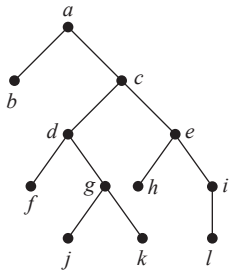
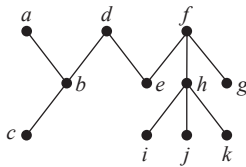


The **eccentricity** of a vertex in an unrooted tree is the length of the longest simple path beginning at this vertex. A vertex is called a **center** if no vertex in the tree has smaller eccentricity than this vertex. In Exercises 39–41 find every vertex that is a center in the given tree.

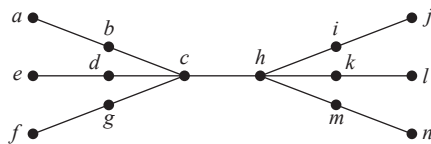
39.



40.



41.



42. Show that a center should be chosen as the root to produce a rooted tree of minimal height from an unrooted tree.

\*43. Show that a tree has either one center or two centers that are adjacent.

44. Show that every tree can be colored using two colors.

The **rooted Fibonacci trees**  $T_n$  are defined recursively in the following way.  $T_1$  and  $T_2$  are both the rooted tree consisting of a single vertex, and for  $n = 3, 4, \dots$ , the rooted tree  $T_n$  is constructed from a root with  $T_{n-1}$  as its left subtree and  $T_{n-2}$  as its right subtree.

45. Draw the first seven rooted Fibonacci trees.

\*46. How many vertices, leaves, and internal vertices does the rooted Fibonacci tree  $T_n$  have, where  $n$  is a positive integer? What is its height?

47. What is wrong with the following “proof” using mathematical induction of the statement that every tree with  $n$  vertices has a path of length  $n - 1$ . *Basis step:* Every tree with one vertex clearly has a path of length 0. *Inductive step:* Assume that a tree with  $n$  vertices has a path of length  $n - 1$ , which has  $u$  as its terminal vertex. Add a vertex  $v$  and the edge from  $u$  to  $v$ . The resulting tree has  $n + 1$  vertices and has a path of length  $n$ . This completes the inductive step.

\*48. Show that the average depth of a leaf in a binary tree with  $n$  vertices is  $\Omega(\log n)$ .

## 11.2 Applications of Trees

### Introduction

We will discuss three problems that can be studied using trees. The first problem is: How should items in a list be stored so that an item can be easily located? The second problem is: What series of decisions should be made to find an object with a certain property in a collection of objects of a certain type? The third problem is: How should a set of characters be efficiently coded by bit strings?

### Binary Search Trees



Links

Searching for items in a list is one of the most important tasks that arises in computer science. Our primary goal is to implement a searching algorithm that finds items efficiently when the items are totally ordered. This can be accomplished through the use of a **binary search tree**, which is a binary tree in which each child of a vertex is designated as a right or left child, no vertex has more than one right child or left child, and each vertex is labeled with a key, which is one of the items. Furthermore, vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

This recursive procedure is used to form the binary search tree for a list of items. Start with a tree containing just one vertex, namely, the root. The first item in the list is assigned as the key of the root. To add a new item, first compare it with the keys of vertices already in the tree, starting at the root and moving to the left if the item is less than the key of the respective vertex if this vertex has a left child, or moving to the right if the item is greater than the key of the

