Chapter 10

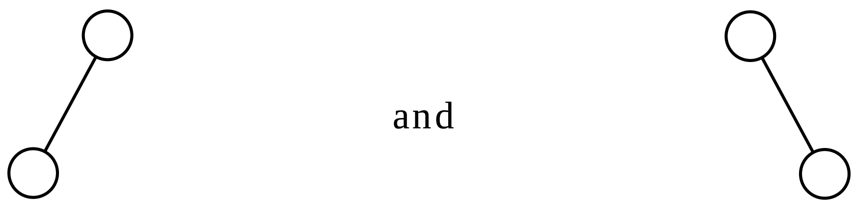
BINARY TREES

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

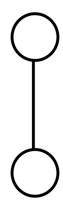
Binary Trees

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.

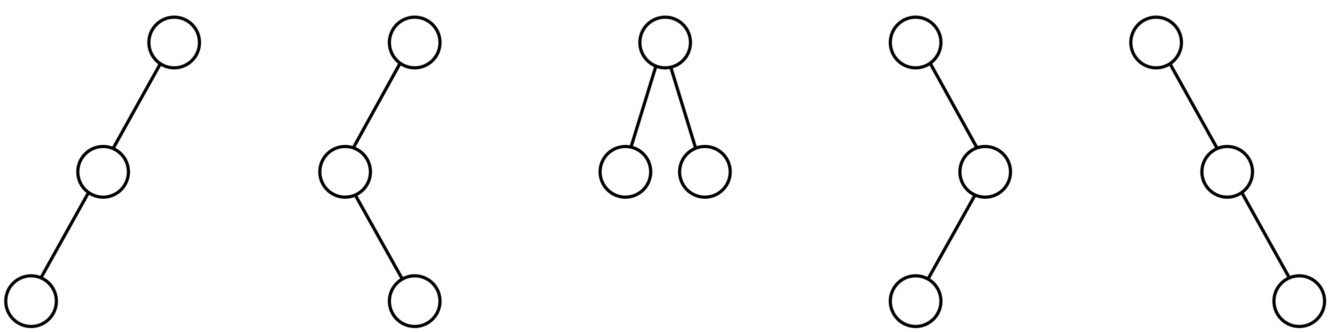
There is one empty binary tree, one binary tree with one node, and two with two nodes:



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:

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

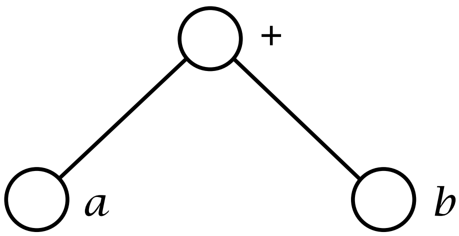
By standard convention, these are reduced to three by considering only the ways in which the left subtree is traversed before the right.

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

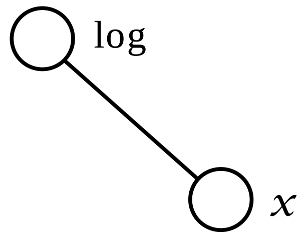
These three names are chosen according to the step at which the given node is visited.

- 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.

