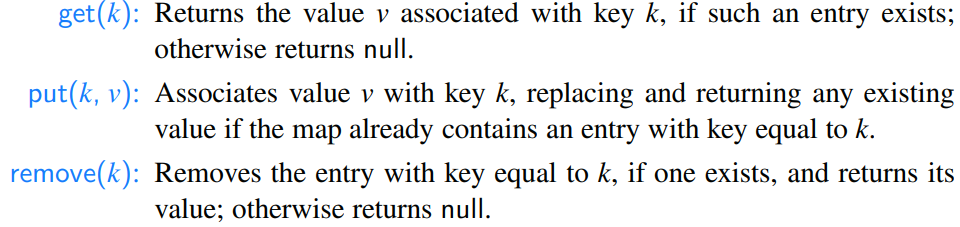
Week7

Search Trees

Binary Search Trees

In this chapter, we use a search-tree structure to efficiently implement a sorted map.

The three most fundamental methods of of a map:

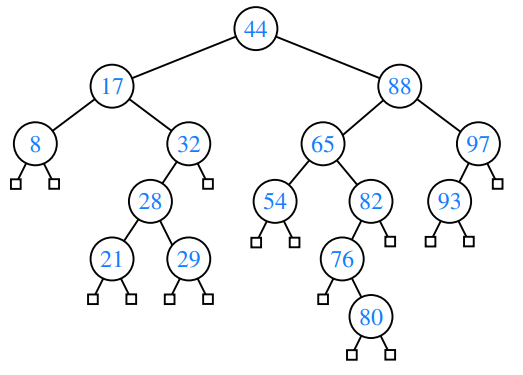


In this chapter, we define a binary search tree as a proper binary tree (see Section 8.2) such that each internal position p stores a key-value pair (k,v) such that:

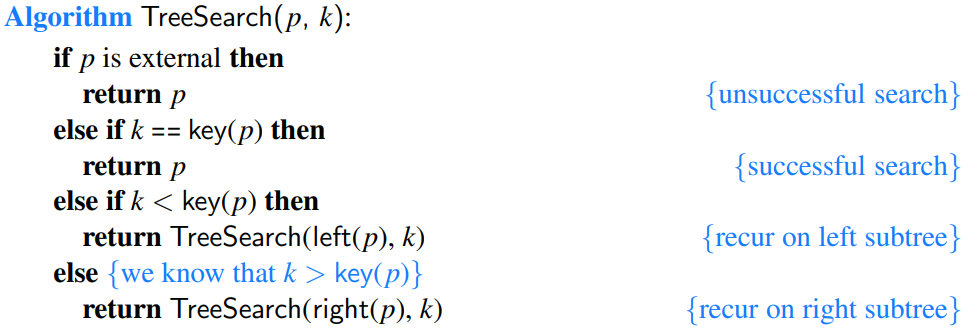
• Keys stored in the left subtree of p are less than k.

• Keys stored in the right subtree of p are greater than k.

Example:

