Chapter 6

Storage/Retrieval I: Ordered Keys

In this chapter, we begin an examination of data structures for general storage and retrieval. We will assume that with each data item is associated a key that uniquely identifies the data item. An example of this kind of key might be a bank account number. Thus, if we provide a bank’s database with a customer’s account number, we should be able to retrieve that customer’s account information.

We will let Key denote the type to which the keys belong. We will assume that, even though the Key may not be a numeric type, it is still possible to sort elements of this type using a comparison operator $\leq$. Moreover, we assume that we need an efficient way to obtain data items in order of their keys. In the next chapter, we will examine data structures that do not require these two assumptions.

In order to facilitate arbitrary processing of all of the items in the data structure, we provide the Visitor interface, shown in Figure 6.1. We can then define an implementation of Visitor so that its Visit() operation does whatever processing we wish on a data item. (The use of this interface is known as the visitor pattern.)

We then define a Dictionary as a finite set of data items, each having a unique key of type Key, together with the operations shown in Figure 6.2. In the next chapter, we will examine implementations of this ADT. In this chapter, we will explore several implementations of the OrderedDictionary ADT, which is the extension of the Dictionary ADT obtained by adding the interface shown in Figure 6.3.
Figure 6.1 The Visitor interface

Precondition: true.
Postcondition: Completes without an error. May modify the state of $x$.
Visitor.Visit($x$)

Figure 6.2 The Dictionary ADT

Precondition: true.
Postcondition: Constructs an empty Dictionary.

Dictionary()
Precondition: $k$ is a Key.
Postcondition: Returns the element with key $k$, or nil if no item with key $k$ is contained in the set.

Dictionary.Get($k$)
Precondition: $x \neq$ nil, and $k$ is a Key that is not associated with any item in the set.
Postcondition: Adds $x$ to the set with key $k$.

Dictionary.Put($x$, $k$)
Precondition: $k$ is a Key.
Postcondition: If there is an item with key $k$ in the set, this item is removed.

Dictionary.Remove($k$)
Precondition: true.
Postcondition: Returns the number of items in the set.

Dictionary.Size()
Figure 6.3 Interface extending Dictionary to OrderedDictionary

Precondition: \( v \) is a Visitor.
Postcondition: Applies \( v.\text{Visit}(x) \) to every item \( x \) in the set in order of their keys.

\[
\text{OrderedDictionary.VisitInOrder}(v)
\]

Figure 6.4 Printer implementation of Visitor

Structural Invariant: true.

Precondition: true.
Postcondition: Prints \( x \).

\[
\text{Printer.Visit}(x) \\
\text{ } \text{print } x
\]

Example 6.1 Suppose we would like to print all of the data items in an instance of OrderedDictionary in order of keys. We can accomplish this by implementing Visitor so that its Visit operation prints its argument. Such an implementation, Printer, is shown in Figure 6.4. Note that we have strengthened the postcondition over what is specified in Figure 6.1. This is allowable because an implementation with a stronger postcondition and/or a weaker precondition is still consistent with the specification. Having defined Printer, we can print the contents of the OrderedDictionary \( d \) with the statement,

\[
d.\text{VisitInOrder(new Printer())}
\]

In order to implement OrderedDictionary, it is possible to store the data items in a sorted array, as we did with SortedArrayPriorityQueue in Section 5.1. Such an implementation has similar advantages and disadvantages to those of SortedArrayPriorityQueue. Using binary search, we can find an arbitrary data item in \( \Theta(\lg n) \) time, where \( n \) is the number of items in the dictionary. Thus, the Get operation can be implemented to run in \( \Theta(\lg n) \) time. However, to add or remove an item requires \( \Theta(n) \) time in the worst case. Thus, Put and Remove are inefficient using such an implementation. The fact that the elements of an array are located continuosly
allows random access, which, together with the fact that the elements are sorted, facilitates the fast binary search algorithm. However, it is exactly this contiguity that causes updates to be slow, because in order to maintain sorted order, elements must be moved to make room for new elements or to take the place of those removed.

If we were to use a linked list instead of an array, we would be able to change the structure without moving elements around, but we would no longer be able to use binary search. The “shape” of a linked list demands a sequential search; hence, look-ups will be slow. In order to provide fast updates and retrievals, we need a linked structure on which we can approximate a binary search.

6.1 Binary Search Trees

Consider the binary tree structure shown in Figure 6.5. Integer keys of several data items are shown in the nodes. Note that the value of the root is roughly the median of the keys in the structure; i.e., about half of the remaining keys are smaller. The smaller keys are all in the left child, whereas the larger keys are all in the right child. The two children are then structured in a similar way. Thus, if our search target is smaller than the
key at the current node, we look in the left child, and if it is larger, we look in the right child. Because this type of search approximates a binary search, this structure is called a *binary search tree*.

More formally we define a *binary search tree (BST)* to be a binary tree satisfying the following properties:

- Each node contains a data item together with its key.
- If the BST is nonempty, then both of its children are BSTs, and the key in its root node is
  - strictly larger than all keys in its left child; and
  - strictly smaller than all keys in its right child.

We can represent a binary search tree with the following variables:

- *elements*, which refers to a `BinaryTreeNode` (see Section 5.2); and
- *size*, which is an integer giving the number of data items in the set.

We interpret *elements* as representing a binary search tree as follows. If its *root* variable is `nil`, then the represented BST is empty. Otherwise, its *root* variable is a `Keyed` item (see page 148) containing the data item stored in the root with its associated key, and its *leftChild* and *rightChild* variables represent the left and right children, respectively. Our structural invariant is that *elements* refers to a binary search tree according to the above interpretation, and that *size* gives the number of nodes in the represented binary search tree. Note that this invariant implies that all non-nil *root* variables in the `BinaryTreeNode`s refer to `Keyed` items.

Because both *Get* and *Remove* require that we find a node with a given key, we will design an internal function `Find(k, t)` to find the node in the BST rooted at *t* with key *k*. If no such key exists in the tree, `Find(k, t)` will return a reference to the empty subtree in which it could be inserted; thus, we can also use `Find` in implementing *Put*.

Consider how we might implement `Find(k, t)`. If *t* is empty, then obviously it does not contain the key. Otherwise, we should first check the root. If we don’t find our key, we should see how it compares with the key at the root. Proceeding top-down, we can then find the key in either the left or right child, depending on the outcome of the comparison. This algorithm is a transformation, and so can be implemented as a loop, as is shown in Figure 6.6.
Figure 6.6 BSTDictionary implementation of OrderedDictionary, part 1

**Structural Invariant:** elements refers to a BST, and size gives the number of nodes in the represented BST.

```plaintext
BSTDictionary()  
    elements ← new BinaryTreeNode(); size ← 0

BSTDictionary.Get(k)  
    return Find(k, elements).Root().Data()

BSTDictionary.Put(x, k)  
    if x = nil  
        error  
    else  
        t ← Find(k, elements)  
        if t.Root() = nil  
            size ← size + 1  
            t.SetRoot(new Keyed(x, k))  
            t.SetLeftChild(new BinaryTreeNode())  
            t.SetRightChild(new BinaryTreeNode())
```

— Internal Functions Follow —

**Precondition:** k is a key and t is a BinaryTreeNode representing a BST.

**Postcondition:** Returns the BinaryTreeNode in t with key k, or in which k could be inserted if k is not in t.

```plaintext
BSTDictionary.Find(k, t)  
    // Invariant: k belongs in the subtree represented by t  
    while t.Root() ≠ nil and k ≠ t.Root().Key()  
        if k < t.Root().Key()  
            t ← t.LeftChild()  
        else  
            t ← t.RightChild()  
    return t
```
Complete implementations of Get and Put are shown in Figure 6.6. The implementation of Remove requires a bit more work, however. Once we find the node containing the key \( k \), we need to be able to remove that node \( t \) while maintaining a binary search tree. If \( t \) has no nonempty children, all we need to do is remove it (i.e., make the subtree empty). If \( t \) has only one nonempty child, we can safely replace the node to which \( t \) refers by that child. The difficulty comes when \( t \) has two nonempty children. In order to take care of this case, we replace \( t \) with the node in its right child having smallest key. That node must have an empty left child, so we can easily remove it. Furthermore, because it is in the right child of \( t \), its key is greater than any key in the left child of \( t \); hence, moving it to the root of \( t \) maintains the BST structure.

We can find the smallest key in a nonempty BST by first looking at the left child of the root. If it is empty, then the root contains the smallest key. Otherwise, the smallest key is in the left child. This is a transformation, and so can be implemented using a loop. The complete implementation of Remove is shown in Figure 6.7. Figure 6.8 shows the result of deleting 54 from the BST shown in Figure 6.5. Specifically, because 54 has two children, it is replaced by the smallest key (64) in its right child, and 64 is replaced by its right child (71).

The VisitInOrder operation requires us to apply v.Visit to each data item in the BST, in order of keys. If the BST is empty, then there is nothing to do. Otherwise, we must visit all of the data items in the left child prior to visiting the root, then we must visit all of the data items in the right child. Because the left and right children are themselves BSTs, they comprise smaller instances of this problem. Applying the top-down approach in a straightforward way, we obtain the recursive internal function TraverseInOrder shown in Figure 6.7.

The above algorithm implemented by TraverseInOrder is known as an inorder traversal. Inorder traversal applies strictly to binary trees, but two other traversals apply to rooted trees in general. A preorder traversal visits the root prior to recursively visiting all of its children, whereas a postorder traversal visits the root after recursively visiting all of its children.

Let us now analyze the running time of Find. Let \( n \) be the number of data items in the BST. Clearly, the time required outside the loop and the time for a single iteration of the loop are each in \( \Theta(1) \). We therefore need to analyze the worst-case number of iterations of the loop. Initially, \( t \) refers to a BST with \( n \) nodes. A single iteration has the effect of resetting \( t \) to refer to one of its children. In the worst case, this child may contain all nodes except the root. Thus, in the worst case, the loop may iterate \( n \)
BSTDictionary.Remove(k)
  t ← Find(k, elements)
  if t.Root() ≠ nil
    if t.LeftChild().Root() = nil
      Copy(t.RightChild(), t)
    else if t.RightChild().Root() = nil
      Copy(t.LeftChild(), t)
    else
      m ← t.RightChild()
      // Invariant: The smallest key in the right child of t is in the
      // subtree rooted at m.
      while m.LeftChild().Root() ≠ nil
        m ← m.LeftChild()
      t.SetRoot(m.Root()); Copy(m.RightChild(), m)

BSTDictionary.VisitInOrder(v)
 TraverseInOrder(elements, v)

— Internal Functions Follow —

Precondition: source and dest are BinaryTreeNode.
Postcondition: Copies the contents of source to dest.

