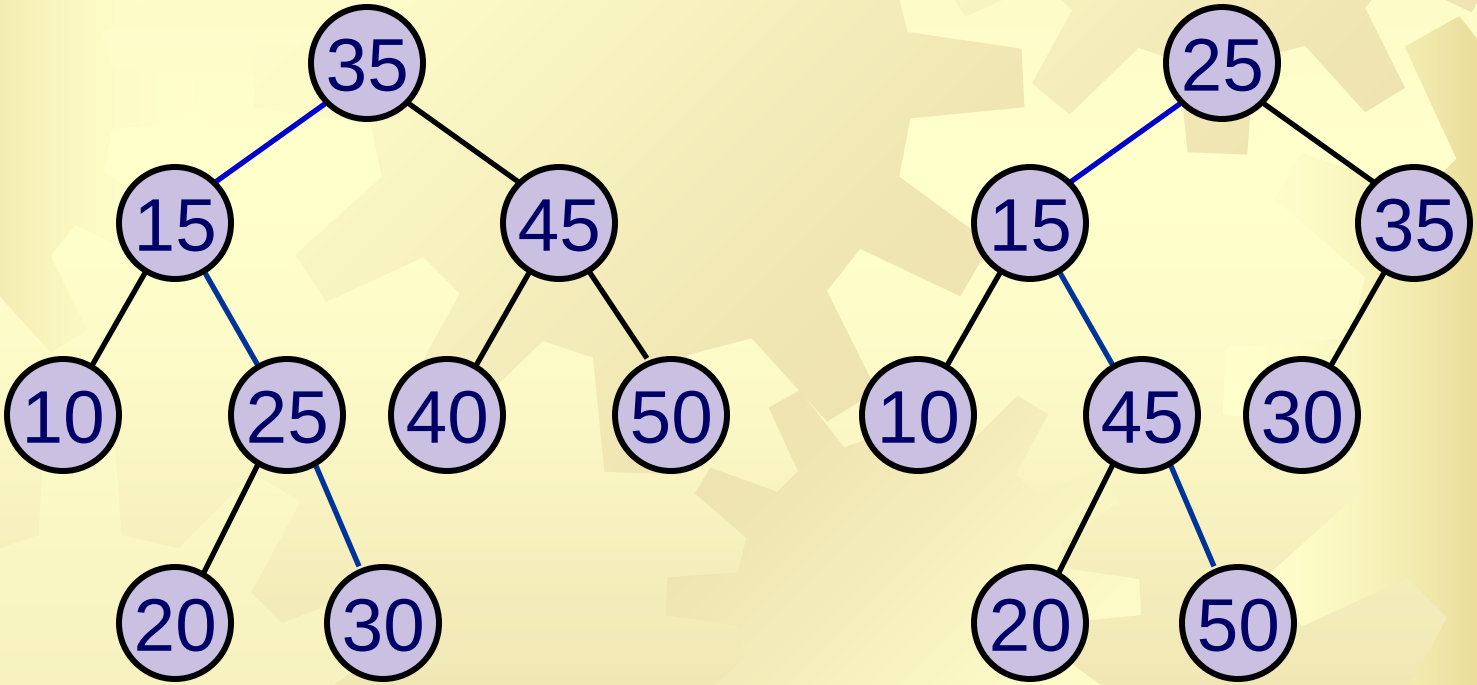
Binary Search Trees

- **DEFINITION: A binary search tree is a binary tree that is either empty or in which the data entry of every node has a key and satisfies the conditions:**
  - **The key of the left child of a node (if it exists) is less than the key of its parent node.**
  - **The key of the right child of a node (if it exists) is greater than the key of its parent node.**
  - **The left and right subtrees of the root are again binary search trees.**

always require

- No two entries in a binary search tree may have equal keys.



The Binary Search Tree Class

- The binary search tree class will be derived from the binary tree class; hence all binary tree methods are inherited.
  - template <class Record>
    class Search_tree: public Binary_tree<Record> {
    public:
        Error_code insert(const Record &new data);
        Error_code remove(const Record &old data);
        Error_code tree search(Record &target) const;
    private: *// Add auxiliary function prototypes here.*
    };

The Binary Search Tree Class

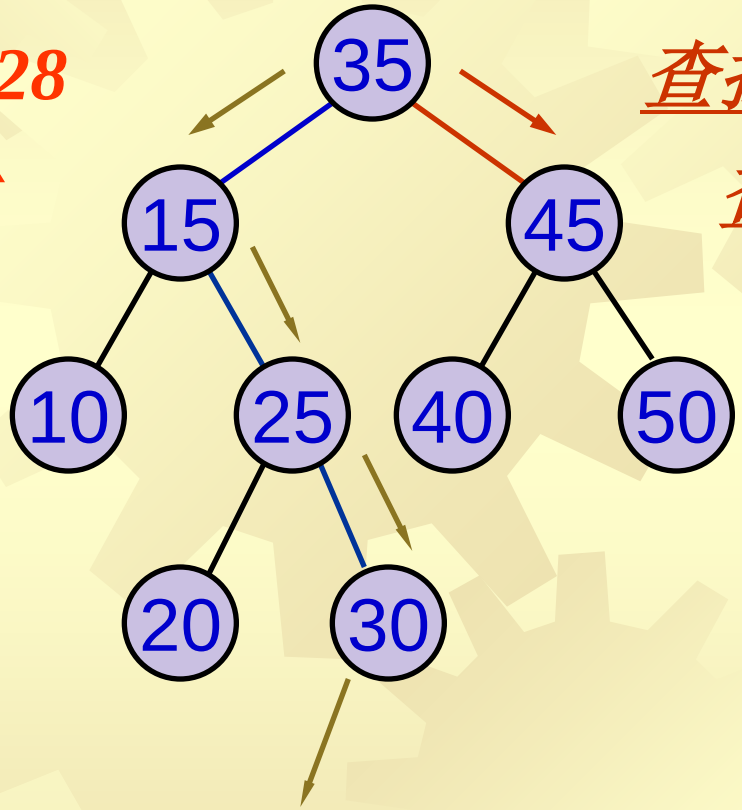
- The inherited methods include the constructors, the destructor, clear, empty, size, height, and the traversals preorder, inorder, and postorder.

- A binary search tree also admits specialized methods called insert, remove, and tree_search.

Tree Search

查找28

查找失败



查找45

查找成功

Tree Search

- Error_code Search_tree<Record> :: tree_search(Record &target) const;
  - Post: If there is an entry in the tree whose key matches that in target, the parameter target is replaced by the corresponding record from the tree and a code of success is returned. Otherwise a code of not present is returned.

Algorithm

- This method will often be called with a parameter target that contains only a key value. The method will fill target with the complete data belonging to any corresponding Record in the tree.
- To search for the target, we first compare it with the entry at the root of the tree. If their keys match, then we are finished. Otherwise, we go to the left subtree or right subtree as appropriate and repeat the search in that subtree.

Algorithm

- We program this process by calling an auxiliary recursive function.
- The process terminates when it either finds the target or hits an empty subtree.

Algorithm

- The auxiliary search function returns a pointer to the node that contains the target back to the calling program. Since it is private in the class, this pointer manipulation will not compromise tree encapsulation.

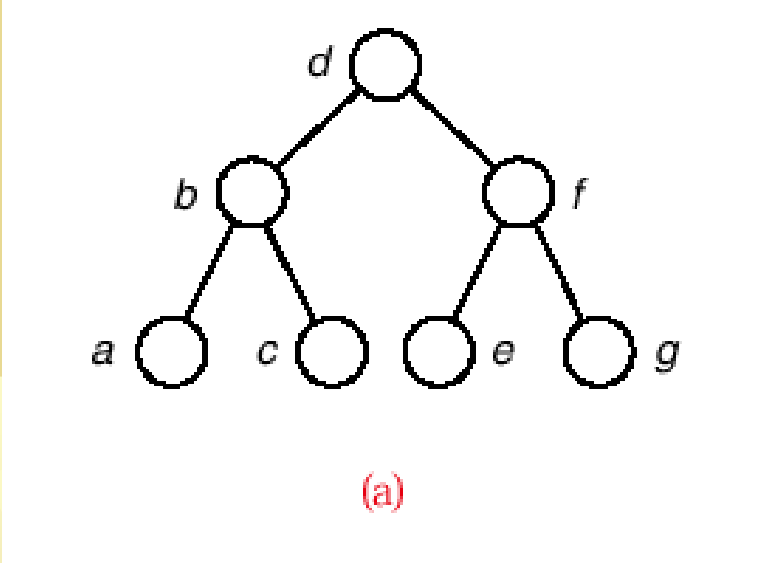
Recursive auxiliary function

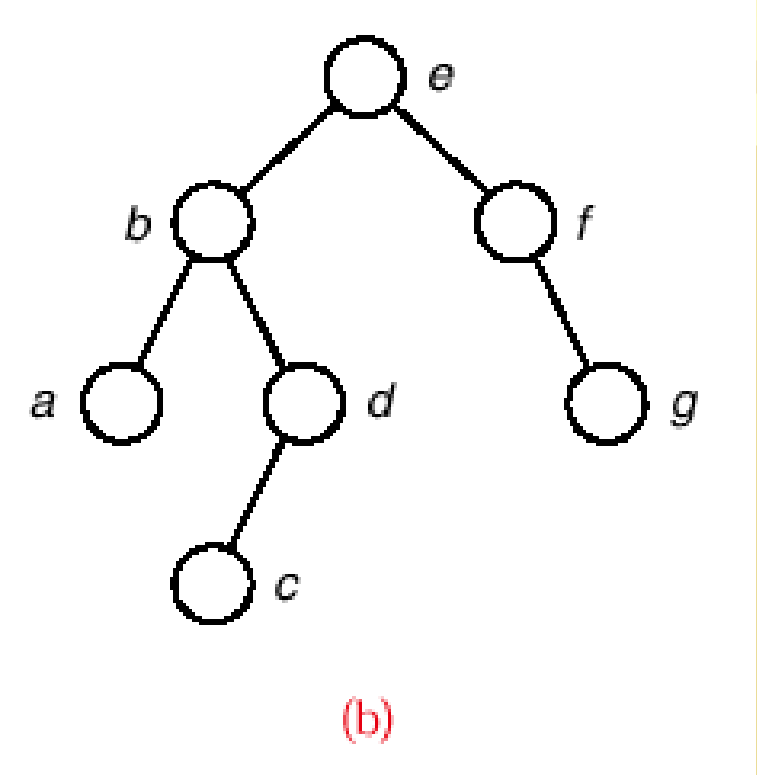
```
template <class Record>
Binary_node<Record>
    *Search_tree<Record> ::search_for_node(
     Binary_node<Record> * sub_root, const Record &target)
    const
{
    if (sub_root == NULL || sub_root->data == target)
            return sub_root;
    else if (sub_root->data < target)
            return search_for_node(sub_root->right, target);
    else return search_for_node(sub_root->left, target);
}
```

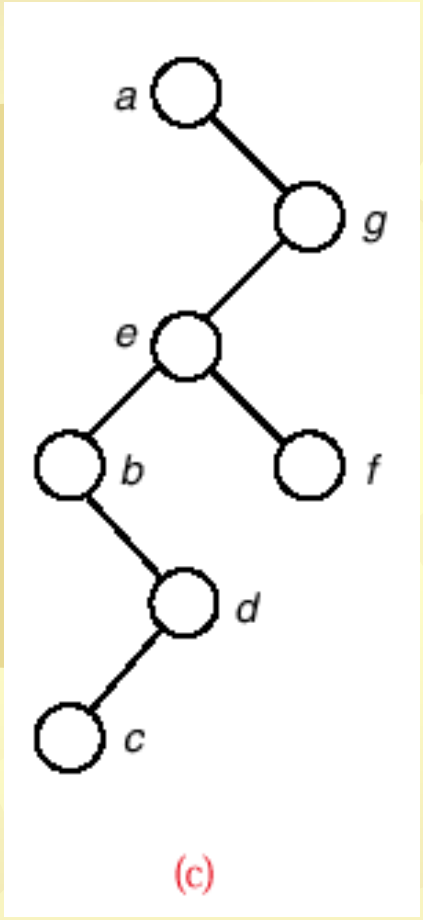
Nonrecursive version

```
template <class Record>
Binary_node<Record>
    *Search_tree<Record> ::search_for_node(  Binary_node<R
    ecord> *sub_root, const Record &target) const
{
    while (sub_root != NULL && sub_root->data != target)
          if (sub_root->data < target) sub_root = sub_root->right;
          else sub_root = sub_root->left;
    return sub_root;
}
```

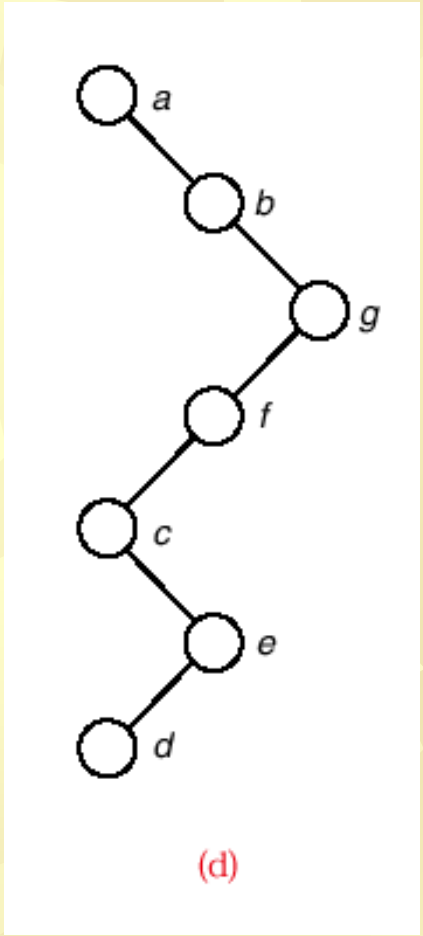
Binary Search Trees with the Same Keys



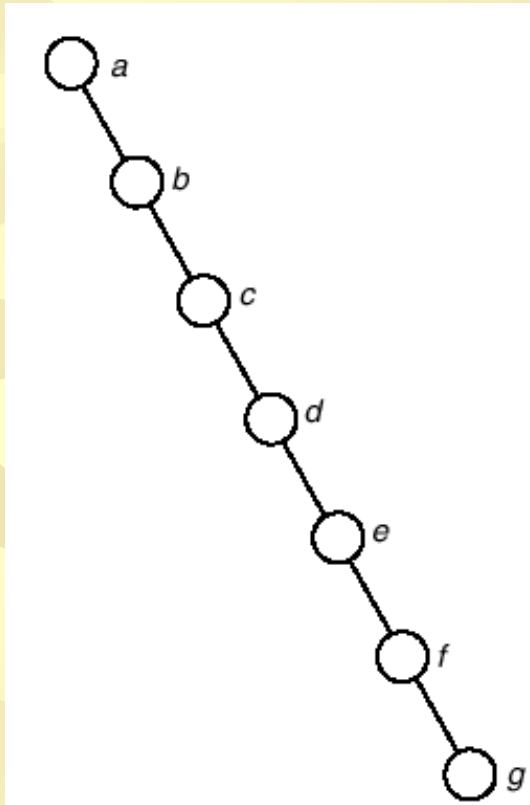




(c)



(d)



(e)

Analysis of Tree Search

- The same keys may be built into binary search trees of many different shapes.

