Module 5

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 5 Study Guide and Deliverables

Topics:	• Search Rrees • Sorting and Selection • Huffman Code and Greedy Algorithm • Dynamic Programming
Readings:	• Module 5 online content • Textbook: Chapter 11 (11.1, 11.2, 11.3), Chapter 12, Chapter 13 (13.4, 13.5)
Assignments:	• Assignment 5 due **Tuesday, April 19 at 6:00 AM ET**
Assessments:	• Quiz 5 due **Tuesday, April 19 at 6:00 AM ET**
Live Classrooms:	• **Tuesday, April 12 from 8:00-9:30 PM** • **Thursday, April 14 from 8:00-10:00 PM** • Another one-hour live office hour session led by your facilitator: TBD

Module 5 Learning Objectives

In this section, we discuss search trees, AVL trees, and sorting algorithms. We also briefly illustrate the greedy method and dynamic programming using simple problems.

After successfully completing this module, you will be able to:

1. Illustrate binary search tree operations, including search, insertion, and deletion operations.
2. Implement binary search tree operations, including search, insertion, and deletion operations.
3. Implement a sorted map using a binary search tree.
4. Illustrate rotation operations of balanced trees.
5. Illustrate insertion and deletion operations of AVL trees.
6. Implement merge-sort using an array or a linked list.
7. Illustrate partitioning of a sequence for quick-sort.
8. Implement quick-sort using an array.
9. Analyze the running time of quick-sort.

10. Illustrate bucket-sort.
11. Illustrate radix-sort.
12. Analyze and compare the running times of sorting algorithms.
13. Illustrate Huffman code algorithm.
14. Illustrate dynamic programming using a simple example.

Section 5.1 Search Trees

Section 5.1. Search Trees

Overview

In this section, we discuss various search trees, including binary search trees and AVL trees.

Section 5.1.1. Binary Search Trees

In this section we discuss the general properties and basic operations of binary search trees and we illustrate how to implement a sorted map using a binary search tree.

The three fundamental operations of a map are as follow (refer to Section 4.2.1.1):

- get(k)—Returns the value v associated with the key k, if such entry exists. Returns *null,* otherwise.
- put(k, v)—If there is no entry in M with a key equal to k, then adds the entry (k, v) to M and returns *null*. Otherwise, replaces the existing value associated with the key k with v and returns the old value.
- remove(k)—Removes from M the entry with the key k and returns its value. If there is not entry in M with the key k, returns *null*.

A sorted-map ADT defines additional operations, such as *firstEntry*, *floorEntry*, and *subMap* (refer to Section 4.2.3).

A binary search tree is a *proper binary tree* (recall that, in a *proper binary tree*, each node has zero or two children) with the properties listed below:

The following is true for each internal position p with a (k, v) pair:

- Keys stored in the left subtree of p are less than k.
- Keys stored in the right subtree of p are greater than k.

Note that, here, we are discussing implementation of a sorted map in which all keys are distinct. If duplicate keys are allowed, the first property can be changed as follows:

- Keys stored in the left subtree of *p* are less than or equal to k.

Figure 5.1 is an example of a binary search tree. In the figure, for simplicity, only keys are shown. Leaf nodes are sentinel nodes, which are only "placeholders." In the figure, a leaf node is shown as a small square.

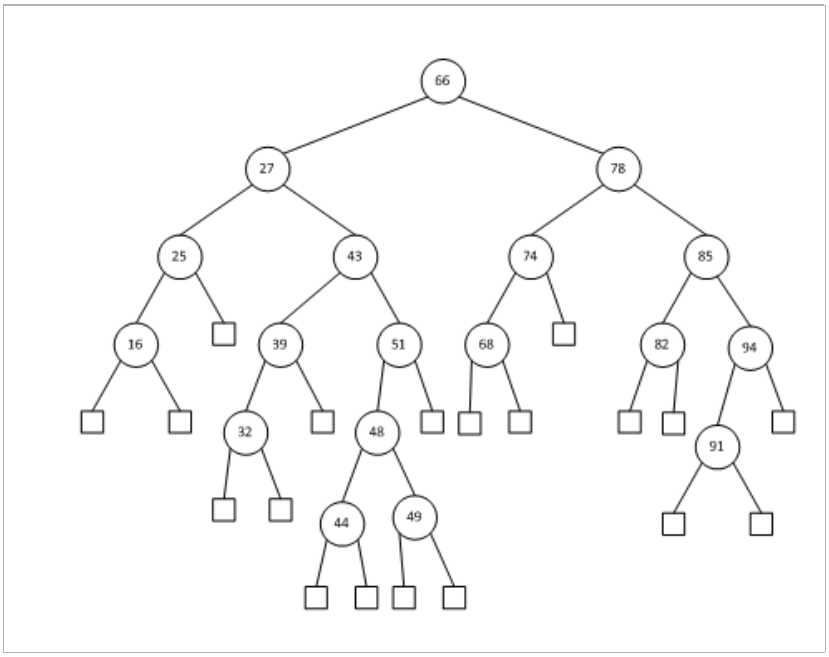


Figure 5.1. A binary search tree storing integers.

Section 5.1.1.1. Searching

The binary-search properties stated in the previous section make a search of a binary search tree very efficient. Assume that we are searching a binary search tree for an entry with the key k. We start at the root node and, at each node p, we compare k with p's key. If k is less than p's key, we move down to the left subtree of p. If k is greater than p's key, we move down to the right subtree of p. We repeat this process until we find k or reach a leaf node. If we reach a leaf node, that indicates that the search was unsuccessful.

The following figure is an example of a successful search for the key 48. The search path and the node with the key 48 are highlighted.

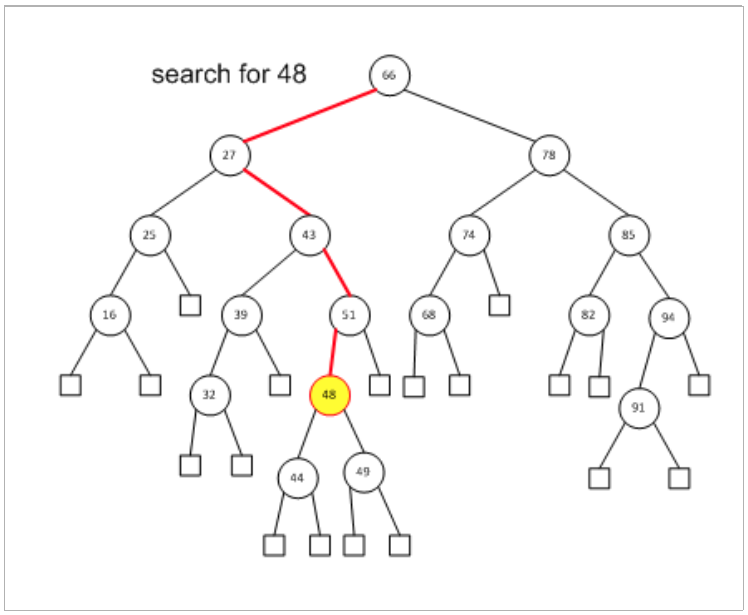


Figure 5.2. Example of a successful search.

An example of an unsuccessful search is illustrated below. This figure represents the search for an entry with the key 70 that ends up at leaf node.

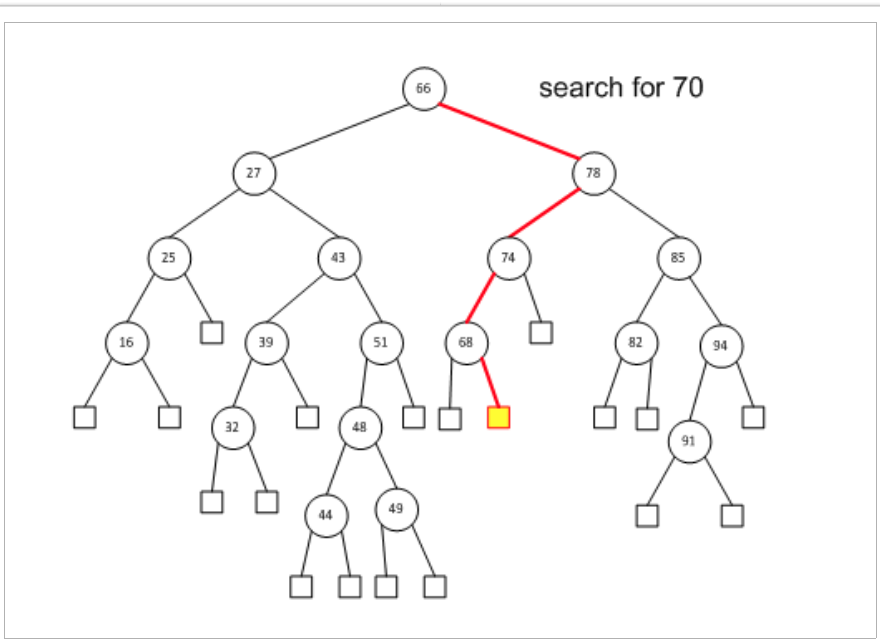


Figure 5.3. Example of an unsuccessful search.

The following is a recursive-binary search algorithm:

Code Segment 5.1

```
Algorithm TreeSearch(p, k)
if p is external then              // unsuccessful search
  return p
else if k == key(p)                // successful search
```

```
    return p
  else if k < key(p)
    return TreeSearch(left(p), k)     // recurse on left subtree
  else
    return TreeSearch(right(p), k)    // recurse on right subtree
```

The running time of a binary search depends on the height of the tree. The length of a search path, from the root node to a leaf node, is the height of the tree, *h*. The operation at each node needs $O(1)$. Therefore, the running time of a search is $O(h)$.

The search operation will be used by the *get*, *insertion*, and *deletion* methods of the sorted-map ADT.

Section 5.1.1.2. Insertions

When we insert an entry with a (k, v) pair, we first perform a search operation. If an entry with the key k is found (i.e., the search is successful), the existing value is replaced with the new value v. If there is no entry with the key k, then we add an entry at the leaf node where the unsuccessful search ended up. The leaf node is replaced with the new entry and two sentinel nodes are created and made leaf nodes of the new entry. The following figure shows the insertion of an entry with the key 30, which was not in the original tree:

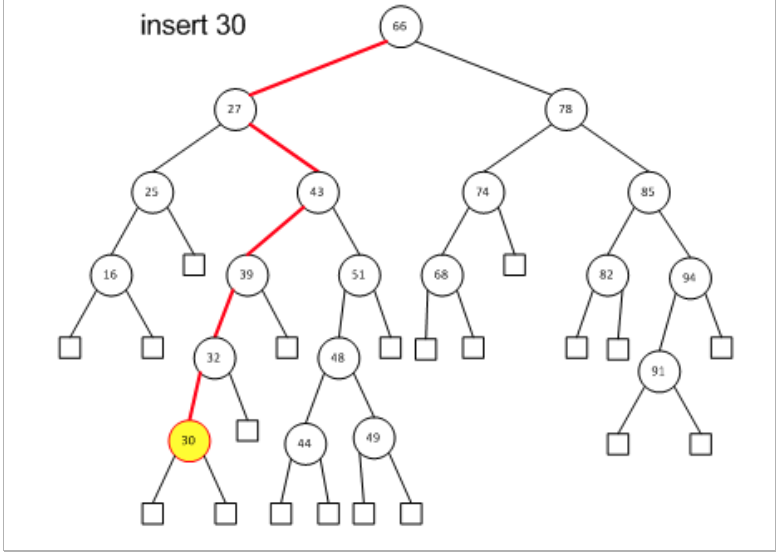


Figure 5.4. Illustration of an insertion.

The worst case occurs when the entry with the key *k* does not exist in the tree. In this case, the search goes all the way down to a leaf node. Since adding a node at the leaf level takes $O(1)$, the total running time of the insertion operation is $O(h)$.

Section 5.1.1.3. Deletions

The deletion operation is slightly more complex than the insertion operation. This is because the action of deleting depends on the location of the node to be deleted.

Suppose that we want to delete an entry with a (k, v) pair. The first step is to perform a search for the key k. If we reach a leaf

node, then there is no node with that key, so nothing needs to be done. If not, we need to consider the following two cases. Let p be the node to be deleted.

In the first case, at most one child of p is an internal node. If both children are leaf nodes, then p is replaced with a leaf node. If p has one internal-node child, then that child node replaces p. Let r be the child of p, which is internal. Then the leaf node, which is a sibling of r, is removed first. After that, r is promoted to replace p and brings with it the whole subtree rooted at r.

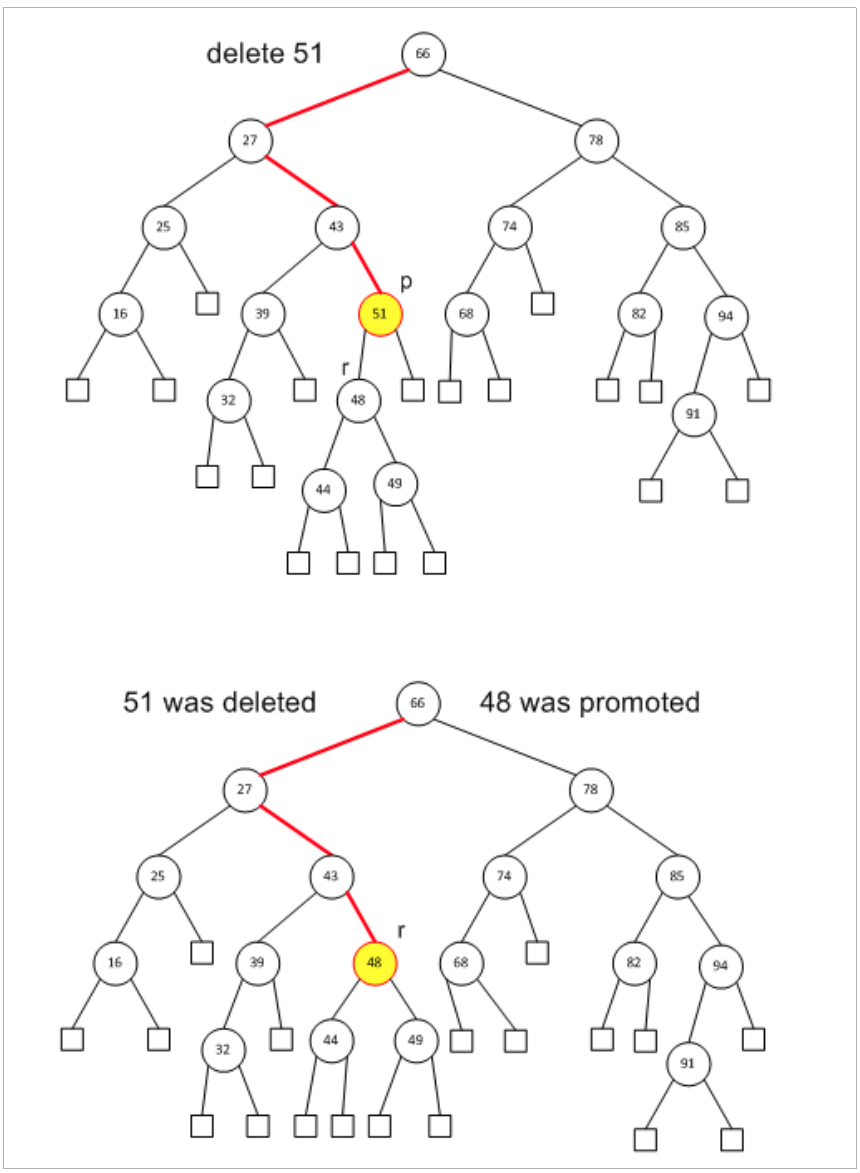


Figure 5.5. Illustration of a deletion: at most one child is an internal node.

In the second case, p has two children, both of which are internal.

- First, we find the node r that has the greatest key that is strictly less than p's key. This node is called the *predecessor* of p in the ordering of keys. The predecessor is always the rightmost node in p's left subtree.
- We let r replace p.
- Since r is the rightmost node in p's left subtree, it does not have a right child. It has only a left child. The node r is removed and the subtree rooted at r's left child is promoted to r's position.

