

# Height Balance: AVL Trees

## Definition:

An *AVL tree* is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.

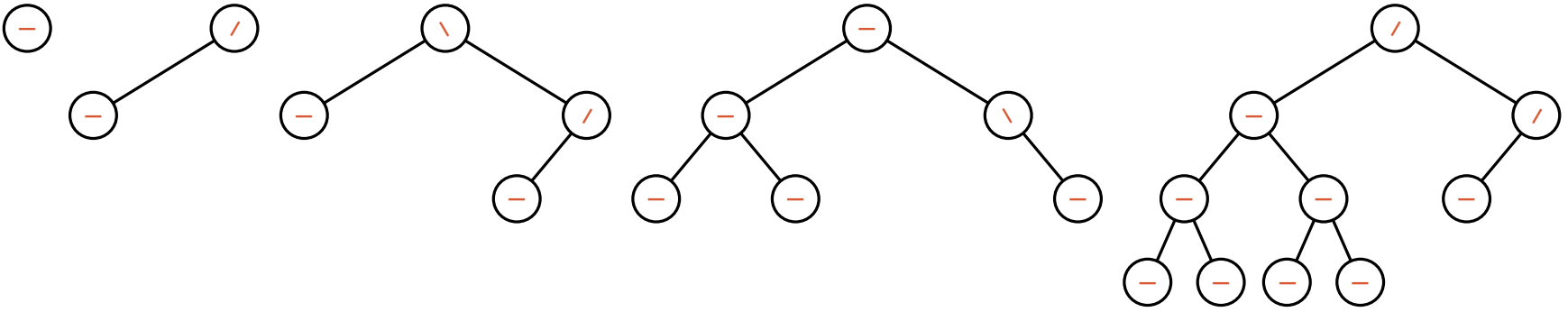
With each node of an AVL tree is associated a *balance factor* that is *left higher*, *equal*, or *right higher* according, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.

## History:

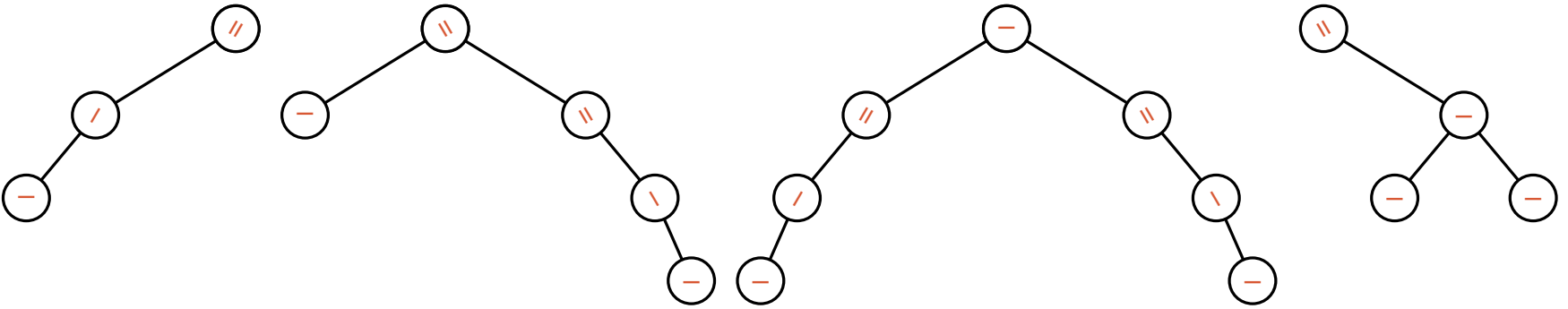
The name *AVL* comes from the discoverers of this method, G. M. Adel'son-Vel'skiï and E. M. Landis. The method dates from 1962.

## Convention in diagrams:

In drawing diagrams, we shall show a left-higher node by '/', a node whose balance factor is equal by '-', and a right-higher node by '\.'



AVL trees



non-AVL trees

## C++ Conventions for AVL Trees

- We employ an enumerated data type to record balance factors:

```
enum Balance_factor { left_higher, equal_height, right_higher };
```

- AVL nodes are structures **derived** from binary search tree nodes with balance factors included:

```
template <class Record>
struct AVL_node: public Binary_node<Record> {

//    additional data member:
    Balance_factor balance;

//    constructors:
    AVL_node();
    AVL_node(const Record &x);

//    overridden virtual functions:
    void set_balance(Balance_factor b);
    Balance_factor get_balance() const;

};
```

- In a Binary\_node, left and right have type Binary\_node \*, so the inherited pointer members of an AVL\_node have this type too, not the more restricted AVL\_node \*. In the insertion method, we must make sure to insert only genuine AVL nodes.
- The benefit of implementing AVL nodes with a derived structure is the reuse of all of our functions for processing nodes of binary trees and search trees.