- If a binary search tree is nearly completely balanced (“bushy”), then tree search on a tree with n vertices will also do O(log n) comparisons of keys.

Analysis of Tree Search

- If the tree degenerates into a long chain, then tree search becomes the same as sequential search, doing O(n) comparisons on n vertices. This is the worst case for tree search.

- The number of vertices between the root and the target, inclusive, is the number of comparisons that must be done to find the target. The bushier the tree, the smaller the number of comparisons that will usually need to be done.

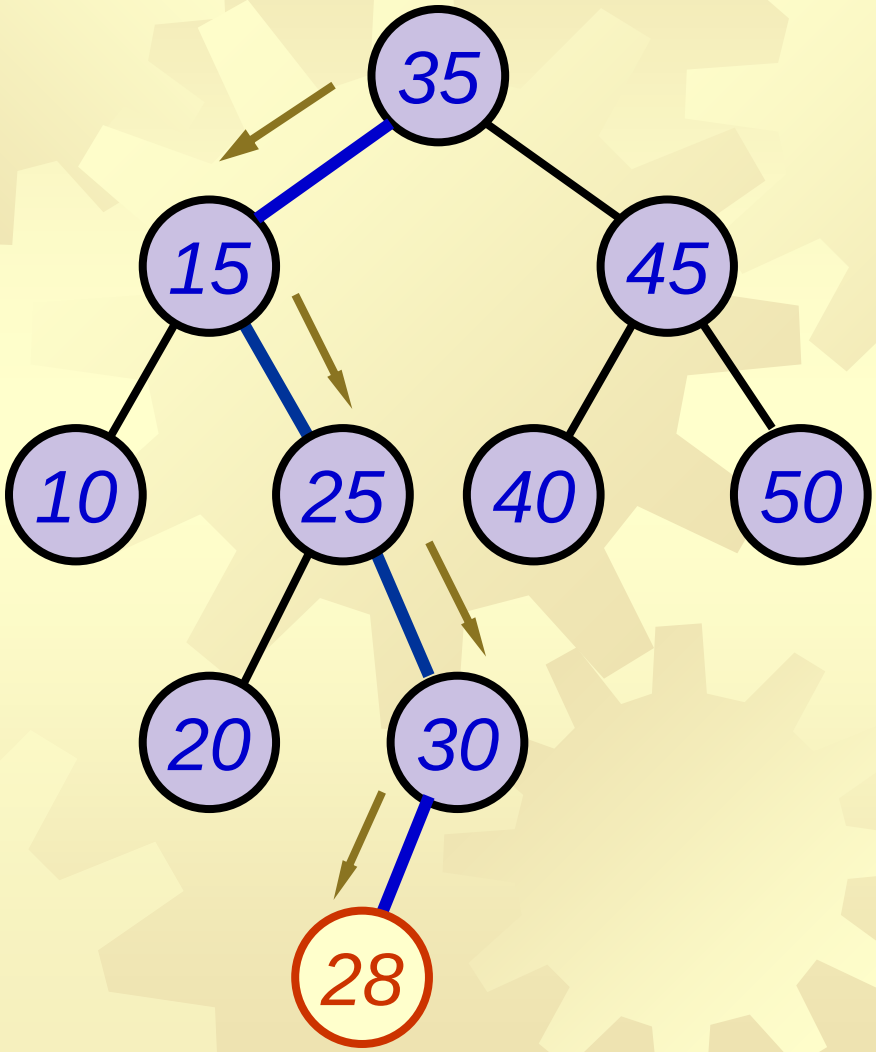
Analysis of Tree Search

- It is often not possible to predict (in advance of building it) what shape of binary search tree will occur.
- In practice, if the keys are built into a binary search tree in random order, then it is extremely unlikely that a binary search tree degenerates badly; tree search usually performs almost as well as binary search.

Insertion into a Binary Search Tree

- Error code Search tree<Record> :: insert(const Record &new_data);
  - Post: If a Record with a key matching that of new_data already belongs to the Search tree a code of duplicate error is returned.
  - Otherwise, the Record new_data is inserted into the tree in such a way that the properties of a binary search tree are preserved, and a code of success is returned.

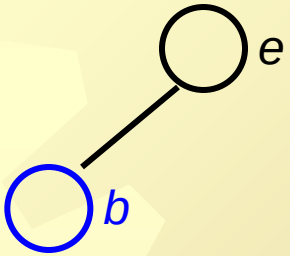
Insert New Node 28



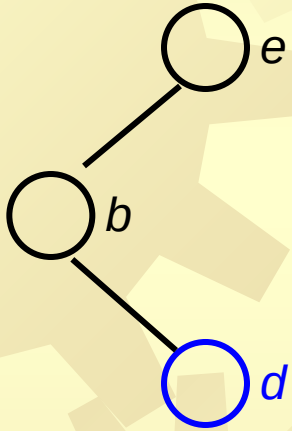
Insertion into a Binary Search Tree

- **Insert: e b d f a g c**

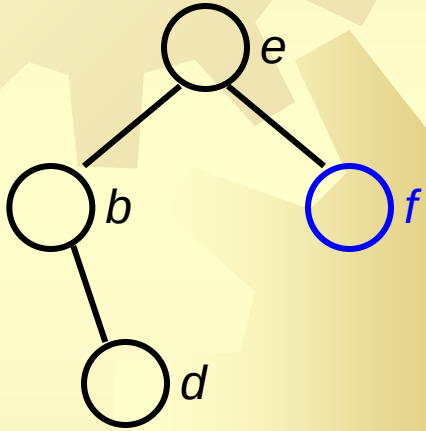
Insert e



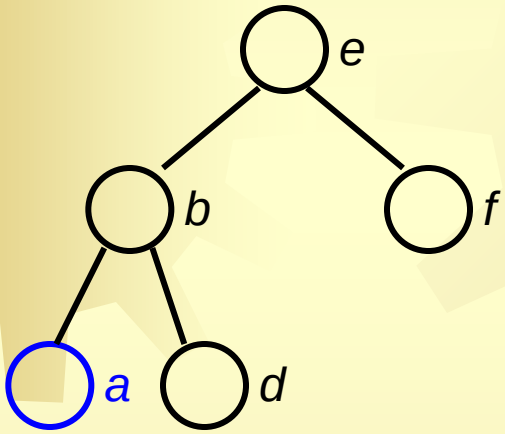
Insert b



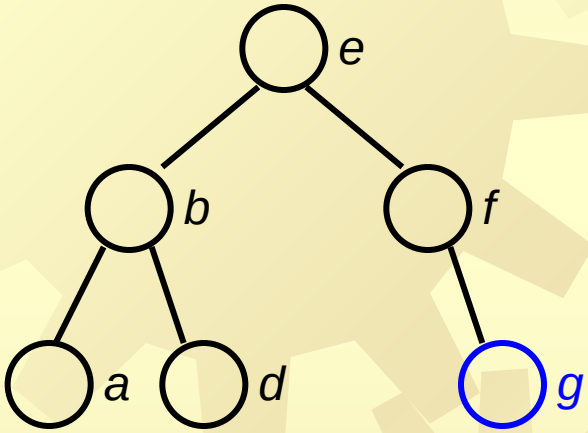
Insert d



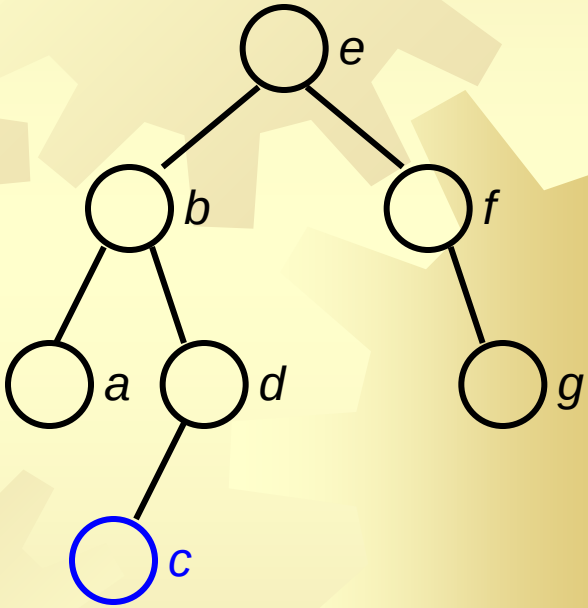
Insert f



Insert a



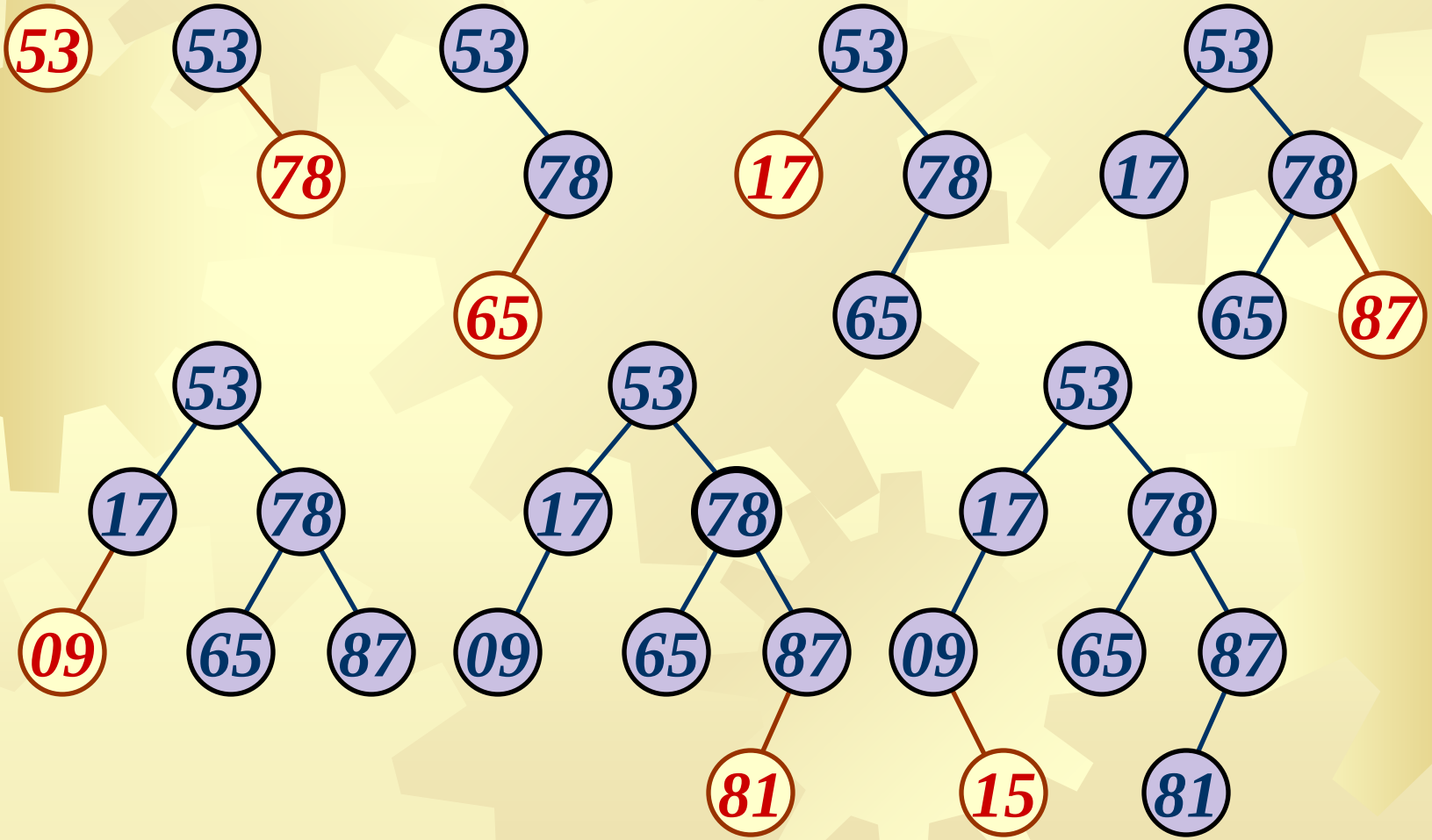
Insert g



Insert c

Create a BST

- Input { 53, 78, 65, 17, 87, 09, 81, 15 }



Method for Insertion

```
template <class Record>
Error_code Search_tree<Record> ::
    insert(const Record &new_data)
{
    return search_and_insert(root,
                             new_data);
}
```

Method for Insertion

```
template <class Record>
Error_code Search tree<Record> ::search_and_insert(
Binary_node<Record> * &sub_root, const Record
&new_data)
{
      if (sub_root == NULL) {
         sub_root = new Binary_node<Record>(new_data);
         return success;
      }
      else if (new_data < sub_root->data)
         return search_and_insert(sub_root->left, new_data);
      else if (new_data > sub_root->data)
         return search_and_insert(sub_root->right, new_data);
      else return duplicate_error;
}
```

Insertion

- The method insert can usually insert a new node into a random binary search tree with n nodes in O(log n) steps. It is possible, but extremely unlikely, that a random tree may degenerate so that insertions require as many as n steps.

- If the keys are inserted in sorted order into an empty tree, however, this degenerate case will occur.

Treesort

- When a binary search tree is traversed in inorder, the keys will come out in sorted order.

- This is the basis for a sorting method, called treesort: Take the entries to be sorted, use the method insert to build them
into a binary search tree, and then use inorder traversal to put them out in order.

Advantage

- First advantage of treesort : The nodes need not all be available at the start of the process, but are built into the tree one by one as they become available.