The deletion method is illustrated in the following figure:

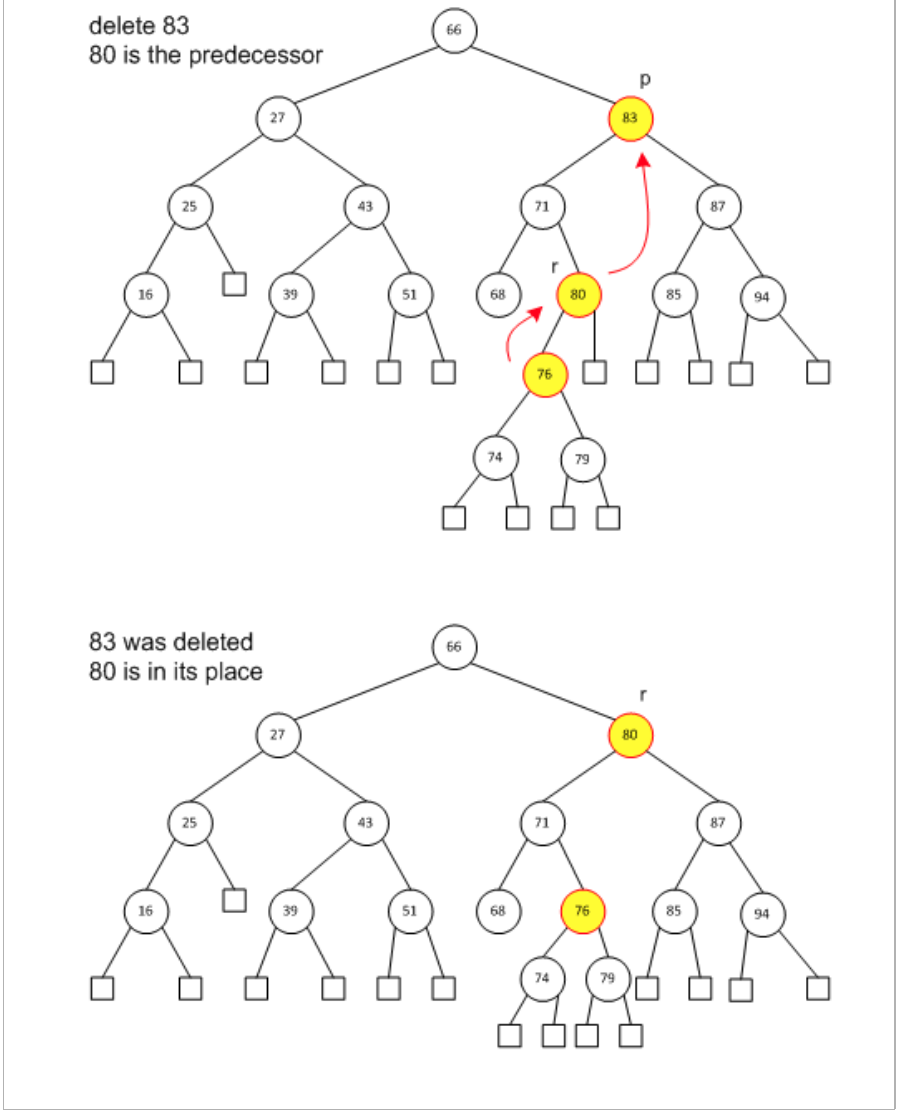


Figure 5.6. Illustration of a deletion: both children are internal nodes.

The deletion operation performs a search operation, which takes $O(h)$. Finding the predecessor of the node to be deleted, if needed, takes $O(h)$ at most. Other operations, such as replacing a node with another node or promoting a node, take $O(1)$. So, the total running time of the deletion operation takes $O(h)$.

Section 5.1.1.4. Java Implementation

In this section, we briefly discuss a Java implementation of the sorted-map ADT using a binary search tree. It is implemented as the *TreeMap* class, which extends the *AbstractSortedMap* class. The *AbstractSortedMap* class includes the implementation of the following comparison methods:

Code Segment 5.2

```
1   /** Method for comparing two entries according to key */
2   protected int compare(Entry<K,V> a, Entry<K,V> b) {
3     return comp.compare(a.getKey(), b.getKey());
4   }

5   /** Method for comparing a key and an entry's key */
6   protected int compare(K a, Entry<K,V> b) {
7     return comp.compare(a, b.getKey());
8   }

9   /** Method for comparing a key and an entry's key */
10  protected int compare(Entry<K,V> a, K b) {
11    return comp.compare(a.getKey(), b);
12  }

13  /** Method for comparing two keys */
14  protected int compare(K a, K b) {
15    return comp.compare(a, b);
16  }
```

Here, *comp* is a comparator object, which is an instance variable in the *AbstractSortedMap* class. The *TreeMap* class inherits all these methods.

In this implementation, a *BalanceableBinaryTree* object is used as a storage. The *BlanceableBinaryTree* class is defined as a nested class within the *TreeMap* class. It extends the *LinkedBinaryTree* class.

The *TreeMap* class implements the binary-search tree operation discussed in the previous section, as well as sorted-map operations (refer to Section 4.2.3). In this implementation, a proper binary tree is used as a binary search tree. All leaf nodes are sentinel nodes and internal nodes store map entries. We now discuss the *treeSearch*, *get*, *put*, and *remove* methods.

The *treeSearch* method receives two arguments. The first argument is the position of the node, *p*, where the search must begin. The second argument is the key, *key*, of the entry that is searched for. If an entry with the key is found in the subtree rooted at *p*, it is returned. Otherwise, the leaf node where the search ended is returned.

The Java code is shown below:

Code Segment 5.3

```
1   private Position<Entry<K,V>> treeSearch(Position<Entry<K,V>>  p, K key) {
2     if (isExternal(p))
3       return p;                          // key not found; return the final leaf
4     int comp = compare(key, p.getElement());
5     if (comp == 0)
6       return p;                          // key found; return its position
7     else if (comp < 0)
8       return treeSearch(left(p), key);   // search left subtree
9     else
10      return treeSearch(right(p), key);  // search right subtree
11  }
```

Line 2 checks whether the search ended up at a leaf node, which would indicate an unsuccessful search. If it did, then the leaf node, which is a sentinel node, is returned in line 3. Line 4 compares the given key with the key of the node that is being considered using the *compare* method inherited from the *AbstractSortedMap* class. If they are equivalent, the entry has been found and is returned (line 6). If the given key is smaller, then the search continues to the left subtree (line 8). Otherwise, the search moves on to the right subtree (line 10).

The *get* method invokes the *treeSearch* method to find the entry with the given key. If the entry exists, then the value associated with the key is returned. Otherwise, *null* is returned. The Java code is shown below:

Code Segment 5.4

```
1  public V get(K key) throws IllegalArgumentException {
2     checkKey(key);                          // may throw IllegalArgumentException
3     Position<Entry<K,V>> p = treeSearch(root(), key);
4     rebalanceAccess(p);                     // hook for balanced tree subclasses
5     if (isExternal(p)) return null;         // unsuccessful search
6     return p.getElement().getValue();       // match found
7  }
```

Line 2 checks whether the given key is valid. Then the *treeSearch* method is invoked in line 3. The *rebalanceAccess* is a *hook* that will be customized later. When the search is unsuccessful, *null* is returned in line 5. If the search is successful, the value associated with the key is returned in line 6.

The *put* method also invokes the *treeSearch* method first. If the entry with the given key is found, the associated value is set to the new value. If there is no entry with the key, a new entry is inserted. The following is the Java code:

Code Segment 5.5

```
1   public V put(K key, V value) throws IllegalArgumentException {
2     checkKey(key);                          // may throw IllegalArgumentException
3     Entry<K,V> newEntry = new MapEntry<>(key, value);
4     Position<Entry<K,V>> p = treeSearch(root(), key);
5     if (isExternal(p)) {                    // key is new
6       expandExternal(p, newEntry);
```

```
7       rebalanceInsert(p);                  // hook for balanced tree subclasses
8       return null;
9     } else {                               // replacing existing key
10      V old = p.getElement().getValue();
11      set(p, newEntry);
12      rebalanceAccess(p);                  // hook for balanced tree subclasses
13      return old;
14    }
15  }
```

Line 3 creates a new entry with the given key and value and line 4 calls the *treeSearch* method. If the search ends at a leaf node (line 5), then the new entry is added at that leaf node by calling the *expandExternal* method in line 6 (refer to the complete code for details of the *expandExternal* method). If there is already an entry with the given key (line 9), then the associated value is updated by the *set* method in line 11.

The *remove* method implements the deletion operation described in Section 5.1.1.3. The Java code is shown below:

Code Segment 5.6

```
1   public V remove(K key) throws IllegalArgumentException {
2     checkKey(key);                         // may throw IllegalArgumentException
3     Position<Entry<K,V>> p = treeSearch(root(), key);
4     if (isExternal(p)) {                   // key not found
5       rebalanceAccess(p);                  // hook for balanced tree subclasses
6       return null;
7     } else {
8       V old = p.getElement().getValue();
9       if (isInternal(left(p)) && isInternal(right(p))) {
10        // both children are internal
11        Position<Entry<K,V>> replacement = treeMax(left(p));
12        set(p, replacement.getElement());
13        p = replacement;
14      } // now p has at most one child that is an internal node
15      Position<Entry<K,V>> leaf = (isExternal(left(p)) ? left(p) : right(p));
16      Position<Entry<K,V>> sib = sibling(leaf);
17      remove(leaf);
18      remove(p);                           // sib is promoted in p's place
19      rebalanceDelete(sib);                // hook for balanced tree subclasses
20      return old;
21    }
22  }
```

Line 3 invokes the *treeSearch* method, which returns an entry *p*. If p is a leaf node (line 4), then it returns *null* in line 6. Otherwise, there are two cases: (1) p has two internal-node children and (2) p has at most one internal-node child. In the first case (line 9), its predecessor—the rightmost node of its left subtree—is first found in line 11. Then the predecessor replaces p in line 12. In the second case (beginning at line 15), the non-leaf (if there is one) or a leaf node (if both children are leaf nodes) is promoted to p's place. Note that the *remove* method in line 18 is the one defined in the *LinkedBinaryTree* class.

A complete code of *TreeMap* class can be found at the *TreeMap.java* file.

As discussed in previous sections, the search, insertion, and the deletion operations each take $O(h)$ time. The *get*, *put*, and *remove* methods, which are implemented based on those three operations, also run in $O(h)$. The running times of other methods in the *TreeMap* class are shown in the following table:

Figure 5.7 Running Time of TreeMap Operations

Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(h)$
firstEntry, lastEntry	$O(h)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(h)$
subMap	$O(s+h)$
entrySet, keySet, values	$O(n)$

As shown in the table, most methods—including the fundamental tree methods get, put, and remove—take $O(h)$ or $O(\log n)$ time, n being the number of entries in the tree.

However, in the worst case, the tree becomes a long-linked list, as shown below:

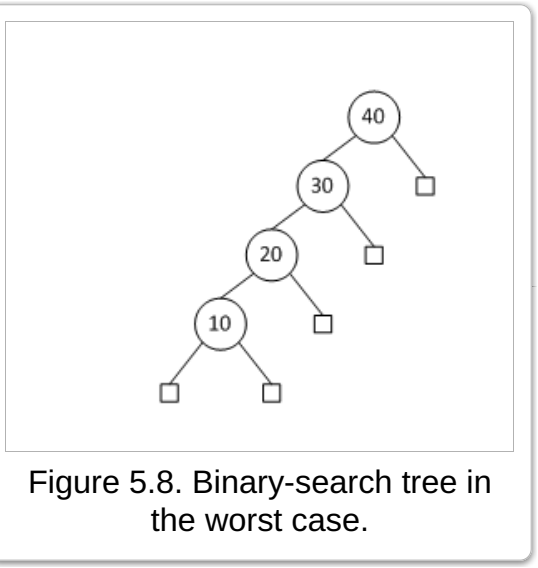


Figure 5.8. Binary-search tree in the worst case.

In this case, the height of the tree becomes n. Therefore, all instances of $O(h)$ in the above table become $O(n)$.

Section 5.1.2. Balanced Search Trees

As discussed in the previous section, a binary search tree can become a long, single link. Though rare, this could still happen in some applications. Balanced trees can solve this problem. In a balanced tree, the depths of all leaf nodes are more or less the same. So, the height of the tree is close to $\log n$. This makes it possible to achieve $O(\log n)$ performance for many tree operations.

There are different types of balanced trees. In this section, we describe basic *rebalance* operations, which are applied to a tree when it becomes unbalanced. In the next section, we discuss a balanced tree called an *AVL* tree.

The primary operation for rebalancing a binary search tree is a *rotation* operation illustrated below:

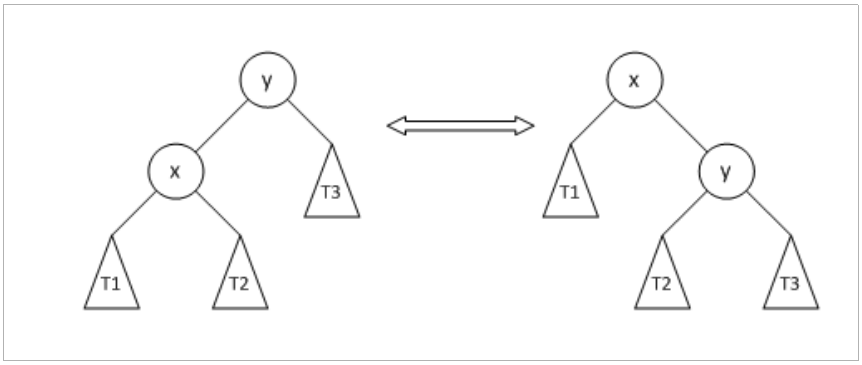


Figure 5.9. Illustration of a rotation operation.

In the figure, circles represent nodes and triangles represent subtrees. The rotation can be performed from the tree on the left to the tree on the right, or vice versa. In either case, the rotation operation preserves the binary-search tree properties. For example, all keys in T_2 are larger than the key of node x and smaller than the key of node y. This property holds in both trees.

The goal of rebalance operations is to reduce the overall height of a tree. In the rightward transformation of the above figure, the depths of nodes in T_1 are reduced by one. However, the depths of nodes in T_3 are increased by one. If T_1 is a "tall" tree and T_3 is a "short" tree, then this can reduce the overall height of the tree.

Some rebalance operations combine two or more rotations to perform broader rebalancing of a tree. The *trinode restructuring* is an example of such an operation. It involves three nodes and four subtrees. These three nodes are labeled x, y, and z. The node y is the parent of x, and the node z is the grandparent of x.

There are four possible orientations. In the first two orientations, x, y, and z form a linear link in the tree. For example, x is the right child of y and y is the right child of z. These two orientations involve only one rotation each. In the other two orientations, the three nodes form an angle in the tree. For example, x is the left child of y, and y is the right child of z. These two orientations involve two rotations each.

We assign the second label to each node. The nodes are labeled a, b, and c in such a way that a precedes b and b precedes c in the inorder traversal of the tree. This relabeling makes a uniform description of the restructuring operation that can be applied to all four orientations. The restructuring is performed in the following way:

- The node z is replaced with the node labeled b.
- The nodes labeled a and c become the children of the node labeled b.
- The four subtrees in the previous tree become the children of a and c.

The following is the pseudocode of the trinode restructuring operation:

Code Segment 5.7

```
Algorithm restructure(x)
Input: A node x in a binary search tree T that has both a parent y and a
       grandparent z
Output: Tree T after a trinode restructuring involving nodes x, y, and z
1: Let (a, b, c) be a left-to-right (inorder) listing of the nodes x, y, and z,
```

and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

2: Replace the subtree rooted at z with a new subtree rooted at b.

3: Make a the left child of b and make T_1 and T_2 the left and right subtrees of a, respectively.

4: Make c the right child of b and make T_3 and T_4 the left and right subtrees of c, respectively.

We illustrate these four orientations below. In the figure, the node labels *x*, *y*, and *z* are shown inside the nodes.

