software stack. For the tree traversal, node references are pushed onto the stack as it moves down into the tree and the references are popped as the process backtracks. Listing 14.7 shows the implementation of the iterator using a software stack.

Listing 14.7 An iterator for the binary search tree using a software stack.

```
class _BSTMapIterator :
  def __init__( self, root ):
     # Create a stack for use in traversing the tree.
    self._theStack = Stack()
     # We must traverse down to the node containing the smallest key
     # during which each node along the path is pushed onto the stack.
    self._traverseToMinNode( root )

  def __iter__( self ):
    return self

   # Returns the next item from the BST in key order.
  def __next__( self ):
     # If the stack is empty, we are done.
    if self._theStack.isEmpty() :
      raise StopIteration
    else :
       # The top node on the stack contains the next key.
      node = self._theStack.pop()
      key = node.key
       # If this node has a subtree rooted as the right child, we must
       # find the node in that subtree that contains the smallest key.
       # Again, the nodes along the path are pushed onto the stack.
      if node.right is not None :
        self._traverseToMinNode( node.right )

   # Traverses down the subtree to find the node containing the smallest
   # key during which the nodes along that path are pushed onto the stack.
  def _traverseToMinNode( self, subtree ):
    if subtree is not None :
      self._theStack.push( subtree )
      self._traverseToMinNode( subtree.left )
```

14.3 AVL Trees

The binary search tree provides a convenient structure for storing and searching data collections. The efficiency of the search, insertion, and deletion operations depend on the height of the tree. In the best case, a binary tree of size n has a height of $\log n$, but in the worst case, there is one node per level, resulting in a height of n. Thus, it would be to our advantage to try to build a binary search tree that has height $\log n$.

If we were constructing the tree from the complete set of search keys, this would be easy to accomplish. The keys can be sorted in ascending order and then

using a technique similar to that employed with the linked list version of the merge sort, the interior nodes can be easily identified. But this requires knowing all of the keys up front, which is seldom the case in real applications where keys are routinely being added and removed. We could rebuild the binary search tree each time a new key is added or an existing one is removed. But the time to accomplish this would be extreme in comparison to using a simple brute-force search on one of the sequential list structures. What we need is a way to maintain the optimal tree height in real time, as the entries in the tree change.

The ***AVL tree***, which was invented by G. M. **A**del'son-**V**elskii and Y. M. **L**andis in 1962, improves on the binary search tree by always guaranteeing the tree is height balanced, which allows for more efficient operations. A binary tree is ***balanced*** if the heights of the left and right subtrees of every node differ by at most 1. Figure 14.14 illustrates two examples of AVL trees.

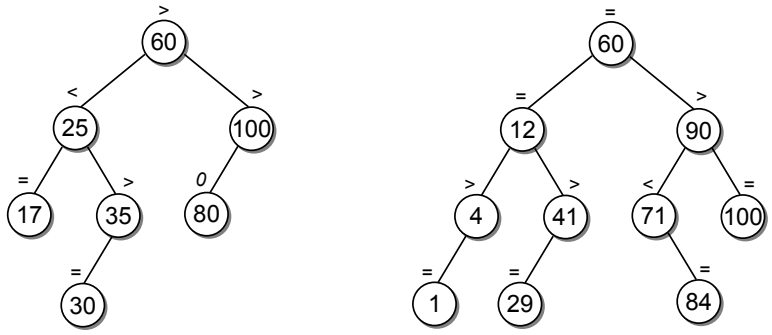


Figure 14.14: Examples of balanced binary search trees.

With each node in an AVL tree, we associate a ***balance factor***, which indicates the height difference between the left and right branch. The balance factor can be one of three states:

left high: When the left subtree is higher than the right subtree.

equal high: When the two subtrees have equal height.

right high: When the right subtree is higher than the left subtree.

The balance factors of the tree nodes in our illustrations are indicated by symbols: > for a left high state, = for the equal high state, and < for a right high state. When a node is out of balance, we will use either << or >> to indicate which subtree is higher.

The search and traversal operations are the same with an AVL tree as with a binary search tree. The insertion and deletion operations have to be modified in order to maintain the balance property of the tree as new keys are inserted and existing ones removed. By maintaining a balanced tree, we ensure its height never exceeds $1.44 \log n$. This height is sufficient for providing $O(\log n)$ time operations even in the worst case.

14.3.1 Insertions

Inserting a key into an AVL tree begins with the same process used with a binary search tree. We search for the new key in the tree and add a new node at the child link where we fall off the tree. When a new key is inserted into an AVL tree, the balance property of the tree must be maintained. If the insertion of the new key causes any of the subtrees to become unbalanced, they will have to be rebalanced.

Some insertions are simpler than others. For example, suppose we want to add key 120 to the sample AVL tree from Figure 14.14(a). Following the insertion operation of the binary search tree, the new key will be inserted as the right child of node 100, as illustrated in Figure 14.15(a). The tree remains balanced since the insertion does not change the height of any subtree, but it does cause a change in the balance factors. After the key is inserted, the balance factors have to be adjusted in order to determine if any subtree is out of balance. There is a limited set of nodes that can be affected when a new key is added. This set is limited to the nodes along the path to the insertion point. Figure 14.15(b) shows the new balance factors after key 120 is added.

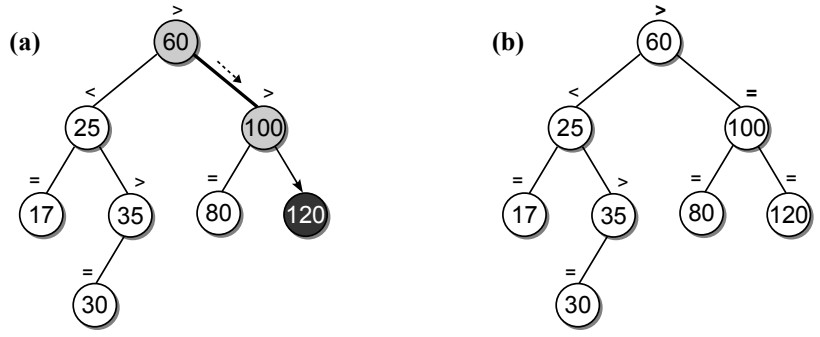


Figure 14.15: A simple insertion into an AVL tree: (a) with key 120 inserted; and (b) the new balance factors.

What happens if we add key 28 to the AVL? The new node is inserted as the left child of node 30, as illustrated in Figure 14.16(a). When the balance factors are recalculated, as in Figure 14.16(b), we can see all of the subtrees along the path that are above node 30 are now out of balance, which violates the AVL balance property. For this example, we can correct the imbalance by rearranging the subtree rooted at node 35, as illustrated in Figure 14.16(c).