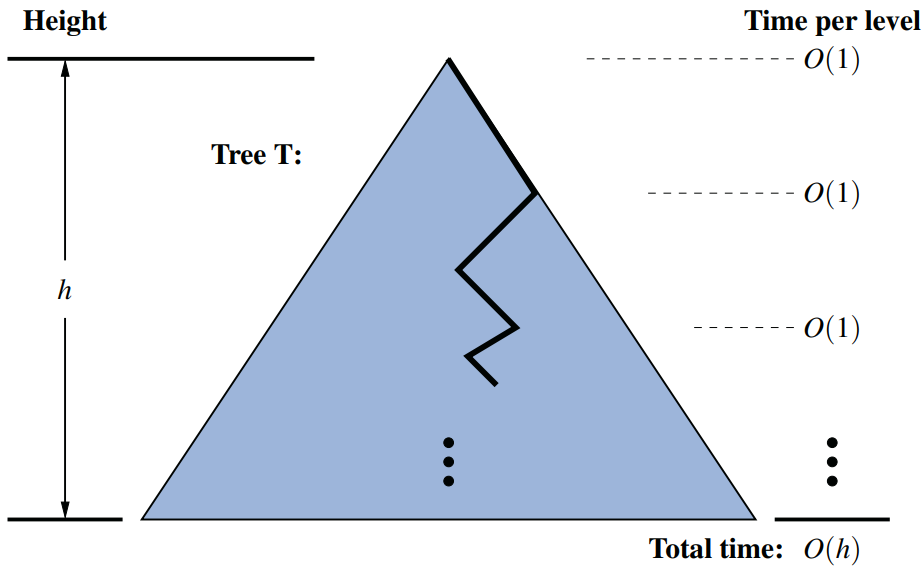


Recursive Search in a Binary Tree



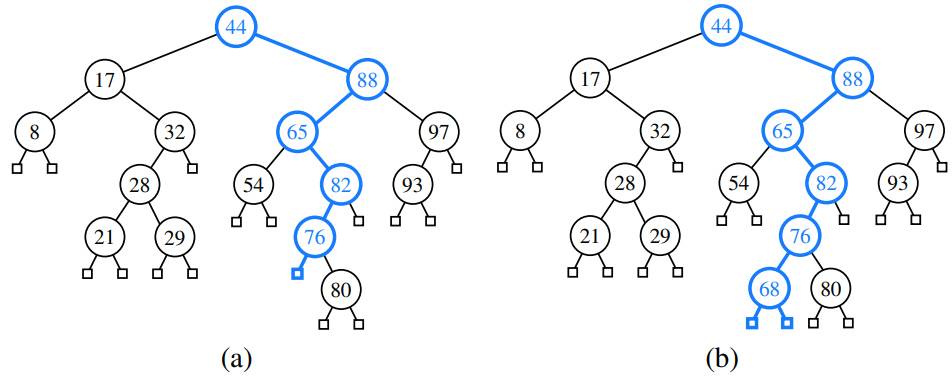
Analysis of Binary Tree Searching

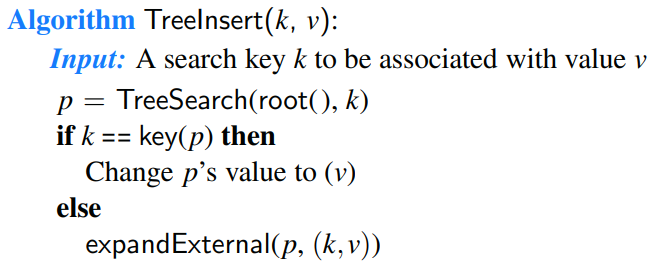
Since we spend O(1) time per position encountered in the search, the overall search runs in O(h) time, where h is the height of the binary search tree T.



Insertions and Deletions

Insertion of an entry with key 68 into the search tree. Finding the position to insert is shown in (a), and the resulting tree is shown in (b).

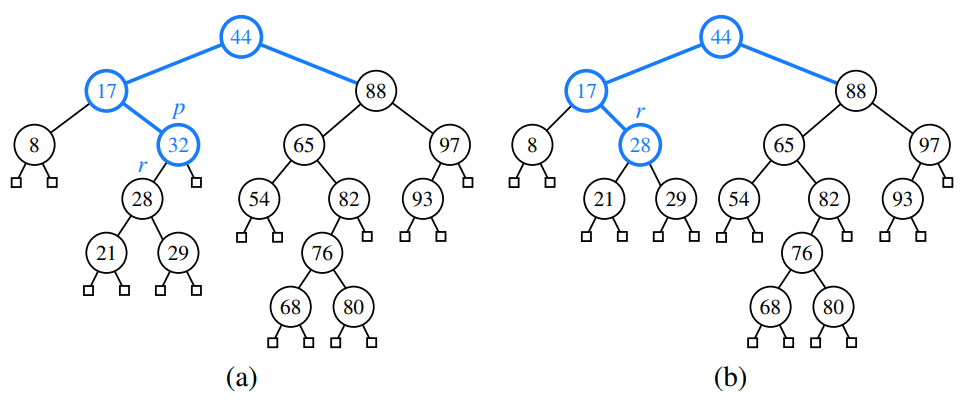




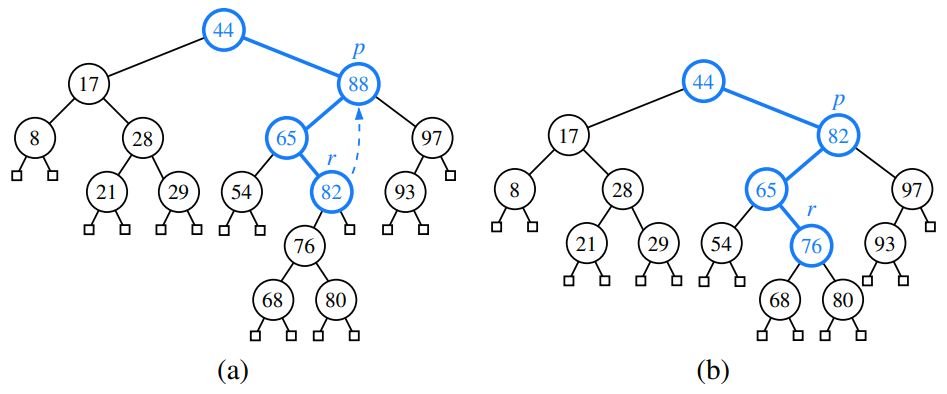
Deletion

Deleting an entry from a binary search tree is a bit more complex than inserting a new entry because the position of an entry to be deleted might be anywhere in the tree (as opposed to insertions, which always occur at a leaf).