Single Rotation-1

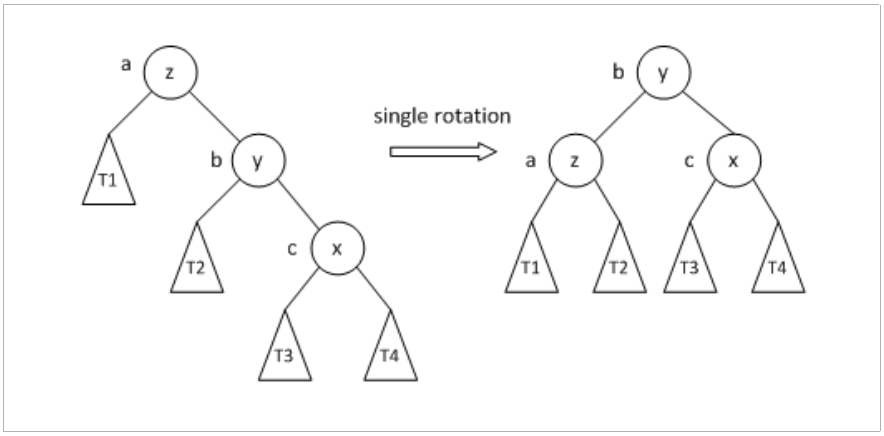


Figure 5.10(a). Single rotation-1.

Single Rotation-2

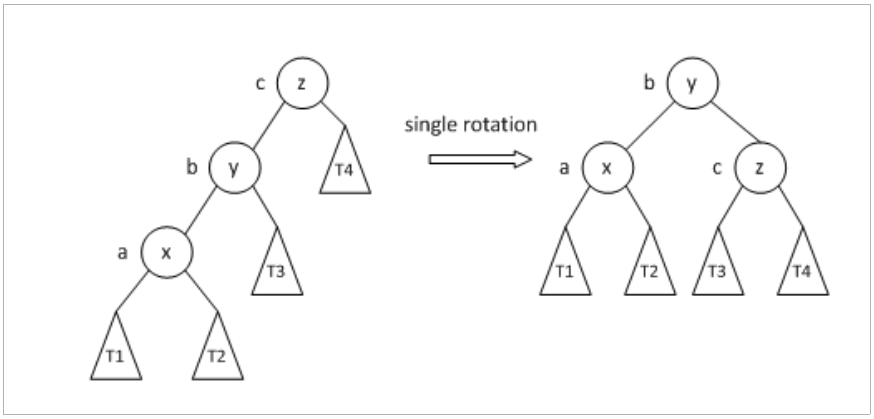


Figure 5.10(b). Single rotation-2.

Double Rotation-1

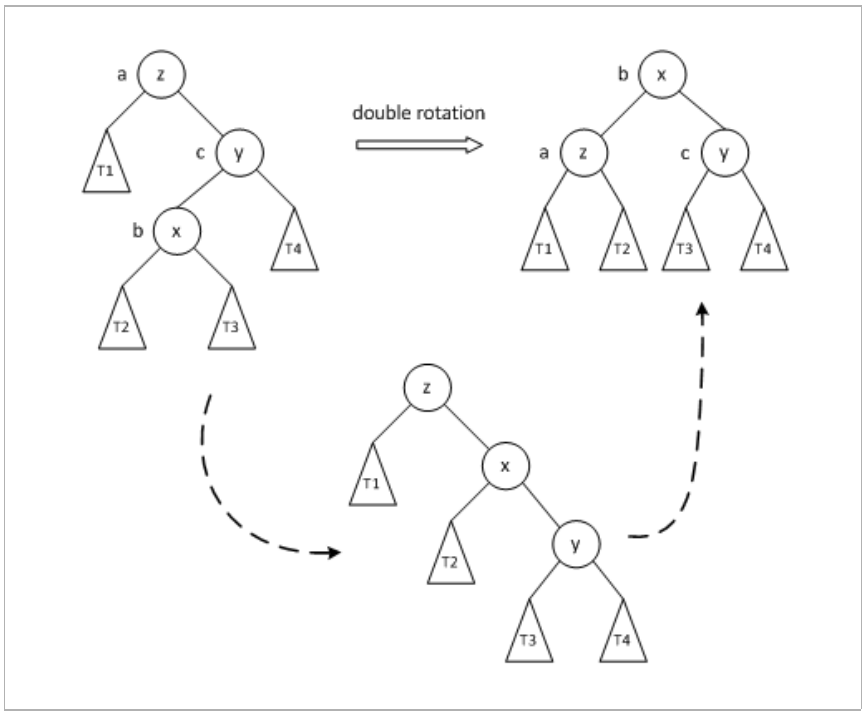


Figure 5.11(a). Double rotation-1.

Double Rotation-2

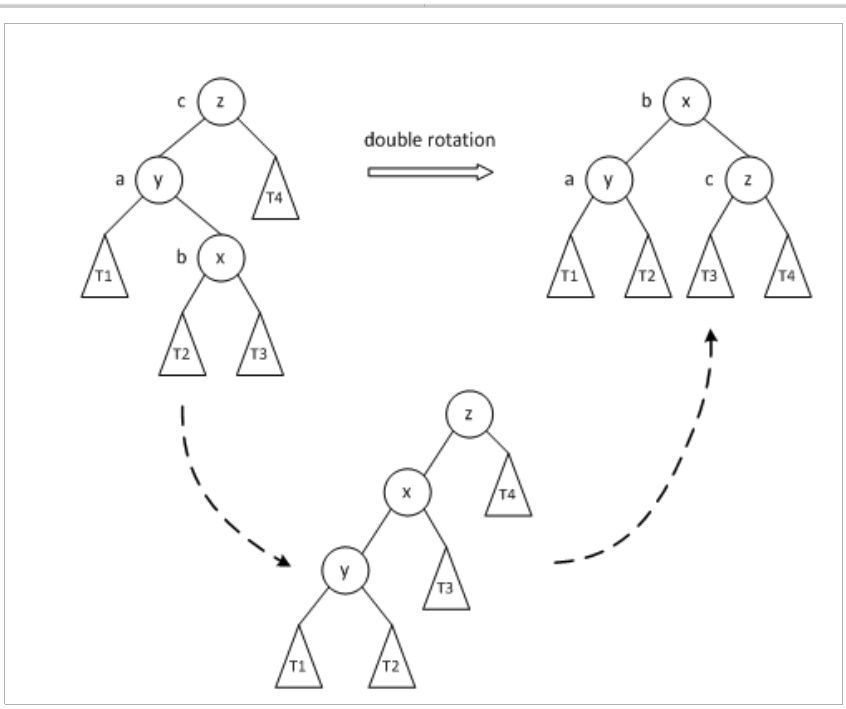


Figure 5.11(b). Double rotation-2.

A Java implementation of the trinode restructuring is described in our textbook. Interested students are referred to pages 475 through 478.

Section 5.1.3. AVL Trees

In this section, we discuss a balanced search tree called an AVL tree, which guarantees worst-case logarithmic time for basic operations. An AVL tree is one of the earliest balanced search trees, which was introduced in 1962. The tree was named after its inventors Adel'son-Vel'skii and Landis.

Section 5.1.3.1. Basics

Before we present the definition of an AVL tree, we restate the definitions of the height of a node and the height of a tree (or a subtree). The *height of a node* is the number of edges on the longest path from that node to a leaf node. The *height of a tree* (or a subtree) is the height of the root of the tree (or a subtree). The height of a leaf node is zero.

An AVL tree is a binary search tree that satisfies the following *height-balance property*:

> For every internal node p of T, the heights of the children of p differ by at most one.

The following figure illustrates an AVL tree. In the figure, the height of an internal node is shown next to the node, but the heights of leaf nodes—which are zero—are not shown, except one leaf node in the tree on the right. Shown inside each node is a key of the entry stored in that node. The tree on the left is an AVL tree. The tree on the right is not an AVL tree because the node with the key 42 does not satisfies the height-balance property (the height of its left child is two and that of its right child is zero).

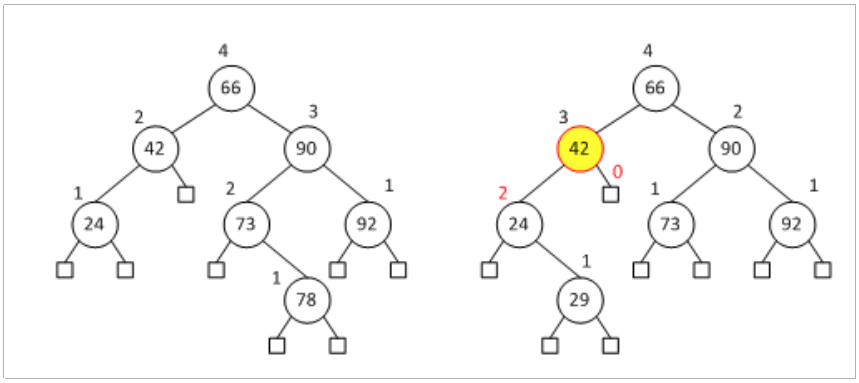


Figure 5.12. Tree on the left is an AVL tree. Tree on the right is not an AVL tree.

We saw in Section 5.1.1 that the performance of many fundamental operations on binary search trees depends on the height of a given tree. So, we discuss an upper bound on the height of AVL trees.

We first consider the minimum number of internal nodes in an AVL tree with height h, denoted as $G(h)$. When $h = 1, G(h) = 1$. When $h = 2, G(h) = 2$. Figure 5.13 shows four AVL trees, each of which has a minimum number of internal nodes, with $h = 1$ through 4. When $h \geq 3$, the number of internal nodes of an AVL tree becomes minimum when it has two children and both of them are AVL trees with the minimum number of internal nodes. The height of one child is $h - 1$ and the height of the other child is $h - 2$. For example, the tree in Figure 5.13 (c) is an AVL tree with $h = 3$, and it has the minimum number of internal nodes. Its left child is an AVL tree with $h = 2$, and its right child is an AVL tree with $h = 1$. In Figure 5.13 (d), the left child is an AVL tree with $h = 3$, and the right child is an AVL with $h = 2$.

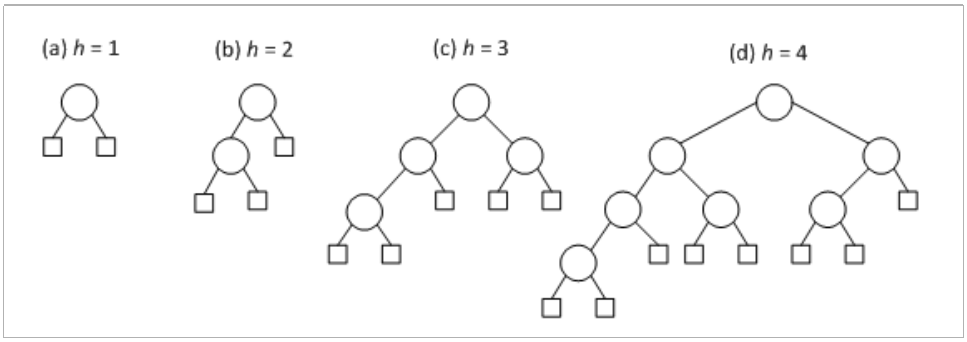


Figure 5.13. Minimum number of nodes in an AVL tree.

So, the minimum number of internal nodes of an AVL tree with the height h is the sum of (the minimum number of internal nodes of an AVL tree with the height $h-1$), (minimum number of internal nodes of an AVL tree with the height $h-2$), and 1. This gives us the following:

$$G(h) = 1 + G(h-1) + G(h-2)\,. \qquad (5.1)$$

From Equation (5.1), we can see that $G(h)$ is a strictly increasing function. So, $G(h-1) \geq G(h-2)$. Replacing $G(h-1)$ with $G(h-2)$ in Equation (5.1), we have the following:

$$\begin{aligned} G(h) &= 1 + G(h-1) + G(h-2) \\ &> 1 + G(h-2) + G(h-2) \\ &= 1 + 2.G(h-2) \end{aligned} \qquad (5.2)$$

We can observer from Equation (5.2) that $G(h)$ at least doubles each time h increases by 2. If we expand the right side of Equation (5.2) in terms of a smaller h, we have the following:

$$\begin{aligned} G(h) &> 2.G(h-2) \\ &> 4.G(h-4) \\ &> 8.G(h-6) \\ &\ldots \\ &> 2^i\,.\,G(h-2i) \end{aligned} \qquad (5.3)$$

Therefore, $G(h) > 2^i\,.\,G(h-2i)$ for any integer i, such that $h-2i \geq 1$. We can pick i so that $h-2i=2$ (because we know $G(2)$). In other words, we pick the following:

$$i = \left\lceil \frac{h}{2} \right\rceil - 1$$

Note that the ceiling function is used because i is an integer. By substituting this value of i in Equation (5.3), we obtain the following:

$$\begin{aligned} G(h) &> 2^{\left\lceil \frac{h}{2} \right\rceil - 1}\,.\,G\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\left\lceil \frac{h}{2} \right\rceil - 1}\,.\,G(1) \\ &\geq 2^{\left\lceil \frac{h}{2} \right\rceil - 1} \end{aligned} \qquad (5.4)$$

If we take the logarithms of both sides, we have the following:

$$\log(G(h)) > \frac{h}{2} - 1$$

Therefore, we obtain the following:

$$h < 2.\log\left(G(h)\right) + 2 \qquad (5.5)$$

So, an AVL tree with n entries has a height of at most $2.\log\left(G(h)\right) + 2$. Therefore, we can conclude that the height of an AVL tree with n entries is $h = O(\log n)$.

Section 5.1.3.2. Update Operations

A node p in a binary search tree is said to be *balanced* if the heights of p's children differ by at most one. Otherwise, a node is said to be *unbalanced.* Therefore, every node in an AVL tree is balanced.

Insertion and deletion operations on an AVL tree are similar to those on a standard binary search tree. The earlier parts of the operations are basically the same. The difference is that, on an AVL tree, those operations need *post-processing* to rebalance the tree, which may have become unbalanced due to the insertion or deletion.

Insertion

When a node is inserted into a binary search tree, a leaf node p is expanded to become a new internal node with two leaf nodes. This expansion may introduce a violation of the height-balance property. However, the only changes that may occur are to the heights of the p's ancestors . Therefore, we need to consider only p's ancestors when rebalancing a tree. Now we illustrate how a rebalancing is performed.

The following figure shows the initial tree, which is balanced:

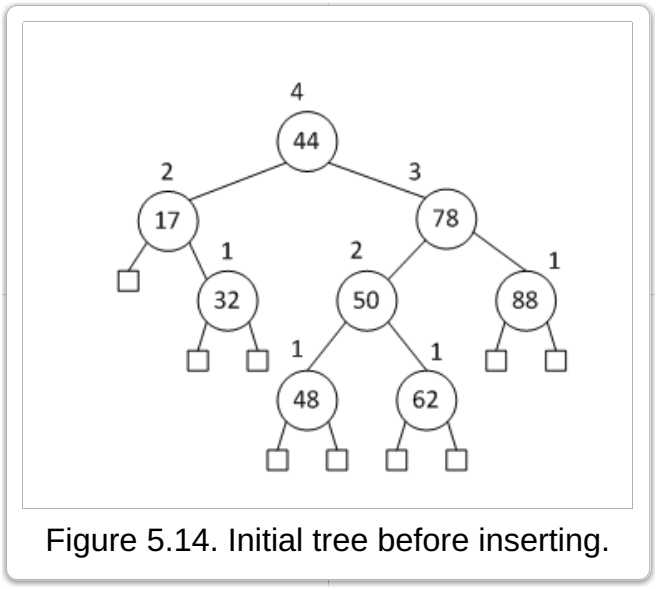


Figure 5.14. Initial tree before inserting.

The following figure shows how an unbalanced tree, which resulted from the insertion of a node, is rebalanced using a double rotation operation. In Figure 5.15(a), a new node with the key 54 is inserted and, as a result, the height-balance property is violated. Specifically, the node with the key 78 is not balanced because the height of its left child is three and the height of its right child is one. This violation can be corrected by performing the double rotation-2 operation. Afterward, the new tree, Figure 5.15(b), is balanced again.