Rotations

Multiple subtrees can become unbalanced after inserting a new key, all of which have roots along the insertion path. But only one will have to be rebalanced: the one deepest in the tree and closest to the new node. After inserting the key, the balance factors are adjusted during the unwinding of the recursion. The first

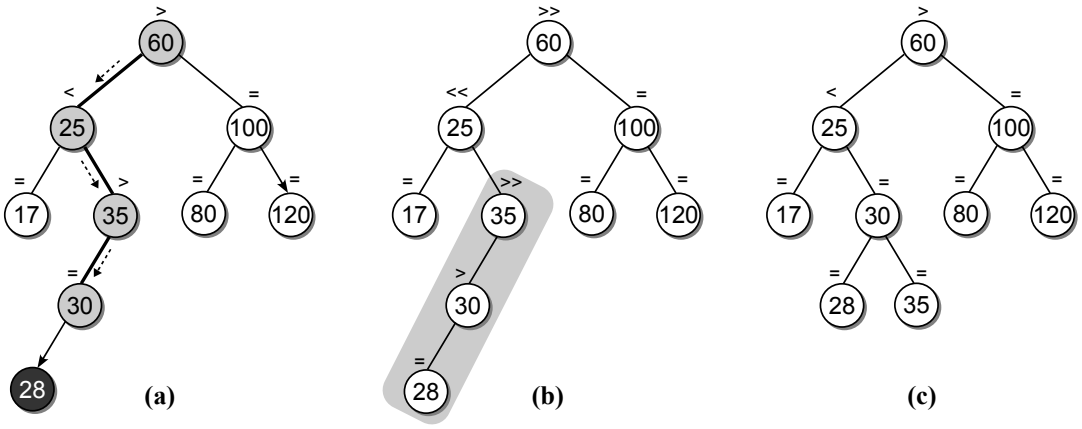


Figure 14.16: An insertion that causes the AVL tree to become unbalanced: (a) the new key is inserted; (b) the balance factors showing an out-of-balance tree; and (c) the subtree after node 35 is rearranged.

subtree encountered that is out of balance has to be rebalanced. The root node of this subtree is known as the ***pivot node***.

An AVL subtree is rebalanced by performing a ***rotation*** around the pivot node. This involves rearranging the links of the pivot node, its children, and possibly one of its grandchildren. The actual modifications depend on which descendant's subtree of the pivot node the new key was inserted into and the balance factors. There are four possible cases:

- *Case 1:* This case, as illustrated in Figure 14.17, occurs when the balance factor of the pivot node (P) is left high before the insertion and the new key is inserted into the left child (C) of the pivot node. To rebalance the subtree, the pivot node has to be rotated right over its left child. The rotation is accomplished by changing the links such that P becomes the right child of C and the right child of C becomes the left child of P.

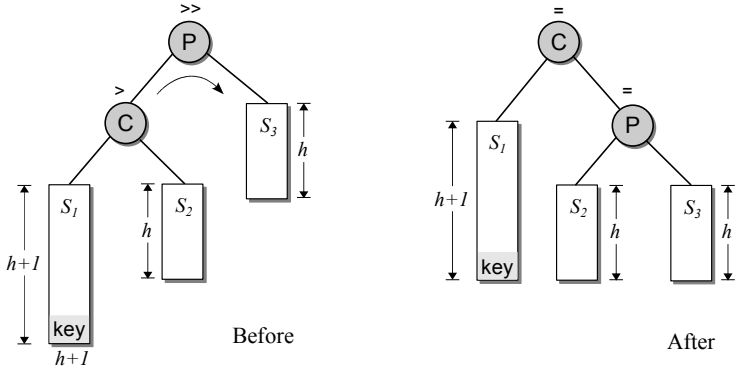


Figure 14.17: Case 1: a right rotation of the pivot node over its left child.

- *Case 2:* This case involves three nodes: the pivot (P), the left child of the pivot (C), and the right child (G) of C. For this case to occur, the balance factor of the pivot is left high before the insertion and the new key is inserted into either the right subtree of C. This case, which is illustrated in Figure 14.18, requires two rotations. Node C has to be rotated left over node V and the pivot node has to be rotated right over its left child. The link modifications required to accomplish this rotation include setting the right child of G as the new left child of the pivot node, changing the left child of G to become the right child of C, and setting C to be the new left child of G.

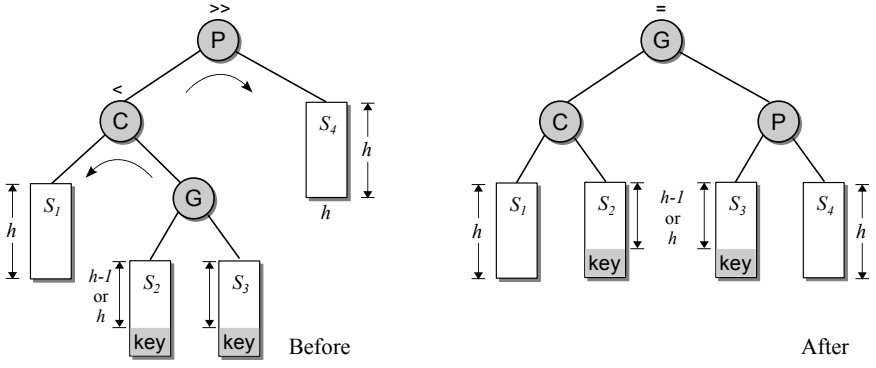


Figure 14.18: Case 2: a double rotation with the pivot's left child rotated left over its right child and the pivot rotated right over its left child.

- *Cases 3 and 4:* The third case is a mirror image of the first case and the fourth case is a mirror image of the second case. The difference is the new key is inserted in the right subtree of the pivot node or a descendant of its right subtree. The two cases are illustrated in Figure 14.19.

New Balance Factors

When a new key is inserted into the tree, the balance factors of the nodes along the path from the root to the insertion point may have to be modified to reflect the insertion. The balance factor of a node along the path changes if the subtree into which the new node was inserted grows taller. The new balance factor of a node depends on its current balance factor and the subtree into which the new node was inserted. The resulting balance factors are provided here:

current factor	*left subtree*	*right subtree*
>	>>	=
=	>	<
<	=	<<

Modifications to the balance factors are made in reverse order as the recursion unwinds. When a node has a left high balance and the new node is inserted into

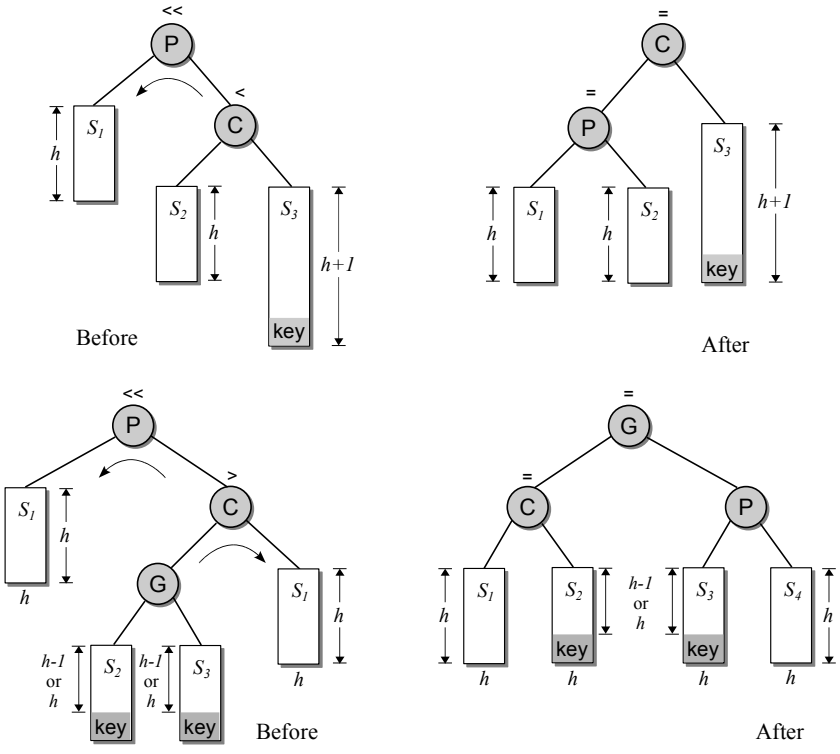


Figure 14.19: Cases 3 (top) and 4 (bottom) are mirror images of cases 1 and 2.

its left child or it has a right high balance and the new node is inserted into its right child, the node is out of balance and its subtree has to be rebalanced. After rebalancing, the subtree will shrink by one level, which results in the balance factors of its ancestors remaining the same. The balance factors of the ancestors will also remain the same when the balance factor changes to equal high.

