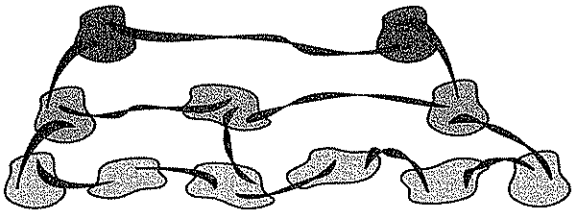
Chapter

# 3 Search Trees and Skip Lists



## Contents

People like choices. We like to have different ways of solving the same problem, so that we can explore different trade-offs and efficiencies. This chapter is devoted to the exploration of different ways of implementing an ordered dictionary. We begin this chapter by discussing binary search trees, and how they support a simple tree-based implementation of an ordered dictionary, but do not guarantee efficient worst-case performance. Nevertheless, they form the basis of many tree-based dictionary implementations, and we discuss several in this chapter. One of the classic implementations is the AVL tree, presented in Section 3.2, which is a binary search tree that achieves logarithmic-time search and update operations.

In Section 3.3, we introduce the concept of bounded-depth trees, which keep all external nodes at the same depth or "pseudo-depth." One such tree is the multi-way search tree, which is an ordered tree where each internal node can store several items and have several children. A multi-way search tree is a generalization of the binary search tree, and like the binary search tree, it can be specialized into an efficient data structure for ordered dictionaries. A specific kind of multi-way search tree discussed in Section 3.3 is the $(2,4)$ tree, which is a bounded-depth search tree in which each internal node stores 1, 2, or 3 keys and has 2, 3, or 4 children, respectively. The advantage of these trees is that they have algorithms for inserting and removing keys that are simple and intuitive. Update operations rearrange a $(2,4)$ tree by means of natural operations that split and merge "nearby" nodes or transfer keys between them. A $(2,4)$ tree storing $n$ items uses $O(n)$ space and supports searches, insertions, and removals in $O(\log n)$ worst-case time. Another kind of bounded-depth search tree studied in this section is the red-black tree. These are binary search trees whose nodes are colored "red" and "black" in such a way that the coloring scheme guarantees each external node is at the same (logarithmic) "black depth." The pseudo-depth notion of black depth results from an illuminating correspondence between red-black and $(2,4)$ trees. Using this correspondence, we motivate and provide intuition for the somewhat more complex algorithms for insertion and removal in red-black trees, which are based on rotations and recolorings. An advantage that a red-black tree achieves over other binary search trees (such as AVL trees) is that it can be restructured after an insertion or removal with only $O(1)$ rotations.

In Section 3.4, we discuss splay trees, which are attractive due to the simplicity of their search and update methods. Splay trees are binary search trees that, after each search, insertion, or deletion, move the node accessed up to the root by means of a carefully choreographed sequence of rotations. This simple "move-to-the-top" heuristic helps this data structure ***adapt*** itself to the kinds of operations being performed. One of the results of this heuristic is that splay trees guarantee that the amortized running time of each dictionary operation is logarithmic.

Finally, in Section 3.5, we discuss skip lists, which are not a tree data structure, but nevertheless have a notion of depth that keeps all elements at logarithmic depth. These structures are randomized, however, so their depth bounds are probabilistic. In particular, we show that with very high probability the height of a skip list storing

$n$ elements is $O(\log n)$. This is admittedly not as strong as a true worst-case bound, but the update operations for skip lists are quite simple and they compare favorably to search trees in practice.

We focus on the practice of implementing binary search trees in Section 3.6, giving Java implementations for both AVL and red-black trees. We highlight how both of these data structures can build upon the tree ADT discussed in Section 2.3.

There are admittedly quite a few kinds of search structures discussed in this chapter, and we recognize that a reader or instructor with limited time might be interested in studying only selected topics. For this reason, we have designed this chapter so that each section can be studied independent of any other section except for the first section, which we present next.

## 3.1 Ordered Dictionaries and Binary Search Trees

In an ordered dictionary, we wish to perform the usual dictionary operations, discussed in Section 2.5.1, such as the operations findElement($k$), insertItem($k,e$), and removeElement($k$), but also maintain an order relation for the keys in our dictionary. We can use a comparator to provide the order relation among keys, and, as we will see, such an ordering helps us to efficiently implement the dictionary ADT. In addition, an ordered dictionary also supports the following methods:

closestKeyBefore($k$): Return the key of the item with largest key less than or equal to $k$.

closestElemBefore($k$): Return the element for the item with largest key less than or equal to $k$.

closestKeyAfter($k$): Return the key of the item with smallest key greater than or equal to $k$.

closestElemAfter($k$): Return the element for the item with smallest key greater than or equal to $k$.

Each of these methods returns the special NO_SUCH_KEY object if no item in the dictionary satisfies the query.

The ordered nature of the above operations makes the use of a log file or a hash table inappropriate for implementing the dictionary, for neither of these data structures maintains any ordering information for the keys in the dictionary. Indeed, hash tables achieve their best search speeds when their keys are distributed almost at random. Thus, we should consider new dictionary implementations when dealing with ordered dictionaries.

Having defined the dictionary abstract data type, let us now look at some simple ways of implementing this ADT.

### 3.1.1 Sorted Tables

If a dictionary $D$ is ordered, we can store its items in a vector $S$ by nondecreasing order of the keys. We specify that $S$ is a vector, rather than a general sequence, for the ordering of the keys in the vector $S$ allows for faster searching than would be possible had $S$ been, say, a linked list. We refer to this ordered vector implementation of a dictionary $D$ as a ***lookup table***. We contrast this implementation with the log file, which uses an unordered sequence to implement the dictionary.

The space requirement of the lookup table is $\Theta(n)$, which is similar to the log file, assuming we grow and shrink the array supporting the vector $S$ to keep the size of this array proportional to the number of items in $S$. Unlike a log file, however, performing updates in a lookup table takes a considerable amount of time. In particular, performing the insertItem$(k,e)$ operation in a lookup table requires $O(n)$ time in the worst case, since we need to shift up all the items in the vector with key greater than $k$ to make room for the new item $(k,e)$. The lookup table implementation is therefore inferior to the log file in terms of the worst-case running times of the dictionary update operations. Nevertheless, we can perform the operation findElement much faster in a sorted lookup table.

#### Binary Search

A significant advantage of using an array-based vector $S$ to implement an ordered dictionary $D$ with $n$ items is that accessing an element of $S$ by its ***rank*** takes $O(1)$ time. We recall from Section 2.2.1 that the rank of an element in a vector is the number of elements preceding it. Thus, the first element in $S$ has rank 0, and the last element has rank $n-1$.

The elements in $S$ are the items of dictionary $D$, and since $S$ is ordered, the item at rank $i$ has a key no smaller than keys of the items at ranks $0,\ldots,i-1$, and no larger than keys of the items at ranks $i+1,\ldots,n-1$. This observation allows us to quickly "home in" on a search key $k$ using a variant of the children's game "high-low." We call an item $I$ of $D$ a ***candidate*** if, at the current stage of the search, we cannot rule out that $I$ has key equal to $k$. The algorithm maintains two parameters, low and high, such that all the candidate items have rank at least low and at most high in $S$. Initially, low $= 0$ and high $= n-1$, and we let key$(i)$ denote the key at rank $i$, which has elem$(i)$ as its element. We then compare $k$ to the key of the median candidate, that is, the item with rank

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor.$$

We consider three cases:

- If $k = \text{key}(\text{mid})$, then we have found the item we were looking for, and the search terminates successfully returning elem(mid).
- If $k < \text{key}(\text{mid})$, then we recur on the first half of the vector, that is, on the range of ranks from low to mid $-1$.
- If $k > \text{key}(\text{mid})$, we recur on the range of ranks from mid $+1$ to high.

This search method is called ***binary search***, and is given in pseudo-code in Algorithm 3.1. Operation findElement($k$) on an $n$-item dictionary implemented with a vector $S$ consists of calling BinarySearch($S, k, 0, n-1$).

**Algorithm** BinarySearch($S, k$, low, high):
*Input:* An ordered vector $S$ storing $n$ items, whose keys are accessed with method key($i$) and whose elements are accessed with method elem($i$); a search key $k$; and integers low and high
*Output:* An element of $S$ with key $k$ and rank between low and high, if such an element exists, and otherwise the special element NO_SUCH_KEY

```
if low > high then
  return NO_SUCH_KEY
else
  mid ← ⌊(low + high)/2⌋
  if k = key(mid) then
    return elem(mid)
  else if k < key(mid) then
    return BinarySearch(S, k, low, mid − 1)
  else
    return BinarySearch(S, k, mid + 1, high)
```

**Algorithm 3.1:** Binary search in an ordered vector.

We illustrate the binary search algorithm in Figure 3.2.

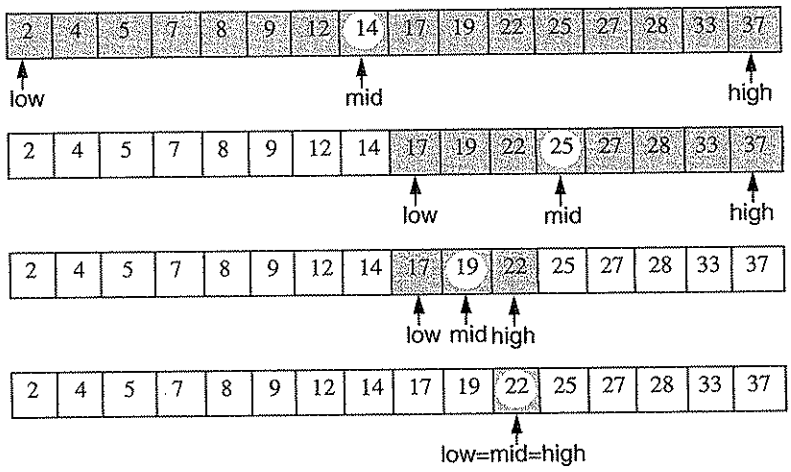



**Figure 3.2:** Example of a binary search to perform operation findElement(22), in a dictionary with integer keys, implemented with an array-based ordered vector. For simplicity, we show the keys stored in the dictionary but not the elements.

Considering the running time of binary search, we observe that a constant number of operations are executed at each recursive call. Hence, the running time is proportional to the number of recursive calls performed. A crucial fact is that with each recursive call the number of candidate items still to be searched in the sequence $S$ is given by the value $\mathsf{high} - \mathsf{low} + 1$. Moreover, the number of remaining candidates is reduced by at least one half with each recursive call. Specifically, from the definition of mid the number of remaining candidates is either

$$(\mathsf{mid} - 1) - \mathsf{low} + 1 = \left\lfloor \frac{\mathsf{low} + \mathsf{high}}{2} \right\rfloor - \mathsf{low} \leq \frac{\mathsf{high} - \mathsf{low} + 1}{2}$$

or

$$\mathsf{high} - (\mathsf{mid} + 1) + 1 = \mathsf{high} - \left\lfloor \frac{\mathsf{low} + \mathsf{high}}{2} \right\rfloor \leq \frac{\mathsf{high} - \mathsf{low} + 1}{2}.$$

Initially, the number of candidate is $n$; after the first call to BinarySearch, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. That is, if we let a function, $T(n)$, represent the running time of this method, then we can characterize the running time of the recursive binary search algorithm as follows:

$$T(n) \leq \begin{cases} b & \text{if } n < 2 \\ T(n/2) + b & \text{else,} \end{cases}$$

where $b$ is a constant. In general, this recurrence equation shows that the number of candidate items remaining after each recursive call is at most $n/2^i$. (We discuss recurrence equations like this one in more detail in Section 5.2.1.) In the worst case (unsuccessful search), the recursive calls stop when there are no more candidate items. Hence, the maximum number of recursive calls performed is the smallest integer $m$ such that $n/2^m < 1$ . In other words (recalling that we omit a logarithm's base when it is 2), $m > \log n$. Thus, we have $m = \lfloor \log n \rfloor + 1$, which implies that $\mathsf{BinarySearch}(S, k, 0, n-1)$ runs in $O(\log n)$ time.

Table 3.3 compares the running times of the methods of a dictionary realized by either a log file or a lookup table. A log file allows for fast insertions but slow searches and removals, whereas a lookup table allows for fast searches but slow insertions and removals.

| Method | Log File | Lookup Table |
|---|---|---|
| findElement | $O(n)$ | $O(\log n)$ |
| insertItem | $O(1)$ | $O(n)$ |
| removeElement | $O(n)$ | $O(n)$ |
| closestKeyBefore | $O(n)$ | $O(\log n)$ |

**Table 3.3:** Comparison of the running times of the primary methods of an ordered dictionary realized by means of a log file or a lookup table. We denote the number of items in the dictionary at the time a method is executed with $n$. The performance of the methods closestElemBefore, closestKeyAfter, closestElemAfter is similar to that of closestKeyBefore.