## Methods for Balance Factors

```
template <class Record>
void AVL_node<Record> :: set_balance(Balance_factor b)
{
    balance = b;
}
```

```
template <class Record>
Balance_factor AVL_node<Record> :: get_balance() const
{
    return balance;
}
```

- We often invoke these methods through pointers to nodes, such as `left->get_balance()`. But `left` could (for the compiler) point to any `Binary_node`, not just to an `AVL_node`, and these methods are declared only for `AVL_node`.
- A C++ compiler must reject a call such as `left->get_balance()`, since `left` might point to a `Binary_node` that is not an `AVL_node`.

## Dummy Methods and Virtual Methods

- To enable calls such as `left->get_balance()`, we include *dummy* versions of `get_balance()` and `set_balance()` in the underlying `Binary_node` structure. These do nothing:

```
template <class Entry>
void Binary_node<Entry> :: set_balance(Balance_factor b)
{}

template <class Entry>
Balance_factor Binary_node<Entry> :: get_balance() const
{return equal_height; }
```

- The correct choice between the AVL version and the dummy version of the method can only be made at run time, when the type of the object `*left` is known.
- We therefore declare the `Binary_node` versions of `set_balance` and `get_balance` as **virtual** methods, selected at run time:

```
template <class Entry>
struct Binary_node {
// data members:
    Entry data;
    Binary_node<Entry> *left;
    Binary_node<Entry> *right;
// constructors:
    Binary_node();
    Binary_node(const Entry &x);
// virtual methods:
    virtual void set_balance(Balance_factor b);
    virtual Balance_factor get_balance() const;
};
```

## Class Declaration for AVL Trees

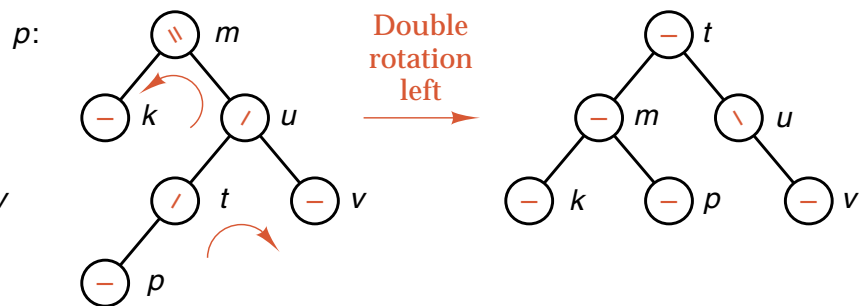
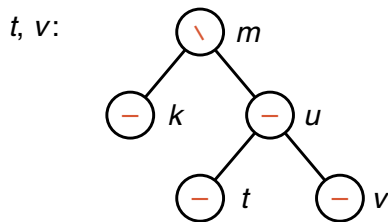
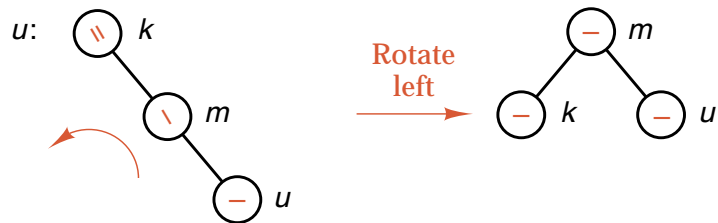
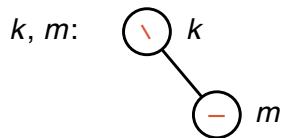
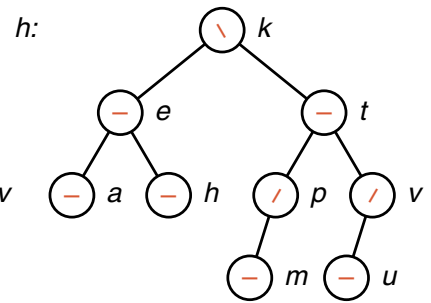
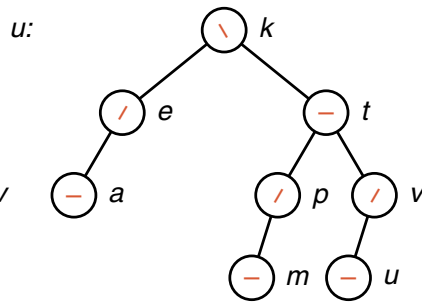
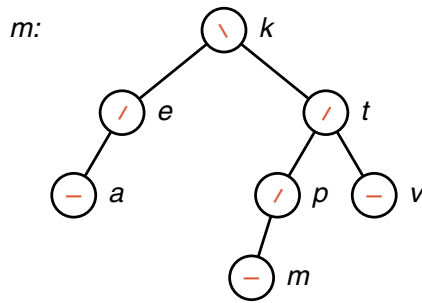
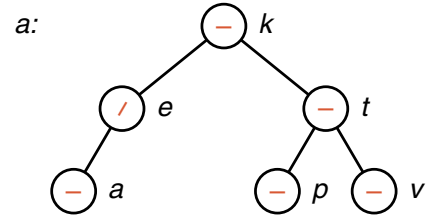
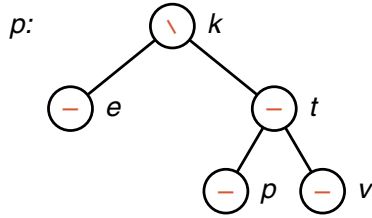
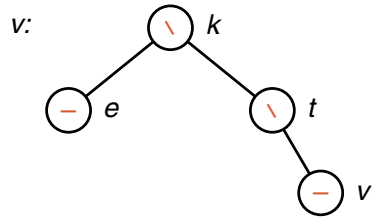
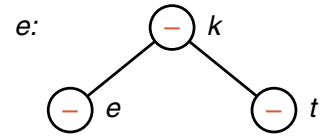
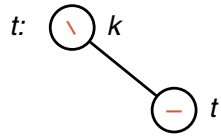
- We must **override** the earlier insertion and deletion functions for binary search trees with versions that maintain the balanced structure of AVL trees.
- All other binary search tree methods can be inherited without any changes.

```
template <class Record>
class AVL_tree: public Search_tree<Record> {
public:
    Error_code insert(const Record &new_data);
    Error_code remove(const Record &old_data);

private:
    // Add auxiliary function prototypes here.
};
```

