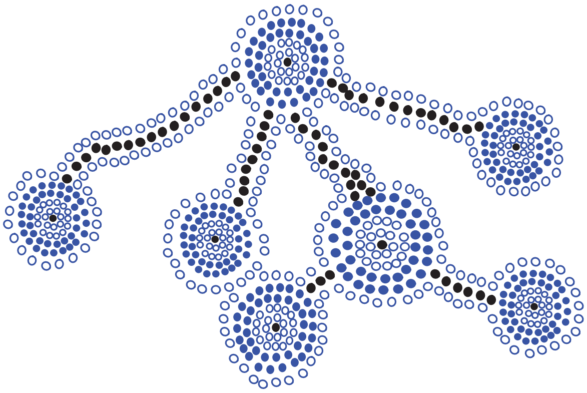
Chapter

10 Search Trees



Contents

10.1 Binary Search Trees

All of the structures that we discuss in this chapter are ***search trees***, that is, tree data structures that can be used to implement ordered maps and ordered dictionaries. Recall from Chapter 9 that a map is a collection of key-value entries, with each value associated with a distinct key. A dictionary differs in that multiple values may share the same key value. Our presentation focuses mostly on maps, but we consider both data structures in this chapter.

We assume that maps and dictionaries provide a special pointer object, called an ***iterator***, which permits us to reference and enumerate the entries of the structure. In order to indicate that an object is not present, there exists a special sentinel iterator called end. By convention, this sentinel refers to an imaginary element that lies just beyond the last element of the structure.

Let M be a map. In addition to the standard container operations (size, empty, begin, and end) the map ADT (Section 9.1) includes the following:

find(k):	If M contains an entry $e = (k, v)$, with key equal to k, then return an iterator p referring to this entry, and otherwise return the special iterator end.
put(k, v):	If M does not have an entry with key equal to k, then add entry (k, v) to M, and otherwise, replace the value field of this entry with v; return an iterator to the inserted/modified entry.
erase(k):	Remove from M the entry with key equal to k; an error condition occurs if M has no such entry.
erase(p):	Remove from M the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.
begin$()$:	Return an iterator to the first entry of M.
end$()$:	Return an iterator to a position just beyond the end of M.

The dictionary ADT (Section 9.5) provides the additional operations insert(k, v), which inserts the entry (k, v), and findAll(k), which returns an iterator range (b, e) of all entries whose key value is k.

Given an iterator p, the associated entry may be accessed using $*p$. The individual key and value can be accessed using p->key$()$ and p->value$()$, respectively. We assume that the key elements are drawn from a total order, which is defined by overloading the C++ relational less-than operator ("<"). Given an iterator p to some entry, it may be advanced to the next entry in this order using the increment operator ("$++p$").

The ordered map and dictionary ADTs also include some additional functions for finding predecessor and successor entries with respect to a given key, but their performance is similar to that of find. So, we focus on find as the primary search operation in this chapter.

Binary Search Trees and Ordered Maps

Binary trees are an excellent data structure for storing the entries of a map, assuming we have an order relation defined on the keys. As mentioned previously (Section 7.3.6), a ***binary search tree*** is a binary tree T such that each internal node v of T stores an entry (k,x) such that:

- Keys stored at nodes in the left subtree of v are less than or equal to k
- Keys stored at nodes in the right subtree of v are greater than or equal to k.

An example of a search tree storing integer keys is shown in Figure 10.1.

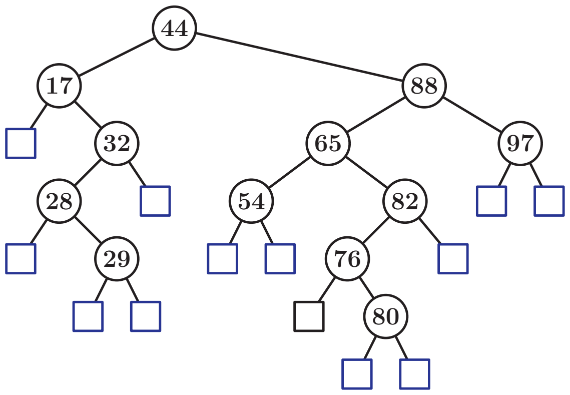


Figure 10.1: A binary search tree T representing a map with integer keys.

As we show below, the keys stored at the nodes of T provide a way of performing a search by making comparisons at a series of internal nodes. The search can stop at the current node v or continue at v's left or right child. Thus, we take the view here that binary search trees are nonempty proper binary trees. That is, we store entries only at the internal nodes of a binary search tree, and the external nodes serve as "placeholders." This approach simplifies several of our search and update algorithms. Incidentally, we could allow for improper binary search trees, which have better space usage, but at the expense of more complicated search and update functions.

Independent of whether we view binary search trees as proper or not, the important property of a binary search tree is the realization of an ordered map (or dictionary). That is, a binary search tree should hierarchically represent an ordering of its keys, using relationships between parent and children. Specifically, an inorder traversal (Section 7.3.6) of the nodes of a binary search tree T should visit the keys in nondecreasing order. Incrementing an iterator through a map visits the entries in this same order.

10.1.1 Searching

To perform operation find(k) in a map M that is represented with a binary search tree T, we view the tree T as a decision tree (recall Figure 7.10). In this case, the question asked at each internal node v of T is whether the search key k is less than, equal to, or greater than the key stored at node v, denoted with key(v). If the answer is "smaller," then the search continues in the left subtree. If the answer is "equal," then the search terminates successfully. If the answer is "greater," then the search continues in the right subtree. Finally, if we reach an external node, then the search terminates unsuccessfully. (See Figure 10.2.)

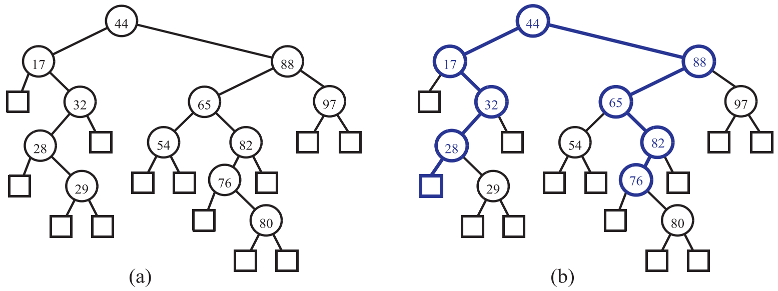


Figure 10.2: (a) A binary search tree T representing a map with integer keys; (b) nodes of T visited when executing operations find(76) (successful) and find(25) (unsuccessful) on M. For simplicity, we show only the keys of the entries.

We describe this approach in detail in Code Fragment 10.1. Given a search key k and a node v of T, this function, TreeSearch, returns a node (position) w of the subtree $T(v)$ of T rooted at v, such that one of the following occurs:

- w is an internal node and w's entry has key equal to k
- w is an external node representing k's proper place in an inorder traversal of $T(v)$, but k is not a key contained in $T(v)$

Thus, function find(k) can be performed by calling TreeSearch$(k, T.\text{root}())$. Let w be the node of T returned by this call. If w is an internal node, then we return w's entry; otherwise, we return *null*.

```
Algorithm TreeSearch($k, v$):
    if $T$.isExternal($v$) then
        return $v$
    if $k <$ key($v$) then
        return TreeSearch($k, T$.left($v$))
    else if $k >$ key($v$) then
        return TreeSearch($k, T$.right($v$))
    return $v$            {we know $k =$ key($v$)}
```

Code Fragment 10.1: Recursive search in a binary search tree.

Analysis of Binary Tree Searching

The analysis of the worst-case running time of searching in a binary search tree T is simple. Algorithm TreeSearch is recursive and executes a constant number of primitive operations for each recursive call. Each recursive call of TreeSearch is made on a child of the previous node. That is, TreeSearch is called on the nodes of a path of T that starts at the root and goes down one level at a time. Thus, the number of such nodes is bounded by $h+1$, where h is the height of T. In other words, since we spend $O(1)$ time per node encountered in the search, function find on map M runs in $O(h)$ time, where h is the height of the binary search tree T used to implement M. (See Figure 10.3.)

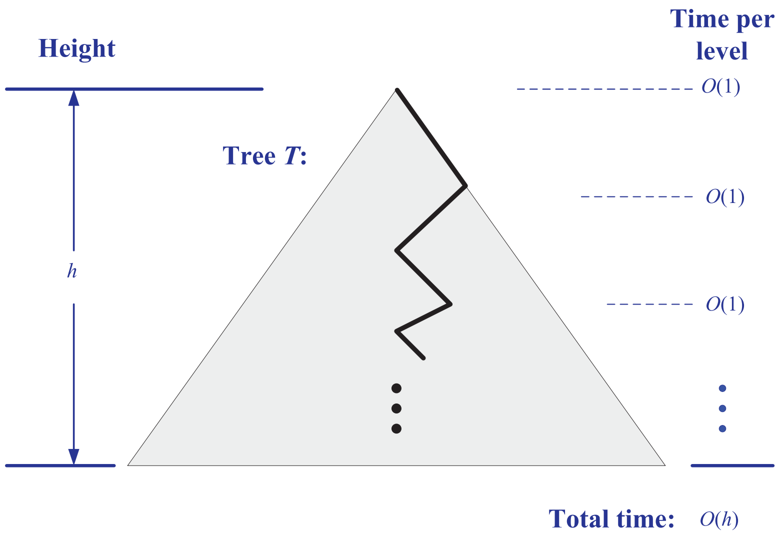


Figure 10.3: The running time of searching in a binary search tree. We use a standard visualization shortcut of viewing a binary search tree as a big triangle and a path from the root as a zig-zag line.

We can also show that a variation of the above algorithm performs operation findAll(k) of the dictionary ADT in time $O(h+s)$, where s is the number of entries returned. However, this function is slightly more complicated, and the details are left as an exercise (Exercise C-10.2).

Admittedly, the height h of T can be as large as the number of entries, n, but we expect that it is usually much smaller. Indeed, we show how to maintain an upper bound of $O(\log n)$ on the height of a search tree T in Section 10.2. Before we describe such a scheme, however, let us describe implementations for map update functions.

10.1.2 Update Operations

Binary search trees allow implementations of the insert and erase operations using algorithms that are fairly straightforward, but not trivial.

Insertion

Let us assume a proper binary tree T supports the following update operation:

insertAtExternal(v, e): Insert the element e at the external node v, and expand v to be internal, having new (empty) external node children; an error occurs if v is an internal node.

Given this function, we perform insert(k, x) for a dictionary implemented with a binary search tree T by calling TreeInsert$(k, x, T$.root$())$, which is given in Code Fragment 10.2.

```
**Algorithm** TreeInsert$(k,x,v)$:
    *Input:* A search key $k$, an associated value, $x$, and a node $v$ of $T$
    *Output:* A new node $w$ in the subtree $T(v)$ that stores the entry $(k,x)$
    $w \leftarrow$ TreeSearch$(k,v)$
    **if** $T$.isInternal$(w)$ **then**
        **return** TreeInsert$(k,x,T$.left$(w))$ {going to the right would be correct too}
    $T$.insertAtExternal$(w,(k,x))$          {this is an appropriate place to put $(k,x)$}
    **return** $w$
```

Code Fragment 10.2: Recursive algorithm for insertion in a binary search tree.

This algorithm traces a path from T's root to an external node, which is expanded into a new internal node accommodating the new entry. An example of insertion into a binary search tree is shown in Figure 10.4.

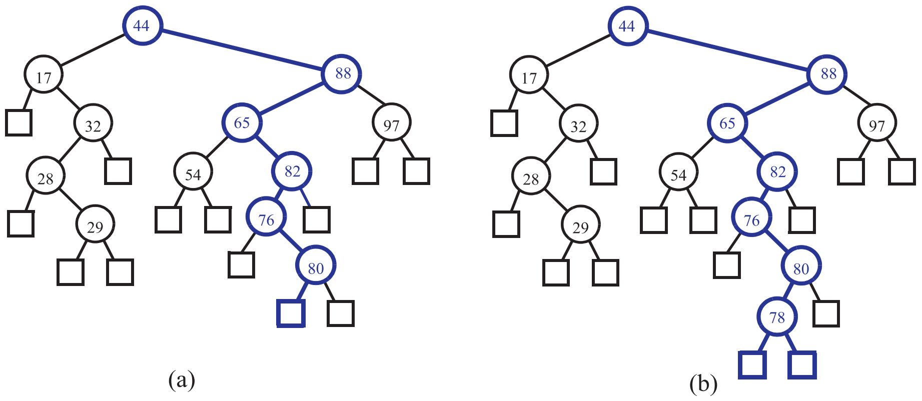


Figure 10.4: Insertion of an entry with key 78 into the search tree of Figure 10.1: (a) finding the position to insert; (b) the resulting tree.

Removal

The implementation of the erase(k) operation on a map M implemented with a binary search tree T is a bit more complex, since we do not wish to create any "holes" in the tree T. We assume, in this case, that a proper binary tree supports the following additional update operation:

removeAboveExternal(v): Remove an external node v and its parent, replacing v's parent with v's sibling; an error occurs if v is not external.

Given this operation, we begin our implementation of operation erase(k) of the map ADT by calling TreeSearch(k, T.root()) on T to find a node of T storing an entry with key equal to k. If TreeSearch returns an external node, then there is no entry with key k in map M, and an error condition is signaled. If, instead, TreeSearch returns an internal node w, then w stores an entry we wish to remove, and we distinguish two cases:

- If one of the children of node w is an external node, say node z, we simply remove w and z from T by means of operation removeAboveExternal(z) on T. This operation restructures T by replacing w with the sibling of z, removing both w and z from T. (See Figure 10.5.)
- If both children of node w are internal nodes, we cannot simply remove the node w from T, since this would create a "hole" in T. Instead, we proceed as follows (see Figure 10.6):
  - We find the first internal node y that follows w in an inorder traversal of T. Node y is the left-most internal node in the right subtree of w, and is found by going first to the right child of w and then down T from there, following the left children. Also, the left child x of y is the external node that immediately follows node w in the inorder traversal of T.
  - We move the entry of y into w. This action has the effect of removing the former entry stored at w.
  - We remove nodes x and y from T by calling removeAboveExternal(x) on T. This action replaces y with x's sibling, and removes both x and y from T.

As with searching and insertion, this removal algorithm traverses a path from the root to an external node, possibly moving an entry between two nodes of this path, and then performs a removeAboveExternal operation at that external node.

The position-based variant of removal is the same, except that we can skip the initial step of invoking TreeSearch(k, T.root()) to locate the node containing the key.

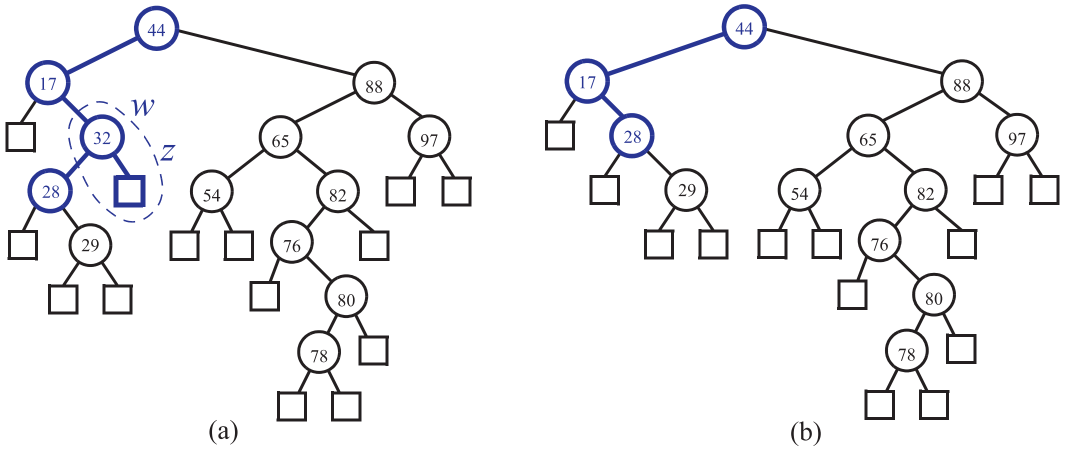


Figure 10.5: Removal from the binary search tree of Figure 10.4b, where the entry to remove (with key 32) is stored at a node (w) with an external child: (a) before the removal; (b) after the removal.

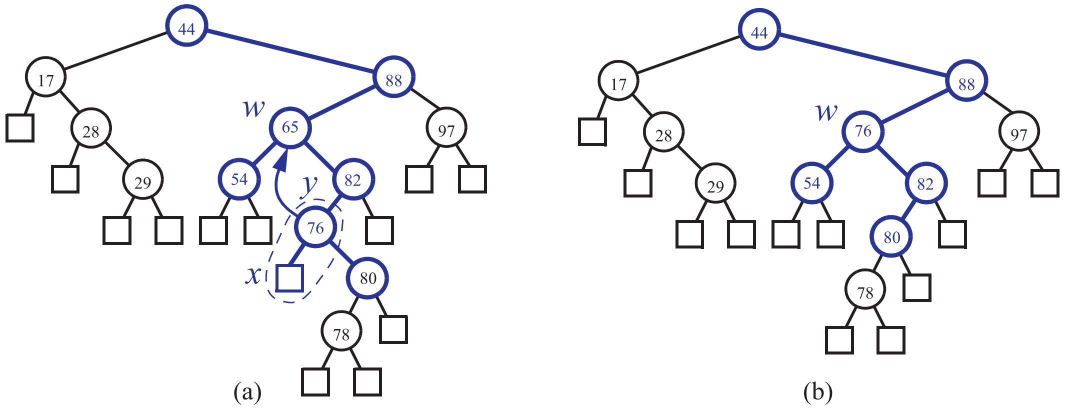


Figure 10.6: Removal from the binary search tree of Figure 10.4b, where the entry to remove (with key 65) is stored at a node (w) whose children are both internal: (a) before the removal; (b) after the removal.

Performance of a Binary Search Tree

The analysis of the search, insertion, and removal algorithms are similar. We spend $O(1)$ time at each node visited, and, in the worst case, the number of nodes visited is proportional to the height h of T. Thus, in a map M implemented with a binary search tree T, the find, insert, and erase functions run in $O(h)$ time, where h is the height of T. Thus, a binary search tree T is an efficient implementation of a map with n entries only if the height of T is small. In the best case, T has height $h = \lceil \log(n+1) \rceil$, which yields logarithmic-time performance for all the map operations. In the worst case, however, T has height n, in which case it would look and feel like an ordered list implementation of a map. Such a worst-case configuration arises, for example, if we insert a series of entries with keys in increasing or decreasing order. (See Figure 10.7.)

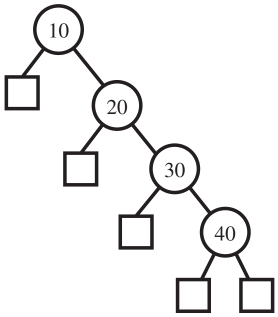


Figure 10.7: Example of a binary search tree with linear height, obtained by inserting entries with keys in increasing order.

The performance of a map implemented with a binary search tree is summarized in the following proposition and in Table 10.1.

Proposition 10.1: *A binary search tree T with height h for n key-value entries uses $O(n)$ space and executes the map ADT operations with the following running times. Operations* size *and* empty *each take $O(1)$ time. Operations* find, insert, *and* erase *each take $O(h)$ time.*

Operation	***Time***
size, empty	$O(1)$
find, insert, erase	$O(h)$

Table 10.1: Running times of the main functions of a map realized by a binary search tree. We denote the current height of the tree with h. The space usage is $O(n)$, where n is the number of entries stored in the map.

Note that the running time of search and update operations in a binary search tree varies dramatically depending on the tree's height. We can nevertheless take comfort that, on average, a binary search tree with n keys generated from a random series of insertions and removals of keys has expected height $O(\log n)$. Such a statement requires careful mathematical language to precisely define what we mean by a random series of insertions and removals, and sophisticated probability theory to prove; hence, its justification is beyond the scope of this book. Nevertheless, keep in mind the poor worst-case performance and take care in using standard binary search trees in applications where updates are not random. There are, after all, applications where it is essential to have a map with fast worst-case search and update times. The data structures presented in the next sections address this need.

10.1.3 C++ Implementation of a Binary Search Tree

In this section, we present a C++ implementation of the dictionary ADT based on a binary search tree, which we call SearchTree. Recall that a dictionary differs from a map in that it allows multiple copies of the same key to be inserted. For simplicity, we have not implemented the findAll function.

To keep the number of template parameters small, rather than templating our class on the key and value types, we have chosen instead to template our binary search tree on just the entry type denoted E. To obtain access to the key and value types, we assume that the entry class defines two public types defining them. Given an entry object of type E, we may access these types E::Key and E::Value. Otherwise, our entry class is essentially the same as the entry class given in Code Fragment 9.1. It is presented in Code Fragment 10.3.

```
template <typename K, typename V>
class Entry {                                        // a (key, value) pair
public:                                              // public types
  typedef K Key;                                     // key type
  typedef V Value;                                   // value type
public:                                              // public functions
  Entry(const K& k = K(), const V& v = V())          // constructor
    : _key(k), _value(v) { }
  const K& key() const { return _key; }              // get key (read only)
  const V& value() const { return _value; }          // get value (read only)
  void setKey(const K& k) { _key = k; }              // set key
  void setValue(const V& v) { _value = v; }          // set value
private:                                             // private data
  K _key;                                            // key
  V _value;                                          // value
};
```

Code Fragment 10.3: A C++ class for a key-value entry.

In Code Fragment 10.4, we present the main parts of the class definition for our binary search tree. We begin by defining the publicly accessible types for the entry, key, value, and the class iterator. This is followed by a declaration of the public member functions. We define two local types, BinaryTree and TPos, which represent a binary search tree and position within this binary tree, respectively. We also declare a number of local utility functions to help in finding, inserting, and erasing entries. The member data consists of a binary tree and the number of entries in the tree.

```
template <typename E>
class SearchTree {                                         // a binary search tree
public:                                                    // public types
  typedef typename E::Key K;                               // a key
  typedef typename E::Value V;                             // a value
  class Iterator;                                          // an iterator/position
public:                                                    // public functions
  SearchTree();                                            // constructor
  int size() const;                                        // number of entries
  bool empty() const;                                      // is the tree empty?
  Iterator find(const K& k);                               // find entry with key k
  Iterator insert(const K& k, const V& x);                 // insert (k,x)
  void erase(const K& k) throw(NonexistentElement); // remove key k entry
  void erase(const Iterator& p);                           // remove entry at p
  Iterator begin();                                        // iterator to first entry
  Iterator end();                                          // iterator to end entry
protected:                                                 // local utilities
  typedef BinaryTree<E> BinaryTree;                        // linked binary tree
  typedef typename BinaryTree::Position TPos;              // position in the tree
  TPos root() const;                                       // get virtual root
  TPos finder(const K& k, const TPos& v);                  // find utility
  TPos inserter(const K& k, const V& x);                   // insert utility
  TPos eraser(TPos& v);                                    // erase utility
  TPos restructure(const TPos& v)                          // restructure
      throw(BoundaryViolation);
private:                                                   // member data
  BinaryTree T;                                            // the binary tree
  int n;                                                   // number of entries
public:
  // ...insert Iterator class declaration here
};
```

Code Fragment 10.4: Class SearchTree, which implements a binary search tree.