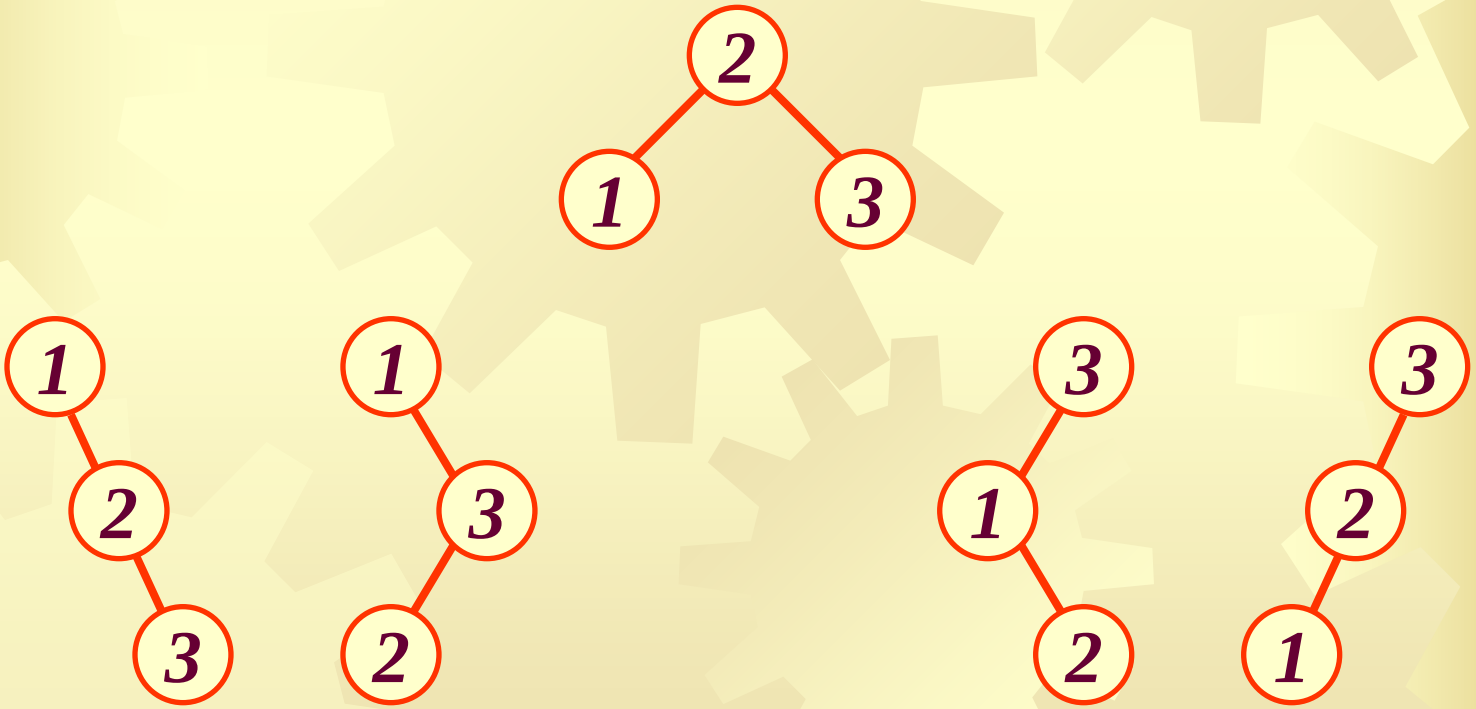
- Second advantage: The search tree remains available for later insertions and removals.

Drawback

- Drawback: If the keys are already sorted, then treesort will be a disaster--the search tree it builds will reduce to a chain. Treesort should never be used if the keys are already sorted, or are nearly so.

