Chapter 10

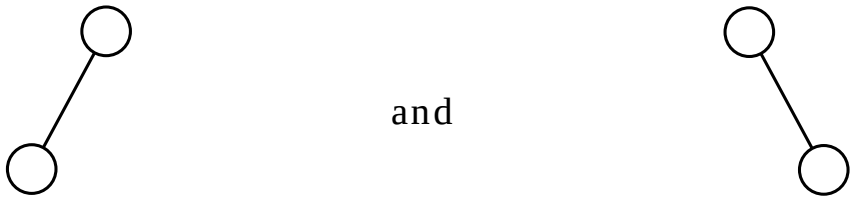
BINARY TREES

1. General Binary Trees
2. Binary Search Trees
3. Building a Binary Search Tree
4. Height Balance: AVL Trees
5. Splay Trees: A Self-Adjusting Data Structure

Binary Trees

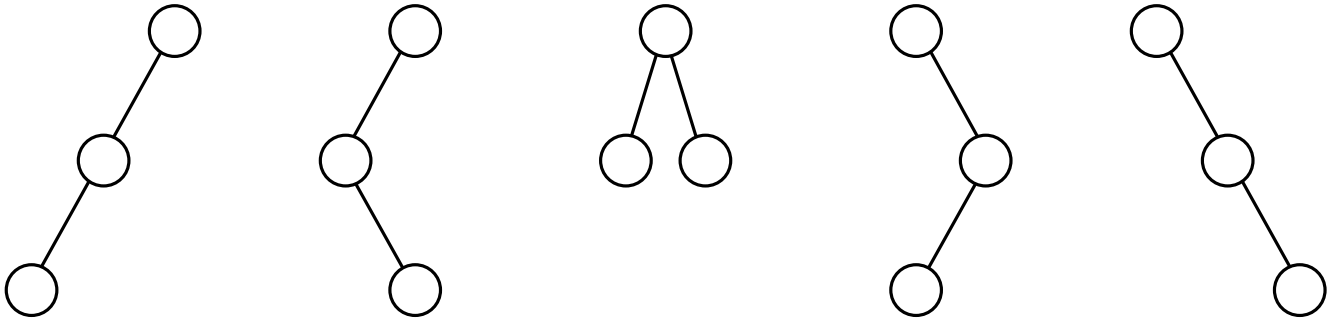
DEFINITION A ***binary tree*** is either empty, or it consists of a node called the ***root*** together with two binary trees called the ***left subtree*** and the ***right subtree*** of the root.

There is one empty binary tree, one binary tree with one node, and two with two nodes:



These are different from each other. We never draw any part of a binary tree to look like

The binary trees with three nodes are:



Traversal of Binary Trees

At a given node there are three tasks to do in some order: Visit the node itself (V); traverse its left subtree (L); traverse its right subtree (R). There are six ways to arrange these tasks:

$$V\,L\,R \qquad L\,V\,R \qquad L\,R\,V \qquad V\,R\,L \qquad R\,V\,L \qquad R\,L\,V.$$

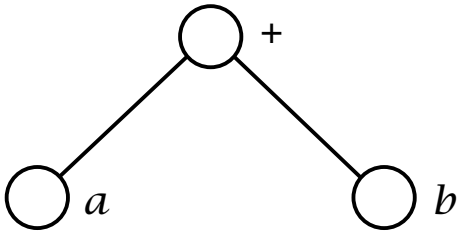
By standard convention, these are reduced to three by considering only the ways in which the left subtree is traversed before the right.

$V\,L\,R$	$L\,V\,R$	$L\,R\,V$
preorder	*inorder*	*postorder*

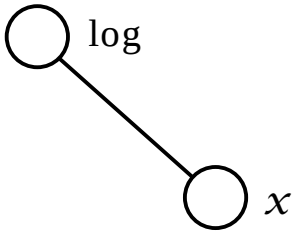
These three names are chosen according to the step at which the given node is visited.

- With ***preorder traversal*** we first visit a node, then traverse its left subtree, and then traverse its right subtree.
- With ***inorder traversal*** we first traverse the left subtree, then visit the node, and then traverse its right subtree.
- With ***postorder traversal*** we first traverse the left subtree, then traverse the right subtree, and finally visit the node.

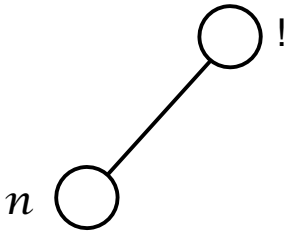
Expression Trees



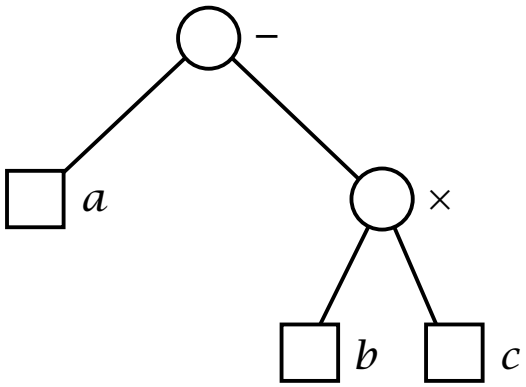
$a + b$



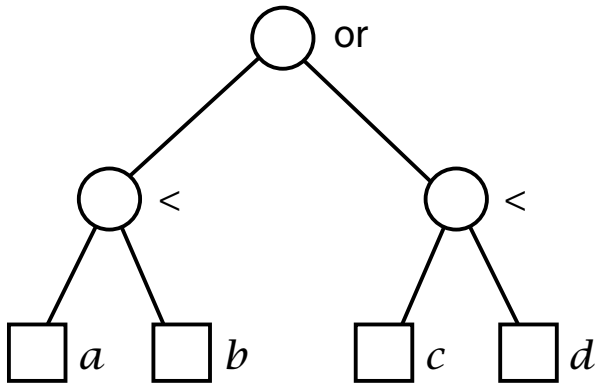
$\log x$



$n!$



$a - (b \times c)$



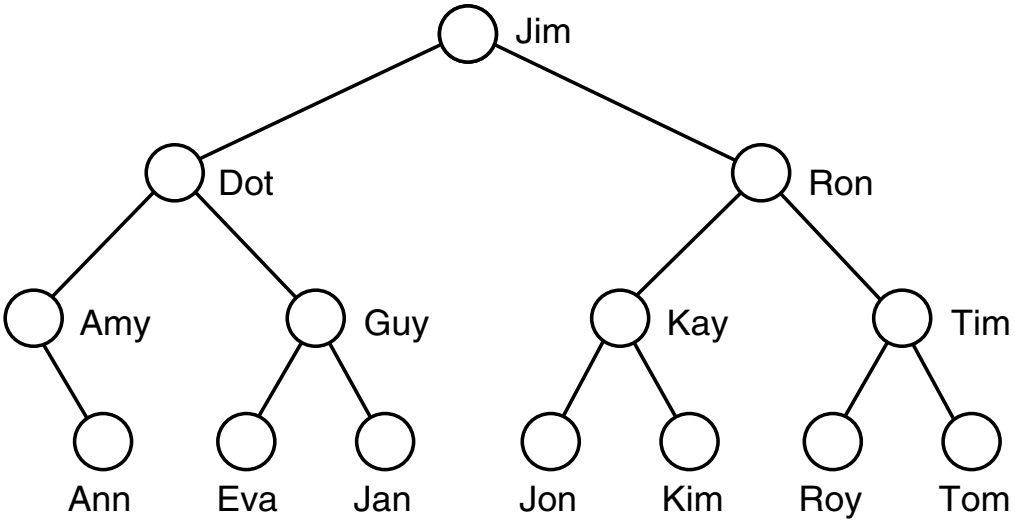
$(a < b)$ or $(c < d)$

Expression:	$a + b$	$\log x$	$n!$	$a - (b \times c)$	$(a < b)$ or $(c < d)$
Preorder :	$+\ a\ b$	$\log\ x$	$!\ n$	$-\ a\ \times\ b\ c$	or $<\ a\ b\ <\ c\ d$
Inorder :	$a\ +\ b$	$\log\ x$	$n\ !$	$a\ -\ b\ \times\ c$	$a\ <\ b$ or $c\ <\ d$
Postorder :	$a\ b\ +$	$x\ \log$	$n\ !$	$a\ b\ c\ \times\ -$	$a\ b\ <\ c\ d\ <$ or

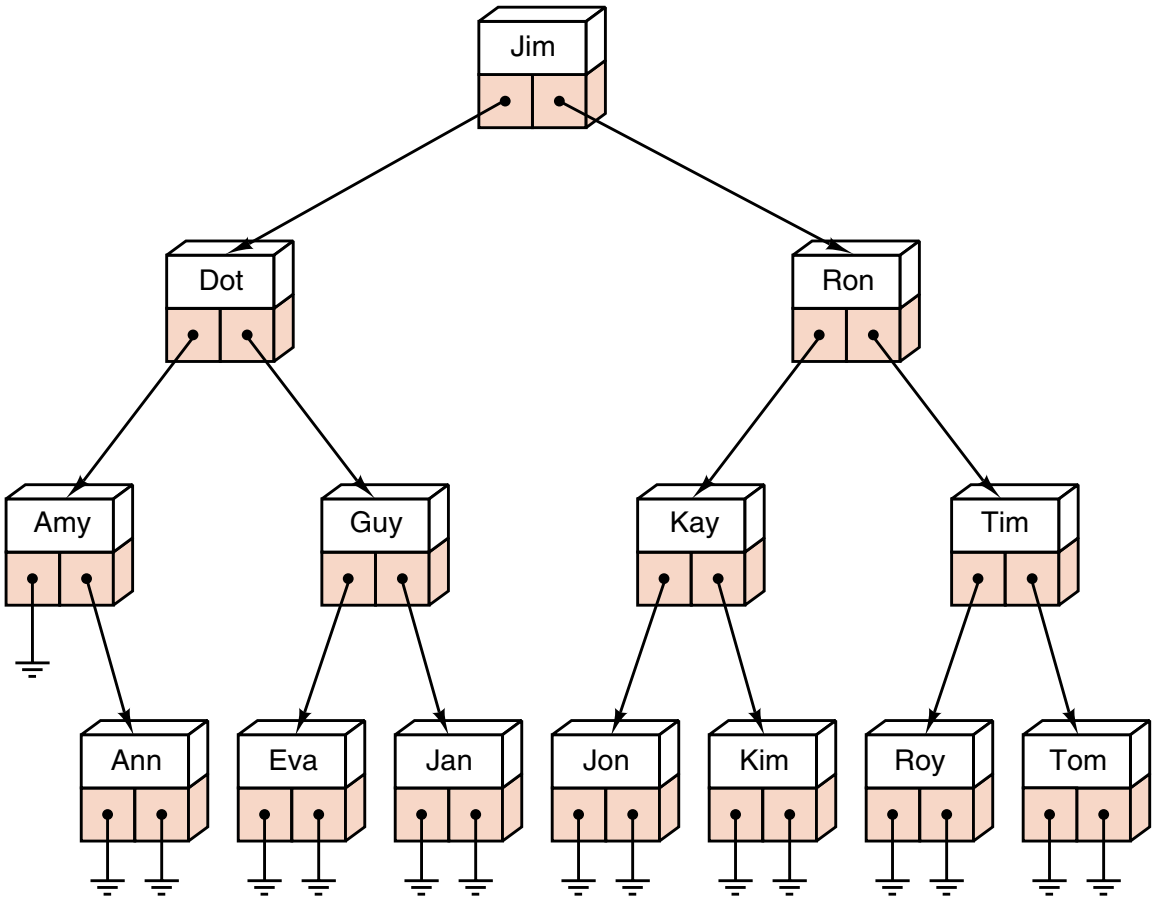
$$x := (-b + (b \uparrow 2 - 4 \times a \times c) \uparrow 0.5)/(2 \times a)$$

Linked Binary Trees

Comparison tree:



Linked implementation of binary tree:



Linked Binary Tree Specifications

Binary tree class:

```
template <class Entry>
class Binary_tree {
public:
//    Add methods here.
protected:
   //    Add auxiliary function prototypes here.
   Binary_node<Entry> *root;
};
```

Binary node class:

```
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);

};
```

Constructor:

```
template <class Entry>
Binary_tree<Entry> :: Binary_tree( )
/* Post:  An empty binary tree has been created.  */
{
   root = NULL;
}
```

Empty:

```
template <class Entry>
bool Binary_tree<Entry> :: empty( ) const
/* Post:  A result of true is returned if the binary tree is empty. Otherwise, false is
          returned.  */
{
   return root ==  NULL;
}
```

Inorder traversal:

```
template <class Entry>
void Binary_tree<Entry> :: inorder(void (*visit)(Entry &))
/* Post:  The tree has been been traversed in infix order sequence.
   Uses:  The function recursive_inorder  */
{
   recursive_inorder(root, visit);
}
```

Most Binary_tree methods described by recursive processes can be implemented by calling an auxiliary recursive function that applies to subtrees.

```
template <class Entry>
void Binary_tree<Entry> ::
      recursive_inorder(Binary_node<Entry> *sub_root,
                        void (*visit)(Entry &))
/* Pre:   sub_root is either NULL or points to a subtree of the Binary_tree.
   Post:  The subtree has been traversed in inorder sequence.
   Uses:  The function recursive_inorder recursively  */
{
   if (sub_root != NULL) {
      recursive_inorder(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_inorder(sub_root->right, visit);
   }
}
```

Binary Tree Class Specification

```
template <class Entry>
class Binary_tree {
public:
   Binary_tree( );
   bool empty( ) const;
   void preorder(void (*visit)(Entry &));
   void inorder(void (*visit)(Entry &));
   void postorder(void (*visit)(Entry &));


   int size( ) const;
   void clear( );
   int height( ) const;
   void insert(const Entry &);


   Binary_tree (const Binary_tree<Entry> &original);
   Binary_tree & operator = (const Binary_tree<Entry> &original);
    ~Binary_tree( );
protected:
   //    Add auxiliary function prototypes here.
   Binary_node<Entry> *root;
};
```

Binary Search Trees

Can we find an implementation for ordered lists in which we can search quickly (as with binary search on a contiguous list) and in which we can make insertions and deletions quickly (as with a linked list)?

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:

1. The key of the left child of a node (if it exists) is less than the key of its parent node.
2. The key of the right child of a node (if it exists) is greater than the key of its parent node.
3. The left and right subtrees of the root are again binary search trees.

We always require:
No two entries in a binary search tree may have equal keys.

- We can regard binary search trees as a new ADT.
- We may regard binary search trees as a specialization of binary trees.
- We may study binary search trees as a new implementation of the ADT *ordered list*.

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 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.

- The class Record has the behavior outlined in Chapter 7: Each Record is associated with a Key. The keys can be compared with the usual comparison operators. By casting records to their corresponding keys, the comparison operators apply to records as well as to keys.

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.

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

```
Binary_node<Record> *Search_tree<Record>::search_for_node(
   Binary_node<Record>* sub_root, const Record &target) const;
```

Pre: sub_root is NULL or points to a subtree of a Search_tree

Post: If the key of target is not in the subtree, a result of NULL is returned. Otherwise, a pointer to the subtree node containing the target is returned.