After a rotation is performed, the balance factor of the impacted nodes have to be changed to reflect the new node heights. The changes required depend on which of the four cases triggered the rotation. The balance factor settings in cases 2 and 4 depend on the balance factor of the original pivot nodes grandchild (the right child of node L or the left child of node R). The new balance factors for the nodes involved in a rotation are provided in Table 14.2.

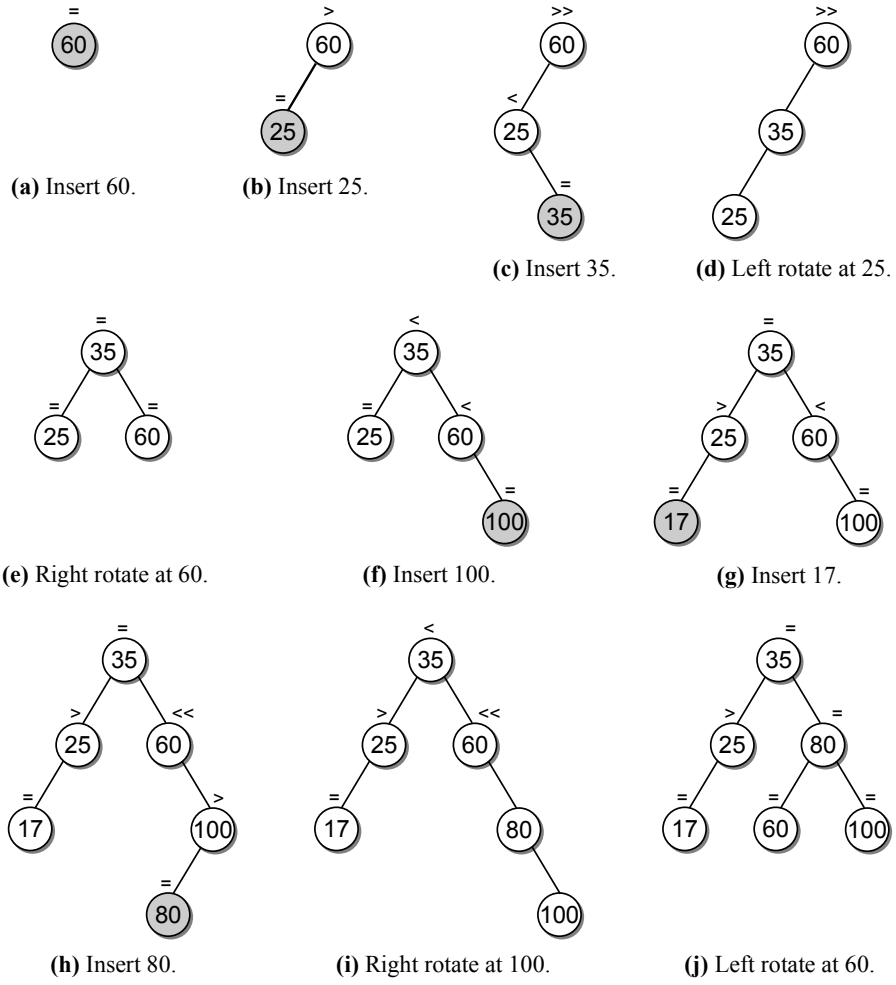
Figure 14.20 illustrates the construction of an AVL tree by inserting the keys from the list `[60, 25, 35, 100, 17, 80]`, one key at a time. Each tree in the figure shows the results after performing the indicate operation. Two double rotations are required to construct the tree: one after node 35 is inserted and one after node 80 is inserted.

14.3.2 Deletions

When an entry is removed from an AVL tree, we must ensure the balance property is maintained. As with the insert operation, deletion begins by using the corre-

original G	*new* P	*new* L	*new* R	*new* G	
case 1	·	=	=	·	·
case 2	$>$	$<$	=	·	=
	=	=	=	·	=
	$<$	=	$>$		=
case 3	·	=	·	=	·
case 4	$>$	=	·	=	$<$
	=	=	·	=	=
	$<$	=	·	=	$>$

Table 14.2: The new balance factors for the nodes after a rotation.



(a) Insert 60. **(b)** Insert 25. **(c)** Insert 35. **(d)** Left rotate at 25.

(e) Right rotate at 60. **(f)** Insert 100. **(g)** Insert 17.

(h) Insert 80. **(i)** Right rotate at 100. **(j)** Left rotate at 60.

Figure 14.20: Building an AVL tree from the list of keys `[60, 25, 35, 100, 17, 80]`. Each tree shows the results after performing the indicated operation.

sponding operation from the binary search tree. After removing the targeted entry, subtrees may have to be rebalanced. For example, suppose we want to remove key 17 from the AVL tree in Figure 14.21(a). After removing the leaf node, the subtree rooted at node 25 is out of balance, as shown in Figure 14.21(b). A left rotation has to be performed pivoting on node 25 to correct the imbalance, as shown in Figure 14.21(c).

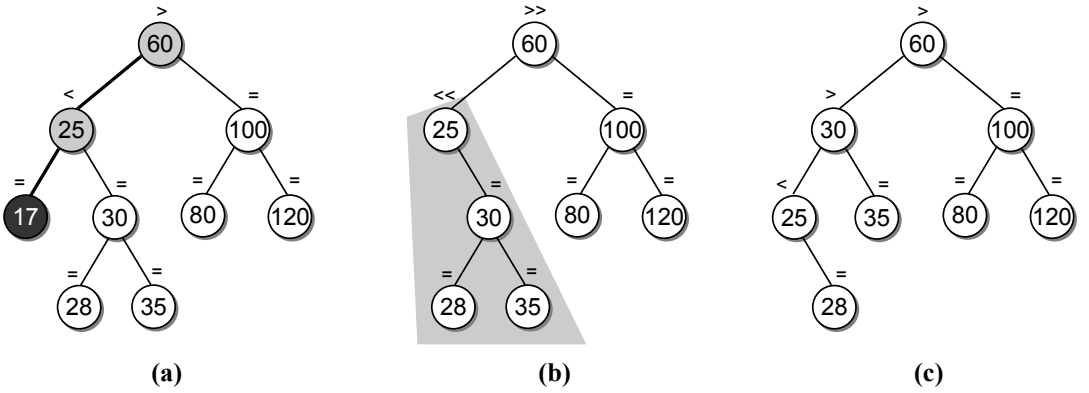


Figure 14.21: A deletion that causes the AVL tree to become unbalanced: (a) the node is located; (b) the balance factors change showing an out-of-balance tree; and (c) the tree after a left rotation.