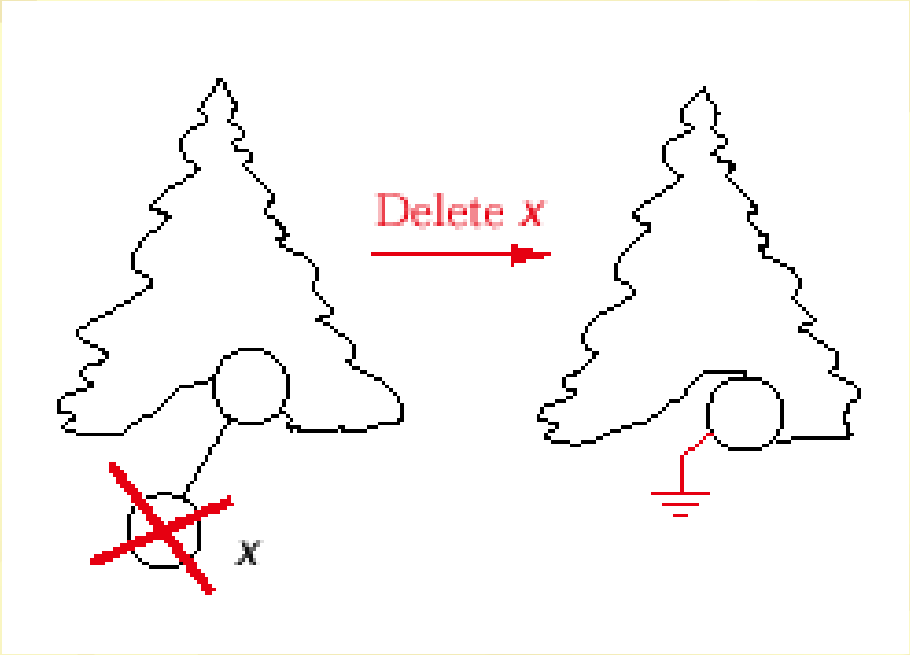
- 输入序列决定树型。

{1, 2, 3} {1, 3, 2} {2, 3, 1} {3, 1, 2} {3, 2, 1}



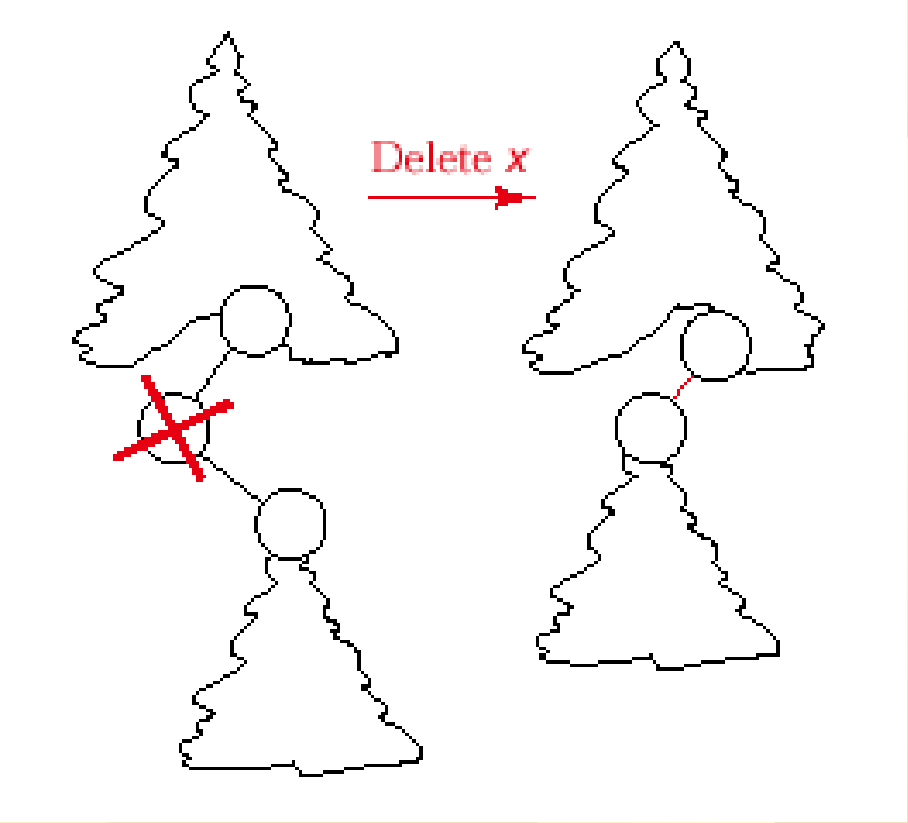
Removal from a Binary Search Tree

- Case: deletion of a leaf



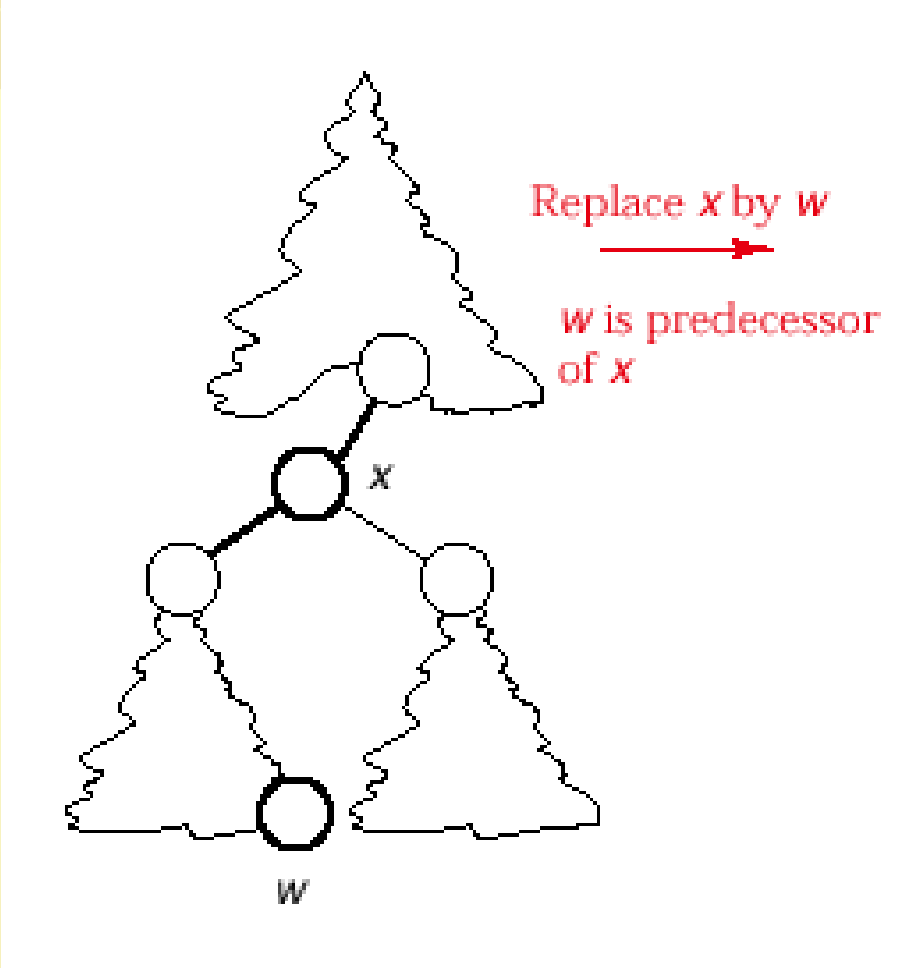
Removal from a Binary Search Tree

- **Case: one subtree empty**

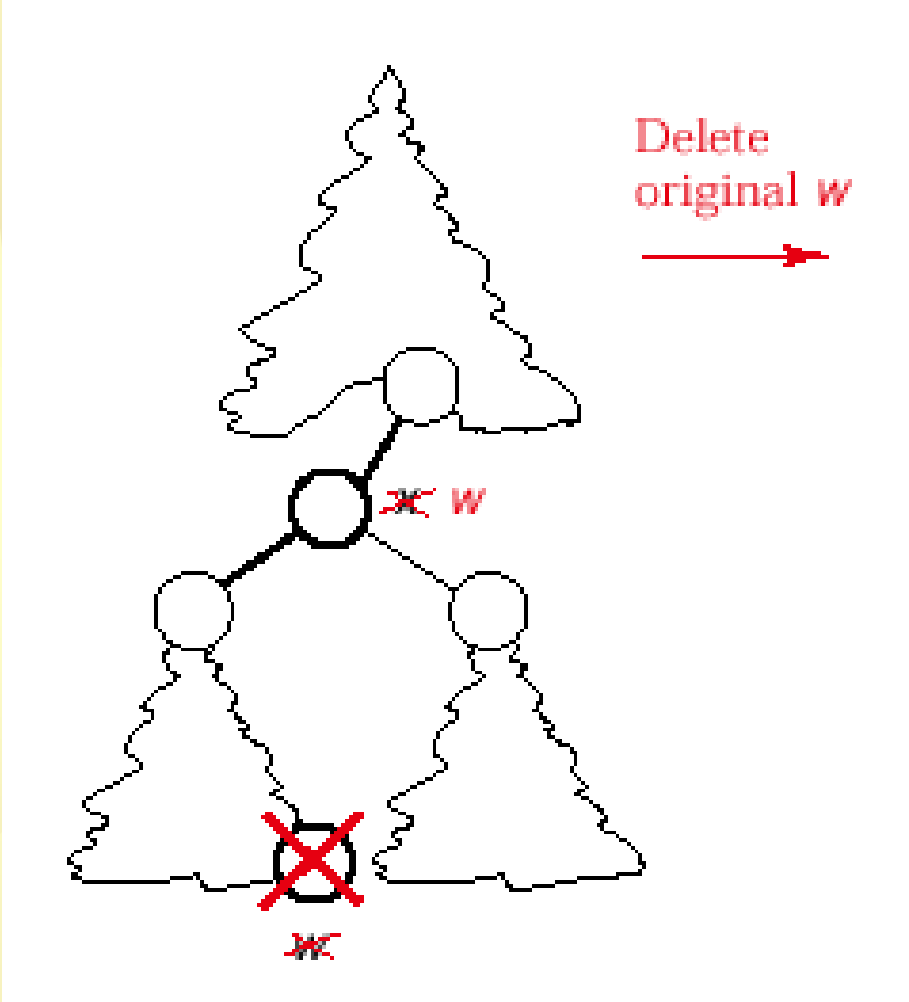


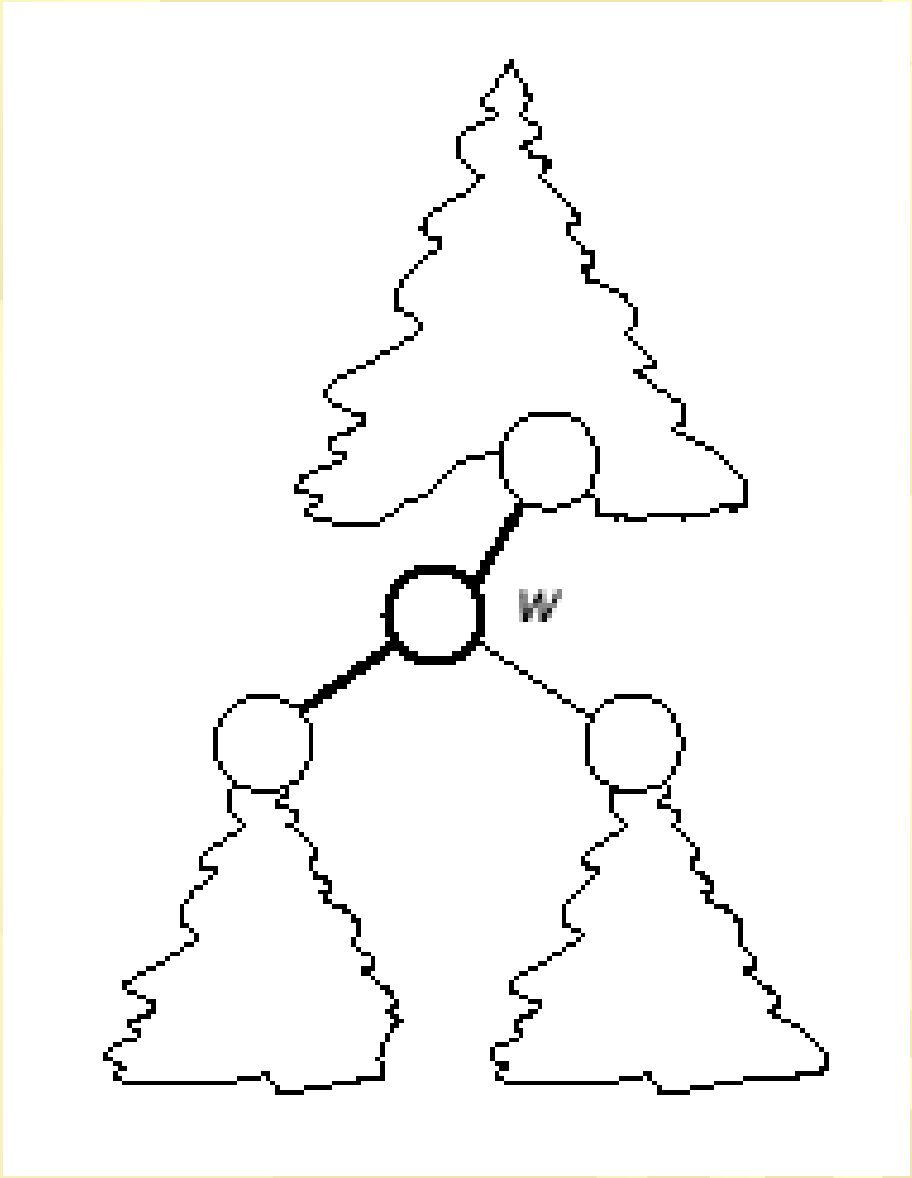
Removal from a Binary Search Tree

- **Case: neither subtree empty**



Removal from a Binary Search Tree





Auxiliary Function to Remove One Node

```
template <class Record>
    Error_code
    Search_tree<Record> ::remove_root(
        Binary_node<Record> * &sub_root)
    /* Pre: sub_root is either NULL or points to a
    subtree of the Search_tree .
    Post: If sub_root is NULL , a code of not_present
    is returned. Otherwise, the root of the subtree is
    removed in such a way that the properties of a
    binary search tree are preserved. The parameter
    sub_root is reset as the root of the modified
    subtree, and success is returned. */
```

Auxiliary Function to Remove One Node

```
{
    if (sub_root == NULL) return not_present;
    Binary_node<Record> *to_delete = sub_root;
                        // Remember node to_delete at end.
    if (sub_root->right == NULL)        // one child
            sub_root = sub_root->left;
    else if (sub_root->left == NULL)            // one child
            sub_root = sub_root->right;
    else {              // Neither subtree is empty, two children
            to_delete = sub_root->left;
                        // Move left to find predecessor.
            Binary_node<Record> *parent = sub_root;
                        // parent of to_delete
```

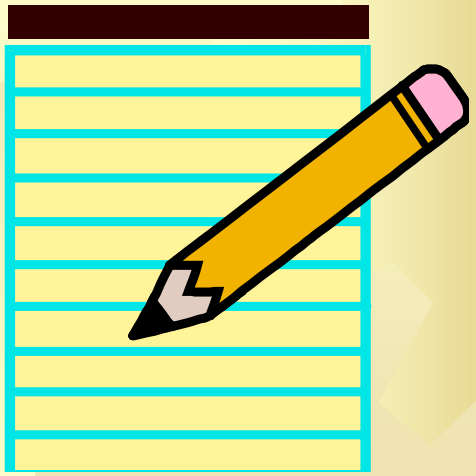
Auxiliary Function to Remove One Node

```
        while (to_delete->right != NULL) {
                // to_delete is not the predecessor.
                parent = to_delete;
                to_delete = to_delete->right;
        }
        sub_root->data = to_delete->data;
                // Move from to_delete to root.
        if (parent == sub_root)
                sub_root->left = to_delete->left;
        else parent->right = to_delete->left;
    }
    delete to_delete; //Remove to_delete from tree.
    return success;
}
```

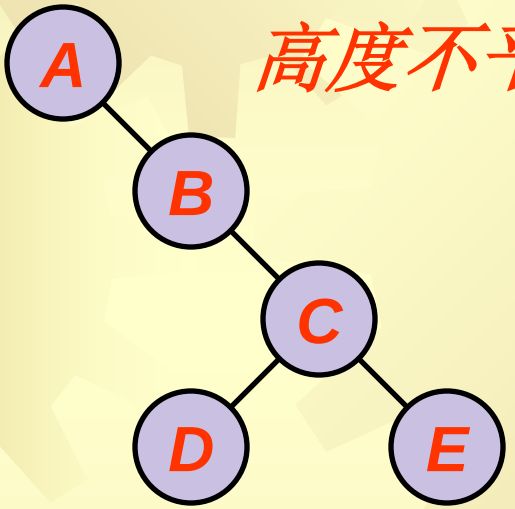
Removal Method

```
template <class Record>
Error_code Search_tree<Record> :: search_and_destroy
( Binary_node<Record> * &sub_root, const Record &target)
/* Pre: sub_root is either NULL or points to a subtree of the Search
tree .
Post: If the key of target is not in the subtree, a code of
not_present is returned. Otherwise, a code of success is returned
and the subtree node containing target has been removed in such
a way that the properties of a binary search tree have been
preserved.
Uses: Functions search_and_destroy recursively and remove root
*/
{
        if (sub_root == NULL || sub_root->data == target)
            return remove_root(sub_root);
        else if (target < sub_root->data)
            return search_and_destroy(sub_root->left, target);
        else
            return search_and_destroy(sub_root->right, target);
}
```

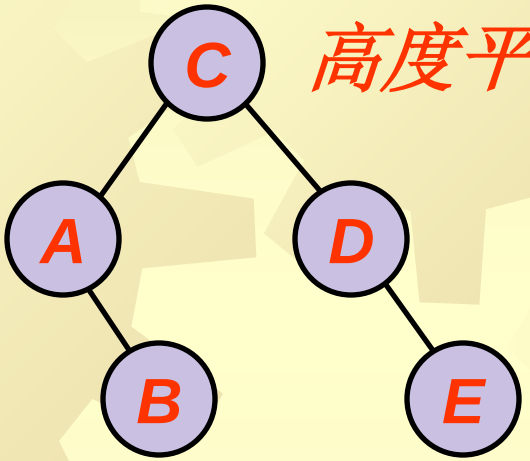
Questions?



BST



高度不平衡



高度平衡

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:

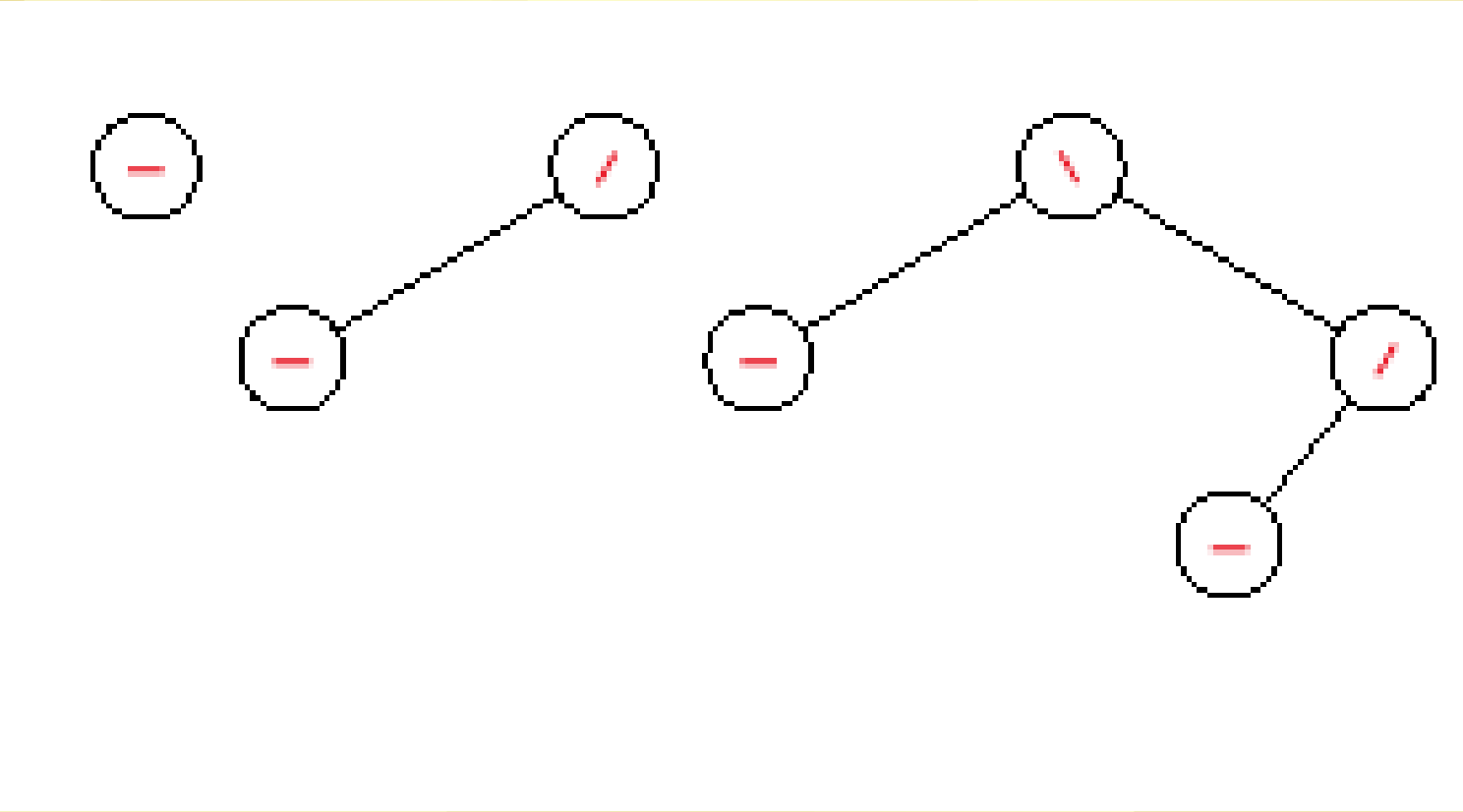
- The name AVL comes from the discoverers of this method, G. M. Adel'son-Vel'skiĭ and E. M. Landis. The method dates from 1962.

Convention in diagrams:

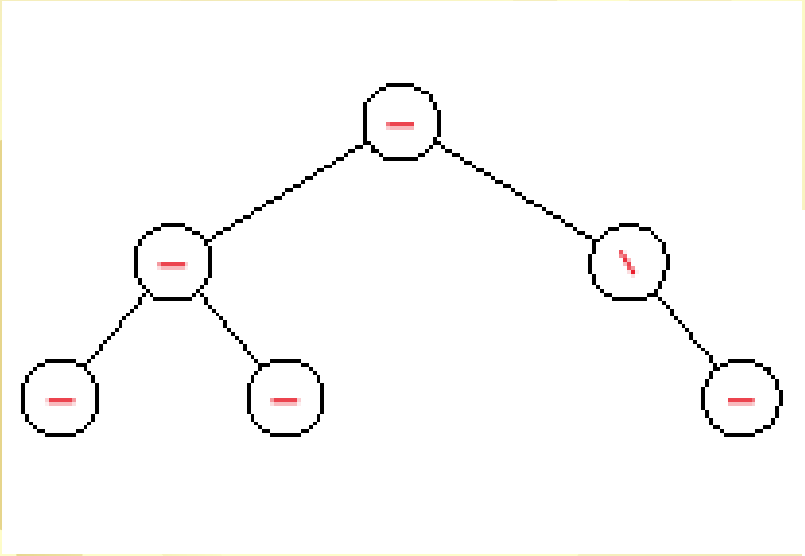
- In drawing diagrams, we shall show a left-higher node by `/,' a node whose balance factor is equal by ` - ,' and a right-higher node by `\ `.