We have omitted the definition of the iterator class for our binary search tree. This is presented in Code Fragment 10.5. An iterator consists of a single position in the tree. We overload the dereferencing operator ("*") to provide both read-only and read-write access to the node referenced by the iterator. We also provide an operator for checking the equality of two iterators. This is useful for checking whether an iterator is equal to end.

```
class Iterator {                                    // an iterator/position
private:
  TPos v;                                           // which entry
public:
  Iterator(const TPos& vv) : v(vv) { }              // constructor
  const E& operator*() const { return *v; }         // get entry (read only)
  E& operator*() { return *v; }                     // get entry (read/write)
  bool operator==(const Iterator& p) const          // are iterators equal?
    { return v == p.v; }
  Iterator& operator++();                           // inorder successor
  friend class SearchTree;                          // give search tree access
};
```

Code Fragment 10.5: Declaration of the Iterator class, which is part of SearchTree.

Code Fragment 10.6 presents the definition of the iterator's increment operator, which advances the iterator from a given position of the tree to its inorder successor. Only internal nodes are visited, since external nodes do not contain entries. If the node *v* has a right child, the inorder successor is the leftmost internal node of its right subtree. Otherwise, *v* must be the largest key in the left subtree of some node *w*. To find *w*, we walk up the tree through successive ancestors. As long as we are the right child of the current ancestor, we continue to move upwards. When this is no longer true, the parent is the desired node *w*. Note that we employ the condensed function notation, which we introduced in Section 9.2.7, where the messy scoping qualifiers involving SearchTree have been omitted.

```
/* SearchTree⟨E⟩ :: */                              // inorder successor
  Iterator& Iterator::operator++() {
    TPos w = v.right();
    if (w.isInternal()) {                           // have right subtree?
      do { v = w; w = w.left(); }                   // move down left chain
      while (w.isInternal());
    }
    else {
      w = v.parent();                               // get parent
      while (v == w.right())                        // move up right chain
        { v = w; w = w.parent(); }
      v = w;                                        // and first link to left
    }
    return *this;
  }
```

Code Fragment 10.6: The increment operator ("++") for Iterator.

The implementation of the increment operator appears to contain an obvious bug. If the iterator points to the rightmost node of the entire tree, then the above function would loop until arriving at the root, which has no parent. The rightmost node of the tree has no successor, so the iterator *should* return the value end.

There is a simple and elegant way to achieve the desired behavior. We add a special sentinel node to our tree, called the ***super root***, which is created when the initial tree is constructed. The root of the binary search tree, which we call the ***virtual root***, is made the left child of the super root. We define end to be an iterator that returns the position of the super root. Observe that, if we attempt to increment an iterator that points to the rightmost node of the tree, the function given in Code Fragment 10.6 moves up the right chain until reaching the virtual root, and then stops at its parent, the super root, since the virtual root is its left child. Therefore, it returns an iterator pointing to the super root, which is equivalent to end. This is exactly the behavior we desire.

To implement this strategy, we define the constructor to create the super root. We also define a function root, which returns the virtual root's position, that is, the left child of the super root. These functions are given in Code Fragment 10.7.

```
/* SearchTree⟨E⟩ :: */                               // constructor
  SearchTree() : T(), n(0)
    { T.addRoot(); T.expandExternal(T.root()); }      // create the super root

/* SearchTree⟨E⟩ :: */                               // get virtual root
  TPos root() const
    { return T.root().left(); }                       // left child of super root
```

Code Fragment 10.7: The constructor and the utility function root. The constructor creates the super root, and root returns the virtual root of the binary search tree.

Next, in Code Fragment 10.8, we define the functions begin and end. The function begin returns the first node according to an inorder traversal, which is the leftmost internal node. The function end returns the position of the super root.

```
/* SearchTree⟨E⟩ :: */                               // iterator to first entry
  Iterator begin() {
    TPos v = root();                                  // start at virtual root
    while (v.isInternal()) v = v.left();              // find leftmost node
    return Iterator(v.parent());
  }

/* SearchTree⟨E⟩ :: */                               // iterator to end entry
  Iterator end()
    { return Iterator(T.root()); }                    // return the super root
```

Code Fragment 10.8: The begin and end functions of class SearchTree. The function end returns a pointer to the super root.

We are now ready to present implementations of the principal class functions, for finding, inserting, and removing entries. We begin by presenting the function find(k) in Code Fragment 10.9. It invokes the recursive utility function finder starting at the root. This utility function is based on the algorithm given in Code Fragment 10.1. The code has been structured so that only the less-than operator needs to be defined on keys.

```
/* SearchTree⟨E⟩ :: */                                // find utility
  TPos finder(const K& k, const TPos& v) {
    if (v.isExternal()) return v;                     // key not found
    if (k < v−>key()) return finder(k, v.left());     // search left subtree
    else if (v−>key() < k) return finder(k, v.right()); // search right subtree
    else return v;                                    // found it here
  }

/* SearchTree⟨E⟩ :: */                                // find entry with key k
  Iterator find(const K& k) {
    TPos v = finder(k, root());                       // search from virtual root
    if (v.isInternal()) return Iterator(v);           // found it
    else return end();                                // didn't find it
  }
```

Code Fragment 10.9: The functions of SearchTree related to finding keys.

The insertion functions are shown in Code Fragment 10.10. The inserter utility does all the work. First, it searches for the key. If found, we continue to search until reaching an external node. (Recall that we allow duplicate keys.) We then create a node, copy the entry information into this node, and update the entry count. The insert function simply invokes the inserter utility, and converts the resulting node position into an iterator.

```
/* SearchTree⟨E⟩ :: */                                // insert utility
  TPos inserter(const K& k, const V& x) {
    TPos v = finder(k, root());                       // search from virtual root
    while (v.isInternal())                            // key already exists?
      v = finder(k, v.right());                       // look further
    T.expandExternal(v);                              // add new internal node
    v−>setKey(k); v−>setValue(x);                     // set entry
    n++;                                              // one more entry
    return v;                                         // return insert position
  }

/* SearchTree⟨E⟩ :: */                                // insert (k,x)
  Iterator insert(const K& k, const V& x)
    { TPos v = inserter(k, x); return Iterator(v); }
```

Code Fragment 10.10: The functions of SearchTree for inserting entries.

Finally, we present the removal functions in Code Fragment 10.11. We implement the approach presented in Section 10.1.2. If the node has an external child, we set *w* to point to this child. Otherwise, we let *w* be the leftmost external node in *v*'s right subtree. Let *u* be *w*'s parent. We copy *u*'s entry contents to *v*. In all cases, we then remove the external node *w* and its parent through the use of the binary tree functions removeAboveExternal.

```
/* SearchTree⟨E⟩ :: */                                  // remove utility
  TPos eraser(TPos& v) {
    TPos w;
    if (v.left().isExternal()) w = v.left();            // remove from left
    else if (v.right().isExternal()) w = v.right();     // remove from right
    else {                                              // both internal?
      w = v.right();                                    // go to right subtree
      do { w = w.left(); } while (w.isInternal());      // get leftmost node
      TPos u = w.parent();
      v->setKey(u->key()); v->setValue(u->value()); // copy w's parent to v
    }
    n−−;                                                // one less entry
    return T.removeAboveExternal(w);                    // remove w and parent
  }

/* SearchTree⟨E⟩ :: */                                  // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos v = finder(k, root());                         // search from virtual root
    if (v.isExternal())                                 // not found?
      throw NonexistentElement("Erase of nonexistent");
    eraser(v);                                          // remove it
  }

/* SearchTree⟨E⟩ :: */                                  // erase entry at p
  void erase(const Iterator& p)
    { eraser(p.v); }
```

Code Fragment 10.11: The functions of SearchTree involved with removing entries.

When updating node entries (in inserter and eraser), we explicitly change only the key and value (using setKey and setValue). You might wonder, what else is there to change? Later in this chapter, we present data structures that are based on modifying the Entry class. It is important that only the key and value data are altered when copying nodes for these structures.

Our implementation has focused on the main elements of the binary search tree implementation. There are a few more things that could have been included. It is a straightforward exercise to implement the dictionary operation findAll. It would also be worthwhile to implement the decrement operator ("−−"), which moves an iterator to its inorder predecessor.

10.2 AVL Trees

In the previous section, we discussed what should be an efficient map data structure, but the worst-case performance it achieves for the various operations is linear time, which is no better than the performance of list- and array-based map implementations (such as the unordered lists and search tables discussed in Chapter 9). In this section, we describe a simple way of correcting this problem in order to achieve logarithmic time for all the fundamental map operations.

Definition of an AVL Tree

The simple correction is to add a rule to the binary search tree definition that maintains a logarithmic height for the tree. The rule we consider in this section is the following ***height-balance property***, which characterizes the structure of a binary search tree T in terms of the heights of its internal nodes (recall from Section 7.2.1 that the height of a node v in a tree is the length of the longest path from v to an external node):

***Height-Balance Property*:** For every internal node v of T, the heights of the children of v differ by at most 1.

Any binary search tree T that satisfies the height-balance property is said to be an ***AVL tree***, named after the initials of its inventors, Adel'son-Vel'skii and Landis. An example of an AVL tree is shown in Figure 10.8.

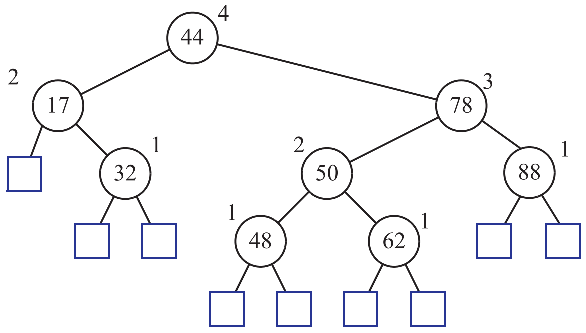


Figure 10.8: An example of an AVL tree. The keys of the entries are shown inside the nodes, and the heights of the nodes are shown next to the nodes.

An immediate consequence of the height-balance property is that a subtree of an AVL tree is itself an AVL tree. The height-balance property has also the important consequence of keeping the height small, as shown in the following proposition.

Proposition 10.2: *The height of an AVL tree storing n entries is $O(\log n)$.*

Justification: Instead of trying to find an upper bound on the height of an AVL tree directly, it turns out to be easier to work on the "inverse problem" of finding a lower bound on the minimum number of internal nodes $n(h)$ of an AVL tree with height h. We show that $n(h)$ grows at least exponentially. From this, it is an easy step to derive that the height of an AVL tree storing n entries is $O(\log n)$.

To start with, notice that $n(1) = 1$ and $n(2) = 2$, because an AVL tree of height 1 must have at least one internal node and an AVL tree of height 2 must have at least two internal nodes. Now, for $h \geq 3$, an AVL tree with height h and the minimum number of nodes is such that both its subtrees are AVL trees with the minimum number of nodes: one with height $h-1$ and the other with height $h-2$. Taking the root into account, we obtain the following formula that relates $n(h)$ to $n(h-1)$ and $n(h-2)$, for $h \geq 3$:

$$n(h) = 1 + n(h-1) + n(h-2). \tag{10.1}$$

At this point, the reader familiar with the properties of Fibonacci progressions (Section 2.2.3 and Exercise C-4.17) already sees that $n(h)$ is a function exponential in h. For the rest of the readers, we will proceed with our reasoning.

Formula 10.1 implies that $n(h)$ is a strictly increasing function of h. Thus, we know that $n(h-1) > n(h-2)$. Replacing $n(h-1)$ with $n(h-2)$ in Formula 10.1 and dropping the 1, we get, for $h \geq 3$,

$$n(h) \; > \; 2 \cdot n(h-2). \tag{10.2}$$

Formula 10.2 indicates that $n(h)$ at least doubles each time h increases by 2, which intuitively means that $n(h)$ grows exponentially. To show this fact in a formal way, we apply Formula 10.2 repeatedly, yielding the following series of inequalities:

$$\begin{aligned} n(h) \; &> \; 2 \cdot n(h-2) \\ &> \; 4 \cdot n(h-4) \\ &> \; 8 \cdot n(h-6) \\ &\;\; \vdots \\ &> \; 2^i \cdot n(h-2i). \end{aligned} \tag{10.3}$$

That is, $n(h) > 2^i \cdot n(h-2i)$, for any integer i, such that $h - 2i \geq 1$. Since we already know the values of $n(1)$ and $n(2)$, we pick i so that $h-2i$ is equal to either 1 or 2. That is, we pick

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

By substituting the above value of i in formula 10.3, we obtain, for $h \geq 3$,

$$\begin{aligned} n(h) &> 2^{\left\lceil \frac{h}{2} \right\rceil - 1} \cdot n\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\left\lceil \frac{h}{2} \right\rceil - 1} n(1) \\ &\geq 2^{\frac{h}{2} - 1}. \end{aligned} \qquad (10.4)$$

By taking logarithms of both sides of formula 10.4, we obtain

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

from which we get

$$h < 2\log n(h) + 2, \qquad (10.5)$$

which implies that an AVL tree storing n entries has height at most $2\log n + 2$. ■

By Proposition 10.2 and the analysis of binary search trees given in Section 10.1, the operation find, in a map implemented with an AVL tree, runs in time $O(\log n)$, where n is the number of entries in the map. Of course, we still have to show how to maintain the height-balance property after an insertion or removal.

10.2.1 Update Operations

The insertion and removal operations for AVL trees are similar to those for binary search trees, but with AVL trees we must perform additional computations.

Insertion

An insertion in an AVL tree T begins as in an insert operation described in Section 10.1.2 for a (simple) binary search tree. Recall that this operation always inserts the new entry at a node w in T that was previously an external node, and it makes w become an internal node with operation insertAtExternal. That is, it adds two external node children to w. This action may violate the height-balance property, however, for some nodes increase their heights by one. In particular, node w, and possibly some of its ancestors, increase their heights by one. Therefore, let us describe how to restructure T to restore its height balance.

Given a binary search tree T, we say that an internal node v of T is ***balanced*** if the absolute value of the difference between the heights of the children of v is at most 1, and we say that it is ***unbalanced*** otherwise. Thus, the height-balance property characterizing AVL trees is equivalent to saying that every internal node is balanced.

Suppose that T satisfies the height-balance property, and hence is an AVL tree, prior to our inserting the new entry. As we have mentioned, after performing the operation insertAtExternal on T, the heights of some nodes of T, including w, increase. All such nodes are on the path of T from w to the root of T, and these are the only nodes of T that may have just become unbalanced. (See Figure 10.9(a).) Of course, if this happens, then T is no longer an AVL tree; hence, we need a mechanism to fix the "unbalance" that we have just caused.

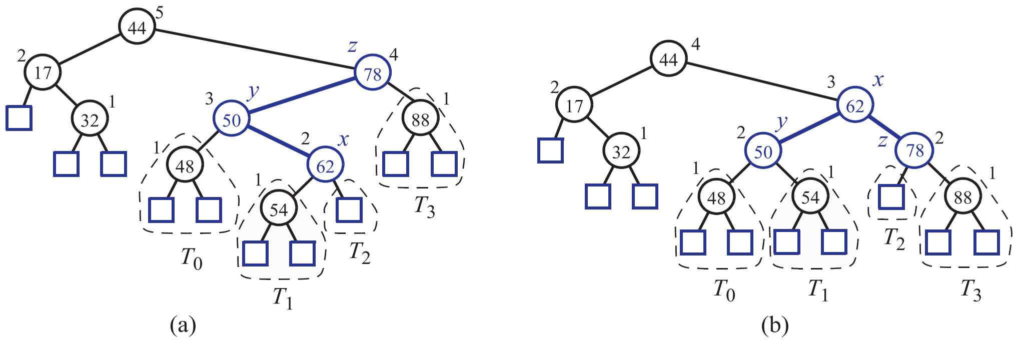


Figure 10.9: An example insertion of an entry with key 54 in the AVL tree of Figure 10.8: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes next to them, and we identify the nodes x, y, and z participating in the trinode restructuring.

We restore the balance of the nodes in the binary search tree T by a simple "search-and-repair" strategy. In particular, let z be the first node we encounter in going up from w toward the root of T such that z is unbalanced. (See Figure 10.9(a).) Also, let y denote the child of z with higher height (and note that node y must be an ancestor of w). Finally, let x be the child of y with higher height (there cannot be a tie and node x must be an ancestor of w). Also, node x is a grandchild of z and could be equal to w. Since z became unbalanced because of an insertion in the subtree rooted at its child y, the height of y is 2 greater than its sibling.

We now rebalance the subtree rooted at z by calling the ***trinode restructuring*** function, restructure(x), given in Code Fragment 10.12 and illustrated in Figures 10.9 and 10.10. A trinode restructuring temporarily renames the nodes x, y, and z as a, b, and c, so that a precedes b and b precedes c in an inorder traversal of T. There are four possible ways of mapping x, y, and z to a, b, and c, as shown in Figure 10.10, which are unified into one case by our relabeling. The trinode restructuring then replaces z with the node called b, makes the children of this node be a and c, and makes the children of a and c be the four previous children of x, y, and z (other than x and y) while maintaining the inorder relationships of all the nodes in T.

```
Algorithm restructure($x$):
    Input: A node $x$ of a binary search tree $T$ that has both a parent $y$ and a grand-
      parent $z$
    Output: Tree $T$ after a trinode restructuring (which corresponds to a single or
      double rotation) involving nodes $x$, $y$, and $z$
  1: Let $(a,b,c)$ be a left-to-right (inorder) listing of the nodes $x$, $y$, and $z$, and let
     $(T_0,T_1,T_2,T_3)$ be a left-to-right (inorder) listing of the four subtrees of $x$, $y$, and
     $z$ not rooted at $x$, $y$, or $z$.
  2: Replace the subtree rooted at $z$ with a new subtree rooted at $b$.
  3: Let $a$ be the left child of $b$ and let $T_0$ and $T_1$ be the left and right subtrees of $a$,
     respectively.
  4: Let $c$ be the right child of $b$ and let $T_2$ and $T_3$ be the left and right subtrees of $c$,
     respectively.
```

Code Fragment 10.12: The trinode restructuring operation in a binary search tree.

The modification of a tree T caused by a trinode restructuring operation is often called a ***rotation***, because of the geometric way we can visualize the way it changes T. If $b = y$, the trinode restructuring method is called a ***single rotation***, for it can be visualized as "rotating" y over z. (See Figure 10.10(a) and (b).) Otherwise, if $b = x$, the trinode restructuring operation is called a ***double rotation***, for it can be visualized as first "rotating" x over y and then over z. (See Figure 10.10(c) and (d), and Figure 10.9.) Some computer researchers treat these two kinds of rotations as separate methods, each with two symmetric types. We have chosen, however, to unify these four types of rotations into a single trinode restructuring operation. No matter how we view it, though, the trinode restructuring method modifies parent-child relationships of $O(1)$ nodes in T, while preserving the inorder traversal ordering of all the nodes in T.

In addition to its order-preserving property, a trinode restructuring changes the heights of several nodes in T, so as to restore balance. Recall that we execute the function restructure(x) because z, the grandparent of x, is unbalanced. Moreover, this unbalance is due to one of the children of x now having too large a height relative to the height of z's other child. As a result of a rotation, we move up the "tall" child of x while pushing down the "short" child of z. Thus, after performing restructure(x), all the nodes in the subtree now rooted at the node we called b are balanced. (See Figure 10.10.) Thus, we restore the height-balance property ***locally*** at the nodes x, y, and z. In addition, since after performing the new entry insertion the subtree rooted at b replaces the one formerly rooted at z, which was taller by one unit, all the ancestors of z that were formerly unbalanced become balanced. (See Figure 10.9.) (The justification of this fact is left as Exercise C-10.14.) Therefore, this one restructuring also restores the height-balance property ***globally***.

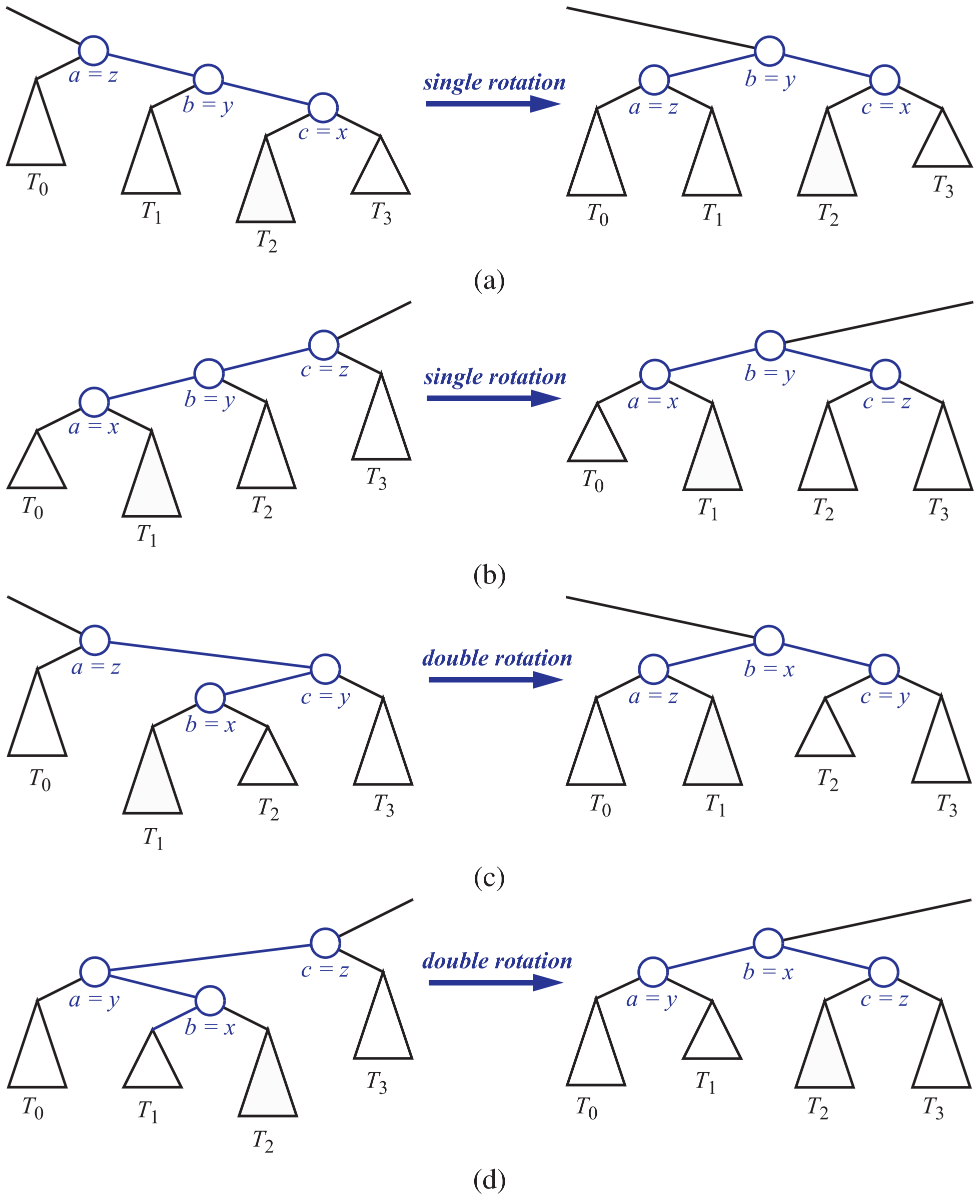


Figure 10.10: Schematic illustration of a trinode restructuring operation (Code Fragment 10.12): (a) and (b) a single rotation; (c) and (d) a double rotation.