BSTDictionary.Copy(source, dest)
  dest.SetRoot(source.Root())
  dest.SetLeftChild(source.LeftChild())
  dest.SetRightChild(source.RightChild())

Precondition: t is a BinaryTreeNode representing a BST, and v is a Visitor
Postcondition: Applies v.Visit to every node in t in order of their keys.

BSTDictionary.TraverseInOrder(t, v)
  if t.Root() ≠ nil
    TraverseInOrder(t.LeftChild(), v)
    v.Visit(t.Root().Data())
    TraverseInOrder(t.RightChild(), v)
times. This can happen, for example, if all left children are empty, so that elements refers to a BST that consists of a single chain of nodes going to the right (see Figure 6.9). The worst-case running time is therefore in $\Theta(n)$.

**Example 6.2** Suppose we build a BSTDictionary by inserting $n$ items with integer keys $1, 2, \ldots, n$, in that order. As each key is inserted, it is larger than any key already in the BST. It is therefore inserted to the right of every key already in the BST. The result is shown in Figure 6.9. It is easily seen that to insert key $i$ requires $\Theta(i)$ time. The total time to build the BSTDictionary is therefore in $\Theta(n^2)$, by Theorem 3.28.

The worst-case performance of a binary search tree is no better than either a sorted array or a sorted linked list for implementing ORDEREDDICTIONARY. In fact, in the worst case, a binary search tree degenerates into a sorted linked list, as Figure 6.9 shows. However, the performance of a binary search tree is not nearly as bad as the worst-case analysis suggests. In practice, binary search trees often yield very good performance.

In order to help us to understand better the performance of a binary search tree, let us analyze the worst-case running time of FIND in a somewhat different way. Instead of analyzing the running time in terms of the
number of data items, let us instead analyze it in terms of the height of the tree. The analysis is similar, because again in the worst case the child selected in the loop has height one less than the entire subtree. Thus, the worst-case running time of \texttt{Find} is in $\Theta(h)$, where $h$ is the height of the tree.

What this analysis tells us is that in order to achieve good performance from a binary search tree, we need a tree whose height is not too large. In practice, provided the pattern of insertions and deletions is somewhat random, the height of a binary search tree tends to be in $\Theta(l \! \! \log n)$. It is not hard to see that the worst-case running time of \texttt{Get}, \texttt{Put}, and \texttt{Remove} are all in $\Theta(h)$. Thus, if the height of the tree is logarithmic, we can achieve good performance. In the next section, we will show how we can modify our implementation to guarantee logarithmic height in the worst case, and thereby achieve logarithmic performance from these three operations.

Before we leave ordinary binary search trees, however, let us analyze the worst-case running time of \texttt{TraverseInOrder}, which does not use the \texttt{Find} function. We cannot analyze this operation completely because we cannot analyze $v.$\texttt{Visit} without knowing how it is implemented. Assuming the correctness of \texttt{TraverseInOrder} (whose proof we leave as an exercise), we can nevertheless conclude that $v.$\texttt{Visit} is called exactly once for each data item, and so must take at least $\Omega(n)$ time. What we would like to analyze is the time required for everything else. This amounts to analyzing the overhead involved in applying $v.$\texttt{Visit} to every data item.

Ignoring the call to $v.$\texttt{Visit} and the recursive calls, it is easily seen that
the remaining code runs in \( \Theta(1) \) time. However, setting up a recurrence describing the worst-case running time, including the recursion but excluding calls to \( v.\text{Visit} \), is not easy. We must make two recursive calls, but all we know about the sizes of the trees in these calls is that their sum is one less than the size of the entire tree.

Let us therefore take a different approach to the analysis. As we have already argued, \( v.\text{Visit} \) is called exactly once for each data item. Furthermore, it is easily seen that, excluding the calls made on empty trees, \( v.\text{Visit} \) is called exactly once in each call to \( \text{TraverseInOrder} \). A total of exactly \( n \) calls are therefore made on nonempty trees. The calls made on empty trees make no further recursive calls. We can therefore obtain the total number of recursive calls (excluding the initial call made from \( \text{VisitInOrder} \)) by counting the recursive calls made by each of the calls on nonempty trees. Because each of these calls makes two recursive calls, the total number of recursive calls is exactly \( 2n \). Including the initial call the total number of calls made to \( \text{TraverseInOrder} \) is \( 2n + 1 \). Because each of these calls runs in \( \Theta(1) \) time (excluding the time taken by \( v.\text{Visit} \)), the total time is in \( \Theta(n) \). Note that we cannot hope to do any better than this because the specification requires that \( v.\text{Visit} \) be called \( n \) times.

### 6.2 AVL Trees

In this section, we present a variant of binary search trees that guarantees logarithmic height. As a result, we obtain an implementation of \texttt{OrderedDictionary} for which the \texttt{Get}, \texttt{Put}, and \texttt{Remove} operations all run in \( \Theta(\lg n) \) time.

The key to keeping the height of a binary tree relatively small is in maintaining balance. The trick is to be able to achieve balance without too much overhead; otherwise the overhead in maintaining balance may result in poor overall performance. Consequently, we need to be careful how we define “balance”, so that our balance criterion is not too difficult to maintain.

The balance criterion that we choose is that in any subtree, the heights of the two children differ by at most 1. For the purpose of this definition, we consider an empty tree to have height \( -1 \), or one less than the height of a tree containing a single node. A binary search tree obeying this balance criterion is known as an AVL tree; “AVL” stands for the names of its inventors, Adelson-Vel’skii and Landis.

Figure 6.10 shows an AVL tree of height 4 containing integer keys. Note that its balance is not perfect – it is not hard to construct a binary tree of
height 3 with even more nodes. Nevertheless, the children of each nonempty subtree have heights differing by at most 1, so it is an AVL tree.

Before we begin designing an AVL tree implementation of ORDERED-DICTIONARY, let us first derive an upper bound on the height of an AVL tree with \(n\) nodes. We will not derive this bound directly. Instead, we will first derive a lower bound on the number of nodes in an AVL tree of height \(h\). We will then transform this lower bound into our desired upper bound.

Consider an AVL tree with height \(h\) having a minimum number of nodes. By definition, both children of a nonempty AVL tree must also be AVL trees. By definition of the height of a tree, at least one child must have height \(h-1\). By definition of an AVL tree, the other child must have height at least \(h-2\). In order to minimize the number of nodes in this child, its height must be exactly \(h-2\), provided \(h \geq 1\). Thus, the two children are AVL trees of heights \(h-1\) and \(h-2\), each having a minimum number of nodes.

The above discussion suggests a recurrence giving the minimum number of nodes in an AVL tree of height \(h\). Let \(g(h)\) give this number. Then for \(h \geq 1\), the number of nodes in the two children are \(g(h-1)\) and \(g(h-2)\). Then for \(h \geq 1\),

\[
g(h) = g(h-1) + g(h-2) + 1, \tag{6.1}
\]

where \(g(-1) = 0\) (the number of nodes in an empty tree) and \(g(0) = 1\) (the number of nodes in a tree of height 0).
Example 6.3 Consider the subtree rooted at 25 in Figure 6.10. It is of height 1, and its two children are minimum-sized subtrees of height 0 and \(-1\), respectively. It is therefore a minimum-sized AVL tree of height 1, so \(g(1) = 2\). Likewise, the subtrees rooted at 53 and 74 are also minimum-sized AVL trees of height 1. The subtrees rooted at 31 and 79 are then easily seen to be minimum-sized AVL trees of height 2, so \(g(2) = 4\). In like manner it can be seen that \(g(3) = 7\), \(g(4) = 12\), and the entire tree is a minimum-sized AVL tree of height 4.

Recurrence (6.1) does not fit any of the forms we saw in Chapter 3. However, it is somewhat similar to the form of Theorem 3.31. Recall that we only need a lower bound for \(g(n)\). In what follows, we will derive a recurrence that fits the form of Theorem 3.31 and gives a lower bound for \(g(n)\).

We first observe that \(g\) must be a nondecreasing function. Thus, for \(h \geq 1\), if
\[
g_1(h) = 2g_1(h - 2) + 1
\]
where \(g_1(h) = g(h)\) for \(h < 1\), then
\[
g_1(h) \leq g(h) \quad \text{(6.2)}
\]
for all \(h \geq -1\).

Now if we let \(g_2(h) = g_1(2h)\) for all \(h\), we obtain
\[
g_2(h) = g_1(2h)
\]
\[
= 2g_1(2h - 2) + 1
\]
\[
= 2g_1(2(h - 1)) + 1
\]
\[
= 2g_2(h - 1) + 1.
\]

\(g_2\) then fits the form of Theorem 3.31. Applying this theorem, we obtain
\[
g_2(h) \in \Theta(2^h).
\]

Thus, for sufficiently large \(h\), there is a positive real number \(c_1\) such that
\[
g_1(2h) = g_2(h)
\]
\[
\geq c_1 2^h.
\]

Then for sufficiently large even \(h\),
\[
g_1(h) \geq c_1 2^{h/2}.
\]
For sufficiently large odd $h$, we have
\[
g_1(h) \geq g_1(h - 1) \\
\geq c_12^{(h-1)/2} \quad \text{(because $h - 1$ is even)} \\
= \frac{c_1}{\sqrt{2}} 2^{h/2},
\]
so that for some positive real number $c_2$ and all sufficiently large $h$,
\[
g_1(h) \geq c_22^{h/2}.
\]
(6.3)

Combining (6.3) with (6.2), we obtain
\[
c_22^{h/2} \leq g(h)
\]
for sufficiently large $h$. Applying $\lg$ to both sides and rearranging terms, we obtain
\[
h \leq 2(\lg g(h) - \lg c_2) \\
\in O(\lg g(h)).
\]

Because $g(h)$ is the minimum number of nodes in an AVL tree of height $h$, it follows that the height of an AVL tree is in $O(\lg n)$, where $n$ is the number of nodes. By a similar argument, it can be shown that the height is in $\Omega(\lg n)$ as well. We therefore have the following theorem.