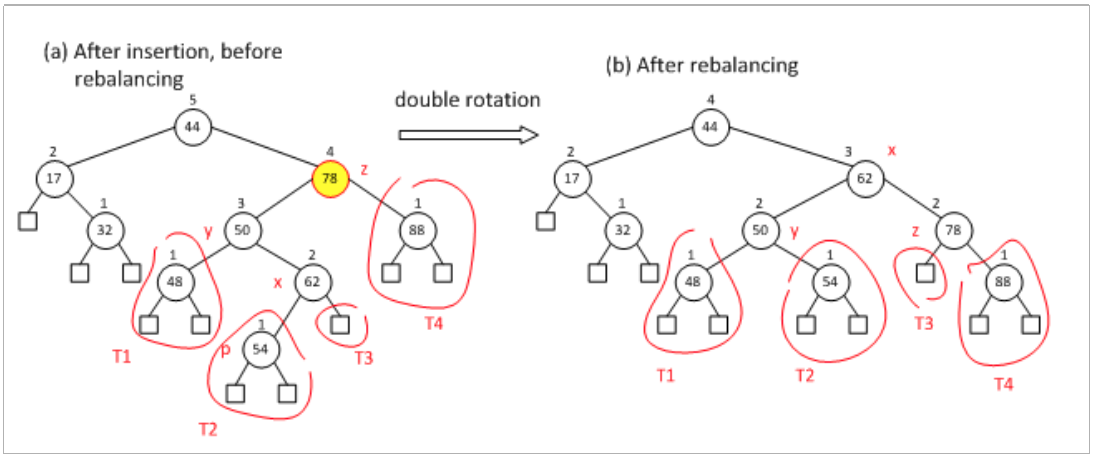


Figure 5.15 (a). After inserting a node, the key 54, tree is unbalanced. (b). A double rotation makes the tree balanced again.

The rebalancing operation is performed with a "search-and-repair" approach. Let p be the new node, which was inserted. In the above figure, the node with the key 54 in the left tree before the double rotation is performed is p. We move up the tree from p and find the first node that is unbalanced (this is the "search"). In this example, the node with the key 78 is the one, the z node. Of z's two children, the one with the greater height is y; this is the node with the key 50. And the child of y with the grater height is x; this is the node with the key 62. Then we rebalance the tree by calling *restructure*(x), which performs a trinode restructuring (this is "repair"). After the restructuring, the node is balanced, as shown in the tree on the right.

We illustrate the rebalancing operation in a more general context:

The tree in Figure 5.16(a) is balanced, and a node is being inserted into the subtree T_3. Figure 5.16(b) shows the tree after the new node has been inserted. Since the new node was inserted into the subtree T_3, its height has been incremented, as have been the heights of all of its ancestors—x, y, and z. Note that the only nodthees affected by the insertion are the ancestors of the subtree T_3. As a result of the insertion, the node z is unbalanced because the height of its left child is h, and the height of its right child is $h + 2$. Note that z is the first unbalanced node when we move upward from T_3. To rebalance the tree, a double rotation is performed by invoking the *restructure* method on the node x. The result is shown in Figure.5.16(c), which is now a balanced tree. After rebalancing, the overall height of the tree remains unchanged.

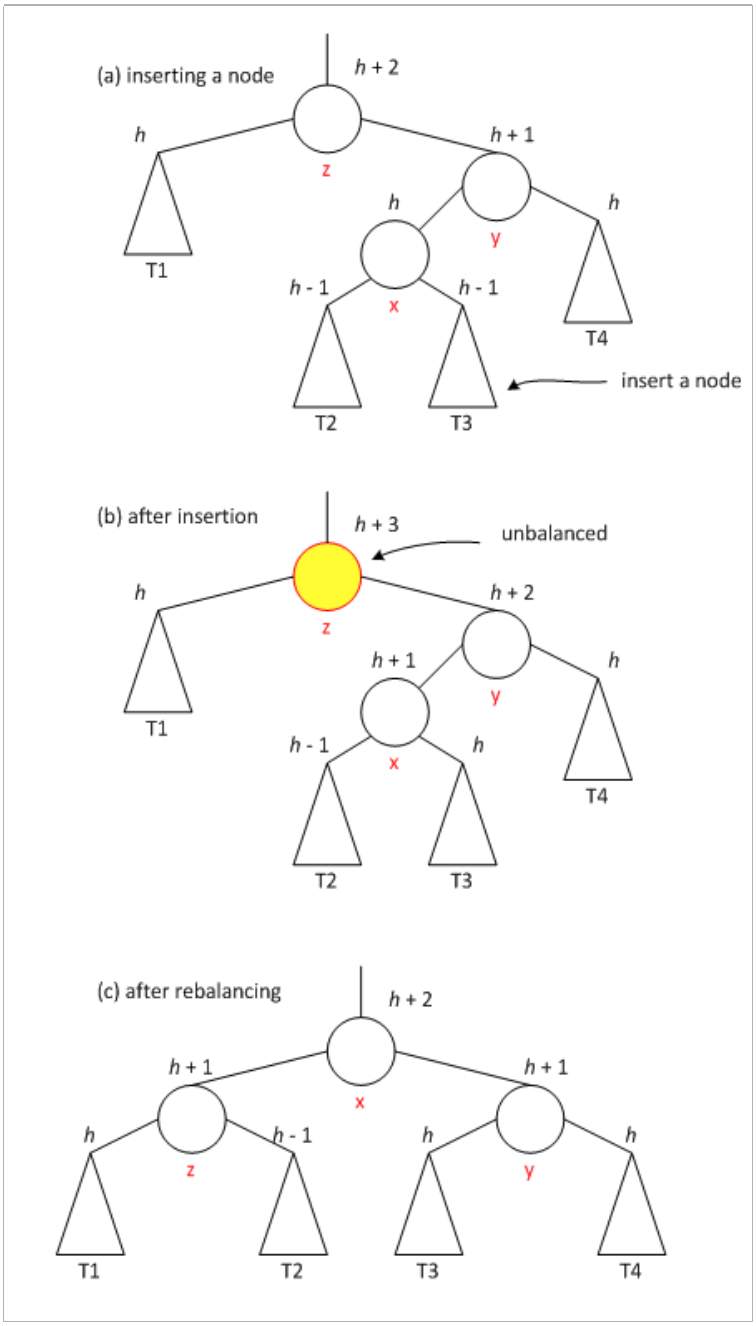


Figure 5.16. Illustration of rebalancing after inserting a node.

Deletion

If we delete a node from a binary search tree, a node with zero or one internal child node is removed. This removal may make the tree unbalanced violating the height-balance property. Suppose a node p is a child of the node that was deleted. One of the nodes on the path from the root to p may become unbalanced (it can be shown that there is at most one unbalanced node). We use the same search-and-repair approach to rebalance the tree.

Beginning at the node p, we move upward to the root to find the first unbalanced node, z along the path. Let y be the child of z with the greater height (y is not an ancestor of p). The node x is determined as follows: If the heights of y's children are different, then the child with the greater height is x; otherwise (i.e., if both children have the same height), the child that is on the same side as

y becomes x. In other words, if y is the right child of its parent, y's right child becomes x, and if y is the left child of its parent, y's left child becomes x. Then, we perform rebalancing by calling the *restructure*(x) method.

The rebalancing after the deletion of a node is illustrated below. In figure 5.17(a), a node that was the right child of the node with the key 17 has been deleted from the position p. As a result, the node with the key 4 is unbalanced. It is the first unbalanced node we find when we move upward from p, and it becomes z. The node with the key 62 has a greater height than its sibling, so it becomes y. Both children of y have the same height of two. Since y is the right child of z, the node with the key 78, which is the right child of y, becomes x.

After we apply double rotation (by calling *restructure*(x)), the new tree in (b) is balanced.

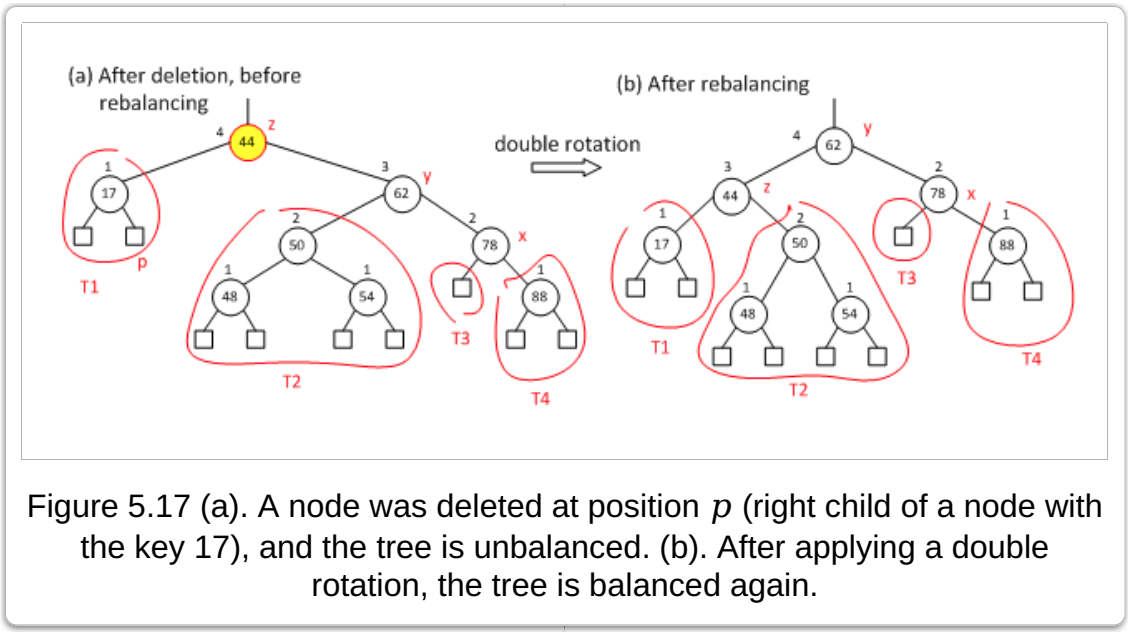


Figure 5.17 (a). A node was deleted at position p (right child of a node with the key 17), and the tree is unbalanced. (b). After applying a double rotation, the tree is balanced again.

Note that the tree shown in the above example is not necessarily a whole tree. In general, it represents a subtree within another tree. After the restructuring, the node y becomes the new root of the subtree and the tree is now locally rebalanced within the subtree. In the above example, the height of the restructured subtree does not change (it was four before restructuring and is still four after restructuring).

However, in some cases, the height of the subtree is decremented. If this occurs, the ancestors of the subtree may be unbalanced; the restructuring may have to be performed on one of the ancestors of the subtree, and this may propagate further upward from the root. Since the height of an AVL tree is $O(\log n)$, the running time of this restructuring is still bounded by $O(\log n)$.

Performance of AVL Trees

An insertion operation on an AVL tree consists of an insertion (on a standard binary search tree) plus restructuring. Since the restructuring takes $O(1)$, an insertion operation on an AVL tree takes $O(\log n)$.

A deletion operation on an AVL tree also consists of a deletion plus restructuring. The restructuring takes, in the worst case, $O(\log n)$. So, the total running time of a deletion operation on an AVL tree is $O(\log n)$.

The performance of AVL trees, including the running times of other operations, is summarized below:

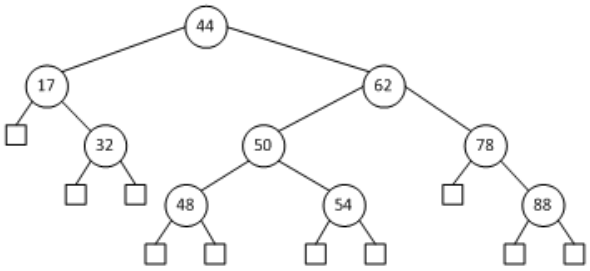
Figure 5.18. Performance of AVL Trees.

Method	Running Time

size, isEmpty	$O(1)$
get, put, remove	$O(\log n)$
firstEntry, lastEntry	$O(\log n)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$
entrySet, keySet, values	$O(n)$

Test Yourself 5.1

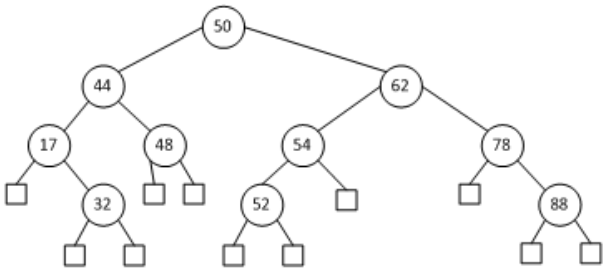
Consider the following AVL tree:



Show the resulting AVL tree after inserting a node with key 52 into the tree.

Please think carefully, write your answer, and then click "Show Answer" to compare yours to the possible algorithm.

Suggested answer: After inserting 52, the tree is not height-balanced. So, restructuring is performed.



Section 5.2 Sorting

Section 5.2. Sorting

Overview

Sorting is one of the most fundamental algorithms in computer science. A sorting algorithm can be used as a stand-alone program to sort a collection of elements. More important, sorting algorithms are used as part of many other algorithms. In this section, we discuss some popular sorting algorithms.

Section 5.2.1. Merge-Sort

In this section, we discuss the *merge-sort* algorithm.

Section 5.2.1.1 Divide-and-Conquer

The *divide-and-conquer* design pattern involves the following three steps:

1. *Divide*—If the problem is smaller than a predefined size threshold, solve the problem by brute force. Otherwise, divide the problem into smaller subproblems.
2. *Conquer*—Recursively solve the smaller problems.
3. *Combine*—Combine the solutions to the subproblems to obtain a solution to the original problem.

The merge-sort algorithm solves the sorting problem using the divide-and-conquer design pattern. We first present a high-level description of the merge-sort algorithm. Let S be a set of elements to be sorted, and n be the number of elements in S.

1. *Divide*—If S has zero or one element, return S (because it is already sorted). Otherwise, divide S into two separate arrays, S_1 and S_2, of approximately equal size. S_1 contains the first $\lfloor n/2 \rfloor$ elements of S, and S_2 contains the remaining $\lceil n/2 \rceil$ elements.
2. *Conquer*—Sort S_1 and S_2 recursively.
3. *Combine*—Put the elements back into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

The following figure illustrates the divide-and-conquer process of the merge-sort. In the figure, thick, red lines indicate where a sequence is divided. The left side of figure depicts the dividing steps, and the right side illustrates the combining steps, whereby two smaller sequences are merged into a larger sequence. Note that when two smaller sequences are merged, elements are sorted in the merged sequence.

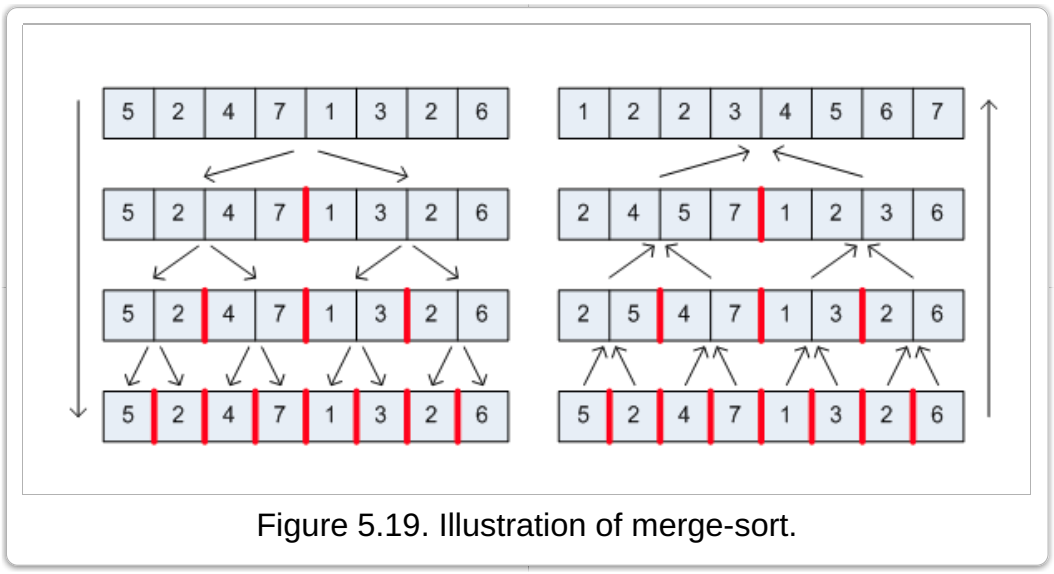


Figure 5.19. Illustration of merge-sort.