Removal

As was the case for the insert map operation, we begin the implementation of the erase map operation on an AVL tree T by using the algorithm for performing this operation on a regular binary search tree. The added difficulty in using this approach with an AVL tree is that it may violate the height-balance property. In particular, after removing an internal node with operation removeAboveExternal and elevating one of its children into its place, there may be an unbalanced node in T on the path from the parent w of the previously removed node to the root of T. (See Figure 10.11(a).) In fact, there can be one such unbalanced node at most. (The justification of this fact is left as Exercise C-10.13.)

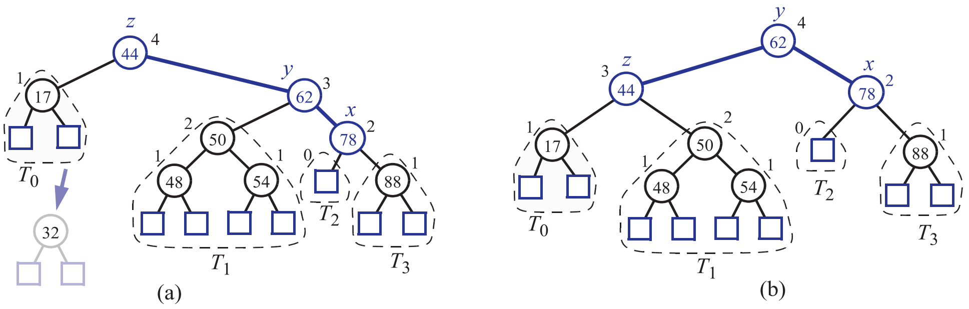


Figure 10.11: Removal of the entry with key 32 from the AVL tree of Figure 10.8: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

As with insertion, we use trinode restructuring to restore balance in the tree T. In particular, let z be the first unbalanced node encountered going up from w toward the root of T. Also, let y be the child of z with larger height (note that node y is the child of z that is not an ancestor of w), and let x be the child of y defined as follows: if one of the children of y is taller than the other, let x be the taller child of y; else (both children of y have the same height), let x be the child of y on the same side as y (that is, if y is a left child, let x be the left child of y, else let x be the right child of y). In any case, we then perform a restructure(x) operation, which restores the height-balance property ***locally***, at the subtree that was formerly rooted at z and is now rooted at the node we temporarily called b. (See Figure 10.11(b).)

Unfortunately, this trinode restructuring may reduce the height of the subtree rooted at b by 1, which may cause an ancestor of b to become unbalanced. So, after rebalancing z, we continue walking up T looking for unbalanced nodes. If we find another, we perform a restructure operation to restore its balance, and continue marching up T looking for more, all the way to the root. Still, since the height of T is $O(\log n)$, where n is the number of entries, by Proposition 10.2, $O(\log n)$ trinode restructurings are sufficient to restore the height-balance property.

Performance of AVL Trees

We summarize the analysis of the performance of an AVL tree T as follows. Operations find, insert, and erase visit the nodes along a root-to-leaf path of T, plus, possibly, their siblings, and spend $O(1)$ time per node. Thus, since the height of T is $O(\log n)$ by Proposition 10.2, each of the above operations takes $O(\log n)$ time. In Table 10.2, we summarize the performance of a map implemented with an AVL tree. We illustrate this performance in Figure 10.12.

Operation	**Time*Time***
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

Table 10.2: Performance of an n-entry map realized by an AVL tree. The space usage is $O(n)$.

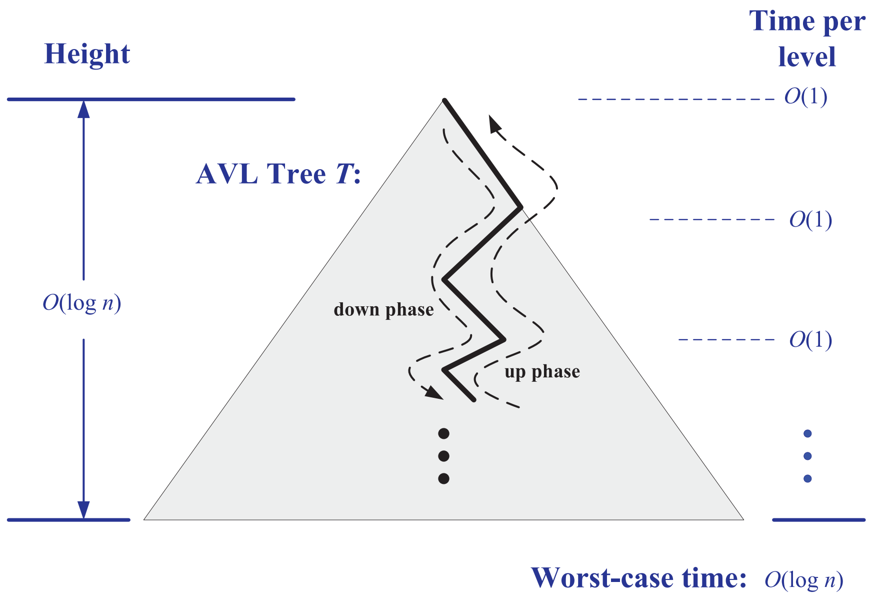


Figure 10.12: Illustrating the running time of searches and updates in an AVL tree. The time performance is $O(1)$ per level, broken into a down phase, which typically involves searching, and an up phase, which typically involves updating height values and performing local trinode restructurings (rotations).

10.2.2 C++ Implementation of an AVL Tree

Let us now turn to the implementation details and analysis of using an AVL tree T with n internal nodes to implement an ordered dictionary of n entries. The insertion and removal algorithms for T require that we are able to perform trinode restructurings and determine the difference between the heights of two sibling nodes. Regarding restructurings, we now need to make sure our underlying implementation of a binary search tree includes the method restructure(x), which performs a trinode restructuring operation (Code Fragment 10.12). (We do not provide an implementation of this function, but it is a straightforward addition to the linked binary tree class given in Section 7.3.4.) It is easy to see that a restructure operation can be performed in $O(1)$ time if T is implemented with a linked structure. We assume that the SearchTree class includes this function.

Regarding height information, we have chosen to store the height of each internal node, v, explicitly in each node. Alternatively, we could have stored the ***balance factor*** of v at v, which is defined as the height of the left child of v minus the height of the right child of v. Thus, the balance factor of v is always equal to -1, 0, or 1, except during an insertion or removal, when it may become ***temporarily*** equal to -2 or $+2$. During the execution of an insertion or removal, the heights and balance factors of $O(\log n)$ nodes are affected and can be maintained in $O(\log n)$ time.

In order to store the height information, we derive a subclass, called AVLEntry, from the standard entry class given earlier in Code Fragment 10.3. It is templated with the base entry type, from which it inherits the key and value members. It defines a member variable ht, which stores the height of the subtree rooted at the associated node. It provides member functions for accessing and setting this value. These functions are protected, so that a user cannot access them, but AVLTree can.

```
template <typename E>
class AVLEntry : public E {                              // an AVL entry
private:
  int ht;                                                // node height
protected:                                               // local types
  typedef typename E::Key K;                             // key type
  typedef typename E::Value V;                           // value type
  int height() const { return ht; }                      // get height
  void setHeight(int h) { ht = h; }                      // set height
public:                                                  // public functions
  AVLEntry(const K& k = K(), const V& v = V()) // constructor
      : E(k,v), ht(0) { }
  friend class AVLTree<E>;                               // allow AVLTree access
};
```

Code Fragment 10.13: An enhanced key-value entry for class AVLTree, containing the height of the associated node.

In Code Fragment 10.14, we present the class definition for AVLTree. This class is derived from the class SearchTree, but using our enhanced AVLEntry in order to maintain height information for the nodes of the tree. The class defines a number of typedef shortcuts for referring to entities such as keys, values, and tree positions. The class declares all the standard dictionary public member functions. At the end, it also defines a number of protected utility functions, which are used in maintaining the AVL tree balance properties.

```
template <typename E>                                    // an AVL tree
class AVLTree : public SearchTree< AVLEntry<E> > {
public:                                                  // public types
  typedef AVLEntry<E> AVLEntry;                          // an entry
  typedef typename SearchTree<AVLEntry>::Iterator Iterator; // an iterator
protected:                                               // local types
  typedef typename AVLEntry::Key K;                      // a key
  typedef typename AVLEntry::Value V;                    // a value
  typedef SearchTree<AVLEntry> ST;                       // a search tree
  typedef typename ST::TPos TPos;                        // a tree position
public:                                                  // public functions
  AVLTree();                                             // constructor
  Iterator insert(const K& k, const V& x);               // insert (k,x)
  void erase(const K& k) throw(NonexistentElement); // remove key k entry
  void erase(const Iterator& p);                         // remove entry at p
protected:                                               // utility functions
  int height(const TPos& v) const;                       // node height utility
  void setHeight(TPos v);                                // set height utility
  bool isBalanced(const TPos& v) const;                  // is v balanced?
  TPos tallGrandchild(const TPos& v) const;              // get tallest grandchild
  void rebalance(const TPos& v);                         // rebalance utility
};
```

Code Fragment 10.14: Class AVLTree, an AVL tree implementation of a dictionary.

Next, in Code Fragment 10.15, we present the constructor and height utility function. The constructor simply invokes the constructor for the binary search tree, which creates a tree having no entries. The function height returns the height of a node, by extracting the height information from the AVLEntry. We employ the condensed function notation that we introduced in Section 9.2.7.

```
/* AVLTree⟨E⟩ :: */                                      // constructor
  AVLTree() : ST() { }

/* AVLTree⟨E⟩ :: */                                      // node height utility
  int height(const TPos& v) const
    { return (v.isExternal() ? 0 : v−>height()); }
```

Code Fragment 10.15: The constructor for class AVLTree and a utility for extracting heights.

In Code Fragment 10.16, we present a few utility functions needed for maintaining the tree's balance. The function setHeight sets the height information for a node as one more than the maximum of the heights of its two children. The function isBalanced determines whether a node satisfies the AVL balance condition, by checking that the height difference between its children is at most 1. Finally, the function tallGrandchild determines the tallest grandchild of a node. Recall that this procedure is needed by the removal operation to determine the node to which the restructuring operation will be applied.

```
/* AVLTree⟨E⟩ :: */                                  // set height utility
  void setHeight(TPos v) {
    int hl = height(v.left());
    int hr = height(v.right());
    v−>setHeight(1 + std::max(hl, hr));                // max of left & right
  }

/* AVLTree⟨E⟩ :: */                                  // is v balanced?
  bool isBalanced(const TPos& v) const {
    int bal = height(v.left()) − height(v.right());
    return ((−1 <= bal) && (bal <= 1));
  }

/* AVLTree⟨E⟩ :: */                                  // get tallest grandchild
  TPos tallGrandchild(const TPos& z) const {
    TPos zl = z.left();
    TPos zr = z.right();
    if (height(zl) >= height(zr))                      // left child taller
      if (height(zl.left()) >= height(zl.right()))
        return zl.left();
      else
        return zl.right();
    else                                               // right child taller
      if (height(zr.right()) >= height(zr.left()))
        return zr.right();
      else
        return zr.left();
  }
```

Code Fragment 10.16: Some utility functions used for maintaining balance in the AVL tree.

Next, we present the principal function for rebalancing the AVL tree after an insertion or removal. The procedure starts at the node v affected by the operation. It then walks up the tree to the root level. On visiting each node z, it updates z's height information (which may have changed due to the update operation) and checks whether z is balanced. If not, it finds z's tallest grandchild, and applies the restructuring operation to this node. Since heights may have changed as a result, it updates the height information for z's children and itself.

```
/* AVLTree⟨E⟩ :: */                                // rebalancing utility
  void rebalance(const TPos& v) {
    TPos z = v;
    while (!(z == ST::root())) {                  // rebalance up to root
      z = z.parent();
      setHeight(z);                               // compute new height
      if (!isBalanced(z)) {                       // restructuring needed
        TPos x = tallGrandchild(z);
        z = restructure(x);                       // trinode restructure
        setHeight(z.left());                      // update heights
        setHeight(z.right());
        setHeight(z);
      }
    }
  }
```

Code Fragment 10.17: Rebalancing the tree after an update operation.

Finally, in Code Fragment 10.18, we present the functions for inserting and erasing keys. (We have omitted the iterator-based erase function, since it is very simple.) Each invokes the associated utility function (inserter or eraser, respectively) from the base class SearchTree. Each then invokes rebalance to restore balance to the tree.

```
/* AVLTree⟨E⟩ :: */                                // insert (k,x)
  Iterator insert(const K& k, const V& x) {
    TPos v = inserter(k, x);                      // insert in base tree
    setHeight(v);                                 // compute its height
    rebalance(v);                                 // rebalance if needed
    return Iterator(v);
  }

/* AVLTree⟨E⟩ :: */                                // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos v = finder(k, ST::root());               // find in base tree
    if (Iterator(v) == ST::end())                 // not found?
      throw NonexistentElement("Erase of nonexistent");
    TPos w = eraser(v);                           // remove it
    rebalance(w);                                 // rebalance if needed
  }
```

Code Fragment 10.18: The insertion and erasure functions.

10.3 Splay Trees

Another way we can implement the fundamental map operations is to use a balanced search tree data structure known as a ***splay tree***. This structure is conceptually quite different from the other balanced search trees we discuss in this chapter, for a splay tree does not use any explicit rules to enforce its balance. Instead, it applies a certain move-to-root operation, called ***splaying***, after every access, in order to keep the search tree balanced in an amortized sense. The splaying operation is performed at the bottom-most node x reached during an insertion, deletion, or even a search. The surprising thing about splaying is that it allows us to guarantee an amortized running time for insertions, deletions, and searches, that is logarithmic. The structure of a ***splay tree*** is simply a binary search tree T. In fact, there are no additional height, balance, or color labels that we associate with the nodes of this tree.

10.3.1 Splaying

Given an internal node x of a binary search tree T, we ***splay*** x by moving x to the root of T through a sequence of restructurings. The particular restructurings we perform are important, for it is not sufficient to move x to the root of T by just any sequence of restructurings. The specific operation we perform to move x up depends upon the relative positions of x, its parent y, and (if it exists) x's grandparent z. There are three cases that we consider.

***zig-zig*:** The node x and its parent y are both left children or both right children. (See Figure 10.13.) We replace z by x, making y a child of x and z a child of y, while maintaining the inorder relationships of the nodes in T.

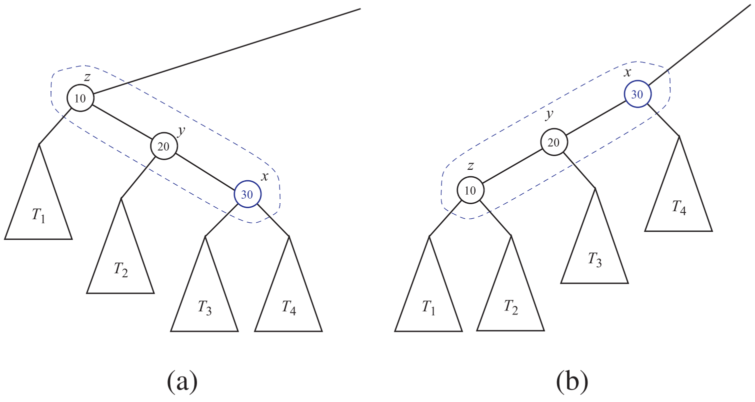


Figure 10.13: Zig-zig: (a) before; (b) after. There is another symmetric configuration where x and y are left children.

***zig-zag*:** One of x and y is a left child and the other is a right child. (See Figure 10.14.) In this case, we replace z by x and make x have y and z as its children, while maintaining the inorder relationships of the nodes in T.

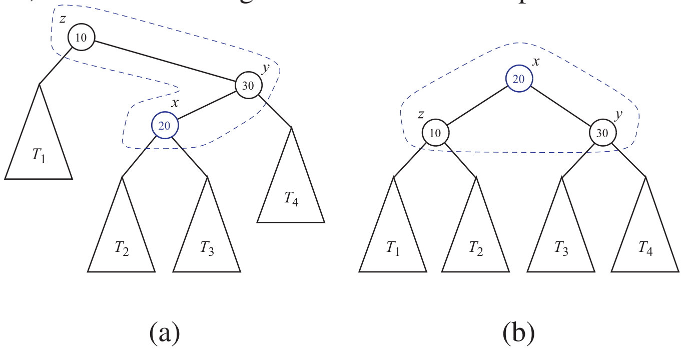


Figure 10.14: Zig-zag: (a) before; (b) after. There is another symmetric configuration where x is a right child and y is a left child.

***zig*:** x does not have a grandparent (or we are not considering x's grandparent for some reason). (See Figure 10.15.) In this case, we rotate x over y, making x's children be the node y and one of x's former children w, in order to maintain the relative inorder relationships of the nodes in T.

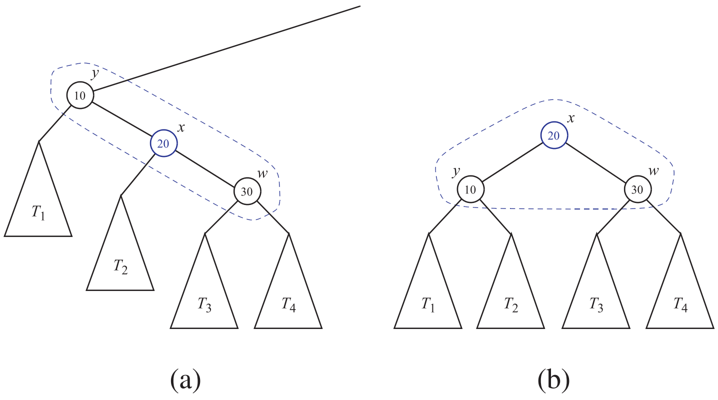
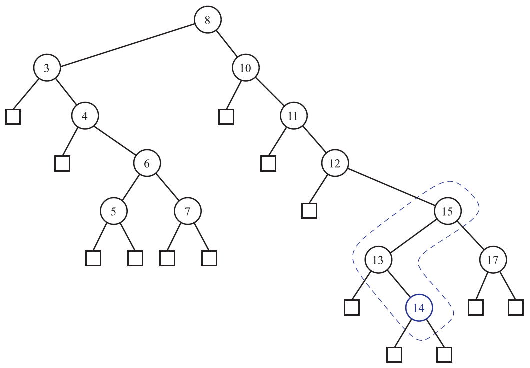
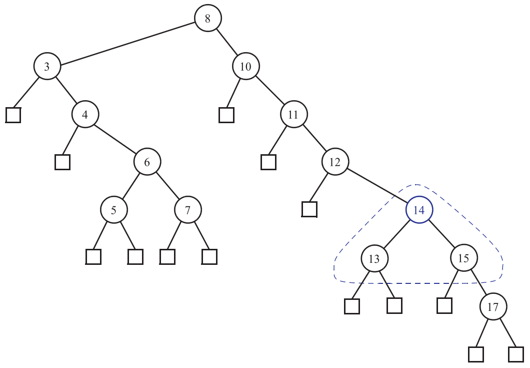


Figure 10.15: Zig: (a) before; (b) after. There is another symmetric configuration where x and w are left children.

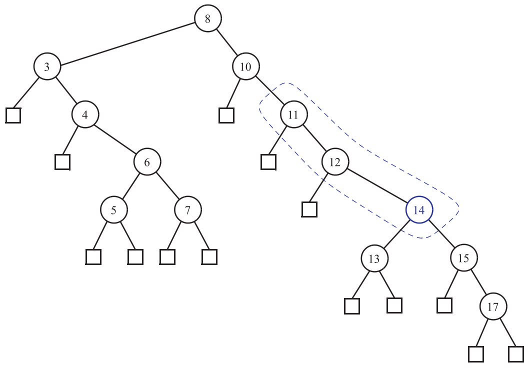
We perform a zig-zig or a zig-zag when x has a grandparent, and we perform a zig when x has a parent but not a grandparent. A ***splaying*** step consists of repeating these restructurings at x until x becomes the root of T. Note that this is not the same as a sequence of simple rotations that brings x to the root. An example of the splaying of a node is shown in Figures 10.16 and 10.17.



(a)

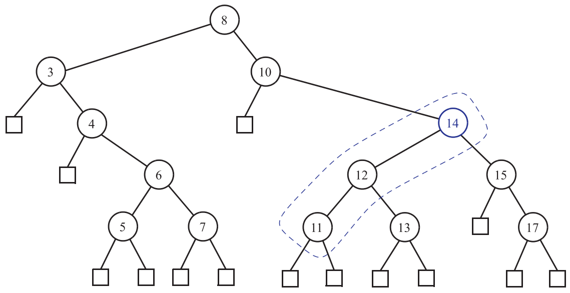


(b)

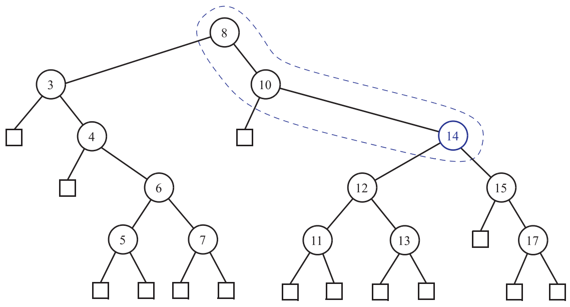


(c)

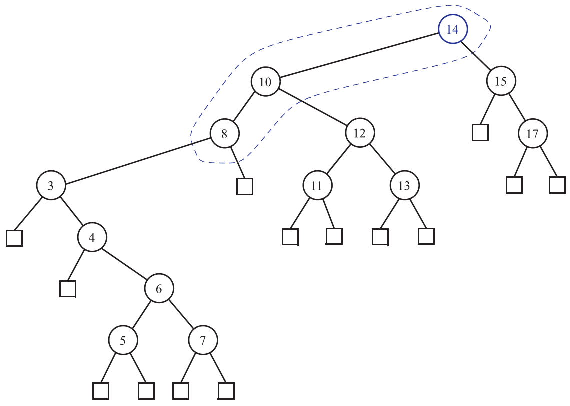
Figure 10.16: Example of splaying a node: (a) splaying the node storing 14 starts with a zig-zag; (b) after the zig-zag; (c) the next step is a zig-zig. (Continues in Figure 10.17.)