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);
}
```

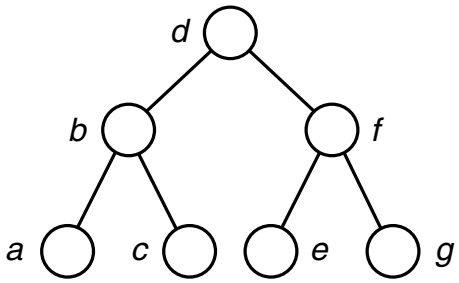
Nonrecursive version:

```
template <class Record>
Binary_node<Record> *Search_tree<Record>::search_for_node(
  Binary_node<Record> *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;
}
```

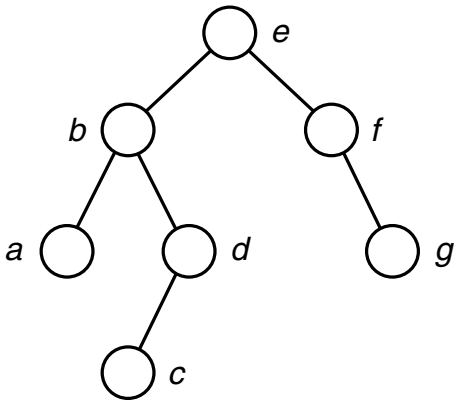
Public method for tree search:

```
template <class Record>
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.
   Uses:  function search_for_node  */
{
   Error_code result = success;
   Binary_node<Record> *found = search_for_node(root, target);
   if (found ==  NULL)
      result = not_present;
   else
      target = found->data;
   return result;
}
```

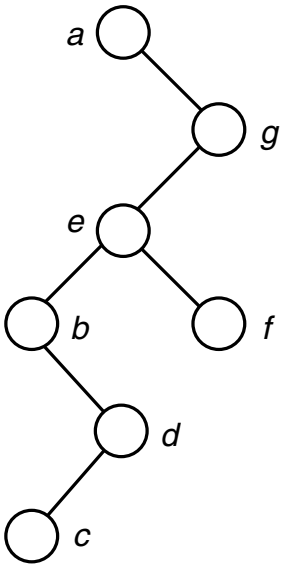
Binary Search Trees with the Same Keys



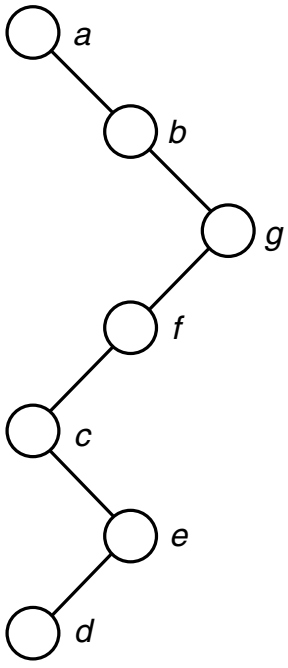
(a)



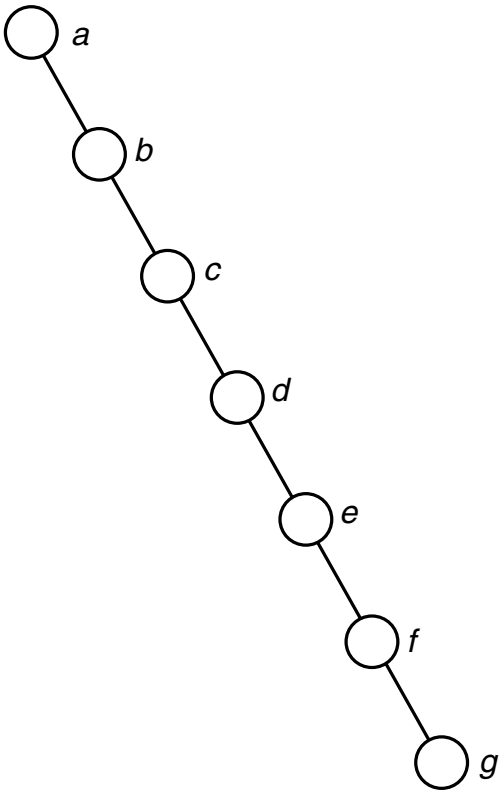
(b)



(c)



(d)



(e)

Analysis of Tree Search

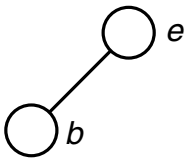
- Draw the comparison tree for a binary search (on an ordered list). Binary search on the list does exactly the same comparisons as tree_search will do if it is applied to the comparison tree. By Section 7.4, binary search performs $O(\log n)$ comparisons for a list of length n. This performance is excellent in comparison to other methods, since $\log n$ grows very slowly as n increases.
- 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.
- If the tree degenerates into a long chain, then tree search becomes the same as sequential search, doing $\Theta(n)$ 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.
- 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

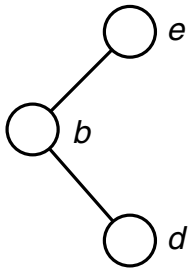
```
Error_code Search_tree<Record> ::
        insert(const Record &new_data);
```

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.

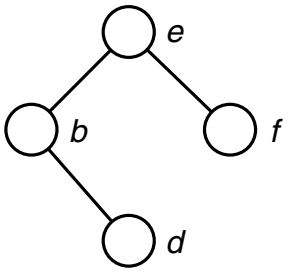
(a) Insert *e*



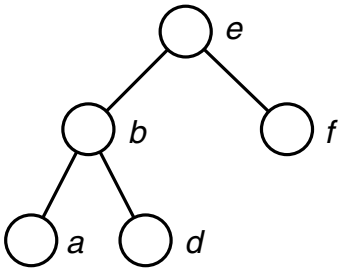
(b) Insert *b*



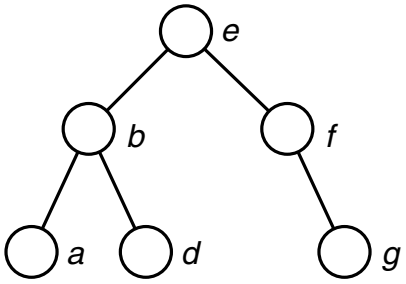
(c) Insert *d*



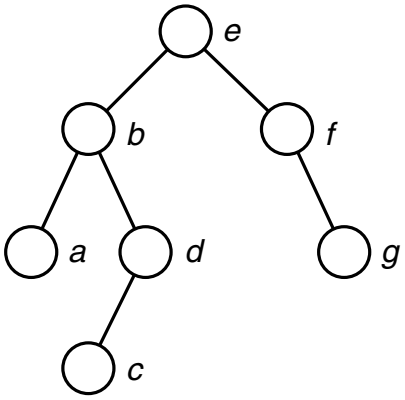
(d) Insert *f*



(e) Insert *a*



(f) Insert *g*



(g) Insert *c*

Method for Insertion

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

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;
}
```

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.

Theorem 10.1 Treesort makes exactly the same comparisons of keys as does quicksort when the pivot for each sublist is chosen to be the first key in the sublist.

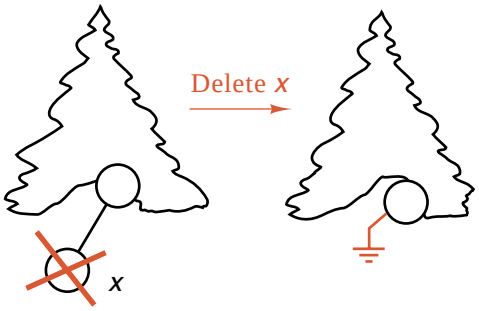
Corollary 10.2 In the average case, on a randomly ordered list of length n, treesort performs

$$2n \ln n + O(n) \approx 1.39n \lg n + O(n)$$

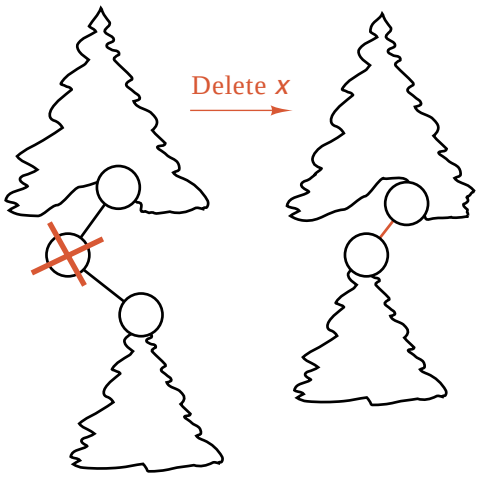
comparisons of keys.

- First advantage of treesort over quicksort: 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: 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.

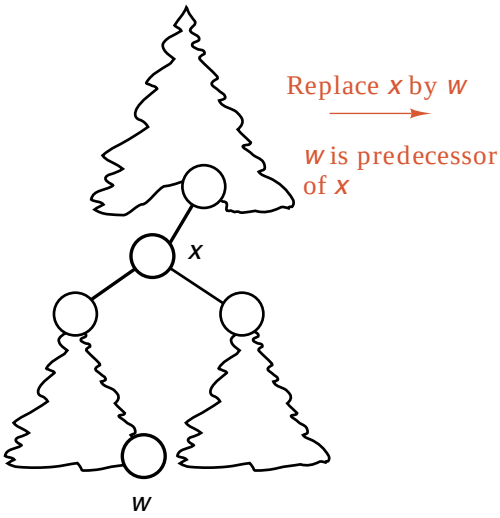
Removal from a Binary Search Tree

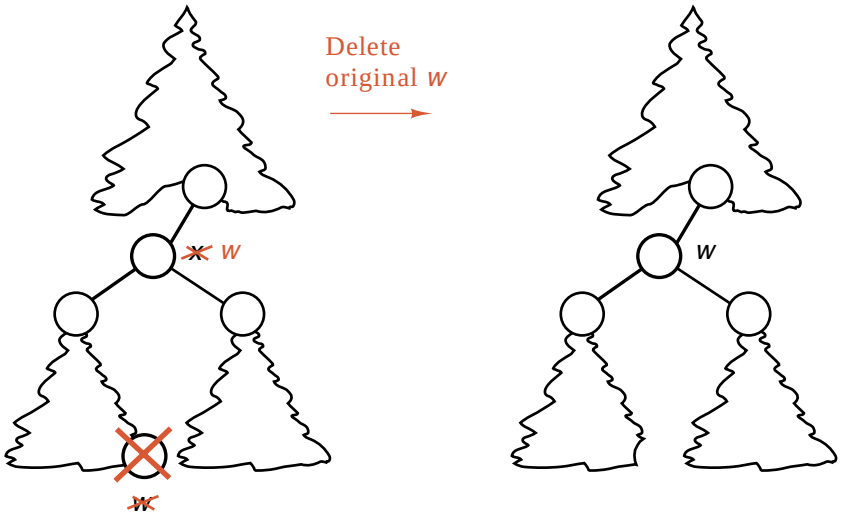


Case: deletion of a leaf



Case: one subtree empty





Case: neither subtree empty

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. */
{
  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) sub_root = sub_root->left;
  else if (sub_root->left == NULL) sub_root = sub_root->right;

  else {                                //  Neither subtree is empty.
    to_delete = sub_root->left;   //  Move left to find predecessor.
    Binary_node<Record> *parent = sub_root;   //  parent of to_delete
    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;
}
```

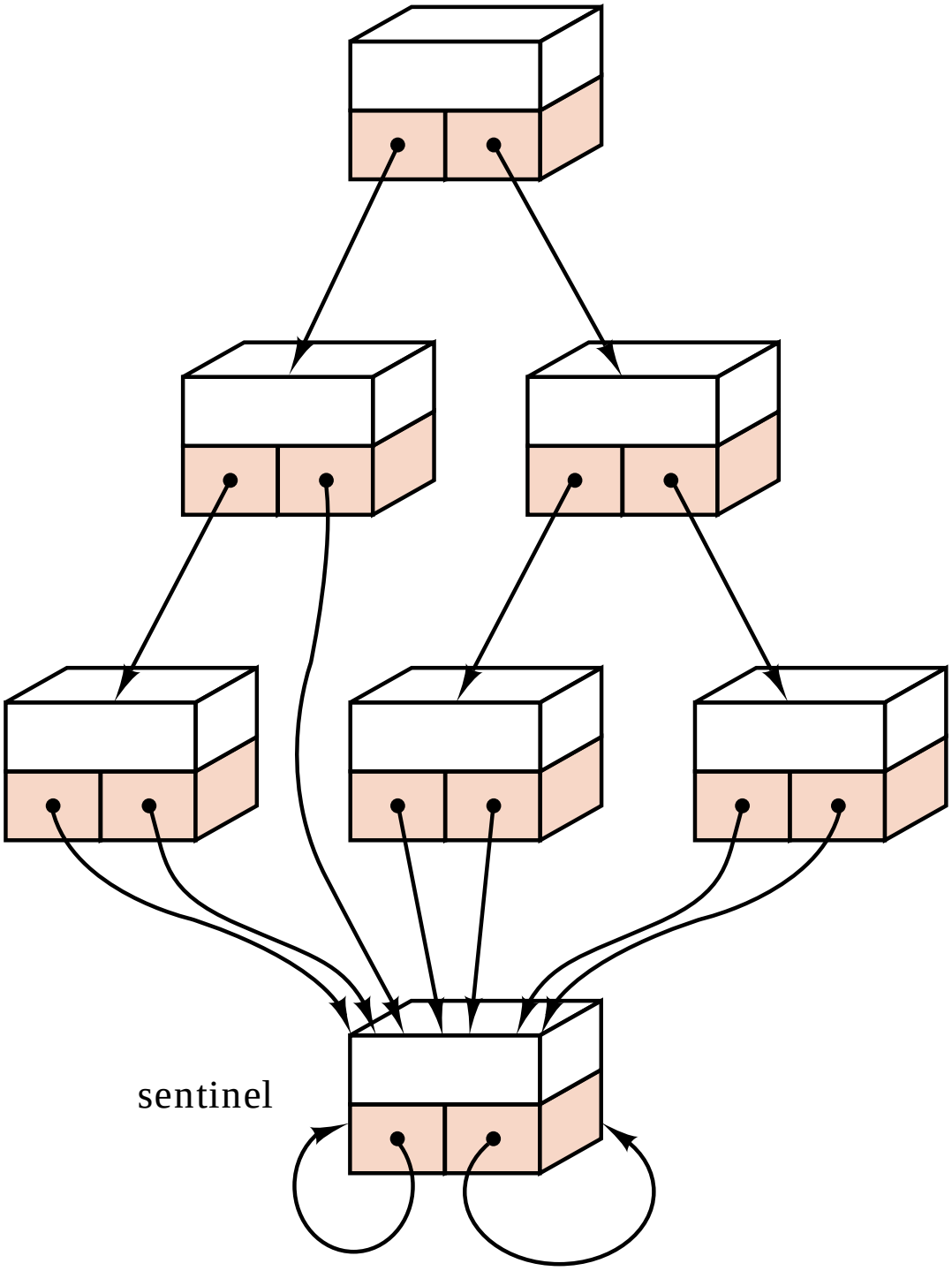
Removal Method

```
template <class Record>
Error_code Search_tree<Record> :: remove(const Record &target)
/* Post:  If a Record with a key matching that of target belongs to the Search_tree a
          code of success is returned and the corresponding node is removed from the
          tree. Otherwise, a code of not_present is returned.
   Uses:  Function search_and_destroy  */

{
  return search_and_destroy(root, target);
}
```

As usual, this method uses an auxiliary recursive function that refers to the actual nodes in the tree.