Section 5.2.1.2. Array-Based Implementation

In this section, we describe an array-based merge-sort algorithm. As illustrated in the previous section, actual sorting occurs when two smaller sequences are merged. The following is a Java code that implements the merge step:

Code Segment 5.8

```
1  public static <K> void merge(K[] S1, K[] S2, K[] S, Comparator<K> comp) {
2    int i = 0, j = 0;
3    while (i + j < S.length) {
4      if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
5        S[i+j] = S1[i++];        // copy ith element of S1 and increment i
6      else
7        S[i+j] = S2[j++];        // copy jth element of S2 and increment j
8    }
9  }
```

The first two arguments, S_1 and S_2, are subarrays to be merged; the result is put in the third argument, S. The last argument is a comparator, used to compare elements. (Note that this is a generic method with the generic type parameter K. So, the comparator that is defined for an actual type is passed as an argument.) The *if* statement of line 4 compares a pair of elements, one from S_1 and the other from S_2. If the element from S_1 is smaller than that from S_2, it is copied to S. Otherwise, the element from S_2 is copied to S. Embedded in the code is a mechanism to handle a situation in which all elements from one subarray are copied to S, but the other subarray still has some elements left. If all elements from S_2 are copied to S and there are still elements in S_1, the remaining elements from S_1 are copied to S in line 5. If all elements from S_1 are copied to S and there are still elements in S_2, the remaining elements from S_2 are copied to S in line 7.

The following figure illustrates the merge-sort process:

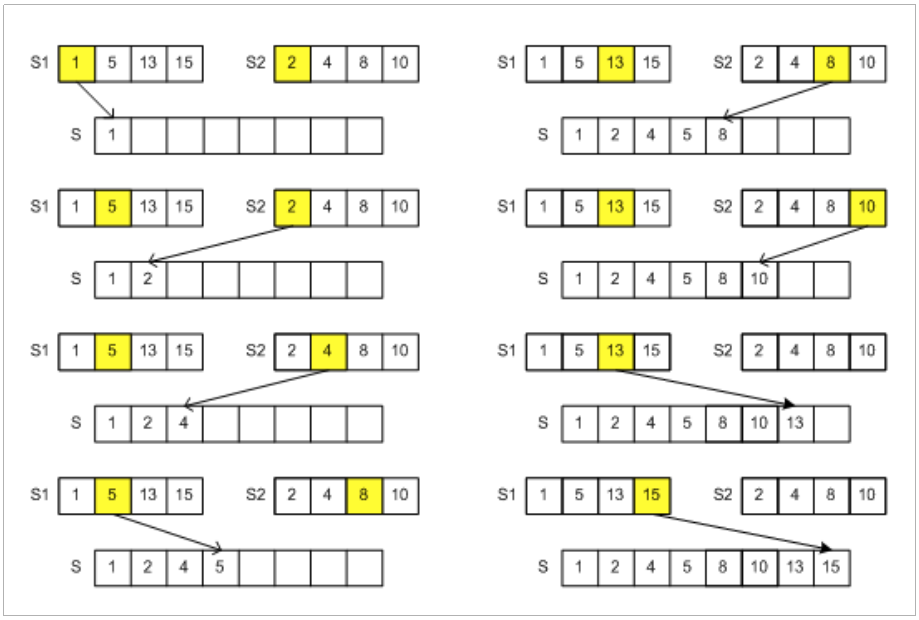


Figure 5.20. Illustration of merge-sort implemented using an array.

A Java code of the merge-sort is given below:

Code Segment 5.9

```
1  public static <K> void mergeSort(K[] S, Comparator<K> comp) {
2    int n = S.length;
3    if (n < 2) return;                      // array is trivially sorted
4    // divide
5    int mid = n/2;
6    K[] S1 = Arrays.copyOfRange(S, 0, mid);  // copy of first half
7    K[] S2 = Arrays.copyOfRange(S, mid, n);  // copy of second half
8    // conquer (with recursion)
9    mergeSort(S1, comp);                     // sort copy of first half
10   mergeSort(S2, comp);                    // sort copy of second half
11   // merge results
12   merge(S1, S2, S, comp);              // merge sorted halves back into original
13 }
```

If there is only one element, then we return in line 3. The given array is divided into S_1 and S_2 in lines 6 and 7. Lines 9 and 10 sort the two subarrays recursively. Two sorted subarrays are merged in line 12.

5.2.1.3 Running-Time Analysis

We assume that, for simplicity, the number of elements n is a power of two. An analysis based on this assumption still holds when n is not a power of two. First, we determine the running time of the *merge* method. Let n_1 and n_2 be the number of elements in S_1 and S_1, respectively. In each iteration of the *while* loop (in lines 4 through 9), an element is copied from either S_1 or S_2 to S. So the number of iterations is $n_1 + n_2$ and the running time of the *merge* method is $O(n_1 + n_2)$. Since $n_1 + n_2$ is bounded by the total number of elements n, we can conclude that the *merge* takes $O(n)$ time.

In the *mergeSort* method, lines 7 and 8 copy elements from S to S_1 and S_2. This takes $O(n)$ time because n total elements are copied. Lines 10 and 11 are recursive calls, and line 12 is an invocation of the *merge* method. So, excluding the recursive calls, the *mergeSort* takes $O(n) + O(n) = O(n)$. Each recursive call is made on a subarray with $n/2$ elements, and the running time of the *mergeSort* on such a subarray is $O(n/2)$. As the successive recursive calls are made, the size of subarray becomes $n/2, n/4, n/8, \ldots,$ and so on, eventually becoming one. The trace of recursive calls, along with the size of each subarray, is repeated in Figure 5.21. Since the trace of recursive calls naturally forms a tree, as shown in the figure, this tree is sometimes referred to as a *recursion tree*.

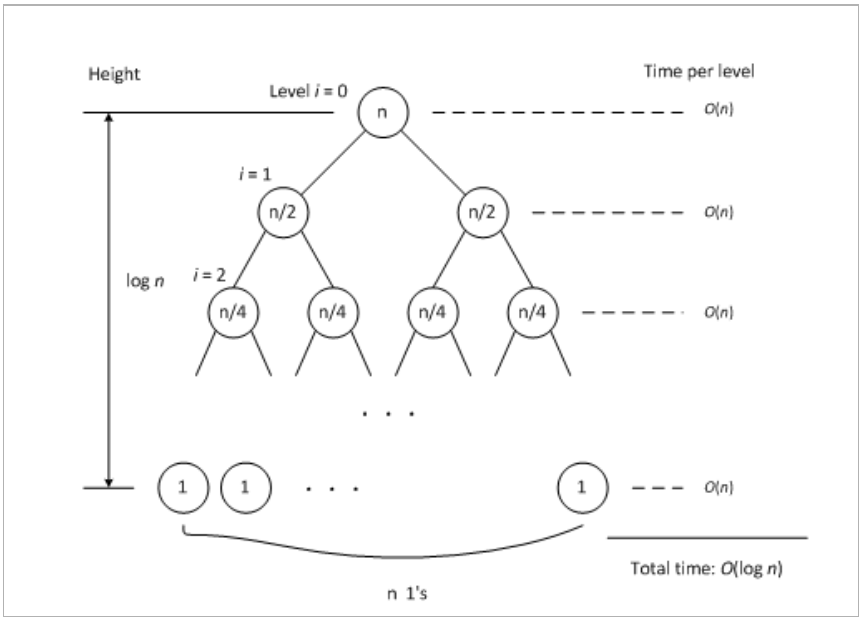


Figure 5.21. A recursion tree representing the running time of merge-sort.

The height of the tree is log n, so, the number of levels is $(\log n + 1)$. Let i be the number of each level in the tree. The level of the root is zero. In general, at level i, there are 2^i nodes, and each node takes $O(n/2^i)$ time. Therefore, the running time at level $O(2^i . n/2^i) = O(n)$. Then, the total running time is:

$$O(n)(\log n + 1) = O(n \log n) + O(n) = O(n \log n)$$

Section 5.2.1.4. Alternative Implementation of Merge-Sort

In the previous section, we discussed an array-based, recursive implementation of the merge-sort. In this section, we describe a linked list-based implementation and a nonrecursive implementation.

Sorting a Linked List

In this implementation, we use a linked list as a storage, or we sort elements stored in a linked list. This is also a recursive implementation, and elements are sorted in the same way as that they are in the array-based version. This implementation uses the *LinkedQueue*, which is a linked list-based queue.

A Java code of both the *merge* methods follows:

Code Segment 5.10

```
1   public static <K> void merge(Queue<K> S1, Queue<K> S2, Queue<K> S,
2                                                    Comparator<K> comp) {
3     while (!S1.isEmpty() && !S2.isEmpty()) {
4       if (comp.compare(S1.first(), S2.first()) < 0)
5        S.enqueue(S1.dequeue());       // take next element from S1
6       else
```

```
7         S.enqueue(S2.dequeue());       // take next element from S2
8     }
9     while (!S1.isEmpty())
10      S.enqueue(S1.dequeue());         // move any elements that remain in S1
11    while (!S2.isEmpty())
12      S.enqueue(S2.dequeue());         // move any elements that remain in S2
13  }
```

In line 4, two front elements from S_1 and S_2 are compared. If the front element from S_1 is smaller than that from S_2, it is removed from S_1 and added to S in line 5. Otherwise, the front element is removed from S_2 and added to S in line 6. After all elements are moved from either S_1 or S_2, any remaining elements in the other list are moved to S in lines 9 through 13.

Code Segment 5.11

```
1   public static <K> void mergeSort(Queue<K> S, Comparator<K> comp) {
2     int n = S.size();
3     if (n < 2) return;                 // queue is trivially sorted
4     // divide
5     Queue<K> S1 = new LinkedQueue<>(); // (or any queue implementation)
6     Queue<K> S2 = new LinkedQueue<>();
7     while (S1.size() < n/2)
8       S1.enqueue(S.dequeue());         // move the first n/2 elements to S1
9     while (!S.isEmpty())
10      S2.enqueue(S.dequeue());         // move remaining elements to S2
11    // conquer (with recursion)
12    mergeSort(S1, comp);               // sort first half
13    mergeSort(S2, comp);               // sort second half
14    // merge results
15    merge(S1, S2, S, comp);            // merge sorted halves back into original
16  }
```

Two empty queues are created in lines 5 and 6. The first $n/2$ elements are moved to S_1 in lines 7 and 8, and the second half is moved to S_2 in lines 9 and 10. Two recursive calls are made in lines 12 and 13, and the results are merged in line 15.

A Bottom-Up (Nonrecursive) Merge-Sort

The merge-sort problem can be easily described as a divide-and-conquer problem, and most of the time, a recursive algorithm is a natural choice to solve a divide-and-conquer problem. However, a recursive implementation involves the overhead of maintaining recursive call stacks. An alternative is a bottom-up, nonrecursive implementation.

A general strategy is as follows. We merge each successive pair of elements into a sorted sequence of the length two. Next, we merge each pair of two-element sequences into a sorted sequence of the length four, merge two sequences of the length four to create a sorted sequence of the length eight, and so on. A complete Java code of this implementation can be found at the *MergeSort.java* file. This file also includes the recursive implementation and the linked list-based implementation.

Section 5.2.2. Quick-Sort

In this section, we describe another divide-and-conquer soring algorithm, which runs in $O(n \log n)$, on average.

Section 5.2.2.1. Outline of Quick-Sort

The following is an outline of the quick-sort algorithm, which sorts a sequence S with n elements. In the dividing step, a given sequence is divided across a particular element called *pivot*. In this section, we use the last element as the pivot. Let x denote the pivot element.

1. *Divide*—If S has only one element, return. Otherwise, remove all elements from S and put them into three sequences:
   - L—This sequence contains the elements that are less than x.
   - E—This sequence contains the elements that are equal to x.
   - G—This sequence contains the elements that are greater than x.

   If the elements in S are distinct, then E has only one element, which is the pivot.
2. *Conquer*—Recursively sort L and G.
3. *Combine*—Put back the elements from the three parts into S in order.

The following figure illustrates the sorting process:

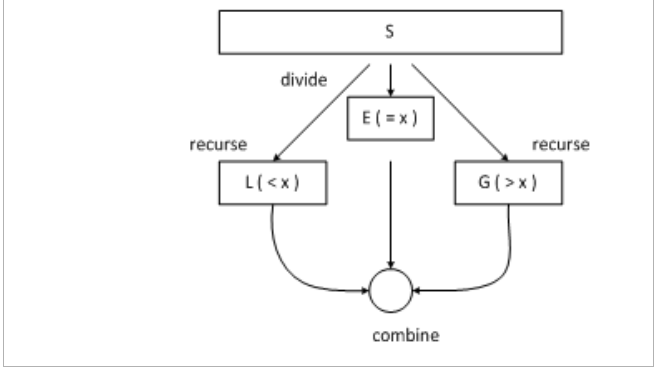


Figure 5.22. A high-level depiction of quick-sort.

Figure 5.23 shows the dividing and combining steps on a small sequence of numbers:

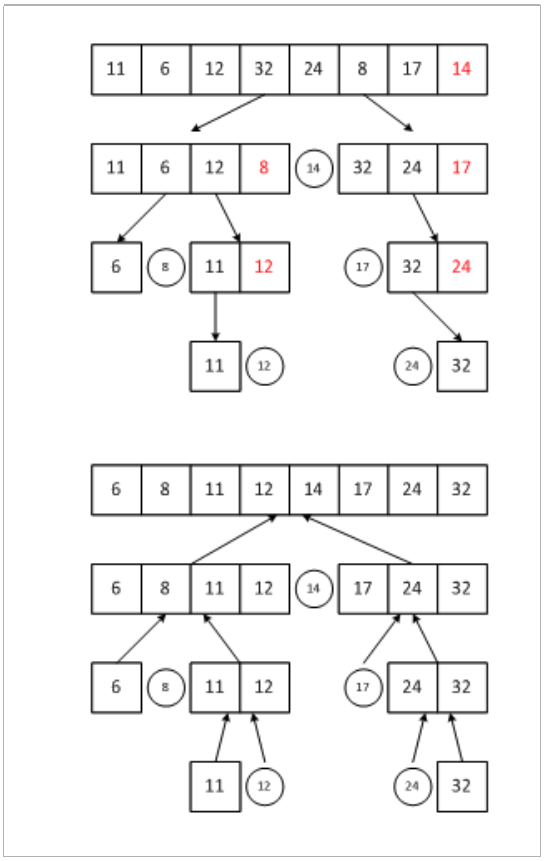


Figure 5.23. Illustration of quick-sort on a small sequence.

Section 5.2.2.2. Implementation of Quick-Sort Using an Array

The quick-sort algorithm described in the previous section requires additional storage for L, E, and G. If we implement the quick-sort using an array, we can avoid the additional storage. A sorting algorithm is called an *in-place* sorting algorithm if it uses only a small amount of memory in addition to that needed for the original input.

In this section, we describe an array-based quick-sort implementation, which is an in-place sorting. In this implementation, partitioning of the given array into subarrays is performed by swapping elements within the array itself. So, no additional storage is used. The following is a high-level description of the partitioning:

- Given array S has n elements, and the last element, $S[n-1]$, is used as the pivot.
- Two indexes, *left* and *right,* are maintained.
- Initially, *left* points to $S[0]$, and *right* points to $S[n-2]$.
- *left* keeps moving to the right until it meets the first element that is equal to or larger than the pivot, *right marker*.
- *right* keeps moving to the left until it meets the first element that is equal to or smaller than the pivot, *left marker*.
- The right and left markers are swapped.
- This is repeated until *left* and *right* cross each other.
- After that, the right marker (or *S*[*left*]) is swapped with the pivot, placing the pivot in the right position.

The partitioning process is illustrated in the following figure:

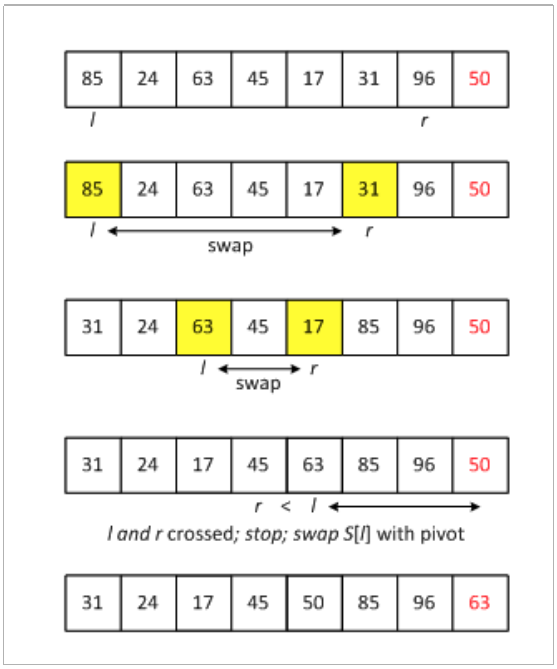


Figure 5.24. Illustration of partitioning.