### 3.1.2 Binary Search Trees

The data structure we discuss in this section, the binary search tree, applies the motivation of the binary search procedure to a tree-based data structure. We define a binary search tree to be a binary tree in which each internal node $v$ stores an element $e$ such that the elements stored in the left subtree of $v$ are less than or equal to $e$, and the elements stored in the right subtree of $v$ are greater than or equal to $e$. Furthermore, let us assume that external nodes store no elements; hence, they could in fact be null or references to a NULL_NODE object.

An inorder traversal of a binary search tree visits the elements stored in such a tree in nondecreasing order. A binary search tree supports searching, where the question asked at each internal node is whether the element at that node is less than, equal to, or larger than the element being searched for.

We can use a binary search tree $T$ to locate an element with a certain value $x$ by traversing down the tree $T$. At each internal node we compare the value of the current node to our search element $x$. If the answer to the question 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 (which is empty), then the search terminates unsuccessfully. (See Figure 3.4.)

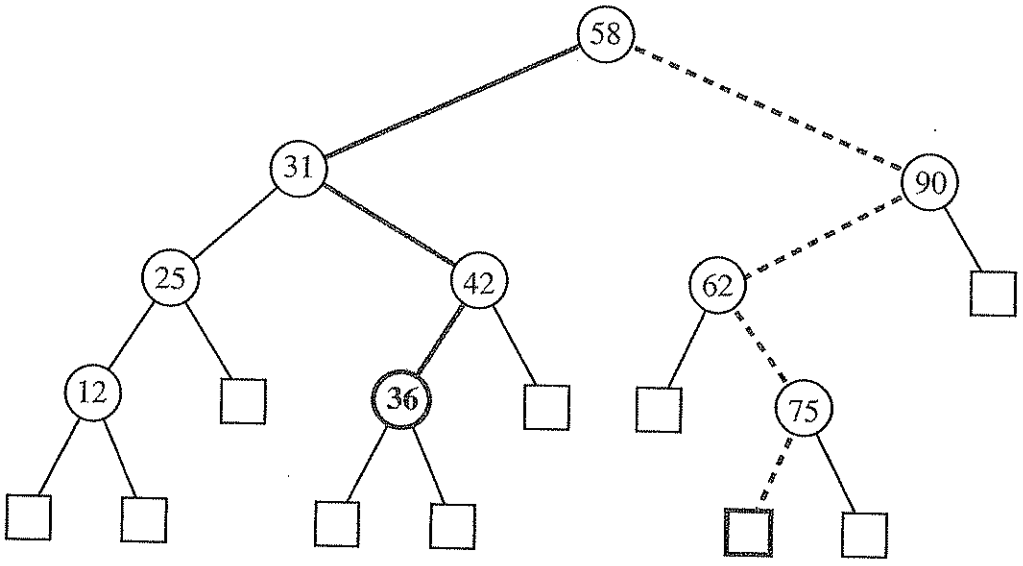



**Figure 3.4:** A binary search tree storing integers. The thick solid path drawn with thick lines is traversed when searching (successfully) for 36. The thick dashed path is traversed when searching (unsuccessfully) for 70.

### 3.1.3 Searching in a Binary Search Tree

Formally, a ***binary search tree*** is a binary tree $T$ in which each internal node $v$ of $T$ stores an item $(k,e)$ of a dictionary $D$, and keys stored at nodes in the left subtree of $v$ are less than or equal to $k$, while keys stored at nodes in the right subtree of $v$ are greater than or equal to $k$.

In Algorithm 3.5, we give a recursive method TreeSearch, based on the above strategy for searching in a binary search tree $T$. Given a search key $k$ and a node $v$ of $T$, method TreeSearch returns a node (position) $w$ of the subtree $T(v)$ of $T$ rooted at $v$, such that one of the following two cases occurs:

- $w$ is an internal node of $T(v)$ that stores key $k$.
- $w$ is an external node of $T(v)$. All the internal nodes of $T(v)$ that precede $w$ in the inorder traversal have keys smaller than $k$, and all the internal nodes of $T(v)$ that follow $w$ in the inorder traversal have keys greater than $k$.

Thus, method findElement$(k)$ can be performed on dictionary $D$ by calling the method TreeSearch$(k,T.\text{root}())$ on $T$. Let $w$ be the node of $T$ returned by this call of the TreeSearch method. If node $w$ is internal, we return the element stored at $w$; otherwise, if $w$ is external, then we return NO_SUCH_KEY.

**Algorithm** TreeSearch$(k,v)$:
*Input:* A search key $k$, and a node $v$ of a binary search tree $T$
*Output:* A node $w$ of the subtree $T(v)$ of $T$ rooted at $v$, such that either $w$ is an internal node storing key $k$ or $w$ is the external node where an item with key $k$ would belong if it existed

```
if v is an external node then
  return v
if k = key(v) then
  return v
else if k < key(v) then
  return TreeSearch(k, T.leftChild(v))
else
  {we know k > key(v)}
  return TreeSearch(k, T.rightChild(v))
```

**Algorithm 3.5:** Recursive search in a binary search tree.

Note that the running time of searching in a binary search tree $T$ is proportional to the height of $T$. Since the height of a tree with $n$ nodes can be as small as $O(\log n)$ or as large as $\Omega(n)$, binary search trees are most efficient when they have small height.

### Analysis of Binary Tree Searching

The formal analysis of the worst-case running time of searching in a binary search tree $T$ is simple. The binary tree search algorithm executes a constant number of primitive operations for each node it traverses in the tree. Each new step in the traversal is made on a child of the previous node. That is, the binary tree search algorithm is performed on the nodes of a path of $T$ that starts from 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, method findElement (or any other standard search operation) runs in $O(h)$ time, where $h$ is the height of the binary search tree $T$ used to implement the dictionary $D$. (See Figure 3.6.)

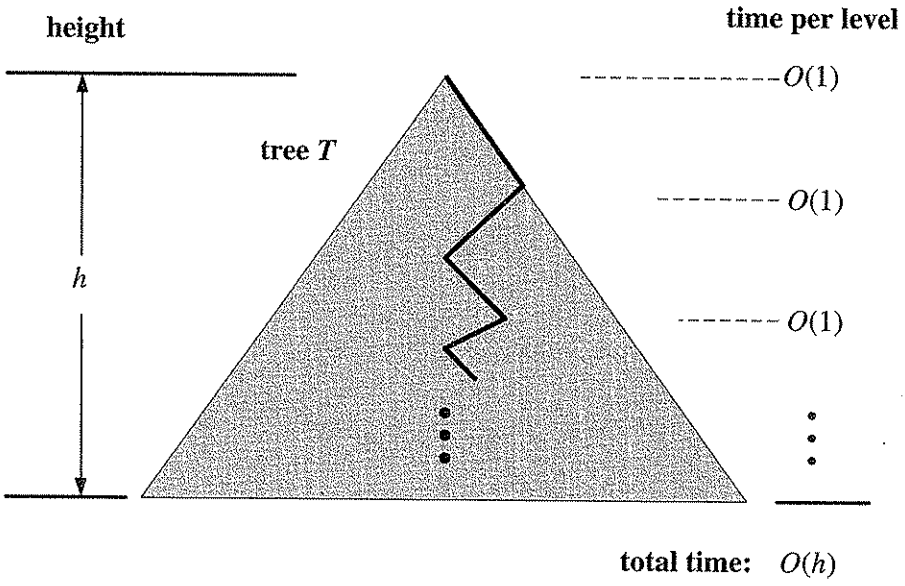



**Figure 3.6:** Illustrating the running time of searching in a binary search tree. The figure uses standard visualization shortcuts of viewing a binary search tree as a big triangle and a path from the root as a zig-zag line.

We can also show that a variation of the above algorithm performs operation findAllElements($k$), which finds all the items in the dictionary with key $k$, in time $O(h+s)$, where $s$ is the number of elements returned. However, this method is slightly more complicated, and the details are left as an exercise (C-3.3).

Admittedly, the height $h$ of $T$ can be as large as $n$, but we expect that it is usually much smaller. Indeed, we will show in subsequent sections in this chapter how to maintain an upper bound of $O(\log n)$ on the height of a search tree $T$. Before we describe such a scheme, however, let us describe implementations for dictionary update methods in a possibly unbalanced binary search tree.

### 3.1.4 Insertion in a Binary Search Tree

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

To perform the operation insertItem$(k,e)$ on a dictionary $D$ implemented with a binary search tree $T$, we start by calling the method TreeSearch$(k,T.\text{root}())$ on $T$. Let $w$ be the node returned by TreeSearch.

- If $w$ is an external node (no item with key $k$ is stored in $T$), we replace $w$ with a new internal node storing the item $(k,e)$ and two external children, by means of operation expandExternal$(w)$ on $T$ (see Section 2.3.3). Note that $w$ is the appropriate place to insert an item with key $k$.
- If $w$ is an internal node (another item with key $k$ is stored at $w$), we call TreeSearch$(k,\text{rightChild}(w))$ (or, equivalently, TreeSearch$(k,\text{leftChild}(w))$) and recursively apply the algorithm to the node returned by TreeSearch.

The above insertion algorithm eventually traces a path from the root of $T$ down to an external node, which gets replaced with a new internal node accommodating the new item. Hence, an insertion adds the new item at the "bottom" of the search tree $T$. An example of insertion into a binary search tree is shown in Figure 3.7.

The analysis of the insertion algorithm is analogous to that for searching. The number of nodes visited is proportional to the height $h$ of $T$ in the worst case. Also, assuming a linked structure implementation for $T$ (see Section 2.3.4), we spend $O(1)$ time at each node visited. Thus, method insertItem runs in $O(h)$ time.

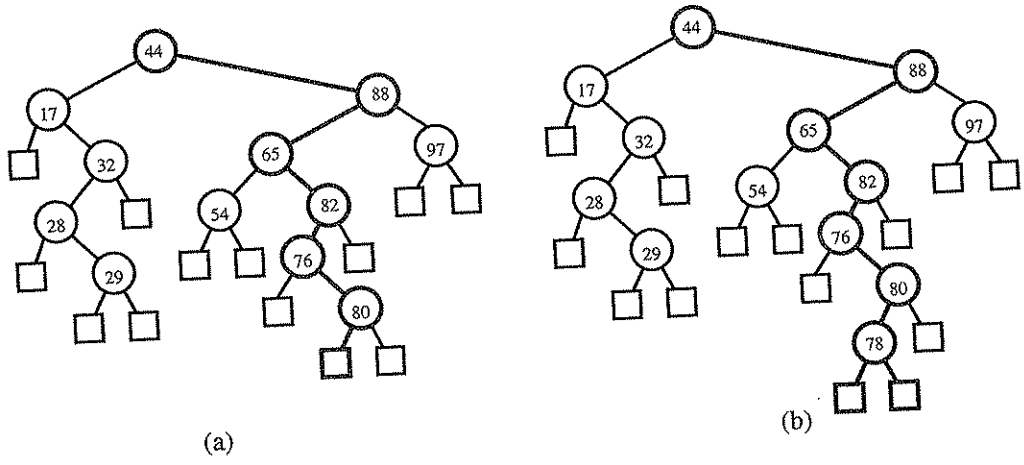



**Figure 3.7:** Insertion of an item with key 78 into a binary search tree. Finding the position to insert is shown in (a), and the resulting tree is shown in (b).

### 3.1.5 Removal in a Binary Search Tree

Performing the removeElement($k$) operation on a dictionary $D$ implemented with a binary search tree $T$ is a bit more complex, since we do not wish to create any "holes" in the tree $T$. Such a hole, where an internal node would not store an element, would make it difficult if not impossible for us to correctly perform searches in the binary search tree. Indeed, if we have many removals that do not restructure the tree $T$, then there could be a large section of internal nodes that store no elements, which would confuse any future searches.

The removal operation starts out simple enough, since we begin by executing algorithm TreeSearch($k$,$T$.root()) on $T$ to find a node storing key $k$. If TreeSearch returns an external node, then there is no element with key $k$ in dictionary $D$, and we return the special element NO_SUCH_KEY and we are done. If TreeSearch returns an internal node $w$ instead, then $w$ stores an item we wish to remove.

We distinguish two cases (of increasing difficulty) of how to proceed based on whether $w$ is a node that is easily removed or not:

- 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 (also see Figure 2.26 and Section 2.3.4) restructures $T$ by replacing $w$ with the sibling of $z$, removing both $w$ and $z$ from $T$.

This case is illustrated in Figure 3.8.