```
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);
}
```

Linked Binary Tree with Sentinel



Information Retrieval Project

- Purpose of project: Compare several different kinds of binary search trees useful for information retrieval. The current, first part of the project is to produce a driver program and the information-retrieval package using ordinary binary search trees.
- Outline of project:

  1. Create the data structure (binary search tree).
  2. Ask the user for the name of a text file and open it to read.
  3. Read the file, split it apart into individual words, and insert the words into the data structure. With each word will be kept a frequency count and, when duplicate words are encountered, the frequency count will be increased. The same word will not be inserted twice in the tree.
  4. Print the number of comparisons done and the CPU time used in part 3.
  5. If the user wishes, print out all the words in the data structure, in alphabetical order, with their frequency counts.
  6. Put everything in parts 2–5 into a **do** . . . **while** loop that will run as many times as the user wishes. Thus the user can build the data structure with more than one file if desired. By reading the same file twice, the user can compare time for retrieval with the time for the original insertion.

Further Specifications, Information Retrieval

- The input to the driver will be a file. The program will be executed with several different files; the name of the file to be used should be requested from the user while the program is running.

- A word is defined as a sequence of letters, together with apostrophes (') and hyphens (-), provided that the apostrophe or hyphen is both immediately preceded and followed by a letter. Uppercase and lowercase letters should be regarded as the same (by translating all letters into either uppercase or lowercase, as you prefer). A word is to be truncated to its first 20 characters (that is, only 20 characters are to be stored in the data structure) but words longer than 20 characters may appear in the text. Nonalphabetic characters (such as digits, blanks, punctuation marks, control characters) may appear in the text file. The appearance of any of these terminates a word, and the next word begins only when a letter appears.

- Be sure to write your driver so that it will not be changed at all when you change implementation of data structures later.

Function Specifications, Information Retrieval

```
void update(const String &word,
            Search_tree<Record> &structure, int &num_comps);
```

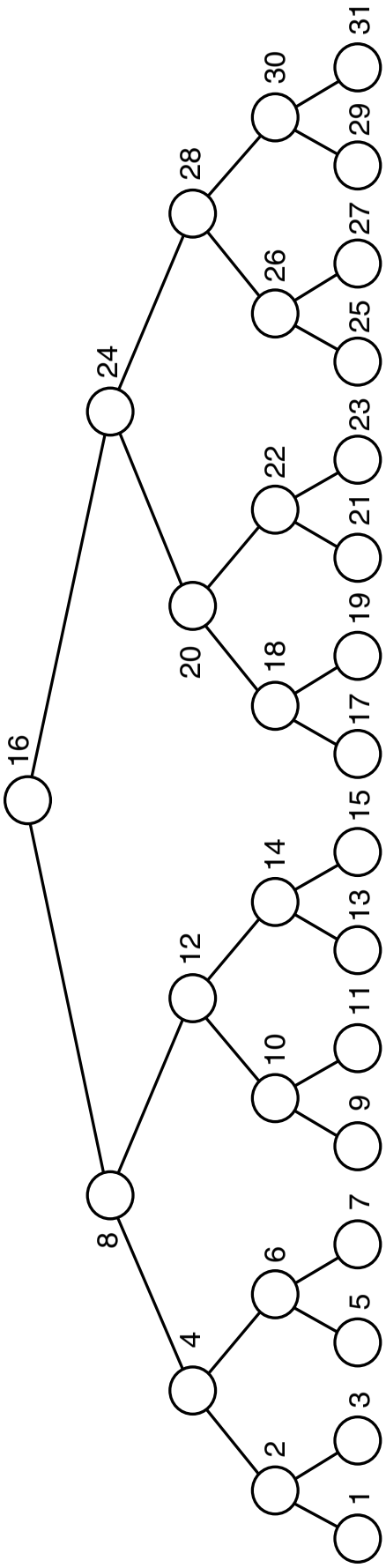
Post: If word was not already present in structure, then word has been inserted into structure and its frequency count is 1. If word was already present in structure, then its frequency count has been increased by 1. The variable parameter num_comps is set to the number of comparisons of words done.

```
void print(const Search_tree<Record> &structure);
```

Post: All words in structure are printed at the terminal in alphabetical order together with their frequency counts.

```
void write_method( );
```

Post: The function has written a short string identifying the abstract data type used for structure.

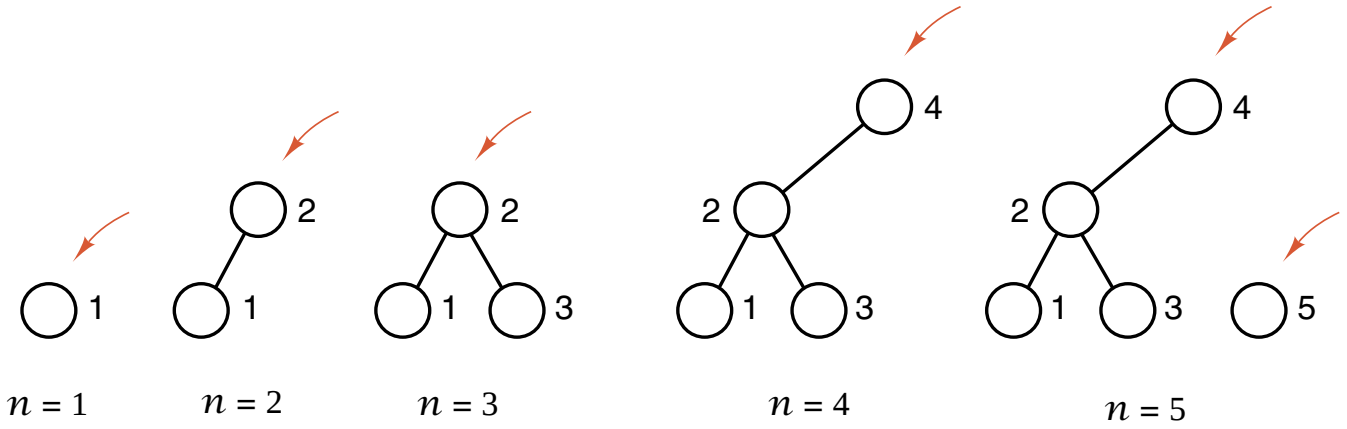


Complete binary tree with 31 nodes

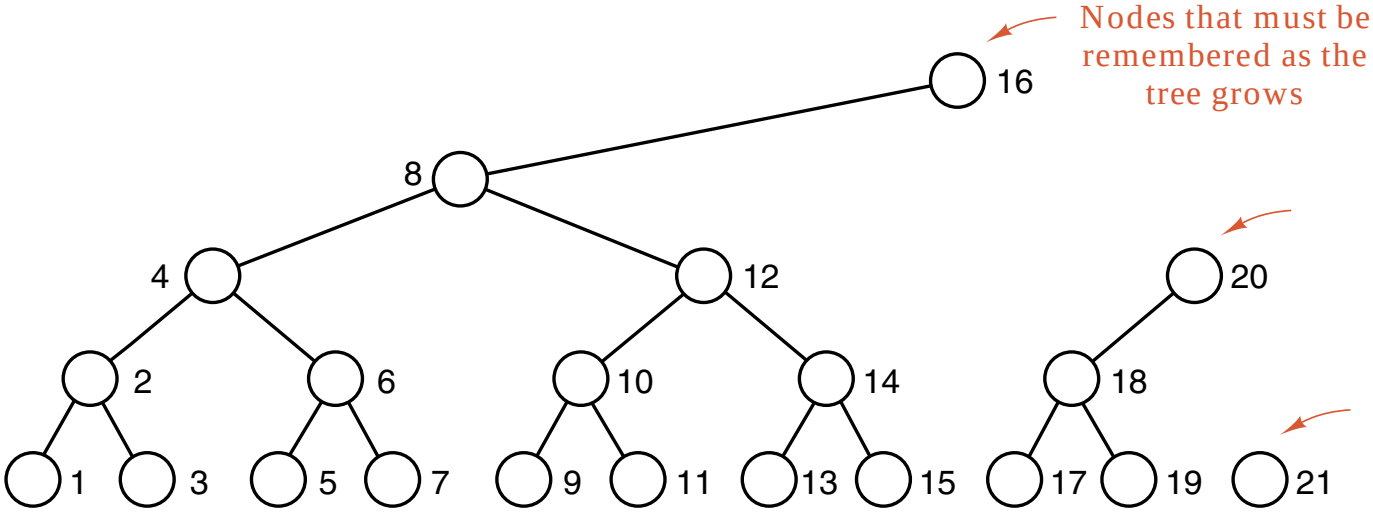
Building a Balanced Binary Search Tree

Problem: Start with an ordered list and build its entries into a binary search tree that is nearly balanced ("bushy").

> If the nodes of a complete binary tree are labeled in inorder sequence, starting with 1, then each node is exactly as many levels above the leaves as the highest power of 2 that divides its label.



$n = 1$ $n = 2$ $n = 3$ $n = 4$ $n = 5$



$n = 21$

To establish future links, we must remember pointers to one node on each level, the last node processed on that level.

Algorithm Development

- As each new node is added, it is clearly the last one received in the order, so we can place it in the List last_node and set its right pointer to NULL (at least temporarily).
- The left pointer of the new node is NULL if it is a leaf. Otherwise it is the entry in last_node one level lower than the new node.
- So that we can treat the leaves in the same way as other nodes, we consider the leaves to be on level 1, and we set up the initial element of last_node, in position 0, to have the pointer value NULL permanently.
- This convention means that we shall always count levels above the leaves *inclusively*, so the leaves themselves are on level 1, and so on.
- While we build up a tree, we need access to the internal structure of the tree in order to create appropriate links. Therefore, the new function will be implemented as a (public) method for a class of search trees. We will therefore create a new class called a Buildable_tree that is derived from the class Search_tree and possesses a new method, build_tree. The specification for a buildable tree is thus:

```
template <class Record>
class Buildable_tree: public Search_tree<Record> {
public:
   Error_code build_tree(const List<Record> &supply);
private:                              //    Add auxiliary function prototypes here.
};
```

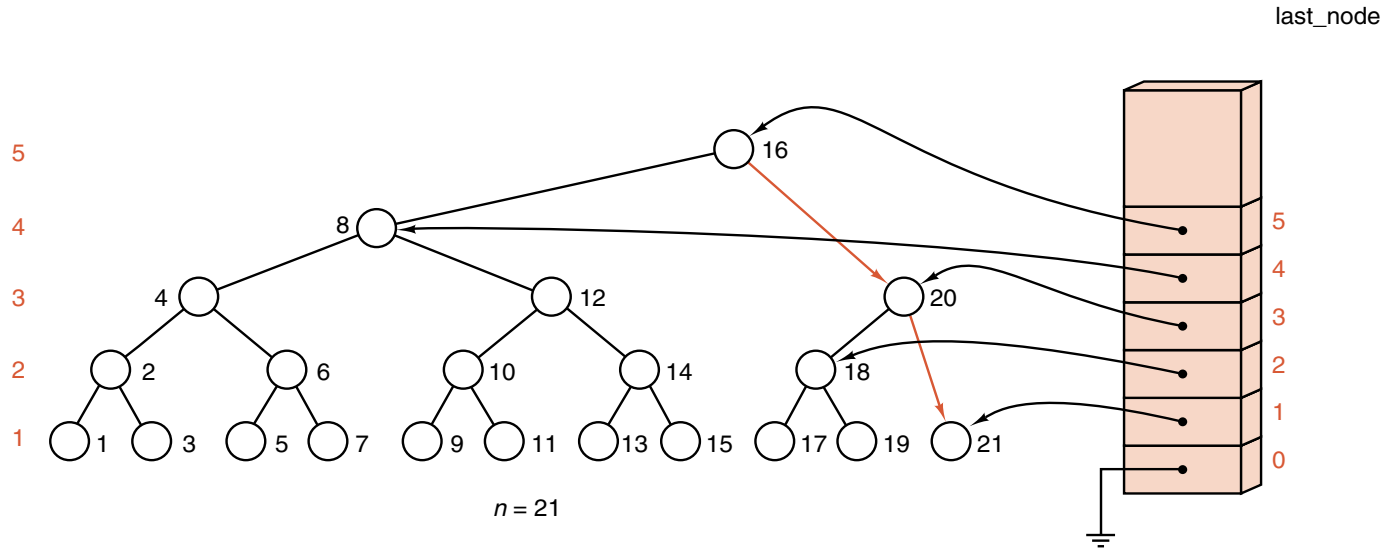
Method to Build Binary Search Tree

```
template <class Record>
Error_code Buildable_tree<Record>::
                build_tree(const List<Record> &supply)
/* Post:  If the entries of supply are in increasing order, a code of success is returned
          and the Search_tree is built out of these entries as a balanced tree.  Otherwise,
          a code of fail is returned and a balanced tree is constructed from the longest
          increasing sequence of entries at the start of supply.
   Uses:  The methods of class List; the functions build_insert, connect_subtrees,
          and find_root  */
{
   Error_code ordered_data = success;
                                        //    Set this to fail if keys do not increase.
   int count = 0;                       //    number of entries inserted so far
   Record x, last_x;
   List < Binary_node<Record> * > last_node;
                                        //    pointers to last nodes on each level
   Binary_node<Record> *none = NULL;
   last_node.insert(0, none);           //    permanently NULL (for children of leaves)

   while (supply.retrieve(count, x) ==  success) {
      if (count > 0 && x <= last_x) {
         ordered_data = fail;
         break;
      }
      build_insert(++count, x, last_node);
      last_x = x;
   }
   root = find_root(last_node);
   connect_trees(last_node);
   return ordered_data;          //    Report any data-ordering problems back to client.
}
```

Inserting a Node

```
template <class Record>
void Buildable_tree<Record> ::
   build_insert(int count, const Record &new_data,
                List < Binary_node<Record>* >  &last_node)
/* Post: A new node, containing the Record new_data, has been inserted as the
         rightmost node of a partially completed binary search tree. The level of this new
         node is one more than the highest power of 2 that divides count.
   Uses: Methods of class List  */

{
   int level;                  //    level of new node above the leaves, counting inclusively
   for (level = 1;  count % 2 ==  0;  level++)
      count /= 2;                  //    Use count to calculate level of next_node.
   Binary_node<Record>
                  *next_node = new Binary_node<Record>(new_data),
                  *parent;           //    one level higher in last_node
   last_node.retrieve(level - 1, next_node->left);
   if (last_node.size( ) <= level)
      last_node.insert(level, next_node);
   else
      last_node.replace(level, next_node);
   if (last_node.retrieve(level + 1, parent) ==  success &&
                                                 parent->right ==  NULL)
      parent->right = next_node;
}
```

Finishing the Task



Finding the root:

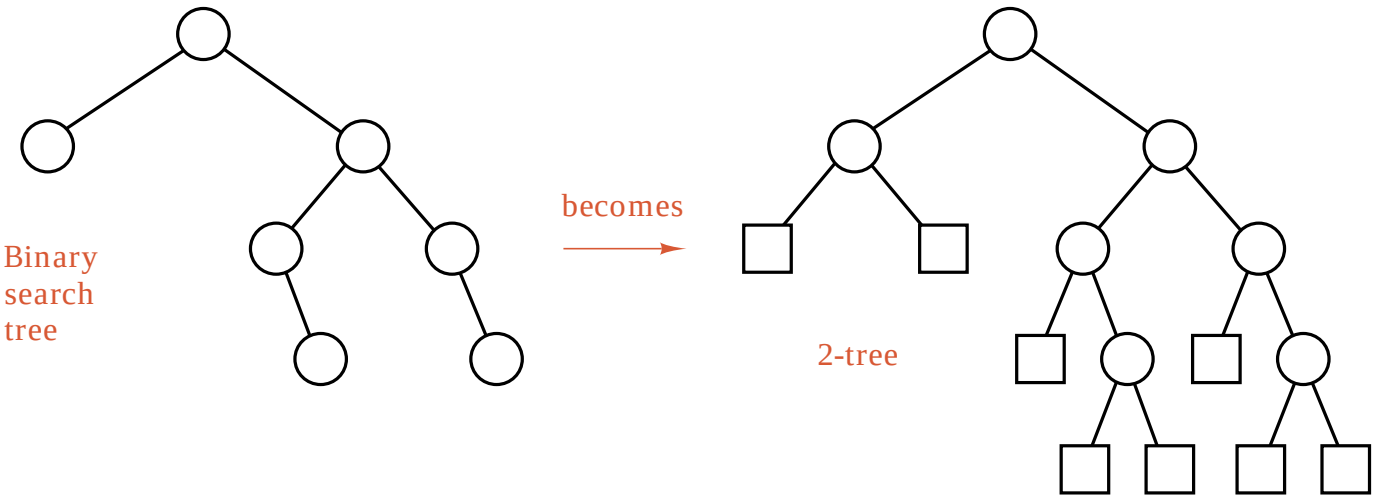
```
template <class Record>
Binary_node<Record> *Buildable_tree<Record>::find_root(
                List < Binary_node<Record>* >  &last_node)
/* Pre:   The list last_node contains pointers to the last node on each occupied
          level of the binary search tree.
   Post:  A pointer to the root of the newly created binary search tree is returned.
   Uses:  Methods of class List  */
{
   Binary_node<Record> *high_node;
   last_node.retrieve(last_node.size( ) – 1, high_node);
      //     Find root in the highest occupied level in last_node.
   return high_node;
}
```

Tying Subtrees Together