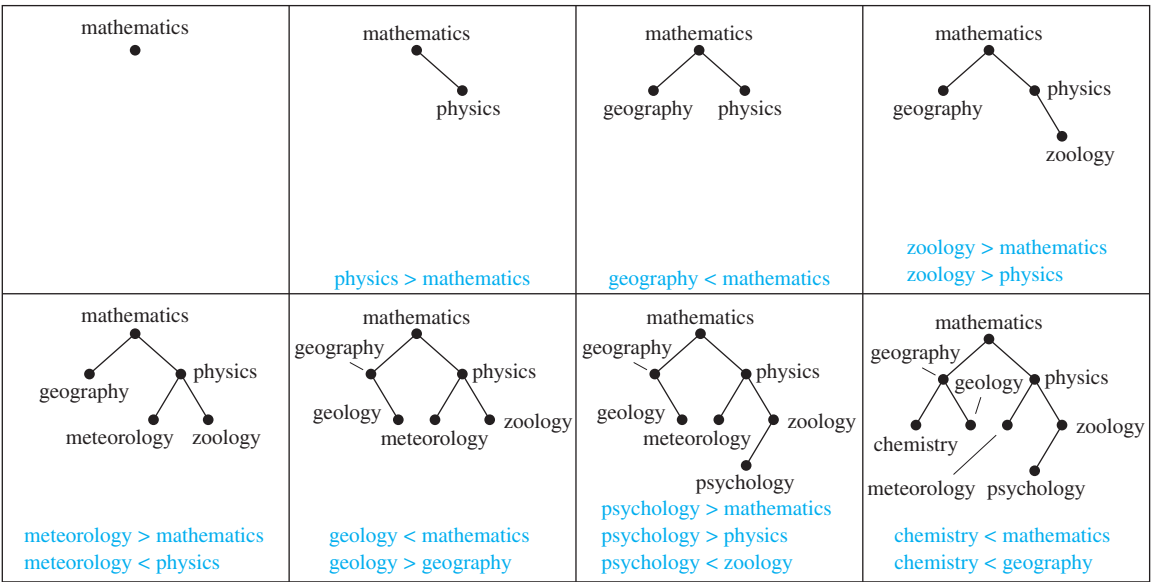
12/31 15:46  
| read  
16:03  
| notes  
16:15

respective vertex if this vertex has a right child. When the item is less than the respective vertex and this vertex has no left child, then a new vertex with this item as its key is inserted as a new left child. Similarly, when the item is greater than the respective vertex and this vertex has no right child, then a new vertex with this item as its key is inserted as a new right child. We illustrate this procedure with Example 1.

**EXAMPLE 1** Form a binary search tree for the words *mathematics*, *physics*, *geography*, *zoology*, *meteorology*, *geology*, *psychology*, and *chemistry* (using alphabetical order).

**Solution:** Figure 1 displays the steps used to construct this binary search tree. The word *mathematics* is the key of the root. Because *physics* comes after *mathematics* (in alphabetical order), add a right child of the root with key *physics*. Because *geography* comes before *mathematics*, add a left child of the root with key *geography*. Next, add a right child of the vertex with key *physics*, and assign it the key *zoology*, because *zoology* comes after *mathematics* and after *physics*. Similarly, add a left child of the vertex with key *physics* and assign this new vertex the key *meteorology*. Add a right child of the vertex with key *geography* and assign this new vertex the key *geology*. Add a left child of the vertex with key *zoology* and assign it the key *psychology*. Add a left child of the vertex with key *geography* and assign it the key *chemistry*. (The reader should work through all the comparisons needed at each step.)

Once we have a binary search tree, we need a way to locate items in the binary search tree, as well as a way to add new items. Algorithm 1, an insertion algorithm, actually does both of these tasks, even though it may appear that it is only designed to add vertices to a binary search tree. That is, Algorithm 1 is a procedure that locates an item  $x$  in a binary search tree if it is present, and adds a new vertex with  $x$  as its key if  $x$  is not present. In the pseudocode,  $v$  is the vertex currently under examination and  $label(v)$  represents the key of this vertex. The algorithm begins by examining the root. If  $x$  equals the key of  $v$ , then the algorithm has found the location of  $x$  and terminates; if  $x$  is less than the key of  $v$ , we move to the left child of  $v$  and repeat the procedure; and if  $x$  is greater than the key of  $v$ , we move to the right child of  $v$  and repeat the procedure. If at any step we attempt to move to a child that is not present, we know that  $x$  is not present in the tree, and we add a new vertex as this child with  $x$  as its key.



