *new node is inserted into the right sub-tree of b:*

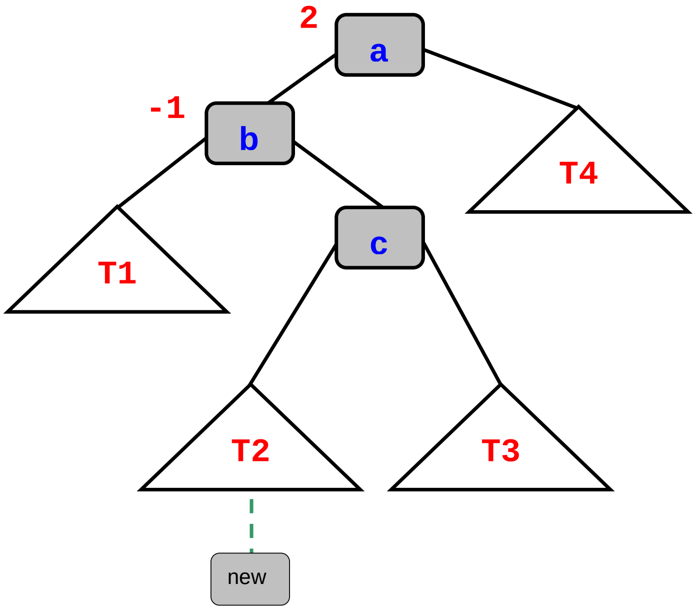
there are 3 sub-cases to consider here:-

(a) new node becomes the right child of b;

(b) new node is inserted into the left sub-tree of the right child of b;

(c) new node is inserted into the right sub-tree of the right child of b.

Illustrate with case (b).



T1 and T4 are each trees of height n.

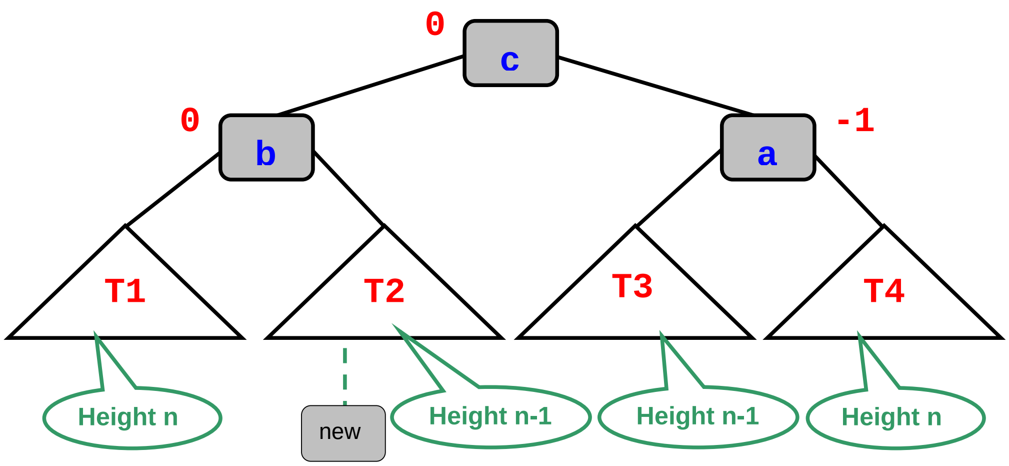
T2 and T3 are each trees of height n-1.

Node a is first ancestor to become unbalanced when a node is inserted in its left sub-tree.

New node is inserted in the left sub-tree of the right child of b.

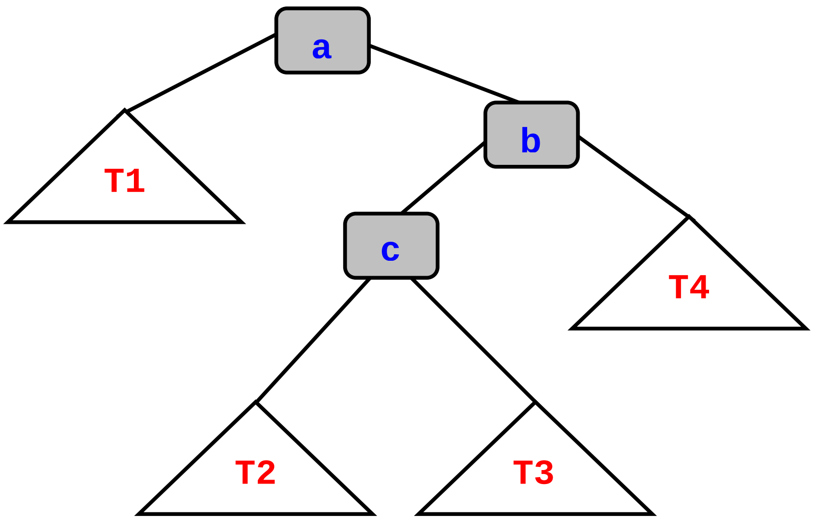
◇ Do a *double rotation*

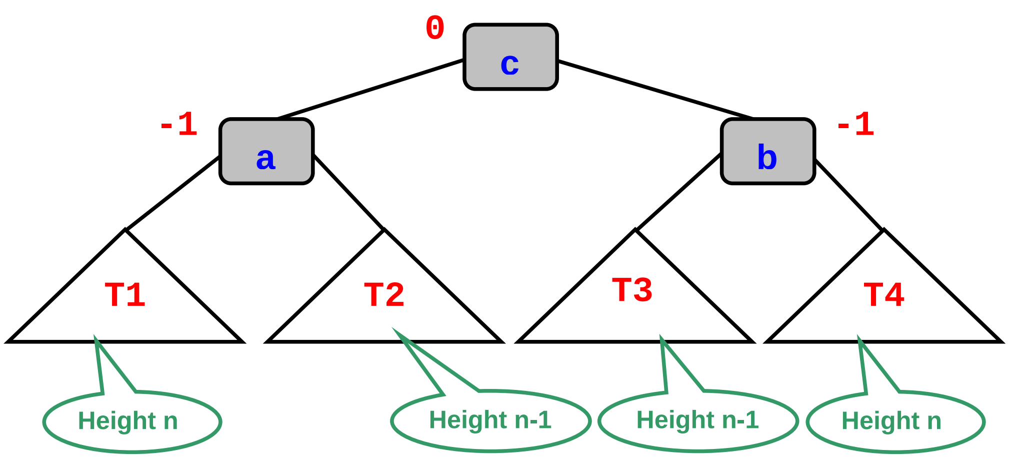
◇ This is equivalent to doing a (single) *left rotation* on *b* followed by a *right rotation* on *a*.



◇ There is a second type of double rotation:

◇ do a (single) right rotation on b followed by a left rotation on a:





FACT: Any previously balanced bst made unbalanced by a single insertion (or deletion) can be re-balanced by performing either a single rotation or a double rotation.