**Theorem 6.4** The worst-case height of an AVL tree is in $\Theta(\lg n)$, where $n$ is the number of nodes.

By Theorem 6.4, if we can design operations that run in time linear in the height of an AVL tree, these operations will run in time logarithmic in the size of the data set. Certainly, adding or deleting a node will change the heights of some of the subtrees in an AVL tree; hence, these operations must re-establish balance. Computing the height of a binary tree involves finding the longest path, which apparently requires examining the entire tree. However, we can avoid recomputing heights from scratch if we record the height of each subtree. If the heights of both children are known, computing the height of the tree is straightforward.

We therefore define the data type `AVLNode`, which is just like `BinaryTreeNode`, except that it has an additional representation variable, `height`. This variable is used to record the height of the tree as an integer. As for the other three variables, we allow read/write access to `height`. The constructor
for AVLNode is just like the constructor for BinaryTreeNode, except that it also initializes height to −1.

To represent an OrderedDictionary using an AVL tree, we again use two variables, elements and size, as we did for BSTDictionary. In this representation, however, elements will refer to an AVLNode. Our structural invariant is that elements represents an AVL tree. We interpret this statement as implying that each height variable gives the height of the subtree at which it is rooted, or −1 if that subtree is empty.

We could define a Find function for this implementation as we did for BSTDictionary; in fact, because an AVL tree is a binary search tree, the same function would work. However, this function would not be useful in implementing the Put or Remove operations because we might need to change the shape of the tree at some other location in order to maintain the balance criterion.

Let us therefore consider how Put might be implemented. More generally, let us consider how a data item x might be inserted into an arbitrary AVL tree t, which may be a subtree of a larger AVL tree. To a certain extent, we need to proceed as in BSTDictionary.Find. Specifically, if t is empty, we can replace it with a single-node AVL tree containing x. Otherwise, we’ll need to compare keys and insert into the appropriate child. However, we are not yet finished, because the insertion into the child will have changed its shape; hence, we need to compare the heights of the two children and restore balance if necessary. Note that this reduction is not a transformation, due to the additional work required following the insertion into the child.

In order to complete the insertion function, we need to be able to restore the balance criterion after an insertion into one of the children. Clearly, if we insert into one particular child, the other child will be unchanged. Furthermore, if we specify the insertion function to cause the result to be an AVL tree, we know that both children will be AVL trees; hence, we only need to worry about restoring balance at the root. Before we can talk about how to restore balance at the root, we should consider how much difference there might be in the heights of the children. It stands to reason that an insertion should either leave the height unchanged or increase it by 1. We will therefore include this condition in the postcondition of our insertion function.

Based on the above discussion, it suffices to show how to balance a binary search tree whose children are AVL trees having a height difference of exactly 2. This restoration of balance is accomplished via rotations. Consider, for example, the rotation shown in Figure 6.11. In this figure, circles denote
Figure 6.11 A single rotate right

![Diagram](image)

single nodes, and triangles denote arbitrary subtrees, which in some cases may be empty. All nodes and subtrees are labeled in a way that corresponds to the order of nodes in a BST (e.g., subtree $c$ is to the right of node $b$ and to the left of node $d$). The rotation shown is known as a single rotate right. It is accomplished by promoting node $b$ to the root, then filling in the remaining pieces in the only way that maintains the ordering of keys in the BST. Suppose that in the “before” picture, the right child ($e$) has height $h$ and the left child has height $h + 2$. Because the left child is an AVL tree, one of its two children has a height of $h + 1$ and the other has a height of either $h$ or $h + 1$. Suppose subtree $a$ has a height of $h + 1$. Then it is easily seen that this rotation results in an AVL tree.

The rotation shown in Figure 6.11 does not restore balance, however, if subtree $a$ has height $h$. Because the left child in the “before” picture has height $h + 2$, subtree $c$ must have height $h + 1$ in this case. After the rotation, the left child has height $h$, but the right child has height $h + 2$. To take care of this case, we need another kind of rotation called a double rotate right, shown in Figure 6.12. It is accomplished by promoting node $d$ to the root and again filling in the remaining pieces in the only way that maintains the ordering of keys. Suppose that subtrees $a$ and $g$ have height $h$ and that the subtree rooted at $d$ in the “before” picture has height $h + 1$. This is then the case for which a single rotate fails to restore balance. Subtrees $c$ and $e$ may have heights of either $h$ or $h - 1$ (though at least one must have height $h$). It is therefore easily seen that following the rotation, balance is restored.

These two rotations handle the cases in which the left child has height 2 greater than the right child. When the right child has height 2 greater than the left child, a single rotate left or a double rotate left may be applied. These rotations are simply mirror images of the rotations shown in Figures
To complete our discussion of the insertion function, we must convince ourselves that if it changes the height of the tree, then it increases it by exactly 1. This is clearly the case if no rotation is done. Let us then consider the rotations shown in Figures 6.11 and 6.12. If either of these rotations is applied, then the data item must have been inserted into the left child, causing its height to increase from \( h + 1 \) to \( h + 2 \). The overall height of the tree had to have been \( h + 2 \) prior to the insertion. Following a single rotate right, it is easily seen that the height is either \( h + 2 \) or \( h + 3 \). Likewise, following a double rotate right, it is easily seen that the height is \( h + 2 \). Thus, the result of the insertion is to either leave the height unchanged or increase it by 1.

The insertion algorithm is shown as AVLDictionary.Insert in Figure 6.13. The Balance function can also be used by the deletion algorithm. The remainder of the AVLDictionary implementation, including the rotations, is left as an exercise. Note that the rotations must ensure that all height values except that of the root are correct. Specifically, the heights of node \( d \) in Figure 6.11 and nodes \( b \) and \( f \) in Figure 6.12 must be recomputed.

**Example 6.5** Suppose we were to insert the key 39 into the AVL tree shown in Figure 6.14(a). Using the ordinary BST insertion algorithm, 39 should be made the right child of 35, as shown in Figure 6.14(b). To complete the insertion, we must check the balance along the path to 39, starting at the bottom. Both 35 and 23 satisfy the balance criterion; however, the left child of 42 has height 2, whereas the right child has height 0. We
Figure 6.13 Some internal functions for AVLDICTIONARY implementation of OrderedDictionary

**Precondition:** \( t \) represents an AVL tree, and \( x \) is a Keyed item.

**Postcondition:** Inserts \( x \) into \( t \) if its key is not already there, resulting in an AVL tree whose height is either unchanged or increased by 1.

\[
\text{AVLDICTIONARY.Insert}(x, t) \\
\text{if } t.\text{Height}() = -1 \\
\quad t.\text{SetRoot}(x) \\
\quad t.\text{SetLeftChild}(\text{new AVLNode}) \\
\quad t.\text{SetRightChild}(\text{new AVLNode}) \\
\quad t.\text{SetHeight}(0) \\
\text{else if } x.\text{Key}() < t.\text{Root}().\text{Key}() \\
\quad \text{Insert}(x, t.\text{LeftChild}()) \\
\quad \text{Balance}(t) \\
\text{else if } x.\text{Key}() > t.\text{Root}().\text{Key}() \\
\quad \text{Insert}(x, t.\text{RightChild}()) \\
\quad \text{Balance}(t)
\]

**Precondition:** \( t \) refers to a nonempty AVLN ode representing a BST. The children of \( t \) are AVL trees whose heights differ by at most 2.

**Postcondition:** Arranges \( t \) into an AVL tree.

\[
\text{AVLDICTIONARY.Balance}(t) \\
\quad l \leftarrow t.\text{LeftChild}() \\
\quad r \leftarrow t.\text{RightChild}() \\
\text{if } l.\text{Height}() = r.\text{Height}() + 2 \\
\quad \text{if } l.\text{LeftChild}().\text{Height}() > r.\text{Height}() \\
\quad\quad \text{SingleRotateRight}(t) \\
\quad\text{else} \\
\quad\quad \text{DoubleRotateRight}(t) \\
\text{else if } r.\text{Height}() = l.\text{Height}() + 2 \\
\quad \text{if } r.\text{RightChild}().\text{Height}() > l.\text{Height}() \\
\quad\quad \text{SingleRotateLeft}(t) \\
\quad\text{else} \\
\quad\quad \text{DoubleRotateLeft}(t) \\
\quad t.\text{SetHeight}(\text{Max}(l.\text{Height}(), r.\text{Height}()) + 1)
\]
therefore need to perform a rotation at 42. To determine which rotation is appropriate, we compare the height of the left child of the left child of 42 (i.e., the subtree rooted at 11) with the right child of 42. Because both of these subtrees have height 0, a double rotate right is required at 42. To accomplish this rotation, we promote 35 to the root of the subtree (i.e., where 42 currently is), and place the nodes 23 and 42, along with the subtrees rooted at 11, 39, and 50, at the only locations that preserve the order of the BST. The result of this rotation is shown in Figure 6.14(c). Because the balance criterion is satisfied at 54, this tree is the final result.

It is not hard to see that each of the rotations can be implemented to run in $\Theta(1)$ time, and that BALANCE therefore runs in $\Theta(1)$ time. Let us now analyze INSERT. Excluding the recursion, this function clearly runs in $\Theta(1)$ time. At most one recursive call is made, and its second parameter has height strictly less than the height of $t$; in the worst case, it is 1 less. If $h$ is the height of $t$, then the worst-case running time of INSERT is given by

$$f(h) \in f(h - 1) + \Theta(1)$$
for \( h > 0 \). By Theorem 3.31, \( f(h) \in \Theta(h) \). By Theorem 6.4, \textsc{insert} therefore runs in \( \Theta(\lg n) \) time, where \( n \) is the size of the data set. Clearly, \textsc{put} can be written to operate in \( \Theta(\lg n) \) time. We leave as an exercise the design and analysis of logarithmic algorithms for \textsc{get} and \textsc{remove}.

### 6.3 Splay Trees

While a worst-case bound of \( \Theta(\lg n) \) for \textsc{ordereddictionary} accesses is good, bounding the worst case does not always result in the best performance. For example, in many applications, the so-called “80-20” rule holds; i.e., 80% of the accesses involve roughly 20% of the data items. This rule then applies recursively, so that 64% of the accesses involve roughly 4% of the data items. In order to get good performance in such an environment, we would like to structure the data so that the most commonly-accessed data items can be accessed more quickly.