- The inherited data member of this class is the pointer root, which has type `Binary_node<Record> *` and thus can store the address of either an ordinary binary tree node or an AVL tree node.
- We must ensure that the overridden insert method only creates nodes of type `AVL_node`; doing so will guarantee that all nodes reached via the root pointer of an AVL tree are AVL nodes.

# Insertions into an AVL tree



## Public Insertion Method

```
template <class Record>
Error_code AVL_tree<Record> :: insert(const Record &new_data)
/* Post: If the key of new_data is already in the AVL_tree, a code of duplicate_error
is returned. Otherwise, a code of success is returned and the Record
new_data is inserted into the tree in such a way that the properties of an AVL
tree are preserved.
Uses: avl_insert. */
{
    bool taller;                // Has the tree grown in height?
    return avl_insert(root, new_data, taller);
}
```

## Recursive Function Specifications

```
template <class Record>
Error_code AVL_tree<Record> ::
    avl_insert(Binary_node<Record> * &sub_root,
               const Record &new_data, bool &taller)
/* Pre: sub_root is either NULL or points to a subtree of the AVL_tree
Post: If the key of new_data is already in the subtree, a code of duplicate_error
is returned. Otherwise, a code of success is returned and the Record
new_data is inserted into the subtree in such a way that the properties of
an AVL tree have been preserved. If the subtree is increased in height, the
parameter taller is set to true; otherwise it is set to false.
Uses: Methods of struct AVL_node; functions avl_insert recursively,
left_balance, and right_balance. */
```

## Recursive Insertion

```
{
  Error_code result = success;
  if (sub_root == NULL) {
    sub_root = new AVL_node<Record>(new_data);
    taller = true;
  }

  else if (new_data == sub_root->data) {
    result = duplicate_error;
    taller = false;
  }

  else if (new_data < sub_root->data) { // Insert in left subtree.
    result = avl_insert(sub_root->left, new_data, taller);
    if (taller == true)
      switch (sub_root->get_balance()) { // Change balance factors.
        case left_higher:
          left_balance(sub_root);
          taller = false; // Rebalancing always shortens the tree.
          break;

        case equal_height:
          sub_root->set_balance(left_higher);
          break;

        case right_higher:
          sub_root->set_balance(equal_height);
          taller = false;
          break;
      }
  }
}
```