As with an insertion, the only subtrees that can become unbalanced are those along the path from the root to the original node containing the target. Remember, if the key being removed is in an interior node, its successor is located and copied to the node and the successor's original node is removed. In the insertion operation, at most one subtree can become unbalanced. After the appropriate rotation is performed on the subtree, the balance factors of the node's ancestors do not change. Thus, it restores the ***height-balance property*** both locally at the subtree and globally for the entire tree. This is not the case with a deletion. When a subtree is rebalanced due to a deletion, it can cause the ancestors of the subtree to then become unbalanced. This effect can ripple up all the way to the root node. So, all of the nodes along the path have to be evaluated and rebalanced if necessary.

14.3.3 Implementation

A partial implementation of the Map ADT using a balanced binary search tree is provided in Listing 14.8. The implementation of the non-helper methods is very similar to that of the binary search tree version. Since the traversal and search operations of the AVL tree are identical to those of the binary search tree, the `valueOf()` method can use the `_bstSearch()` helper method and the `__iter__` method can create an instance of the `_BSTMapIterator`. Of course, we assume the implementation of these components are included within the `AVLMap` class. The

Listing 14.8 Partial implementation of the `avltree.py` module.

```
# Constants for the balance factors.
LEFT_HIGH =  1
EQUAL_HIGH = 0
RIGHT_HIGH = -1

# Implementation of the Map ADT using an AVL tree.
class AVLMap :
  def __init__( self ):
    self._root = None
    self._size = 0

  def __len__( self ):
    return self._size

  def __contains__( self, key ):
    return self._bstSearch( self._root, key ) is not None

  def add( self, key, value ):
    node = self._bstSearch( key )
    if node is not None :
      node.value = value
      return False
    else :
      (self._root, tmp) = self._avlInsert( self._root, key, value )
      self._size += 1
      return True

  def valueOf( self, key ):
    node = self._bstSearch( self._root, key )
    assert node is not None, "Invalid map key."
    return node.value

  def remove( self, key ):
    assert key in self, "Invalid map key."
    (self._root, tmp) = self._avlRemove( self._root, key )
    self._size -= 1

  def __iter__( self ):
    return _BSTMapIterator( self._root )

# Storage class for creating the AVL tree node.
class _AVLMapNode :
  def __init__( self, key, value ):
    self.key = key
    self.value = value
    self.bfactor = EQUAL_HIGH
    self.left = None
    self.right = None
```

only change required to the `add()` and `remove()` methods is that each method must call an AVL specific helper method. We provide the implementation for adding a new element to an AVL tree and leave the removal as an exercise.

The nodes in an AVL tree must store their balance factor in addition to the key, data, and two child links. The `_AVLTreeNode` is provided in lines 42–48 of Listing 14.8. We also create and initialize three named constants to represent the three balance factor values. By using named constants, we avoid possible confusion in having to remember what value represents which of the possible balance factors.

The implementation of the insertion operation is divided into several helper methods for a better modular solution. First, we provide helper methods for performing the left and right rotations, as shown in Listing 14.9. A right rotation on a given pivot node is performed using `_avlRotateRight()`. The operation is illustrated in Figure 14.17. The `_avlRotateLeft()` function handles a left rotation, as illustrated in the top of Figure 14.19. Both methods return a reference to the new root node of the subtree after the rotation.

Listing 14.9 Helper functions for performing the AVL tree rotations.

```
1  class AVLMap :
2  # ...
3    # Rotates the pivot to the right around its left child.
4    def _avlRotateRight( self, pivot ):
5      C = pivot.left
6      pivot.left = C.right
7      C.right = pivot
8      return C
9
10   # Rotates the pivot to the left around its right child.
11   def _avlRotateLeft( self, pivot ):
12     C = pivot.right
13     pivot.right = C.left
14     C.left = pivot
15     return C
```

When a subtree becomes unbalanced, we have to determine which of the four possible cases caused the event. Cases 1 and 3 occur when the left subtree of the pivot node is two levels higher than the right subtree, whereas cases 2 and 4 occur when the right subtree becomes two levels higher than the left. We divide the cases into two groups based on which subtree of the pivot node is higher.

The `_avlLeftBalance()` method, provided in Listing 14.10, handles the rotations when the left subtree is higher. To distinguish between the two cases, we have to examine the balance factor of the pivot node's left child. Case one occurs when the left child has a factor of left high (its left child is higher). For this case, the balance factors are adjusted appropriately and a right rotation is performed using the `_avlRightRotate()` method, as shown in lines 9–13. Case 3 occurs when the balance factor of the left child of the pivot node is right high. Note that the case of the pivot's left child having equal balance can never occur, so we do not have to check for this condition. After setting the balance factors, case 3 requires a double rotation, as shown in lines 16–34. We first perform a left rotation on the right child (C) of the pivot node. The root node of the subtree resulting from this rotation becomes the new left child of the pivot node. A right rotation is then performed on

Listing 14.10 Helper functions used to rebalance AVL subtrees.

```
1  class AVLMap :
2  # ...
3    # Rebalance a node when its left subtree is higher.
4    def _avlLeftBalance( self, pivot ):
5       # Set L to point to the left child of the pivot.
6      C = pivot.left
7
8       # See if the rebalancing is due to case 1.
9      if C.bfactor == LEFT_HIGH :
10       pivot.bfactor = EQUAL_HIGH
11       C.bfactor = EQUAL_HIGH
12       pivot = _avlRotateRight( pivot )
13       return pivot
14
15      # Otherwise, a balance from the left is due to case 3.
16     else :
17        # Change the balance factors.
18       if G.bfactor == LEFT_HIGH :
19         pivot.bfactor = RIGHT_HIGH
20         C.bfactor = EQUAL_HIGH
21       elif G.bfactor == EQUAL_HIGH :
22         pivot.bfactor = EQUAL_HIGH
23         C.bfactor = EQUAL_HIGH
24       else : # G.bfactor == RIGHT_HIGH
25         pivot.bfactor = EQUAL_HIGH
26         C.bfactor = LEFT_HIGH
27
28        # All three cases set G's balance factor to equal high.
29       G.bfactor = EQUAL_HIGH
30
31        # Perform the double rotation.
32       pivot.left = _avlRotateLeft( L )
33       pivot = _avlRotateRight( pivot )
34       return pivot
```

the pivot node, resulting in the grandchild (G) of the original pivot node becoming the new root node. The `_avlRightBalance()` method can be implemented in a similar fashion. The actual implementation is left as an exercise.

The insert operation for the binary search tree returned a reference to the existing subtree or the new node, depending on the current invocation of the recursive function. When inserting into an AVL tree, the method must also return a boolean flag indicating if the subtree grew taller. In order to return both values, the `_avlInsert()` function, shown in Listing 14.11, returns a tuple with the first element containing the node reference and the second containing the boolean flag.