A Java code of the array-based implementation is given below:

Code Segment 5.12

```
1   private static <K> void quickSortInPlace(K[] S, Comparator<K> comp,
2                                                              int a, int b) {
3     if (a >= b) return;                // subarray is trivially sorted
4     int left = a;
5     int right = b-1;
6     K pivot = S[b];
7     K temp;                            // temp object used for swapping
8     while (left <= right) {
9       // scan until reaching value equal or larger than pivot (or right marker)
10      while (left <= right && comp.compare(S[left], pivot) < 0) left++;
11      // scan until reaching value equal or smaller than pivot (or left marker)
12      while (left <= right && comp.compare(S[right], pivot) > 0) right--;
13      if (left <= right) {             // indices did not strictly cross
14        // so swap values and shrink range
15        temp = S[left]; S[left] = S[right]; S[right] = temp;
16        left++; right--;
17      }
18    }
19    // put pivot into its final place (currently marked by left index)
20    temp = S[left]; S[left] = S[b]; S[b] = temp;
21    // make recursive calls
22    quickSortInPlace(S, comp, a, left - 1);
23    quickSortInPlace(S, comp, left + 1, b);
```

```
24  }
```

The *while* loop of line 10 moves *left,* and the while loop of line 12 moves *right*. If they do not cross, then swap the right marker and the left maker in line 15. If they cross each other, swap the right marker and the pivot in line 20. This completes the partitioning. Then recursive calls are made on the left partition in line 22 and on the right partition in line 23. Since this is an in-place sorting, no combining step is needed.

Running-Time Analysis

We can analyze the running time of quick-sort using the same technique we used for merge-sort. In both sorting algorithms, a given sequence is divided into two subsequences, each of which is sorted recursively. One difference is the "balance" of partitioning. Merge-sort divides an array near the middle, so the sizes of the two partitions are approximately the same, and partitioning is always balanced (i.e., there are approximately the same number of elements in both partitions). In quick-sort, however, the partitioning point is determined by the value of the pivot. In the best case, we may have a perfectly balanced partitioning in which the pivot is placed in the middle of the array. In the worst case, we may have a partitioning in which the pivot is at one end of the array, leaving one part with $n-1$ elements and the other empty. This is extremely unbalanced partitioning.

In the best case, at each level of recursion, we have the perfect partitioning. Then the analysis is basically the same as that for merge-sort, and the best-case running time of quick-sort is $O(n \log n)$.

In the worst case, we always have the extremely unbalanced partitioning. For simplicity, let $quicksort(n)$ denote an invocation of a quick-sort method on an array of size n. The recursive call sequence, in this case, will be $quicksort(n) - quicksort(n-1) - quicksort(n-2)$, and so on. It can be shown that the running time is $O(n^2)$. If an array is already sorted and the last element is chosen as the pivot, then this worst case occurs.

It can be shown that, even when partitioning is not perfect, we can still achieve $O(n \log n)$ running time. For example, even if we have one-to-nine partitioning every time, the running time is still $O(n \log n)$.

Improvement

To minimize the possibility of the worst-case scenario, we can choose the pivot element in a different manner.

One way is to randomly select an element from the array as the pivot. The quick-sort using this method is called *randomized quick-sort*. The running time of randomized quick-sort is also $O(n \log n)$. There is a formal analysis of the running time of randomized quick sort in our textbook. Interested students are referred to pages 551 and 552.

Another method is to use the median of the three values—the first, the middle, and the last elements of the given sequence. This method is called the *median-of-three* method, and it usually gives a good pivot.

There are hybrid approaches to increasing sorting performance. Quick-sort algorithms usually work well on large datasets. However, their performance is poor on small datasets. To resolve this problem, a typical hybrid method modifies quick-sort in such a way that when the size of a partition falls below a certain threshold (such as 50 elements), the partition is sorted using a quadratic-time sorting algorithm, such as insertion-sort.

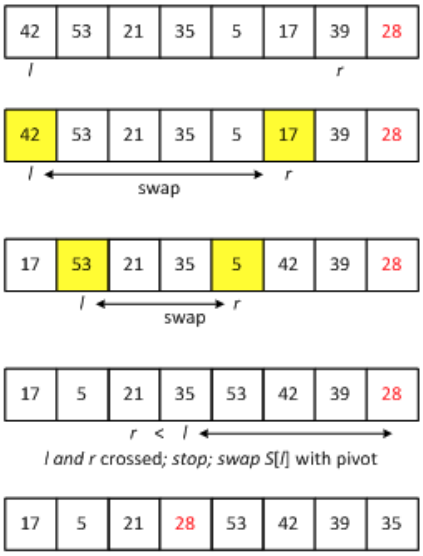
Test Yourself 5.2

Illustrate the execution of the partitioning of the following sequence, using Figure 5.24 as a model.

<42, 53, 21, 35, 5, 17, 39, 28>

Please think carefully, write your answer, and then click "Show Answer" to compare yours to the possible algorithm.

Suggested answer:



Section 5.2.3. Sorting as an Algorithmic Problem

The sorting algorithms we discussed so far have running times of $O(n^2)$ or $O(n \log n)$. One may ask the following question: Is there a sorting algorithm that is faster than $O(n \log n)$? The answer is yes, but with some qualifications. In this section, we describe two sorting algorithms that run in $O(n)$ time.

Before we discuss those two algorithms, we first note that if a sorting algorithm uses comparison operations when ordering the elements, then the running time is bounded from below by $n \log n$. In other words, the comparison-based sorting algorithms have a $\Omega(n \log n)$ worst-case lower bound. A proof of this lower bound is given on pages 556 and 557 of our textbook. Interested students are encouraged to read the book.

Section 5.2.3.1. Bucket-Sort

The bucket-sort algorithm can sort a sequence of elements in linear time with a constraint. The constraint is that the elements are integers in the range $[0, N - 1]$, for some integer $N \geq 2$. If the elements to be sorted are objects, then the objects must have integer keys with total ordering. The integer keys must be in the range $[0, N - 1]$, for some integer $N \geq 2$.

Let S be a sequence of n entries to be sorted. We use a bucket array B of size N. Each element of the bucket array B is a sequence of entries with an integer key. In other words, we can store a sequence of entries in each slot of the B array. We first distribute n entries across the B array, with an entry with the key k being added to $B[k]$. Multiple entries added to the same slot are included in the sequence associated with the slot. Then, we remove the entries from the bucket array—from $B[0]$] to $B[N - 1]$, in that order—and put them back into S.

A pseudocode of bucket-sort is given below:

Code Segment 5.13

```
Algoritm bucketSort(S)
Input: Sequence S of entries with integer keys in range [0, N – 1]
Output: Sequence S sorted in nondecreasing order of keys
create an empty B an array of size N
for each entry e in S do
  let k be the key of e
  remove e from S and add it to the end of bucket B[k], which is a
       sequence
for i = 0 to N – 1  do
  for each entry in sequence B[i] do
    remove e from B[i] and insert it at the end of S
```

The first *for* loop inserts all n entries into the bucket array B. The second *for* loop scans the bucket array B and, from each slot of the bucket, which is a sequence of entries, moves entries back to S. The running time is $O(n + N)$. If $N = O(n)$, then the running time becomes $O(n)$. The bucket-sort works (as it was assumed in the above algorithm) when there are duplicate keys.

Assume a sequence of entries $S = ((k_0, v_0), (k_1, v_1), ..., (k_{n-1}, v_{n-1}))$. Suppose that two entries, (k_i, v_i) and (k_j, v_j), have an identical key—i.e, $k_i = k_j, i \neq j$. We say a sorting algorithm is *stable* if (k_i, v_i) precedes (k_j, v_j) in S both before and after sorting.

Consider a sequence $S = ((12, W), (4, F), (7, H), (4, A), (2, P))$, before sorting. A stable sorting algorithm will generate $S = ((2, P), (4, F), (4, A), (7, H), (12, W))$. Notice that $(4, F)$ precedes $(4, A)$ before and after sorting. If the sorting result is $S = ((2, P), (4, A), (4, F), (7, H), (12, W))$, then it is not a stable sorting.

The bucket-sort algorithm, described above, is a stable sorting as far as all sequences are implemented as queues, in which elements are processed and removed from the front of a sequence and added to the rear of a sequence. In other words, when elements in S are initially moved to the bucket array B (in the first *for* loop), we should process S from front to rear and add each entry to the rear of its bucket. Later, when we move the entries in the bucket array B back to S (in the second *for* loop), we should remove each entry from the front of each bucket sequence and add it to the rear of S.

Section 5.2.3.2. Radix-Sort

In this section, we discuss another linear time sorting algorithm called *radix sort*.

Suppose that a sequence S stores entries with keys that are pairs, (k, l), and we want to sort entries in S by lexicographic order. In other words, $(k_1, l_1) < (k_2, l_2)$ if $k_1 < k_2$ or if $k_1 = k_2$ and $l_1 < l_2$. The radix-sort algorithm can sort S correctly by applying a stable sorting algorithm twice. First, it sorts the entries in S by using the second component. Next, it sorts the entries in S again by using the first component. It is important that we sort by the second component first.

Consider the following sequence of pairs:

$$S = ((4, 8), (6, 2), (4, 5), (5, 3), (3, 6), (3, 9), (5, 7))$$

First, we sort S using a stable sorting algorithm on the first component:

$$S_1 = ((3, 6), (3, 9), (4, 8), (4, 5), (5, 3), (5, 7), (6, 2))$$

Then we sort S_1 using a stable sorting algorithm on the second component:

$$S_2 = ((6,2),(5,3),(4,5),(3,6),(5,7),(4,8),(3,9))$$

This result is obviously wrong.

If we sort by the second component first and then by the first component, the result is correct, as shown below:

$$S_1 = ((6,2),(5,3),(4,5),(3,6),(5,7),(4,8),(4,9))$$

$$S_2 = ((3,6),(4,5),(4,8),(4,9),(5,3),(5,7),(6,2))$$

The radix-sort algorithm can also be used to sort *d*-digit numbers. The following figure illustrates how three-digit numbers are sorted by the radix-sort algorithm:

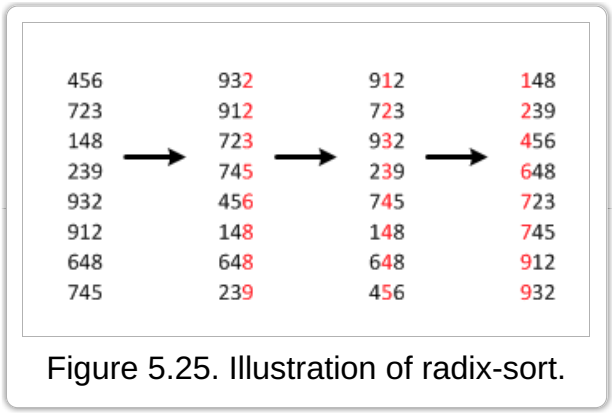


Figure 5.25. Illustration of radix-sort.

Running-Time Analysis

Let S be a sequence of n key-value pairs, in which each key is a -tuple $(k_1, k_2, ..., k_d)$, and each k_i is in the range $[0, N-1]$ for some integer $N \geq 2$. The radix-sort algorithm can sort S by applying a stable sorting algorithm d times. It sorts S by k_d first, then by k_{d-1}, then by k_{d-2}, ..., and so on, until finally by k_1.

If we use the bucket-sort—which is a stable sorting algorithm—in each pass, the running time is $O(n+N)$ (refer to the running-time analysis of bucket-sort earlier in this section). Since there are *d* passes, the total running time of radix-sort is $O(d(n+N))$.

Section 5.2.4. Comparing Sorting Algorithms

We discussed sorting algorithms with different running times, which are summarized in the following table:

Figure 5.26. Performance Comparison of Sorting Algorithms

Running Time (Average)	Sorting Algorithms
$O(n)$	bucket-sort, radix-sort
$O(n \log n)$	heap-sort, quick-sort, merge-sort
$O(n^2)$	insertion-sort, selection-sort

In general, we can select a sorting algorithm based on running times. However, depending on the specific properties of the

application, a seemingly slower algorithm may be more suitable than a faster algorithm. In many cases, selecting a sorting algorithm involves a careful analysis that considers trade-offs among efficiency, storage requirements, and stability. This type of analysis requires a good understanding of different sorting algorithms.

In this section we summarize the properties of different sorting algorithms.

Insertion-Sort

- When the number of inversions is small, the running time can be reduced to $O(n + m)$, where m is the number of inversions. The number of inversion is the number of pairs of elements that are out of order.
- When the number of elements in the input sequence is small (typically less than 50), insertion-sort is very efficient.
- Insertion-sort is very efficient for an "almost"-sorted sequence (because an almost-sorted sequence has a small number of inversions).
- In general, due to its quadratic running time, insertion-sort is not a good choice except for the situations listed above.

Heap-Sort

- Heap-sort runs in $O(n \log n)$ in the worst case.
- It works well on small and medium-sized sequences.
- It can be made an in-place sorting algorithm.
- Its performance on large sequences is poorer than that of quick-sort and merge-sort.
- Heap-sort is not a stable sorting algorithm.

Quick-Sort

- Quick-sort's worst-case running time is $O(n^2)$.
- Experimental studies showed that quick-sort outperformed heap-sort and merge-sort.
- Quick-sort has been a default general-purpose, in-memory sorting algorithm.
- It was used in C libraries.
- Java uses it as the standard sorting algorithm for sorting arrays of primitive types.

Merge-Sort

- Merge-sort's worst-case running time is $O(n \log n)$.
- It is difficult to make merge-sort an in-place sorting algorithm. So, it is less attractive than heap-sort and quick-sort.
- Merge-sort is an excellent algorithm for sorting data that resides on the disk (or storage outside the main memory).
- *Tim-sort* is a hybrid algorithm that uses merge-sort and insertion-sort.
- Tim-sort has been the standard sorting algorithm in Python since 2003.
- Java uses Tim-sort for sorting arrays of objects.

Bucket-Sort and Radix-Sort

- Bucket-sort and radix-sort are excellent choices if an application involves sorting entries with small integer keys, character strings, or d-duple keys from a small range.
- For such applications, these choices are better than quick-sort, heap-sort, and merge-sort.

Section 5.2.5. Selection Problem

In some applications, we want to find the k^{th} smallest element from an unordered set of elements. Suppose we have an unordered collection with n elements. When $k = 1$, we are looking for the minimum, and when $k = n$, we are looking for the maximum. This problem is called the *selection* problem.

Formally, the selection problem is defined as follows:

Given a set S of n comparable elements and an integer k, for which $1 \leq k \leq n$, find the element $e \in S$ that is larger than exactly $k - 1$ elements of S.

The k^{th} smallest element is also referred to as the k^{th} *order statistic*. So, the selection problem is to find the k^{th} order statistic from S. In what follows, we assume that the input S is a sequence.

In this section, we describe an algorithm that solves the selection problem with the expected running time of $O(n)$. This algorithm is known as *randomized quick-select*. The randomized-quick-select algorithm is very similar to the randomized quick-sort algorithm. We choose the pivot element randomly and divide S into three subsequences: L, E, G. L contains the elements that are smaller than the pivot, E contains the elements that are equal to the pivot, and G stores the elements that are greater than the pivot. Then we compare k with the sizes of the subsequences and, based on the results of the comparisons, we recurse on one of the subsequences.