(d)



(e)



(f)

Figure 10.17: Example of splaying a node: (d) after the zig-zig; (e) the next step is again a zig-zig; (f) after the zig-zig (Continued from Figure 10.17.)

10.3.2 When to Splay

The rules that dictate when splaying is performed are as follows:

- When searching for key k, if k is found at a node x, we splay x, else we splay the parent of the external node at which the search terminates unsuccessfully. For example, the splaying in Figures 10.16 and 10.17 would be performed after searching successfully for key 14 or unsuccessfully for key 14.5.
- When inserting key k, we splay the newly created internal node where k gets inserted. For example, the splaying in Figures 10.16 and 10.17 would be performed if 14 were the newly inserted key. We show a sequence of insertions in a splay tree in Figure 10.18.

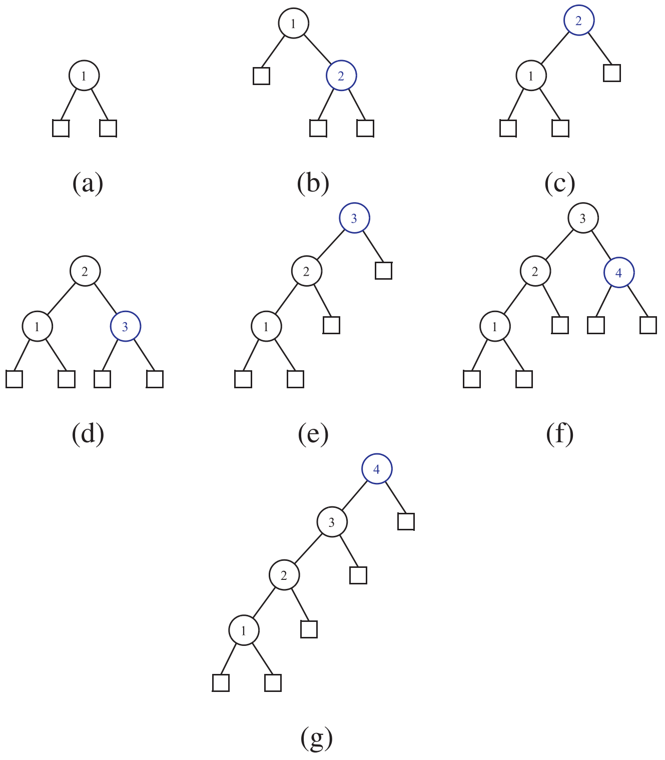


Figure 10.18: A sequence of insertions in a splay tree: (a) initial tree; (b) after inserting 2; (c) after splaying; (d) after inserting 3; (e) after splaying; (f) after inserting 4; (g) after splaying.

- When deleting a key k, we splay the parent of the node w that gets removed, that is, w is either the node storing k or one of its descendents. (Recall the removal algorithm for binary search trees.) An example of splaying following a deletion is shown in Figure 10.19.

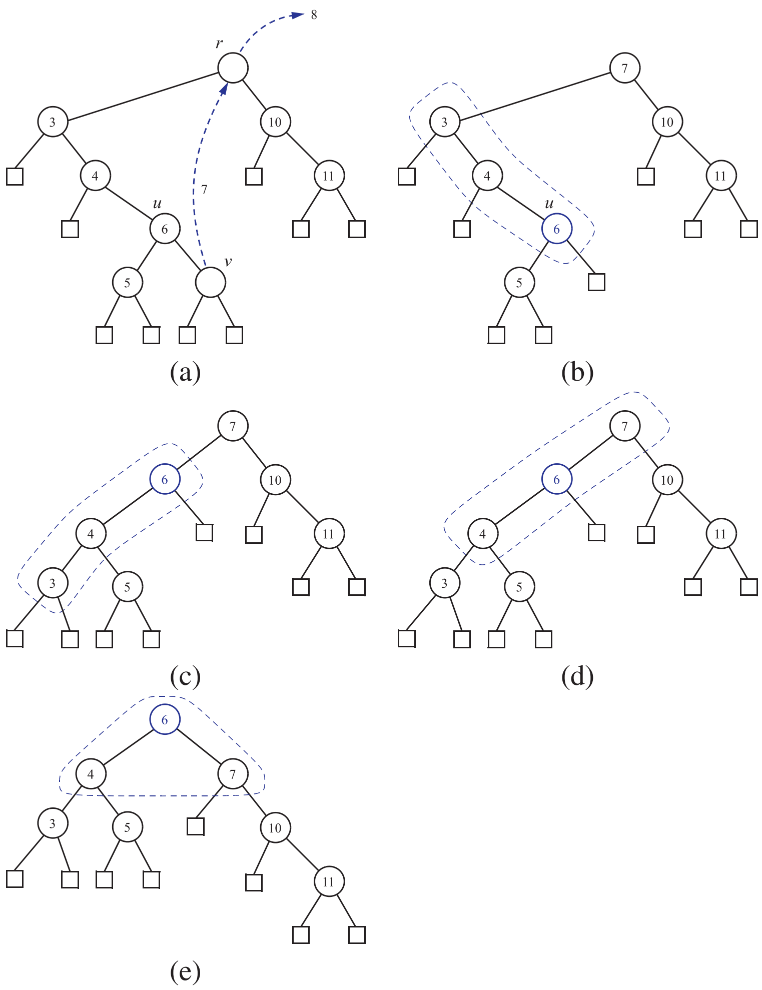


Figure 10.19: Deletion from a splay tree: (a) the deletion of 8 from node r is performed by moving the key of the right-most internal nodr v to r, in the left subtree of r, deleting v, and splaying the parent u of v; (b) splaying u starts with a zig-zig; (c) after the zig-zig; (d) the next step is a zig; (e) after the zig.

10.3.3 Amortized Analysis of Splaying ⋆

After a zig-zig or zig-zag, the depth of x decreases by two, and after a zig the depth of x decreases by one. Thus, if x has depth d, splaying x consists of a sequence of $\lfloor d/2 \rfloor$ zig-zigs and/or zig-zags, plus one final zig if d is odd. Since a single zig-zig, zig-zag, or zig effects a constant number of nodes, it can be done in $O(1)$ time. Thus, splaying a node x in a binary search tree T takes time $O(d)$, where d is the depth of x in T. In other words, the time for performing a splaying step for a node x is asymptotically the same as the time needed just to reach that node in a top-down search from the root of T.

Worst-Case Time

In the worst case, the overall running time of a search, insertion, or deletion in a splay tree of height h is $O(h)$, since the node we splay might be the deepest node in the tree. Moreover, it is possible for h to be as large as n, as shown in Figure 10.18. Thus, from a worst-case point of view, a splay tree is not an attractive data structure.

In spite of its poor worst-case performance, a splay tree performs well in an amortized sense. That is, in a sequence of intermixed searches, insertions, and deletions, each operation takes, on average, logarithmic time. We perform the amortized analysis of splay trees using the accounting method.

Amortized Performance of Splay Trees

For our analysis, we note that the time for performing a search, insertion, or deletion is proportional to the time for the associated splaying. So let us consider only splaying time.

Let T be a splay tree with n keys, and let v be a node of T. We define the ***size*** $n(v)$ of v as the number of nodes in the subtree rooted at v. Note that this definition implies that the size of an internal node is one more than the sum of the sizes of its two children. We define the ***rank*** $r(v)$ of a node v as the logarithm in base 2 of the size of v, that is, $r(v) = \log(n(v))$. Clearly, the root of T has the maximum size $(2n+1)$ and the maximum rank, $\log(2n+1)$, while each external node has size 1 and rank 0.

We use cyber-dollars to pay for the work we perform in splaying a node x in T, and we assume that one cyber-dollar pays for a zig, while two cyber-dollars pay for a zig-zig or a zig-zag. Hence, the cost of splaying a node at depth d is d cyber-dollars. We keep a virtual account storing cyber-dollars at each internal node of T. Note that this account exists only for the purpose of our amortized analysis, and does not need to be included in a data structure implementing the splay tree T.

An Accounting Analysis of Splaying

When we perform a splaying, we pay a certain number of cyber-dollars (the exact value of the payment will be determined at the end of our analysis). We distinguish three cases:

- If the payment is equal to the splaying work, then we use it all to pay for the splaying.
- If the payment is greater than the splaying work, we deposit the excess in the accounts of several nodes.
- If the payment is less than the splaying work, we make withdrawals from the accounts of several nodes to cover the deficiency.

We show below that a payment of $O(\log n)$ cyber-dollars per operation is sufficient to keep the system working, that is, to ensure that each node keeps a nonnegative account balance.

An Accounting Invariant for Splaying

We use a scheme in which transfers are made between the accounts of the nodes to ensure that there will always be enough cyber-dollars to withdraw for paying for splaying work when needed.

In order to use the accounting method to perform our analysis of splaying, we maintain the following invariant:

> ***Before and after a splaying, each node*** v ***of*** T ***has*** $r(v)$ ***cyber-dollars in its account.***

Note that the invariant is "financially sound," since it does not require us to make a preliminary deposit to endow a tree with zero keys.

Let $r(T)$ be the sum of the ranks of all the nodes of T. To preserve the invariant after a splaying, we must make a payment equal to the splaying work plus the total change in $r(T)$. We refer to a single zig, zig-zig, or zig-zag operation in a splaying as a splaying ***substep***. Also, we denote the rank of a node v of T before and after a splaying substep with $r(v)$ and $r'(v)$, respectively. The following proposition gives an upper bound on the change of $r(T)$ caused by a single splaying substep. We repeatedly use this lemma in our analysis of a full splaying of a node to the root.

Proposition 10.3: *Let δ be the variation of $r(T)$ caused by a single splaying substep (a zig, zig-zig, or zig-zag) for a node x in T. We have the following:*

- *$\delta \le 3(r'(x) - r(x)) - 2$ if the substep is a zig-zig or zig-zag*
- *$\delta \le 3(r'(x) - r(x))$ if the substep is a zig*

Justification: We use the fact (see Proposition A.1, Appendix A) that, if $a > 0$, $b > 0$, and $c > a + b$,

$$\log a + \log b \le 2\log c - 2. \tag{10.6}$$

Let us consider the change in $r(T)$ caused by each type of splaying substep.

***zig-zig*:** (Recall Figure 10.13.) Since the size of each node is one more than the size of its two children, note that only the ranks of x, y, and z change in a zig-zig operation, where y is the parent of x and z is the parent of y. Also, $r'(x) = r(z)$, $r'(y) \le r'(x)$, and $r(y) \ge r(x)$. Thus

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)\\ &\le r'(y) + r'(z) - r(x) - r(y)\\ &\le r'(x) + r'(z) - 2r(x).\end{aligned} \tag{10.7}$$

Note that $n(x) + n'(z) \le n'(x)$. By 10.6, $r(x) + r'(z) \le 2r'(x) - 2$, that is,

$$r'(z) \le 2r'(x) - r(x) - 2.$$

This inequality and 10.7 imply

$$\begin{aligned}\delta &\le r'(x) + (2r'(x) - r(x) - 2) - 2r(x)\\ &\le 3(r'(x) - r(x)) - 2.\end{aligned}$$

***zig-zag*:** (Recall Figure 10.14.) Again, by the definition of size and rank, only the ranks of x, y, and z change, where y denotes the parent of x and z denotes the parent of y. Also, $r'(x) = r(z)$ and $r(x) \le r(y)$. Thus

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)\\ &\le r'(y) + r'(z) - r(x) - r(y)\\ &\le r'(y) + r'(z) - 2r(x).\end{aligned} \tag{10.8}$$

Note that $n'(y) + n'(z) \le n'(x)$; hence, by 10.6, $r'(y) + r'(z) \le 2r'(x) - 2$. Thus

$$\begin{aligned}\delta &\le 2r'(x) - 2 - 2r(x)\\ &\le 3(r'(x) - r(x)) - 2.\end{aligned}$$

***zig*:** (Recall Figure 10.15.) In this case, only the ranks of x and y change, where y denotes the parent of x. Also, $r'(y) \le r(y)$ and $r'(x) \ge r(x)$. Thus

$$\begin{aligned}\delta &= r'(y) + r'(x) - r(y) - r(x)\\ &\le r'(x) - r(x)\\ &\le 3(r'(x) - r(x)).\end{aligned}$$

■

Proposition 10.4: *Let T be a splay tree with root t, and let Δ be the total variation of $r(T)$ caused by splaying a node x at depth d. We have*

$$\Delta \le 3(r(t) - r(x)) - d + 2.$$

Justification: Splaying node x consists of $p = \lceil d/2 \rceil$ splaying substeps, each of which is a zig-zig or a zig-zag, except possibly the last one, which is a zig if d is odd. Let $r_0(x) = r(x)$ be the initial rank of x, and for $i = 1, \ldots, p$, let $r_i(x)$ be the rank of x after the ith substep and δ_i be the variation of $r(T)$ caused by the ith substep. By Lemma 10.3, the total variation Δ of $r(T)$ caused by splaying x is

$$\begin{aligned}
\Delta &= \sum_{i=1}^{p} \delta_i \\
&\le \sum_{i=1}^{p} (3(r_i(x) - r_{i-1}(x)) - 2) + 2 \\
&= 3(r_p(x) - r_0(x)) - 2p + 2 \\
&\le 3(r(t) - r(x)) - d + 2.
\end{aligned}$$

■

By Proposition 10.4, if we make a payment of $3(r(t) - r(x)) + 2$ cyber-dollars towards the splaying of node x, we have enough cyber-dollars to maintain the invariant, keeping $r(v)$ cyber-dollars at each node v in T, and pay for the entire splaying work, which costs d dollars. Since the size of the root t is $2n+1$, its rank $r(t) = \log(2n+1)$. In addition, we have $r(x) < r(t)$. Thus, the payment to be made for splaying is $O(\log n)$ cyber-dollars. To complete our analysis, we have to compute the cost for maintaining the invariant when a node is inserted or deleted.

When inserting a new node v into a splay tree with n keys, the ranks of all the ancestors of v are increased. Namely, let $v_0, v_i, \ldots, v_d$ be the ancestors of v, where $v_0 = v$, v_i is the parent of v_{i-1}, and v_d is the root. For $i = 1, \ldots, d$, let $n'(v_i)$ and $n(v_i)$ be the size of v_i before and after the insertion, respectively, and let $r'(v_i)$ and $r(v_i)$ be the rank of v_i before and after the insertion, respectively. We have

$$n'(v_i) = n(v_i) + 1.$$

Also, since $n(v_i) + 1 \le n(v_{i+1})$, for $i = 0, 1, \ldots, d-1$, we have the following for each i in this range

$$r'(v_i) = \log(n'(v_i)) = \log(n(v_i) + 1) \le \log(n(v_{i+1})) = r(v_{i+1}).$$

Thus, the total variation of $r(T)$ caused by the insertion is

$$\begin{aligned}
\sum_{i=1}^{d} \left(r'(v_i) - r(v_i)\right) &\le r'(v_d) + \sum_{i=1}^{d-1} (r(v_{i+1}) - r(v_i)) \\
&= r'(v_d) - r(v_0) \\
&\le \log(2n+1).
\end{aligned}$$

Therefore, a payment of $O(\log n)$ cyber-dollars is sufficient to maintain the invariant when a new node is inserted.

When deleting a node v from a splay tree with n keys, the ranks of all the ancestors of v are decreased. Thus, the total variation of $r(T)$ caused by the deletion is negative, and we do not need to make any payment to maintain the invariant when a node is deleted. Therefore, we may summarize our amortized analysis in the following proposition (which is sometimes called the "balance proposition" for splay trees).

Proposition 10.5: *Consider a sequence of m operations on a splay tree, each one a search, insertion, or deletion, starting from a splay tree with zero keys. Also, let n_i be the number of keys in the tree after operation i, and n be the total number of insertions. The total running time for performing the sequence of operations is*

$$O\left(m + \sum_{i=1}^{m} \log n_i\right),$$

which is $O(m \log n)$.

In other words, the amortized running time of performing a search, insertion, or deletion in a splay tree is $O(\log n)$, where n is the size of the splay tree at the time. Thus, a splay tree can achieve logarithmic-time amortized performance for implementing an ordered map ADT. This amortized performance matches the worst-case performance of AVL trees, $(2,4)$ trees, and red-black trees, but it does so using a simple binary tree that does not need any extra balance information stored at each of its nodes. In addition, splay trees have a number of other interesting properties that are not shared by these other balanced search trees. We explore one such additional property in the following proposition (which is sometimes called the "Static Optimality" proposition for splay trees).

Proposition 10.6: *Consider a sequence of m operations on a splay tree, each one a search, insertion, or deletion, starting from a splay tree T with zero keys. Also, let $f(i)$ denote the number of times the entry i is accessed in the splay tree, that is, its frequency, and let n denote the total number of entries. Assuming that each entry is accessed at least once, then the total running time for performing the sequence of operations is*

$$O\left(m + \sum_{i=1}^{n} f(i) \log(m/f(i))\right).$$

We omit the proof of this proposition, but it is not as hard to justify as one might imagine. The remarkable thing is that this proposition states that the amortized running time of accessing an entry i is $O(\log(m/f(i)))$.

10.4 (2,4) Trees

Some data structures we discuss in this chapter, including $(2,4)$ trees, are multi-way search trees, that is, trees with internal nodes that have two or more children. Thus, before we define $(2,4)$ trees, let us discuss multi-way search trees.

10.4.1 Multi-Way Search Trees

Recall that multi-way trees are defined so that each internal node can have many children. In this section, we discuss how multi-way trees can be used as search trees. Recall that the ***entries*** that we store in a search tree are pairs of the form (k,x), where k is the ***key*** and x is the value associated with the key. However, we do not discuss how to perform updates in multi-way search trees now, since the details for update methods depend on additional properties we want to maintain for multi-way trees, which we discuss in Section 14.3.1.

Definition of a Multi-way Search Tree

Let v be a node of an ordered tree. We say that v is a ***d-node*** if v has d children. We define a ***multi-way search tree*** to be an ordered tree T that has the following properties, which are illustrated in Figure 10.1(a):

- Each internal node of T has at least two children. That is, each internal node is a d-node such that $d \geq 2$.
- Each internal d-node v of T with children $v_1, \ldots, v_d$ stores an ordered set of $d-1$ key-value entries $(k_1,x_1), \ldots, (k_{d-1},x_{d-1})$, where $k_1 \leq \cdots \leq k_{d-1}$.
- Let us conventionally define $k_0 = -\infty$ and $k_d = +\infty$. For each entry (k,x) stored at a node in the subtree of v rooted at v_i, $i = 1, \ldots, d$, we have that $k_{i-1} \leq k \leq k_i$.

That is, if we think of the set of keys stored at v as including the special fictitious keys $k_0 = -\infty$ and $k_d = +\infty$, then a key k stored in the subtree of T rooted at a child node v_i must be "in between" two keys stored at v. This simple viewpoint gives rise to the rule that a d-node stores $d-1$ regular keys, and it also forms the basis of the algorithm for searching in a multi-way search tree.

By the above definition, the external nodes of a multi-way search do not store any entries and serve only as "placeholders," as has been our convention with binary search trees (Section 10.1); hence, a binary search tree can be viewed as a special case of a multi-way search tree, where each internal node stores one entry and has two children. In addition, while the external nodes could be *null*, we make the simplifying assumption here that they are actual nodes that don't store anything.

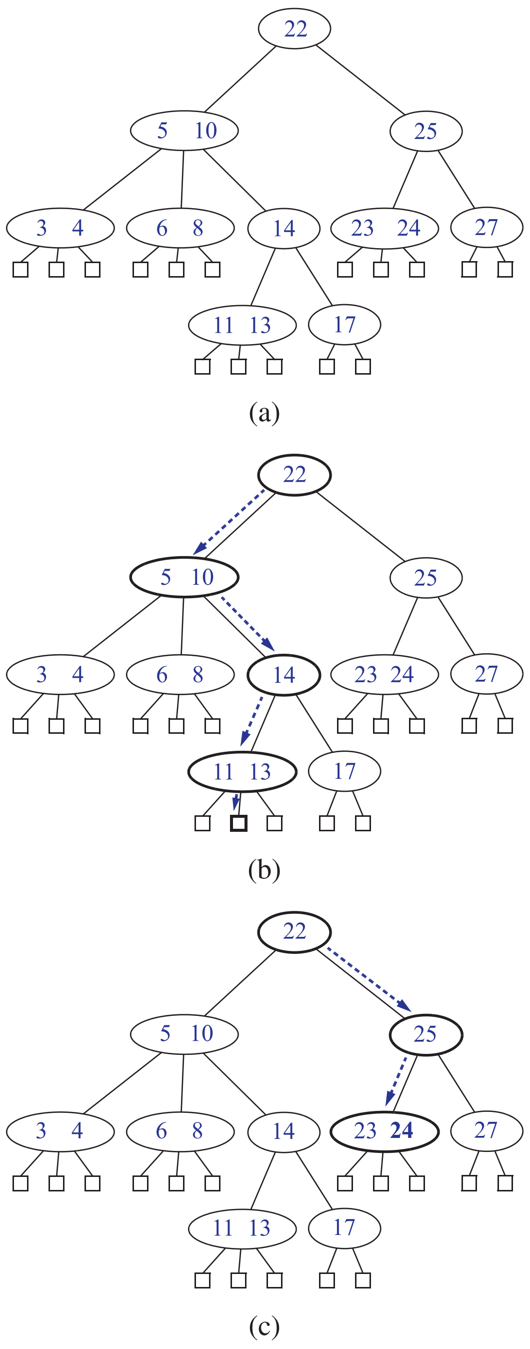


Figure 10.20: (a) A multi-way search tree T; (b) search path in T for key 12 (unsuccessful search); (c) search path in T for key 24 (successful search).

Whether internal nodes of a multi-way tree have two children or many, however, there is an interesting relationship between the number of entries and the number of external nodes.