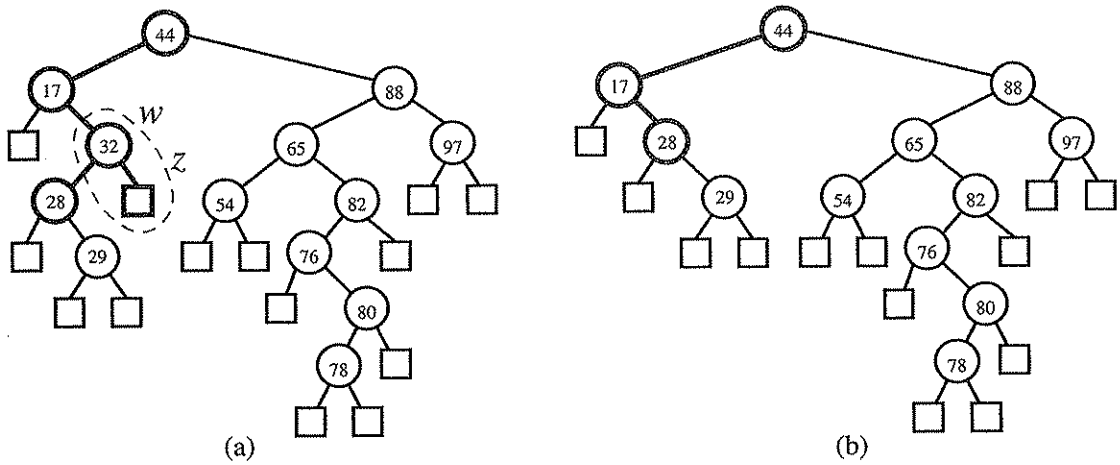



**Figure 3.8:** Removal from the binary search tree of Figure 3.7b, where the key to remove (32) is stored at a node ($w$) with an external child: (a) shows the tree before the removal, together with the nodes affected by the removeAboveExternal($z$) operation on $T$; (b) shows the tree $T$ after the removal.

- 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 3.9):
  1. 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 left children. Also, the left child $x$ of $y$ is the external node that immediately follows node $w$ in the inorder traversal of $T$.
  2. We save the element stored at $w$ in a temporary variable $t$, and move the item of $y$ into $w$. This action has the effect of removing the former item stored at $w$.
  3. We remove $x$ and $y$ from $T$ using operation removeAboveExternal($x$) on $T$. This action replaces $y$ with $x$'s sibling, and removes both $x$ and $y$ from $T$.
  4. We return the element previously stored at $w$, which we had saved in the temporary variable $t$.

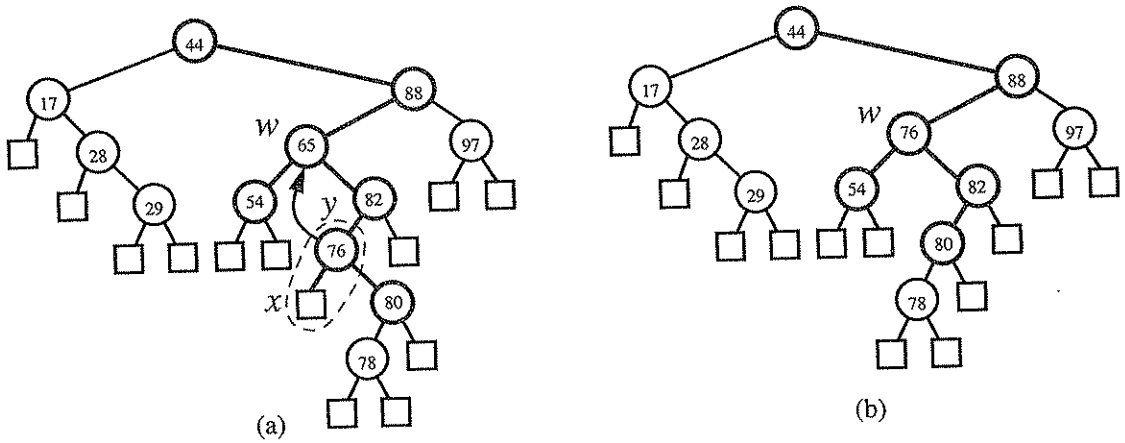



**Figure 3.9:** Removal from the binary search tree of Figure 3.7b, where the key to remove (65) is stored at a node whose children are both internal: (a) before the removal; (b) after the removal.

The analysis of the removal algorithm is analogous to that of the insertion and search algorithms. 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 dictionary $D$ implemented with a binary search tree $T$, the removeElement method runs in $O(h)$ time, where $h$ is the height of $T$.

We can also show that a variation of the above algorithm performs operation removeAllElements($k$) in time $O(h+s)$, where $s$ is the number of elements in the iterator returned. The details are left as an exercise (C-3.4).

### 3.1.6 Performance of Binary Search Trees

The performance of a dictionary implemented with a binary search is summarized in the following theorem and in Table 3.10.

**Theorem 3.1:** *A binary search tree $T$ with height $h$ for $n$ key-element items uses $O(n)$ space and executes the dictionary ADT operations with the following running times. Operations* size *and* isEmpty *each take $O(1)$ time. Operations* findElement, insertItem, *and* removeElement *each take time $O(h)$ time. Operations* findAllElements *and* removeAllElements *each take $O(h+s)$ time, where $s$ is the size of the iterators returned.*

| Method | Time |
|---|---|
| size, isEmpty | $O(1)$ |
| findElement, insertItem, removeElement | $O(h)$ |
| findAllElements, removeAllElements | $O(h+s)$ |

**Table 3.10:** Running times of the main methods of a dictionary realized by a binary search tree. We denote with $h$ the current height of the tree and with $s$ the size of the iterators returned by findAllElements and removeAllElements. The space usage is $O(n)$, where $n$ is the number of items stored in the dictionary.

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 justify; hence, its justification is beyond the scope of this book. Thus, we can be content knowing that random update sequences give rise to binary search trees that have logarithmic height on average, but, keeping in mind their poor worst-case performance, we should also take care in using standard binary search trees in applications where updates are not random.

The relative simplicity of the binary search tree and its good average-case performance make binary search trees a rather attractive dictionary data structure in applications where the keys inserted and removed follow a random pattern and occasionally slow response time is acceptable. There are, however, applications where it is essential to have a dictionary with fast worst-case search and update time. The data structures presented in the next sections address this need.

# 3.3 Bounded-Depth Search Trees

Some search trees base their efficiency on rules that explicitly bound their depth. In fact, such trees typically define a depth function, or a "pseudo-depth" function closely related to depth, so that every external node is at the same depth or pseudo-depth. In so doing, they maintain every external node to be at depth $O(\log n)$ in a tree storing $n$ elements. Since tree searches and updates usually run in times that are proportional to depth, such a ***depth-bounded tree*** can be used to implement an ordered dictionary with $O(\log n)$ search and update times.

## 3.3.1 Multi-Way Search Trees

Some bounded-depth search trees are multi-way trees, that is, trees with internal nodes that have two or more children. In this section, we describe how multi-way trees can be used as search trees, including how multi-way trees store items and how we can perform search operations in multi-way search trees. Recall that the ***items*** that we store in a search tree are pairs of the form $(k,x)$, where $k$ is the ***key*** and $x$ is the element associated with the key.

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 3.17a):

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

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 node with $d$ children 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 items and serve only as "placeholders." Thus, we view a binary search tree (Section 3.1.2) as a special case of a multi-way search tree. At the other extreme, a multi-way search tree may have only a single internal node storing all the items. 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.

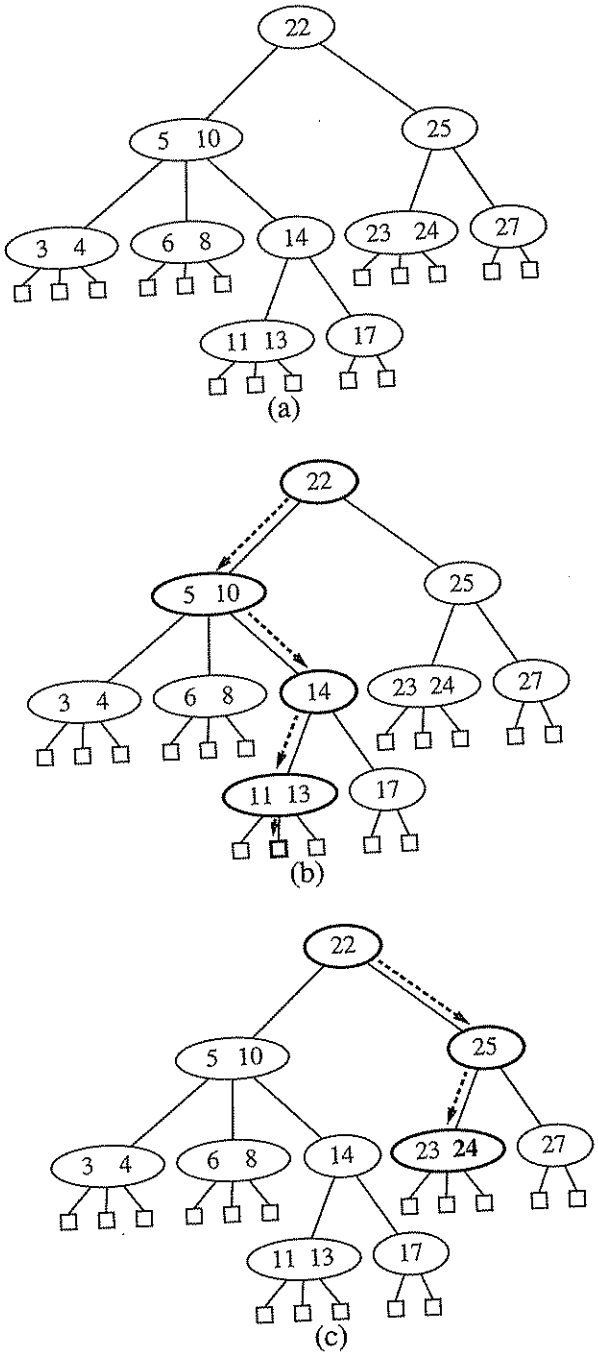



**Figure 3.17:** (a) A multi-way search tree $T$; (b) search path in $T$ for key 12 (unsuccessful search); (c) search path in $T$ for key 24 (successful search).

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

**Theorem 3.3:** *A multi-way search tree storing $n$ items has $n+1$ external nodes.*

We leave the justification of this theorem as an exercise (C-3.16).

### Searching in a Multi-Way Tree

Given a multi-way search tree $T$, searching for an element with key $k$ is simple. We perform such a search by tracing a path in $T$ starting at the root. (See Figure 3.17b 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 consider $k_0 = -\infty$ and $k_d = +\infty$.) If we reach an external node, then we know that there is no item with key $k$ in $T$, and the search terminates unsuccessfully.

### Data Structures for Multi-Way Search Trees

In Section 2.3.4, we discussed different ways of representing general trees. Each of these representations can also be used for multi-way search trees. In fact, in using a general multi-way tree to implement a multi-way search tree, the only additional information that we need to store at each node is the set of items (including keys) associated with that node. That is, we need to store with $v$ a reference to some container or collection object that stores the items for $v$.

Recall that when we use a binary tree to represent an ordered dictionary $D$, we simply store a reference to a single item at each internal node. In using a multi-way search tree $T$ to represent $D$, we must store a reference to the ordered set of items 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 dictionary to represent an ordered dictionary. 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 dictionary data structure that we have previously constructed (for example, a lookup table based on an ordered vector, as shown in Section 3.1.1). In particular, assuming we already have a way of implementing ordered dictionaries, we can realize a multi-way search tree by taking a tree $T$ and storing such a dictionary at each $d$-node $v$ of $T$.

The dictionary we store at each node $v$ is known as a ***secondary*** data structure, for we are using it to support the bigger, ***primary*** data structure. We denote the dictionary stored at a node $v$ of $T$ as $D(v)$. The items we store in $D(v)$ will allow us to find which child node to move to next during a search operation. Specifically, for

each node $v$ of $T$, with children $v_1, \ldots, v_d$ and items $(k_1, x_1), \ldots, (k_{d-1}, x_{d-1})$, we store in the dictionary $D(v)$ the items $(k_1, x_1, v_1), (k_2, x_2, v_2), \ldots, (k_{d-1}, x_{d-1}, v_{d-1}), (+\infty, \mathit{null}, v_d)$. That is, an item $(k_i, x_i, v_i)$ of dictionary $D(v)$ has key $k_i$ and element $(x_i, v_i)$. Note that the last item stores the special key $+\infty$.

With the above realization of a multi-way search tree $T$, processing a $d$-node $v$ while searching for an element of $T$ with key $k$ can be done by performing a search operation to find the item $(k_i, x_i, v_i)$ in $D(v)$ with smallest key greater than or equal to $k$, such as in the closestElemAfter($k$) operation (see Section 3.1). 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.