**FIGURE 1** Constructing a Binary Search Tree.

**ALGORITHM 1** Locating an Item in or Adding an Item to a Binary Search Tree.

```

procedure insertion( $T$ : binary search tree,  $x$ : item)
 $v$  := root of  $T$ 
{a vertex not present in  $T$  has the value null}
while  $v \neq \text{null}$  and  $\text{label}(v) \neq x$ 
  if  $x < \text{label}(v)$  then
    if left child of  $v \neq \text{null}$  then  $v$  := left child of  $v$ 
    else add new vertex as a left child of  $v$  and set  $v$  := null
  else
    if right child of  $v \neq \text{null}$  then  $v$  := right child of  $v$ 
    else add new vertex as a right child of  $v$  and set  $v$  := null
if root of  $T = \text{null}$  then add a vertex  $v$  to the tree and label it with  $x$ 
else if  $v$  is null or  $\text{label}(v) \neq x$  then label new vertex with  $x$  and let  $v$  be this new vertex
return  $v$  { $v$  = location of  $x$ }

```

Example 2 illustrates the use of Algorithm 1 to insert a new item into a binary search tree.

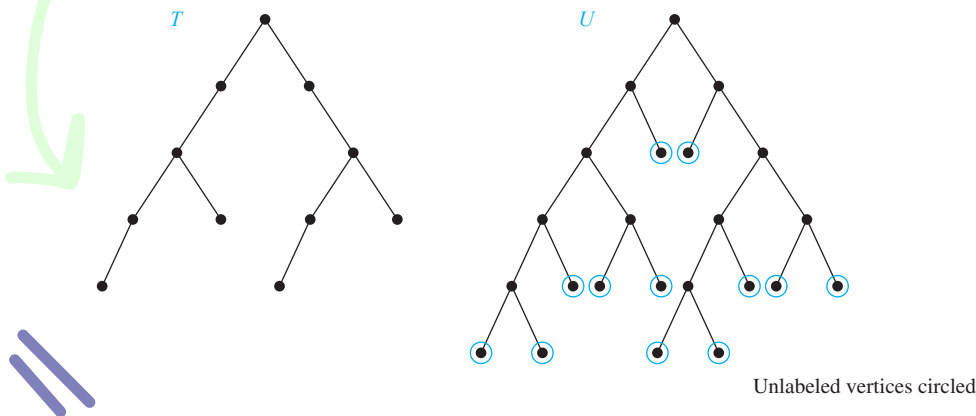
**EXAMPLE 2** Use Algorithm 1 to insert the word *oceanography* into the binary search tree in Example 1.

**Solution:** Algorithm 1 begins with  $v$ , the vertex under examination, equal to the root of  $T$ , so  $\text{label}(v) = \text{mathematics}$ . Because  $v \neq \text{null}$  and  $\text{label}(v) = \text{mathematics} < \text{oceanography}$ , we next examine the right child of the root. This right child exists, so we set  $v$ , the vertex under examination, to be this right child. At this step we have  $v \neq \text{null}$  and  $\text{label}(v) = \text{physics} > \text{oceanography}$ , so we examine the left child of  $v$ . This left child exists, so we set  $v$ , the vertex under examination, to this left child. At this step, we also have  $v \neq \text{null}$  and  $\text{label}(v) = \text{metereology} < \text{oceanography}$ , so we try to examine the right child of  $v$ . However, this right child does not exist, so we add a new vertex as the right child of  $v$  (which at this point is the vertex with the key *metereology*) and we set  $v$  := *null*. We now exit the **while** loop because  $v = \text{null}$ . Because the root of  $T$  is not *null* and  $v = \text{null}$ , we use the **else if** statement at the end of the algorithm to label our new vertex with the key *oceanography*. ◀

We will now determine the computational complexity of this procedure. Suppose we have a binary search tree  $T$  for a list of  $n$  items. We can form a full binary tree  $U$  from  $T$  by adding unlabeled vertices whenever necessary so that every vertex with a key has two children. This is illustrated in Figure 2. Once we have done this, we can easily locate or add a new item as a key without adding a vertex.