Proposition 10.7: *An n-entry multi-way search tree has $n+1$ external nodes.*

We leave the justification of this proposition as an exercise (Exercise C-10.17).

Searching in a Multi-Way Tree

Given a multi-way search tree T, we note that searching for an entry with key k is simple. We perform such a search by tracing a path in T starting at the root. (See Figure 10.1(b) and (c).) When we are at a d-node v during this search, we compare the key k with the keys $k_1, \ldots, k_{d-1}$ stored at v. If $k = k_i$ for some i, the search is successfully completed. Otherwise, we continue the search in the child v_i of v such that $k_{i-1} < k < k_i$. (Recall that we conventionally define $k_0 = -\infty$ and $k_d = +\infty$.) If we reach an external node, then we know that there is no entry with key k in T, and the search terminates unsuccessfully.

Data Structures for Representing Multi-way Search Trees

In Section 7.1.4, we discuss a linked data structure for representing a general tree. This representation can also be used for a multi-way search tree. In fact, in using a general tree to implement a multi-way search tree, the only additional information that we need to store at each node is the set of entries (including keys) associated with that node. That is, we need to store with v a reference to some collection that stores the entries for v.

Recall that when we use a binary search tree to represent an ordered map M, we simply store a reference to a single entry at each internal node. In using a multi-way search tree T to represent M, we must store a reference to the ordered set of entries associated with v at each internal node v of T. This reasoning may at first seem like a circular argument, since we need a representation of an ordered map to represent an ordered map. We can avoid any circular arguments, however, by using the ***bootstrapping*** technique, where we use a previous (less advanced) solution to a problem to create a new (more advanced) solution. In this case, bootstrapping consists of representing the ordered set associated with each internal node using a map data structure that we have previously constructed (for example, a search table based on a sorted array, as shown in Section 9.3.1). In particular, assuming we already have a way of implementing ordered maps, we can realize a multi-way search tree by taking a tree T and storing such a map at each node of T.

The map we store at each node v is known as a ***secondary*** data structure, because we are using it to support the bigger, ***primary*** data structure. We denote the map stored at a node v of T as $M(v)$. The entries we store in $M(v)$ allow us to find which child node to move to next during a search operation. Specifically, for each node v of T, with children $v_1,\dots,v_d$ and entries (k_1,x_1), ..., (k_{d-1},x_{d-1}), we store, in the map $M(v)$, the entries

$$(k_1,(x_1,v_1)),(k_2,(x_2,v_2)),\dots,(k_{d-1},(x_{d-1},v_{d-1})),(+\infty,\ (\emptyset,\ v_d)).$$

That is, an entry $(k_i,(x_i,v_i))$ of map $M(v)$ has key k_i and value (x_i,v_i). Note that the last entry stores the special key $+\infty$.

With the realization of the multi-way search tree T above, processing a d-node v while searching for an entry of T with key k can be done by performing a search operation to find the entry $(k_i,(x_i,v_i))$ in $M(v)$ with smallest key greater than or equal to k. We distinguish two cases:

- If $k < k_i$, then we continue the search by processing child v_i. (Note that if the special key $k_d = +\infty$ is returned, then k is greater than all the keys stored at node v, and we continue the search processing child v_d.)
- Otherwise $(k = k_i)$, then the search terminates successfully.

Consider the space requirement for the above realization of a multi-way search tree T storing n entries. By Proposition 10.7, using any of the common realizations of an ordered map (Chapter 9) for the secondary structures of the nodes of T, the overall space requirement for T is $O(n)$.

Consider next the time spent answering a search in T. The time spent at a d-node v of T during a search depends on how we realize the secondary data structure $M(v)$. If $M(v)$ is realized with a sorted array (that is, an ordered search table), then we can process v in $O(\log d)$ time. If $M(v)$ is realized using an unsorted list instead, then processing v takes $O(d)$ time. Let $d_{\max}$ denote the maximum number of children of any node of T, and let h denote the height of T. The search time in a multi-way search tree is either $O(hd_{\max})$ or $O(h\log d_{\max})$, depending on the specific implementation of the secondary structures at the nodes of T (the map $M(v)$). If $d_{\max}$ is a constant, the running time for performing a search is $O(h)$, irrespective of the implementation of the secondary structures.

Thus, the primary efficiency goal for a multi-way search tree is to keep the height as small as possible, that is, we want h to be a logarithmic function of n, the total number of entries stored in the map. A search tree with logarithmic height such as this is called a ***balanced search tree***.

Definition of a $(2,4)$ Tree

A multi-way search tree that keeps the secondary data structures stored at each node small and also keeps the primary multi-way tree balanced is the $(\mathbf{2},\mathbf{4})$ ***tree***, which is sometimes called 2-4 tree or 2-3-4 tree. This data structure achieves these goals by maintaining two simple properties (see Figure 10.21):

Size Property: Every internal node has at most four children

Depth Property: All the external nodes have the same depth

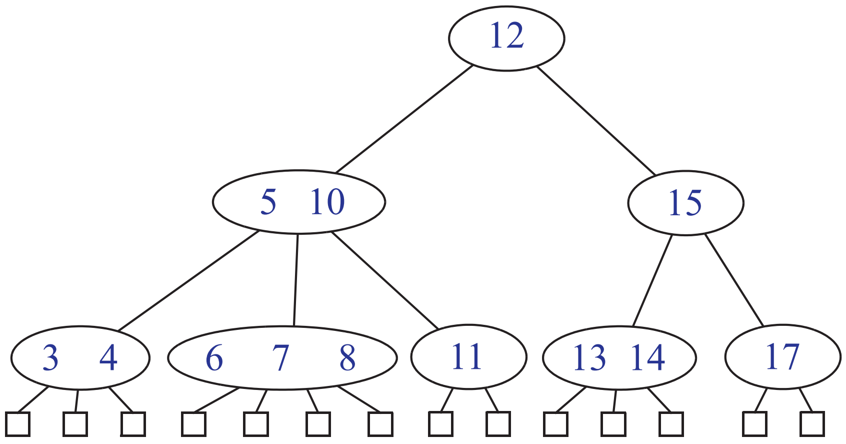


Figure 10.21: A $(2,4)$ tree.

Again, we assume that external nodes are empty and, for the sake of simplicity, we describe our search and update methods assuming that external nodes are real nodes, although this latter requirement is not strictly needed.

Enforcing the size property for $(2,4)$ trees keeps the nodes in the multi-way search tree simple. It also gives rise to the alternative name "2-3-4 tree," since it implies that each internal node in the tree has 2, 3, or 4 children. Another implication of this rule is that we can represent the map $M(v)$ stored at each internal node v using an unordered list or an ordered array, and still achieve $O(1)$-time performance for all operations (since $d_{\max} = 4$). The depth property, on the other hand, enforces an important bound on the height of a $(2,4)$ tree.

Proposition 10.8: *The height of a $(2,4)$ tree storing n entries is $O(\log n)$.*

Justification: Let h be the height of a $(2,4)$ tree T storing n entries. We justify the proposition by showing that the claims

$$\frac{1}{2}\log(n+1) \le h \tag{10.9}$$

and

$$h \le \log(n+1) \tag{10.10}$$

are true.

To justify these claims note first that, by the size property, we can have at most 4 nodes at depth 1, at most 4^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at most 4^h. Likewise, by the depth property and the definition of a $(2,4)$ tree, we must have at least 2 nodes at depth 1, at least 2^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at least 2^h. In addition, by Proposition 10.7, the number of external nodes in T is $n+1$. Therefore, we obtain

$$2^h \le n+1$$

and

$$n+1 \le 4^h.$$

Taking the logarithm in base 2 of each of the above terms, we get that

$$h \le \log(n+1)$$

and

$$\log(n+1) \le 2h,$$

which justifies our claims (10.9 and 10.10). ■

Proposition 10.8 states that the size and depth properties are sufficient for keeping a multi-way tree balanced (Section 10.4.1). Moreover, this proposition implies that performing a search in a $(2,4)$ tree takes $O(\log n)$ time and that the specific realization of the secondary structures at the nodes is not a crucial design choice, since the maximum number of children $d_{\max}$ is a constant (4). We can, for example, use a simple ordered map implementation, such as an array-list search table, for each secondary structure.

10.4.2 Update Operations for $(2,4)$ Trees

Maintaining the size and depth properties requires some effort after performing insertions and removals in a $(2,4)$ tree, however. We discuss these operations next.

Insertion

To insert a new entry (k,x), with key k, into a $(2,4)$ tree T, we first perform a search for k. Assuming that T has no entry with key k, this search terminates unsuccessfully at an external node z. Let v be the parent of z. We insert the new entry into node v and add a new child w (an external node) to v on the left of z. That is, we add entry (k,x,w) to the map $M(v)$.

Our insertion method preserves the depth property, since we add a new external node at the same level as existing external nodes. Nevertheless, it may violate the size property. Indeed, if a node v was previously a 4-node, then it may become a 5-node after the insertion, which causes the tree T to no longer be a $(2,4)$ tree. This type of violation of the size property is called an ***overflow*** at node v, and it must be resolved in order to restore the properties of a $(2,4)$ tree. Let $v_1,\dots,v_5$ be the children of v, and let $k_1,\dots,k_4$ be the keys stored at v. To remedy the overflow at node v, we perform a ***split*** operation on v as follows (see Figure 10.22):

- Replace v with two nodes v' and v'', where
  - v' is a 3-node with children v_1,v_2,v_3 storing keys k_1 and k_2
  - v'' is a 2-node with children v_4,v_5 storing key k_4
- If v was the root of T, create a new root node u; else, let u be the parent of v
- Insert key k_3 into u and make v' and v'' children of u, so that if v was child i of u, then v' and v'' become children i and $i+1$ of u, respectively

We show a sequence of insertions in a $(2,4)$ tree in Figure 10.23.

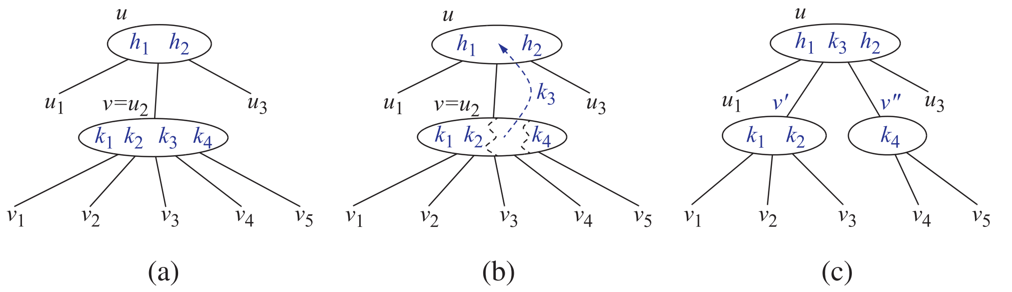


Figure 10.22: A node split: (a) overflow at a 5-node v; (b) the third key of v inserted into the parent u of v; (c) node v replaced with a 3-node v' and a 2-node v''.

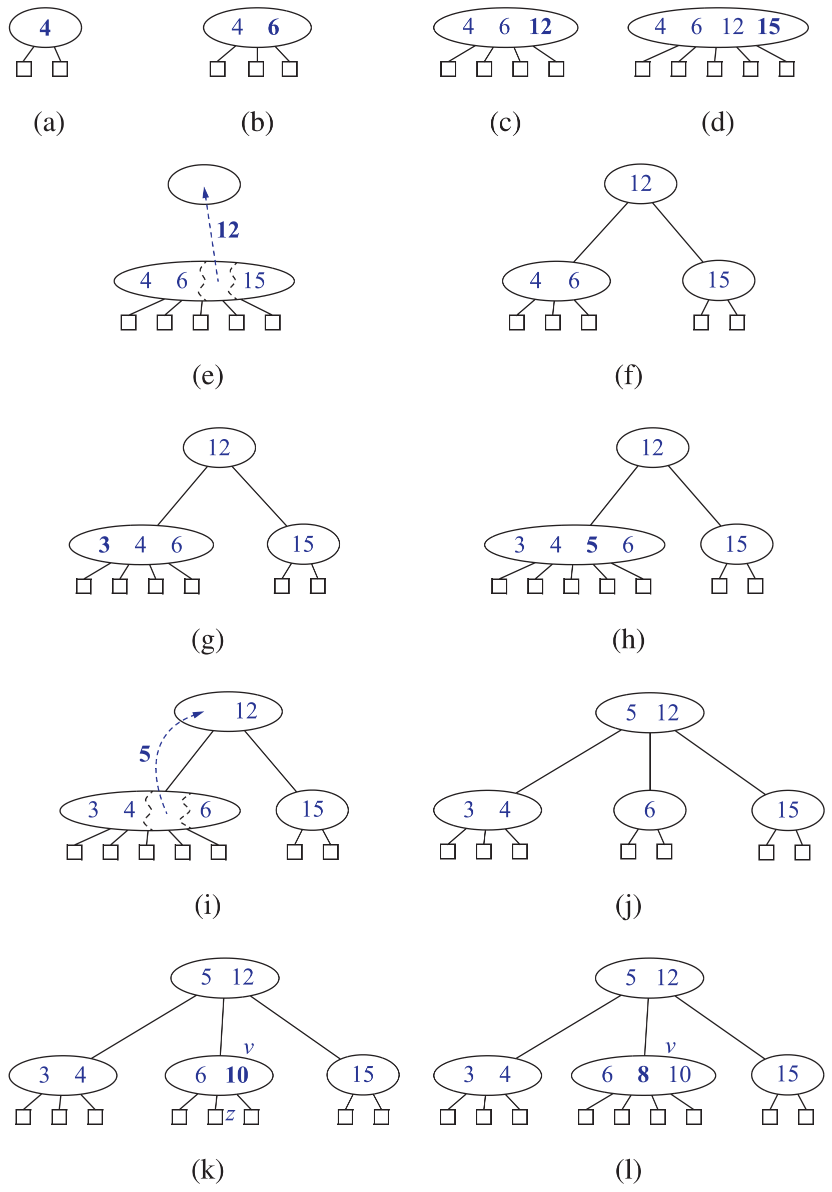


Figure 10.23: A sequence of insertions into a $(2,4)$ tree: (a) initial tree with one entry; (b) insertion of 6; (c) insertion of 12; (d) insertion of 15, which causes an overflow; (e) split, which causes the creation of a new root node; (f) after the split; (g) insertion of 3; (h) insertion of 5, which causes an overflow; (i) split; (j) after the split; (k) insertion of 10; (l) insertion of 8.

Analysis of Insertion in a $(2,4)$ Tree

A split operation affects a constant number of nodes of the tree and $O(1)$ entries stored at such nodes. Thus, it can be implemented to run in $O(1)$ time.

As a consequence of a split operation on node v, a new overflow may occur at the parent u of v. If such an overflow occurs, it triggers a split at node u in turn. (See Figure 10.24.) A split operation either eliminates the overflow or propagates it into the parent of the current node. Hence, the number of split operations is bounded by the height of the tree, which is $O(\log n)$ by Proposition 10.8. Therefore, the total time to perform an insertion in a $(2,4)$ tree is $O(\log n)$.

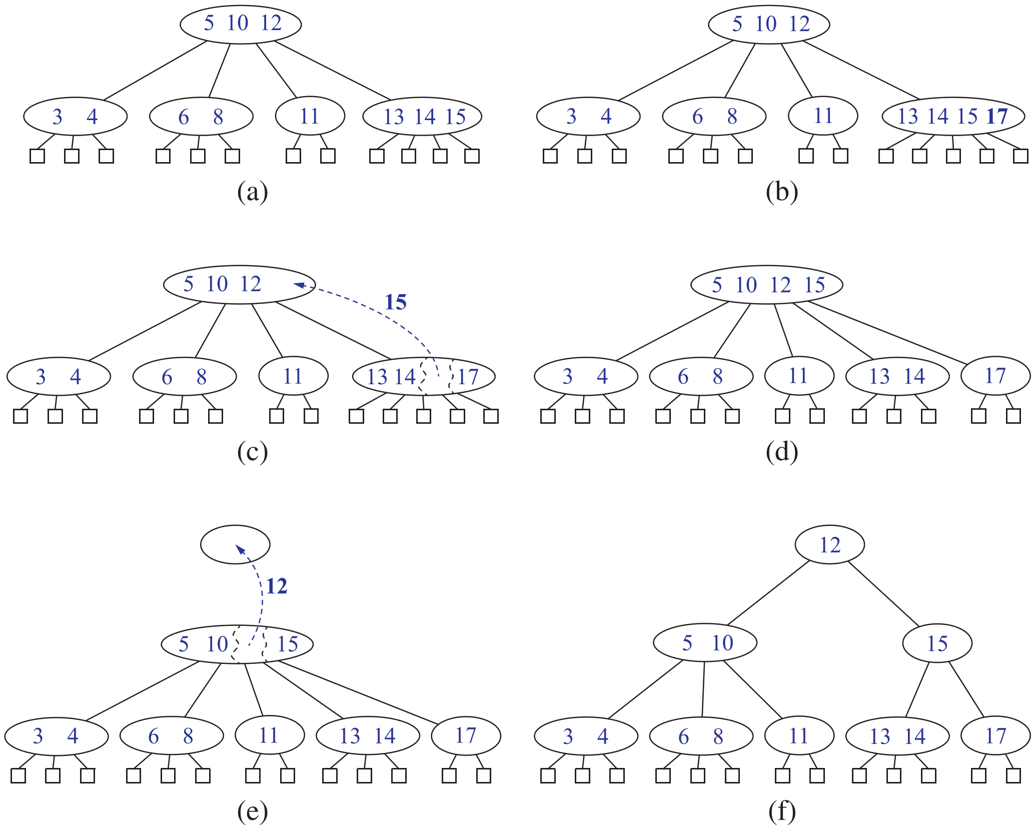


Figure 10.24: An insertion in a $(2,4)$ tree that causes a cascading split: (a) before the insertion; (b) insertion of 17, causing an overflow; (c) a split; (d) after the split a new overflow occurs; (e) another split, creating a new root node; (f) final tree.

Removal

Let us now consider the removal of an entry with key k from a $(2,4)$ tree T. We begin such an operation by performing a search in T for an entry with key k. Removing such an entry from a $(2,4)$ tree can always be reduced to the case where the entry to be removed is stored at a node v whose children are external nodes. Suppose, for instance, that the entry with key k that we wish to remove is stored in the ith entry (k_i, x_i) at a node z that has only internal-node children. In this case, we swap the entry (k_i, x_i) with an appropriate entry that is stored at a node v with external-node children as follows (see Figure 10.25(d)):

1. We find the right-most internal node v in the subtree rooted at the ith child of z, noting that the children of node v are all external nodes.
2. We swap the entry (k_i, x_i) at z with the last entry of v.

Once we ensure that the entry to remove is stored at a node v with only external-node children (because either it was already at v or we swapped it into v), we simply remove the entry from v (that is, from the map $M(v)$) and remove the ith external node of v.

Removing an entry (and a child) from a node v as described above preserves the depth property, because we always remove an external node child from a node v with only external-node children. However, in removing such an external node we may violate the size property at v. Indeed, if v was previously a 2-node, then it becomes a 1-node with no entries after the removal (Figure 10.25(d) and (e)), which is not allowed in a $(2,4)$ tree. This type of violation of the size property is called an ***underflow*** at node v. To remedy an underflow, we check whether an immediate sibling of v is a 3-node or a 4-node. If we find such a sibling w, then we perform a ***transfer*** operation, in which we move a child of w to v, a key of w to the parent u of v and w, and a key of u to v. (See Figure 10.25(b) and (c).) If v has only one sibling, or if both immediate siblings of v are 2-nodes, then we perform a ***fusion*** operation, in which we merge v with a sibling, creating a new node v', and move a key from the parent u of v to v'. (See Figure 10.26(e) and (f).)

A fusion operation at node v may cause a new underflow to occur at the parent u of v, which in turn triggers a transfer or fusion at u. (See Figure 10.26.) Hence, the number of fusion operations is bounded by the height of the tree, which is $O(\log n)$ by Proposition 10.8. If an underflow propagates all the way up to the root, then the root is simply deleted. (See Figure 10.26(c) and (d).) We show a sequence of removals from a $(2,4)$ tree in Figures 10.25 and 10.26.

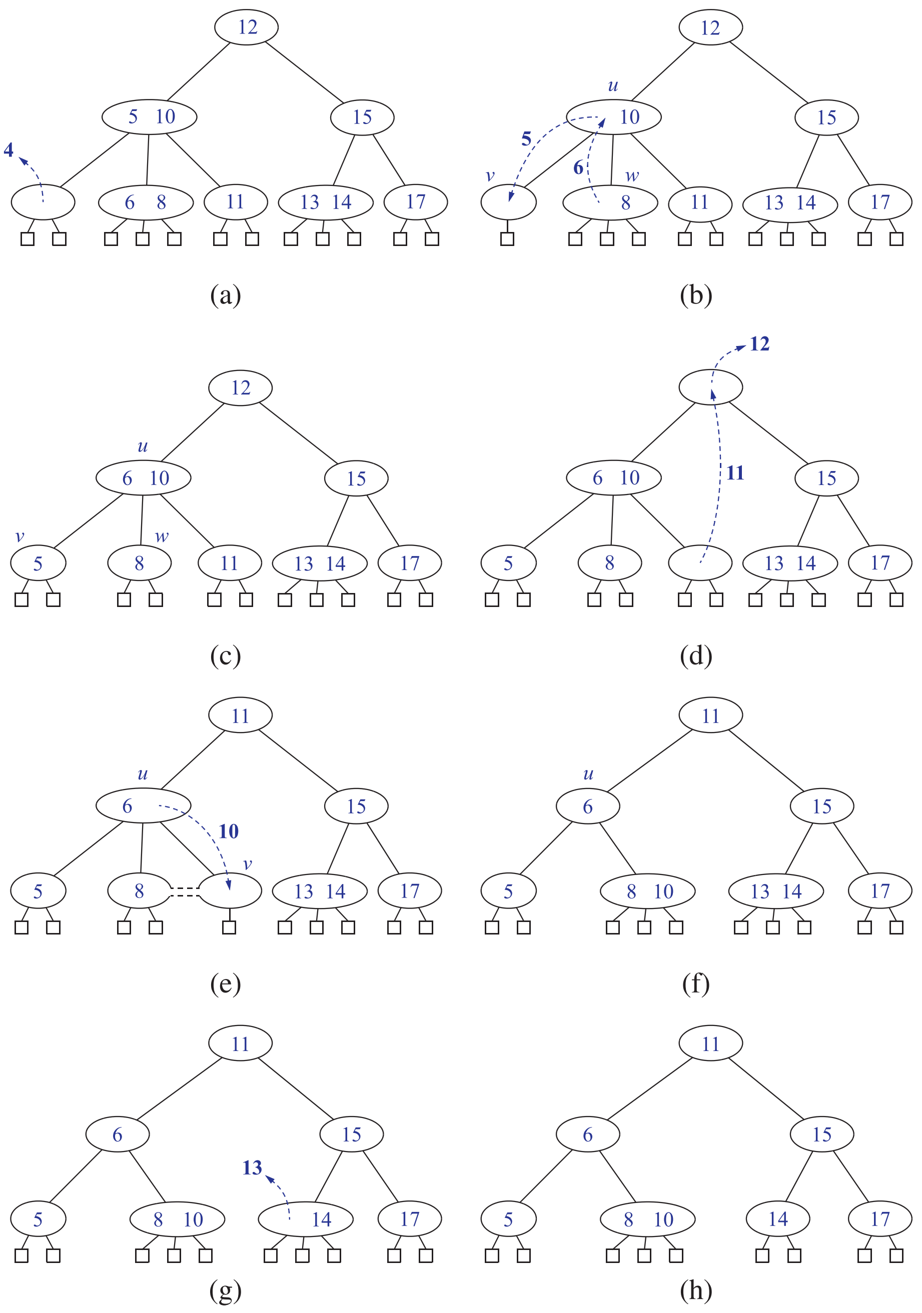


Figure 10.25: A sequence of removals from a $(2,4)$ tree: (a) removal of 4, causing an underflow; (b) a transfer operation; (c) after the transfer operation; (d) removal of 12, causing an underflow; (e) a fusion operation; (f) after the fusion operation; (g) removal of 13; (h) after removing 13.