The algorithm is shown below:

Code Segment 5.14

```
Algorithm quickSelect (S, k)
Input: Sequence S of n comparable elements and an integer k ε [1,N]
Output: The kth smallest element in S
if n == 1
  return the (first) element
pick a random pivot element x of S and divide S into three subsequences:
   • L, storing the elements in S less than x
   • E, storing the elements in S equal to x
   • G, storing the elements in S greater than x
if k ≤ |L| then
  return quickSelect(L, k)
else if k  ≤ |L| + |E|
  return x
else
  return quickSelect(G, k - |L| - |E|)
```

Figure 5.27 illustrates three cases. Note that the three cases are independent of each other, with different three k values. The array is divided to three subsequences. The sizes of three subsequences are as follow: $|L| = 6, |E| = 2$, and $|G| = 4$.

- *First case*: $k = 5$ and $k \leq |L|$; a recursive call is made on L with new $k = 5$.
- *Second case*: $k = 7$ and $k \leq |L| + |E|$ (this condition is checked after the test for the first case); returns 9.
- *Third case*: $k = 10$ and $k > |L| + |E|$; a recursive call is made on G with new $k = 2 (\text{new } k = \text{current } k - |L| - |E| = 10 - 6 - 2)$.

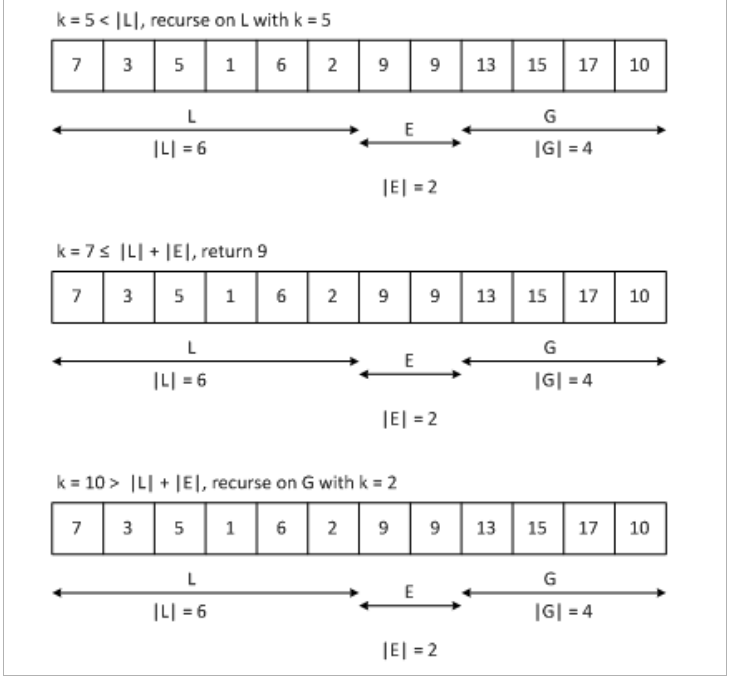


Figure 5.27. Illustration of the selection problem.

Section 5.3 Greedy Method and Dynamic Programming

Section 5.3. Greedy Method and Dynamic Programming

Overview

In this section, we briefly discuss the *greedy method* and *dynamic programming*, both which are used to solve optimization problems.

Section 5.3.1. Huffman Code

When we solve an optimization problem, we need to make a series of choices. When making a choice, the greedy method considers all options that are "available at that moment" and chooses the best option among them. In other words, it chooses a "locally optimal" option. The greedy method does not always lead to a global optimal solution, but it does for many practical problems. As an example of the greedy method, we describe the *Huffman code* algorithm.

Suppose we want to encode a text message to a binary representation. We can assign a fixed number of bits to each character. This type of code is called a *fixed-length code*. For example, let's assume, for simplicity, that there are only eight distinct characters in text messages. Then we can assign three bits to each character. The binary representation—or encoded representation, in general—of each character is called a *code-word.* If the message contains total a total of 100 characters, we need 300 bits to encode the message.

Now, let's assume that we know how frequently each character appears in the message. We can use this information to reduce the total number of bits in the encoded message. The idea is to assign a smaller number of bits to a character that appears more frequently and greater number of bits to a character that appears less frequently. In this code, the numbers of bits assigned to

characters are not the same. This type of code is called a *variable-length code*.

Suppose that each character appears in the message with the frequency shown in the following table. Let's also assume that different numbers of bits are assigned to different characters, and the number of bits assigned to each character is shown in the table:

Character	A	B	C	D	E	F	G	H
Frequency	9	8	3	43	13	11	6	7
Bits	4	4	5	1	3	3	5	4

Then the total number of bits to encode the message is calculated as follows:

9 x 4 + 8 x 4 + 3 x 5 + 43 x 1 + 13 x 3 + 11 x 3 + 6 x 5 + 7 x 4 = 256

So, we can save approximately 15% of storage if we use the variable-length code instead of the fixed-length code.

One potential issue with a variable-length code is that there may be an ambiguity when decoding a binary message. Note that, when we decode an encoded binary message, we scan the message from left to right and decode it in a single pass without backtracking. As an example, suppose that the character A is encoded to 10, the character B is encoded to 101, and the character C is encoded to 0. Suppose also that we encoded a certain message, and the encoded message is 1010. This message can be decoded to two different messages: AA (10 and 10) and BC (101 and 0). To avoid an ambiguity, a variable-length code must be a *prefix code*. In a prefix code, no code-word is a prefix of another code-word. The reason we have an ambiguity in this example is that the code-word of A, 10, is a prefix of the code-word of B, 101.

The *Huffman code* is an optimal variable-length code, which is a prefix code. We now describe how to construct the Huffman code for a given message. The Huffman code is represented as a binary tree, which is called a *coding tree*. We construct the coding tree using the frequency information of the characters in the given message. Before discussing the algorithm to construct the coding tree, we show the coding tree built from the above frequency table as an example.

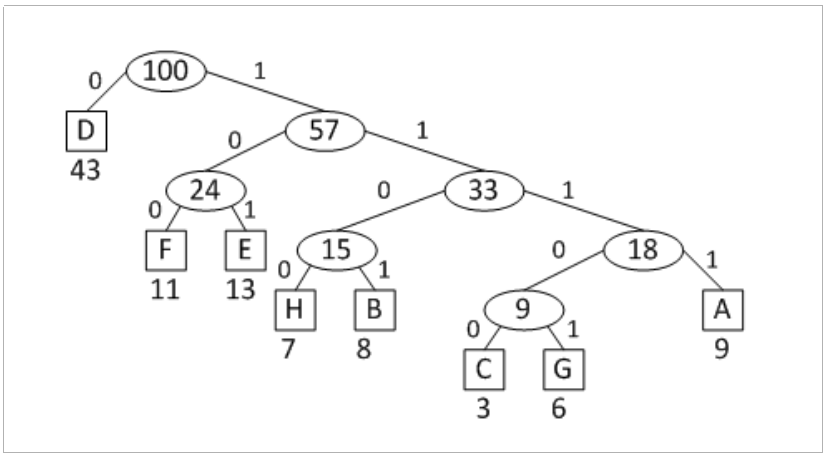


Figure 5.28. An example of a coding tree.

In the tree, each internal node, including the root, represents a frequency. Note that the frequency of a node is the sum of the frequencies of its children. Each leaf node represents a character and is associated with its frequency. Each edge is labeled 0 or 1. A left-child edge is labeled 0 and a right-child edge is labeled 1. The coding tree can be used for both encoding and decoding. When encoding, we can find the code-word of a character by following the path from the root to the leaf node of that character. For example,

the code-word of D is 0, that of E is 101, and that of G is 11101. When decoding, we scan the encoded binary message and, starting at the root, we move downward as dictated by the 1 or 0 that is being scanned. For example, suppose that an encoded message is 110110111110. This message is encoded in the following way:

- We scan the message one bit at a time, starting at the root of the coding tree. If the bit that is being scanned is 0, we follow the left edge. If it is 1, we follow the right edge.
- The first four bits—1101—leads to B, so the four-bit code-word is decoded to B.
- Once we have reached a leaf node and decoded one code-word, we return to the root and repeat the process.
- The next three bits, 101, are decoded to E.
- The next four bits, 1111, are decoded to A.
- Finally, the last bit, 0, is decoded to D.
- So, the original message is $BEAD$.

The following is a pseudocode for building a coding tree:

Code Segment 5.15

```
Algorithm Huffman(X) // X is the original message
Input: String X of length n with d distinct characters
Output: coding tree for X
Determine the frequency of each character c in X. Let f(c) be the frequency
  of the character c
Initialize a priority queue Q
for each character c in X do
  Create a single-node binary tree T storing c
  Insert T into Q with key f(c)
while Q.size() > 1 do
  Entry e1 = Q.removeMin() with e1 having key f1 and value T1
  Entry e2 = Q.removeMin() with e2 having key f2 and value T2
  Create a new binary tree T with left subtree T1 and right subtree T1
  Insert T into Q with key f1 + f2
Entry e = Q.removeMin() with e having tree T as its value
return tree T
```

The running time of the Huffman code algorithm is $O(n + d \log d)$, in which n is the number of characters in the string X, and d is the number of distinct characters in X.

We illustrate the process of building a coding tree using a smaller example. The frequency table used is shown below:

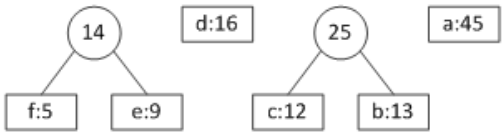
Character	a	b	c	d	e	f
Frequency	45	13	12	16	9	5

Note that this illustration does not include implementation details.

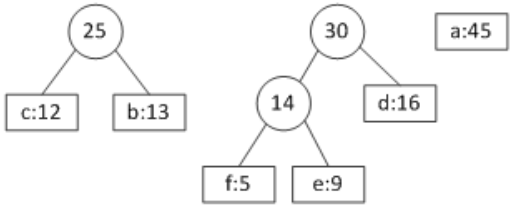
a. First, all characters, along with their frequencies, are inserted into a priority queue *Q*:

b. Two entries at the front of the queue, (*f*:5) and (*e*:9), are removed from *Q*, merged, and inserted back into *Q*. Note that when two entries are merged a new node is created and two entries become its children. The frequency of the new node is the sum of the frequencies of those two entries:

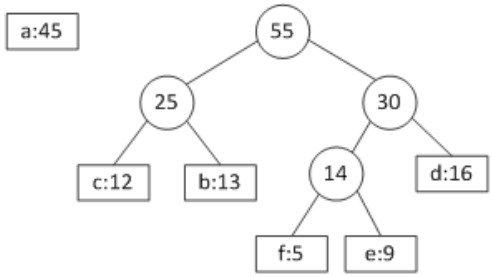
c. Two entries, (*c*:12) and (*b*:13), are removed from *Q*. they are merged and inserted back into *Q*:



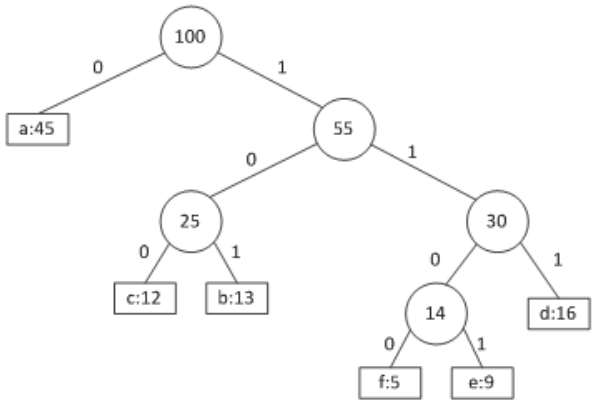
d. Two entries—((*f*:15, *e*:9):14) and (*d*:16)—are removed from *Q*, merged, and inserted back into *Q*.



e. ((*c*:12, *b*:13):25) and (((*f*:15, *e*:9):14, *d*:16):30) are removed from *Q*, merged, and inserted back into Q.



f. (*a*:45) and ((*c*:12, *b*:13):25, (((*f*:5, *e*:9):14, *d*:16):30):55) are removed from *Q*, merged, and inserted back into *Q*:



Section 5.3.2. Dynamic Programming

Consider two baseball teams, X and Y, which are competing for the World Series championship. A team wins the championship title if it wins four out of seven games. While the series is being played, we are interested in the probability that one of the teams—say, X—will eventually win the championship title, given that X still needs to win i more games to win the title, and Y still needs to win j more games to win the title. We use the notation $P(i, j)$ to denote this probability. For example, suppose that, so far, X has won one game and Y has won two games. Then X needs three more games and Y needs two more games, and the probability that X will win the championship title is denoted as $P(3, 2)$. We assume that the two teams are equally likely to win any particular game, or that each team has a 50% probability of winning a particular game.

Two extreme cases are $P(0, j)$ and $P(i, 0)$. The probability $P(0, j) = 1$ for any $j > 0$, for X does not need to win any more game, which means X has already won the championship. On the other hand, the probability $P(j, 0) = 0$ for any $i > 0$ because Y has already won the championship.

When $i > 0$ and $j > 0$, we can calculate $P(i, j)$ as follows. There is at least one more game to be played. If X wins the next game, then the probability that X will eventually win the championship is $P(i - 1, j)$, because now X has one fewer game to win. If Y wins the next game, then the probability that X will eventually win the championship is $P(i, j - 1)$, because Y now has one fewer game to win. Since each team is equally likely to win a particular game, we can calculate $P(i, j)$, after the next game, as an average of $P(i - 1, j)$ and $P(i, j - 1)$.

So, in general, we have the following:

$$\begin{aligned} P(i, j) &= 1, \text{ if } i = 0 \text{ and } j > 0 \\ &= 0, \text{ if } i > 0 \text{ and } j = 0 \\ &= (P(i - 1, j) + P(i, j - 1))/2, \text{ if } i > 0 \text{ and } j > 0 \end{aligned}$$

The following are examples:

$$\begin{aligned} P(7, 7) &= (P(6, 7) + P(7, 6))/2 \\ P(6, 7) &= (P(5, 7) + P(5, 6))/2 \\ P(7, 6) &= (P(6, 6) + P(7, 5))/2 \end{aligned}$$

This is a divide-and-conquer problem. To solve the problem $P(i, j)$, we first solve two smaller problems—($P(i - 1, j)$ and $P(i, j - 1)$—and combine them together as an average of the two solutions. We can implement this problem easily using a top-down recursion. However, a recursive solution, in this case, involves repetition of computations and thus is not efficient. For example, as shown above, to solve $P(7, 7)$ we need to solve $P(6, 7)$ and $P(7, 6)$. But, each of $P(6, 7)$ and $P(7, 6)$ solves

$P(6,6)$. This means that, a recursive call is made on $P(6,6)$ twice.

An alternative is a bottom-up approach. We solve smaller problems first (smaller problems refer to $P(i,j)$ with small i and j) and store the results in a table. When we solve a larger problem, we use the solutions to the smaller problems, which are stored in the table. This approach is called *dynamic programming*.

More specifically, first we solve $P(0,j)$ for all j (i.e., $j = 1,2,3,4,5,6$) and solve $P(i,0)$ for all i (i.e., $i = 1,2,3,4,5,6$). We then store the solution in a table, as shown below:

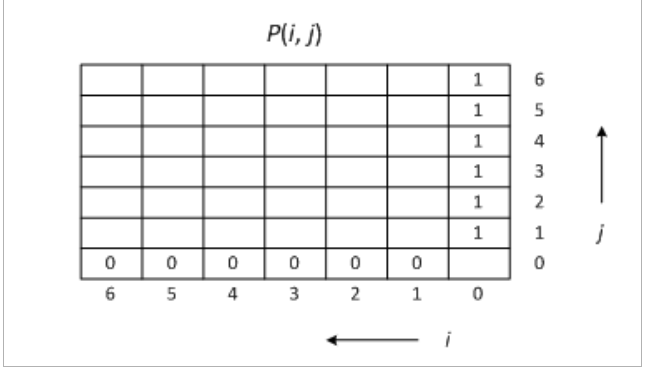


Figure 5.29(a). Table storing $P(0,j)$ and $P(i,0)$.