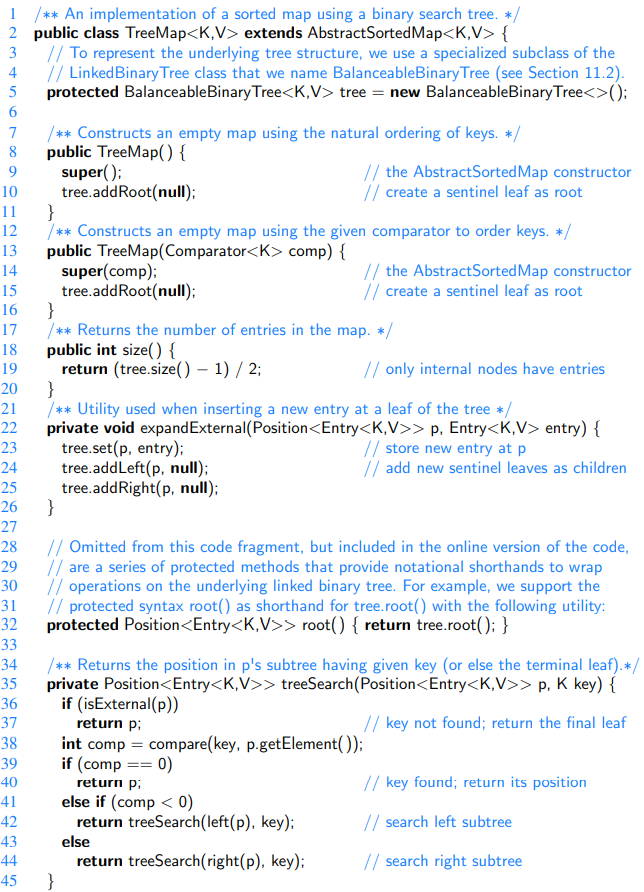
Deletion from the binary search tree, where the entry to delete (with key 32) is stored at a position p with one child r: (a) before the deletion; (b) after the deletion.

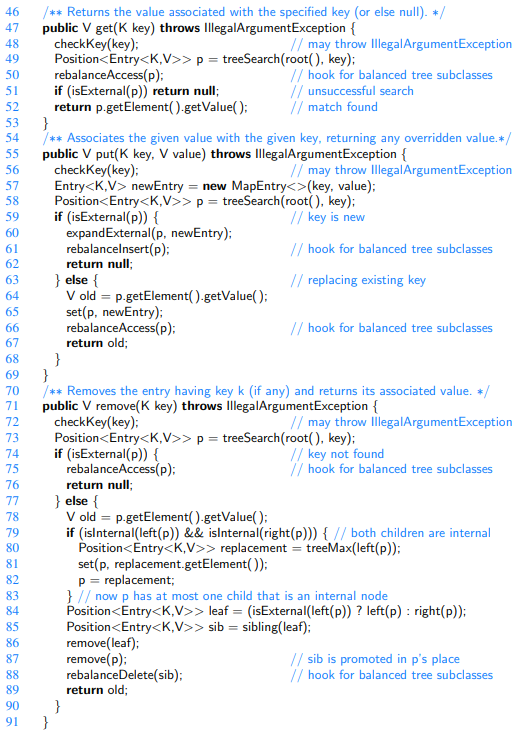


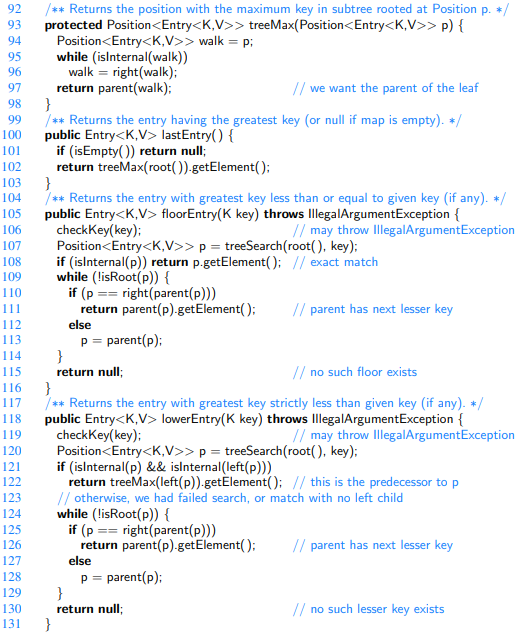
Deletion from the binary search tree of Figure 11.5b, where the entry to delete (with key 88) is stored at a position p with two children, and replaced by its predecessor r: (a) before the deletion; (b) after the deletion.