### Performance Issues for Multi-Way Search Trees

Consider the space requirement for the above realization of a multi-way search tree $T$ storing $n$ items. By Theorem 3.3, using any of the common realizations of ordered dictionaries (Section 2.5) 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 $D(v)$. If $D(v)$ is realized with a vector-based sorted sequence (that is, a lookup table), then we can process $v$ in $O(\log d)$ time. If instead $D(v)$ is realized using an unsorted sequence (that is, a log file), 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 dictionaries $D(v)$). 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 prime 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 number of total items stored in the dictionary. A search tree with logarithmic height, such as this, is called a ***balanced search tree***. Bounded-depth search trees satisfy this goal by keeping each external node at exactly the same depth level in the tree.

Next, we discuss a bounded-depth search tree that is a multi-way search tree that caps $d_{\max}$ at 4. In Section 14.1.2, we discuss a more general kind of multi-way search tree that has applications where our search tree is too large to completely fit into the internal memory of our computer.

### 3.3.2 (2,4) Trees

In using a multi-way search tree in practice, we desire that it be balanced, that is, have logarithmic height. The multi-way search tree we study next is fairly easy to keep balanced. It is the $(2,4)$ tree, which is sometimes also called the 2-4 tree or 2-3-4 tree. In fact, we can maintain balance in a $(2,4)$ tree by maintaining two simple properties (see Figure 3.18):

***Size Property*:** Every node has at most four children.

***Depth Property*:** All the external nodes have the same depth.

Enforcing the size property for $(2,4)$ trees keeps the size of the nodes in the multi-way search tree constant, for it allows us to represent the dictionary $D(v)$ stored at each internal node $v$ using a constant-sized array. The depth property, on the other hand, maintains the balance in a $(2,4)$ tree, by forcing it to be a bounded-depth structure.

**Theorem 3.4:** *The height of a $(2,4)$ tree storing $n$ items is $\Theta(\log n)$.*

**Proof:** Let $h$ be the height of a $(2,4)$ tree $T$ storing $n$ items. Note 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 Theorem 3.3, the number of external nodes in $T$ is $n+1$. Therefore, we obtain

$$2^h \leq n+1 \quad \text{and} \quad n+1 \leq 4^h.$$

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

$$h \leq \log(n+1) \quad \text{and} \quad \log(n+1) \leq 2h,$$

which justifies our theorem. ∎

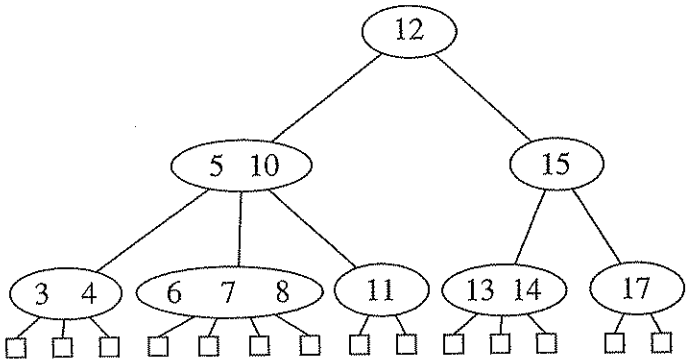



**Figure 3.18:** A $(2,4)$ tree.

### Insertion in a (2,4) Tree

Theorem 3.4 states that the size and depth properties are sufficient for keeping a multi-way tree balanced. Maintaining these properties requires some effort after performing insertions and removals in a (2,4) tree, however. In particular, to insert a new item $(k,x)$, with key $k$, into a (2,4) tree $T$, we first perform a search for $k$. Assuming that $T$ has no element with key $k$, this search terminates unsuccessfully at an external node $z$. Let $v$ be the parent of $z$. We insert the new item into node $v$ and add a new child $w$ (an external node) to $v$ on the left of $z$. That is, we add item $(k,x,w)$ to the dictionary $D(v)$.

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 $v_1,\ldots,v_5$ be the children of $v$, and let $k_1,\ldots,k_4$ be the keys stored at $v$. To remedy the overflow at node $v$, we perform a ***split*** operation on $v$ as follows (see Figure 3.19):

- 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 3.20.

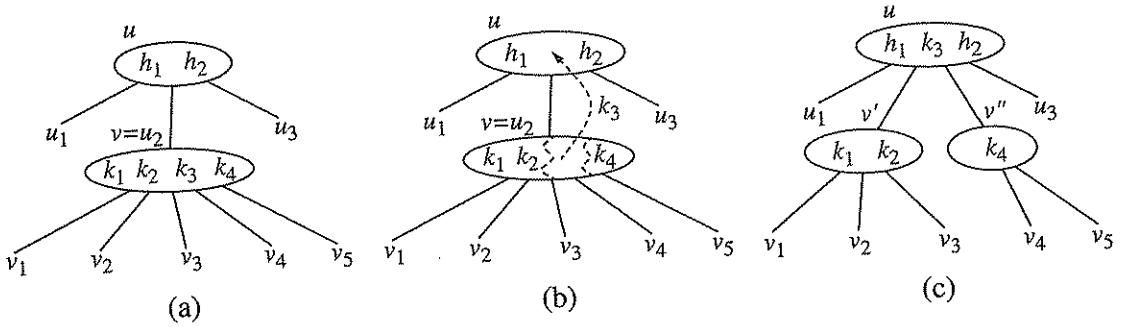



**Figure 3.19:** A node split: (a) overflow at a 5-node $v$; (b) the third key of $v$ inserted into the parent $u$ of $v$; (c) node $v$ replaced with a 3-node $v'$ and a 2-node $v''$.

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

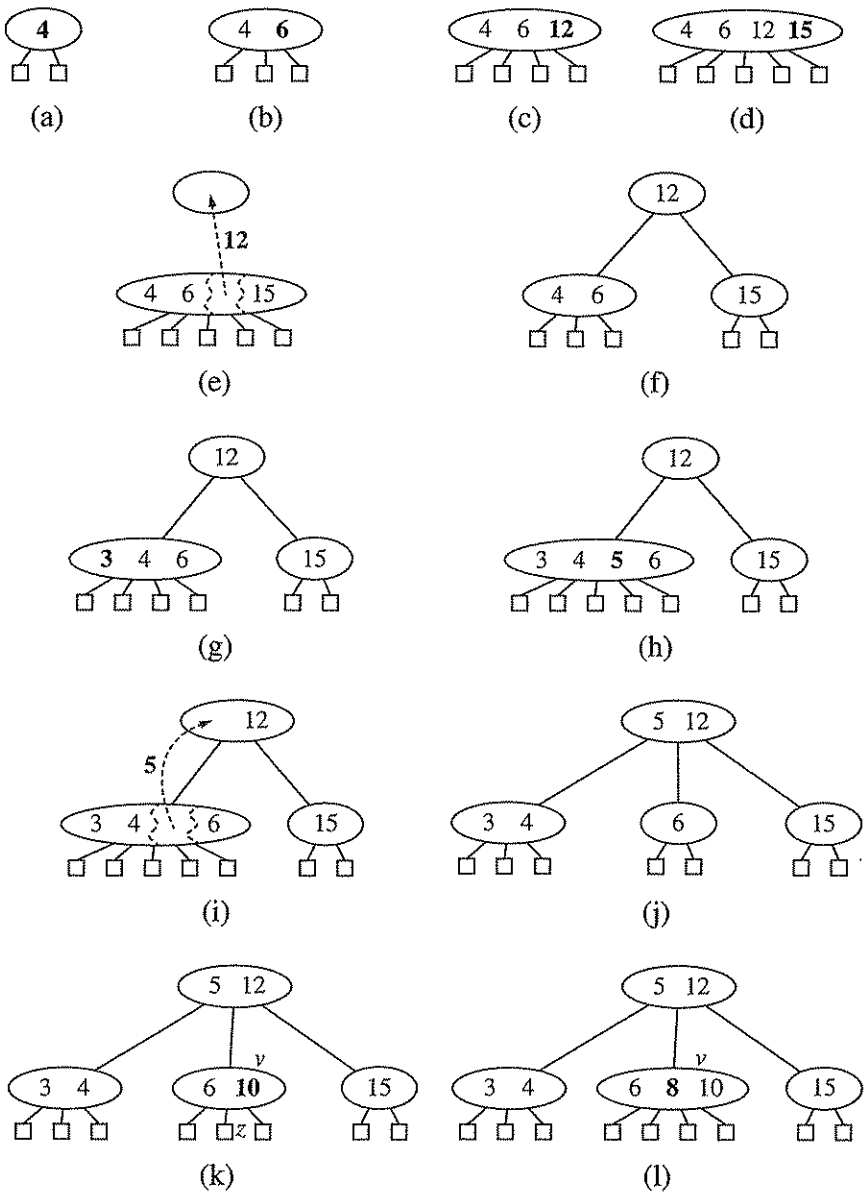



**Figure 3.20:** A sequence of insertions into a (2,4) tree: (a) initial tree with one item; (b) insertion of 6; (c) insertion of 12; (d) insertion of 15, which causes an overflow; (e) split, which causes the creation of a new root node; (f) after the split; (g) insertion of 3; (h) insertion of 5, which causes an overflow; (i) split; (j) after the split; (k) insertion of 10; (l) insertion of 8.

### Performance of (2,4) Tree Insertion

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, in turn, a split at node $u$. (See Figure 3.21.) A split operation either eliminates the overflow or propagates it into the parent of the current node. Indeed, this propagation can continue all the way up to the root of the search tree. But if it does propagate all the way to the root, it will finally be resolved at that point. We show such a sequence of splitting propagations in Figure 3.21.

Thus, the number of split operations is bounded by the height of the tree, which is $O(\log n)$ by Theorem 3.4. Therefore, the total time to perform an insertion in a $(2,4)$ tree is $O(\log n)$.

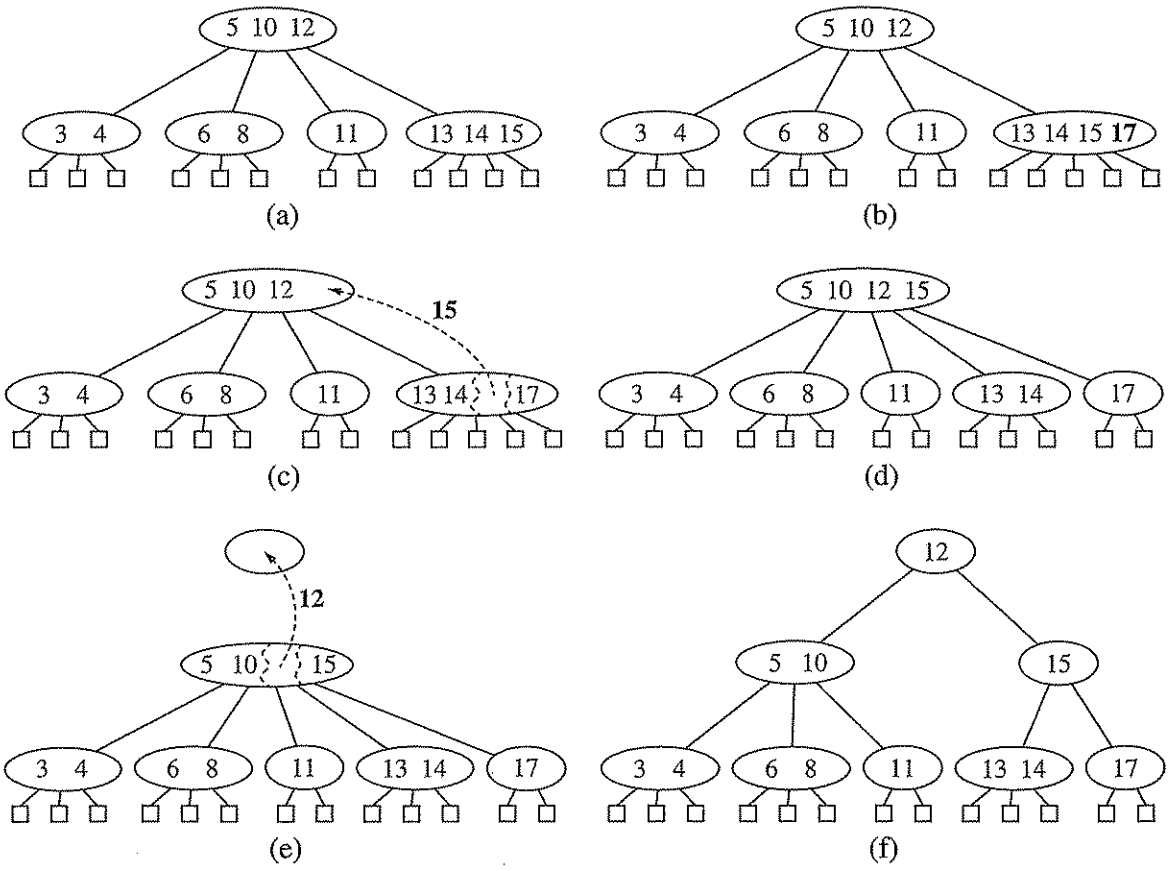



**Figure 3.21:** An insertion in a $(2,4)$ tree that causes a cascading split: (a) before the insertion; (b) insertion of 17, causing an overflow; (c) a split; (d) after the split a new overflow occurs; (e) another split, creating a new root node; (f) final tree.