```
template <class Record>
void Buildable_tree<Record> :: connect_trees(
    const List < Binary_node<Record>* >  &last_node)
/* Pre:   The nearly-completed binary search tree has been initialized. List last_node
          has been initialized and contains links to the last node on each level of the tree.
   Post:  The final links have been added to complete the binary search tree.
   Uses:  Methods of class List  */
{
   Binary_node<Record>
      *high_node,                    //   from last_node with NULL right child
      *low_node;                     //   candidate for right child of high_node
   int high_level = last_node.size( ) - 1,
       low_level;

   while (high_level > 2) {          //   Nodes on levels 1 and 2 are already OK.
      last_node.retrieve(high_level, high_node);
      if (high_node->right != NULL)
         high_level--;               //   Search down for highest dangling node.
      else {                         //   Case: undefined right tree
         low_level = high_level;
         do {                        //   Find the highest entry not in the left subtree.
            last_node.retrieve(--low_level, low_node);
         } while (low_node != NULL &&
                  low_node->data < high_node->data);
         high_node->right = low_node;
         high_level = low_level;
      }
   }
}
```

Random Search Trees and Optimality

Problem: If we assume that the keys have been inserted into a binary search tree in random order, then, on average, how many more comparisons are needed in a search of the resulting tree than would be needed in a completely balanced tree?



There is a one-to-one correspondence between binary search trees and 2-trees in which left and right are considered different from each other.

- Assume that the $n!$ possible orderings of keys are equally likely in building the binary search tree.
- When there are n nodes in the tree, we denote by $S(n)$ the number of comparisons done in the average successful search and by $U(n)$ the number in the average unsuccessful search.
- Recurrence relation:

$$S(n) = 1 + \frac{U(0) + U(1) + \cdots + U(n-1)}{n}.$$

Performance of Random Binary Search Trees

- Relation between internal and external path length:

$$S(n) = \left(1 + \frac{1}{n}\right) U(n) - 3.$$

- Solve, subtracting case $n-1$ from case n:

$$U(n) = U(n-1) + \frac{4}{n+1}.$$

- Evaluate by expanding down to case 0 and using the Harmonic Number:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n.$$

THEOREM 10.3 The average number of nodes visited during a search of the average binary search tree with n nodes is approximately

$$2 \ln n = (2 \ln 2)(\lg n) \approx 1.39 \lg n,$$

and the number of key comparisons is approximately

$$4 \ln n = (4 \ln 2)(\lg n) \approx 2.77 \lg n.$$

COROLLARY 10.4 The average binary search tree requires approximately $2 \ln 2 \approx 1.39$ times as many comparisons as a completely balanced tree.

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.

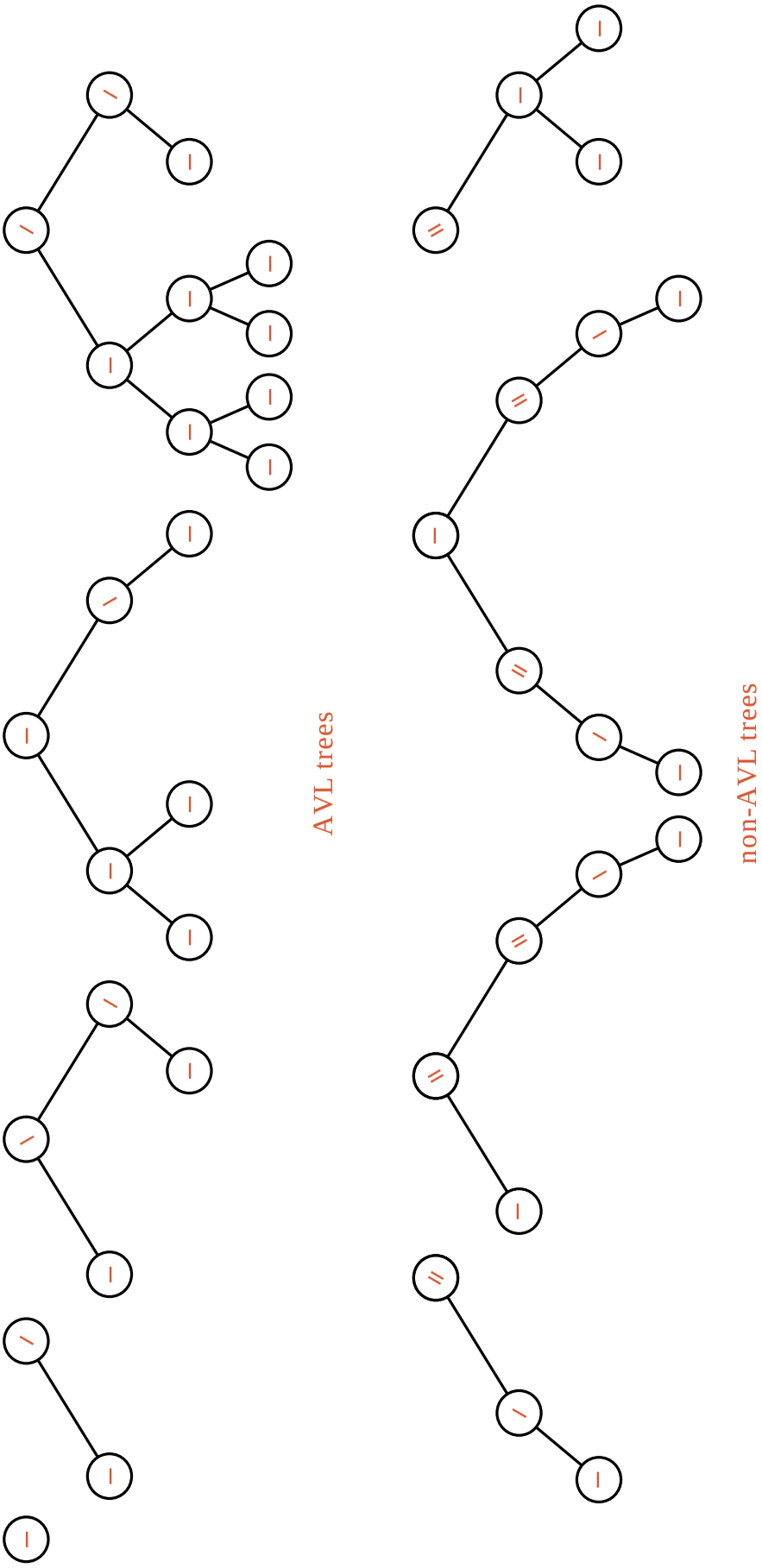
With each node of an AVL tree is associated a ***balance factor*** that is ***left higher***, ***equal***, or ***right higher*** according, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.

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 '\.'



Examples of AVL trees and other binary trees
Transp. 38, Sect. 10.4, Height Balance: 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;
};
```

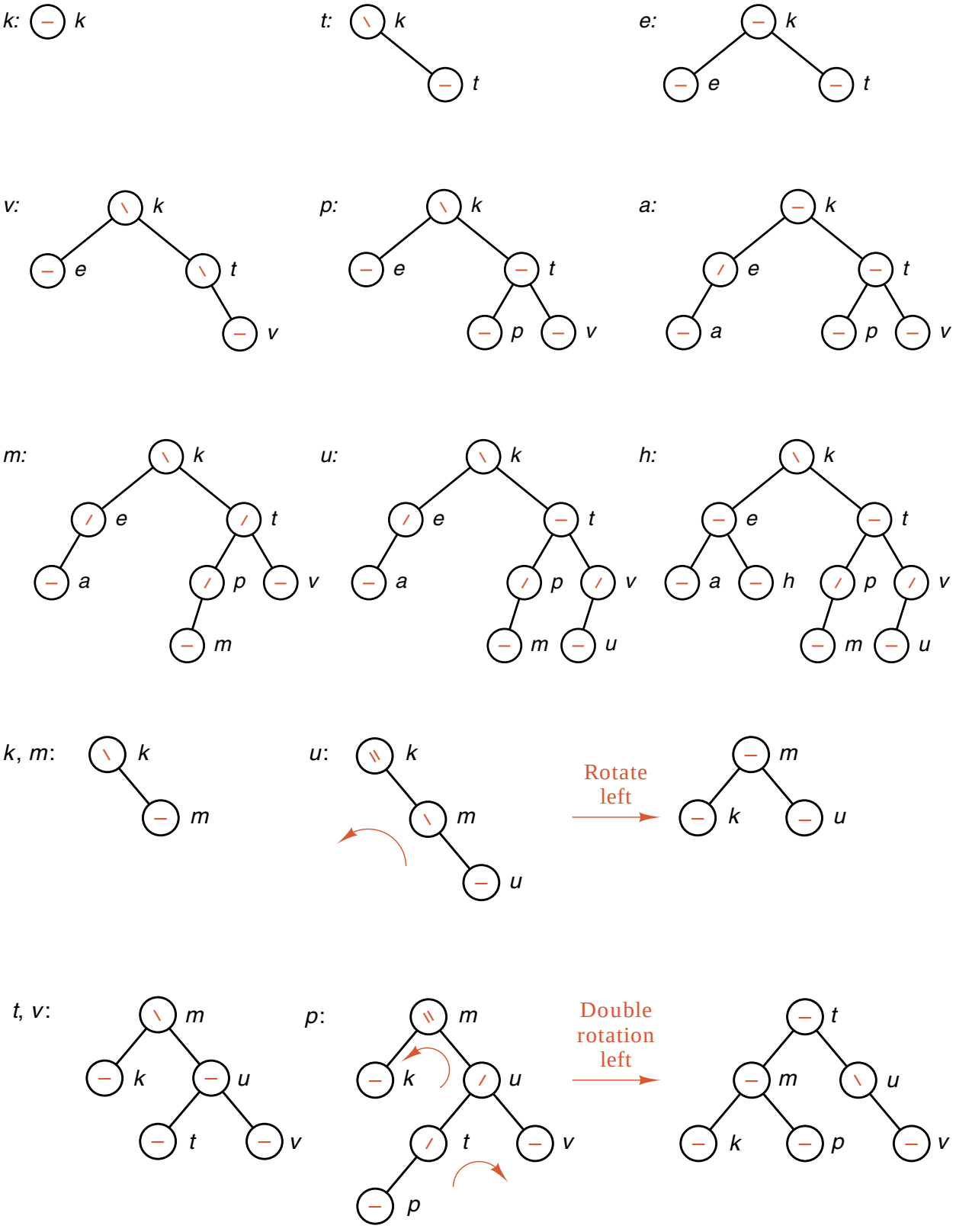
Class Declaration for AVL Trees

- 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.  */
```

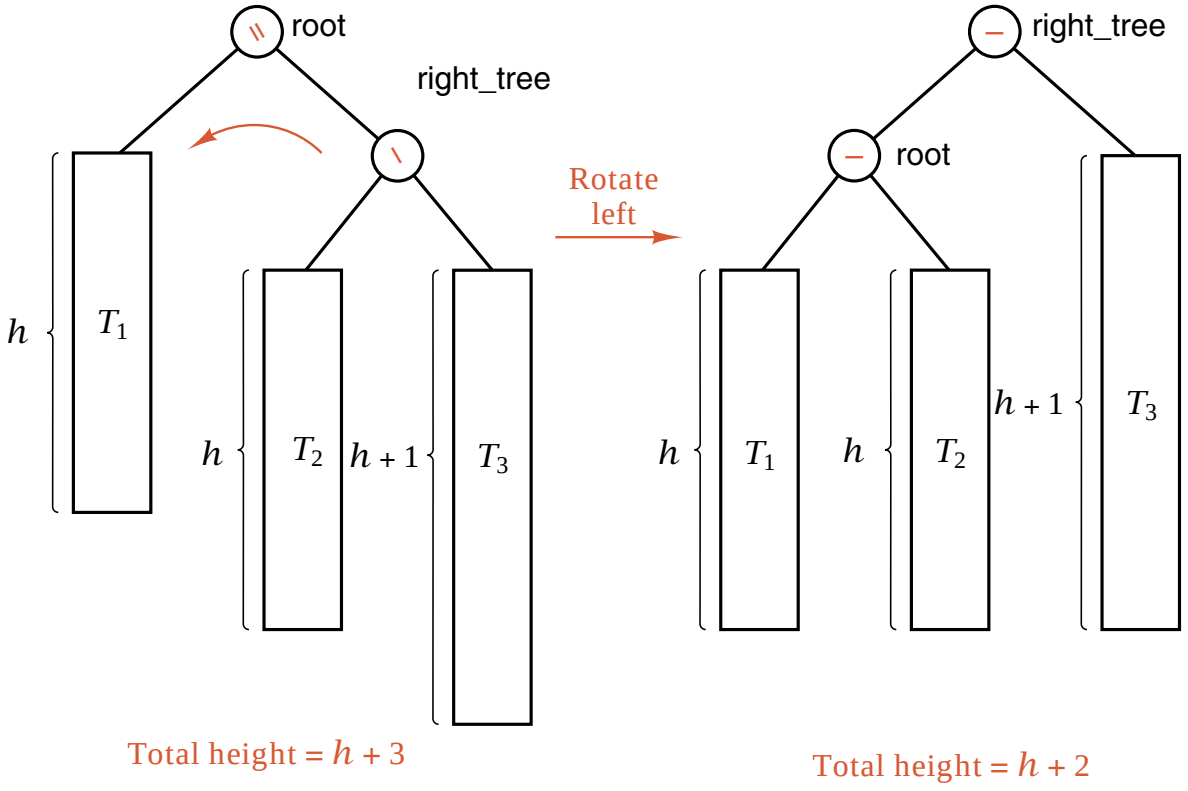
Recursive Insertion

```
{
   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

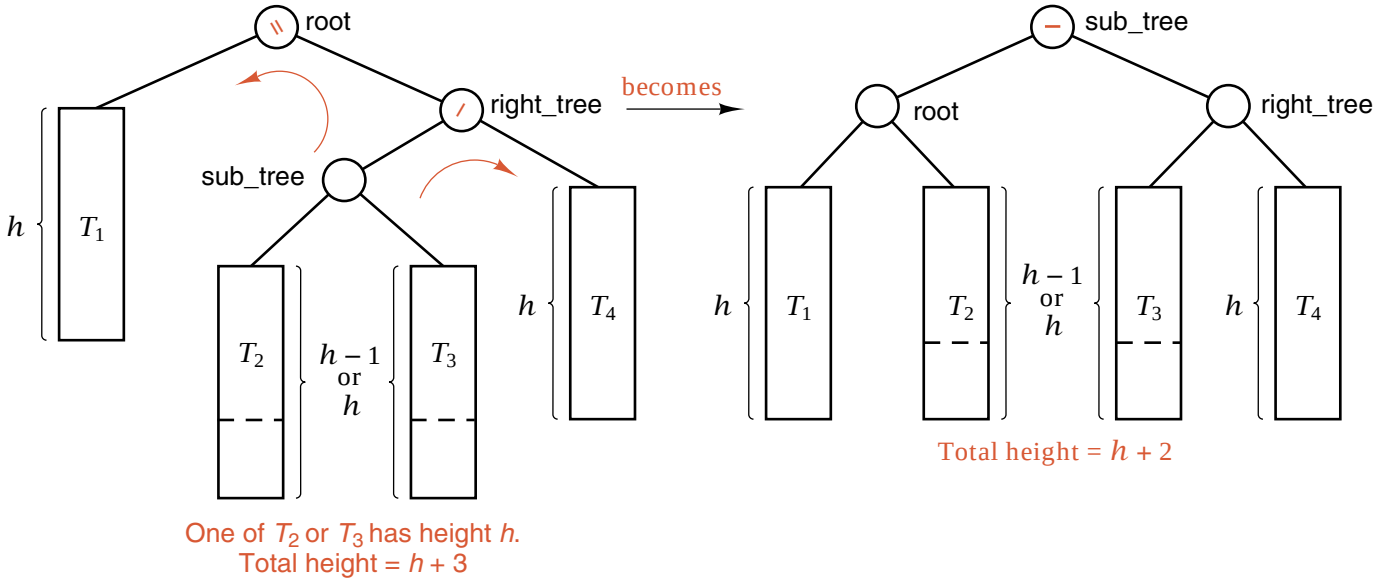
```
   else {                                  //    Insert in right subtree.
      result = avl_insert(sub_root->right, new_data, taller);
      if (taller ==  true)
         switch (sub_root->get_balance( )) {
         case left_higher:
            sub_root->set_balance(equal_height);
            taller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

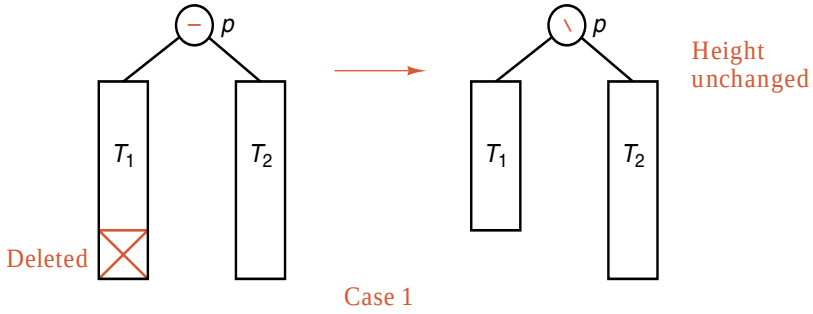
old sub_tree	*new* root	*new* right_tree	*new* sub_tree
–	–	–	–
/	–	\	–
\	/	–	–