Finding the location of the new key and linking its node into the tree uses the same navigation technique as in the binary search tree. The real difference between the insertions into a BST and an AVL tree occurs during the unwinding of the recursion. We have to check to see if the subtree we just visited has grown taller. A taller child subtree means we have to check to see if the current subtree

Listing 14.11 Inserting an entry into an AVL tree.

```
class AVLMap :
# ...
   # Recursive method to handle the insertion into an AVL tree. The
   # function returns a tuple containing a reference to the root of the
   # subtree and a boolean to indicate if the subtree grew taller.
  def _avlInsert( self, subtree, key, newitem ):
     # See if we have found the insertion point.
    if subtree is None :
      subtree = _AVLTreeNode( key, newitem )
      taller = True

     # Is the key already in the tree?
    elif key == subtree.data :
      return (subtree, False)

     # See if we need to navigate to the left.
    elif key < subtree.data :
      (subtree, taller) = _avlInsert( subtree.left, key, newitem )
       # If the subtree grew taller, see if it needs rebalancing.
      if taller :
        if subtree.bfactor == LEFT_HIGH :
          subtree.right = _avlLeftBalance( subtree )
          taller = False
        elif subtree.bfactor == EQUAL_HIGH :
          subtree.bfactor = LEFT_HIGH
          taller = True
        else :     # RIGHT_HIGH
          subtree.bfactor = EQUAL_HIGH
          taller = False

     # Otherwise, navigate to the right.
    else key > subtree.data :
      (node, taller) = _avlInsert( subtree.right, key, newitem )
       # If the subtree grew taller, see if it needs rebalancing.
      if taller :
        if subtree.bfactor == LEFT_HIGH :
          subtree.bfactor = EQUAL_HIGH
          taller = False
        elif subtree.bfactor == EQUAL_HIGH :
          subtree.bfactor = RIGHT_HIGH
          taller = True
        else :    # RIGHT_HIGH
          subtree.right = _avlRightBalance( subtree )
          taller = False

     # Return the results.
    return (subtree, taller)
```

is out of balance and needs to be rebalanced. Regardless if the subtree is out of balance, the balance factor of the current subtree's root node has to be modified as discussed in the previous section. If a subtree did not grow taller, nothing needs to be done.

As the recursion unwinds, the growth status has to be passed back to the parent of each subtree. There are only three circumstances when a subtree grows taller. The first is when a new node is created and linked into the tree. Since the child link in the parent of the new node was originally null, the new node grows from an empty subtree to a subtree of height one. A subtree can also grow taller when its children were originally of equal height and one of the child subtrees has grown taller. In all other instances, the subtree does not grow. Indicating the growth of a subtree is spread throughout the `_avlInsert()` method as appropriate.

14.4 The 2-3 Tree

The binary search tree and the AVL tree are not the only two tree structures that can be used when implementing abstract data types that require fast search operations. The 2-3 tree is a multi-way search tree that can have up to three children. It provides fast operations that are easy to implement. The tree gets its name from the number of keys and children each node can contain. Figure 14.22 provides an abstract view of a simple 2-3 tree.

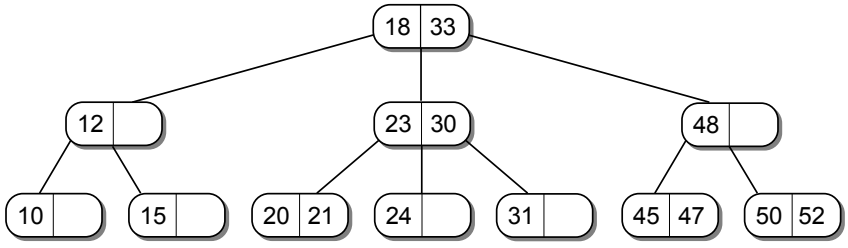


Figure 14.22: A 2-3 tree with integer search keys.

A ***2-3 tree*** is a search tree that is always balanced and whose shape and structure is defined as follows:

- Every node has capacity for one or two keys (and their corresponding payload), which we term key one and key two.
- Every node has capacity for up to three children, which we term the left, middle, and right child.
- All leaf nodes are at the same level.
- Every internal node must contains two or three children. If the node has one key, it must contain two children; if it has two keys, it must contain three children.

In addition, the 2-3 tree has a search property similar to the binary search tree, as illustrated in Figure 14.23. For each interior node, V:

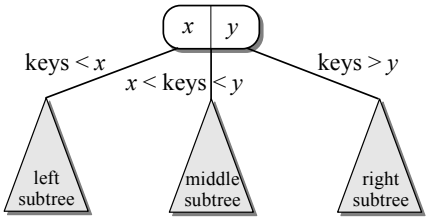


Figure 14.23: A search property of a 2-3 tree.

- All keys less than the first key of node V are stored in the left subtree of V.
- If the node has two children, all keys greater than the first key of node V are stored in the middle subtree of V.
- If the node has three children: (1) all keys greater than the first key of node V but less than the second key are stored in the middle subtree of V; and (2) all keys greater than the second key are stored in the right subtree.

The implementation of 2-3 tree assumes the nodes are constructed from the `_23TreeNode` class as defined in Listing 14.12.

Listing 14.12 Storage class for creating the 2-3 tree nodes.

```
1  class _23TreeNode( object ):
2    def __init__( self, key, data ):
3      self.key1 = key
4      self.key2 = None
5      self.data1 = data
6      self.data2 = None
7      self.left = None
8      self.middle = None
9      self.right = None
10
11     # Is this a leaf node?
12   def isALeaf( self ):
13     return self.left is None and self.middle is None and self.right is None
14
15     # Are there two keys in this node?
16   def isFull( self ):
17     return self.key2 is not None
18
19     # Does the node contain the given target key?
20   def hasKey( self, target ):
21     if (target == self.key1) or
22        (self.key2 is not None and target == self.key2) :
23       return True
24     else :
25       return False
26
```

(Listing Continued)

Listing 14.12 Continued . . .

```
27     # Returns the data associated with the target key or None.
28    def getData( self, target ):
29      if target == self.key1 :
30        return self.data1
31      elif self.key2 is not None and target == self.key2 :
32        return self.data2
33      else :
34        return None
35
36     # Chooses the appropriate branch for the given target.
37    def getBranch( self, target ):
38      if target < self.key1 :
39        return self.left
40      elif self.key2 is None :
41        return self.middle
42      elif target < self.key2 :
43        return self.middle
44      else :
45        return self.right
```

The node class contains seven fields, one for each of the two keys and corresponding data and one for each of the three child links. It also defines three accessor methods that compute information related to the given node. The `isLeaf()` method determines if the node is a leaf, `isFull()` determines if the node contains two keys, `hasKey()` determines if the target key is contained in the node, `getData()` returns the data associated with the given key or `None` if the key is not in the node, and `getBranch()` compares a target key to the nodes key(s) and returns a reference to the appropriate branch that must be followed to find the target. These methods are included to provide meaningful names for those common operations.

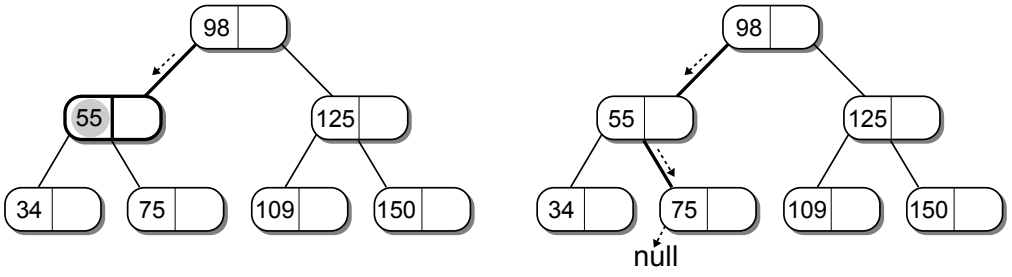
14.4.1 Searching

Searching a 2-3 tree is very similar to that of a binary search tree. We start at the root and follow the appropriate branch based on the value of the target key. The only difference is that we have to compare the target against both keys if the node contains two keys, and we have to choose from among possibly three branches. As in a binary search tree, a successful search will lead to a key in one of the nodes while an unsuccessful search will lead to a null link. That null link will always be in a leaf node. The reason for this is that if an interior node contains one key, it always contains two child links, one for the keys less than its key and one for the keys greater than its key. In a similar fashion, if the node contains two keys, it will always contain three child links that direct us to one of the value ranges: (1) keys less than the node's first key, (2) keys greater than the node's first key but less than its second key, and (3) keys greater than the node's second key. Thus, there is never an opportunity to take a null link from an interior node as there was in a binary

search tree. Figure 14.24 illustrates two searches, one that is successful and one that is not. The search operation for the 2-3 tree is implemented in Listing 14.13.