The most comparisons needed to add a new item is the length of the longest path in  $U$  from the root to a leaf. The internal vertices of  $U$  are the vertices of  $T$ . It follows that  $U$  has  $n$  internal vertices. We can now use part (ii) of Theorem 4 in Section 11.1 to conclude that  $U$  has  $n + 1$  leaves. Using Corollary 1 of Section 11.1, we see that the height of  $U$  is greater than or equal to  $h = \lceil \log(n + 1) \rceil$ . Consequently, it is necessary to perform at least  $\lceil \log(n + 1) \rceil$  comparisons to add some item. Note that if  $U$  is balanced, its height is  $\lceil \log(n + 1) \rceil$  (by Corollary 1 of Section 11.1). Thus, if a binary search tree is balanced, locating or adding an item requires no more than  $\lceil \log(n + 1) \rceil$  comparisons. A binary search tree can become unbalanced as items are added to it. Because balanced binary search trees give optimal worst-case complexity for binary searching, algorithms have been devised that rebalance binary search trees as items are added. The interested reader can consult references on data structures for the description of such algorithms.

Why  
do  
this?



**FIGURE 2** Adding Unlabeled Vertices to Make a Binary Search Tree Full.

## Decision Trees



Rooted trees can be used to model problems in which a series of decisions leads to a solution. For instance, a binary search tree can be used to locate items based on a series of comparisons, where each comparison tells us whether we have located the item, or whether we should go right or left in a subtree. A rooted tree in which each internal vertex corresponds to a decision, with a subtree at these vertices for each possible outcome of the decision, is called a **decision tree**. The possible solutions of the problem correspond to the paths to the leaves of this rooted tree. Example 3 illustrates an application of decision trees.

**EXAMPLE 3** Suppose there are seven coins, all with the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which of the eight coins is the counterfeit one? Give an algorithm for finding this counterfeit coin.




**Solution:** There are three possibilities for each weighing on a balance scale. The two pans can have equal weight, the first pan can be heavier, or the second pan can be heavier. Consequently, the decision tree for the sequence of weighings is a 3-ary tree. There are at least eight leaves in the decision tree because there are eight possible outcomes (because each of the eight coins can be the counterfeit lighter coin), and each possible outcome must be represented by at least one leaf. The largest number of weighings needed to determine the counterfeit coin is the height of the decision tree. From Corollary 1 of Section 11.1 it follows that the height of the decision tree is at least  $\lceil \log_3 8 \rceil = 2$ . Hence, at least two weighings are needed.

It is possible to determine the counterfeit coin using two weighings. The decision tree that illustrates how this is done is shown in Figure 3. ◀

**THE COMPLEXITY OF COMPARISON-BASED SORTING ALGORITHMS** Many different sorting algorithms have been developed. To decide whether a particular sorting algorithm is efficient, its complexity is determined. Using decision trees as models, a lower bound for the worst-case complexity of sorting algorithms that are based on binary comparisons can be found.

We can use decision trees to model sorting algorithms and to determine an estimate for the worst-case complexity of these algorithms. Note that given  $n$  elements, there are  $n!$  possible orderings of these elements, because each of the  $n!$  permutations of these elements can be the correct order. The sorting algorithms studied in this book, and most commonly used sorting algorithms, are based on binary comparisons, that is, the comparison of two elements at a time. The result of each such comparison narrows down the set of possible orderings. Thus, a sorting algorithm based on binary comparisons can be represented by a binary decision tree in which each internal vertex represents a comparison of two elements. Each leaf represents one of the  $n!$  permutations of  $n$  elements.

each level. For example, once we have found the values of the three children of the root, which are 1,  $-1$ , and  $-1$ , we find the value of the root by computing  $\max(1, -1, -1) = 1$ . Because the value of the root is 1, it follows that the first player wins when both players follow a minmax strategy. 