```
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;
   }
}
```

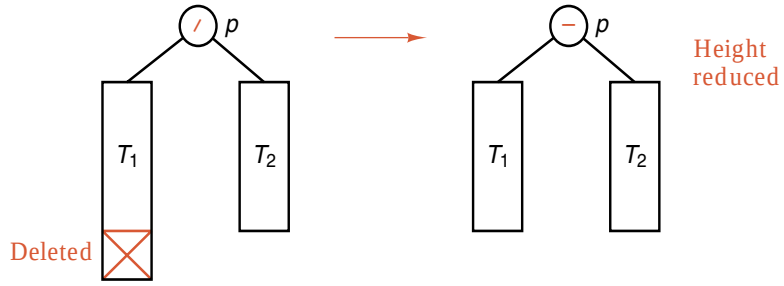
Removal of a Node

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

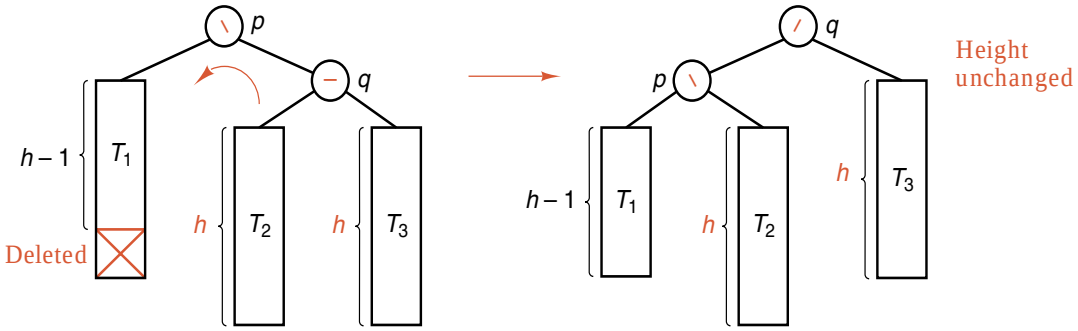


Case 1

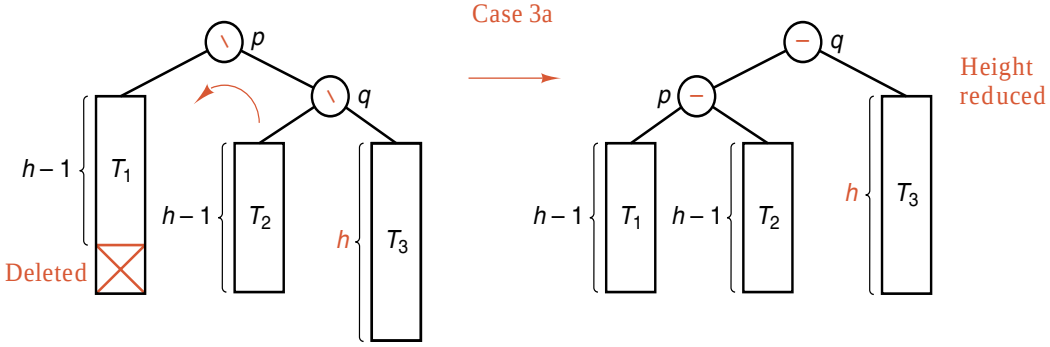


Case 2

single left rotations

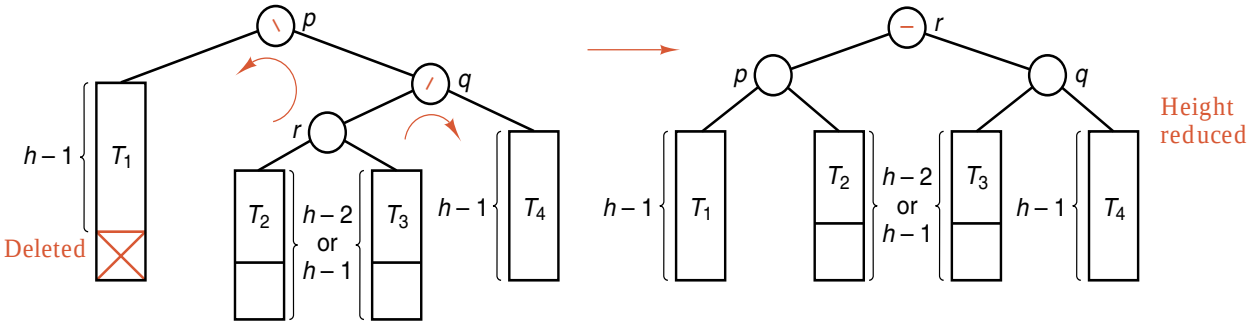


Case 3a

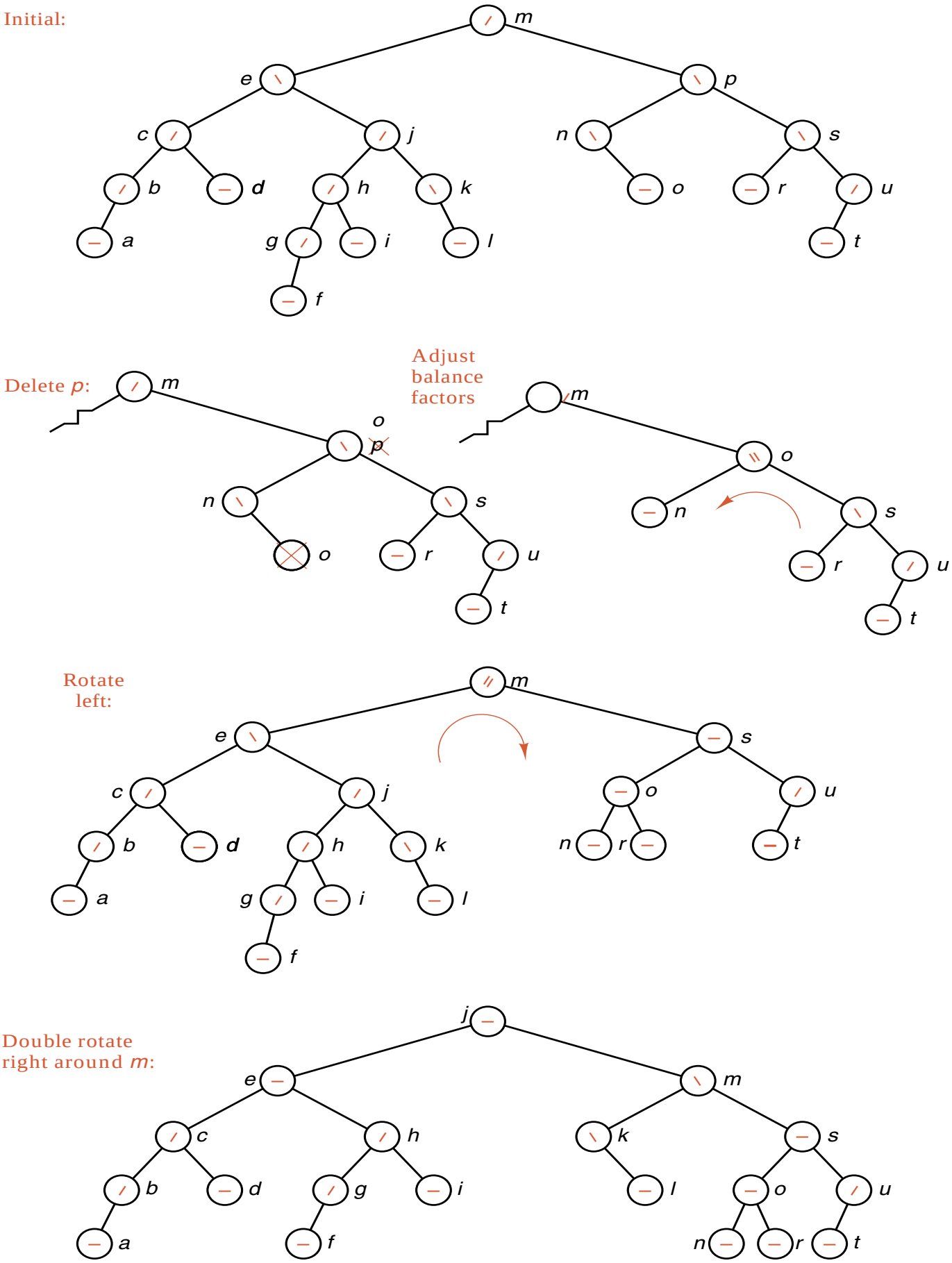


Case 3b

double rotation



Case 3c



Analysis of AVL Trees

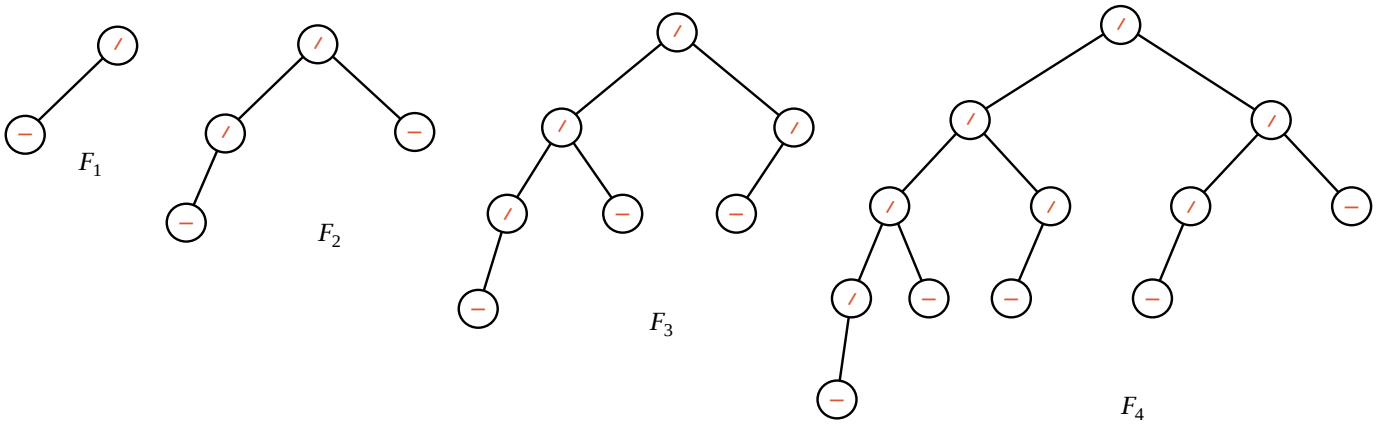
- The number of recursive calls to insert a new node can be as large as the height of the tree.
- At most one (single or double) rotation will be done per insertion.
- A rotation improves the balance of the tree, so later insertions are less likely to require rotations.
- It is very difficult to find 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***.



Analysis of Fibonacci Trees

- If we write $|T|$ for the number of nodes in a tree T, we then have the recurrence relation for Fibonacci trees:

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1,$$

  where $|F_0| = 1$ and $|F_1| = 2$.

- By the evaluation of Fibonacci numbers in Section A.4,

$$|F_h| + 1 \approx \frac{1}{\sqrt{5}} \left[\frac{1 + \sqrt{5}}{2} \right]^{h+2}$$

- Take the logarithms of both sides, keeping only the largest terms:

$$h \approx 1.44 \lg |F_h|.$$

- The sparsest possible AVL tree with n nodes has height about $1.44 \lg n$ compared to:
  - A perfectly balanced binary search tree with n nodes has height about $\lg n$.
  - A random binary search tree, on average, has height about $1.39 \lg n$.
  - A degenerate binary search tree has height as large as n.

- Hence the algorithms for manipulating AVL trees are guaranteed to take no more than about 44 percent more time than the optimum. In practice, AVL trees do much better than this on average, perhaps as small as $\lg n + 0.25$.

Splay Trees: A Self-Adjusting Data Structure

- In some applications, we wish to keep records that are newly inserted or frequently accessed very close to the root, while records that are inactive may be placed far off, near or in the leaves.

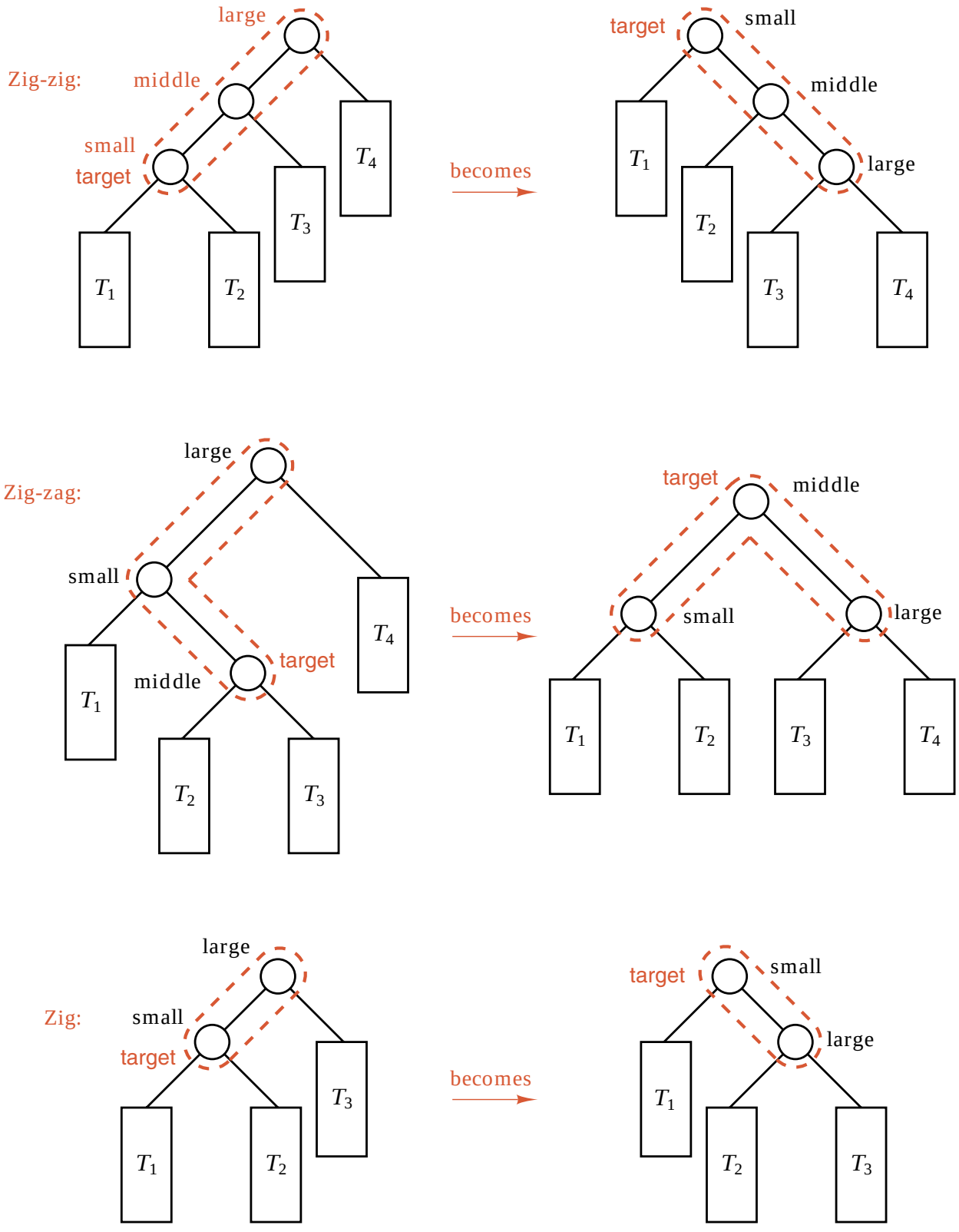
- We make a binary search tree into a ***self-adjusting*** data structure that *automatically* changes its shape to bring records closer to the root as they are more frequently accessed, allowing inactive records to drift slowly out toward the leaves.

- In a ***splay tree***, every time we access a node, whether for insertion or retrieval, we lift the newly-accessed node all the way up to become the root of the modified tree.

- We use rotations like those of AVL trees, but now with *many* rotations done for *every* insertion or retrieval in the tree.

- We move the target node *two* levels up the tree at each step. Consider the path going from the root *down* to the accessed node. If we move left, we say that we ***zig***, and if we move right we say that we ***zag***. A move of two steps left (going down) is then called ***zig-zig***, two steps right ***zag-zag***, left then right ***zig-zag***, and right then left ***zag-zig***. If the length of the path is odd, either a single ***zig*** move or a ***zag*** move occurs at the end.

Splay Rotations

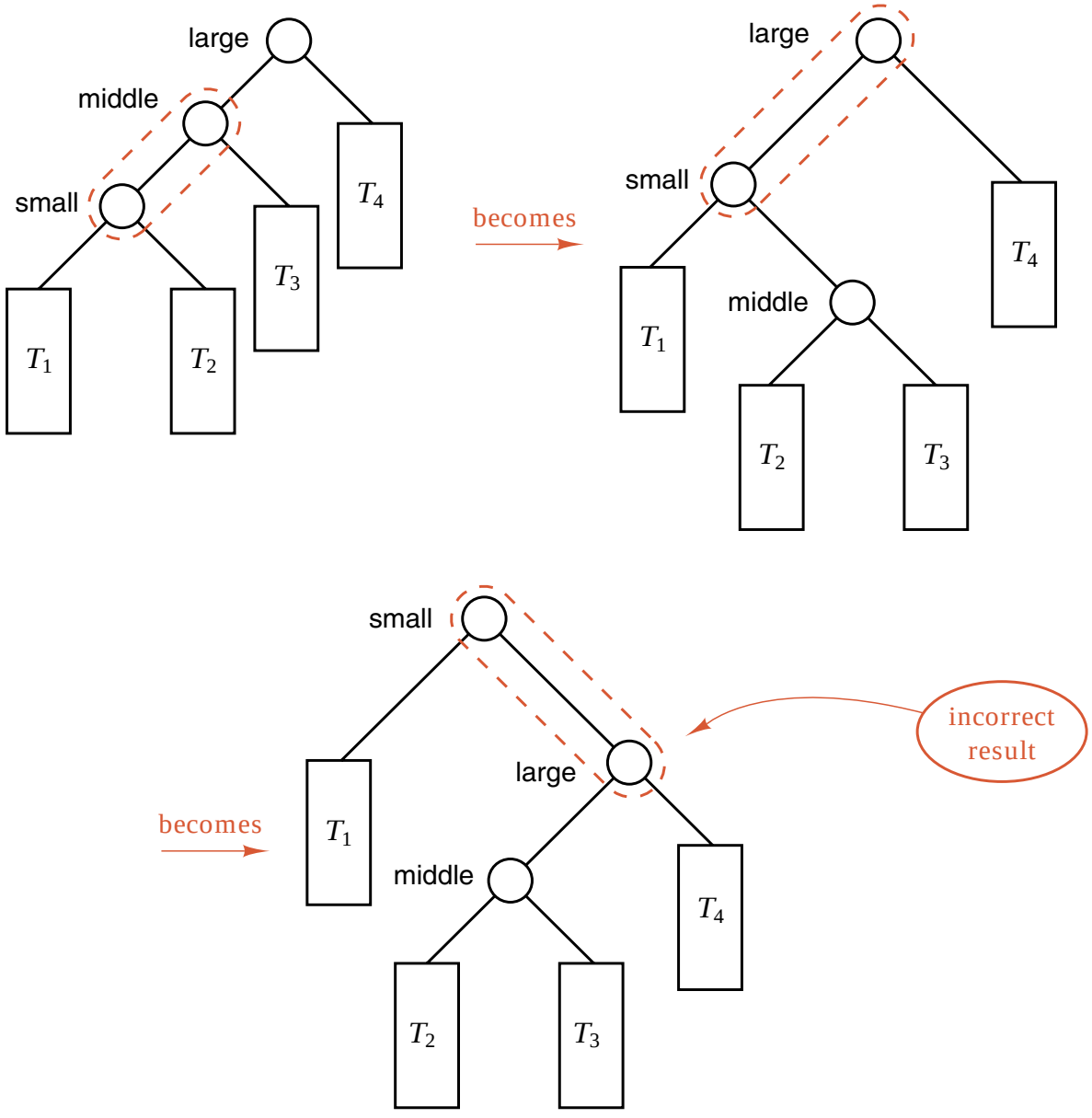


Notes on Splay Rotations

- The zig-zag case is identical to that of an AVL double rotation, and the zig case is identical to a single rotation.
- The zig-zig case is *not* the same as would be obtained by lifting the target node twice with single rotations.
- Always think of lifting the target *two* levels at a time (except for a single zig or zag step at the end).
- It is *only* the nodes on the path from the target to the root whose relative positions are changed, and that only in the ways shown by colored dashed curves in the figures.
- None of the subtrees off the path (T_1, T_2, T_3, and T_4) changes its shape at all, and these subtrees are attached to the path in the only places they can go to maintain the search-tree ordering of all the keys.

Zig-Zig and Single Rotations

A zig-zig movement is *not* the same as would be obtained by lifting the target node twice with single rotations.



Top-Down and Bottom-Up Splaying

- By hand, we perform ***bottom-up*** splaying, beginning at the target node and moving up the path to the root two steps at a time. A single zig or zag move may occur at the top of the tree. Bottom-up splaying is essentially a two-pass method, first searching down the tree and then splaying the target up to the root.

- In a computer algorithm, we splay from the top down *while* we are searching for the target node. When we find the target, it is immediately moved to the root of the tree, or, if the search is unsuccessful, a new root is created that holds the target. In top-down splaying, a single zig or zag move occurs at the bottom of the splaying process.

- If you run the splaying function we develop on the example trees, you will obtain the same results as doing bottom-up splaying by hand when the target is moved an even number of levels, but the results will be different when it moves an odd number of levels.

Algorithm Development

- Given a target key, we develop a function that searches down the tree for the key, splaying as it goes. If it finds the key, then it retrieves it; if not, then the function inserts it in a new node. In either case, the node with the target key becomes the root of the tree.

- We implement splay trees as a derived class of **class** Search_tree.