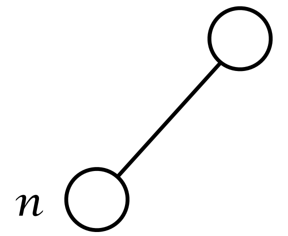
Expression Trees



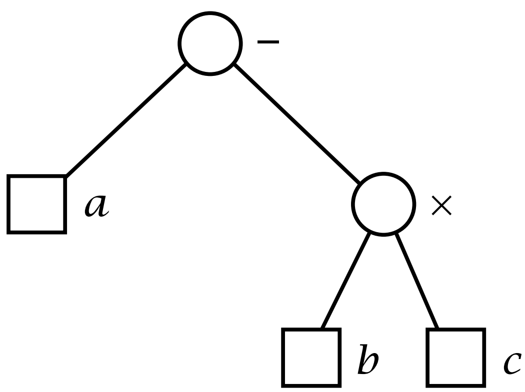
$a + b$



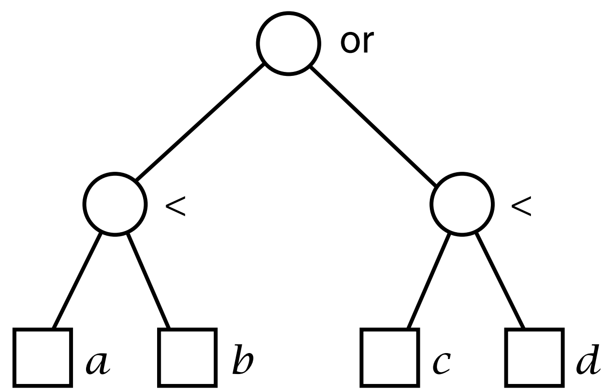
$\log x$



$n!$

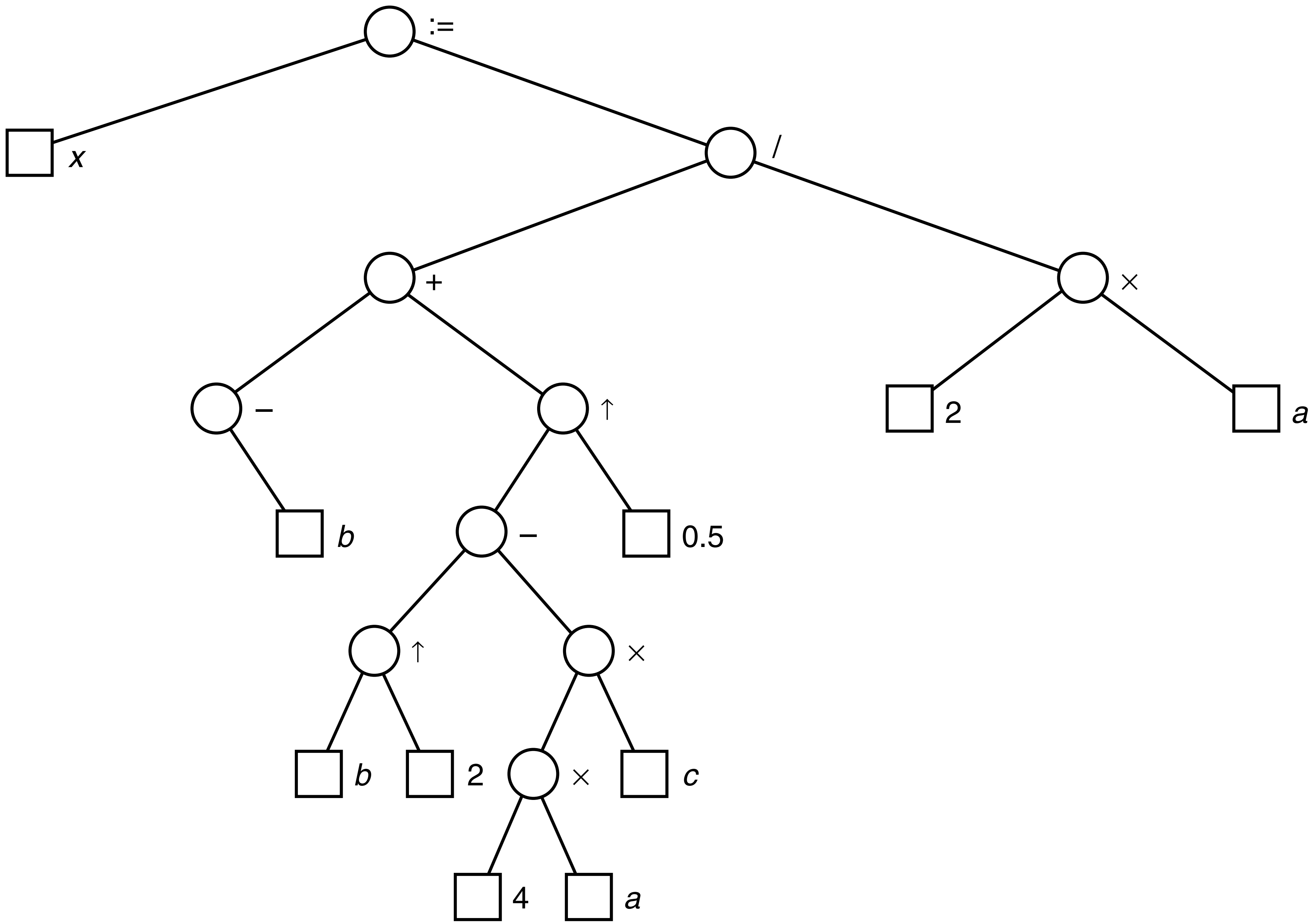


$a - (b \times c)$



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

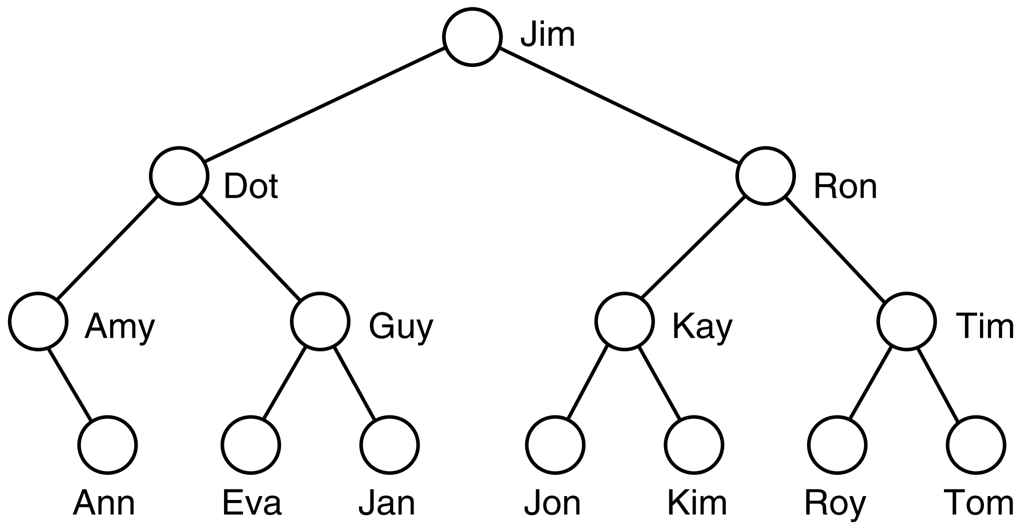
Expression:	$a + b$	$\log x$	$n!$	$a - (b \times c)$	$(a < b)$ or $(c < d)$
Preorder :	$+ \; a \; b$	$\log \; x$	$! \; n$	$- \; a \; \times \; b \; c$	or $< \; a \; b \; < \; c \; d$
Inorder :	$a \; + \; b$	$\log \; x$	$n \; !$	$a \; - \; b \; \times \; c$	$a \; < \; b$ or $c \; < \; d$
Postorder :	$a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