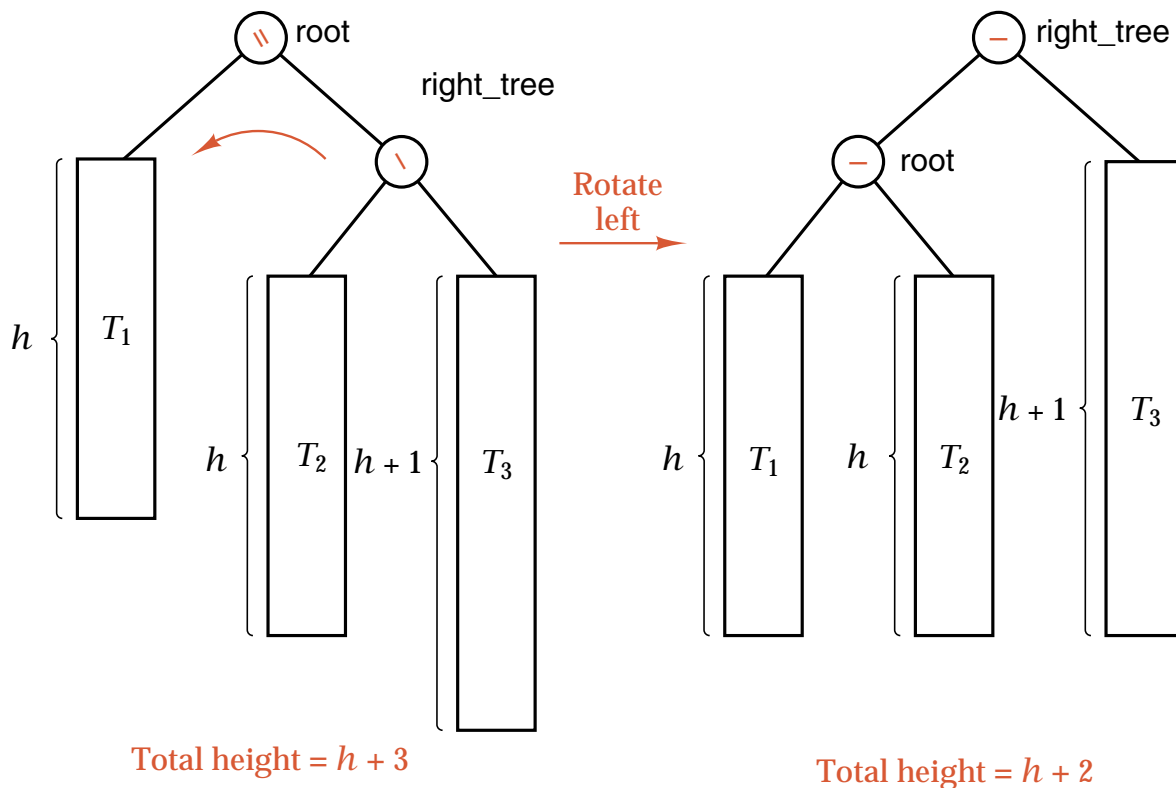
## Recursive Insertion, Continued

```
else {                                     // Insert in right subtree.
    result = avl_insert(sub_root->right, new_data, taller);
    if (taller == true)
        switch (sub_root->get_balance()) {
            case left_higher:
                sub_root->set_balance(equal_height);
                taller = false;
                break;

            case equal_height:
                sub_root->set_balance(right_higher);
                break;

            case right_higher:
                right_balance(sub_root);
                taller = false;           // Rebalancing always shortens the tree.
                break;
        }
    }
return result;
}
```