```
template <class Record>
class Splay_tree: public Search_tree<Record> {
public:
   Error_code splay(const Record &target);
private:                              //  Add auxiliary function prototypes here.
};
```

- A splay tree does *not* keep track of heights and does *not* use any balance factors like an AVL tree.

- Top-down splaying uses the same moves (zig-zig, zig-zag, and the rest) as bottom-up splaying, but expressed differently.

- During splaying, the root is left empty so that, at the end, the target node can be moved or inserted directly into the root.

The Three-Way Tree Partition

- During splaying, the tree temporarily falls apart into three separate subtrees, which are reconnected after the target is made the root.
  - The ***central subtree*** contains nodes within which the target will lie if it is present.
  - The ***smaller-key subtree*** contains nodes whose keys are strictly less than the target. Every key in the smaller-key subtree is less than every key in the central subtree.
  - The ***larger-key subtree*** contains nodes whose keys are strictly greater than the target. Every key in the larger-key subtree is greater than every key in the central subtree.
- These conditions hold throughout the splaying process; we call them the ***three-way invariant***.

- Initially, the central subtree is the whole tree, and the smaller-key and larger-key subtrees are empty.
- As splaying proceeds, nodes are stripped off the central subtree and joined to one of the other two subtrees.
- When the search ends, the root of the central subtree will be the target node if it is present, and the central subtree will be empty if the target was not found.
- All the components will finally be joined together with the target as the root.

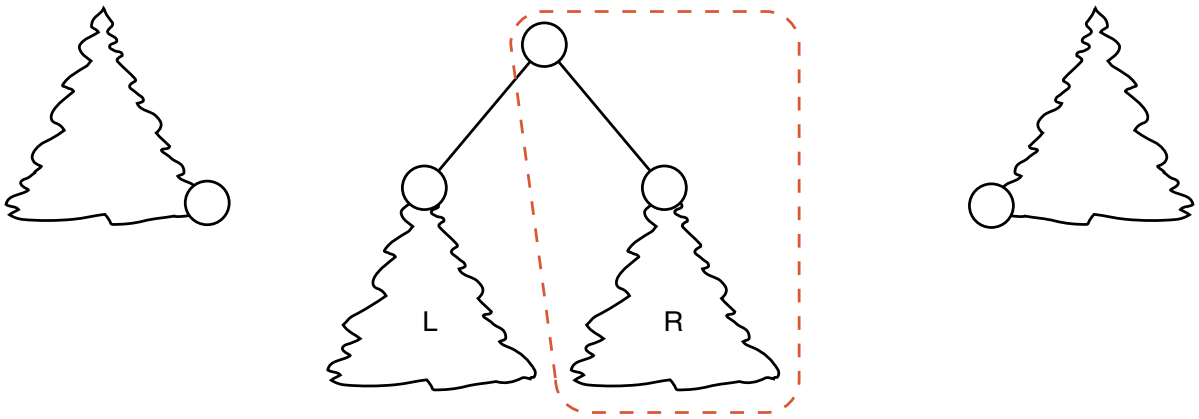
Basic Action: link_right

- Suppose the target is smaller than the key in the root of the central subtree.
- In this case, we take the root and its right subtree and adjoin them to the larger-key tree, reducing the central subtree to the former left subtree of the root.
- We call this action link_right.
- link_right is exactly a zig move, except that the link from the former left child down to the former parent is deleted; instead, the parent (with its right subtree) moves into the larger-key subtree.
- The three-way invariant tells us that every key in the central subtree comes before every key in the larger-key subtree; hence this parent (with its right subtree) must be attached on the left of the leftmost node (first in ordering of keys) in the larger-key subtree.
- Note that, after link_right is performed, the three-way invariant continues to be true.

Action of link_right: a Zig Move

Before:

Case: target is less than root key.

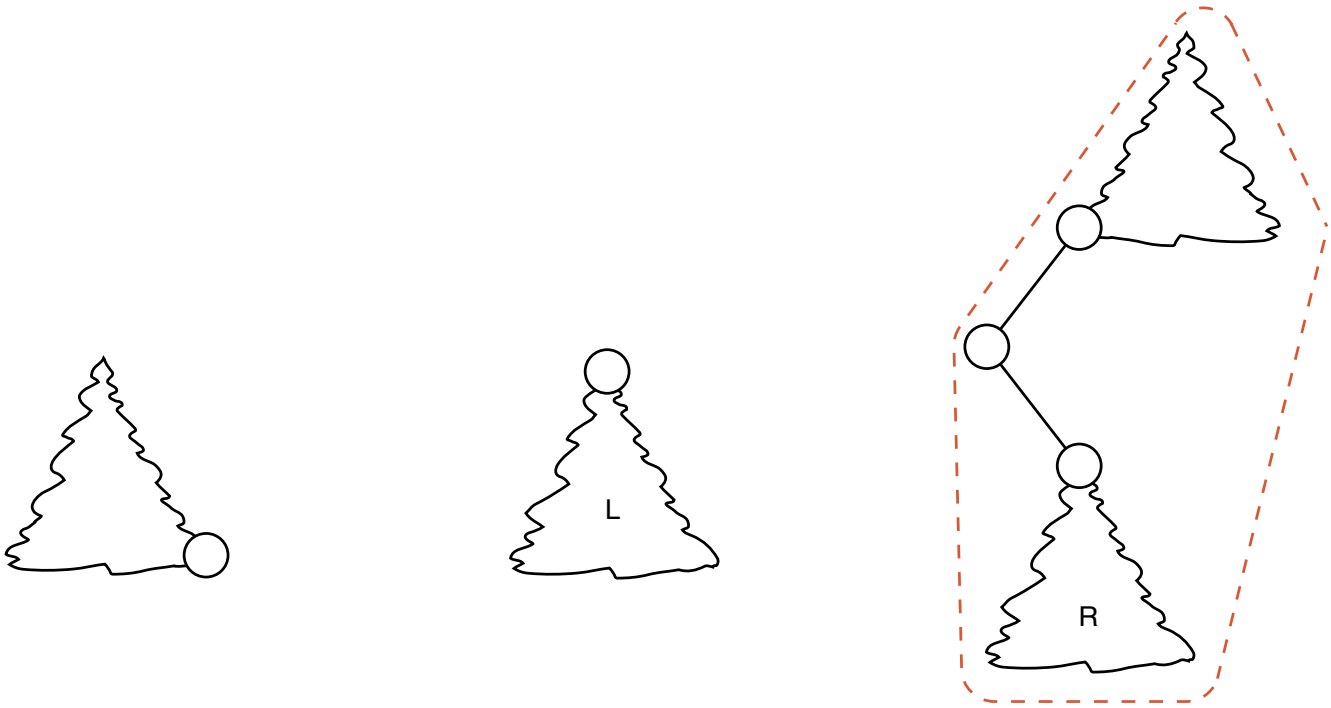


smaller-key subtree

central subtree

larger-key subtree

After:

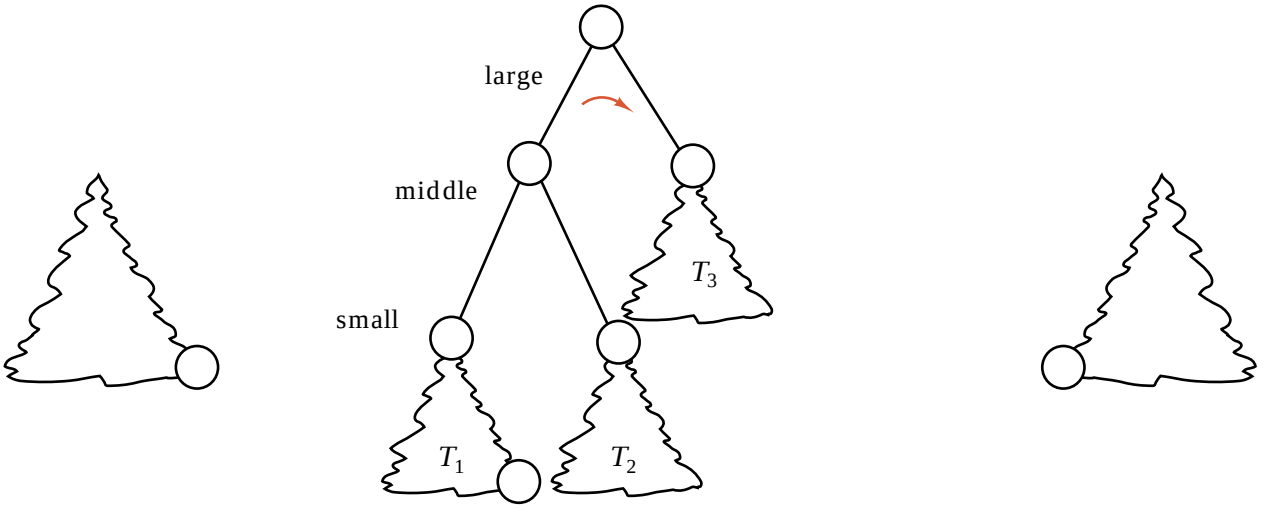


smaller-key subtree

new central subtree

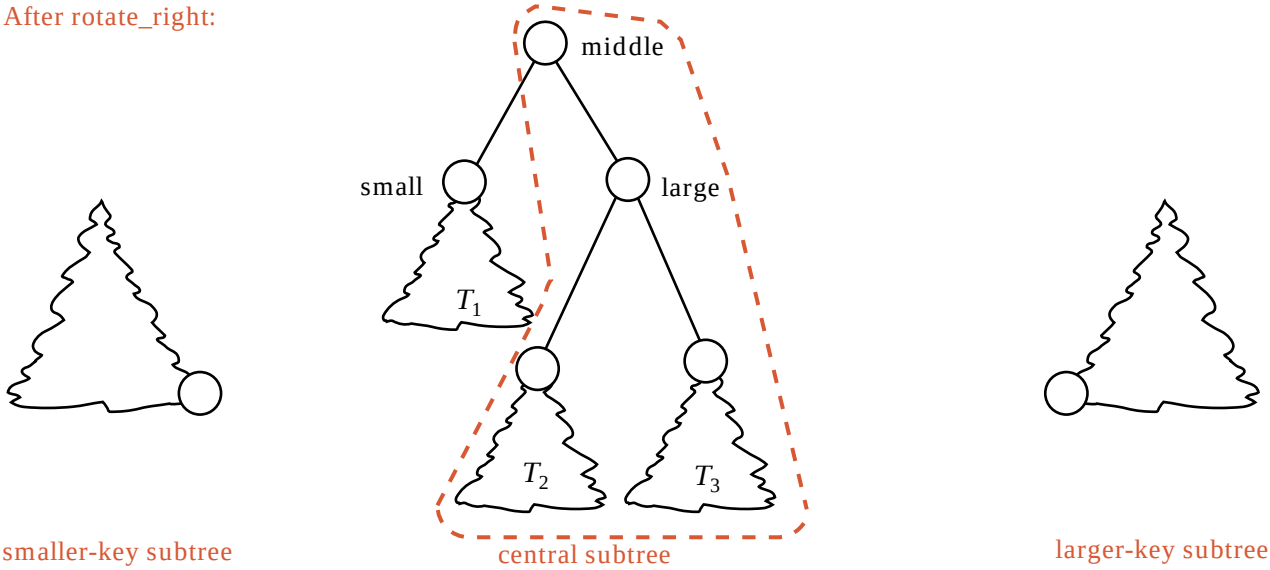
new larger-key subtree

Case: target is less than left child of root.



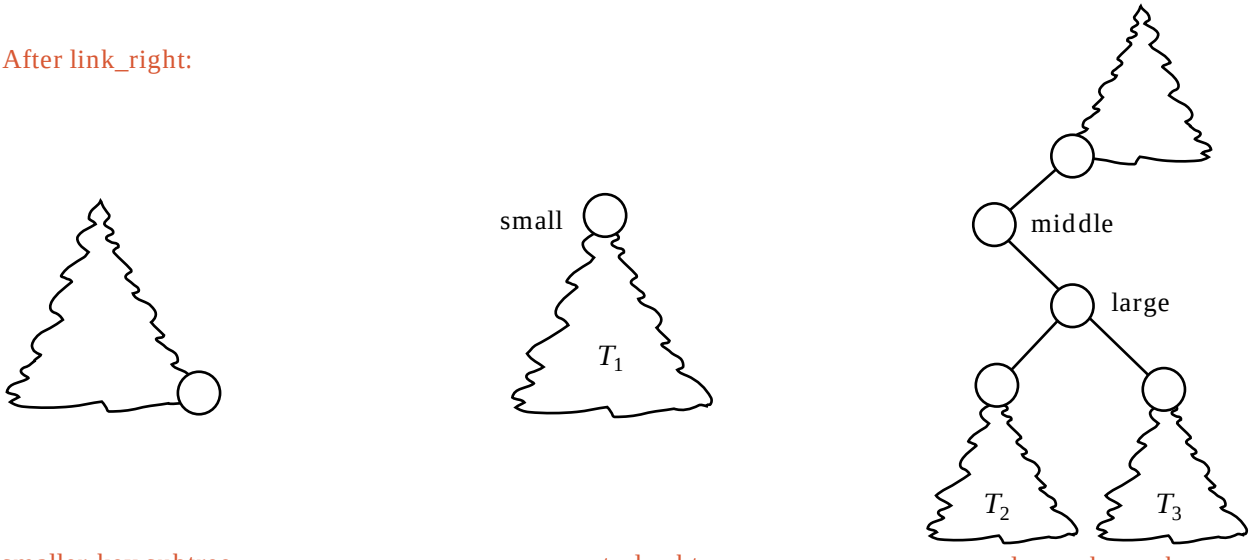
smaller-key subtree | central subtree | larger-key subtree

After rotate_right:



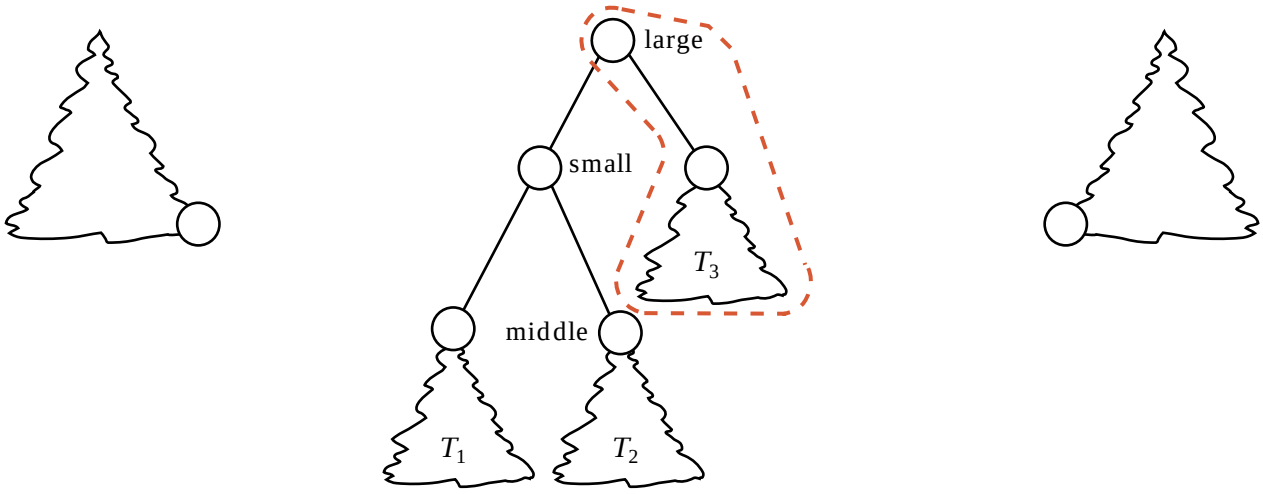
smaller-key subtree | central subtree | larger-key subtree

After link_right:



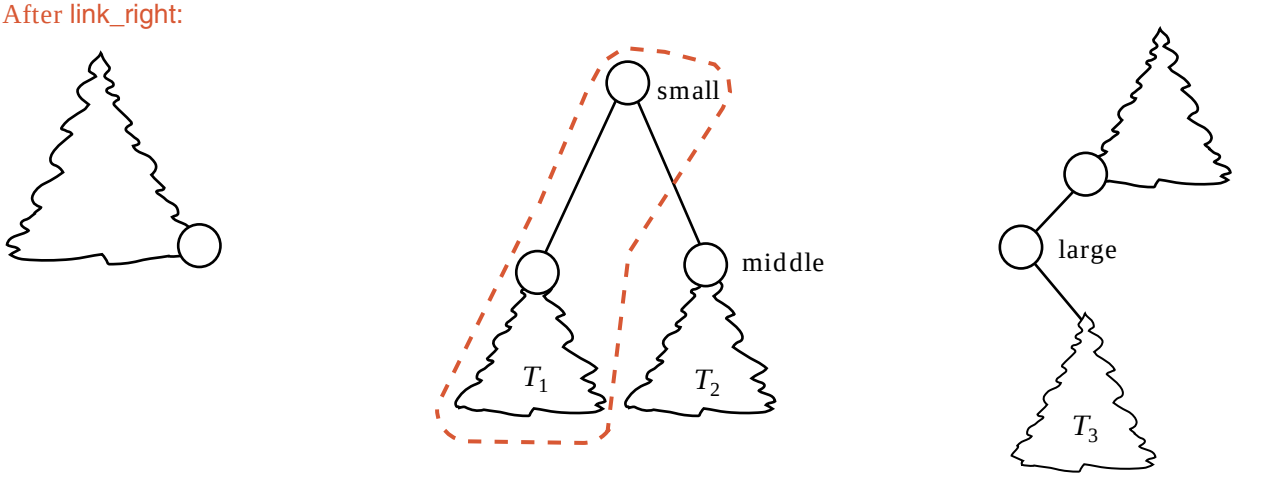
smaller-key subtree | new central subtree | new larger-key subtree

Case: target is between root and its left child.



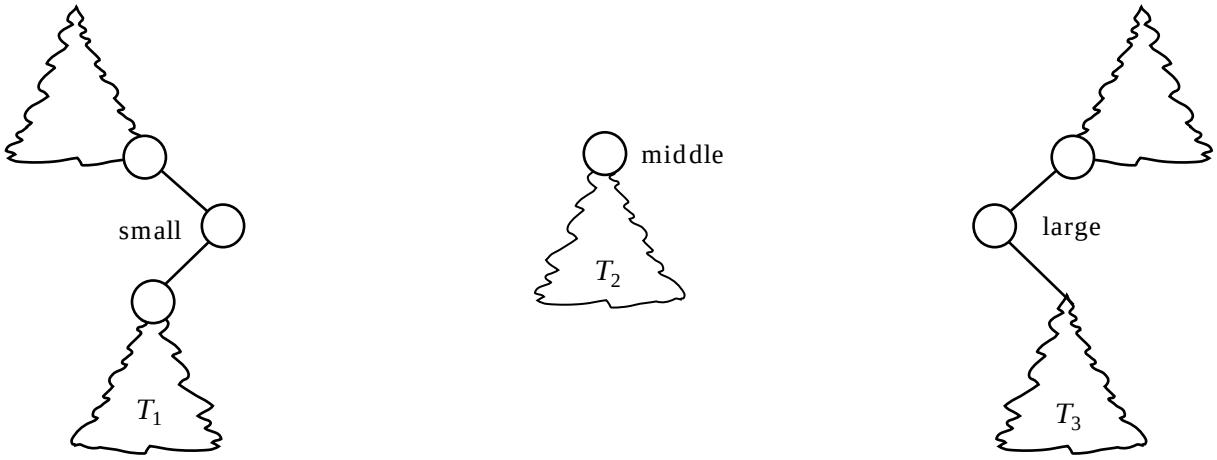
smaller-key subtree | central subtree | larger-key subtree

After link_right:



smaller-key subtree | central subtree | new larger-key subtree

After link_left:



new smaller-key subtree | new central subtree | new larger-key subtree

Programming Conventions

- We use five pointer variables to keep track of required positions in the three subtrees:
  - current gives the root of the central subtree of nodes not yet searched.
  - child refers to either the left or right child of current, as required.
  - last_small gives the largest (that is, the rightmost) node in the smaller-key subtree, which is the place where additional nodes must be attached.
  - first_large gives the smallest (that is, the leftmost) node in the larger-key subtree, which is the place where additional nodes must be attached.
  - dummy points to a dummy node that is created at the beginning of the splaying function and is deleted at its end.
- The ***dummy node*** contains no key or other data.
- We initialize both last_small and first_large to dummy. Hence these pointers always refer to actual nodes, and therefore link_right and link_left will not attempt to dereference NULL pointers.

C++ Functions