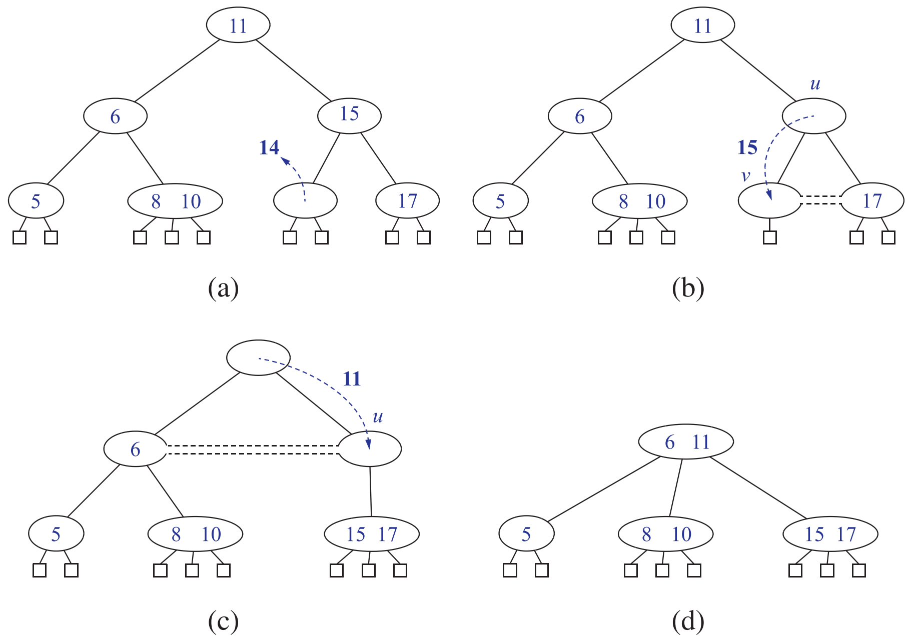


Figure 10.26: A propagating sequence of fusions in a $(2,4)$ tree: (a) removal of 14, which causes an underflow; (b) fusion, which causes another underflow; (c) second fusion operation, which causes the root to be removed; (d) final tree.

Performance of $(2,4)$ Trees

Table 10.3 summarizes the running times of the main operations of a map realized with a $(2,4)$ tree. The time complexity analysis is based on the following:

- The height of a $(2,4)$ tree storing n entries is $O(\log n)$, by Proposition 10.8
- A split, transfer, or fusion operation takes $O(1)$ time
- A search, insertion, or removal of an entry visits $O(\log n)$ nodes.

Operation	***Time***
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

Table 10.3: Performance of an n-entry map realized by a $(2,4)$ tree. The space usage is $O(n)$.

Thus, $(2,4)$ trees provide for fast map search and update operations. $(2,4)$ trees also have an interesting relationship to the data structure we discuss next.

10.5 Red-Black Trees

Although AVL trees and $(2,4)$ trees have a number of nice properties, there are some map applications for which they are not well suited. For instance, AVL trees may require many restructure operations (rotations) to be performed after a removal, and $(2,4)$ trees may require many fusing or split operations to be performed after either an insertion or removal. The data structure we discuss in this section, the red-black tree, does not have these drawbacks, however, as it requires that only $O(1)$ structural changes be made after an update in order to stay balanced.

A ***red-black tree*** is a binary search tree (see Section 10.1) with nodes colored red and black in a way that satisfies the following properties:

***Root Property*:** The root is black.

***External Property*:** Every external node is black.

***Internal Property*:** The children of a red node are black.

Depth Property*:** All the external nodes have the same ***black depth, defined as the number of black ancestors minus one. (Recall that a node is an ancestor of itself.)

An example of a red-black tree is shown in Figure 10.27.

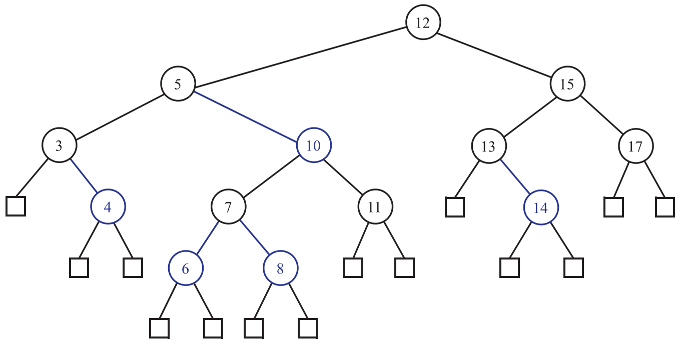


Figure 10.27: Red-black tree associated with the $(2,4)$ tree of Figure 10.21. Each external node of this red-black tree has 4 black ancestors (including itself); hence, it has black depth 3. We use the color blue instead of red. Also, we use the convention of giving an edge of the tree the same color as the child node.

As for previous types of search trees, we assume that entries are stored at the internal nodes of a red-black tree, with the external nodes being empty placeholders. Also, we assume that the external nodes are actual nodes, but we note that, at the expense of slightly more complicated methods, external nodes could be *null*.

We can make the red-black tree definition more intuitive by noting an interesting correspondence between red-black trees and $(2,4)$ trees as illustrated in Figure 10.28. Namely, given a red-black tree, we can construct a corresponding $(2,4)$ tree by merging every red node v into its parent and storing the entry from v at its parent. Conversely, we can transform any $(2,4)$ tree into a corresponding red-black tree by coloring each node black and performing the following transformation for each internal node v:

- If v is a 2-node, then keep the (black) children of v as is
- If v is a 3-node, then create a new red node w, give v's first two (black) children to w, and make w and v's third child be the two children of v
- If v is a 4-node, then create two new red nodes w and z, give v's first two (black) children to w, give v's last two (black) children to z, and make w and z be the two children of v

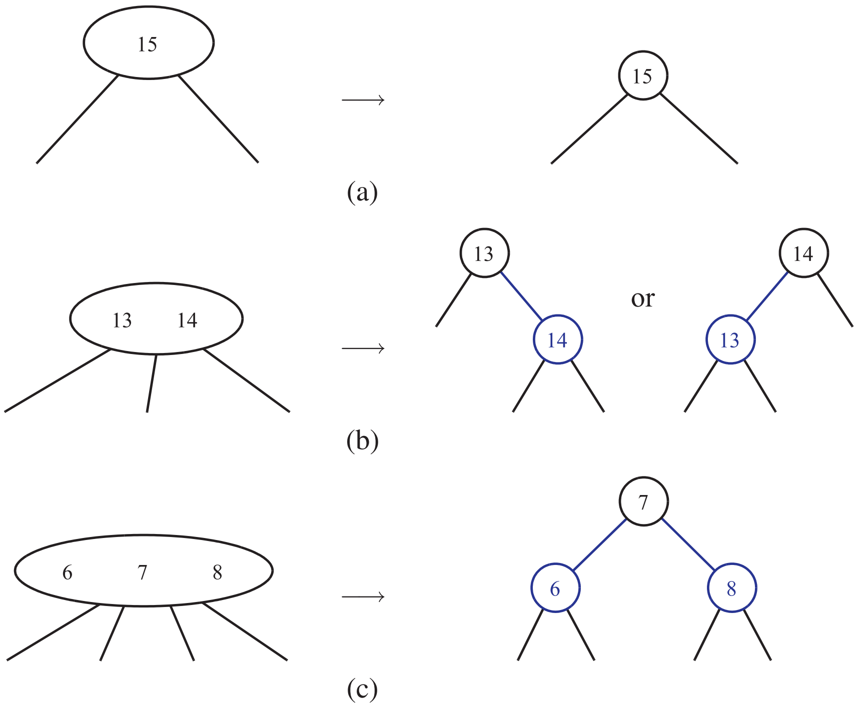


Figure 10.28: Correspondence between a $(2,4)$ tree and a red-black tree: (a) 2-node; (b) 3-node; (c) 4-node.

The correspondence between $(2,4)$ trees and red-black trees provides important intuition that we use in our discussion of how to perform updates in red-black trees. In fact, the update algorithms for red-black trees are mysteriously complex without this intuition.

Proposition 10.9: *The height of a red-black tree storing n entries is $O(\log n)$.*

Justification: Let T be a red-black tree storing n entries, and let h be the height of T. We justify this proposition by establishing the following fact

$$\log(n+1) \leq h \leq 2\log(n+1).$$

Let d be the common black depth of all the external nodes of T. Let T' be the $(2,4)$ tree associated with T, and let h' be the height of T'. Because of the correspondence between red-black trees and $(2,4)$ trees, we know that $h' = d$. Hence, by Proposition 10.8, $d = h' \leq \log(n+1)$. By the internal node property, $h \leq 2d$. Thus, we obtain $h \leq 2\log(n+1)$. The other inequality, $\log(n+1) \leq h$, follows from Proposition 7.10 and the fact that T has n internal nodes. ■

We assume that a red-black tree is realized with a linked structure for binary trees (Section 7.3.4), in which we store a map entry and a color indicator at each node. Thus, the space requirement for storing n keys is $O(n)$. The algorithm for searching in a red-black tree T is the same as that for a standard binary search tree (Section 10.1). Thus, searching in a red-black tree takes $O(\log n)$ time.

10.5.1 Update Operations

Performing the update operations in a red-black tree is similar to that of a binary search tree, except that we must additionally restore the color properties.

Insertion

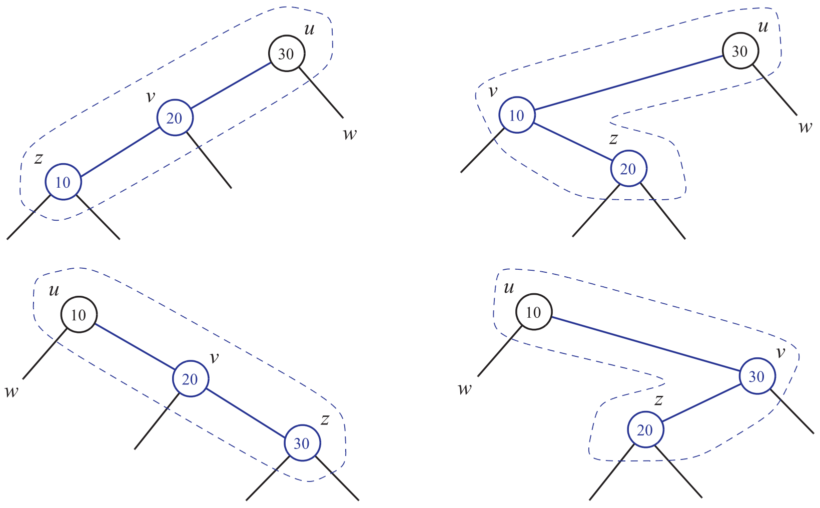
Now consider the insertion of an entry with key k into a red-black tree T, keeping in mind the correspondence between T and its associated $(2,4)$ tree T' and the insertion algorithm for T'. The algorithm initially proceeds as in a binary search tree (Section 10.1.2). Namely, we search for k in T until we reach an external node of T, and we replace this node with an internal node z, storing (k,x) and having two external-node children. If z is the root of T, we color z black, else we color z red. We also color the children of z black. This action corresponds to inserting (k,x) into a node of the $(2,4)$ tree T' with external children. In addition, this action preserves the root, external, and depth properties of T, but it may violate the internal property. Indeed, if z is not the root of T and the parent v of z is red, then we have a parent and a child (namely, v and z) that are both red. Note that by the root property, v cannot be the root of T, and by the internal property (which was previously satisfied), the parent u of v must be black. Since z and its parent are red, but z's grandparent u is black, we call this violation of the internal property a ***double red*** at node z.

To remedy a double red, we consider two cases.

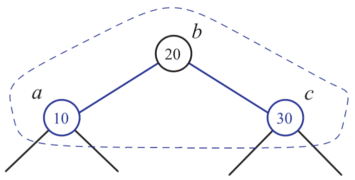
Case 1: ***The Sibling* w *of* v *is Black.*** (See Figure 10.29.) In this case, the double red denotes the fact that we have created in our red-black tree T a malformed replacement for a corresponding 4-node of the $(2,4)$ tree T', which has as its children the four black children of u, v, and z. Our malformed replacement has one red node (v) that is the parent of another red node (z), while we want it to have the two red nodes as siblings instead. To fix this problem, we perform a ***trinode restructuring*** of T. The trinode restructuring is done by the operation restructure(z), which consists of the following steps (see again Figure 10.29; this operation is also discussed in Section 10.2):

- Take node z, its parent v, and grandparent u, and temporarily relabel them as a, b, and c, in left-to-right order, so that a, b, and c will be visited in this order by an inorder tree traversal.
- Replace the grandparent u with the node labeled b, and make nodes a and c the children of b, keeping inorder relationships unchanged.

After performing the restructure(z) operation, we color b black and we color a and c red. Thus, the restructuring eliminates the double red problem.



(a)



(b)

Figure 10.29: Restructuring a red-black tree to remedy a double red: (a) the four configurations for u, v, and z before restructuring; (b) after restructuring.

Case 2: *The Sibling* w *of* v *is Red.* (See Figure 10.30.) In this case, the double red denotes an overflow in the corresponding $(2,4)$ tree T. To fix the problem, we perform the equivalent of a split operation. Namely, we do a ***recoloring***: we color v and w black and their parent u red (unless u is the root, in which case, it is colored black). It is possible that, after such a recoloring, the double red problem reappears, although higher up in the tree T, since u may have a red parent. If the double red problem reappears at u, then we repeat the consideration of the two cases at u. Thus, a recoloring either eliminates the double red problem at node z, or propagates it to the grandparent u of z. We continue going up T performing recolorings until we finally resolve the double red problem (with either a final recoloring or a trinode restructuring). Thus, the number of recolorings caused by an insertion is no more than half the height of tree T, that is, no more than $\log(n+1)$ by Proposition 10.9.

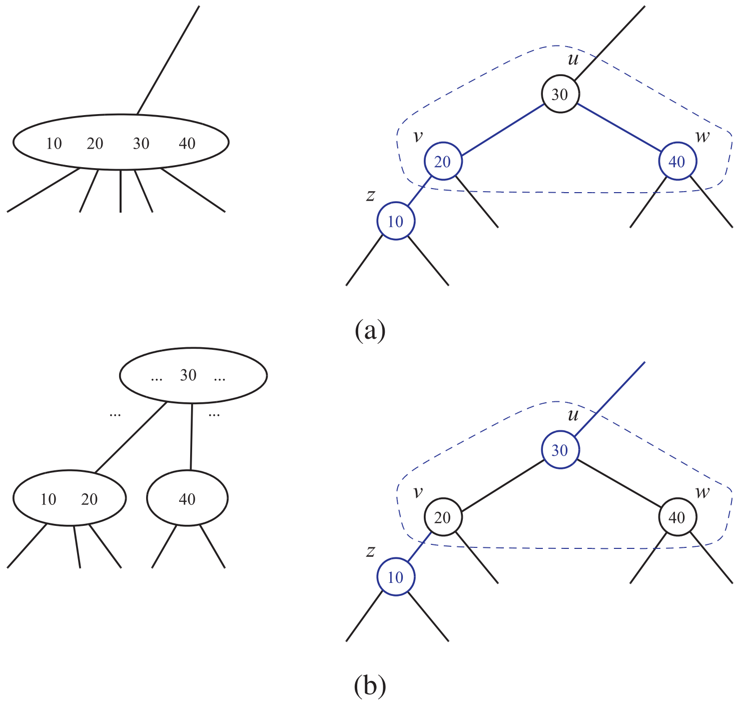


Figure 10.30: Recoloring to remedy the double red problem: (a) before recoloring and the corresponding 5-node in the associated $(2,4)$ tree before the split; (b) after the recoloring (and corresponding nodes in the associated $(2,4)$ tree after the split).

Figures 10.31 and 10.32 show a sequence of insertion operations in a red-black tree.

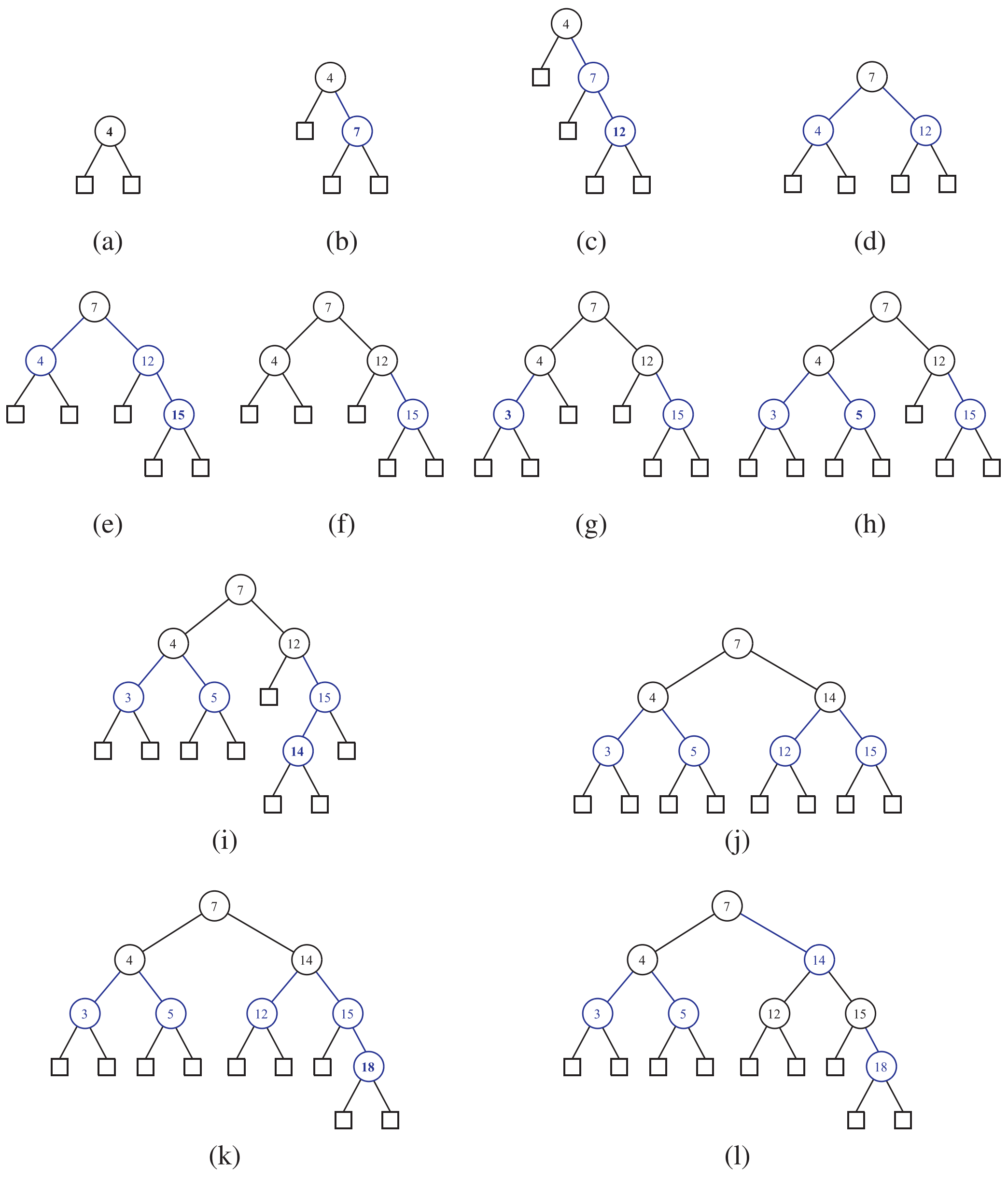


Figure 10.31: A sequence of insertions in a red-black tree: (a) initial tree; (b) insertion of 7; (c) insertion of 12, which causes a double red; (d) after restructuring; (e) insertion of 15, which causes a double red; (f) after recoloring (the root remains black); (g) insertion of 3; (h) insertion of 5; (i) insertion of 14, which causes a double red; (j) after restructuring; (k) insertion of 18, which causes a double red; (l) after recoloring. (Continues in Figure 10.32.)