## Rotations of an AVL Tree

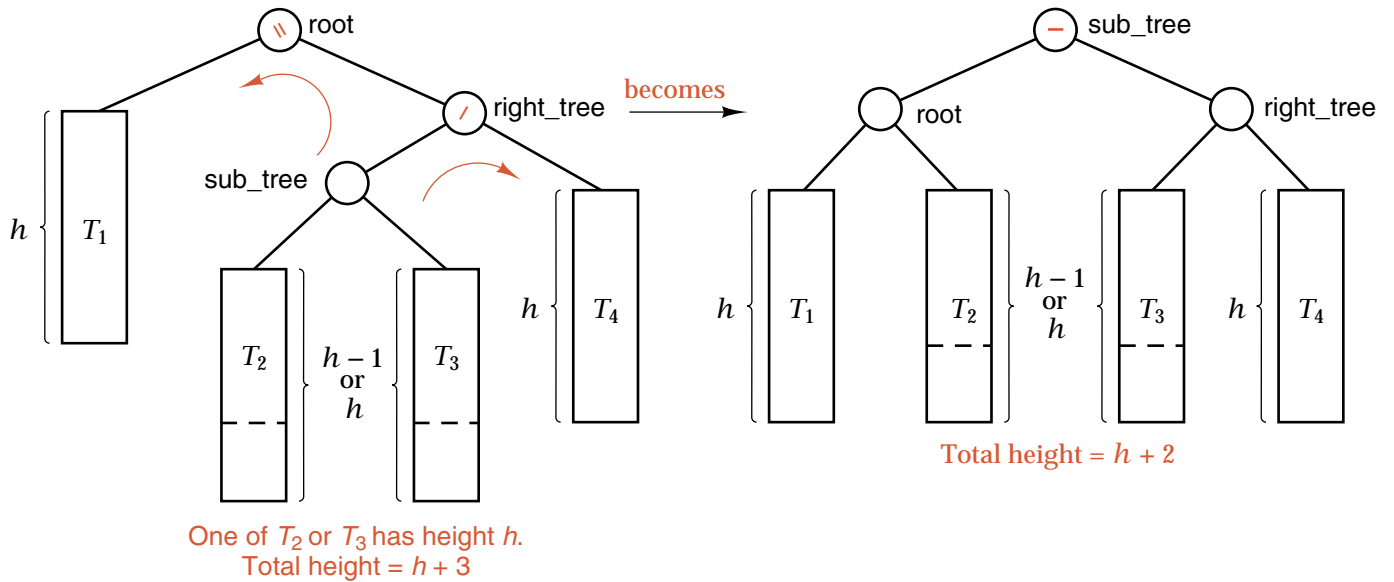


```

template <class Record>
void AVL_tree<Record> :: rotate_left(Binary_node<Record> * &sub_root)
/* Pre:  sub_root points to a subtree of the AVL_tree. This subtree has a nonempty
         right subtree.
   Post: sub_root is reset to point to its former right child, and the former sub_root
         node is the left child of the new sub_root node. */
{
  if (sub_root == NULL || sub_root->right == NULL) // impossible cases
    cout << "WARNING: program error detected in rotate_left" << endl;
  else {
    Binary_node<Record> *right_tree = sub_root->right;
    sub_root->right = right_tree->left;
    right_tree->left = sub_root;
    sub_root = right_tree;
  }
}

```

# Double Rotation



The new balance factors for root and right\_tree depend on the previous balance factor for sub\_tree:

---

| <i>old</i> sub_tree | <i>new</i> root | <i>new</i> right_tree | <i>new</i> sub_tree |
|---------------------|-----------------|-----------------------|---------------------|
| —                   | —               | —                     | —                   |
| /                   | —               | \                     | —                   |
| \                   | /               | —                     | —                   |

---

```

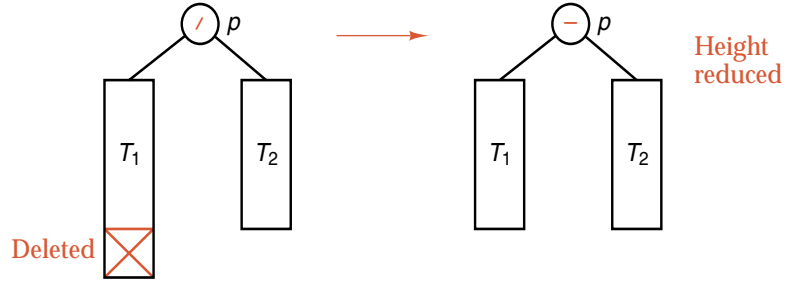
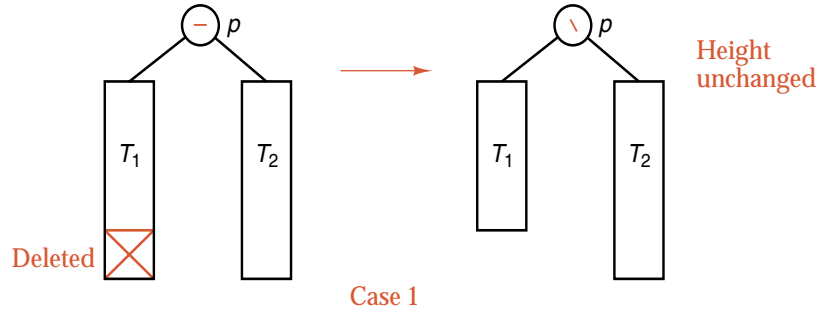
template <class Record>
void AVL_tree<Record> ::
    right_balance(Binary_node<Record> * &sub_root)
/* Pre:  sub_root points to a subtree of an AVL_tree, doubly unbalanced on the right.
   Post:  The AVL properties have been restored to the subtree.
   Uses:  Methods of struct AVL_node; functions rotate_right, rotate_left. */
{
    Binary_node<Record> * &right_tree = sub_root->right;
    switch (right_tree->get_balance()) {
    case right_higher:           //  single rotation left
        sub_root->set_balance(equal_height);
        right_tree->set_balance(equal_height);
        rotate_left(sub_root); break;
    case equal_height:          //  impossible case
        cout << "WARNING: program error in right_balance" << endl;
    case left_higher:           //  double rotation left
        Binary_node<Record> *sub_tree = right_tree->left;
        switch (sub_tree->get_balance()) {
        case equal_height:
            sub_root->set_balance(equal_height);
            right_tree->set_balance(equal_height); break;
        case left_higher:
            sub_root->set_balance(equal_height);
            right_tree->set_balance(right_higher); break;
        case right_higher:
            sub_root->set_balance(left_higher);
            right_tree->set_balance(equal_height); break;
        }
        sub_tree->set_balance(equal_height);
        rotate_right(right_tree);
        rotate_left(sub_root); break;
    }
}
}