One variation of a binary search tree that attempts to achieve this kind of performance is a \textit{splay tree}. Structurally, a splay tree is simply a binary search tree. Operationally, however, it is \textit{self-adjusting} — when it accesses an item, it brings that item to the root of the tree via a series of rotations. (The VisitInOrder operation is an exception to this rule, as it accesses all of the data items.) As a result, the more frequently accessed items tend to remain closer to the root.

No attempt is made to ensure any sort of balance in a splay tree. As a result, the operations run in \( \Theta(n) \) time in the worst case. However, when a long path is traversed, the rotations have the effect of shortening it by roughly half. Thus, an expensive operation improves future performance. As a result, the amortized running times of \textsc{get}, \textsc{put}, and \textsc{remove} are all in \( O(\lg n) \).

To get a rough idea of how rotations improve the data structure, suppose we have a long zig-zag path from the root to some node \( b \); i.e., by starting from the root, then taking first the left child, then the right child, and continuing to alternate, we eventually reach \( b \). We could then bring \( b \) to the root of the tree by a series of double rotations, each promoting \( b \) by two levels. Now referring to Figure 6.12, notice that the distance between the root and any descendant of \( d \) decreases by 1 for each rotation. The number of rotations is half the distance from the root to \( d \), so each descendant of \( d \) ends up closer to the root by half the original distance between the root and \( d \).

Unfortunately, single rotations are not as effective in improving the struc-
Figure 6.15 A zig-zig right rotation

![Diagram of a zig-zig right rotation](image)

...
do a double rotation whenever possible, so rather than using a recursive call at this point, we should go ahead and make another comparison. If we find the key \( k \), or if the appropriate grandchild is empty, we do a single rotation. Otherwise, we recursively look for \( k \) in the appropriate grandchild and do a double rotation. The algorithm is shown in Figure 6.16.

**Example 6.6** Suppose we were to do a Find on 60 in the splay tree shown in Figure 6.17(a). Because the length of the path to 60 is odd, we must begin with a single rotation. Figure 6.17(b) shows the result of doing a single rotate left at 53. We then proceed with double rotations to bring 60 to the root. In this case, only one rotation — a zig-zag left — is required. The result is shown in Figure 6.17(c).

The insertion algorithm cannot use Find because it must insert a new data item when an empty subtree is found. However, it can be patterned after the Find algorithm. The main difference is that because a data item is inserted into an empty tree, we will always rotate that node to the root. We therefore do not need to restrict its use to nonempty trees. The details are left as an exercise.

The deletion algorithm can, however, use Find. Suppose we want to delete key \( k \). We can use Find\((k, \text{elements})\) to move \( k \) to the root if it is present. If the right child is empty, we can simply make the left child the new root. Otherwise, we can use another internal function, Find-Min\((\text{elements}, \text{RightChild}())\), to move the minimum key \( m \) in the right child to the root of the right child. At this point, the right child has an empty left child, because there are no keys with values between \( k \) and its right child. The result is shown in Figure 6.18. We can therefore complete the deletion by making \( A \) the left child of \( m \) and making \( m \) the root (see Figure 6.18). The algorithm is given in Figure 6.19.

Let us now analyze the amortized running times of Get, Put, and Remove for SPLAYDICTIONARY. It is not hard to see that all of the recursive algorithms have constant running time, excluding recursive calls. Furthermore, each time a recursive call is made, a rotation is done. It is therefore sufficient to analyze the total number of rotations. Each rotation, therefore, will have an actual cost of 1.

In order to amortize the number of rotations, we need to find an appropriate potential function. Intuitively, an operation involving many rotations should improve the overall balance of the tree. The potential function should in some way measure this balance, decreasing as the balance increases. If the tree is very unbalanced, as in Figure 6.9, many of the subtrees have a
Figure 6.16 The Find internal function for the SplayDictionary implementation of OrderedDictionary.

**Precondition:** $k$ is a Key and $t$ is a reference to a BinaryTreeNode representing a nonempty BST.

**Postcondition:** If a data item with key $k$ is in $t$, then $t$ is rearranged into a BST with $k$ at the root.

```plaintext
SplayDictionary.Find(k, t)
    if $k < t$.Root().Key()
        if $t$.LeftChild().Root() ≠ nil
            if $k < t$.LeftChild().Root().Key()
                if $t$.LeftChild().LeftChild().Root() ≠ nil
                    Find(k, t.LeftChild().LeftChild()); ZigZigRight(t)
                else
                    SingleRotateRight(t)
            else
                SingleRotateRight(t)
        else if $k > t$.LeftChild().Root().Key()
            if $t$.LeftChild().RightChild().Root() ≠ nil
                Find(k, t.LeftChild().RightChild()); ZigZagRight(t)
            else
                SingleRotateRight(t)
        else
            SingleRotateRight(t)
    else if $k > t$.Root().Key()
        if $t$.RightChild().Root() ≠ nil
            if $k < t$.RightChild().Root().Key()
                if $t$.RightChild().LeftChild().Root() ≠ nil
                    Find(k, t.RightChild().LeftChild()); ZigZagLeft(t)
                else
                    SingleRotateLeft(t)
            else
                SingleRotateLeft(t)
        else if $k > t$.RightChild().Root().Key()
            if $t$.RightChild().RightChild().Root() ≠ nil
                Find(k, t.RightChild().RightChild()); ZigZigLeft(t)
            else
                SingleRotateLeft(t)
        else
            SingleRotateLeft(t)
```
Figure 6.17 Example of doing a Find on 60 in a splay tree

(a) The original tree

(b) A single rotate left is done at 53

(c) A zig-zag left is done at 50
Figure 6.18 The splay tree after the calls to \texttt{Find} and \texttt{FindMin} in \texttt{Remove}

![Diagram](image)

Figure 6.19 The \texttt{Remove} operation for the \texttt{SplayDictionary} implementation of \texttt{OrderedDictionary}

\begin{verbatim}
SplayDictionary.Remove(k)
  if elements.Root() \neq nil
    Find(k, elements)
    if elements.Root().Key() = k
      l ← elements.LeftChild()
      r ← elements.RightChild()
      if r.Root() = nil
        elements = l
      else
        FindMin(r); r.SetLeftChild(l); elements ← r

— \textbf{Internal Functions Follow} —

\textbf{Precondition:} \texttt{t} is a \texttt{BinaryTreeNode} representing a nonempty BST.
\textbf{Postcondition:} Arranges \texttt{t} into a BST whose smallest key is at the root.

\begin{verbatim}
SplayDictionary.FindMin(t)
  if t.LeftChild().Root() \neq nil
    if t.LeftChild().LeftChild().Root() = nil
      SingleRotateRight(t)
    else
      FindMin(t.LeftChild().LeftChild())
    ZigZigRight(t)
\end{verbatim}
\end{verbatim}
comparatively large number of nodes, whereas in a balanced tree, most of the subtrees have only a few nodes. It would therefore make sense to define the potential function to depend on the number of nodes in each subtree.

An example of such a potential function is the sum of the sizes of all of the subtrees. However, this potential function will not work. Consider what happens when 0 is inserted into the tree in Figure 6.9. It is inserted to the left of 1, then rotated to the root via a single rotate right. The original tree therefore ends up as the right child of the result. The potential function therefore increases by the number of nodes in the result. With an increase this large, we cannot achieve a logarithmic amortized cost.

In order to scale back the growth of the potential function, let us try applying the $\lg$ function to the size of each nonempty subtree. Specifically, let $|t|$ denote the number of nodes in a subtree $t$. We then define our potential function $\Phi(T)$ to be the sum of all $\lg |t|$ such that $t$ is a nonempty subtree of the entire tree $T$. In what follows, we will show that for each of the three operations, the amortized cost with respect to $\Phi$ is in $O(\lg n)$.

Because most of the rotations will be double rotations, let us begin by analyzing a zig-zag rotation. We will be focusing on the tree subtrees that are changed by the zig-zag rotation, as shown in Figure 6.20; thus $T_a$, $T_b$, and $T_c$ denote the subtrees rooted at $a$, $b$, and $c$, respectively, prior to the rotation, and $T'_a$, $T'_b$, and $T'_c$ denote the subtrees rooted at these nodes following the rotation. The amortized cost of the rotation will be the actual
cost (i.e., 1) plus the change in the potential function \( \Phi \). Noting that \( |T'_b| = |T_c| \), we conclude that the change in \( \Phi \) is

\[
\lg |T'_a| + \lg |T'_c| - \lg |T_a| - \lg |T_b|.
\]

(6.4)

We need to simplify the above expression. It will be much easier to use if we can bound it in terms of subtrees of the original tree. In particular, we would like to get an expression involving only \( |T_b| \) and \( |T_c| \) so that, when the amortized costs of all rotations in one operation are added together, perhaps most terms will cancel out.

Let us therefore apply Theorem 5.7 (page 174) to \( \lg |T'_a| + \lg |T'_c| \) in (6.4). We know that \( |T'_a| + |T'_c| \leq |T_c| \). By Theorem 5.7, we have \( \lg |T'_a| + \lg |T'_c| \leq 2 \lg |T_c| - 2 \). Using the fact that \( |T_a| > |T_b| \) and adding in the actual cost, we obtain the following upper bound on the amortized cost of a zig-zag rotation:

\[
2 \lg |T_c| - 2 - 2 \lg |T_b| + 1 = 2(\lg |T_c| - \lg |T_b|) - 1
\]

(6.5)

Let us now analyze the amortized cost of a zig-zig rotation. Referring to Figure 6.21 and adopting the same notational conventions as above, we see that the change in \( \Phi \) is

\[
\lg |T'_b| + \lg |T'_c| - \lg |T_a| - \lg |T_b|.
\]

(6.6)
In order to get a tight bound for this expression in terms of $\log |T_c| - \log |T_a|$, we need to be a bit more clever. We would again like to use Theorem 5.7. Note that $|T_a| + |T_c'| \leq |T_c|$; however, $\log |T_a| + \log |T_c'|$ does not occur in (6.6). Let us therefore both add and subtract $\log |T_a|$ to (6.6). Adding in the actual cost, applying Theorem 5.7, and simplifying, we obtain the following bound on the amortized cost of a zig-zig rotation:

$$
\log |T_b'| + \log |T_c'| + \log |T_a| - 2 \log |T_a| - \log |T_b| + 1 \\
\leq \log |T_b'| + 2 \log |T_c| - 2 - 2 \log |T_a| - \log |T_b| + 1 \\
\leq 3 \log |T_c| - 3 \log |T_a| - 1 \\
= 3(\log |T_c| - \log |T_a|) - 1. \tag{6.7}
$$

Finally, let us analyze the amortized cost of a single rotate. We refer to Figure 6.22 for this analysis. Clearly, the amortized cost is bounded by

$$
\log |T_b'| - \log |T_a| + 1 \leq \log |T_b| - \log |T_a| + 1. \tag{6.8}
$$

Because each operation will do at most two single rotations (recall that a deletion can do a single rotation in both the FIND and the FINDMIN), the “+ 1” in this bound will not cause problems.