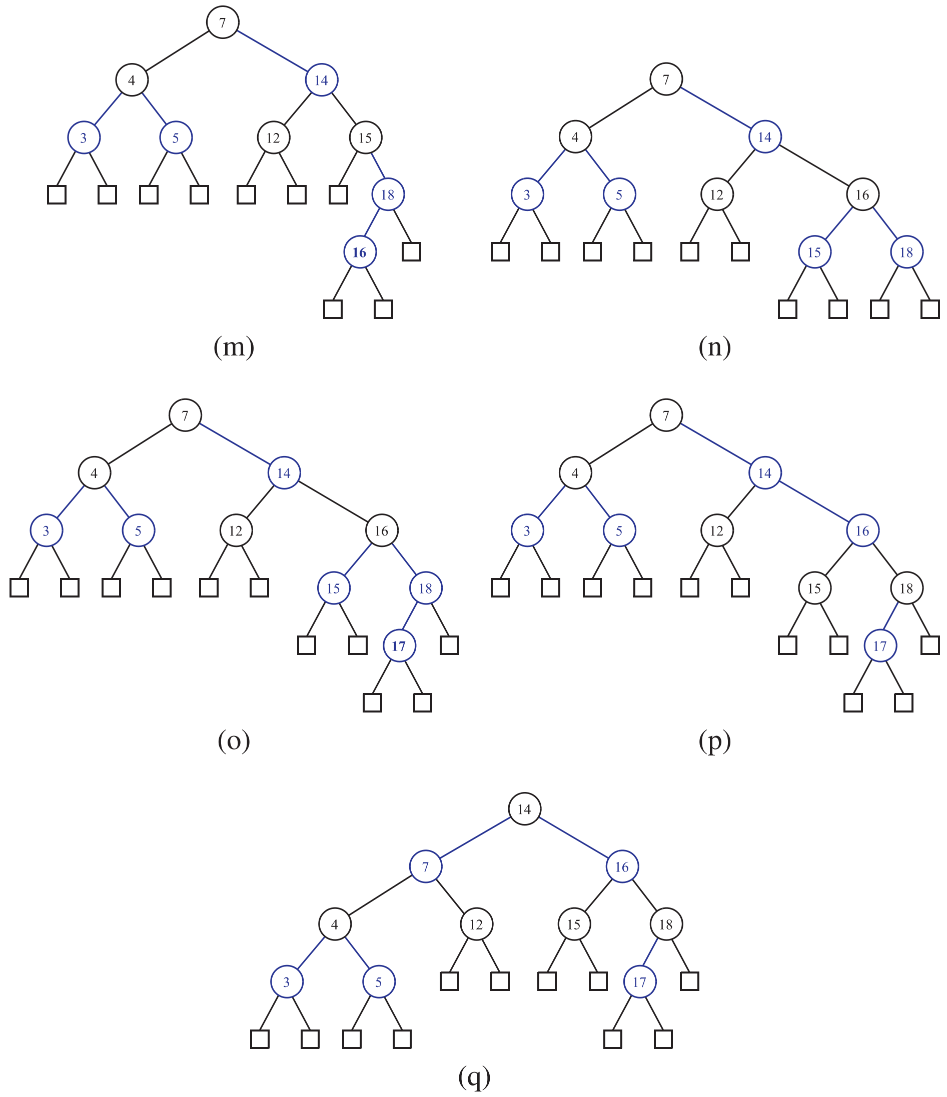


Figure 10.32: A sequence of insertions in a red-black tree: (m) insertion of 16, which causes a double red; (n) after restructuring; (o) insertion of 17, which causes a double red; (p) after recoloring there is again a double red, to be handled by a restructuring; (q) after restructuring. (Continued from Figure 10.31.)

The cases for insertion imply an interesting property for red-black trees. Namely, since the Case 1 action eliminates the double-red problem with a single trinode restructuring and the Case 2 action performs no restructuring operations, at most one restructuring is needed in a red-black tree insertion. By the above analysis and the fact that a restructuring or recoloring takes $O(1)$ time, we have the following.

Proposition 10.10: *The insertion of a key-value entry in a red-black tree storing n entries can be done in $O(\log n)$ time and requires $O(\log n)$ recolorings and one trinode restructuring (a* restructure *operation).*

Removal

Suppose now that we are asked to remove an entry with key k from a red-black tree T. Removing such an entry initially proceeds like a binary search tree (Section 10.1.2). First, we search for a node u storing such an entry. If node u does not have an external child, we find the internal node v following u in the inorder traversal of T, move the entry at v to u, and perform the removal at v. Thus, we may consider only the removal of an entry with key k stored at a node v with an external child w. Also, as we did for insertions, we keep in mind the correspondence between red-black tree T and its associated $(2,4)$ tree T' (and the removal algorithm for T').

To remove the entry with key k from a node v of T with an external child w we proceed as follows. Let r be the sibling of w and x be the parent of v. We remove nodes v and w, and make r a child of x. If v was red (hence r is black) or r is red (hence v was black), we color r black and we are done. If, instead, r is black and v was black, then, to preserve the depth property, we give r a fictitious ***double black*** color. We now have a color violation, called the double black problem. A double black in T denotes an underflow in the corresponding $(2,4)$ tree T'. Recall that x is the parent of the double black node r. To remedy the double-black problem at r, we consider three cases.

Case 1: ***The Sibling* y *of* r *is Black and Has a Red Child* z.** (See Figure 10.33.) Resolving this case corresponds to a transfer operation in the $(2,4)$ tree T'. We perform a ***trinode restructuring*** by means of operation restructure(z). Recall that the operation restructure(z) takes the node z, its parent y, and grandparent x, labels them temporarily left to right as a, b, and c, and replaces x with the node labeled b, making it the parent of the other two. (See the description of restructure in Section 10.2.) We color a and c black, give b the former color of x, and color r black. This trinode restructuring eliminates the double black problem. Hence, at most one restructuring is performed in a removal operation in this case.

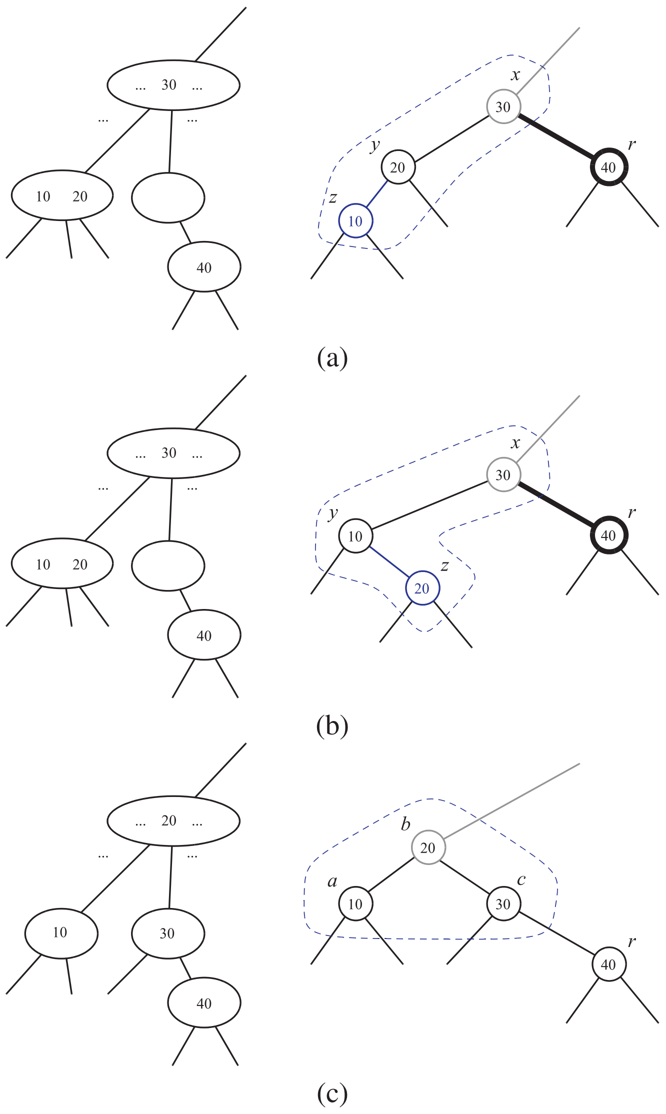


Figure 10.33: Restructuring of a red-black tree to remedy the double black problem: (a) and (b) configurations before the restructuring, where r is a right child and the associated nodes in the corresponding $(2,4)$ tree before the transfer (two other symmetric configurations where r is a left child are possible); (c) configuration after the restructuring and the associated nodes in the corresponding $(2,4)$ tree after the transfer. The grey color for node x in parts (a) and (b) and for node b in part (c) denotes the fact that this node may be colored either red or black.

Case 2: ***The Sibling* y *of* r *is Black and Both Children of* y *Are Black*.** (See Figures 10.34 and 10.35.) Resolving this case corresponds to a fusion operation in the corresponding $(2,4)$ tree T'. We do a ***recoloring***; we color r black, we color y red, and, if x is red, we color it black (Figure 10.34); otherwise, we color x ***double black*** (Figure 10.35). Hence, after this recoloring, the double black problem may reappear at the parent x of r. (See Figure 10.35.) That is, this recoloring either eliminates the double black problem or propagates it into the parent of the current node. We then repeat a consideration of these three cases at the parent. Thus, since Case 1 performs a trinode restructuring operation and stops (and, as we will soon see, Case 3 is similar), the number of recolorings caused by a removal is no more than $\log(n+1)$.

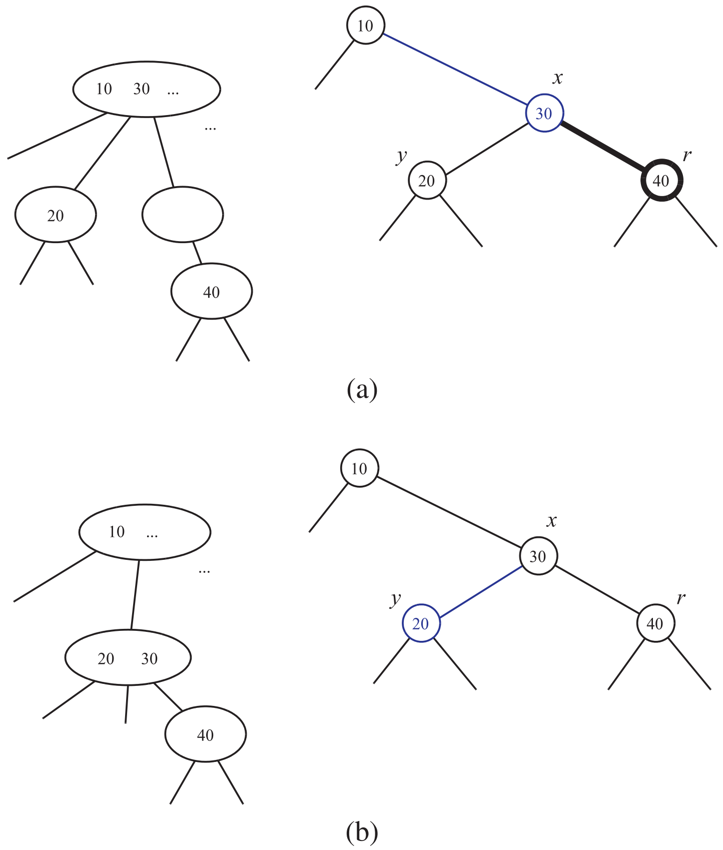
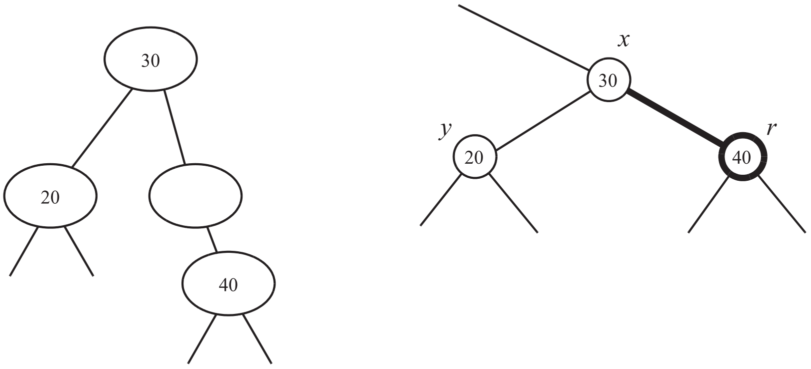
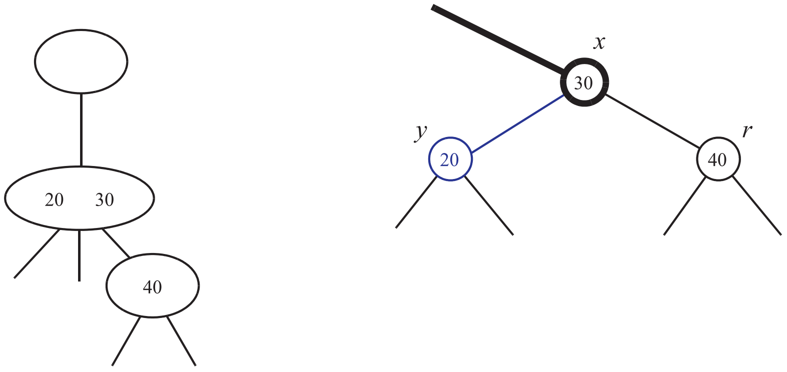


Figure 10.34: Recoloring of a red-black tree that fixes the double black problem: (a) before the recoloring and corresponding nodes in the associated $(2,4)$ tree before the fusion (other similar configurations are possible); (b) after the recoloring and corresponding nodes in the associated $(2,4)$ tree after the fusion.



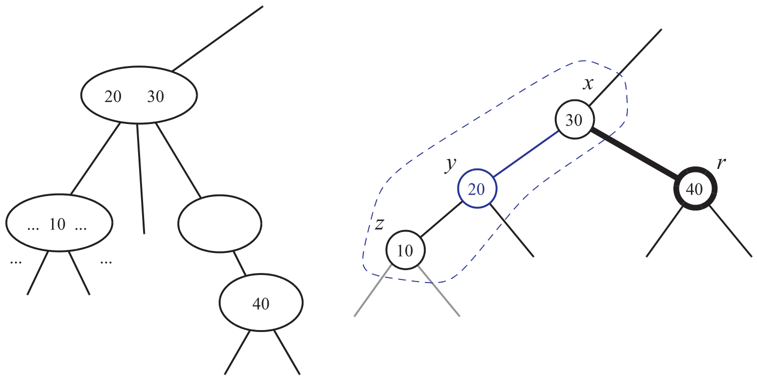
(a)



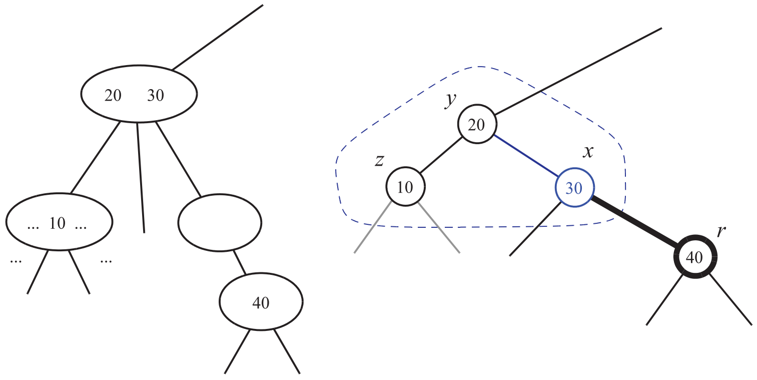
(b)

Figure 10.35: Recoloring of a red-black tree that propagates the double black problem: (a) configuration before the recoloring and corresponding nodes in the associated $(2,4)$ tree before the fusion (other similar configurations are possible); (b) configuration after the recoloring and corresponding nodes in the associated $(2,4)$ tree after the fusion.

Case 3: ***The Sibling* y *of* r *Is Red.*** (See Figure 10.36.) In this case, we perform an ***adjustment*** operation, as follows. If y is the right child of x, let z be the right child of y; otherwise, let z be the left child of y. Execute the trinode restructuring operation restructure(z), which makes y the parent of x. Color y black and x red. An adjustment corresponds to choosing a different representation of a 3-node in the $(2,4)$ tree T'. After the adjustment operation, the sibling of r is black, and either Case 1 or Case 2 applies, with a different meaning of x and y. Note that if Case 2 applies, the double-black problem cannot reappear. Thus, to complete Case 3 we make one more application of either Case 1 or Case 2 above and we are done. Therefore, at most one adjustment is performed in a removal operation.



(a)



(b)

Figure 10.36: Adjustment of a red-black tree in the presence of a double black problem: (a) configuration before the adjustment and corresponding nodes in the associated $(2,4)$ tree (a symmetric configuration is possible); (b) configuration after the adjustment with the same corresponding nodes in the associated $(2,4)$ tree.

From the above algorithm description, we see that the tree updating needed after a removal involves an upward march in the tree T, while performing at most a constant amount of work (in a restructuring, recoloring, or adjustment) per node. Thus, since any changes we make at a node in T during this upward march takes $O(1)$ time (because it affects a constant number of nodes), we have the following.

Proposition 10.11: *The algorithm for removing an entry from a red-black tree with n entries takes $O(\log n)$ time and performs $O(\log n)$ recolorings and at most one adjustment plus one additional trinode restructuring. Thus, it performs at most two* restructure *operations.*

In Figures 10.37 and 10.38, we show a sequence of removal operations on a red-black tree. We illustrate Case 1 restructurings in Figure 10.37(c) and (d). We illustrate Case 2 recolorings at several places in Figures 10.37 and 10.38. Finally, in Figure 10.38(i) and (j), we show an example of a Case 3 adjustment.

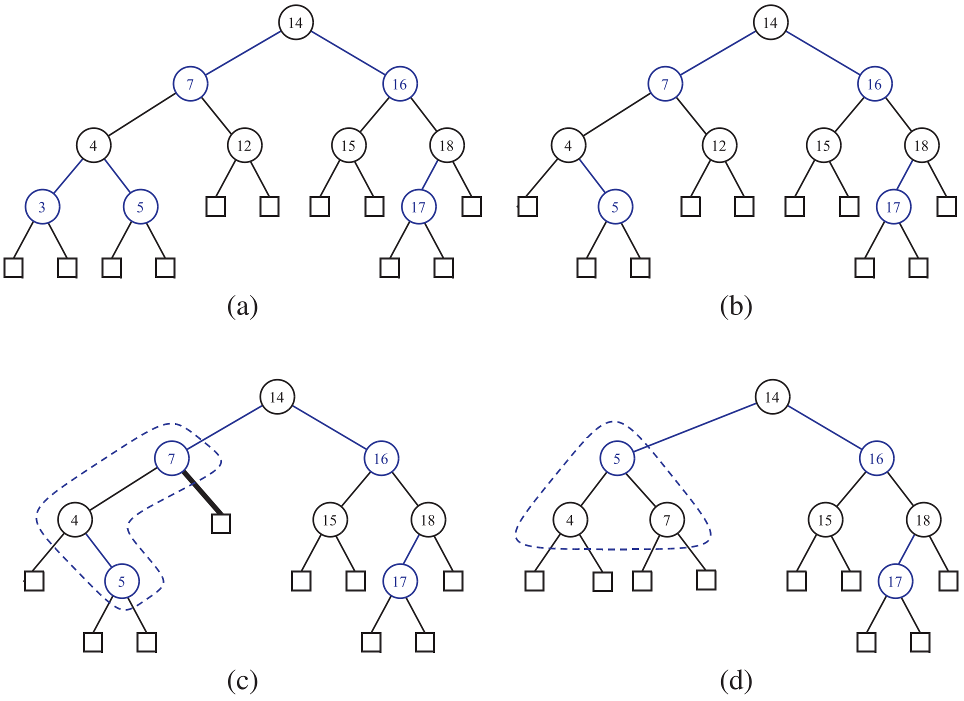


Figure 10.37: Sequence of removals from a red-black tree: (a) initial tree; (b) removal of 3; (c) removal of 12, causing a double black (handled by restructuring); (d) after restructuring. (Continues in Figure 10.38.)

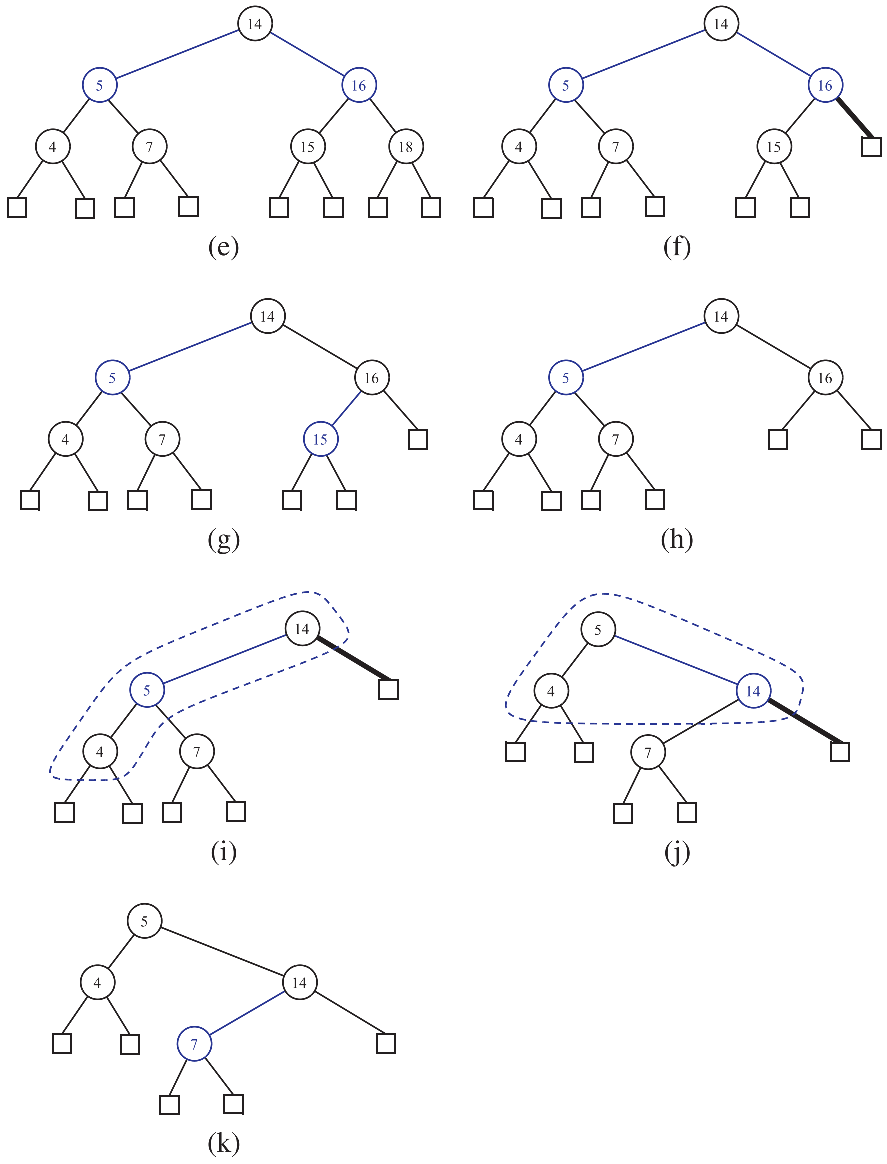


Figure 10.38: Sequence of removals in a red-black tree : (e) removal of 17; (f) removal of 18, causing a double black (handled by recoloring); (g) after recoloring; (h) removal of 15; (i) removal of 16, causing a double black (handled by an adjustment); (j) after the adjustment the double black needs to be handled by a recoloring; (k) after the recoloring. (Continued from Figure 10.37.)