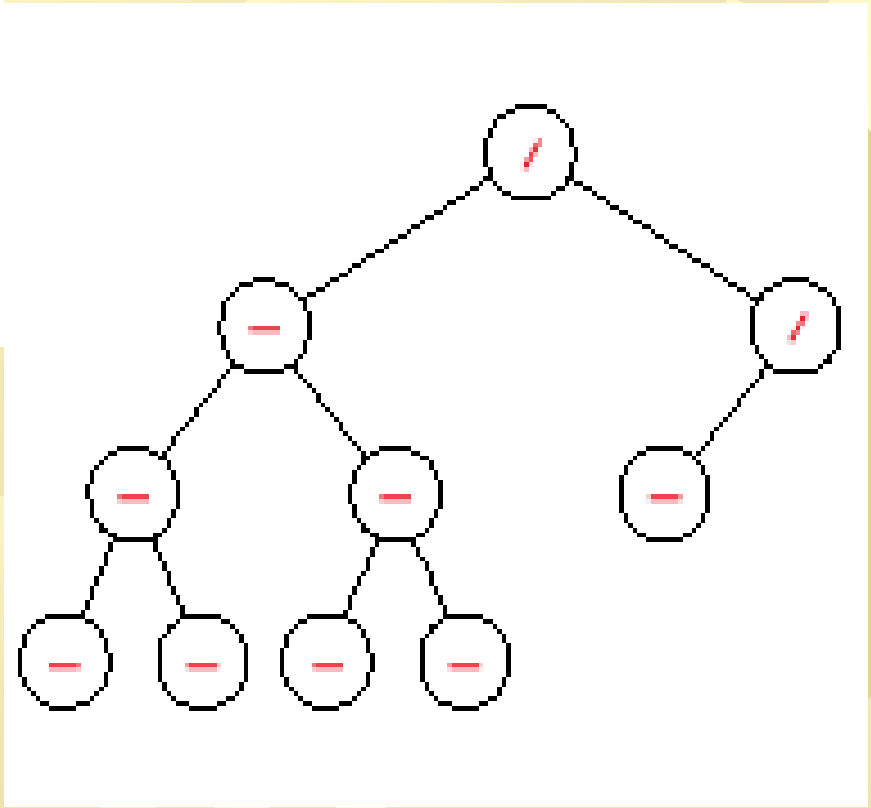
- or, mark the balance factor beside the node

AVL Trees

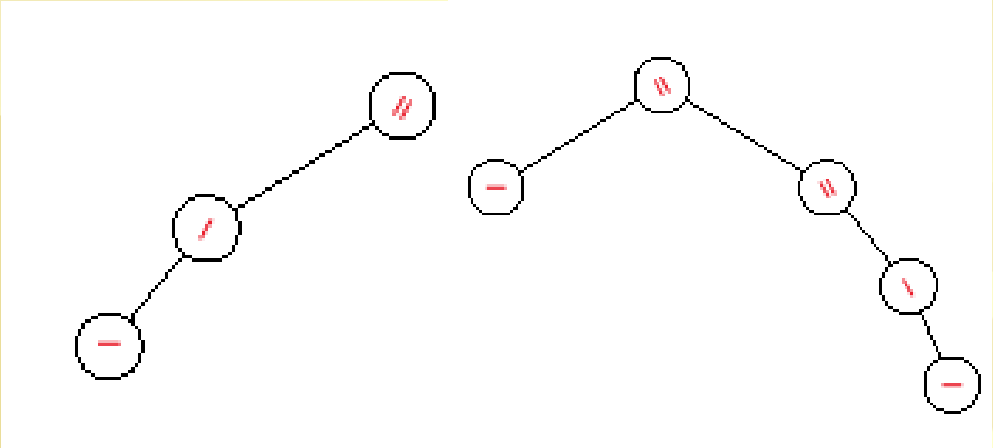


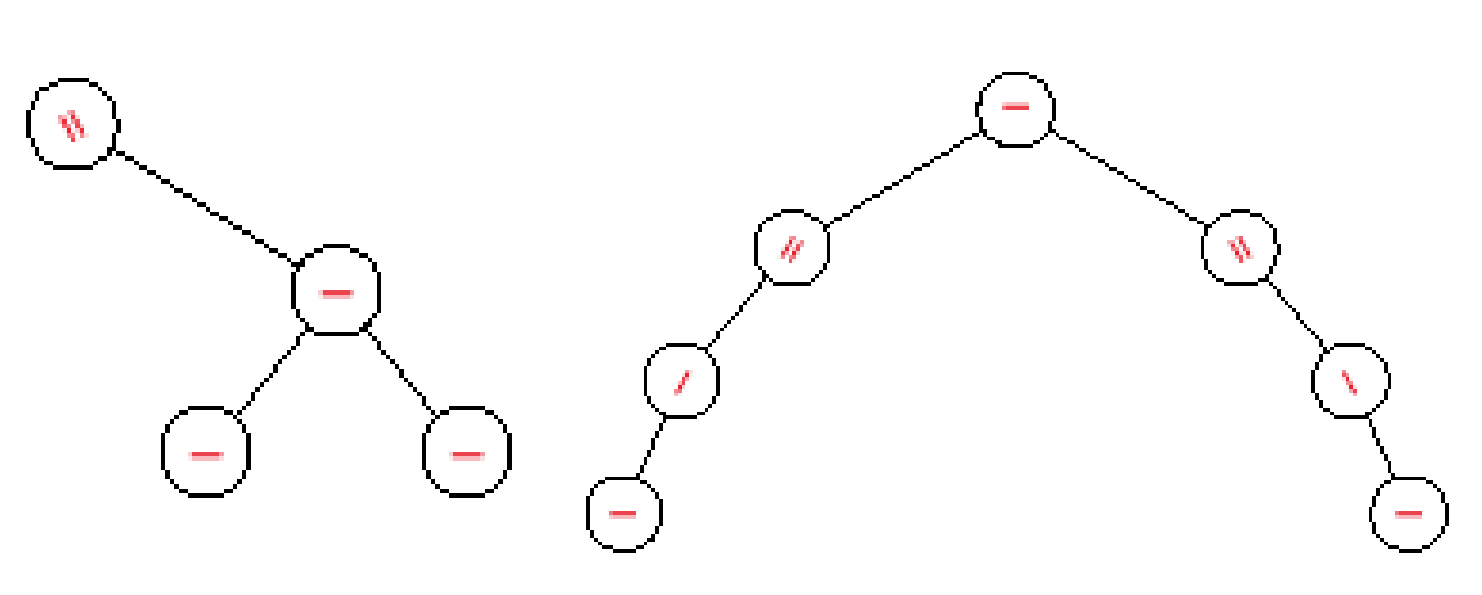
AVL Trees





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;
};
```

```
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;
}
```

C++ Conventions for AVL Trees

- 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.

- 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.

Dummy Methods and Virtual Methods

- 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.

- 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.

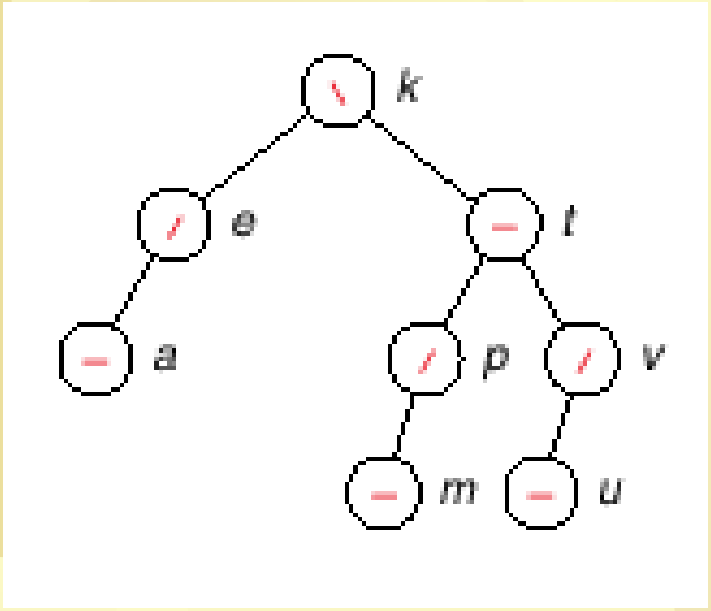
Class Declaration for AVL Trees

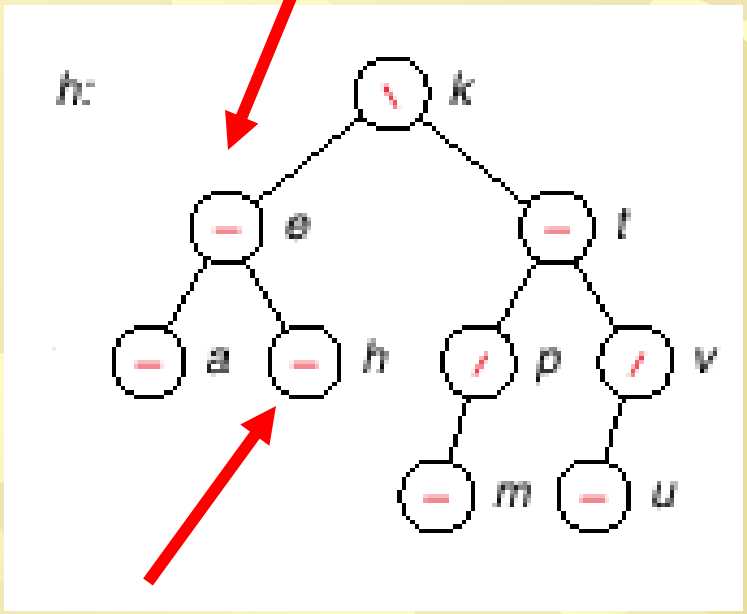
```
template <class Record> class AVL_tree:
public Search_tree<Record> {
public:
    Error_code insert(const Record
&new_data);
    Error_code remove(const Record
&old_data);

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

Class Declaration for AVL Trees

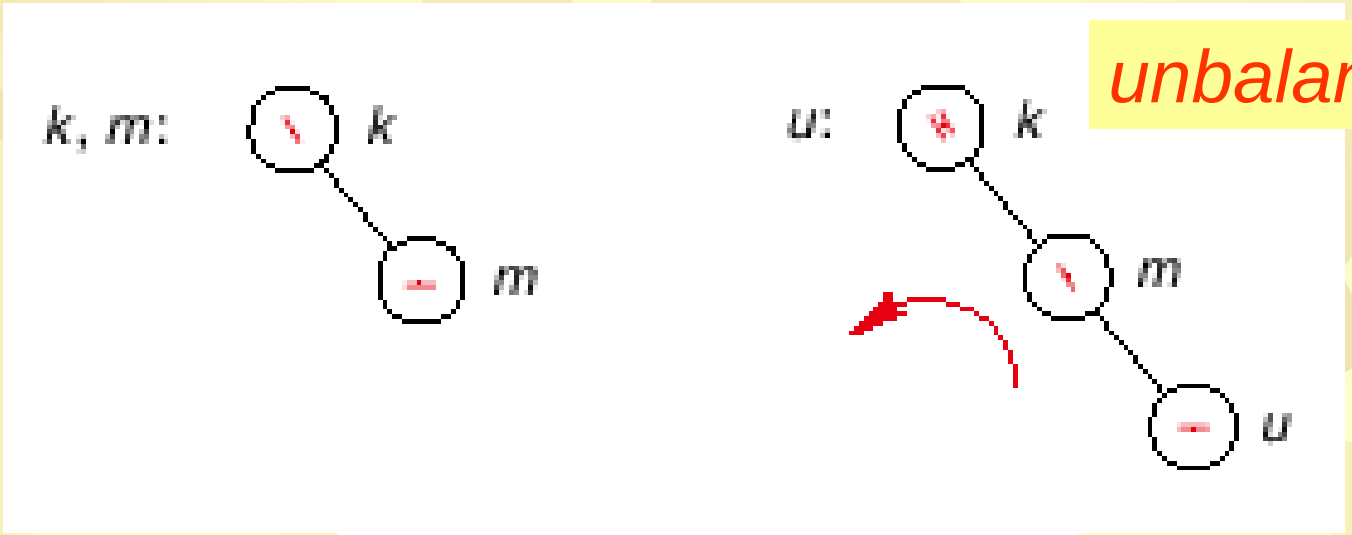
- The inherited data member of this class is the pointer root, which has type Binary_node <Record> * and thus can store the address of either an ordinary binary tree node or an AVL tree node.
- We must ensure that the overridden insert method only creates nodes of type AVL_node; doing so will guarantee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