Then we proceed as follows:

$P(1,1) = (P(0,1) + P(1,0))/2 = (1+0)/2 = 1/2$; store this in the table
$P(1,2) = (P(0,2) + P(1,1))/2 = (1+1/2)/2 = 3/4$; store this in the table
$P(2,1) = (P(1,1) + P(2,0))/2 = (1/2+1)/2 = 1/4$; store this in the table

The updated table is as follows:

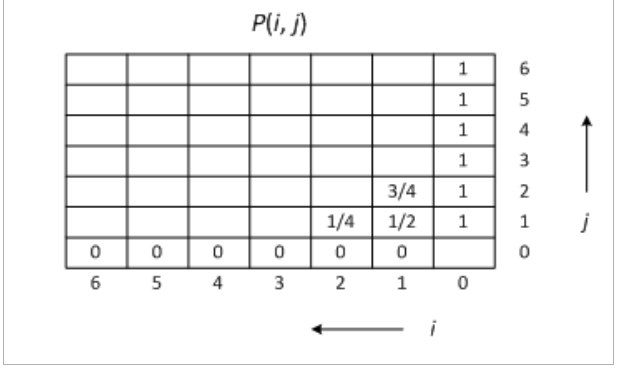


Figure 5.29(b). Table after storing $P(1,1)$, $P(1,2)$, and $P(2,1)$.

Suppose we compute the next three entries:

$P(1,3) = (P(0,3) + P(1,2))/2 = (1 + 3/4)/2 = 7/8$; store this in the table
$P(2,2) = (P(1,2) + P(2,1))/2 = (3/4 + 1/4)/2 = 1/2$; store this in the table
$P((3,1) = (P(2,1) + P(3,0))/2 = (1/4 + 0)/2 = 1/8$; store this in the table

The new updated table is as follows:

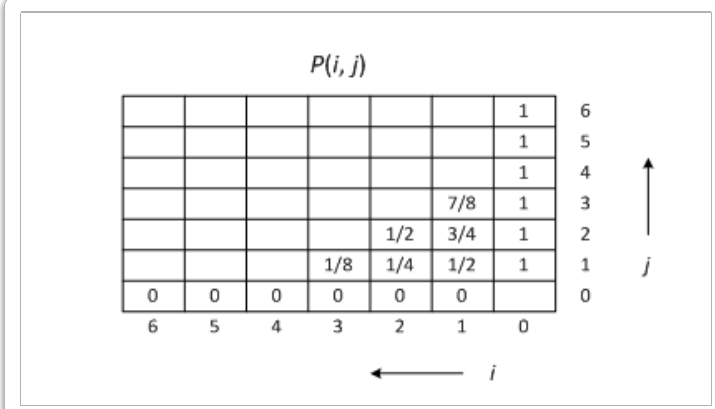


Figure 5.29(c). Table after storing $P(1,3)$, $P(2,2)$, and $P(3,1)$.

When we calculate an entry in the table, we use the entry below it and the entry to the right. The computation begins at the lower-right corner and proceeds diagonally toward the upper-left corner. In this way, no entry is calculated more than once, and this is more efficient than the recursive solution.

When solving a divide-and-conquer problem, we divide the given problem into smaller subproblems, solve the subproblems, and combine the solutions to the subproblems to obtain the solution to the original problem. So, recursion is a natural choice to solve a divide-and-conquer problem.

However, as illustrated by the above example, sometimes subproblems are not disjointed and instead overlap one another, which requires solving some subproblems multiple times. This problem can be avoided if we use a dynamic programming approach.

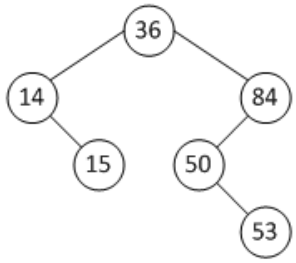
Module 5 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer or write your own program first, and then click "Show Answer" to compare yours to the suggested answer or the possible solution.

Test Yourself 5.3

Suppose that you insert a sequence of keys <36, 84, 14, 50, 15, 53>, in this order, into an empty binary search tree. Draw the resulting tree.

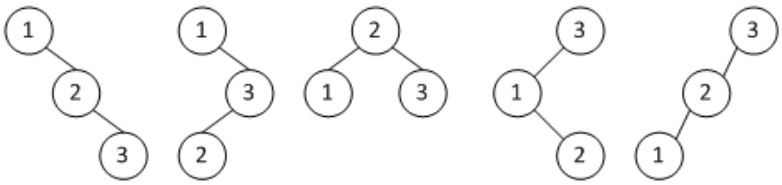
Suggested answer:



Test Yourself 5.4

How many different binary search trees can store keys {1, 2, 3}?

Suggested answer: Five.



Test Yourself 5.5

Code Segment 5.1 shows a recursive algorithm that searches a binary search tree (this code is also shown in page 461 of the textbook). Write an iterative version of the search algorithm in Java.

Suggested answer - one possible implementation:

```
private Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key) {
  Position<Entry<K,V>> walk = p;
  while (isInternal(walk)) {
    int comp = compare(key, walk.getElement( ));
    if (comp == 0)
      return walk; // key found; return its position
    else if (comp < 0)
      walk = left(walk);
    else
      walk = right(walk);
  }
  return walk; // unsuccessful search if this is reached
}
```

Test Yourself 5.6

When an entry is inserted into a binary search tree, it is always inserted at the leaf level. True or False?

True.

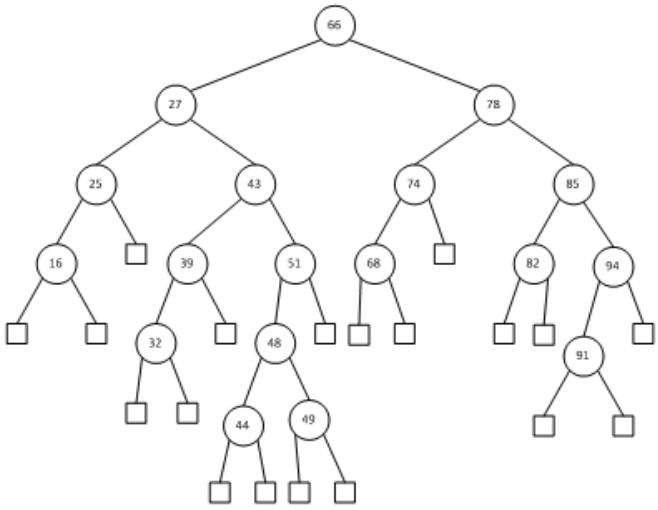
Your option is right.

False.

Your option is wrong.

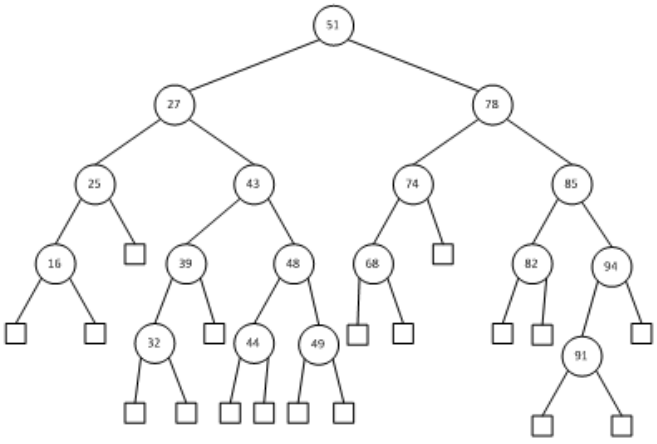
Test Yourself 5.7

Consider the following binary search tree:



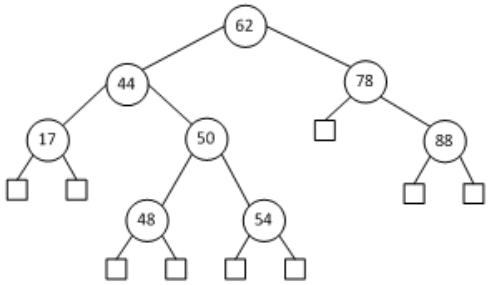
Suppose that you delete the root node (with the key 66) from the tree using the deletion algorithm described in Section 5.1.1.3 (this algorithm is also described in pages 464 in the textbook). Show the resulting tree.

Suggested answer:



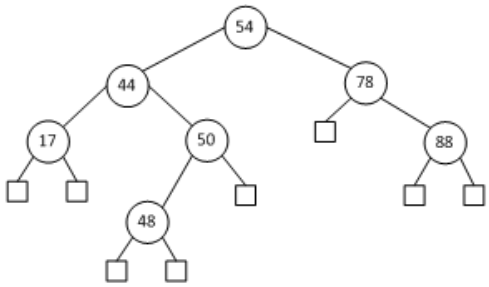
Test Yourself 5.8

Consider the following AVL tree:



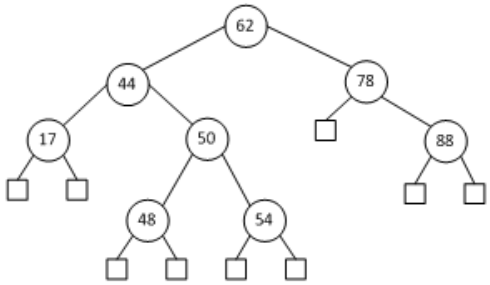
Show the resulting AVL tree after deleting a node with key 62 from the tree.

Suggested answer: After deleting 62, the tree is still height-balanced. So, no restructuring is performed.



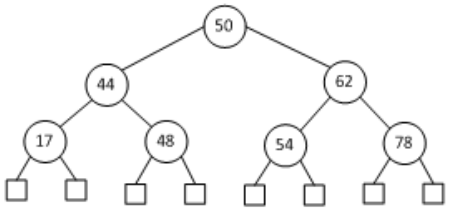
Test Yourself 5.9

Consider the following AVL tree:



Show the resulting AVL tree after deleting a node with key 88 from the tree.

Suggested answer: After deleting 88, the tree is not height-balanced. So, restructuring is performed.

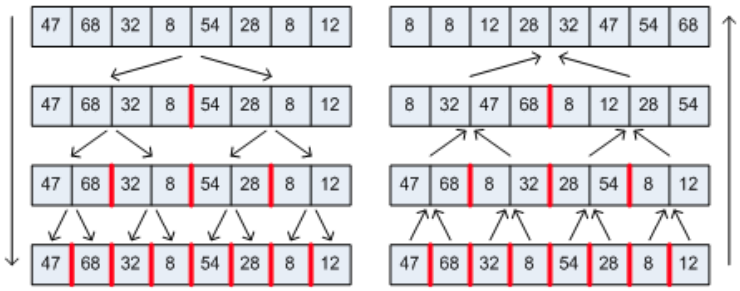


Test Yourself 5.10

Illustrate the execution of the merge-sort on the following sequence, using Figure 5.19 as a model.

<47, 68, 32, 8, 54, 28, 8, 12>

Suggested answer:



Test Yourself 5.11

Is the array-based implementation of the merge-sort described in Section 5.2.1.2 (which is also described in Section 12.1.2 of the textbook) stable?

Suggested answer: No.

Test Yourself 5.12

Suppose we have two *n*-element sorted sequences *A* and *B*. We assume that each sequence has distinct elements but it is possible that some elements are in both *A* and *B*. Describe an algorithm that computes a sorted sequence *C* that is a union of *A* and *B*. Note that the resulting sequence *C* should not have duplicates. Your algorithm must run in $O(n)$ time.

Suggested answer: First, we merge *A* and *B* using the same merge method used in the merge-sort. This takes $O(n)$ time. Then, we scan the resulting sequence and remove all duplicates. This also takes $O(n)$ time. Therefore, the total running time is $O(n)$.

Test Yourself 5.13

This question is about the array-based quick-sort algorithm described in Section 5.2.2.2 (which is also descxribed in Section 12.2.2 of the textbook). Explain what happens to the elements that are equal to the pivot.

Suggested answer:They may not be placed in the subsequence *E*. Instead, they may be dispersed across two subsequences.

Test Yourself 5.14

Describe a $O(n \log n)$ algorithm that removes all duplicates from a collection of n elements.

Suggested answer: First, we sort the collection using a $O(n \log n)$ sorting algorithm. Then, we scan the elements in the collection and remove duplicates, which takes $O(n)$. The total time is $O(n \log n)$.

Test Yourself 5.15

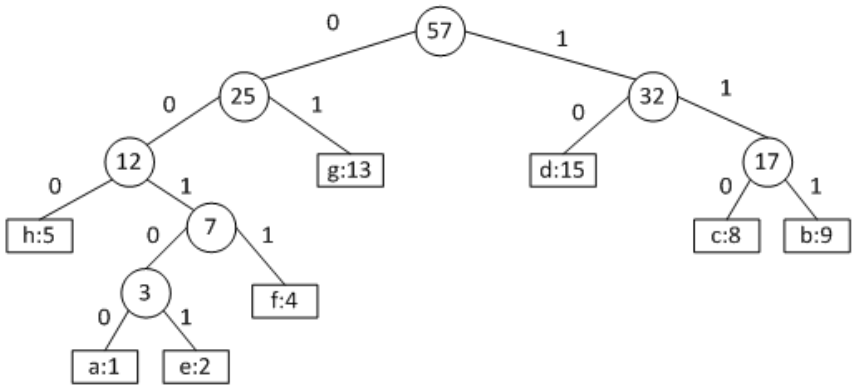
Is the bucket-sort described in Section 5.2.3.1 stable (which is also described in Section 12.3.2 of the textbook)?

Suggested answer: Yes, as far as all sequences are implemented as queues.

Test Yourself 5.16

Show the Huffman tree for the following frequency table:

Character	a	b	c	d	e	f	g	h
Frequency	1	9	8	15	2	4	13	5



Test Yourself 5.17

Encode *bag* and decode 00110010011000101 using the Huffman tree or Question 16.

Suggested answer - *bag* is encoded to 1110010001, and 00110010011000101 is decoded to *face*.

Test Yourself 5.18

18. This problem is about the World Series example used to illustrate the dynamic programming approach, which is discussed in Section 5.3.2. The table 5.29 (c), which is copied below, is partially filled. Your task is to calculate the following four probabilities and place them in the appropriate spaces in the table.

$$P(1,4), P(2,3), P(3,2), P(4,1)$$

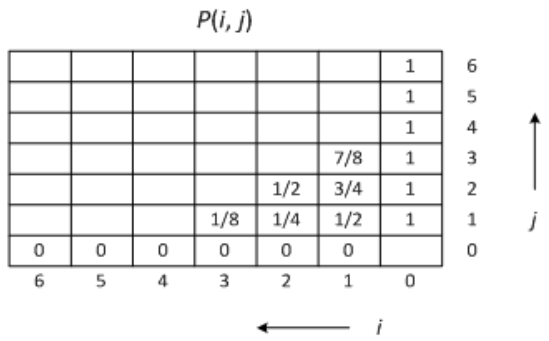
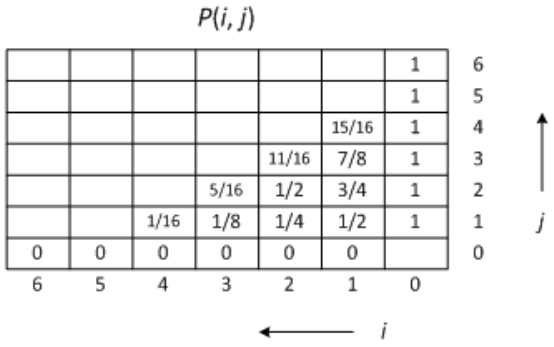


Figure 5.29.(c) Table after storing P(1,3), P(2,2), and P(3,1).

Suggested answer:



Test Yourself 5.19

Describe an efficient greedy algorithm that makes change for n cents using the fewest number of coins. Assume that available coins are quarters, dimes, nickels, and pennies.

Suggested answer: First give as many quarters as possible, then dimes, then nickels and finally pennies.

Test Yourself 5.20

20. Give an example set of denominations of coins so that a greedy change-making algorithm will not result in an optimal solution (the minimum number of coins).

Suggested answer:

Example 1: Denominations are 25 cents, 10 cents, and 1 cent.

> When n = 30, a greedy algorithm gives us six coins, 25, 1, 1, 1, 1, 1
> An optimal solution (the minimum number of coins) is three: 10, 10, 10

Example 2: Denominations are 25 cents, 24 cents, and 1 cent

> When n = 48, a greedy algorithm results in 24 coins: 1 quarter and 23 pennies
> An optimal solution (the minimum number of coins) is two: 2 24-cent coins

References

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java*(6th ed.) Wiley.
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). *The Java® language specification, Java SE 8 edition.* Oracle America Inc.
- Oracle. *The Java Tutorials*. Retrived from https://docs.oracle.com/javase/tutorial/index.html.
- Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.