```

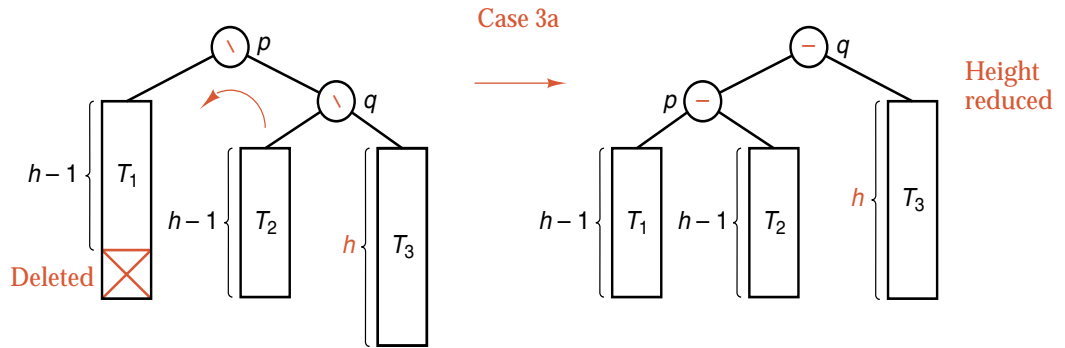
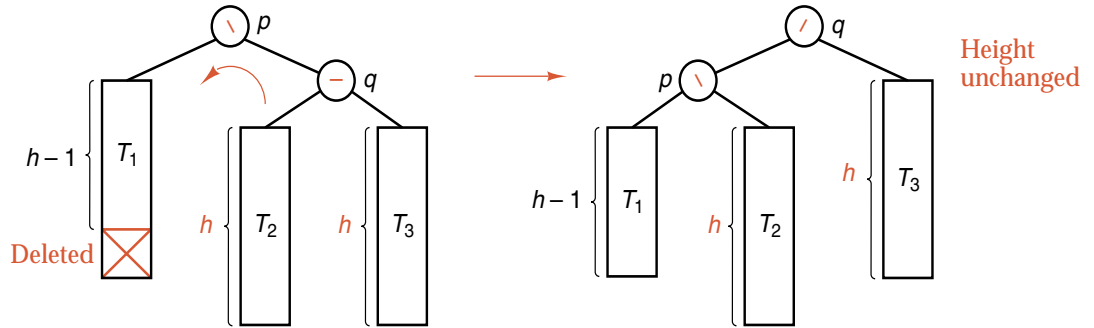
## Removal of a Node

1. Reduce the problem to the case when the node  $x$  to be removed has at most one child.
2. Delete  $x$ . We use a **bool** variable `shorter` to show if the height of a subtree has been shortened.
3. While `shorter` is true do the following steps for each node  $p$  on the path from the parent of  $x$  to the root of the tree. When `shorter` becomes false, the algorithm terminates.
4. *Case 1*: Node  $p$  has balance factor equal. The balance factor of  $p$  is changed according as its left or right subtree has been shortened, and `shorter` becomes false.
5. *Case 2*: The balance factor of  $p$  is not equal, and the taller subtree was shortened. Change the balance factor of  $p$  to equal, and leave `shorter` as true.
6. *Case 3*: The balance factor of  $p$  is not equal, and the shorter subtree was shortened. Apply a rotation as follows to restore balance. Let  $q$  be the root of the taller subtree of  $p$ .
7. *Case 3a*: The balance factor of  $q$  is equal. A single rotation restores balance, and `shorter` becomes false.
8. *Case 3b*: The balance factor of  $q$  is the same as that of  $p$ . Apply a single rotation, set the balance factors of  $p$  and  $q$  to equal, and leave `shorter` as true.
9. *Case 3c*: The balance factors of  $p$  and  $q$  are opposite. Apply a double rotation (first around  $q$ , then around  $p$ ), set the balance factor of the new root to equal and the other balance factors as appropriate, and leave `shorter` as true.