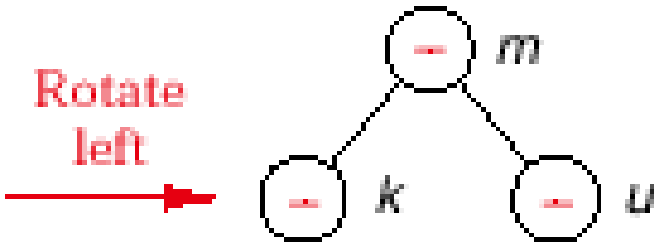
Insertions into an AVL tree



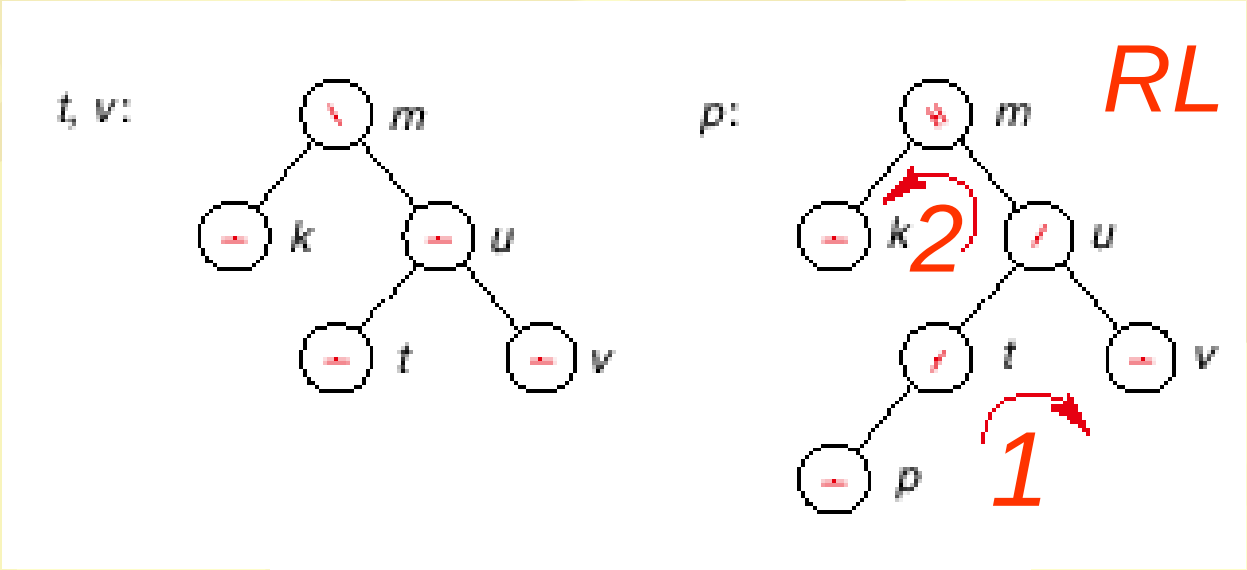


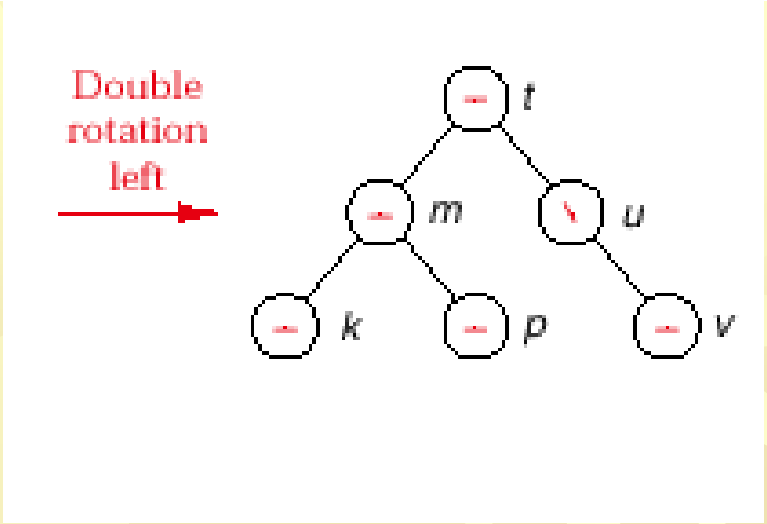
Rotate



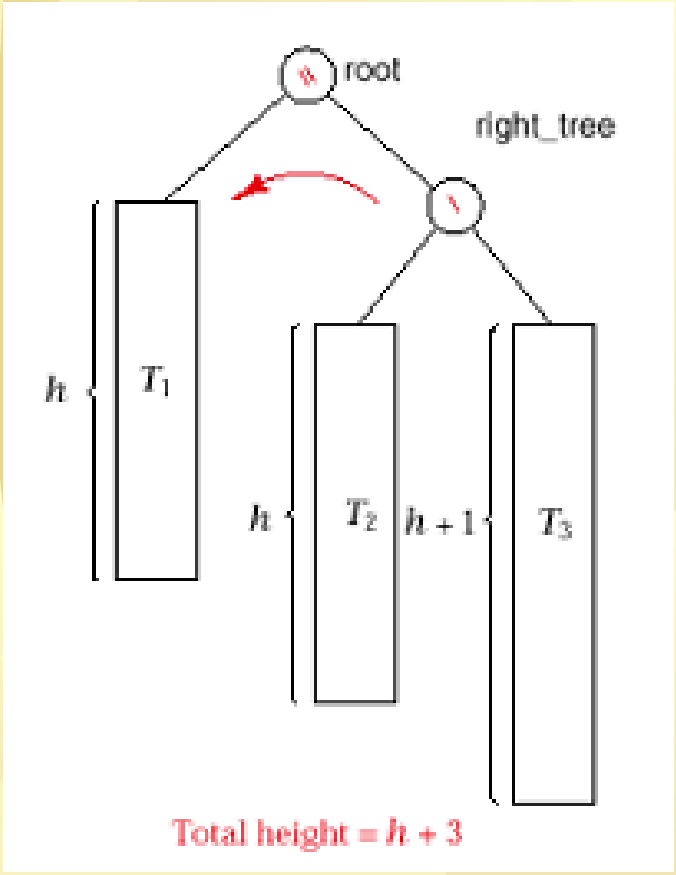


Double Rotate

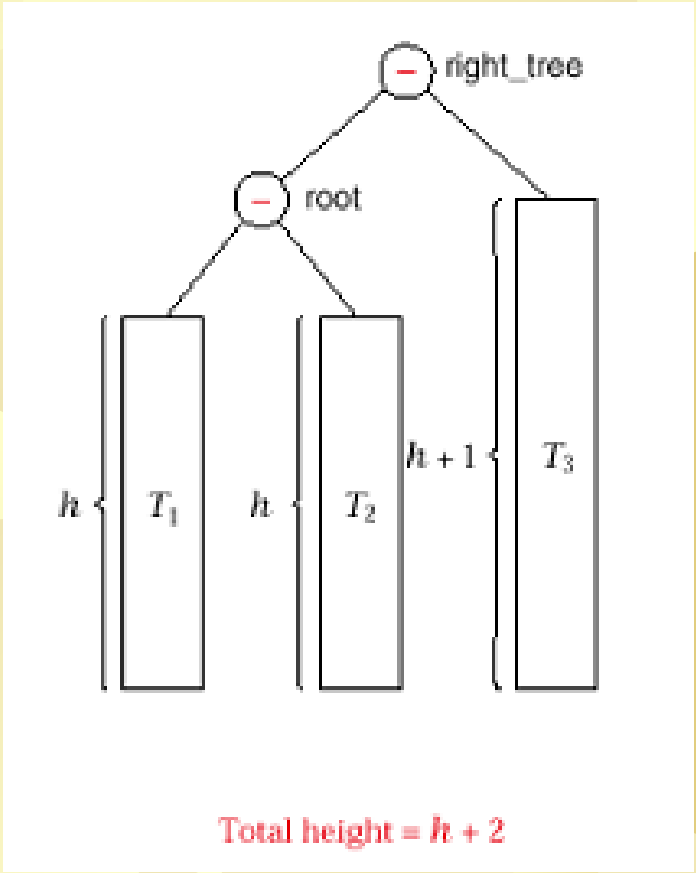




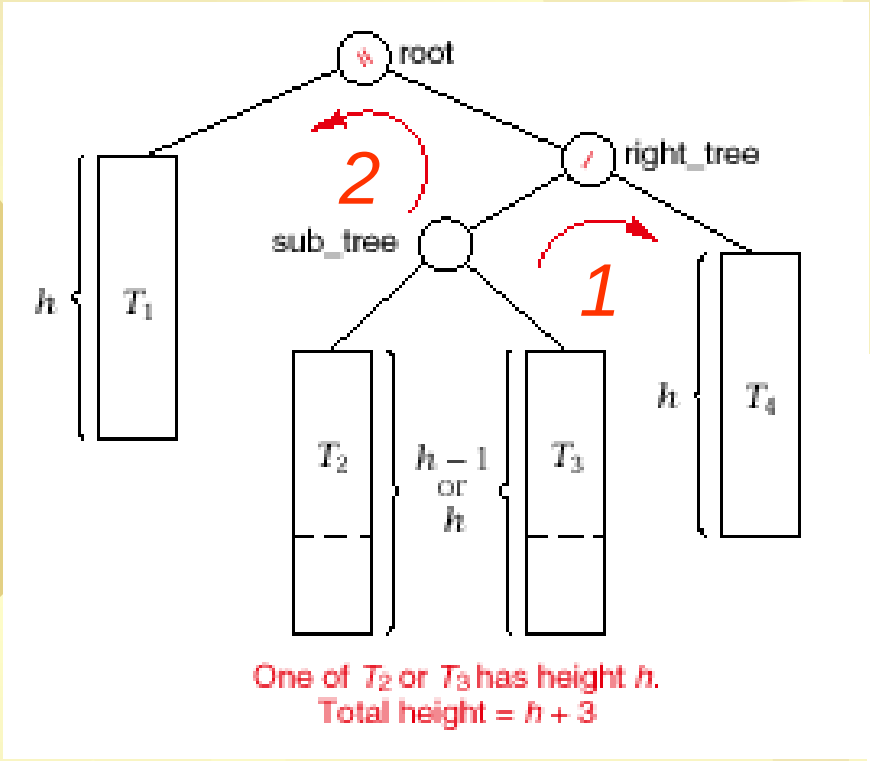
Rotate



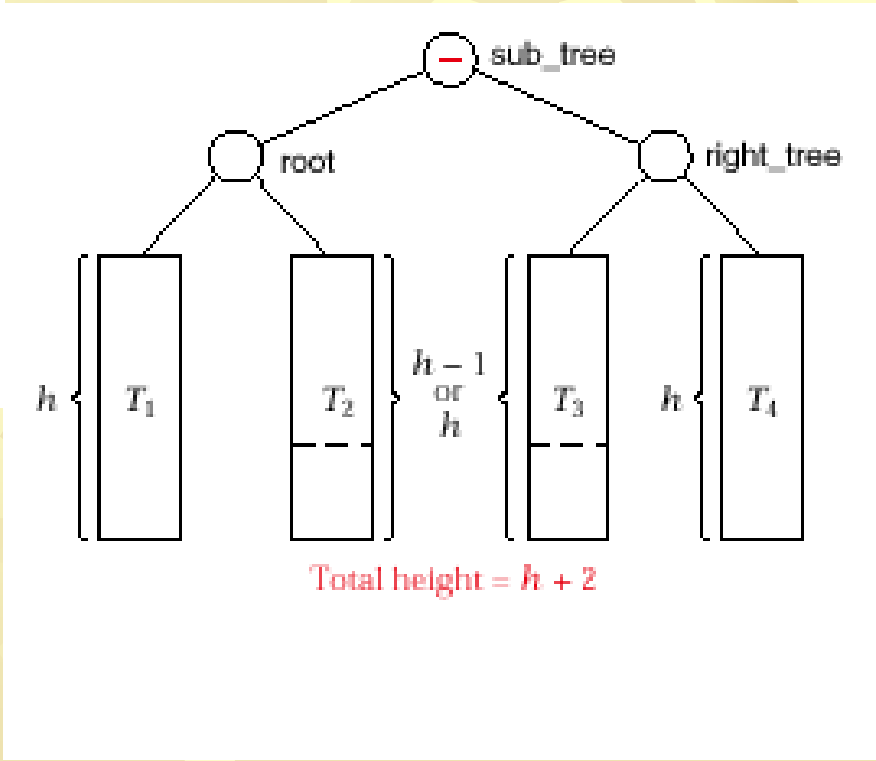
Rotate left



Double Rotation



become



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) { //Ins in lf stree.
    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;
    }
}
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

```
template <class Record> void AVL_tree
    <Record> ::right_balance(Binary_node<Reco
    rd> * &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;
```

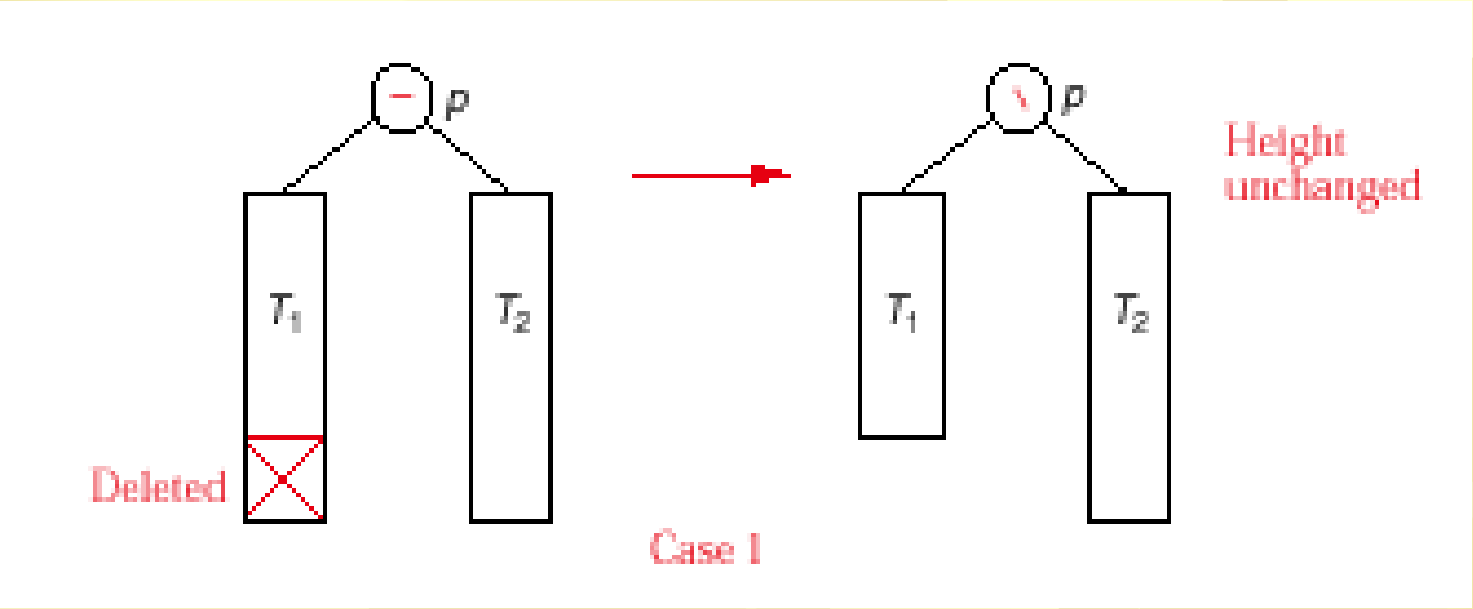
```
switch (sub_tree->get_balance( )) {
    case equal_height:
        sub_root->set_balance(equal_height);
        right_tree->set_balance(equal_height);
        break;
    case left_higher:
        sub_root->set_balance(equal_height);
        right_tree->set_balance(right_higher);
        break;
    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;
    }
}
```

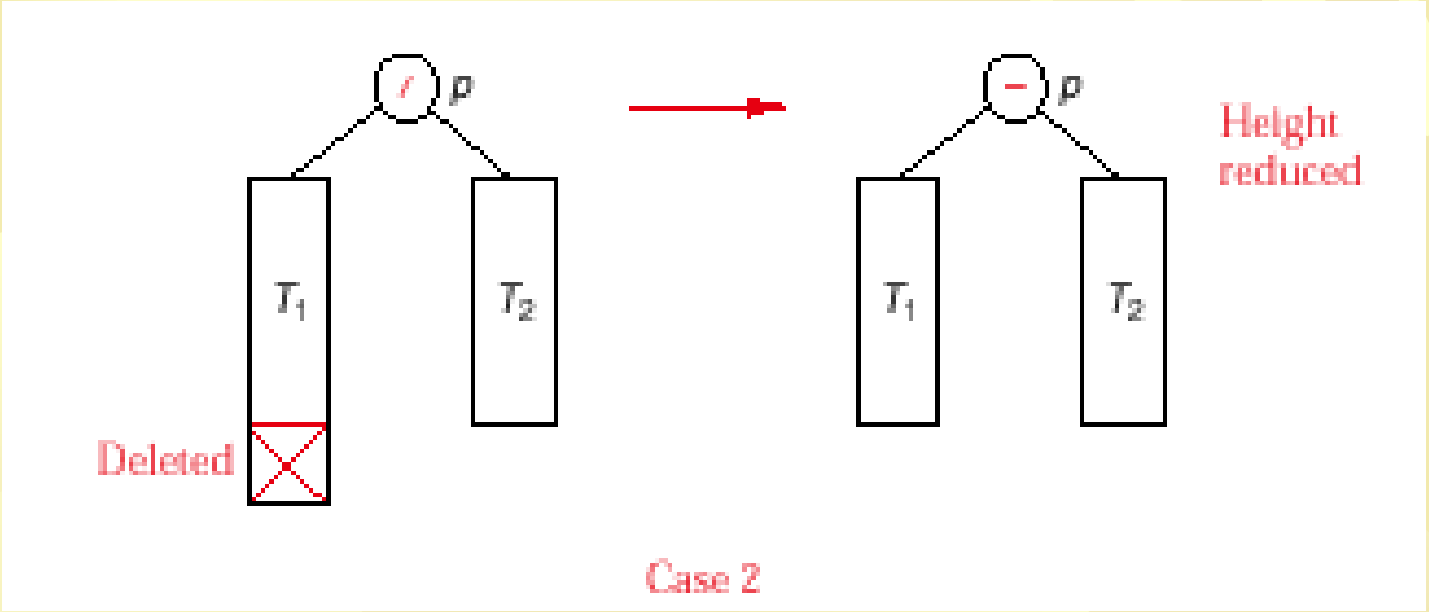
Removal of a Node

- Reduce the problem to the case when the node x to be removed has at most one child.
- Delete x. We use a bool variable shorter to show if the height of a subtree has been shortened.
- 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.