### Removal in a $(2,4)$ Tree

Let us now consider the removal of an item with key $k$ from a $(2,4)$ tree $T$. We begin such an operation by performing a search in $T$ for an item with key $k$. Removing such an item from a $(2,4)$ tree can always be reduced to the case where the item to be removed is stored at a node $v$ whose children are external nodes. Suppose, for instance, that the item with key $k$ that we wish to remove is stored in the $i$th item $(k_i, x_i)$ at a node $z$ that has only internal-node children. In this case, we swap the item $(k_i, x_i)$ with an appropriate item that is stored at a node $v$ with external-node children as follows (Figure 3.22d):

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

Once we ensure that the item 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 item from $v$ (that is, from the dictionary $D(v)$) and remove the $i$th external node of $v$.

Removing an item (and a child) from a node $v$ as described above preserves the depth property, for 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 items after the removal (Figure 3.22d 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 3.22b 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 3.23e 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 3.23.) Hence, the number of fusion operations is bounded by the height of the tree, which is $O(\log n)$ by Theorem 3.4. If an underflow propagates all the way up to the root, then the root is simply deleted. (See Figure 3.23c and d.) We show a sequence of removals from a $(2,4)$ tree in Figures 3.22 and 3.23.

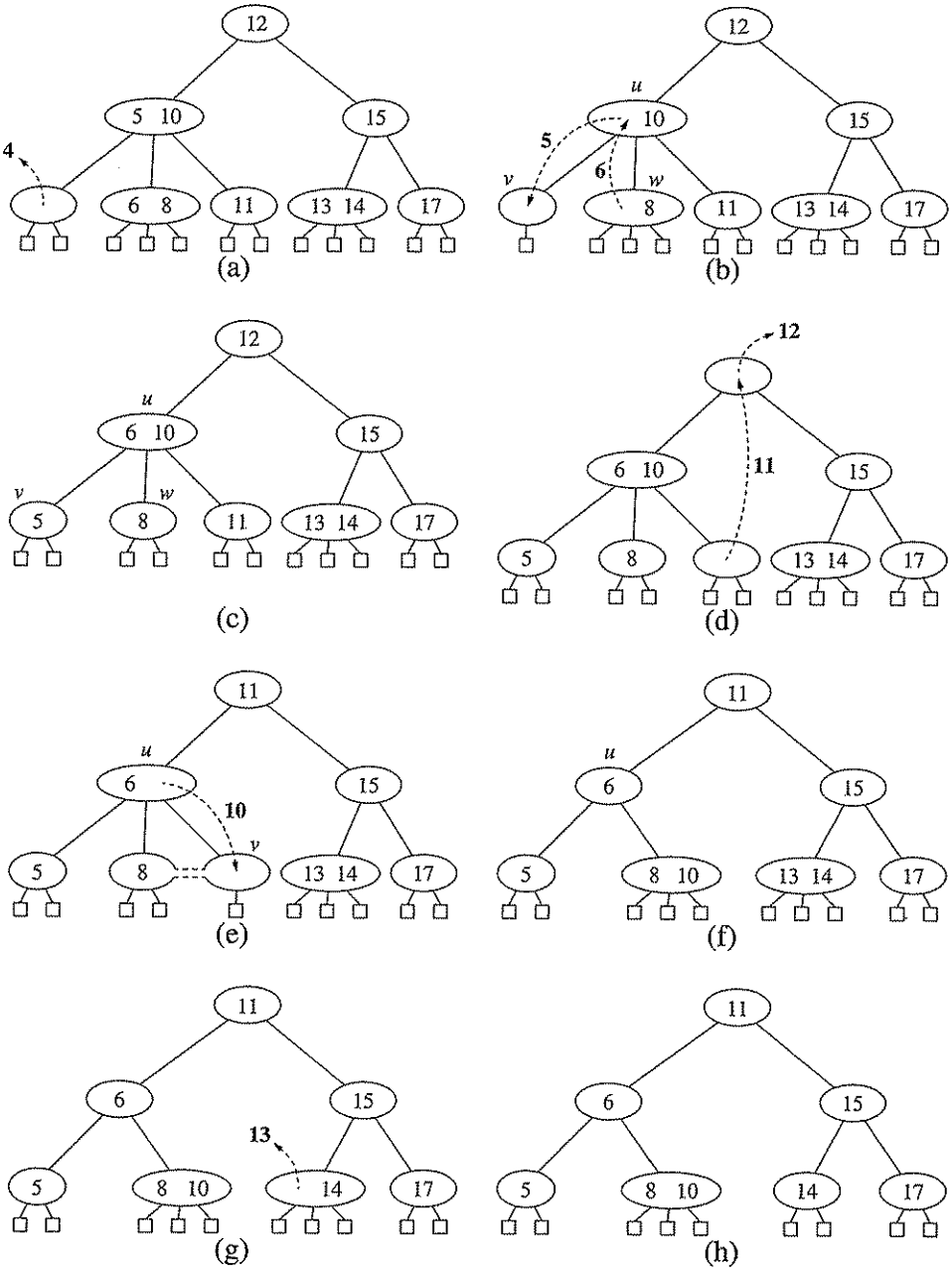



**Figure 3.22:** A sequence of removals from a $(2,4)$ tree: (a) removal of 4, causing an underflow; (b) a transfer operation; (c) after the transfer operation; (d) removal of 12, causing an underflow; (e) a fusion operation; (f) after the fusion operation; (g) removal of 13; (h) after removing 13.

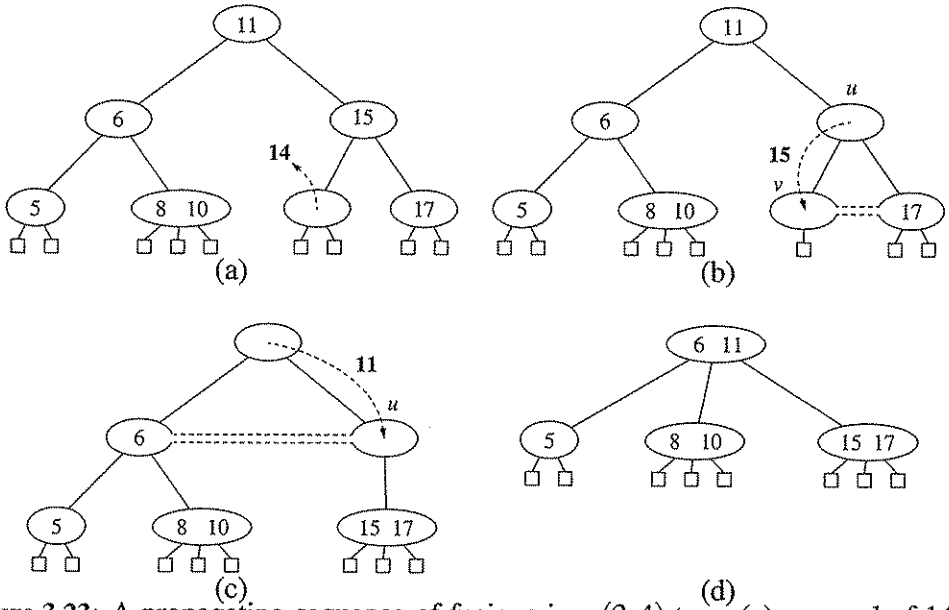



**Figure 3.23:** A propagating sequence of fusions in a $(2,4)$ tree: (a) removal of 14, which causes an underflow; (b) fusion, which causes another underflow; (c) second fusion operation, which causes the root to be removed; (d) final tree.

## Performance of a $(2,4)$ Tree

Table 3.24 summarizes the running times of the main operations of a dictionary realized with a $(2,4)$ tree. The time complexity analysis is based on the following:

- The height of a $(2,4)$ tree storing $n$ items is $O(\log n)$, by Theorem 3.4.
- A split, transfer, or fusion operation takes $O(1)$ time.
- A search, insertion, or removal of an item visits $O(\log n)$ nodes.

| Operation | Time |
|---|---|
| size, isEmpty | $O(1)$ |
| findElement, insertItem, removeElement | $O(\log n)$ |
| findAllElements, removeAllElements | $O(\log n + s)$ |

**Table 3.24:** Performance of an $n$-element dictionary realized by a $(2,4)$ tree, where $s$ denotes the size of the iterators returned by findAllElements and removeAllElements. The space usage is $O(n)$.

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

### 3.3.3 Red-Black Trees

The data structure we discuss in this section, the red-black tree, is a binary search tree that uses a kind of "pseudo-depth" to achieve balance using the approach of a depth-bounded search tree. In particular, a ***red-black tree*** is a binary search tree 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***, which is defined as the number of black ancestors minus one.

An example of a red-black tree is shown in Figure 3.25. Throughout this section, we use the convention of drawing ***black nodes*** and their parent edges with ***thick lines***.

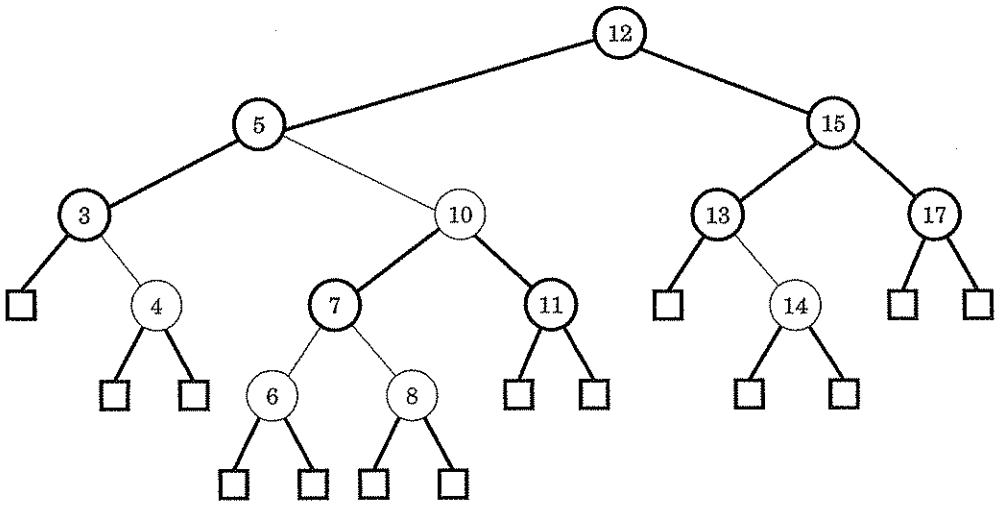



**Figure 3.25:** Red-black tree associated with the $(2,4)$ tree of Figure 3.18. Each external node of this red-black tree has three black ancestors; hence, it has black depth 3. Recall that we use thick lines to denote black nodes.

As has been the convention in this chapter, we assume that items are stored in the internal nodes of a red-black tree, with the external nodes being empty placeholders. Also, we describe our algorithms assuming external nodes are real, but we note in passing that at the expense of slightly more complicated search and update methods, external nodes could be **null** or references to a NULL_NODE object.

The red-black tree definition becomes more intuitive by noting an interesting correspondence between red-black and $(2,4)$ trees, as illustrated in Figure 3.26. 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 item 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 a simple 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 $v$.