Listing 14.13 Searching a 2-3 tree.

```
1  class Tree23Map :
2  # ...
3    def _23Search( subtree, target ):
4       # If we encounter a null pointer, the target is not in the tree.
5      if subtree is None :
6        return None
7       # See if the node contains the key. If so, return the data.
8      elif subtree.hashKey( target ) :
9        return subtree.getData( target )
10      # Otherwise, take the appropriate branch.
11     else :
12       branch = subtree.getBranch( target )
13       return _23Search( branch, target )
```



(a) Successful search for key 55. **(b)** Unsuccessful search for key 68.

Figure 14.24: Searching a 2-3 tree.

14.4.2 Insertions

The process of inserting a key into a 2-3 tree is similar to that of a binary search tree, although it's more complicated. The first step is to search for the key as if it were in the tree. As we saw in the previous section, the search for a non-existent key will lead us to a leaf node. The next step is to determine if there is space in the leaf for the new key. If the leaf contains a single key, we can easily insert the key into the node. Consider the partial 2-3 tree illustrated in Figure 14.25 and suppose we want to insert key value 84. In searching for 84, we end up at the node containing value 75. Since there is space in this node, 84 can be added as the node's second key.

But what if the new key is less than the key stored in the leaf node? Suppose we want to add key 26 to the tree, as shown in Figure 14.26. The search leads us to the leaf node containing value 34. When the new key is smaller than the existing key, the new key is inserted as the first key and the existing one is moved to become the second key.

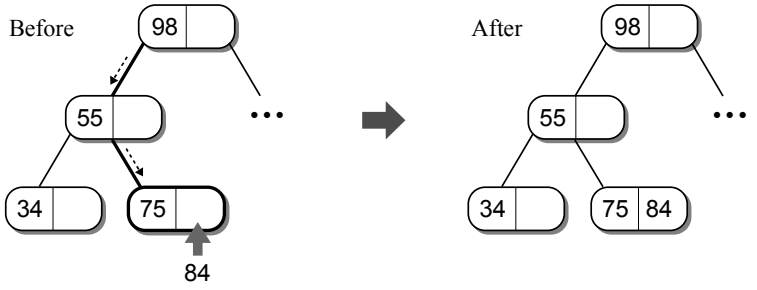


Figure 14.25: Inserting key 84 into a 2-3 tree with space available in the leaf node.

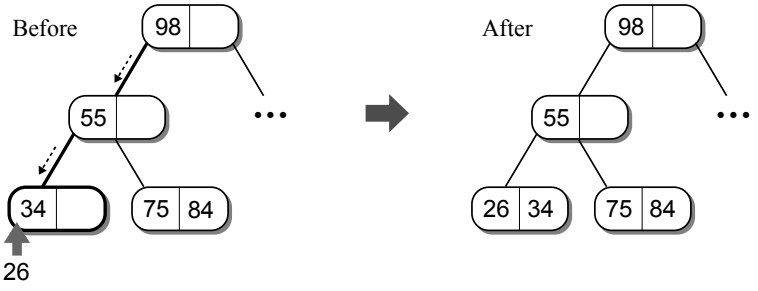


Figure 14.26: Inserting key 26 into a 2-3 tree with space available in the leaf node.

Splitting a Leaf Node

Things become more complicated when the leaf node is full. Suppose we want to insert value 80 into our sample tree. The search for the node leads to the leaf node containing keys 75 and 84, as shown in Figure 14.27. Based on the search property of the 2-3 tree, the new key belongs in this leaf node, but it's full. You might be tempted to create a new leaf node and attach it to the full node as a child. This cannot be done, however, since all leaf nodes must be at the same level and all interior nodes must have at least two children. Instead, the node has to be split, resulting in a new node being created at the same level.

Figure 14.27: Inserting a key into a 2-3 tree with a full leaf node.

The splitting process involves two steps. First, a new node is created, then the new key is compared to the two keys (75 and 84) in the original node. The smallest among the three is inserted into the original node and the largest is inserted into the new node. The middle value is *promoted* to the parent along with a reference to the newly created node. The promoted key and reference are then inserted into the parent node. Figure 14.28 illustrates the three possible cases when splitting a leaf node. k_1 and k_2 are the two keys in the original node and x is the new key that we are trying to insert into the node.

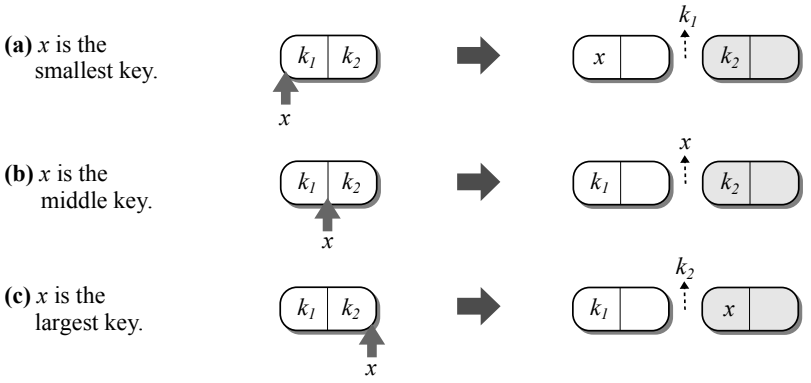


Figure 14.28: Splitting a leaf node into two nodes: each node gets one key and one key is promoted to the parent.

When a key is promoted to the parent, it has to be inserted into the parent's node in a similar fashion to that of a leaf node. The difference is that a reference to the newly created node is also passed up to the parent that has to be inserted into one of the link fields of the parent. Inserting the promoted key and reference into the parent node is simple if the parent contains a single key. The placement of the key and reference depends on which child node was split, as illustrated in Figure 14.29.

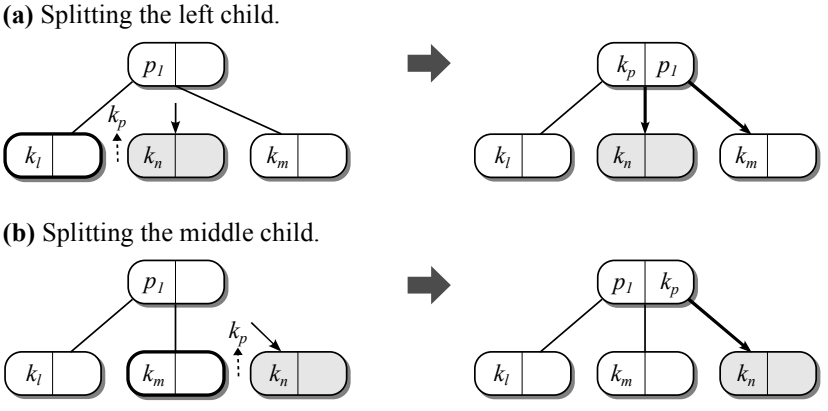


Figure 14.29: Inserting the promoted key and reference into a parent with one key.