```
template <class Record>
void Splay_tree<Record> ::
    link_right(Binary_node<Record> * &current,
               Binary_node<Record> * &first_large)
/* Pre:  The pointer first_large points to an actual Binary_node (in particular, it is not
         NULL). The three-way invariant holds.
   Post: The node referenced by current (with its right subtree) is linked to the left of the
         node referenced by first_large. The pointer first_large is reset to current.
         The three-way invariant continues to hold. */
{
  first_large->left = current;
  first_large = current;
  current = current->left;
}
```

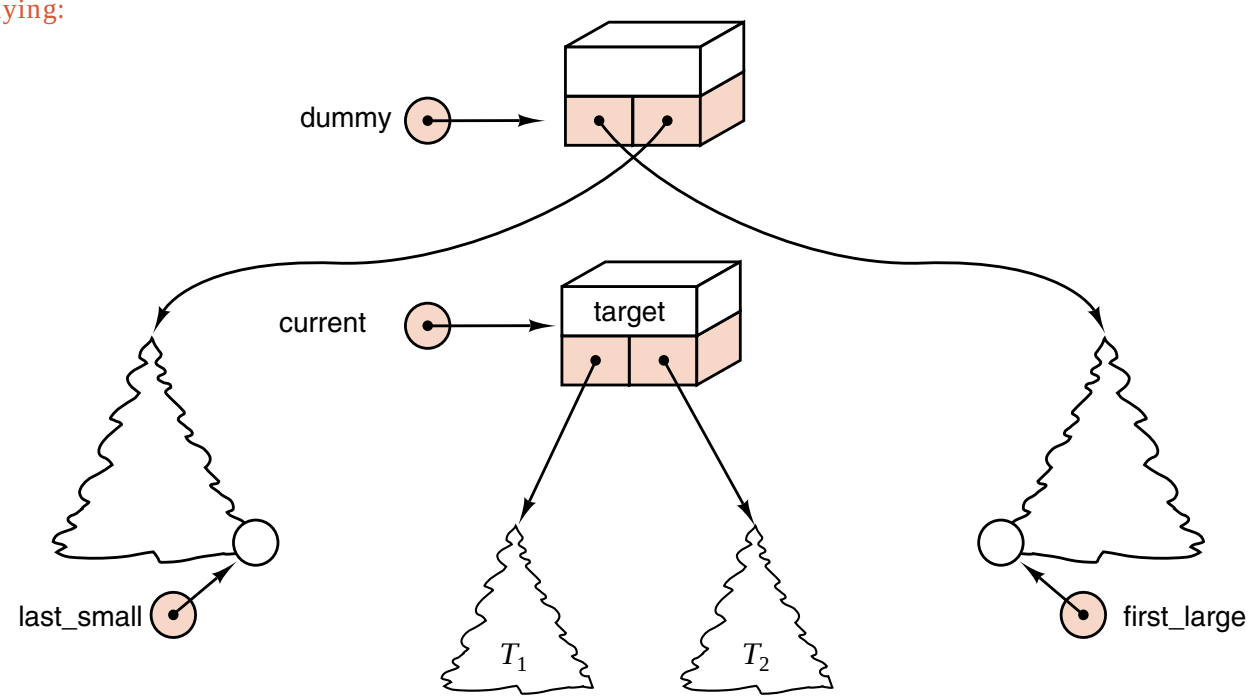
The rotation functions do not use the dummy node, and they do not cause any change in the three-way partition.