Performance of Red-Black Trees

Table 10.4 summarizes the running times of the main operations of a map realized by means of a red-black tree. We illustrate the justification for these bounds in Figure 10.39.

Operation	*Time*
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

Table 10.4: Performance of an n-entry map realized by a red-black tree. The space usage is $O(n)$.

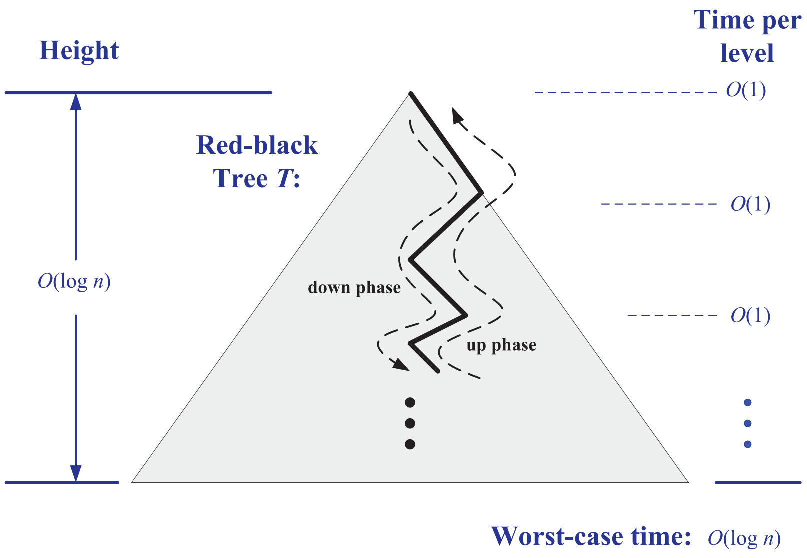


Figure 10.39: The running time of searches and updates in a red-black tree. The time performance is $O(1)$ per level, broken into a down phase, which typically involves searching, and an up phase, which typically involves recolorings and performing local trinode restructurings (rotations).

Thus, a red-black tree achieves logarithmic worst-case running times for both searching and updating in a map. The red-black tree data structure is slightly more complicated than its corresponding $(2,4)$ tree. Even so, a red-black tree has a conceptual advantage that only a constant number of trinode restructurings are ever needed to restore the balance in a red-black tree after an update.

10.5.2 C++ Implementation of a Red-Black Tree

In this section, we discuss a C++ implementation of the dictionary ADT by means of a red-black tree. It is interesting to note that the C++ Standard Template Library uses a red-black tree in its implementation of its classes map and multimap. The difference between the two is similar to the difference between our map and dictionary ADTs. The STL map class does not allow entries with duplicate keys, whereas the STL multimap does. There is a significant difference, however, in the behavior of the map's insert(k,x) function and our map's put(k,x) function. If the key k is not present, both functions insert the new entry (k,x) in the map. If the key is already present, the STL map simply ignores the request, and the current entry is unchanged. In contrast, our put function replaces the existing value with the new value x. The implementation presented in this section allows for multiple keys.

We present the major portions of the implementation in this section. To keep the presentation concise, we have omitted the implementations of a number of simpler utility functions.

We begin by presenting the enhanced entry class, called RBEntry. It is derived from the entry class of Code Fragment 10.3. It inherits the key and value members, and it defines a member variable *col*, which stores the color of the node. The color is either RED or BLACK. It provides member functions for accessing and setting this value. These functions have been protected, so a user cannot access them, but RBTree can.

```
enum Color {RED, BLACK};                          // node colors

template <typename E>
class RBEntry : public E {                        // a red-black entry
private:
  Color col;                                      // node color
protected:                                        // local types
  typedef typename E::Key K;                      // key type
  typedef typename E::Value V;                    // value type
  Color color() const { return col; }             // get color
  bool isRed() const { return col == RED; }
  bool isBlack() const { return col == BLACK; }
  void setColor(Color c) { col = c; }
public:                                           // public functions
  RBEntry(const K& k = K(), const V& v = V()) // constructor
    : E(k,v), col(BLACK) { }
  friend class RBTree<E>;                         // allow RBTree access
};
```

Code Fragment 10.19: A key-value entry for class RBTree, containing the associated node's color.

In Code Fragment 10.20, we present the class definition for RBTree. The declaration is almost entirely analogous to that of AVLTree, except that the utility functions used to maintain the structure are different. We have chosen to present only the two most interesting utility functions, remedyDoubleRed and remedyDoubleBlack. The meanings of most of the omitted utilities are easy to infer. (For example hasTwoExternalChildren(v) determines whether a node v has two external children.)

```
template <typename E>                                // a red-black tree
class RBTree : public SearchTree< RBEntry<E> > {
public:                                              // public types
  typedef RBEntry<E> RBEntry;                        // an entry
  typedef typename SearchTree<RBEntry>::Iterator Iterator; // an iterator
protected:                                           // local types
  typedef typename RBEntry::Key K;                   // a key
  typedef typename RBEntry::Value V;                 // a value
  typedef SearchTree<RBEntry> ST;                    // a search tree
  typedef typename ST::TPos TPos;                    // a tree position
public:                                              // public functions
  RBTree();                                          // constructor
  Iterator insert(const K& k, const V& x);           // insert (k,x)
  void erase(const K& k) throw(NonexistentElement); // remove key k entry
  void erase(const Iterator& p);                     // remove entry at p
protected:                                           // utility functions
  void remedyDoubleRed(const TPos& z);               // fix double-red z
  void remedyDoubleBlack(const TPos& r);             // fix double-black r
  // ...(other utilities omitted)
};
```

Code Fragment 10.20: Class RBTree, which implements a dictionary ADT using a red-black tree.

We first discuss the implementation of the function insert(k,x), which is given in Code Fragment 10.21. We invoke the inserter utility function of SearchTree, which returns the position of the inserted node. If this node is the root of the search tree, we set its color to black. Otherwise, we set its color to red and check whether restructuring is needed by invoking remedyDoubleRed.

This latter utility performs the necessary checks and restructuring presented in the discussion of insertion in Section 10.5.1. Let z denote the location of the newly inserted node. If both z and its parent are red, we need to remedy the situation. To do so, we consider two cases. Let v denote z's parent and let w be v's sibling. If w is black, we fall under Case 1 of the insertion update procedure. We apply restructuring at z. The top vertex of the resulting subtree, denoted by v, is set to black, and its two children are set to red.

On the other hand, if w is red, then we fall under Case 2 of the update procedure.

We resolve the situation by coloring both v and its sibling w black. If their common parent is not the root, we set its color to red. This may induce another double-red problem at v's parent u, so we invoke the function recursively on u.

```
/* RBTree⟨E⟩ :: */                                  // insert (k,x)
  Iterator insert(const K& k, const V& x) {
    TPos v = inserter(k, x);                        // insert in base tree
    if (v == ST::root())
      setBlack(v);                                  // root is always black
    else {
      setRed(v);
      remedyDoubleRed(v);                           // rebalance if needed
    }
    return Iterator(v);
  }

/* RBTree⟨E⟩ :: */                                  // fix double-red z
  void remedyDoubleRed(const TPos& z) {
    TPos v = z.parent();                            // v is z's parent
    if (v == ST::root() || v->isBlack()) return;    // v is black, all ok
                                                    // z, v are double-red
    if (sibling(v)->isBlack()) {                    // Case 1: restructuring
      v = restructure(z);
      setBlack(v);                                  // top vertex now black
      setRed(v.left()); setRed(v.right());          // set children red
    }
    else {                                          // Case 2: recoloring
      setBlack(v); setBlack(sibling(v));            // set v and sibling black
      TPos u = v.parent();                          // u is v's parent
      if (u == ST::root()) return;
      setRed(u);                                    // make u red
      remedyDoubleRed(u);                           // may need to fix u now
    }
  }
```

Code Fragment 10.21: The functions related to insertion for class RBTree. The function insert invokes the inserter utility function, which was given in Code Fragment 10.10.

Finally, in Code Fragment 10.22, we present the implementation of the removal function for the red-black tree. (We have omitted the simpler iterator-based erase function.) The removal follows the process discussed in Section 10.5.1. We first search for the key to be removed, and generate an exception if it is not found. Otherwise, we invoke the eraser utility of class SearchTree, which returns the position of the node r that replaced the deleted node. If either r or its former parent was red, we color r black and we are done. Otherwise, we face a potential double-black problem. We handle this by invoking the function remedyDoubleBlack.

```
/* RBTree⟨E⟩ :: */                                    // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos u = finder(k, ST::root());                     // find the node
    if (Iterator(u) == ST::end())
      throw NonexistentElement("Erase of nonexistent");
    TPos r = eraser(u);                                 // remove u
    if (r == ST::root() || r->isRed() || wasParentRed(r))
      setBlack(r);                                      // fix by color change
    else                                                // r, parent both black
      remedyDoubleBlack(r);                             // fix double-black r
  }

/* RBTree⟨E⟩ :: */                                    // fix double-black r
  void remedyDoubleBlack(const TPos& r) {
    TPos x = r.parent();                                // r's parent
    TPos y = sibling(r);                                // r's sibling
    if (y->isBlack()) {
      if (y.left()->isRed() || y.right()->isRed()) {    // Case 1: restructuring
                                                        // z is y's red child
        TPos z = (y.left()->isRed() ? y.left() : y.right());
        Color topColor = x->color();                    // save top vertex color
        z = restructure(z);                             // restructure x,y,z
        setColor(z, topColor);                          // give z saved color
        setBlack(r);                                    // set r black
        setBlack(z.left()); setBlack(z.right());        // set z's children black
      }
      else {                                            // Case 2: recoloring
        setBlack(r); setRed(y);                         // r=black, y=red
        if (x->isBlack() && !(x == ST::root()))
        remedyDoubleBlack(x);                           // fix double-black x
        setBlack(x);
      }
    }
    else {                                              // Case 3: adjustment
      TPos z = (y == x.right() ? y.right() : y.left()); // grandchild on y's side
      restructure(z);                                   // restructure x,y,z
      setBlack(y); setRed(x);                           // y=black, x=red
      remedyDoubleBlack(r);                             // fix r by Case 1 or 2
    }
  }
```

Code Fragment 10.22: The functions related to removal for class RBTree. The function erase invokes the eraser utility function, which was given in Code Fragment 10.11.

10.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-10.1 If we insert the entries $(1,A)$, $(2,B)$, $(3,C)$, $(4,D)$, and $(5,E)$, in this order, into an initially empty binary search tree, what will it look like?

R-10.2 We defined a binary search tree so that keys equal to a node's key can be in either the left or right subtree of that node. Suppose we change the definition so that we restrict equal keys to the right subtree. What must a subtree of a binary search tree containing only equal keys look like in this case?

R-10.3 Insert, into an empty binary search tree, entries with keys 30, 40, 24, 58, 48, 26, 11, 13 (in this order). Draw the tree after each insertion.

R-10.4 How many different binary search trees can store the keys $\{1,2,3\}$?

R-10.5 Jack claims that the order in which a fixed set of entries is inserted into a binary search tree does not matter—the same tree results every time. Give a small example that proves he is wrong.

R-10.6 Rose claims that the order in which a fixed set of entries is inserted into an AVL tree does not matter—the same AVL tree results every time. Give a small example that proves she is wrong.

R-10.7 Are the rotations in Figures 10.9 and 10.11 single or double rotations?

R-10.8 Draw the AVL tree resulting from the insertion of an entry with key 52 into the AVL tree of Figure 10.11(b).

R-10.9 Draw the AVL tree resulting from the removal of the entry with key 62 from the AVL tree of Figure 10.11(b).

R-10.10 Explain why performing a rotation in an n-node binary tree represented using a vector takes $\Omega(n)$ time.

R-10.11 Is the search tree of Figure 10.1(a) a $(2,4)$ tree? Why or why not?

R-10.12 An alternative way of performing a split at a node v in a $(2,4)$ tree is to partition v into v' and v'', with v' being a 2-node and v'' a 3-node. Which of the keys k_1, k_2, k_3, or k_4 do we store at v's parent in this case? Why?

R-10.13 Cal claims that a $(2,4)$ tree storing a set of entries will always have the same structure, regardless of the order in which the entries are inserted. Show that he is wrong.

R-10.14 Draw four different red-black trees that correspond to the same $(2,4)$ tree.

R-10.15 Consider the set of keys $K = \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$.

a. Draw a $(2,4)$ tree storing K as its keys using the fewest number of nodes.
b. Draw a $(2,4)$ tree storing K as its keys using the maximum number of nodes.

R-10.16 Consider the sequence of keys $(5,16,22,45,2,10,18,30,50,12,1)$. Draw the result of inserting entries with these keys (in the given order) into

a. An initially empty $(2,4)$ tree.
b. An initially empty red-black tree.

R-10.17 For the following statements about red-black trees, provide a justification for each true statement and a counterexample for each false one.

a. A subtree of a red-black tree is itself a red-black tree.
b. The sibling of an external node is either external or it is red.
c. There is a unique $(2,4)$ tree associated with a given red-black tree.
d. There is a unique red-black tree associated with a given $(2,4)$ tree.

R-10.18 Draw an example red-black tree that is not an AVL tree.

R-10.19 Consider a tree T storing 100,000 entries. What is the worst-case height of T in the following cases?

a. T is an AVL tree.
b. T is a $(2,4)$ tree.
c. T is a red-black tree.
d. T is a splay tree.
e. T is a binary search tree.

R-10.20 Perform the following sequence of operations in an initially empty splay tree and draw the tree after each set of operations.

a. Insert keys 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in this order.
b. Search for keys 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, in this order.
c. Delete keys 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in this order.

R-10.21 What does a splay tree look like if its entries are accessed in increasing order by their keys?

R-10.22 Explain how to use an AVL tree or a red-black tree to sort n comparable elements in $O(n \log n)$ time in the worst case.

R-10.23 Can we use a splay tree to sort n comparable elements in $O(n \log n)$ time in the ***worst case***? Why or why not?

R-10.24 Explain why you would get the same output in an inorder listing of the entries in a binary search tree, T, independent of whether T is maintained to be an AVL tree, splay tree, or red-black tree.

Creativity

C-10.1 Describe a modification to the binary search tree data structure that would allow you to find the median entry, that is the entry with rank $\lfloor n/2 \rfloor$, in a binary search tree. Describe both the modification and the algorithm for finding the median assuming all keys are distinct.

C-10.2 Design a variation of algorithm TreeSearch for performing the operation findAll(k) in an ordered dictionary implemented with a binary search tree T, and show that it runs in time $O(h+s)$, where h is the height of T and s is the size of the collection returned.

C-10.3 Describe how to perform an operation eraseAll(k), which removes all the entries whose keys equal k in an ordered dictionary implemented with a binary search tree T, and show that this method runs in time $O(h+s)$, where h is the height of T and s is the size of the iterator returned.

C-10.4 Draw a schematic of an AVL tree such that a single erase operation could require $\Omega(\log n)$ trinode restructurings (or rotations) from a leaf to the root in order to restore the height-balance property.

C-10.5 Show how to perform an operation, eraseAll(k), which removes all entries with keys equal to K, in an ordered dictionary implemented with an AVL tree in time $O(s \log n)$, where n is the number of entries in the map and s is the size of the iterator returned.

C-10.6 Describe the changes that would need to be made to the binary search tree implementation given in the book to allow it to be used to support an ordered dictionary, where we allow for different entries with equal keys.

C-10.7 If we maintain a reference to the position of the left-most internal node of an AVL tree, then operation first (Section 9.3) can be performed in $O(1)$ time. Describe how the implementation of the other map functions needs to be modified to maintain a reference to the left-most position.

C-10.8 Show that any n-node binary tree can be converted to any other n-node binary tree using $O(n)$ rotations.

C-10.9 Let M be an ordered map with n entries implemented by means of an AVL tree. Show how to implement the following operation on M in time $O(\log n + s)$, where s is the size of the iterator returned.

findAllInRange(k_1, k_2): Return an iterator of all the entries in M with key k such that $k_1 \leq k \leq k_2$.

C-10.10 Let M be an ordered map with n entries. Show how to modify the AVL tree to implement the following function for M in time $O(\log n)$.

countAllInRange(k_1, k_2): Compute and return the number of entries in M with key k such that $k_1 \leq k \leq k_2$.

C-10.11 Draw a splay tree, T_1, together with the sequence of updates that produced it, and a red-black tree, T_2, on the same set of ten entries, such that a preorder traversal of T_1 would be the same as a preorder traversal of T_2.

C-10.12 Show that the nodes that become unbalanced in an AVL tree during an insert operation may be nonconsecutive on the path from the newly inserted node to the root.

C-10.13 Show that at most one node in an AVL tree becomes unbalanced after operation removeAboveExternal is performed within the execution of a erase map operation.

C-10.14 Show that at most one trinode restructuring operation is needed to restore balance after any insertion in an AVL tree.

C-10.15 Let T and U be $(2,4)$ trees storing n and m entries, respectively, such that all the entries in T have keys less than the keys of all the entries in U. Describe an $O(\log n + \log m)$ time method for ***joining*** T and U into a single tree that stores all the entries in T and U.

C-10.16 Repeat the previous problem for red-black trees T and U.

C-10.17 Justify Proposition 10.7.

C-10.18 The Boolean indicator used to mark nodes in a red-black tree as being "red" or "black" is not strictly needed when we have distinct keys. Describe a scheme for implementing a red-black tree without adding any extra space to standard binary search tree nodes.

C-10.19 Let T be a red-black tree storing n entries, and let k be the key of an entry in T. Show how to construct from T, in $O(\log n)$ time, two red-black trees T' and T'', such that T' contains all the keys of T less than k, and T'' contains all the keys of T greater than k. This operation destroys T.

C-10.20 Show that the nodes of any AVL tree T can be colored "red" and "black" so that T becomes a red-black tree.

C-10.21 The ***mergeable heap*** ADT consists of operations insert(k,x), removeMin$()$, unionWith(h), and min$()$, where the unionWith(h) operation performs a union of the mergeable heap h with the present one, destroying the old versions of both. Describe a concrete implementation of the mergeable heap ADT that achieves $O(\log n)$ performance for all its operations.

C-10.22 Consider a variation of splay trees, called ***half-splay trees***, where splaying a node at depth d stops as soon as the node reaches depth $\lfloor d/2 \rfloor$. Perform an amortized analysis of half-splay trees.

C-10.23 The standard splaying step requires two passes, one downward pass to find the node x to splay, followed by an upward pass to splay the node x. Describe a method for splaying and searching for x in one downward pass. Each substep now requires that you consider the next two nodes in the path down to x, with a possible zig substep performed at the end. Describe how to perform the zig-zig, zig-zag, and zig steps.

C-10.24 Describe a sequence of accesses to an n-node splay tree T, where n is odd, that results in T consisting of a single chain of internal nodes with external node children, such that the internal-node path down T alternates between left children and right children.

C-10.25 Explain how to implement a vector of n elements so that the functions insert and at take $O(\log n)$ time in the worst case.

Projects

P-10.1 Write a program that performs a simple n-body simulation, called "Jumping Leprechauns." This simulation involves n leprechauns, numbered 1 to n. It maintains a gold value g_i for each leprechaun i, which begins with each leprechaun starting out with a million dollars worth of gold, that is, $g_i = 1000000$ for each $i = 1,2,\ldots,n$. In addition, the simulation also maintains, for each leprechaun, i, a place on the horizon, which is represented as a double-precision floating point number, x_i. In each iteration of the simulation, the simulation processes the leprechauns in order. Processing a leprechaun i during this iteration begins by computing a new place on the horizon for i, which is determined by the assignment

$$x_i \leftarrow x_i + rg_i,$$

where r is a random floating-point number between -1 and 1. The leprechaun i then steals half the gold from the nearest leprechauns on either side of him and adds this gold to his gold value, g_i. Write a program that can perform a series of iterations in this simulation for a given number, n, of leprechauns. You must maintain the set of horizon positions using an ordered map data structure described in this chapter.

P-10.2 Extend class BinarySearchTree (Section 10.1.3) to support the functions of the ordered map ADT (see Section 9.3).

P-10.3 Implement a class RestructurableNodeBinaryTree that supports the functions of the binary tree ADT, plus a function restructure for performing a rotation operation. This class is a component of the implementation of an AVL tree given in Section 10.2.2.

P-10.4 Write a C++ class that implements all the functions of the ordered map ADT (see Section 9.3) using an AVL tree.

P-10.5 Write a C++ class that implements all the functions of the ordered map ADT (see Section 9.3) using a $(2,4)$ tree.

P-10.6 Write a C++ class that implements all the functions of the ordered map ADT (see Section 9.3) using a red-black tree.

P-10.7 Form a three-programmer team and have each member implement a map using a different search tree data structure. Perform a cooperative experimental study to compare the speed of these three implementations.

P-10.8 Write a C++ class that can take any red-black tree and convert it into its corresponding $(2,4)$ tree and can take any $(2,4)$ tree and convert it into its corresponding red-black tree.

P-10.9 Implement the map ADT using a splay tree, and compare its performance experimentally with the STL map class, which uses a red-black tree.

P-10.10 Prepare an implementation of splay trees that uses bottom-up splaying as described in this chapter and another that uses top-down splaying as described in Exercise C-10.23. Perform extensive experimental studies to see which implementation is better in practice, if any.

P-10.11 Implement a binary search tree data structure so that it can support the dictionary ADT, where different entries can have equal keys. In addition, implement the functions entrySetPreorder(), entrySetInorder(), and entrySetPostorder(), which produce an iterable collection of the entries in the binary search tree in the same order they would respectively be visited in a preorder, inorder, and postorder traversal of the tree.

Chapter Notes

Some of the data structures discussed in this chapter are extensively covered by Knuth in his *Sorting and Searching* book [60], and by Mehlhorn in [73]. AVL trees are due to Adel'son-Vel'skii and Landis [1], who invented this class of balanced search trees in 1962. Binary search trees, AVL trees, and hashing are described in Knuth's *Sorting and Searching* [60] book. Average-height analyses for binary search trees can be found in the books by Aho, Hopcroft, and Ullman [5] and Cormen, Leiserson, Rivest and Stein [25]. The handbook by Gonnet and Baeza-Yates [37] contains a number of theoretical and experimental comparisons among map implementations. Aho, Hopcroft, and Ullman [4] discuss $(2,3)$ trees, which are similar to $(2,4)$ trees. [9]. Variations and interesting properties of red-black trees are presented in a paper by Guibas and Sedgewick [42]. The reader interested in learning more about different balanced tree data structures is referred to the books by Mehlhorn [73] and Tarjan [95], and the book chapter by Mehlhorn and Tsakalidis [75]. Knuth [60] is excellent additional reading that includes early approaches to balancing trees. Splay trees were invented by Sleator and Tarjan [89] (see also [95]).

This page intentionally left blank