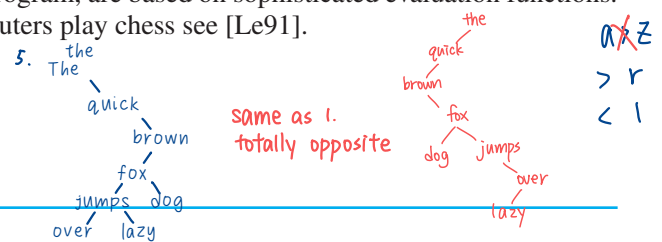
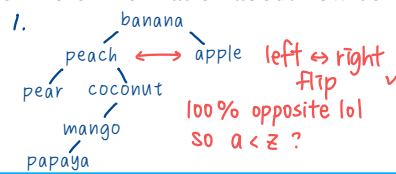
Game trees for some well-known games can be extraordinarily large, because these games have many different possible moves. For example, the game tree for chess has been estimated to have as many as  $10^{100}$  vertices! It may be impossible to use Theorem 3 directly to study a game because of the size of the game tree. Therefore, various approaches have been devised to help determine good strategies and to determine the outcome of such games. One useful technique, called *alpha-beta pruning*, eliminates much computation by pruning portions of the game tree that cannot affect the values of ancestor vertices. (For information about alpha-beta pruning, consult [Gr90].) Another useful approach is to use *evaluation functions*, which estimate the value of internal vertices in the game tree when it is not feasible to compute these values exactly. For example, in the game of tic-tac-toe, as an evaluation function for a position, we may use the number of files (rows, columns, and diagonals) containing no Os (used to indicate moves of the second player) minus the number of files containing no Xs (used to indicate moves of the first player). This evaluation function provides some indication of which player has the advantage in the game. Once the values of an evaluation function are inserted, the value of the game can be computed following the rules used for the minmax strategy. Computer programs created to play chess, such as the famous Deep Blue program, are based on sophisticated evaluation functions. For more information about how computers play chess see [Le91].

Chess programs on smartphones can now play at the grandmaster level.



12/31 16:19  
16:40  
1 correct  
16:49

## Exercises



Describe an algorithm to find the counterfeit coin using this number of weighings.

- Build a binary search tree for the words *banana*, *peach*, *apple*, *pear*, *coconut*, *mango*, and *papaya* using alphabetical order. (didn't specify the root?)
- Build a binary search tree for the words *oenology*, *phrenology*, *campanology*, *ornithology*, *ichthyology*, *limnology*, *alchemy*, and *astrology* using alphabetical order.
- How many comparisons are needed to locate or to add each of these words in the search tree for Exercise 1, starting fresh each time?
  - pear* 2 3
  - banana* 1 ✓
  - kumquat* 3 4
  - orange* 4 5 similar to (c)
- How many comparisons are needed to locate or to add each of the words in the search tree for Exercise 2, starting fresh each time?
  - palmistry*
  - etymology*
  - paleontology*
  - glaciology*
- Using alphabetical order, construct a binary search tree for the words in the sentence "The quick brown fox jumps over the lazy dog."
- How many weighings of a balance scale are needed to find a lighter counterfeit coin among four coins? Describe an algorithm to find the lighter coin using this number of weighings.
- How many weighings of a balance scale are needed to find a counterfeit coin among four coins if the counterfeit coin may be either heavier or lighter than the others?
- How many weighings of a balance scale are needed to find a counterfeit coin among eight coins if the counterfeit coin is either heavier or lighter than the others? Describe an algorithm to find the counterfeit coin using this number of weighings.
- How many weighings of a balance scale are needed to find a counterfeit coin among 12 coins if the counterfeit coin is lighter than the others? Describe an algorithm to find the lighter coin using this number of weighings.
- One of four coins may be counterfeit. If it is counterfeit, it may be lighter or heavier than the others. How many weighings are needed, using a balance scale, to determine whether there is a counterfeit coin, and if there is, whether it is lighter or heavier than the others? Describe an algorithm to find the counterfeit coin and determine whether it is lighter or heavier using this number of weighings.
- Find the least number of comparisons needed to sort four elements and devise an algorithm that sorts these elements using this number of comparisons.
- Find the least number of comparisons needed to sort five elements and devise an algorithm that sorts these elements using this number of comparisons.