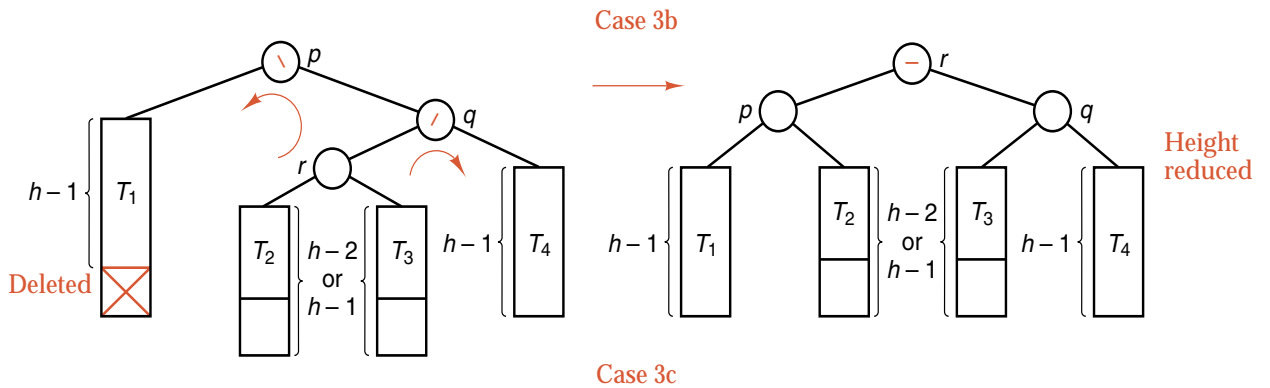
no rotations



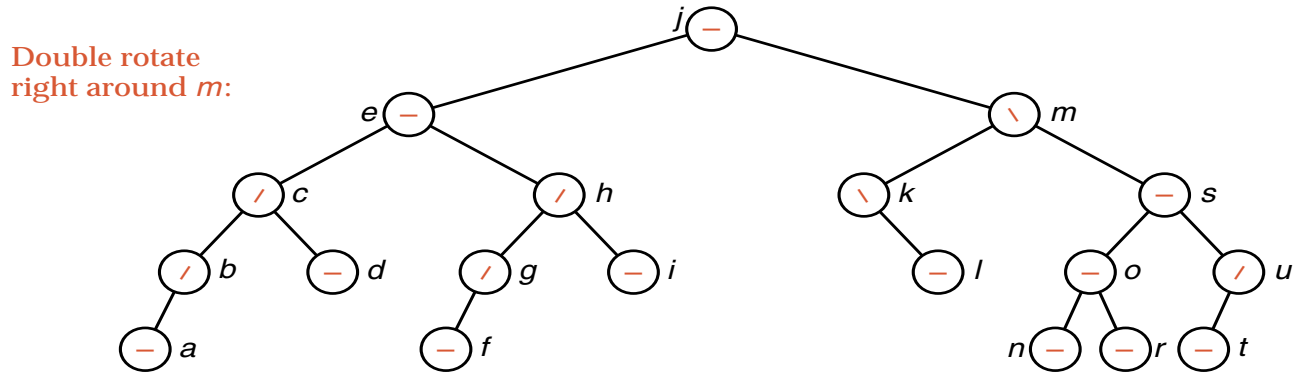
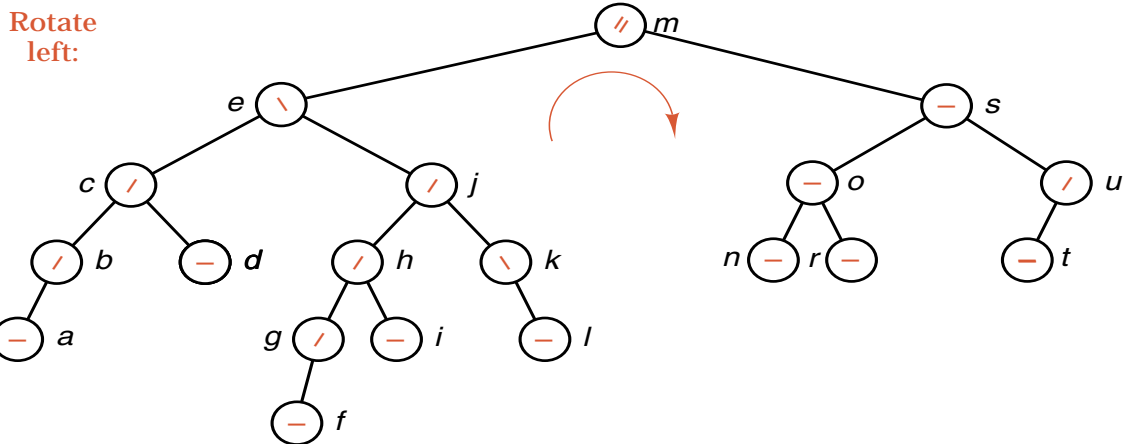
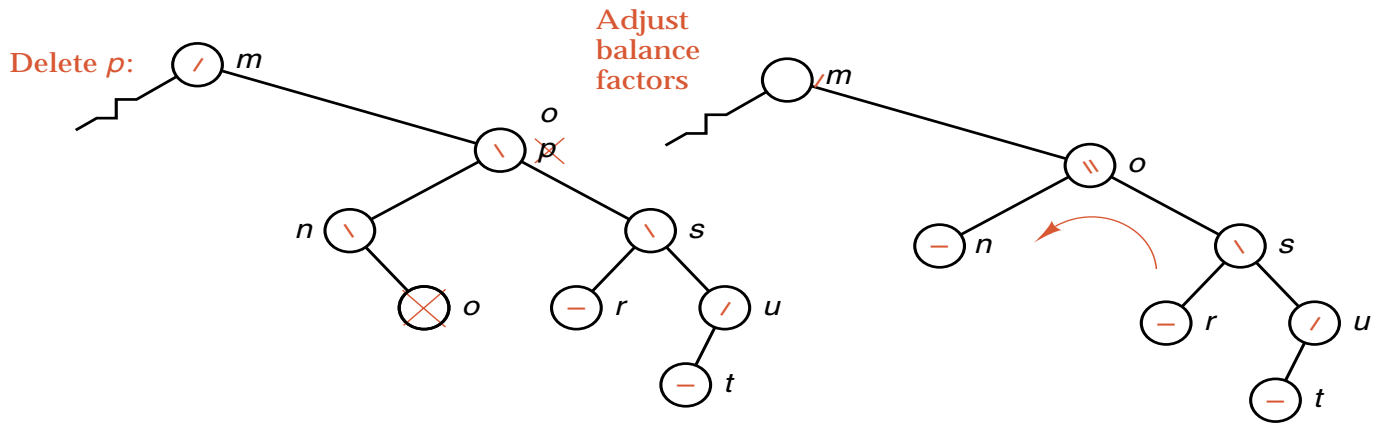
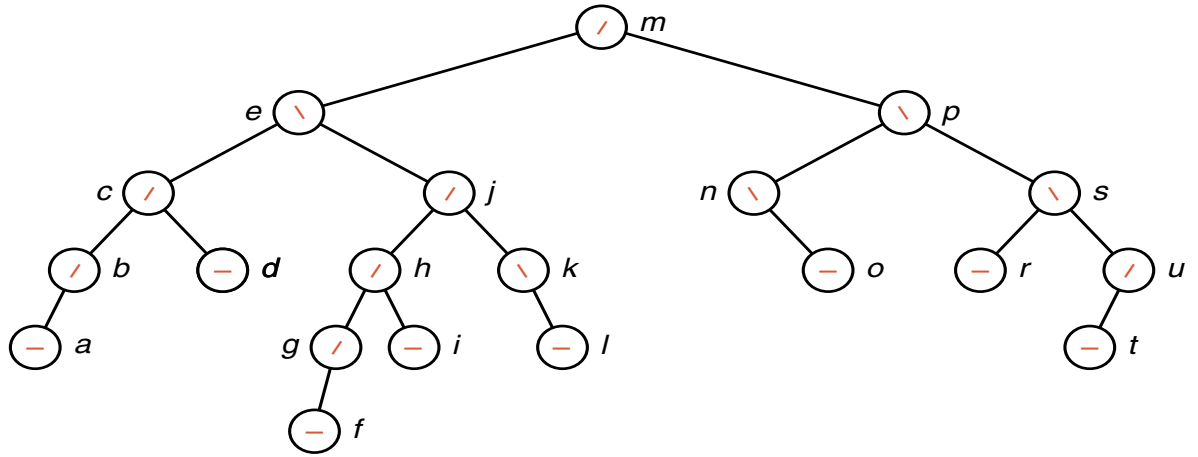
single left rotations



double rotation



Initial:



## Analysis of AVL Trees

- The number of recursive calls to insert a new node can be as large as the height of the tree.
- At most one (single or double) rotation will be done per insertion.
- A rotation improves the balance of the tree, so later insertions are less likely to require rotations.
- It is very difficult to find the height of the *average* AVL tree, but the worst case is much easier. The worst-case behavior of AVL trees is essentially no worse than the behavior of random search trees.
- Empirical evidence suggests that the average behavior of AVL trees is much better than that of random trees, almost as good as that which could be obtained from a perfectly balanced tree.