- **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.**

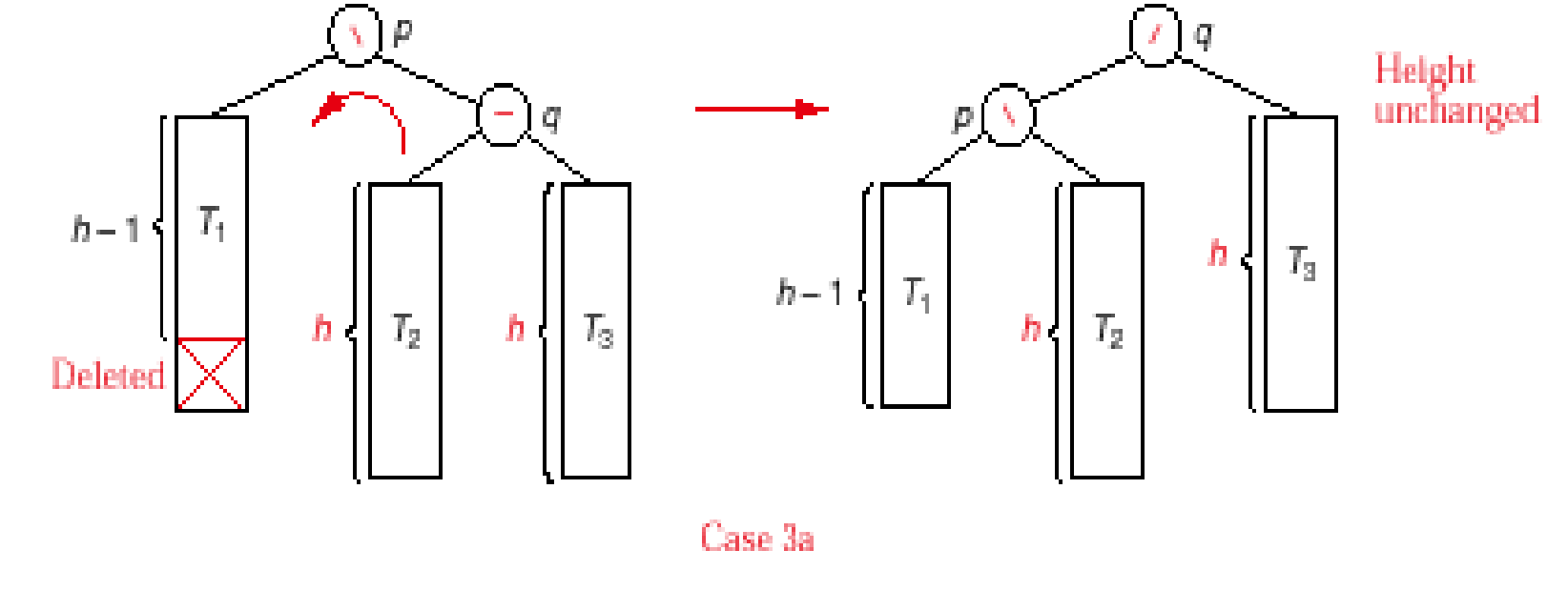


- **Case 2: The balance factor of p is not equal, and the taller subtree was shortened. Change the balance factor of p to equal, and leave shorter as true.**

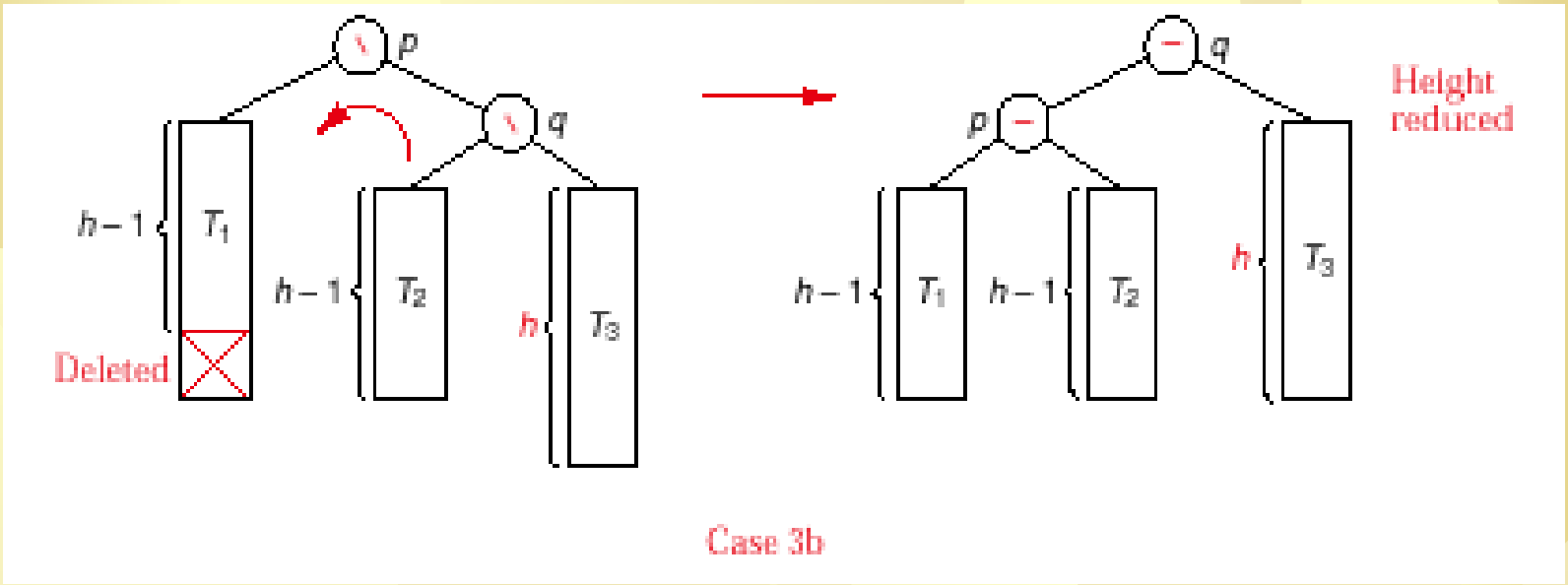


- **Case 3: The balance factor of p is not equal, and the shorter subtree was shortened. Apply a rotation as follows to restore balance. Let q be the root of the taller subtree of p.**

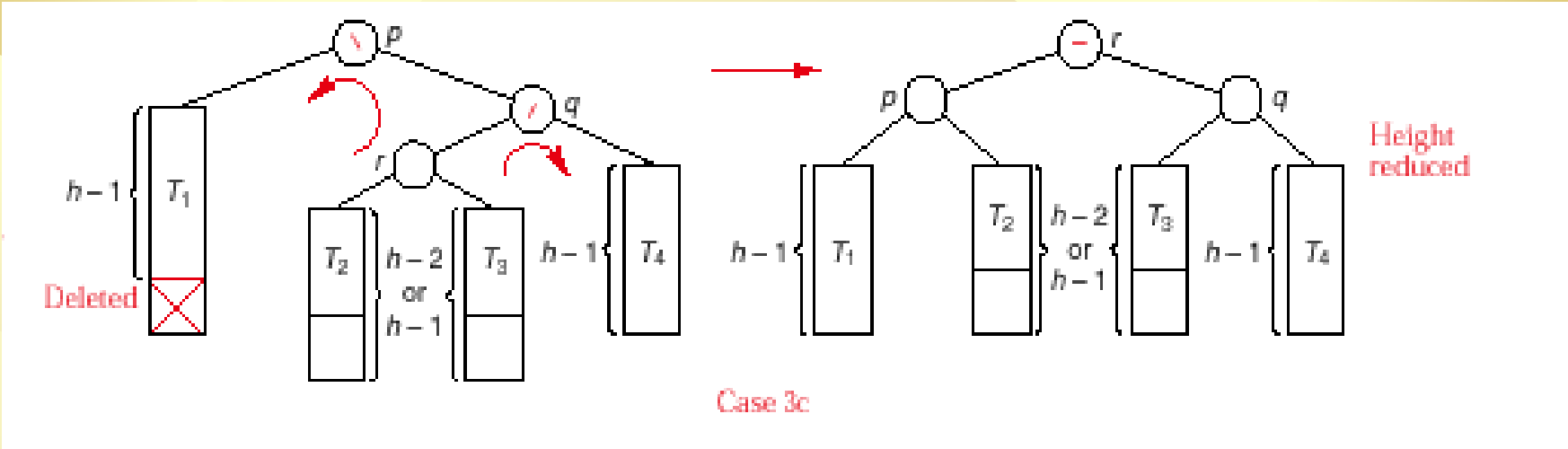
- **Case 3a: The balance factor of q is equal. A single rotation restores balance, and shorter becomes false.**



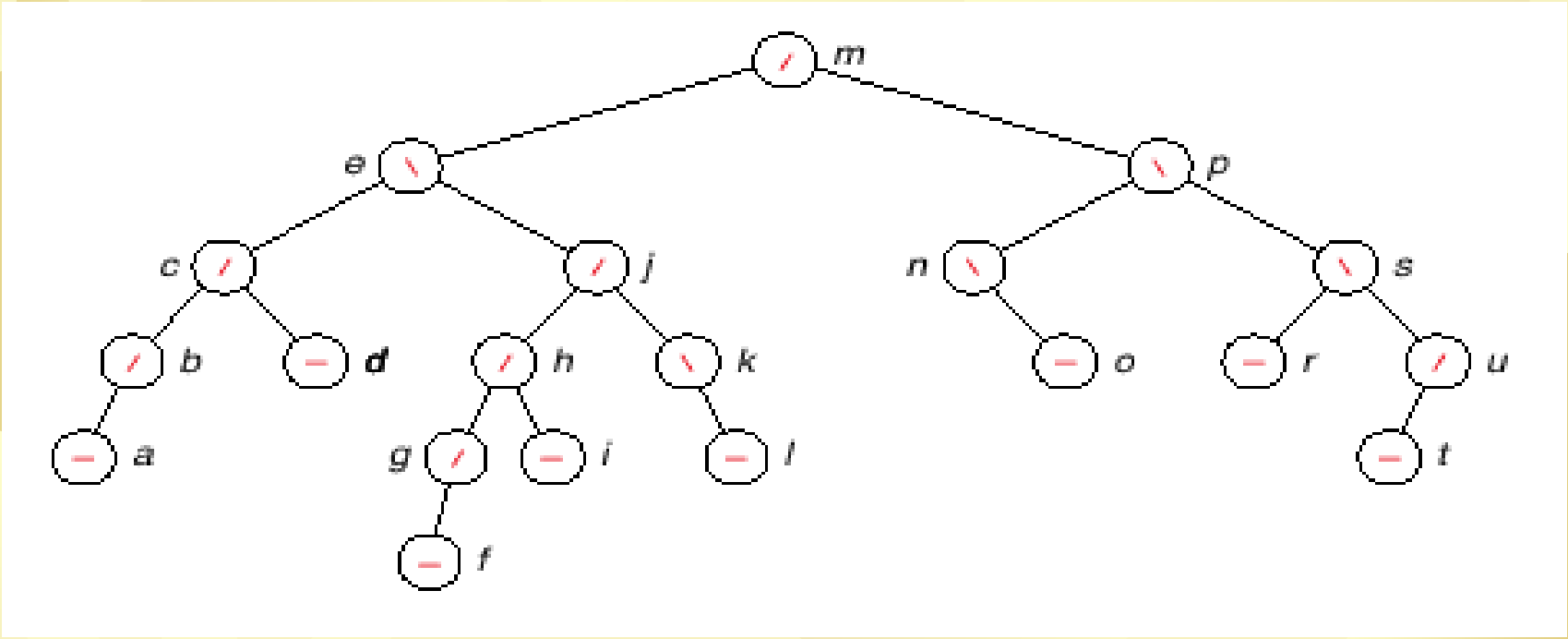
- **Case 3b: The balance factor of q is the same as that of p. Apply a single rotation, set the balance factors of p and q to equal, and leave shorter as true.**