We can now analyze the amortized cost of a FIND. We first combine bounds (6.5), (6.7), and (6.8) into a single recurrence defining a function $f(k, t)$ bounding the amortized cost of FIND($k, t$). Suppose FIND($k, t$) makes a recursive call on a subtree $s$ and performs a double rotation. We can then combine (6.5) and (6.7) to define:

$$
f(k, t) = 3(\log |t| - \log |s|) + f(k, s).
$$
For the base of the recurrence, suppose that either no rotation or a single rotate is done. Using (6.8), we can define
\[ f(k, t) = 3(\lg |t| - \lg |s|) + 1, \]
where \( s \) is the child rotated upward or \( t \) if no rotation is done.

It is easily seen that the above recurrence telescopes; i.e., when the value for \( f(k, s) \) is substituted into the value for \( f(k, t) \), the \( \lg |s| \) terms cancel. The entire recurrence therefore simplifies to
\[ f(k, t) = 3(\lg |t| - \lg |s|) + 1 \]
where \( s \) is the subtree whose root is rotated to the root of \( t \). Clearly, \( f(k, t) \in O(\lg n) \), where \( n \) is the number of nodes in \( t \). The amortized cost of \text{Find} \text{, and hence of Get, is therefore in } O(\lg n).

The analysis of \text{Put} is identical to the analysis of \text{Find,} except that we must also account for the change in \( \Phi \) when the new node is added to the tree. When the new node is added, prior to any subsequent rotations, it is a leaf. Let \( s \) denote the empty subtree into which the new leaf is inserted. The insertion causes each of the ancestors of \( s \), including \( s \) itself, to increase in size by 1. Let \( t \) be one of these ancestors other than the root, and let \( t' \) be the same subtree after the new node is inserted. Note that \( t' \) has no more nodes than does the parent of \( t \). If we think of the insertion as replacing the parent of \( t \) by \( t' \), then this replacement causes no increase in \( \Phi \). The only node for which this argument does not apply is the root. Therefore, the increase in \( \Phi \) is no more than \( \lg(n + 1) \), where \( n \) is the number of nodes in the tree prior to the insertion. The entire amortized cost of \text{Put} is therefore in \( O(\lg n) \).

Finally, let us consider the \text{Remove} operation. The \text{Find} has an amortized cost in \( O(\lg n) \). Furthermore, the amortized analysis of \text{Find} also applies to \text{FindMIN,} so that it is also in \( O(\lg n) \). Finally, it is easily seen that the actual removal of the node does not increase \( \Phi \). The amortized cost of \text{Remove} is therefore in \( O(\lg n) \) as well.

### 6.4 Skip Lists

We conclude this chapter by returning to the idea of using an ordered linked list to implement \text{OrderedDictionary}. Recall that the difficulty with this idea is that items must be accessed sequentially, so that a binary search cannot be used to find an item. A \textit{skip list} overcomes this difficulty by using
additional references to skip over portions of the list (see Figure 6.23). Using these additional references, a binary search can be approximated.

The main building block for a skip list is the data type SkipListNode, which represents a data item, its key, a level $n \geq 1$, and a sequence of $n$ values, each of which is either a SkipListNode or empty. The representation consists of three variables:

- **data**: a data item; and
- **key**: a Key;
- **links[1..n]**: an array of (possibly nil) SkipListNode.

We interpret data as the represented data item, key as its associated key, SIZEOf(links) as the level of the SkipListNode, and links[i] as the ith element of the sequence, where empty is represented by nil. We allow read access to key and data. The complete implementation is shown in Figure 6.24.

We represent the OrderedDictionary with four variables:

- **start**: a non-nil SkipListNode;
- **end**: a non-nil SkipListNode;
- **maxLevel**: a Nat; and
- **size**: a Nat.
Figure 6.24 The data type SkipListNode

**Structural Invariant:** true.

**Precondition:** \( x \) is a data item, \( k \) is a Key, and \( n \) is a non-zero Nat.

**Postcondition:** Constructs a SkipListNode containing data \( x \) with key \( k \) and level \( n \). All \( n \) elements of the sequence are empty.

\[
\text{SkipListNode}(x, k, n) \\
data \leftarrow x; \; \text{key} \leftarrow k; \; \text{links} \leftarrow \text{new Array}[1..n] \\
\text{for } i \leftarrow 1 \text{ to } n \\
\quad \text{links}[i] \leftarrow \text{nil}
\]

**Precondition:** true.

**Postcondition:** Returns the level.

\[
\text{SkipListNode.Level}() \\
\text{return SizeOf(links)}
\]

**Precondition:** \( i \) is a non-zero Nat no greater than the level.

**Postcondition:** Returns the \( i \)th element of the sequence, or nil if that element is empty.

\[
\text{SkipListNode.Link}(i) \\
\text{return links}[i]
\]

**Precondition:** \( i \) is a non-zero Nat no greater than the level, and \( s \) is a (possibly nil) reference to a SkipListNode.

**Postcondition:** Sets the \( i \)th element of the sequence to \( s \), or to empty if \( s \) is nil.

\[
\text{SkipListNode.SetLink}(i, s : \text{SkipListNode}) \\
\text{links}[i] \leftarrow s
\]
We interpret the represented set to be the data items in the linked list beginning with $start$ and ending with $end$, using the variables $\text{links}[1]$ to obtain the next element in the list; the data items in $start$ and $end$ are excluded from the set.

Our structural invariant is:

- Both $start$ and $end$ have a level of $M \geq \text{maxLevel}$.
- $start.key = \text{minKey}$, which is the smallest possible key.
- $end.key = \text{maxKey}$, which is the largest possible key.
- There is a sequence of $\text{SkipListNode}$s with length $\text{size} + 2$ obtained by starting with $start$ and following the $\text{links}[1]$ reference in each $\text{SkipListNode}$ until $end$ is reached. We will refer to this sequence as the level-1 sequence.
- For $1 < i \leq M$, there is a sequence obtained in the same way as above, but using the $\text{links}[i]$ variables instead of the $\text{links}[1]$ variables. We will refer to this sequence as the level-$i$ sequence. The level-$i$ sequence is the subsequence of the level-($i - 1$) sequence containing those $\text{SkipListNode}$s having level $i$ or greater.
- The keys in each sequence are strictly increasing in value.
- $\text{maxLevel}$ is the maximum level of any $\text{SkipListNode}$ in the above sequences, excluding $start$ and $end$. If $start$ and $end$ are the only $\text{SkipListNode}$s in the sequences, $\text{maxLevel} = 1$.

In order to be able to approximate a binary search with this data structure, the level-$i$ sequence should include roughly every second node from the level-($i - 1$) sequence, for $1 < i \leq \text{maxLevel}$. However, as is suggested by Figure 6.23, we will not explicitly maintain this property. Instead, we will use randomization to produce a structure that we expect, on average, to approximate this property.

When we insert a new data item, we first determine the level of its $\text{SkipListNode}$ via a series of flips of a fair coin. As long as the outcome of a coin flip is $\text{heads}$, we continue flipping. We stop flipping when the outcome is $\text{tails}$. The level of the $\text{SkipListNode}$ is the total number of flips. Because we flip the coin at least once, every level will be at least 1. Because the coin is fair, the probability of $\text{tails}$ is $1/2$; hence, we would expect about half of the $\text{SkipListNode}$s to have level 1. The probability of flipping $\text{heads}$ then $\text{tails}$ is $(1/2)^2 = 1/4$, so we would expect about $1/4$ of the $\text{SkipListNode}$s
to have level 2. In general, the probability of flipping \( i - 1 \) heads followed by 1 tails is \( 2^{-i} \). We would therefore expect the fraction of nodes having level \( i \) to be about \( 2^{-i} \). Because the coin is fair, these levels should be randomly distributed over the level-1 sequence.

Suppose a given \texttt{SkipListNode} has level \( k \). In order to insert it into the skip list, we need to insert it into its proper location in the level-\( i \) sequence for each \( 1 \leq i \leq k \). For the purpose of finding these insertion points, we will design a function \texttt{Find}(\( k, l \)), which will return an array of references such that at index \( i \), \( 1 \leq i \leq l \), is the reference to the \texttt{SkipListNode} of level at least \( i \) having the largest key strictly less than \( k \). We will then be able to use this function not only in the \texttt{Put} operation, but also in the \texttt{Get} and \texttt{Remove} operations. The \texttt{Put} operation is shown in Figure 6.25.

The partial implementation shown in Figure 6.25 does not explicitly handle the case in which the level of a new node exceeds the level of \texttt{start} and \texttt{end}. In what follows, we will assume that these arrays are expanded as needed using the expandable array design pattern. Later, we will argue that for all practical purposes, a fixed-sized array can be used. In the meantime, we note that the time needed to expand the array is proportional to the number of iterations of the \texttt{while} loop in \texttt{Put}, so we need not amortize this cost.

We will devote the remainder of this section to analyzing the expected running time of \texttt{Put}. This analysis is rather involved, but it uses some important new tools for analyzing expected running times. Furthermore, much of this analysis can be applied directly to the analyses of \texttt{Get} and \texttt{Remove}; we therefore leave these analyses as exercises. In view of the sometimes counter-intuitive nature of expected values, we will proceed with care.

We will partition the algorithm into five parts:

- the \texttt{while} loop;
- the call to \texttt{Find};
- the construction of a new \texttt{SkipListNode};
- the \texttt{for} loop; and
- the remainder of the algorithm for the case in which \( x \neq \texttt{nil} \).