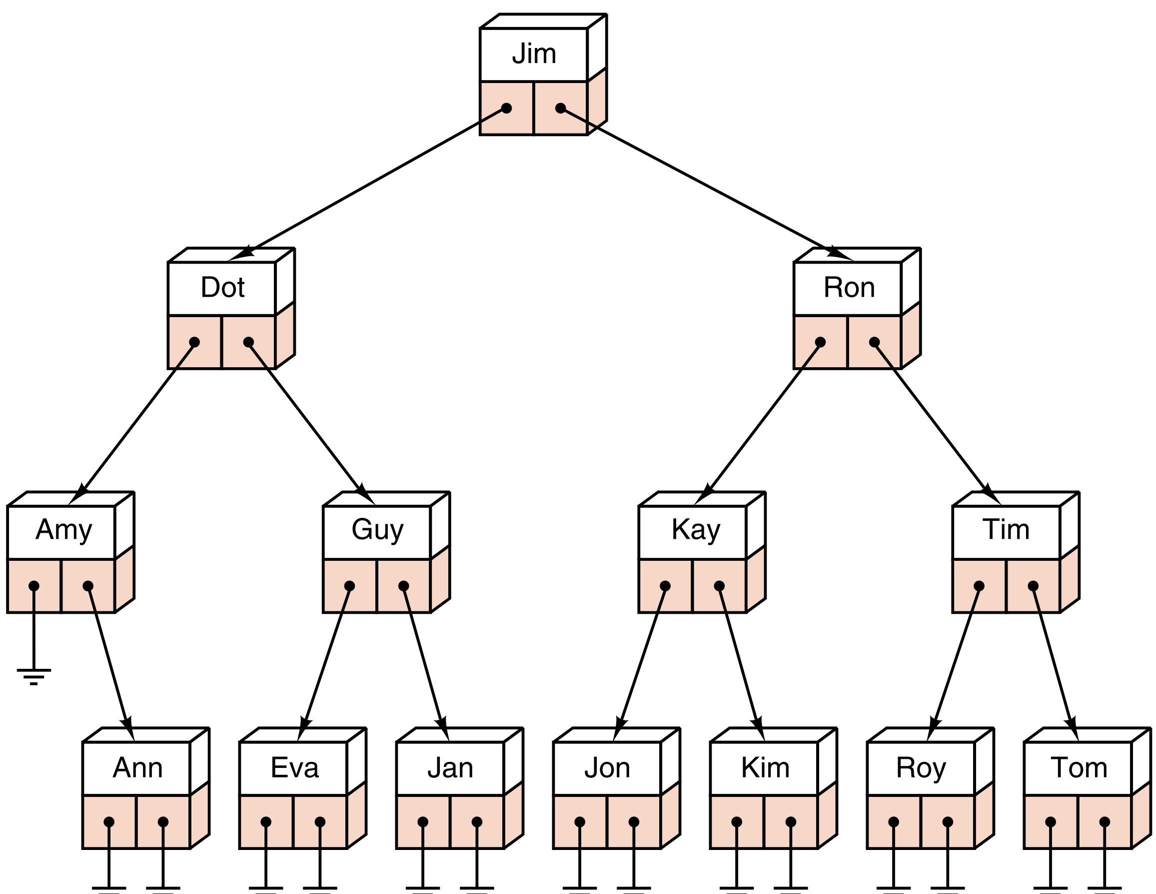
$$x := (-b + (b \uparrow 2 - 4 \times a \times c) \uparrow 0.5)/(2 \times a)$$

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 inorder 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 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);
}
```

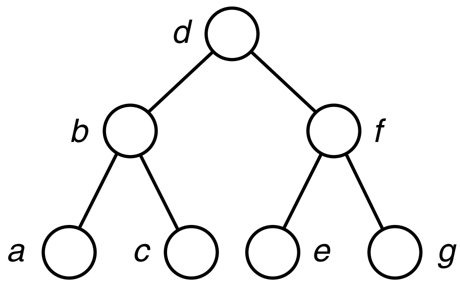
Nonrecursive version:

```
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;
}
```

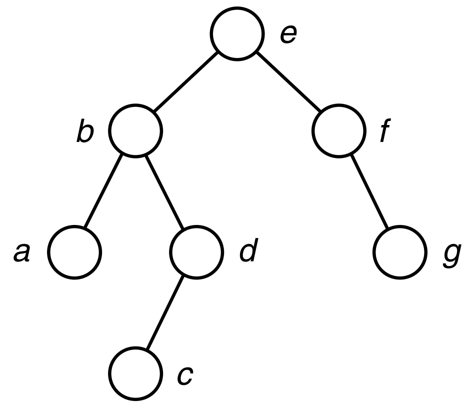
Public method for tree search:

```
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;
}
```

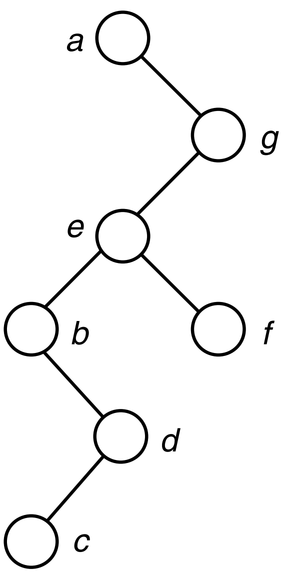
Binary Search Trees with the Same Keys



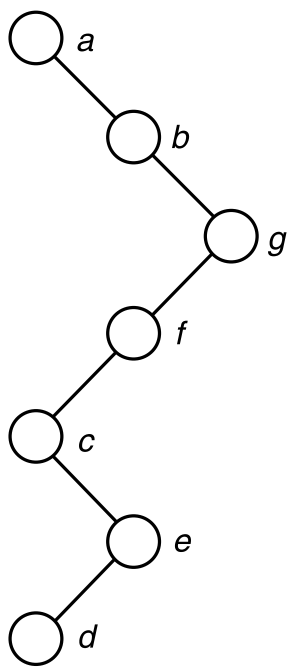
(a)



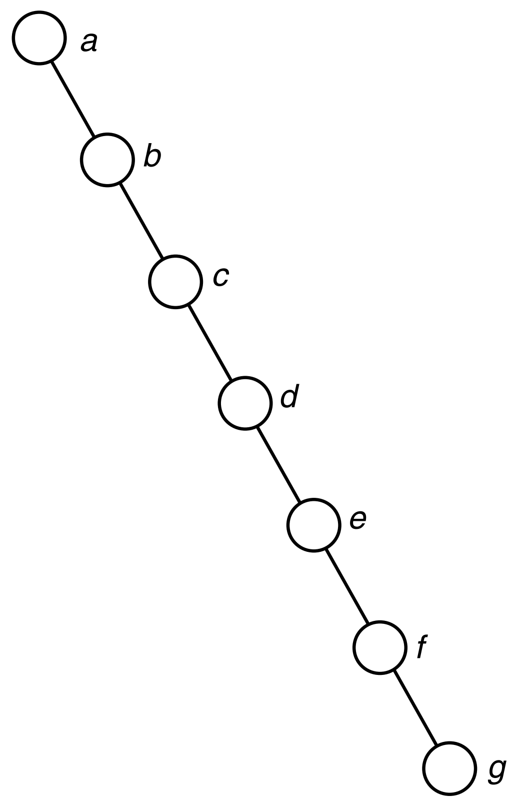
(b)



(c)



(d)



(e)

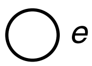
Analysis of Tree Search

- 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 $\Theta(n)$ comparsions 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.

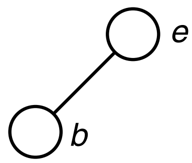
Insertion into a Binary Search Tree

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

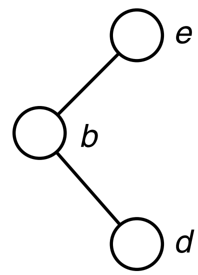
Post: If a Record with a key matching that of new_data already belongs to the Search_tree a code of duplicate_error is returned. Otherwise, the Record new_data is inserted into the tree in such a way that the properties of a binary search tree are preserved, and a code of success is returned.



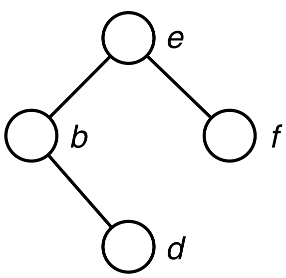
(a) Insert *e*



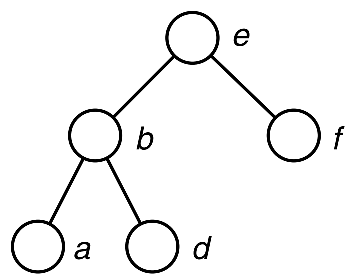
(b) Insert *b*



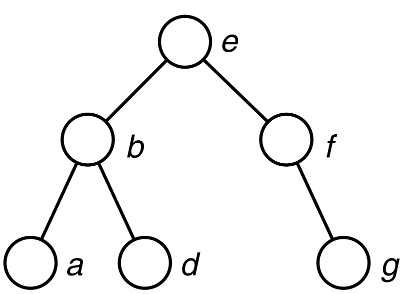
(c) Insert *d*



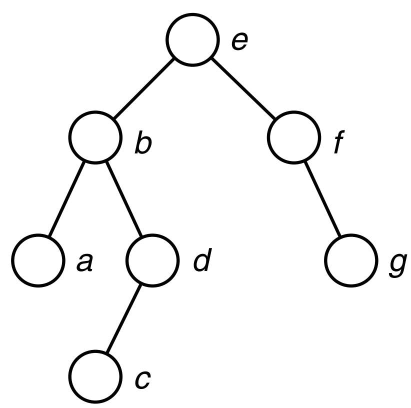
(d) Insert *f*



(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.