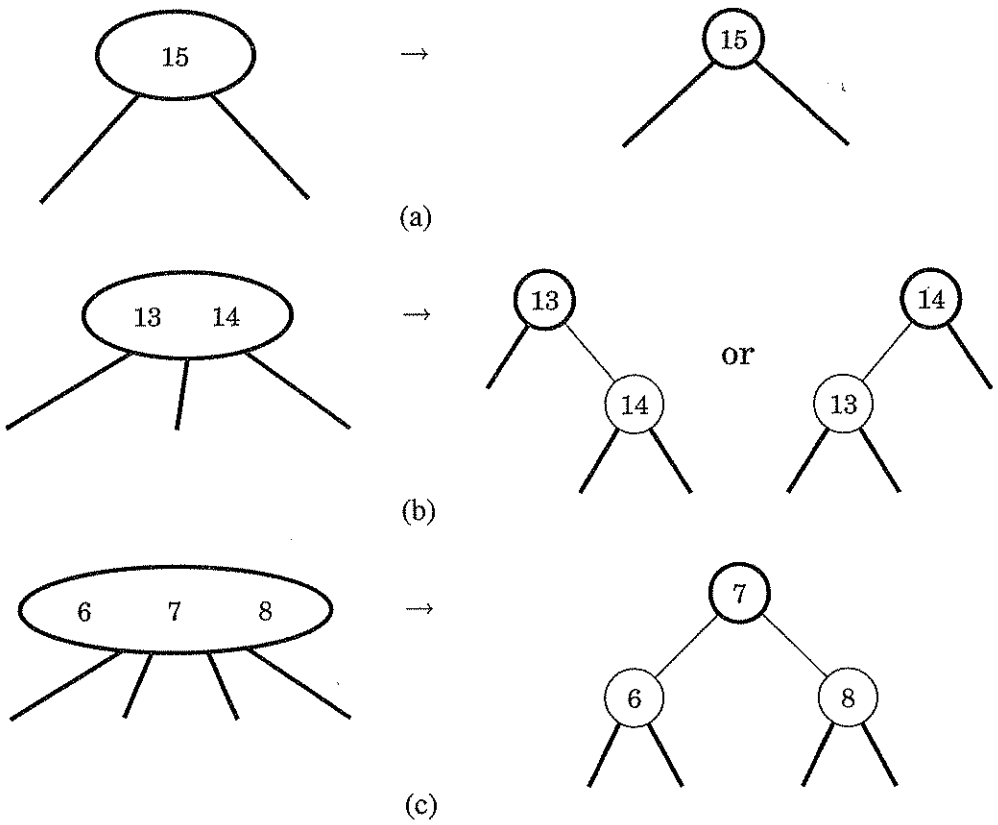



**Figure 3.26:** Correspondence between a $(2,4)$ tree and a red-black tree: (a) 2-node; (b) 3-node; (c) 4-node.

This correspondence between $(2,4)$ trees and red-black trees provides important intuition that we will use in our discussions. In fact, the update algorithms for red-black trees are mysteriously complex without this intuition. We also have the following property for red-black trees.

**Theorem 3.5:** *The height of a red-black tree storing $n$ items is $O(\log n)$.*

**Proof:** Let $T$ be a red-black tree storing $n$ items, and let $h$ be the height of $T$. We justify this theorem 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 $T'$. We know that $h' = d$. Hence, by Theorem 3.4, $d = h' \leq \log(n+1)$. 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 Theorem 2.8 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 2.3.4), in which we store a dictionary item 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 3.1.2). Thus, searching in a red-black tree takes $O(\log n)$ time.

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 in a Red-Black Tree

Consider the insertion of an element $x$ 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 insertion algorithm initially proceeds as in a binary search tree (Section 3.1.4). 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 3.27.) In this case, the double red denotes the fact that we have created a malformed replacement for a corresponding 4-node of the $(2,4)$ tree $T'$ in our red-black tree, 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 3.27; this operation is also discussed in Section 3.2):

- 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.

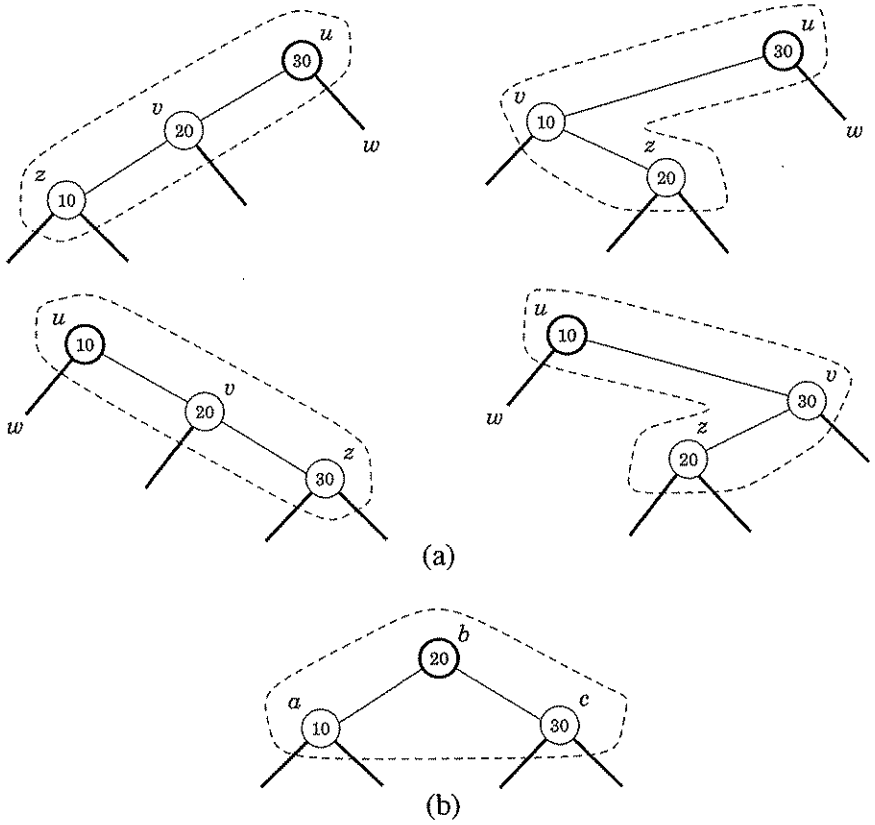



**Figure 3.27:** Restructuring a red-black tree to remedy a double red: (a) the four configurations for $u$, $v$, and $z$ before restructuring; (b) after restructuring.

**Case 2:** ***The Sibling*** **w** ***of*** **v** ***is Red.*** (See Figure 3.28.) 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, albeit 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 Theorem 3.5.

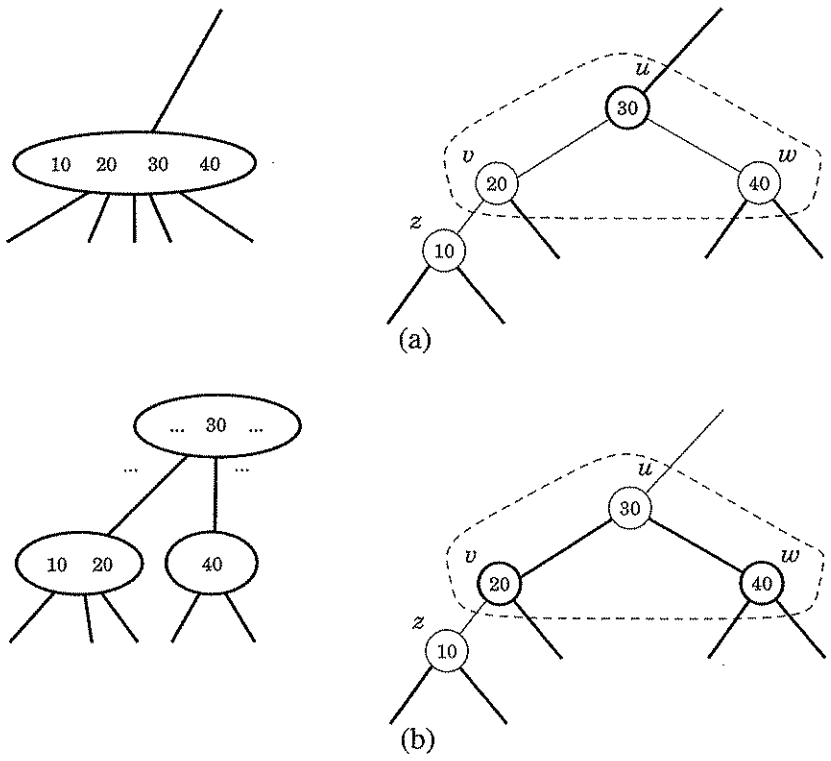



**Figure 3.28:** Recoloring to remedy the double-red problem: (a) before recoloring and the corresponding 5-node in the associated $(2,4)$ tree before the split; (b) after the recoloring (and corresponding nodes in the associated $(2,4)$ tree after the split).

Figures 3.29 and 3.30 show a sequence of insertions in a red-black tree.

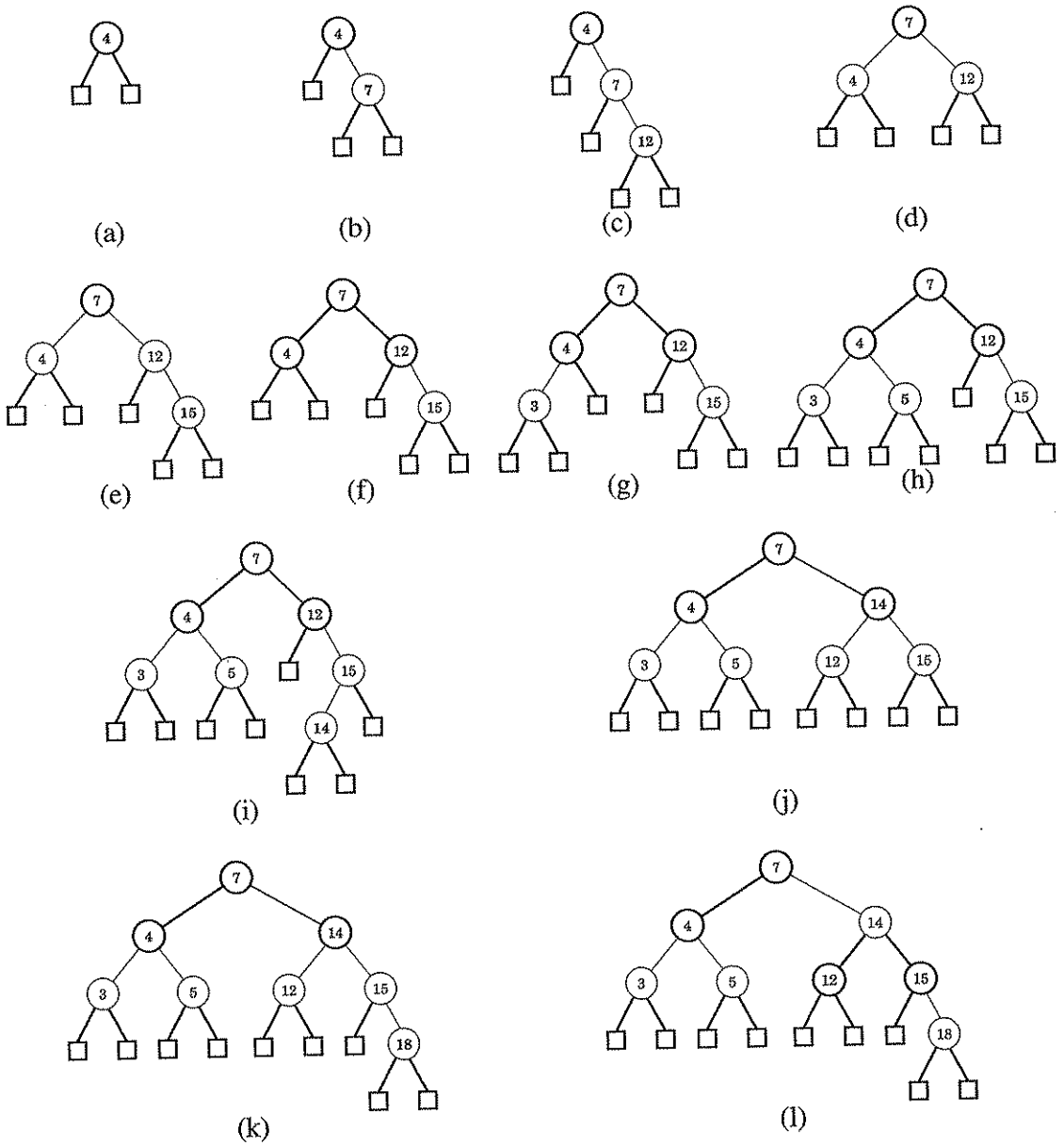



**Figure 3.29:** A sequence of insertions in a red-black tree: (a) initial tree; (b) insertion of 7; (c) insertion of 12, which causes a double red; (d) after restructuring; (e) insertion of 15, which causes a double red; (f) after recoloring (the root remains black); (g) insertion of 3; (h) insertion 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. (Continued in Figure 3.30.)