At this point, let us observe that a worst-case input must have \( x \neq \texttt{nil} \) and \( k \) as a new key, not already in the set. On any such input, the overall running time is the sum of the running times of the above five parts. By the linearity
Figure 6.25 SkipListDictionary implementation (partial) of Ordered-Dictionary

SkipListDictionary()

start ← new SkipListNode(nil, minKey, 100)
end ← new SkipListNode(nil, maxKey, 100)

for i ← 1 to 100
    start.SetLink(i, end)

end ← new SkipListNode(nil, maxKey, 100)

size ← 0; maxLevel ← 1

SkipListDictionary.Put(x, k)

if x = nil
    error
else
    l ← 1
    while FlipCoin() = heads
        l ← l + 1
    p ← Find(k, l)
    if p[1].Link(1).Key() ≠ k
        maxLevel ← Max(maxLevel, l); q ← new SkipListNode(x, k, l)
        for i ← 1 to l
            q.SetLink(i, p[i].Link(i)); p[i].SetLink(i, q)

— Internal Functions Follow —

Precondition: k is a Key and l is a non-zero Nat, no larger than the levels of start or end.

Postcondition: Returns an array A[1..l] such that A[i] refers to the SkipListNode of level i or larger having the largest key strictly less than k.

SkipListDictionary.Find(k, l)

A ← new ARRAY[1..l]; p ← start
for i ← Max(maxLevel, l) to 1 by −1
    while p.Link(i).Key() < k
        p ← p.Link(i)
    if i ≤ l
        A[i] ← p
return A
CHAPTER 6. STORAGE/RETRIEVAL I: ORDERED KEYS

of expectation, the expected running time of the algorithm on a worst-case input is therefore the sum of the expected running times of these five parts on a worst-case input.

We begin by analyzing the while loop. We define the discrete probability space $Seq$ to be the set of all finite sequences of flips containing zero or more heads, followed by exactly one tails. Note that for each positive integer $i$, there is exactly one sequence in $Seq$ with length $i$; hence, $Seq$ is countable. As we have already argued, the probability of achieving a sequence of length $i$ is $2^{-i}$. In order to conclude that $Seq$ is a discrete probability space, we must show that

$$\sum_{i=1}^{\infty} 2^{-i} = 1.$$ 

This fact follows from the following theorem, using $c = 2$ and $a = -1$.

**Theorem 6.7** For any real numbers $a$ and $c$ such that $c > 1$,

$$\sum_{i=0}^{\infty} c^{a-i} = \frac{c^{a+1}}{c-1}.$$ 

**Proof:**

$$\sum_{i=0}^{\infty} c^{a-i} = \lim_{n \to \infty} \sum_{i=0}^{n} c^{a-i}$$

$$= c^a \lim_{n \to \infty} \sum_{i=0}^{n} (1/c)^i$$

$$= c^a \lim_{n \to \infty} \frac{(1/c)^{n+1} - 1}{1 - 1/c} \quad \text{from (2.2)}$$

$$= c^a \lim_{n \to \infty} \frac{c - (1/c)^n}{c - 1}$$

$$= \frac{c^{a+1}}{c-1}$$

because $1/c < 1$. \hfill \Box

We now define the discrete random variable $len$ over $Seq$ such that $len(e)$ is the length of the sequence of flips. Note that $E[len]$ gives us the expected number of times the while loop condition is tested, as well as the expected final value of $l$. As a result, it also gives us the expected number of iterations of the for loop, provided $k$ is not already in the set.
Because \( \text{len}(e) \) is always a natural number, we can apply Theorem 5.5 (page 172) to obtain \( E[\text{len}] \). The probability that a sequence has length at least \( i \) is the probability that \( i - 1 \) flips all result in heads, or \( 2^{1-i} \). Thus,

\[
E[\text{len}] = \sum_{i=1}^{\infty} 2^{1-i} = 2
\]

from Theorem 6.7. We can therefore expect the \textbf{while} loop in \texttt{Put} to iterate once, yielding an expected value of 2 for \( l \), on average. Hence, the \textbf{for} loop, if it executes, iterates twice on average. The expected running times of both loops are therefore in \( \Theta(1) \) for a worst-case input.

In order to determine the expected running time of the \texttt{SkipListNode\_Corrected} constructor, we need to analyze it again, but this time doing an expected-case analysis using \( \text{len} \) as its third parameter. Using the same analysis as we did for the \textbf{for} loop in \texttt{Put}, we see that its expected running time is in \( \Theta(1) \).

In order to complete the expected-case analysis of \texttt{Put}, we need to analyze \texttt{Find}. We will begin by defining an appropriate discrete probability space. Let \( \text{Seq}^n \) be the set of all \( n \)-tuples of elementary events from \( \text{Seq} \); i.e., each elementary event in \( \text{Seq}^n \) is an \( n \)-tuple \( \langle e_1, \ldots, e_n \rangle \) such that each \( e_i \) is a sequence of coin flips containing zero or more heads, followed by exactly one tails. Such an \( n \)-tuple describes the “shape” of a skip list by recording, for each of the \( n \) data elements, the sequence of coin flips which generated the level of the \texttt{SkipListNode\_Corrected} containing it.

In order to show that \( \text{Seq}^n \) is countable, we can label each \( n \)-tuple \( \langle e_1, \ldots, e_n \rangle \in \text{Seq}^n \) with the natural number

\[
p_1^{\text{len}(e_1)} p_2^{\text{len}(e_2)} \cdots p_n^{\text{len}(e_n)},
\]

where \( p_i \) is the \( i \)th prime. Because each elementary event in \( \text{Seq} \) is uniquely identified by its length, and because each positive integer has a unique prime factorization, each tuple has a unique label; hence, \( \text{Seq}^n \) is countable.

We need to define the probabilities of elements in \( \text{Seq}^n \). In order to do this properly, we need to extend the definition of independence given in Section 5.5 to more than two events. We say that a set \( S \) of events is \textit{pairwise independent} if for every pair of events \( e_1, e_2 \in S \), \( e_1 \) and \( e_2 \) are independent. If for every subset \( T \subseteq S \) containing at least two events,

\[
P \left( \bigcap_{e \in T} e \right) = \prod_{e \in T} P(e),
\]

This \( \prod \)-notation denotes the product of the probabilities \( P(e) \) for all events \( e \) in \( T \).
then we say the events in \( S \) are mutually independent. We leave as an exercise to show that pairwise independence does not necessarily imply mutual independence, even for 3-element sets of events.

Returning to \( \text{Seq}^n \), let \( \text{len}_{ij} \) denote the event that component \( i \) has length \( j \), for \( 1 \leq i \leq n \), \( j > 1 \). In any set \( \{ \text{len}_{i_{1j_1}}, \ldots, \text{len}_{i_{mn}} \} \) with \( 2 \leq m \leq n \) and all the \( i_k \)'s different, the events should be mutually independent. Furthermore, in order to be consistent with \( \text{Seq} \), we want \( P(\text{len}_i) = 2^{-j} \) for \( 1 \leq i \leq n \) and \( j > 1 \). We can satisfy these constraints by setting the probability of elementary event \( (e_1, \ldots, e_n) \) to the product of the probabilities in \( \text{Seq} \) of \( e_1, \ldots, e_n \); i.e.,

\[
P((e_1, \ldots, e_n)) = \prod_{i=1}^{n} 2^{-\text{len}(e_i)}.
\]

It can be shown by a straightforward induction on \( n \) that the sum of the probabilities of the elementary events in \( \text{Seq}^n \) is 1. \( \text{Seq}^n \) is therefore a discrete probability space.

We now need to determine what comprises a worst-case input to \text{Find}. As we suggested earlier, the components of an elementary event in \( \text{Seq}^n \) correspond to the \( n \) data elements in a skip list. The length of a component gives the level of the skip list element. Apart from the number of elements in the structure, the shape of a skip list is determined completely at random, independent of the data elements inserted or the order in which they are inserted. Specifically, the keys in the data elements determine their order, but their levels are determined solely by coin flips. Thus, in order to determine a worst-case input, we needn’t worry about how the structure was constructed — we need only concern ourselves with the parameters to \text{Find}. Both of these values can affect the running time.

In order to determine the worst-case input, we need to consider the behavior of the \textbf{while} loop. For a given value of \( i \), the \textbf{while} loop iterates once for each key at level \( i \) that is

- less than \( k \); and

- greater than the largest key less than \( k \) at any level \( j > i \) (or \(-\infty \) if there is no such key).

It is easily seen that at any level \( i \), the expected number of iterations is maximized when the number of keys less than \( k \) is maximized, because the levels of these keys are determined randomly. The worst-case input therefore has \( k \) greater than any key in the data set.
We therefore define \( \text{tail}_i(e) \), where \( e = \langle e_1, \ldots, e_n \rangle \in \text{Seq}^n \), to be the largest natural number \( j \) such that some suffix of \( e \) contains \( j \) components with length exactly \( i \) and no components longer than \( i \). Thus, if \( e \) contains at least one component strictly longer than \( i \), then \( \text{tail}_i(e) \) is the number of components with length \( i \) that follow the last component strictly longer than \( i \). Otherwise, \( \text{tail}_i(e) \) is simply the number of components with length \( i \).

**Example 6.8** Let \( e \) represent the skip list shown in Figure 6.23 on page 224. Then

- \( \text{tail}_1(e) = 0 \) because there are no level-1 nodes following the last node with level greater than 1;
- \( \text{tail}_2(e) = 2 \) because there are 2 level-2 nodes following the last node with level greater than 2; and
- \( \text{tail}_3(e) = 1 \) because there is 1 level-3 node, and there are no nodes with level greater than 3.

Suppose \( e \) describes some skip list with \( n \) elements, and suppose this skip list’s \text{Find} function is called with a key larger than any in the list. The running time of \text{Find} is then proportional to the number of times the while loop condition is tested. On iteration \( i \) of the for loop, the while loop will iterate exactly \( \text{tail}_i(e) \) times, but will be tested \( \text{tail}_i(e) + 1 \) times, including the test that causes the loop to terminate. The expected running time of \text{Find} on a worst-case input is therefore proportional to:

\[
E \left[ \sum_{i=1}^{\max(\maxLevel, l)} (\text{tail}_i + 1) \right]
\]

\[
= E \left[ \left( \sum_{i=1}^{\max(\maxLevel, l)} \text{tail}_i \right) + \max(\maxLevel, l) \right]
\]

\[
= E \left[ \sum_{i=1}^{\max(\maxLevel, l)} \text{tail}_i \right] + E[\max(\maxLevel, l)]. \quad (6.9)
\]

Let us first consider the first term in (6.9). It is tempting to apply linearity of expectation to this term; however, note that \( \maxLevel \) is a random variable, as its value depends on the levels of the nodes in the skip list. Theorem 5.9 therefore does not apply to this term. In particular, note that for
any positive $n$ and $i$, there is a non-zero probability that there is at least one node at level $i$; hence, there is a non-zero probability that $tail_i$ is positive.