```
template <class Record>
void Splay_tree<Record> ::
    rotate_right(Binary_node<Record> * &current)
/* Pre:  current points to the root node of a subtree of a Binary_tree. This subtree
         has a nonempty left subtree.
   Post: current is reset to point to its former left child, and the former current node is
         the right child of the new current node. */
{
  Binary_node<Record> *left_tree = current->left;
  current->left = left_tree->right;
  left_tree->right = current;
  current = left_tree;
}
```

Finishing the Task

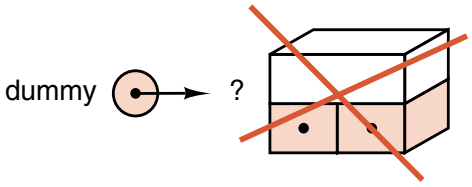
- When the search finishes, the root of the central subtree points at the target node or is NULL.
- If the target is found, it must become the root of the whole tree, but, before that, its left and right subtrees are now known to belong in the smaller-key and larger-key subtrees, respectively, so they should be moved there.
- If the search instead terminates unsuccessfully, with current == NULL, then a new root containing target must be created.
- Finally, the left and right subtrees of the new root should now be the smaller-key and larger-key subtrees.
- How do we find the roots of these subtrees, since we have kept pointers only to their rightmost and leftmost nodes, respectively?
- To answer this question, let us remember what happened at the beginning of the search. Initially, both pointers last_small and first_large were set to refer to the dummy node. When a node (and subtree) are attached to the larger-key subtree, they are attached on its left, by changing first_large->left. Since first_large is initially dummy, we can now, at the end of the search, find the first node inserted into the larger-key subtree, and thus its root, simply as dummy->left. Similarly, dummy->right points to the root of the smaller-key subtree. Hence the dummy node provides us with pointers to the roots of the smaller- and larger-key subtrees that would otherwise be lost.
- Note that the pointers are stored in positions reversed from what one might expect.

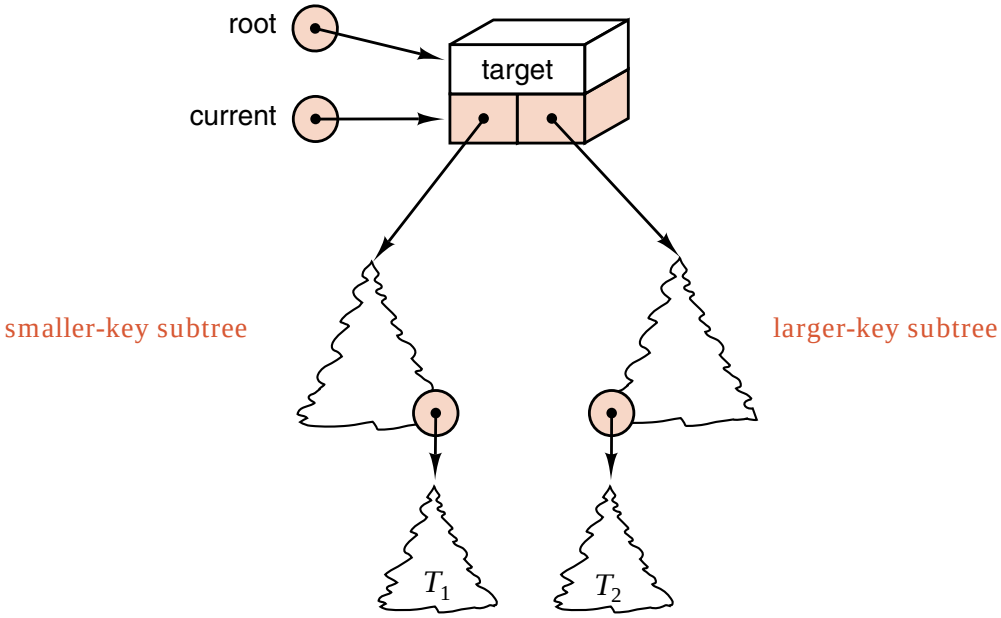
After splaying:



smaller-key subtree central subtree larger-key subtree

Reconnect into one tree:





Splaying: The Final Method

```
template <class Record>
Error_code Splay_tree<Record> :: splay(const Record &target)
/* Post:  If a node of the splay tree has a key matching that of target, it has been moved
          by splay operations to be the root of the tree, and a code of entry_found is
          returned. Otherwise, a new node containing a copy of target is inserted as the
          root of the tree, and a code of entry_inserted is returned.  */
{
  Binary_node<Record> *dummy = new Binary_node<Record>;
  Binary_node<Record> *current = root,
                      *child,
                      *last_small = dummy,
                      *first_large = dummy;

//    Search for target while splaying the tree.
  while (current != NULL && current->data != target)
    if (target < current->data) {
      child = current->left;
      if (child == NULL || target == child->data)  //    zig move
        link_right(current, first_large);
      else if (target < child->data) {  //    zig-zig move
        rotate_right(current);
        link_right(current, first_large);
      }
      else {                            //    zig-zag move
        link_right(current, first_large);
        link_left(current, last_small);
      }
    }
```

Splaying Method, Continued

```
    else {                              //    case: target > current->data
      child = current->right;
      if (child ==  NULL || target ==  child->data)
        link_left(current, last_small);   //    zag move
      else if (target > child->data) {  //    zag-zag move
        rotate_left(current);
        link_left(current, last_small);
      }
      else {                            //    zag-zig move
        link_left(current, last_small);
        link_right(current, first_large);
      }
    }
//    Move root to the current node, which is created if necessary.
  Error_code result;
  if (current ==  NULL) {               //    Search unsuccessful: make a new root.
    current = new Binary_node<Record>(target);
    result = entry_inserted;
    last_small->right = first_large->left = NULL;
  }
  else {                                //    successful search
    result = entry_found;
    last_small->right = current->left;  //    Move remaining central nodes.
    first_large->left = current->right;
  }
  root = current;                       //    Define the new root.
  root->right = dummy->left;            //    root of larger-key subtree
  root->left = dummy->right;            //    root of smaller-key subtree
  delete dummy;
  return result;
}
```

Amortized Algorithm Analysis: Introduction

- In both *worst-case* analysis and *average-case* analysis, we take a single event or single situation and attempted to determine how much work an algorithm does to process it.

- Amortized analysis differs in that it considers a long *sequence* of events rather than a single event in isolation. Amortized analysis then gives a *worst-case* estimate of the cost of a long sequence of events.

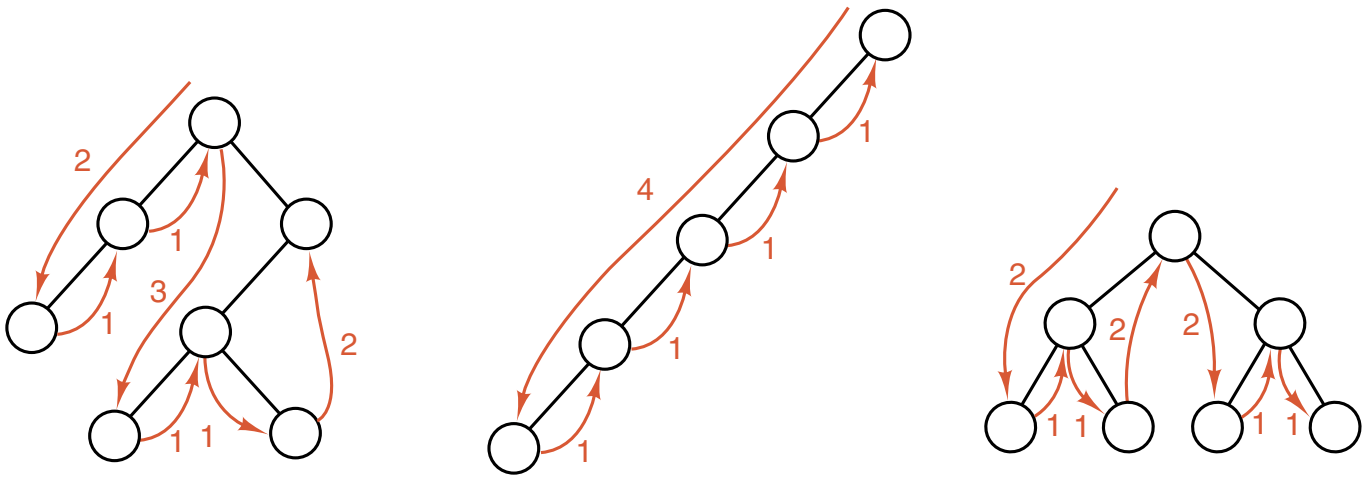
- One event in a sequence may affect the cost of later events. One task may be difficult, but it may leave a data structure in a state where the tasks that follow become much easier.

- In finance, *amortization* means to spread a large expense over a period of time. Accountants amortize a large capital expenditure over it lifetime. Insurance actuaries amortize high-risk cases over the general population.

- Amortized analysis is not the same as average-case analysis, since the former considers a sequence of *related* situations and the latter all possible *independent* situations.

Tree Traversal

Consider the inorder traversal of a binary tree, where we measure the cost of visiting one vertex as the number of branches traversed to reach that vertex from the last one visited.



(a) (b) (c)

- The best-case cost of visiting a vertex is 1, when we go from a vertex to one of its children or to its parent.
- The worst-case cost, for a tree with n vertices, is $n - 1$, as shown by the tree that is one long chain to the left, where it takes $n - 1$ branches to reach the first (leftmost) vertex.
- The amortized cost of going from each vertex to the next over the traversal of any binary tree is less than 2.

Credit Balance: Making Costs Level

- We invent a function, which we call a ***credit balance***, chosen so that it will be large when the next operation is expensive and smaller when the next operation can be done quickly.
- Think of the credit balance as helping to bear some of the cost of expensive operations. For inexpensive operations, we set aside more than the actual cost, using the excess to build up the credit balance for future use.

DEFINITION The ***amortized cost*** a_i of each operation in a sequence of m operations is defined to be $a_i = t_i + c_i - c_{i-1}$ for $i = 1, 2, \ldots, m$, where t_i is the actual cost of operation i, c_0 is the credit balance before the first operation, and c_i is the credit balance after operation i, for $1 \leq i \leq m$.

Goal: Choose the credit-balance function c_i so as to make the amortized costs a_i as nearly equal as possible, no matter how the actual costs t_i may vary.

Lemma 10.5 The total actual cost and total amortized cost of a sequence of m operations on a data structure are related by

$$\sum_{i=1}^{m} t_i = \left(\sum_{i=1}^{m} a_i \right) + c_0 - c_m.$$

Incrementing Binary Integers

Consider an algorithm that continually increments a binary (base 2) integer by 1. The cost of each step is the number of bits (binary digits) that are changed from one number to the next.

step i	*integer*	t_i	c_i	a_i
0	0 0 0 0		0	
1	0 0 0 1	1	1	2
2	0 0 1 0	2	1	2
3	0 0 1 1	1	2	2
4	0 1 0 0	3	1	2
5	0 1 0 1	1	2	2
6	0 1 1 0	2	2	2
7	0 1 1 1	1	3	2
8	1 0 0 0	4	1	2
9	1 0 0 1	1	2	2
10	1 0 1 0	2	2	2
11	1 0 1 1	1	3	2
12	1 1 0 0	3	2	2
13	1 1 0 1	1	3	2
14	1 1 1 0	2	3	2
15	1 1 1 1	1	4	2
16	0 0 0 0	4	0	0

t_i = actual cost = number of digits changed
c_i = credit-balance function = number of 1's in integer
a_i = amortized cost = $t_i + c_i - c_{i-1}$

Amortized Analysis of Splaying

Let T be a binary search tree, T_i be T as it is after step i of splaying, $T_i(x)$ be the subtree with root x in T_i, $|T_i(x)|$ be the number of nodes in $T_i(x)$, and define the ***rank*** of x to be $r_i(x) = \lg |T_i(x)|$.

The Credit Invariant

For every node x of T and after every step i of splaying, node x has credit equal to its rank $r_i(x)$.

The total ***credit balance*** for the tree is defined as the sum of the individual credits for all the nodes in the tree,

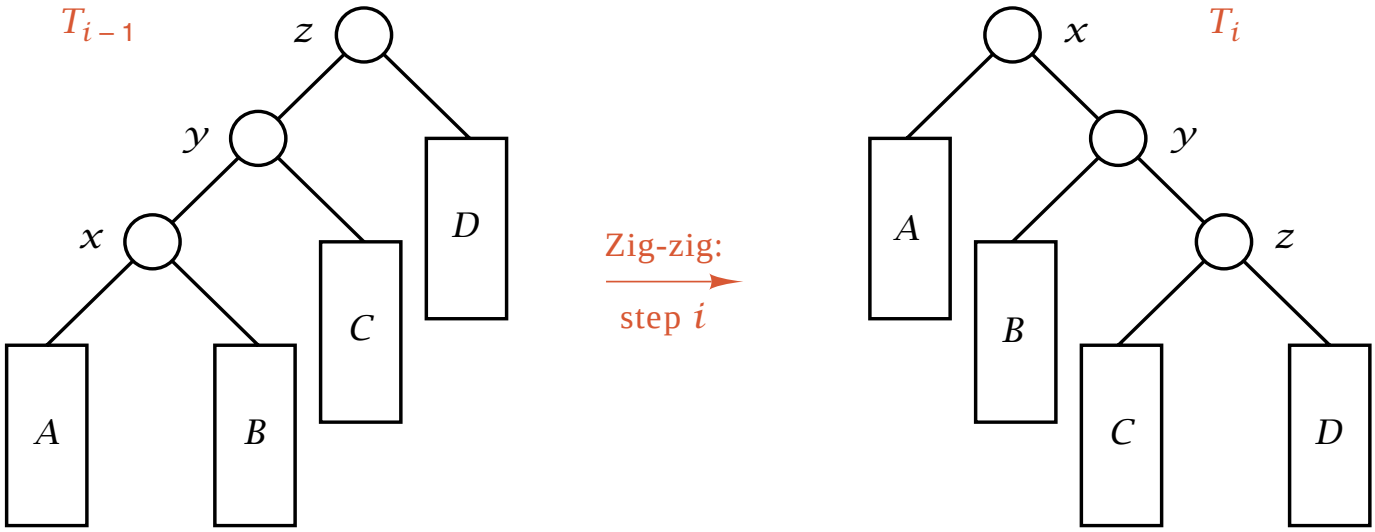
$$c_i = \sum_{x \in T_i} r_i(x).$$

If the tree is empty or contains only one node, then its credit balance is 0. As the tree grows, its credit balance increases, and this balance should reflect the work needed to build the tree.

LEMMA 10.6 If α, β, and γ are positive real numbers with $\alpha + \beta \leq \gamma$, then $\lg \alpha + \lg \beta \leq 2 \lg \gamma - 2$.

Amortized Complexity of Splay Steps

LEMMA 10.7 If the i^{th} splaying step is a zig-zig or zag-zag step at node x, then its amortized complexity a_i satisfies the inequality $a_i < 3\big(r_i(x) - r_{i-1}(x)\big)$.



LEMMA 10.8 If the i^{th} splaying step is a zig-zag or zag-zig step at node x, then its amortized complexity a_i satisfies

$$a_i < 2\big(r_i(x) - r_{i-1}(x)\big).$$

LEMMA 10.9 If the i^{th} splaying step is a zig or a zag step at node x, then its amortized complexity a_i satisfies

$$a_i < 1 + \big(r_i(x) - r_{i-1}(x)\big).$$

Amortized Complexity of Splaying

- To find the total amortized cost of a retrieval or insertion, we must add the costs of all the splay steps done during the retrieval or insertion.

Theorem 10.10 The amortized cost of an insertion or retrieval with splaying in a binary search tree with n nodes does not exceed

$$1 + 3 \lg n$$

upward moves of the target node in the tree.

- Finally, we can relate this amortized cost to the actual cost of each of a long sequence of splay insertions or retrievals.

Corollary 10.11 The total complexity of a sequence of m insertions or retrievals with splaying in a binary search tree that never has more than n nodes does not exceed

$$m(1 + 3 \lg n) + n \lg n$$

upward moves of a target node in the tree.

- In this result, each splaying step counts as two upward moves, except for zig or zag steps, which count as one move each.
- The fact that insertions and retrievals in a splay tree, over a long sequence, are *guaranteed* to take only $O(\log n)$ time is quite remarkable, given that, at any time, it is quite possible for a splay tree to degenerate into a highly unbalanced shape.

Pointers and Pitfalls

1. Consider binary search trees as an alternative to ordered lists. At the cost of an extra pointer field in each node, binary search trees allow random access (with $O(\log n)$ key comparisons) to all nodes while maintaining the flexibility of linked lists for insertions, deletions, and rearrangement.
2. Consider binary search trees as an alternative to tables. At the cost of access time that is $O(\log n)$ instead of $O(1)$, binary search trees allow traversal of the data structure in the order specified by the keys while maintaining the advantage of random access provided by tables.
3. In choosing your data structures, always carefully consider what operations will be required. Binary trees are especially appropriate when the requirements include random access, traversal in a predetermined order, and flexibility in making insertions and deletions.
4. While choosing data structures and algorithms, remain alert to the possibility of highly unbalanced binary search trees. AVL trees provide nearly perfect balance at a slight cost in computer time and space, but with considerable programming cost. If it is necessary for the tree to adapt dynamically to changes in the frequency of the data, then a splay tree may be the best choice.
5. Binary trees are defined recursively; algorithms for manipulating binary trees are usually, but not always, best written recursively.
6. Although binary trees are most commonly implemented as linked structures, remain aware of the possibility of other implementations.