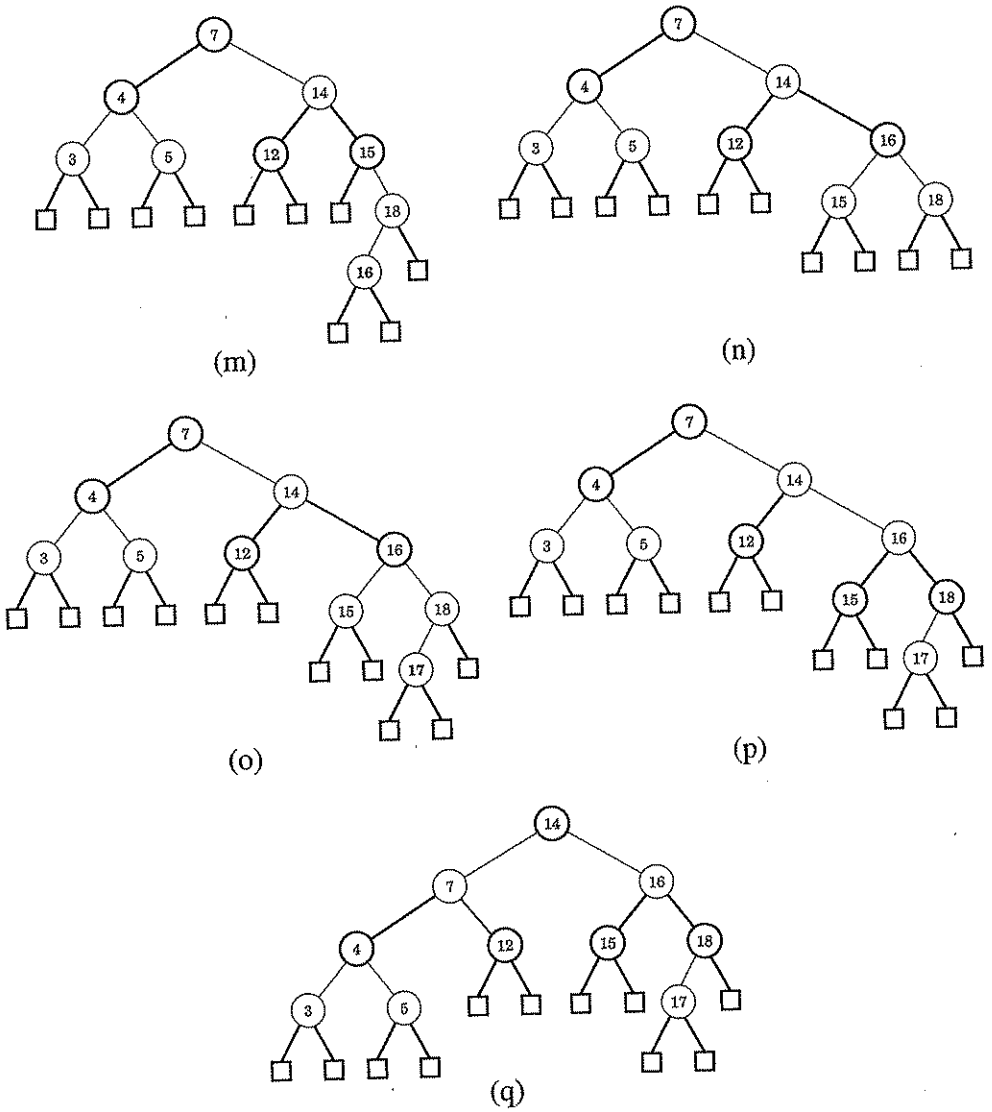



**Figure 3.30:** A sequence of insertions in a red-black tree (continued from Figure 3.29): (m) insertion of 16, which causes a double red; (n) after restructuring; (o) insertion of 17, which causes a double red; (p) after recoloring there is again a double red, to be handled by a restructuring; (q) after restructuring.

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:

**Theorem 3.6:** *The insertion of a key-element item in a red-black tree storing $n$ items can be done in $O(\log n)$ time and requires at most $O(\log n)$ recolorings and one trinode restructuring (a* restructure *operation).*

### Removal in a Red-Black Tree

Suppose now that we are asked to remove an item with key $k$ from a red-black tree $T$. Removing such an item initially proceeds as for a binary search tree (Section 3.1.5). First, we search for a node $u$ storing such an item. 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 item at $v$ to $u$, and perform the removal at $v$. Thus, we may consider only the removal of an item 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 item 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 3.31.) 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 also the description of restructure in Section 3.2.) 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.

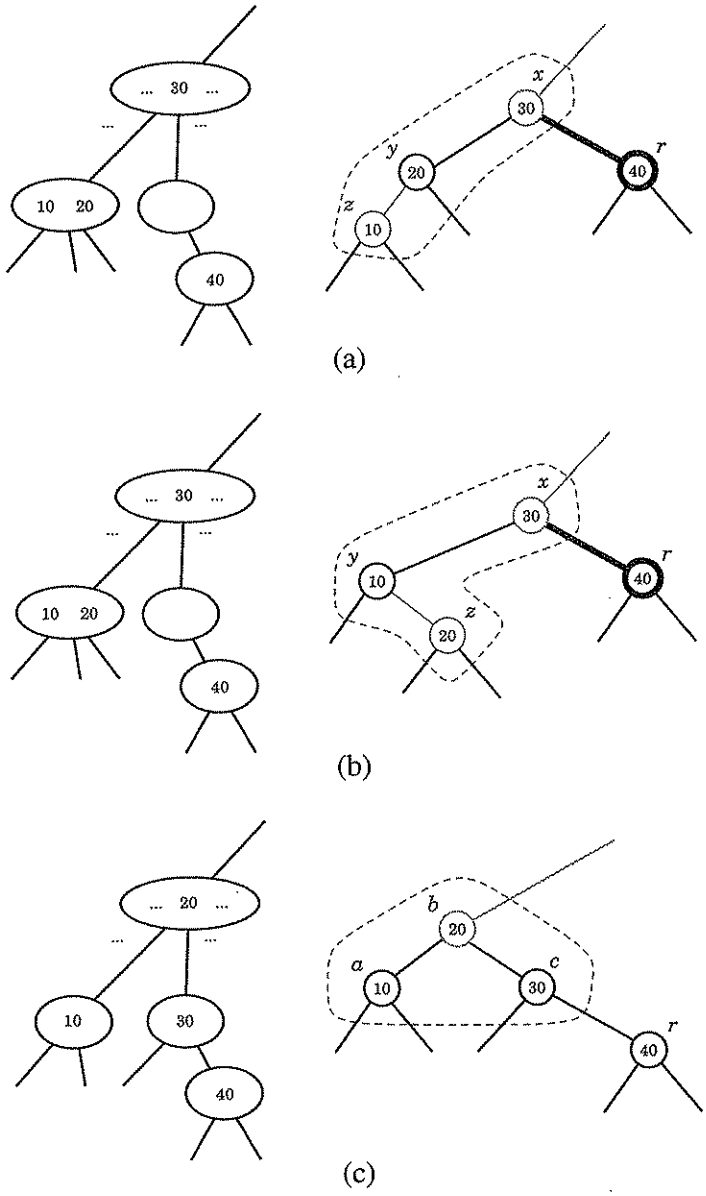



**Figure 3.31:** Restructuring of a red-black tree to remedy the double-black problem: (a) and (b) configurations before the restructuring, where $r$ is a right child and the associated nodes in the corresponding $(2,4)$ tree before the transfer (two other symmetric configurations where $r$ is a left child are possible); (c) configuration after the restructuring and the associated nodes in the corresponding $(2,4)$ tree after the transfer. Node $x$ in parts (a) and (b) and node $b$ in part (c) may be either red or black.

**Case 2:** ***The Sibling*** **y** ***of*** **r** ***is Black and Both Children of*** **y** ***are Black.*** (See Figures 3.32 and 3.33.) 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 3.32); otherwise, we color $x$ ***double black*** (Figure 3.33). Hence, after this recoloring, the double-black problem may reappear at the parent $x$ of $r$. (See Figure 3.33.) 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 $\log(n+1)$.

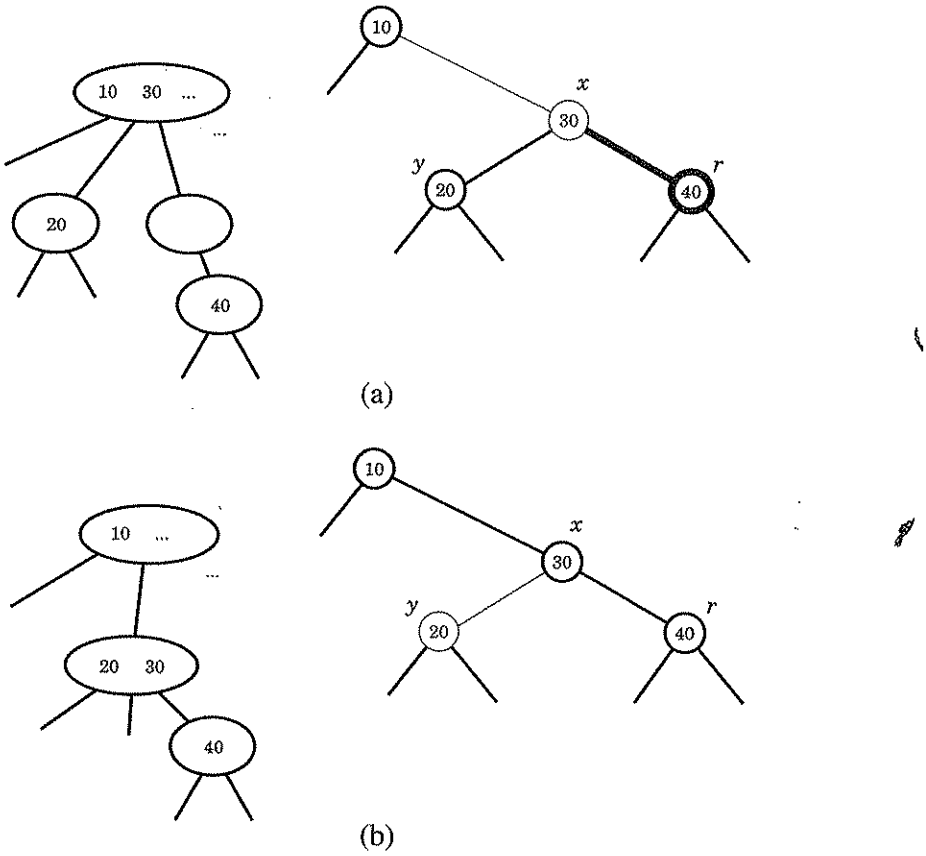



**Figure 3.32:** Recoloring of a red-black tree that fixes the double-black problem: (a) before the recoloring and corresponding nodes in the associated $(2,4)$ tree before the fusion (other similar configurations are possible); (b) after the recoloring and corresponding nodes in the associated $(2,4)$ tree after the fusion.

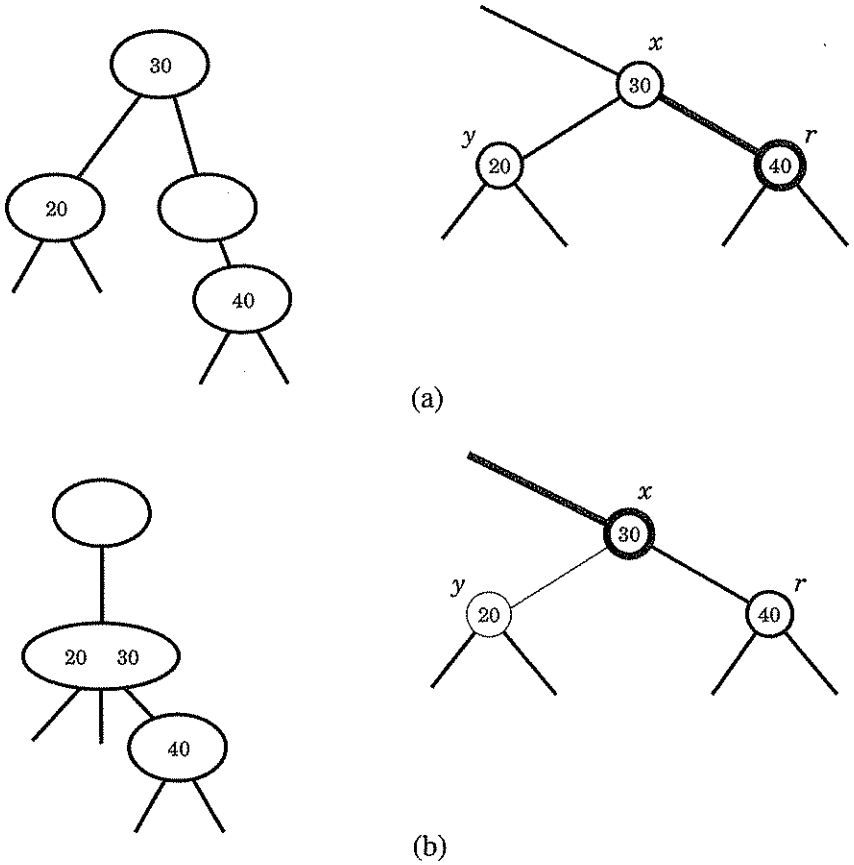



**Figure 3.33:** Recoloring of a red-black tree that propagates the double black problem: (a) configuration before the recoloring and corresponding nodes in the associated $(2,4)$ tree before the fusion (other similar configurations are possible); (b) configuration after the recoloring and corresponding nodes in the associated $(2,4)$ tree after the fusion.

**Case 3:** ***The Sibling*** **y** ***of*** **r** ***is Red.*** (See Figure 3.34.) 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 restructure 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.