The proper way to handle this kind of a summation, therefore, is to convert it to an infinite sum. The term inside the summation should be equal to $tail_i$ when $i \leq \max(maxLevel, l)$, but should be 0 for all larger $i$. In this case, it is easy to derive such a term, as $tail_i = 0$ when $i > maxLevel$. We therefore have:

$$E \left[ \sum_{i=1}^{\max(maxLevel, l)} tail_i \right] = E \left[ \sum_{i=1}^{\infty} tail_i \right] = \sum_{i=1}^{\infty} E[tail_i]. \quad (6.10)$$

By Theorem 5.5,

$$E[tail_i] = \sum_{j=1}^{\infty} P(tail_i \geq j).$$

Suppose that there are at least $j$ components with length at least $i$. In order for $tail_i \geq j$, the $i$th coin flip in each of the last $j$ of these components must be tails. The probability that $j$ independent coin flips are all tails is $2^{-j}$. However, this is not the probability that $tail_i \geq j$, but rather the conditional probability given that there are at least $j$ components with length at least $i$. Let $num_i$ denote the number of components whose length is at least $i$. We then have

$$P(tail_i \geq j \mid num_i \geq j) = 2^{-j}.$$

Fortunately, this conditional probability is closely related to $P(tail_i \geq j)$. Specifically, in order for $tail_i \geq j$, it must be the case that $num_i \geq j$. Thus, the event $tail_i \geq j$ is a subset of the event $num_i \geq j$. Therefore, from (5.2) we have

$$P(tail_i \geq j) = P((tail_i \geq j) \cap (num_i \geq j))$$
$$= P(num_i \geq j)P(tail_i \geq j \mid num_i \geq j)$$
$$= P(num_i \geq j)2^{-j}.$$

Unfortunately, computing the exact value of $P(num_i \geq j)$ is rather difficult. We will therefore content ourselves with observing that because it is
a probability, it can be no more than 1. We therefore have,

\[ E[\text{tail}_i] = \sum_{j=1}^{\infty} P(\text{tail}_i \geq j) \]

\[ = \sum_{j=1}^{\infty} P(\text{num}_i \geq j)2^{-j} \]

\[ \leq \sum_{j=1}^{\infty} 2^{-j} \]

\[ = 1 \]

from Theorem 6.7.

This bound seems quite good, perhaps even surprisingly so. It tells us that on any iteration of the for loop, we can expect the while loop to iterate no more than once, on average. Still, this bound does not give a finite bound for (6.10). However, we have already observed that for any \( e \in \text{Seq}^n \), \( \text{tail}_i(e) \) will be 0 for all but finitely many \( i \). This follows because there are only finitely many nonempty levels. Consequently, we might want to use the fact that \( \text{tail}_i(e) \leq \text{num}_i(e) \); hence, \( E[\text{tail}_i] \leq E[\text{num}_i] \).

While this bound would yield a finite bound for (6.10), it unfortunately is still too loose, as \( \text{num}_1(e) = n \) for every \( e \in \text{Seq}^n \). We would like to derive a logarithmic upper bound, if possible. However, we can use a combination of the two bounds. In particular, the bound of 1 seems to be a good upper bound as long as it is less than \( E[\text{num}_i] \). Once \( i \) is large enough that \( E[\text{num}_i] \leq 1 \), \( E[\text{num}_i] \) would be a better bound. If we can determine the smallest value of \( i \) such that \( E[\text{num}_i] \leq 1 \), we should be able to break the infinite sum into two sums and derive tight bounds for each of them.

In order to analyze \( E[\text{num}_i] \), we observe that for \( e \in \text{Seq}^n \), \( \text{num}_i(e) \) is a count of the number of components whose lengths are at least \( i \). Furthermore, we can express the fact that a component has a length of at least \( i \) as an event in \( \text{Seq} \). The standard technique for counting events is to use an indicator random variable. Specifically, consider the event in \( \text{Seq} \) that \( \text{len} \geq i \); i.e., this event is the set of sequences of coin flips consisting of at least \( i - 1 \) heads, followed by exactly one tails. The indicator for this event is then defined to be

\[ I(\text{len} \geq i)(e_j) = \begin{cases} 1 & \text{if } \text{len}(e_j) \geq i \\ 0 & \text{otherwise.} \end{cases} \]
We can then express \(num_i\) as follows:

\[
num_i(⟨e_1, \ldots, e_n⟩) = \sum_{j=1}^{n} I(\text{len} \geq i)(e_j).
\]

The utility of indicator random variables is shown by the following theorem, whose proof follows immediately from Theorem 5.5.

**Theorem 6.9** Let \(e\) be any event in a discrete probability space. Then \(E[I(e)] = P(e)\).

Applying the above theorem, we obtain

\[
E[num_i] = E\left[\sum_{j=1}^{n} I(\text{len} \geq i)\right] = E[nI(\text{len} \geq i)] = nE[I(\text{len} \geq i)] = nP(\text{len} \geq i) = n2^{1-i}.
\]

Clearly, \(E[num_i] > 1\) iff \(i < 1 + \lg n\). This suggests that \(\text{maxLevel}\) should typically be about \(\lg n\) (however, this is not a proof of the expected value of \(\text{maxLevel}\)). Because we already know that the while loop is expected to iterate no more than once for each level, this suggests that the overall running time is logarithmic in \(n\) (assuming \(l\) is sufficiently small). While we don’t have quite enough yet to show this, we can now show a logarithmic bound on the first term in (6.9):

\[
\sum_{i=1}^{\infty} E[\text{tail}_i] \leq \sum_{i=1}^{\left\lceil \lg n \right\rceil} (\min(1, E[num_i])) = \sum_{i=1}^{\left\lceil \lg n \right\rceil} 1 + \sum_{i=\left\lceil \lg n \right\rceil + 1}^{\infty} n2^{1-i} = \left\lceil \lg n \right\rceil + n \sum_{i=0}^{\infty} 2^{-\left\lceil \lg n \right\rceil - i} = \left\lceil \lg n \right\rceil + n2^{1-\left\lceil \lg n \right\rceil} \leq \left\lceil \lg n \right\rceil + n2^{1-\lg n} = \left\lceil \lg n \right\rceil + 2.
\]

(6.11)
CHAPTER 6. STORAGE/RETRIEVAL I: ORDERED KEYS

In order to complete the analysis of FIND, we must consider the second term in (6.9), namely, \( E[\max(\text{maxLevel}, l)] \). It would be nice if we could use a property like the linearity of expectation to conclude that this value is equal to \( \max(E[\text{maxLevel}], E[l]) \); however, such a property does not necessarily hold (see Exercise 6.16). On the other hand, because \( \text{maxLevel} \) and \( l \) are nonnegative, we can use the fact that \( \max(\text{maxLevel}, l) \leq \text{maxLevel} + l \).

Therefore,

\[
E[\max(\text{maxLevel}, l)] \leq E[\text{maxLevel} + l]
= E[\text{maxLevel}] + E[l]. \tag*{(6.12)}
\]

For the case in which FIND is called from PUT, we know that \( E[l] = 2 \). We therefore need to evaluate \( E[\text{maxLevel}] \). Note that \( \text{maxLevel} \) is the number of nonempty levels. We can therefore use another indicator random variable to express \( \text{maxLevel} \) — specifically,

\[
\text{maxLevel} = \sum_{i=1}^{\infty} I(\text{num}_i > 0).
\]

We therefore have

\[
E[\text{maxLevel}] = E\left[ \sum_{i=1}^{\infty} I(\text{num}_i > 0) \right]
= \sum_{i=1}^{\infty} E[I(\text{num}_i > 0)]. \tag*{(6.13)}
\]

Clearly, \( I(\text{num}_i > 0)(e) \leq 1 \) for all \( e \in \text{Seq}^n \), so that \( E[I(\text{num}_i > 0)] \leq 1 \). Furthermore, \( I(\text{num}_i > 0)(e) \leq \text{num}_i(e) \), so that \( E[I(\text{num}_i > 0)] \leq E[\text{num}_i] \). We therefore have \( E[I(\text{num}_i > 0)] \leq \min(1, E[\text{num}_i]) \), which is the same upper bound we showed for \( E[\text{tail}_i] \). Therefore, following the derivation of (6.11), we have

\[
E[\text{maxLevel}] \leq \lfloor \lg n \rfloor + 2. \tag*{(6.14)}
\]

Now combining (6.9), (6.10), (6.11), (6.12), and (6.14), it follows that the expected number of tests of the while loop condition is no more than

\[
2(\lfloor \lg n \rfloor + 2) + 2 \in O(\lg n)
\]

for a worst-case input when FIND is called by PUT. The expected running time of FIND in this context is therefore in \( O(\lg n) \).
A matching lower bound for the expected running time of \texttt{Find} can also be shown — the details are outlined in Exercise 6.18. We can therefore conclude that the expected running time of \texttt{Find} when called from \texttt{Put} on a worst-case input is in $\Theta(lg \, n)$.

We can now complete the analysis of \texttt{Put}. We have shown that the expected running times for both loops and the constructor for \texttt{SkipListNode} are all in $\Theta(1)$. The expected running time of \texttt{Find}(k, l) is in $\Theta(lg \, n)$. The remainder of the algorithm clearly runs in $\Theta(1)$ time. The total time is therefore expected to be in $\Theta(lg \, n)$ for a worst-case input. We leave as exercises to design \texttt{Get} and \texttt{Remove} to run in $\Theta(lg \, n)$ expected time, as well.

Earlier, we suggested that for all practical purposes, fixed-sized arrays could be used for both \texttt{start.elements} and \texttt{end.elements}. We can now justify that claim by observing that

$$P(num_i > 0) = E[I(num_i > 0)] \\
\leq E[num_i] \\
= n2^{1-i}.$$  

Thus, the probability that some element has a level strictly greater than 100 is at most $n2^{-100}$. Because $2^{-20} < 10^{-6}$, this means that for $n \leq 2^{80} \approx 10^{24}$, the probability that a level higher than 100 is reached is less than one in a million. Such a small probability of error can safely be considered negligible.