There are two cases:

1. *The left child is split*: The existing key p_1 in the parent node becomes the second key and the middle child is moved to become the right child. The promoted key k_p becomes the first key in the parent and the reference to the new node becomes the middle child. Links that have to be modified are shown by directed edges in the figure.
2. *The middle child is split*: The promoted key k_p becomes the second key of the parent and the newly created node becomes the right child.

Splitting a Parent Node

What happens if the node is split and its parent contains three children? For example, suppose we want to insert key 42 into the sample tree shown in Figure 14.30. The node containing keys 26 and 34 has to be split with 34 being promoted to the parent. But the parent also contains two keys (55 and 80). When the parent node is full, it has to be split in a similar fashion as a leaf node, resulting in a key and node reference being promoted to its parent, the grandparent of the child that was split. The splitting process can continue up the tree until either a non-full parent node or the root is located.

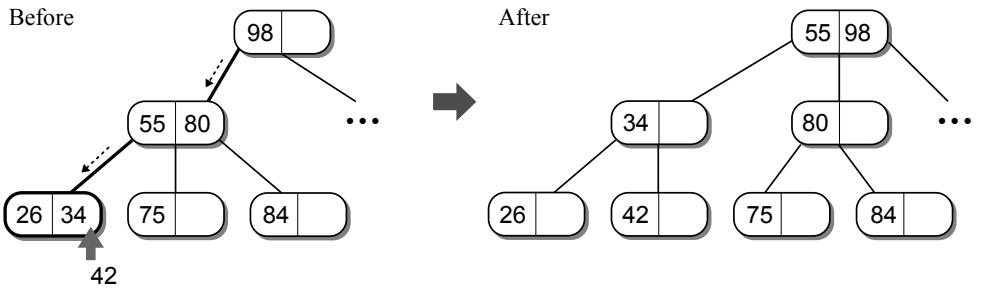


Figure 14.30: A full parent node has to be split to accommodate a promoted key.

When the parent node is split, a new parent node is created and the two will become siblings. Splitting a full interior node is very similar to splitting a leaf node. Two of the three keys, the two in the original parent, p_1 and p_2, and the promoted key, k_p, have to be distributed between the two parents and one has to be promoted to the grandparent. The difference is the connections between the parents and children also have to be changed. The required link modifications depends on which child was split. There are three cases, as illustrated in Figure 14.31. The tree configurations on the left show the nodes and keys before the parent is split and the trees on the right show the resulting configurations. The links that have to be modified are shown with directed edges.

Splitting the Root Node

When the root node has to be split, as illustrated in Figure 14.32, a new root node is created into which the promoted key is stored. The original root becomes the

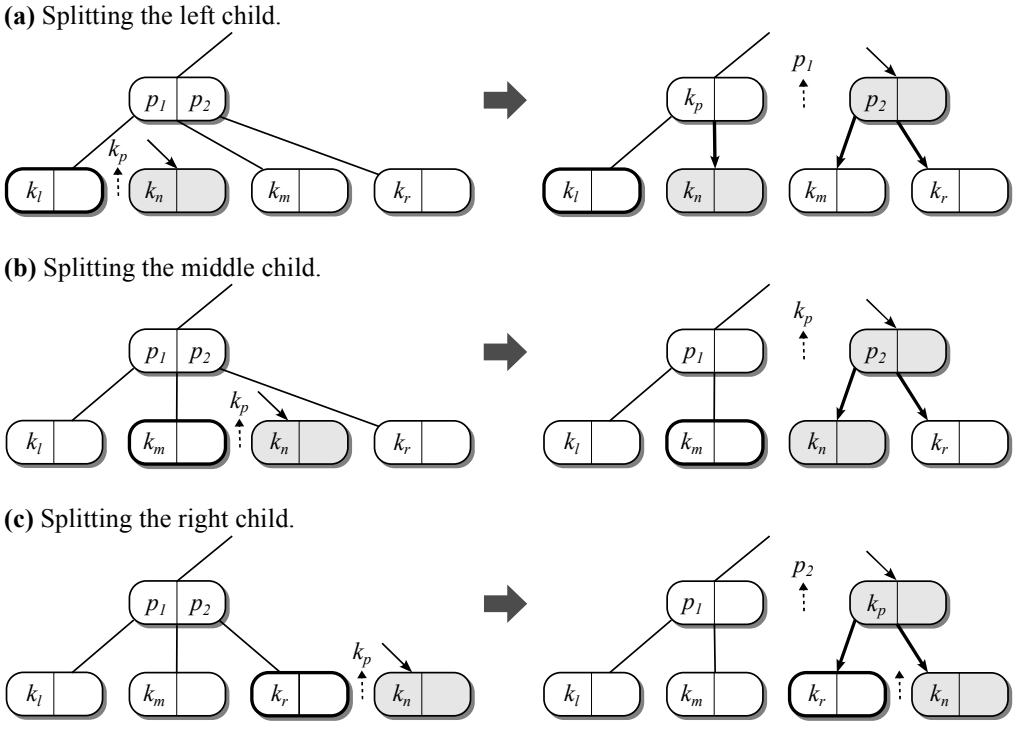


Figure 14.31: Inserting the promoted key and reference into a full parent node.

left child and new child node becomes its middle child. Splitting the root node results in a new level being added to the tree.

Implementation

The 2-3 tree insertion is best implemented recursively. Remember, to insert a new item, not only do we have to navigate down into the tree to find a leaf node, but we may also have to split the nodes along the path as we backtrack to the root node.

The implementation of the 2-3 tree insertion is provided in Listing 14.14. The `_23Insert()` method handles the two special cases involving the root node: the insertion of the first key, resulting in the creation of the first node, and splitting the root node, resulting in a new tree level. If the tree is not initially empty, the

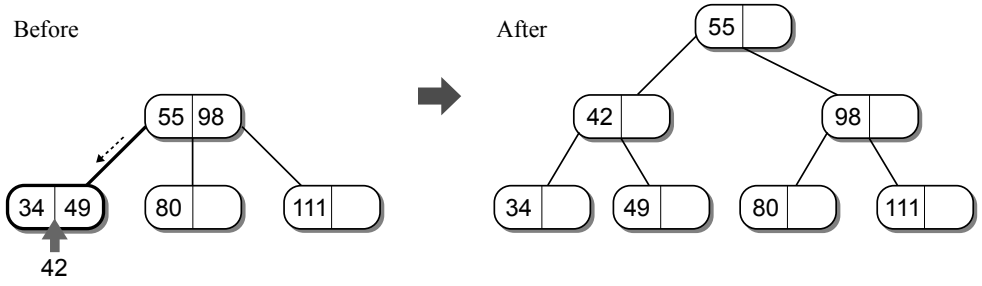


Figure 14.32: Splitting the root node is a special case.