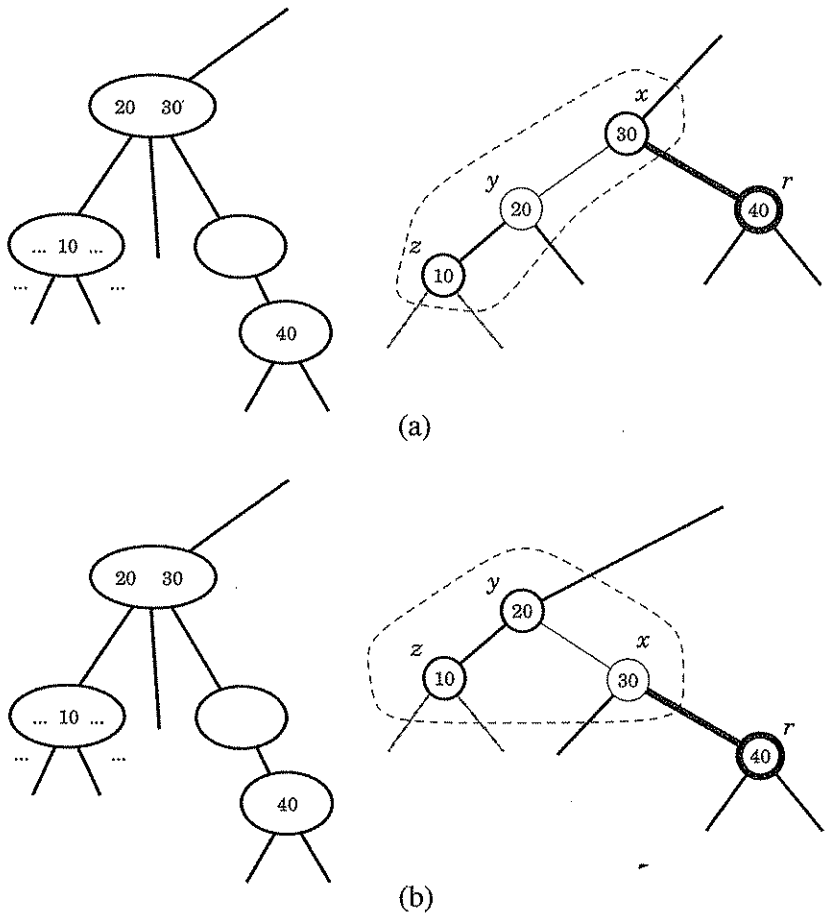



**Figure 3.34:** Adjustment of a red-black tree in the presence of a double black problem: (a) configuration before the adjustment and corresponding nodes in the associated $(2,4)$ tree (a symmetric configuration is possible); (b) configuration after the adjustment with the same corresponding nodes in 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. The changes we make at any node in $T$ during this upward march takes $O(1)$ time, because it affects a constant number of nodes. Moreover, since the restructuring cases terminate upward propagation in the tree, we have the following.

**Theorem 3.7:** *The algorithm for removing an item from a red-black tree with $n$ items 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 3.35 and 3.36, we show a sequence of removal operations on a red-black tree. We illustrate Case 1 restructurings in Figure 3.35c and d. We illustrate Case 2 recolorings at several places in Figures 3.35 and 3.36. Finally, in Figure 3.36i and j, we show an example of a Case 3 adjustment.

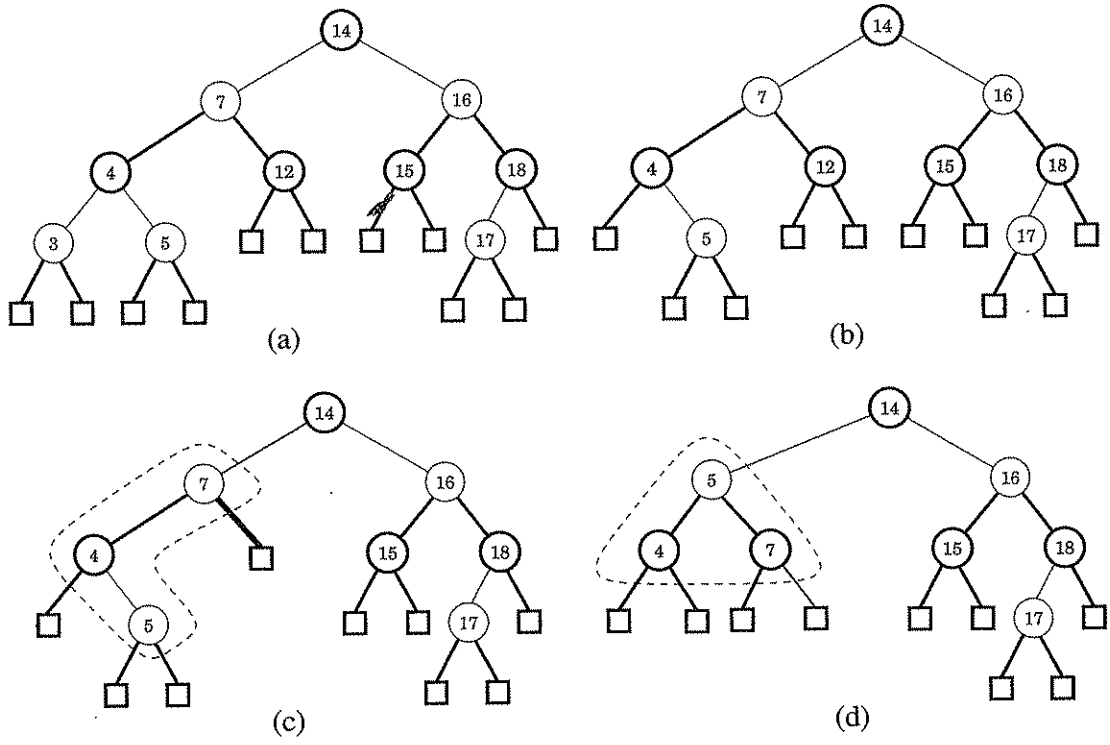



**Figure 3.35:** Sequence of removals from a red-black tree: (a) initial tree; (b) removal of 3; (c) removal of 12, causing a double black (handled by restructuring); (d) after restructuring.

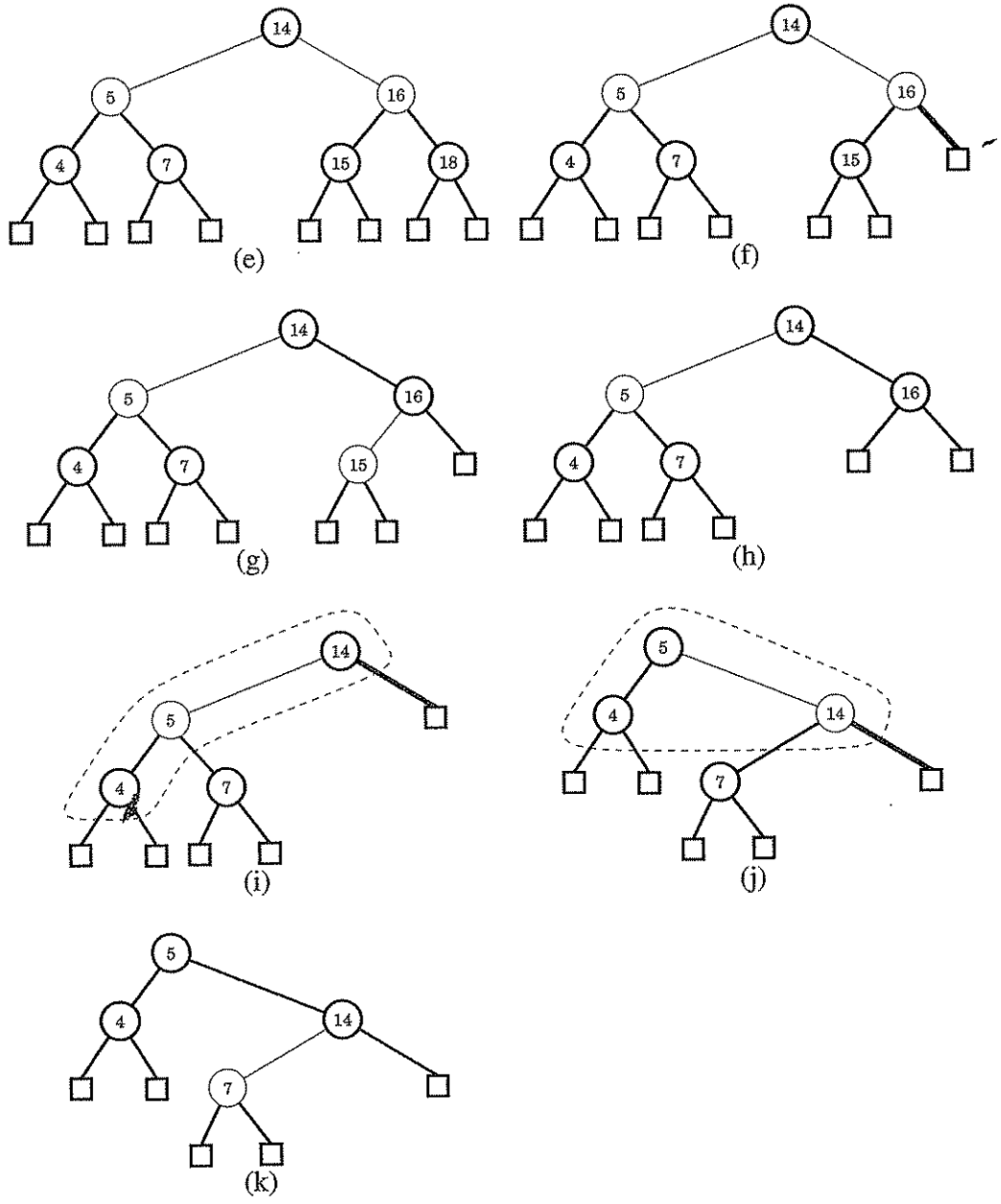



**Figure 3.36:** Sequence of removals in a red-black tree (continued): (e) removal of 17; (f) removal of 18, causing a double black (handled by recoloring); (g) after recoloring; (h) removal of 15; (i) removal of 16, causing a double black (handled by an adjustment); (j) after the adjustment, the double black needs to be handled by a recoloring; (k) after the recoloring.

## Performance of a Red-Black Tree

Table 3.37 summarizes the running times of the main operations of a dictionary realized by means of a red-black tree. We illustrate the justification for these bounds in Figure 3.38.

| Operation | Time |
|---|---|
| size, isEmpty | $O(1)$ |
| findElement, insertItem, removeElement | $O(\log n)$ |
| findAllElements, removeAllElements | $O(\log n + s)$ |

**Table 3.37:** Performance of an $n$-element dictionary realized by a red-black tree, where $s$ denotes the size of the iterators returned by findAllElements and removeAllElements. The space usage is $O(n)$.

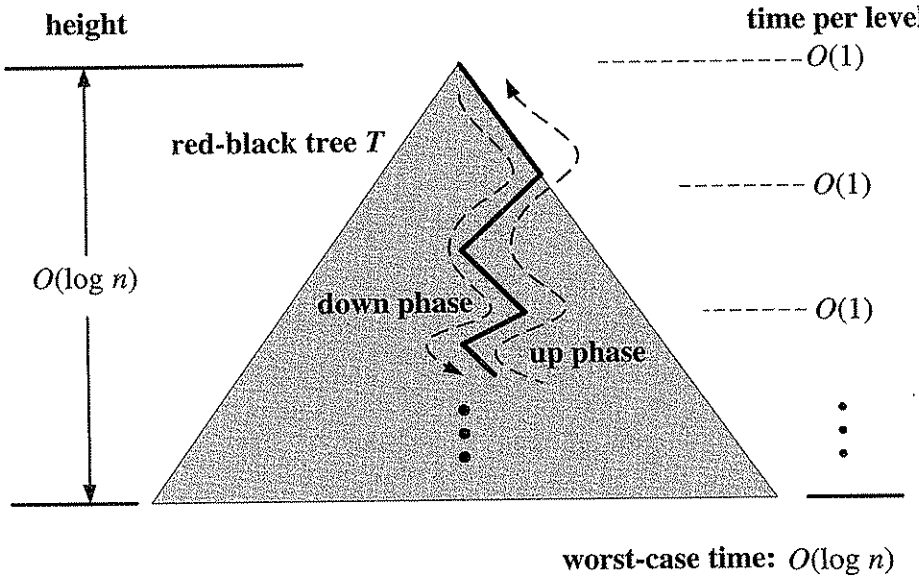



**Figure 3.38:** Illustrating the running time of searches and updates in a red-black tree. The time performance is $O(1)$ per level, broken into a down phase, which typically involves searching, and an up phase, which typically involves recolorings and performing local trinode restructurings (rotations).

Thus, a red-black tree achieves logarithmic worst-case running times for both searching and updating in a dictionary. 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.