### 6.5 Summary

A summary of the running times of the operations for the various implementations of \texttt{OrderedDictionary} is given in Figure 6.26. $\Theta(lg \, n)$-time implementations of the \texttt{Get}, \texttt{Put}, and \texttt{Remove} operations for the \texttt{OrderedDictionary} interface can be achieved in three ways:

- A balanced binary search tree, such as an AVL tree, guarantees $\Theta(lg \, n)$ performance in the worst case.

- A splay tree is a binary search tree that guarantees $O(lg \, n)$ amortized performance by rotating the items accessed to the root. This has an additional benefit of leaving frequently accessed items near the root, so that they are accessed more quickly.

- A skip list uses randomization to achieve $\Theta(lg \, n)$ expected performance for worst-case inputs.

Corrected 4/7/11.
Figure 6.26 Running times of the OrderedDictionary operations for various implementations

<table>
<thead>
<tr>
<th></th>
<th>Get</th>
<th>Put</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSTDictionary</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>AVLDictionary</td>
<td>$\Theta(lg\ n)$</td>
<td>$\Theta(lg\ n)$</td>
<td>$\Theta(lg\ n)$</td>
</tr>
<tr>
<td>SplayDictionary</td>
<td>$O(lg\ n)$</td>
<td>$O(lg\ n)$</td>
<td>$O(lg\ n)$</td>
</tr>
<tr>
<td>amortized</td>
<td>amortized</td>
<td>amortized</td>
<td>amortized</td>
</tr>
<tr>
<td>SkipListDictionary</td>
<td>$\Theta(lg\ n)$</td>
<td>$\Theta(lg\ n)$</td>
<td>$\Theta(lg\ n)$</td>
</tr>
<tr>
<td>expected</td>
<td>expected</td>
<td>expected</td>
<td>expected</td>
</tr>
</tbody>
</table>

Notes:

- $n$ is the number of elements in the dictionary.
- The constructor and the Size operation each run in $\Theta(1)$ worst-case time for each implementation.
- The VisitInOrder operation runs in $\Theta(n)$ worst-case time for each implementation, assuming that the Visit operation for the given Visitor runs in $\Theta(1)$ time.
- Unless otherwise noted, all running times are worst-case.

Section 6.4 introduced the use of indicator random variables for analyzing randomized algorithms. The application of this technique involves converting the expected value of a random variable to the expected values of indicator random variables and ultimately to probabilities. Theorems 5.5, 5.9, and 6.9 are useful in performing this conversion. The probabilities are then computed using the probabilities of the elementary events and the laws of probability theory. Because we are only interested in asymptotic bounds, probabilities which are difficult to compute exactly can often be bounded by probabilities that are easier to compute.

6.6 Exercises

Exercise 6.1 Prove the correctness of BSTDictionary.TraverseInOrder, shown in Figure 6.7.
Exercise 6.2 Draw the result of inserting the following keys in the order given into an initially empty binary search tree:

34, 65, 75, 54, 19, 45, 11, 23, 90, 15

Exercise 6.3 Draw the result of deleting each of the following keys from the tree shown in Figure 6.10, assuming that it is an ordinary binary search tree. The deletions are not cumulative; i.e., each deletion operates on the original tree.

a. 55
b. 74
c. 34

Exercise 6.4 Repeat Exercise 6.2 for an AVL tree.

Exercise 6.5 Repeat Exercise 6.3 assuming the tree is an AVL tree.

Exercise 6.6 Repeat Exercise 6.2 for a splay tree.

Exercise 6.7 Repeat Exercise 6.3 assuming the tree is a splay tree.

Exercise 6.8 Complete the implementation of AVLDICTIONARY, shown in Figure 6.13, so that GET, PUT, and REMOVE run in $\Theta(lg n)$ time in the worst case. Prove the correctness and running time of the resulting implementation.

Exercise 6.9 The depth of a node in a tree is its distance from the root; specifically the root has depth 0 and the depth of any other node is 1 plus the depth of its parent. Prove by induction on the height $h$ of any AVL tree that every leaf has depth at least $h/2$.

* Exercise 6.10 Prove that when a node is inserted into an AVL tree, at most one rotation is performed.

** Exercise 6.11 Prove that if $2^n - 1$ keys are inserted into an AVL tree in increasing order, the result is a perfectly balanced tree. [Hint: You will need to describe the shape of the tree after $n$ insertions for arbitrary $n$, and prove this by induction on $n$.]
Exercise 6.12 A red-black tree is a binary search tree whose nodes are colored either red or black such that

- if a node is red, then the roots of its nonempty children are black; and
- from any given node, every path to any empty subtree has the same number of black nodes.

We call the number of black nodes on a path from a node to an empty subtree to be the black-height of that node. In calculating the black-height of a node, we consider that the node itself is on the path to the empty subtree.

a. Prove by induction on the height of a red-black tree that if the black-height of the root is \( b \), then the tree has at least \( 2^b - 1 \) black nodes.

b. Prove that if a red-black tree has height \( h \), then it has at least \( 2^{h/2} - 1 \) nodes.

c. Prove that if a red-black tree has \( n \) nodes, then its height is at most \( 2 \log(n + 1) \).

Exercise 6.13 Give a splay-tree implementation of Put based on SPLAY-DICTIONARY.FIND, shown in Figure 6.16. You do not need to include algorithms for the rotations. Prove its correctness, assuming the rotations are correct.

Exercise 6.14 Exercise removed, 4/7/11.

Exercise 6.15 Prove by induction on \( n \) that the sum of the probabilities of the elementary events in \( \text{Seq}^n \) is 1.

Exercise 6.16 Let \( S = \{\text{heads}, \text{tails}\} \) be the discrete probability space in which \( P(\text{heads}) = P(\text{tails}) = 1/2 \).

a. Using the definition of expected value, compute

\[
E[\max(I(\text{heads}), I(\text{tails}))].
\]

b. Using Theorem 6.9, compute

\[
\max(E[I(\text{heads})], E[I(\text{tails})]).
\]

Your answer should be different from your answer in part a.
Exercise 6.17 Prove that if \( f \) and \( g \) are discrete random variables in a discrete probability space, then

\[
E[\max(f, g)] \geq \max(E[f], E[g]).
\]

*Exercise 6.18* The goal of this exercise is to show a lower bound on the expected running time of `SkipListDictionary.Find`.

a. Prove that \( P(\text{num}_i > 0) = 1 - (1 - 2^{1-i})^n. \) [**Hint:** First compute \( P(\text{num}_i = 0). \)]

b. Prove the *binomial theorem*, namely, for any real \( a, b \) and natural number \( n \),

\[
(a + b)^n = \sum_{j=0}^{n} \binom{n}{j} a^{n-j} b^j,
\]

where

\[
\binom{n}{j} = \frac{n!}{j!(n-j)!}
\]

are the *binomial coefficients* for \( 0 \leq j \leq n. \) [**Hint:** Use induction on \( n. \)]

c. Using the results of parts a and b, prove that for \( i \leq \lg n + 1, \)

\[
P(\text{num}_i > 0) > 1/2.
\]

d. Using the result of part c, Exercise 6.17, and (6.13), prove that

\[
E[\max(\text{maxLevel}, l)] \in \Omega(\lg n),
\]

and hence, the expected running time of `Find` is in \( \Omega(\lg n). \)

Exercise 6.19 Give algorithms for `SkipListDictionary.Get` and `SkipListDictionary.Remove`. Prove that they meet their specifications and run in expected \( \Theta(\lg n) \) time for worst-case input. Note that in both cases, you will need to modify the analysis of `SkipListDictionary.Find` to use the appropriate value for \( E[l] \). You may use the result of Exercise 6.18 for the lower bounds.

Exercise 6.20 Suppose we define a discrete probability space consisting of all ordered pairs of flips of a fair coin. This probability space contains four elementary events, each having probability \( 1/4 \). We define the following three events:
CHAPTER 6. STORAGE/RETRIEVAL I: ORDERED KEYS

• $e_1$: the first flip is heads;
• $e_2$: the second flip is heads; and
• $e_3$: the two flips are different.

Show that the three events are pairwise independent, but not mutually independent.

* Exercise 6.21 Let $\text{len}$ be as defined in Section 6.4. For each of the following, either find the expected value or show that it diverges (i.e., that it is infinite).
  a. $E[2^{\text{len}}]$.
  b. $E[\sqrt{2^{\text{len}}}]$.

Exercise 6.22 Let $A[1..n]$ be a random permutation of the positive integers less than or equal to $n$, such that each permutation is equally likely. Recall from Exercise 3.28 (page 103) that an inversion is a pair of indices $1 \leq i < j \leq n$ such that $A[i] > A[j]$. Determine the expected number of inversions in $A$. [Hint: Use an indicator random variable for the event that $(i, j)$ is an inversion.]

Exercise 6.23 As in the above exercise, let $A[1..n]$ be a random permutation of the positive integers less than or equal to $n$, such that each permutation is equally likely. What is the expected number of indices $i$ such that $A[i] = i$?

6.7 Chapter Notes

AVL trees, which comprise the first balanced binary search tree scheme, were introduced by Adel’son-Vel’skii and Landis [1]. Splay trees were introduced by Sleator and Tarjan [99]. Red-black trees, mentioned in Exercise 6.12, were introduced by Bayer [8] (see also Gubias and Sedgewick [57]). Balance in red-black trees is maintained using the same rotations as for splay trees. As a result, keys can be accessed in $\Theta(\lg n)$ time in the worst case. Because heights don’t need to be calculated, they tend to perform better than AVL trees and are widely used in practice. A somewhat simpler version of red-black trees, known as AA-trees, was introduced by Andersson [5].
All of the above trees can be manipulated by the tree viewer on this textbook’s web site. The implementations of these trees within this package are all immutable.

Another important balanced search tree scheme is the B-tree, introduced by Bayer and McCreight [9]. A B-tree is a data structure designed for accessing keyed data from an external storage device such as a disk drive. B-trees therefore have high branching factor in order to minimize the number of disk accesses needed.

Skip lists were introduced by Pugh [93].