Listing 14.14 Insert a new key into a 2-3 tree.

```
class Tree23Map :
# ...
  def _23Insert( self, key, newitem ):
     # If the tree is empty, a node has to be created for the first key.
    if self._root is None :
      self._root = _23TreeNode( key, newitem )

     # Otherwise, find the correct leaf and insert the key.
    else :
      (pKey, pData, pRef) = _23Insert( self._root, key, newitem )

       # See if the node was split.
      if pKey is not None :
        newRoot = _23TreeNode( pKey, pData )
        newRoot.left = self._root
        newRoot.middle = pRef
        self._root = newRoot

   # Recursive function to insert a new key into the tree.
  def _23RecInsert( subtree, key, newitem ):
     # Make sure the key is not already in the tree.
    if subtree.hasKey( key ) :
      return (None, None, None)

     # Is this a leaf node?
    elif subtree.isALeaf() :
      return _23AddToNode( subtree, key, newitem, None )

     # Otherwise, it's an interior node.
    else :
       # Which branch do we take?
      branch = subtree.getBranch( key )
      (pKey, pData, pRef) = _23Insert( branch, key, newitem )
       # If the child was split, the promoted key and reference have to be
       # added to the interior node.
      if pKey is None :
        return (None, None, None)
      else :
        return _23AddToNode( subtree, pKey, pData, pRef )
```

recursive `_23RecInsert()` method is called to insert the new key. This method navigates the tree to find the leaf node into which the key is to be inserted. During the unwinding of the recursion, the function checks to see if the child node was split, and if it was, adds the promoted key and reference to the current interior node.

The `_23AddToNode()`, provided in Listing 14.15, is used to insert a key into both leaf and interior nodes. When the key is inserted into an interior node, the `key` argument will contain the promoted key and `pRef` will contain the promoted reference. To insert a new key into a leaf node, the `key` argument contains the new key and `pRef` will be `None`. If there is room for the new key, the function arranges the keys and the links in the proper order and null references are returned in a

Listing 14.15 Helper function for inserting a key into a node of the 2-3 tree.

```
# Handles the insertion of a key into a node. If pRef != None, then
# the insertion is into an interior node.
class Tree23Map :
# ...
  def _23AddToNode( self, subtree, key, data, pRef ):
     # If the leaf is full, it has to be split.
    if subtree.isFull() :
      return self._23SplitNode( subtree, key, data,  None )
     # Otherwise, add the new key in its proper order.
    else :
      if key < subtree.key1 :
        subtree.key2 = subtree.key1
        subtree.data2 = subtree.data1
        subtree.key1 = key
        subtree.data1 = data
        if pRef is not None :  # If interior node, set the links.
          subtree.right = subtree.middle
          subtree.middle = pRef
      else :
        subtree.key2 = key
        subtree.data2 = data
        if pRef is not None :  # If interior node, set the links.
          subtree.right = pRef

      return (None, None, None)
```

tuple to indicate the node was not split. Otherwise, the node has to be split by calling `_23SplitNode()` and the resulting tuple is returned to the parent.

The `_23SplitNode()`, provided in Listing 14.16, handles the creation of the new tree node and the distribution of the keys and links to the proper location. The `pRef` argument is again used to indicate if we are working with a leaf node or an interior node. When an interior node is split, the links have to be rearranged in order to maintain the tree property. The three cases that can occur, which depends on the child node into which the key is inserted, are all handled by the function. The promoted key and reference are returned in a tuple for use by the `_23TreeInsert()` function.

14.4.3 Efficiency of the 2-3 Tree

By definition, a 2-3 tree is height balanced with all leaf nodes at the same level. In the worst case, all nodes in the 2-3 tree will contain a single key and all interior nodes will only have two children. From the discussion of the binary search tree, we know such a structure results in a height of $\log n$ for a tree of size n. The traversal operation must visit every node in the 2-3 tree resulting in a worst case time of $O(n)$. The search operation used with 2-3 tree is identical to that of the binary search tree, which we know depends on the height of the tree. Since the maximum height of a 2-3 tree is $\log n$, the search operation will take no more $\log n$ comparisons, resulting in a worst case time of $O(\log n)$.

Listing 14.16 Helper function that splits a full node.

```
# Splits a non-root node and returns a tuple with the promoted key and ref.
class Tree2dMap :
# ...
   # If pRef != None, then an interior node is being split so the new
   # node N created in the function will also be an interior node. In that
   # case, the links of the interior node have to be set appropriately.

  def _23SplitNode( self, node, key, data, pRef ):
     # Create the new node, the reference to which will be promoted.
    newnode = _23TreeNode( None, None )
     # See where the key belongs.
    if key < node.key1 :     # left
      pKey = node.key1
      pData = node.data1
      node.key1 = key
      node.data1 = data
      newnode.key1 = node.key2
      newnode.data1 = node.data2
      if pRef is not None :  # If interior node, set its links.
        newnode.left = node.middle
        newnode.middle = node.right
        node.middle = pRef
    elif key < node.key2 :   # middle
      pKey = key
      pData = data
      newnode.key1 = node.key2
      newnode.data1 = node.data2
      if pRef is not None :  # If interior node, set its links.
        newnode.left = pRef
        newnode.middle = node.right
    else :                   # right
      pKey = node.key2
      pData = node.data2
      newnode.key1 = key
      newnode.data1 = data
      if pRef is not None :  # If interior node, set its links.
        newnode.left = node.right
        newnode.middle = pRef

     # The second key of the original node has to be set to null.
    node.key2 = None
    node.data2 = None
     # Return the promoted key and reference to the new node.
    return (pKey, pData, newnode)
```

The insertion operation, and the deletion which we leave as an exercise, also works very similarly to that of the binary search tree. The search down the tree to find a leaf into which the new key can be inserted takes logarithmic time. If the leaf is full, it has to be split. A node can be split and the keys distributed between the original node, the new node, and the parent node in constant time. In the worst case, a node split is required at each level of the tree during the unwinding

of the recursion. Since the tree can be no higher than $\log n$ and each split is a constant time operation, the worst case time of an insertion is also $O(\log n)$.

Exercises

14.1 Prove or explain why the `_bstRemove()` method requires $O(n)$ time in the worst case.

14.2 Why can new keys not be inserted into the interior nodes of a 2-3 tree?

14.3 Consider the following set of values and use them to build the indicated type of tree by adding one value at a time in the order listed:

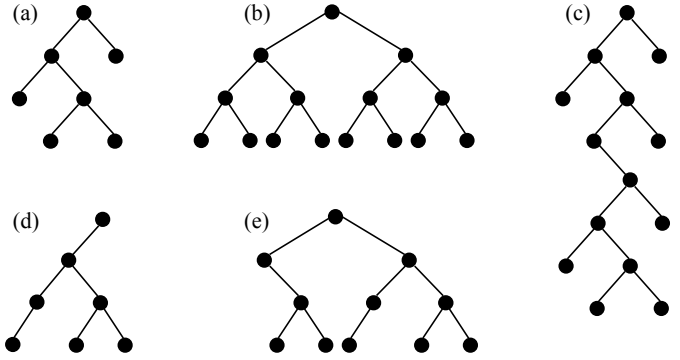
```
30  63  2  89  16  24  19  52  27  9  4  45
```

(a) binary search tree (b) AVL tree (c) 2-3 tree

14.4 Repeat Exercise 14.3, but for the following set of keys:

```
T  I  P  A  F  W  Q  X  E  N  S  B  Z
```

14.5 Given the following binary trees, indicate which trees are height balanced.



14.6 Consider the binary search tree below and show the resulting tree after deleting each of the following keys: 14, 52, and 39.