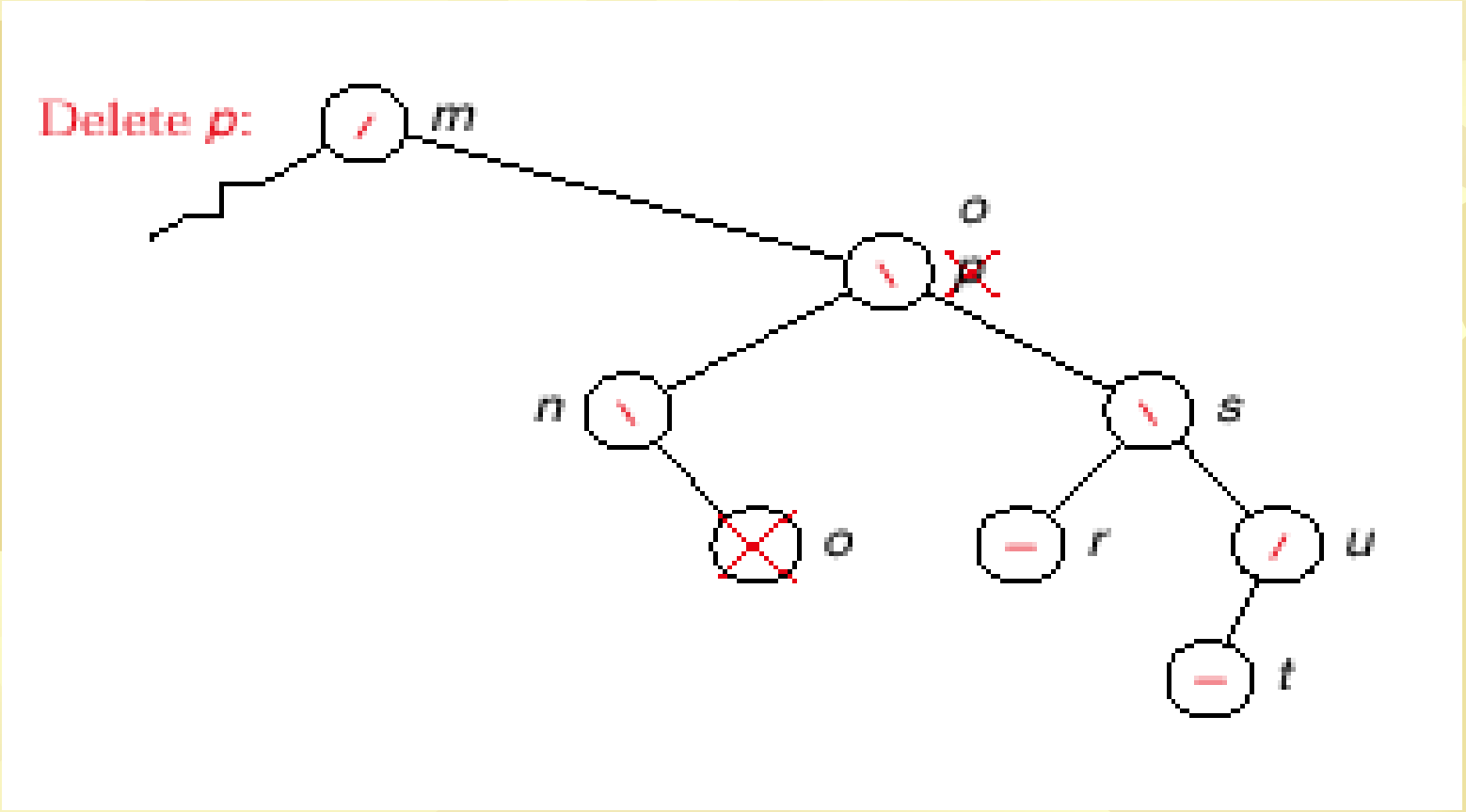
- **Case 3c: The balance factors of p and q are opposite. Apply a double rotation (first around q , then around p), set the balance factor of the new root to equal and the other balance factors as appropriate, and leave shorter as true.**



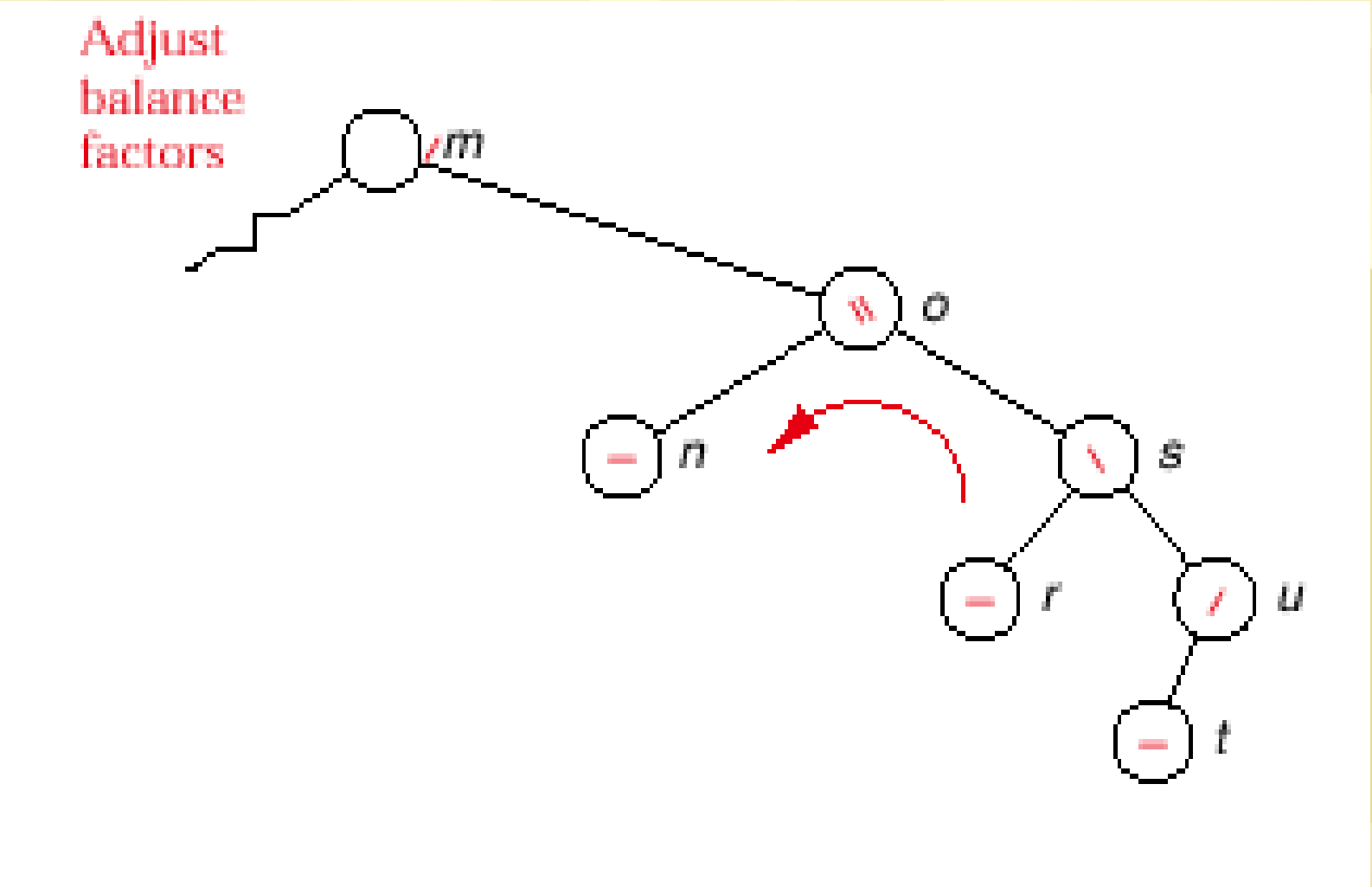
Example--init



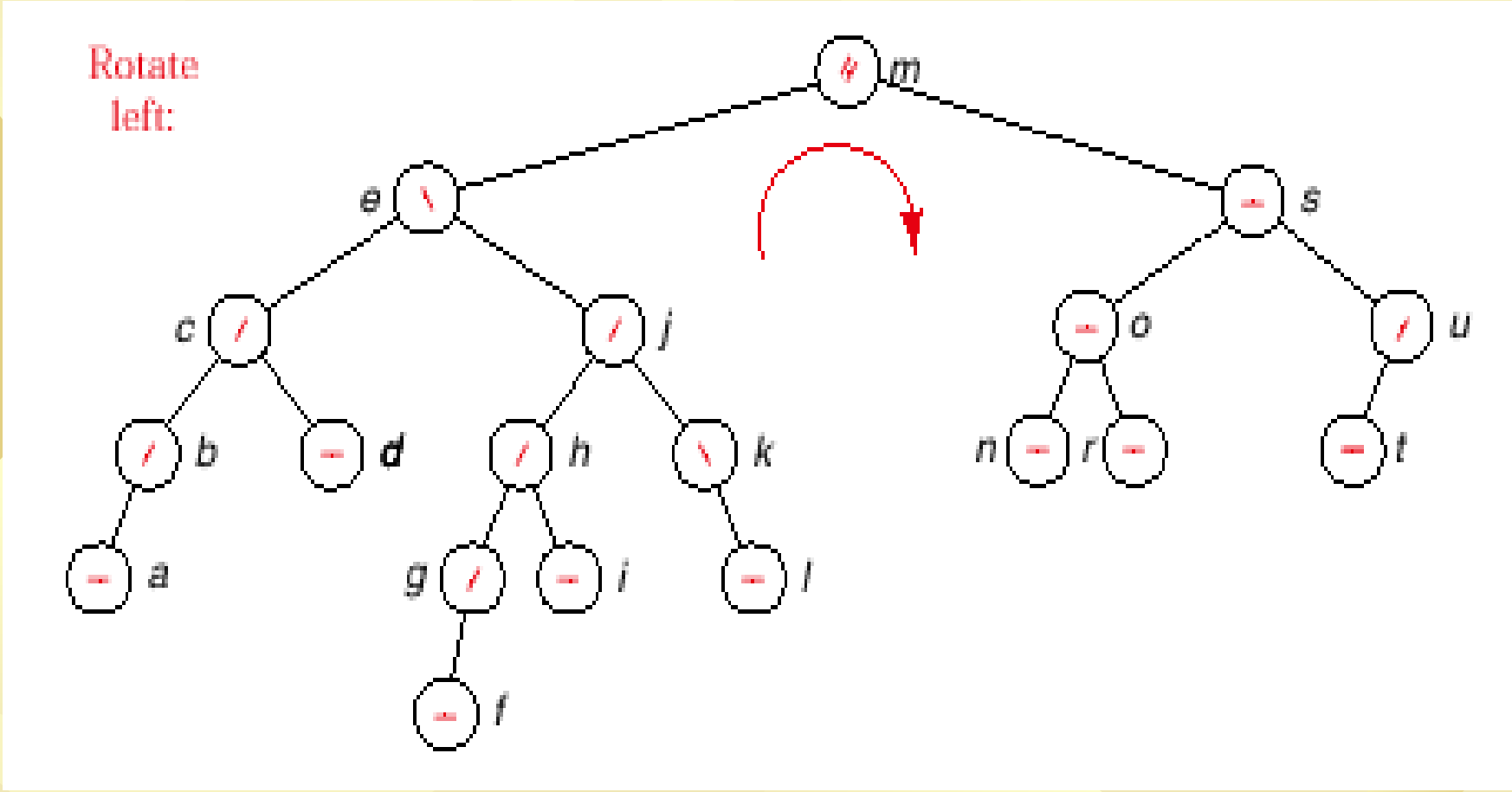
Delete p—replaced by o



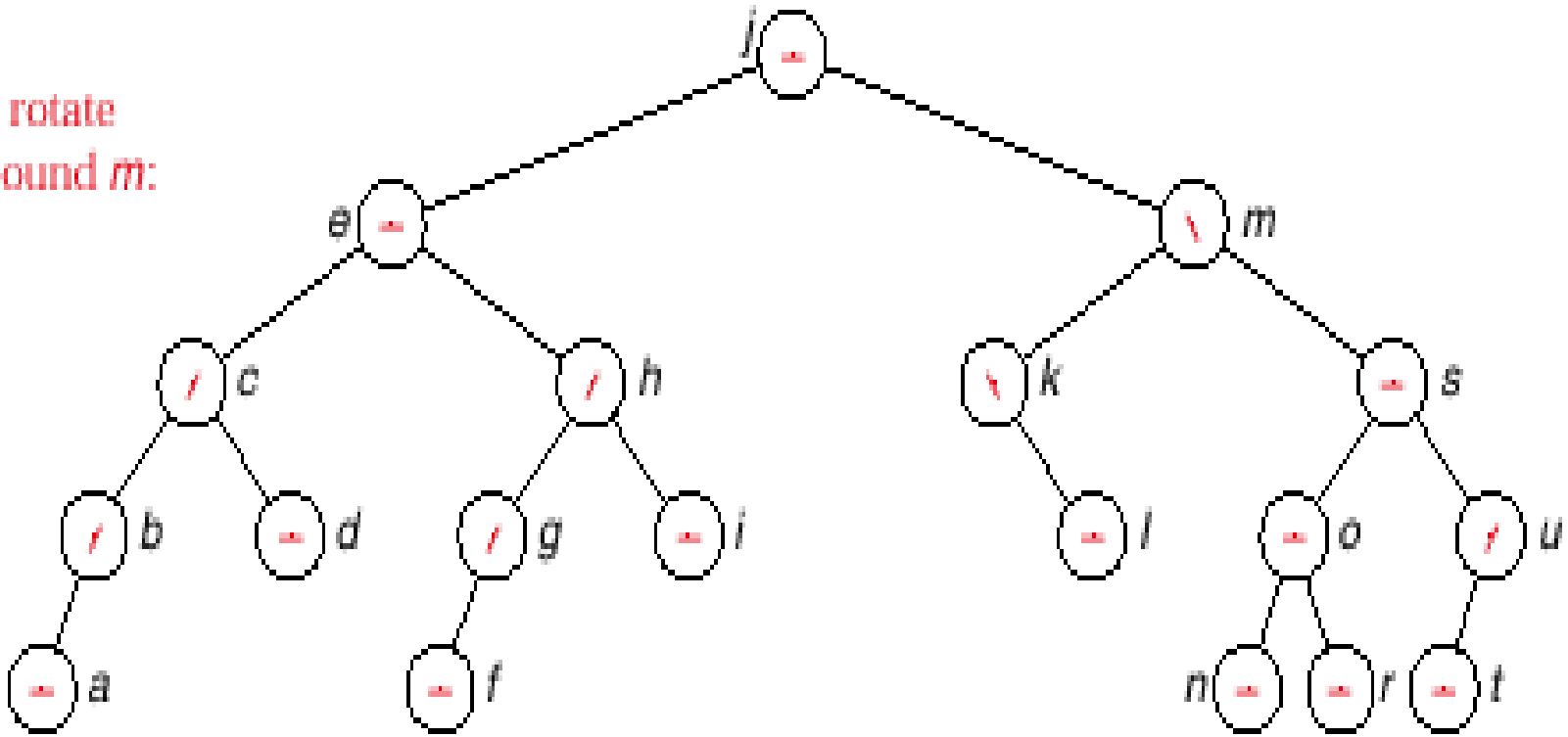
Adjust balance factors



Rotate



Double rotate
right around *m*:



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.

Worst-Case AVL Trees

- To find the maximum height of an AVL tree with n nodes, we instead find the minimum number of nodes that an AVL tree of height h can have.
- Let F_h be such a tree, with left and right subtrees F_l and F_r . Then one of F_l and F_r , say F_l , has height h - 1 and the minimum number of nodes in such a tree, and F_r has height h - 2 with the minimum number of nodes.
- These trees, as sparse as possible for AVL trees, are called Fibonacci trees.

Questions?