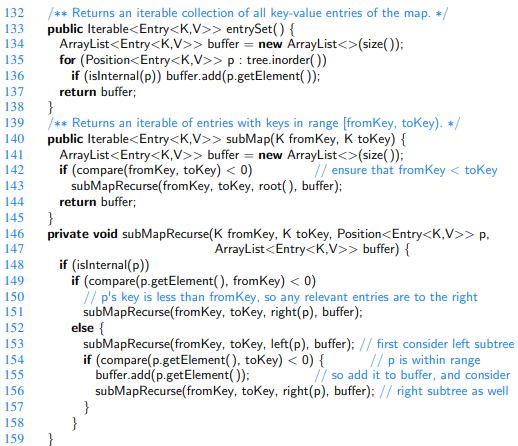


Java Implementation



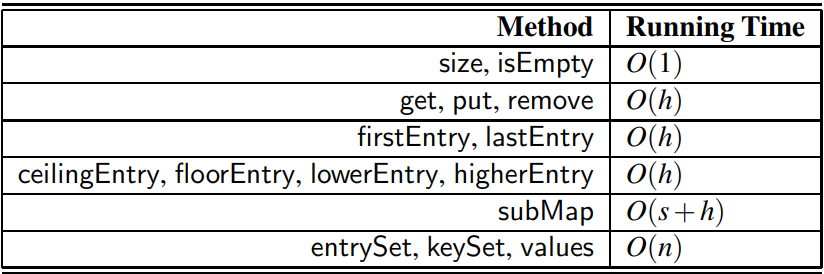






Performance of a Binary Search Tree

We denote the current height of the tree with h, and the number of entries reported by subMap as s. The space usage is O(n), where n is the number of entries stored in the map.



In the best case, T has height h = ⌈log(n+1)⌉−1, which yields logarithmic-time performance for most of 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 entries with keys in increasing or decreasing order.

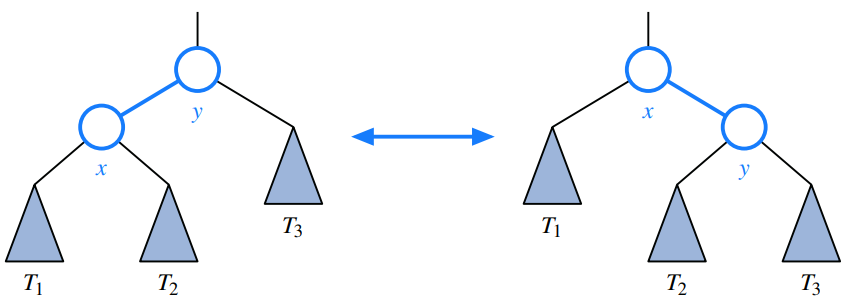
[We can nevertheless take comfort that, on average, a binary search tree with n keys generated from a random series of insertions and removals of keys has expected height O(logn); the justification of this statement is beyond the scope of the book, requiring careful mathematical language to precisely define what we mean by a random series of insertions and removals, and sophisticated probability theory.

In applications where one cannot guarantee the random nature of updates, it is better to rely on variations of search trees, presented in the remainder of this chapter, that guarantee a worst-case height of O(logn), and thus O(logn) worstcase time for searches, insertions, and deletions.]

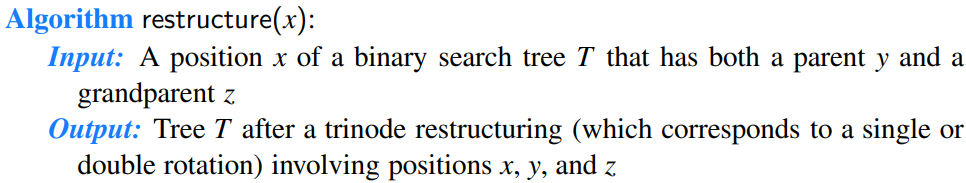
Balanced Search Trees

[In the closing of the previous section, we noted that if we could assume a random series of insertions and removals, the standard binary search tree supports O(logn) expected running times for the basic map operations. However, we may only claim O(n) worst-case time, because some sequences of operations may lead to an unbalanced tree with height proportional to n.]

The primary operation to *rebalance* a binary search tree is known as a **rotation**.

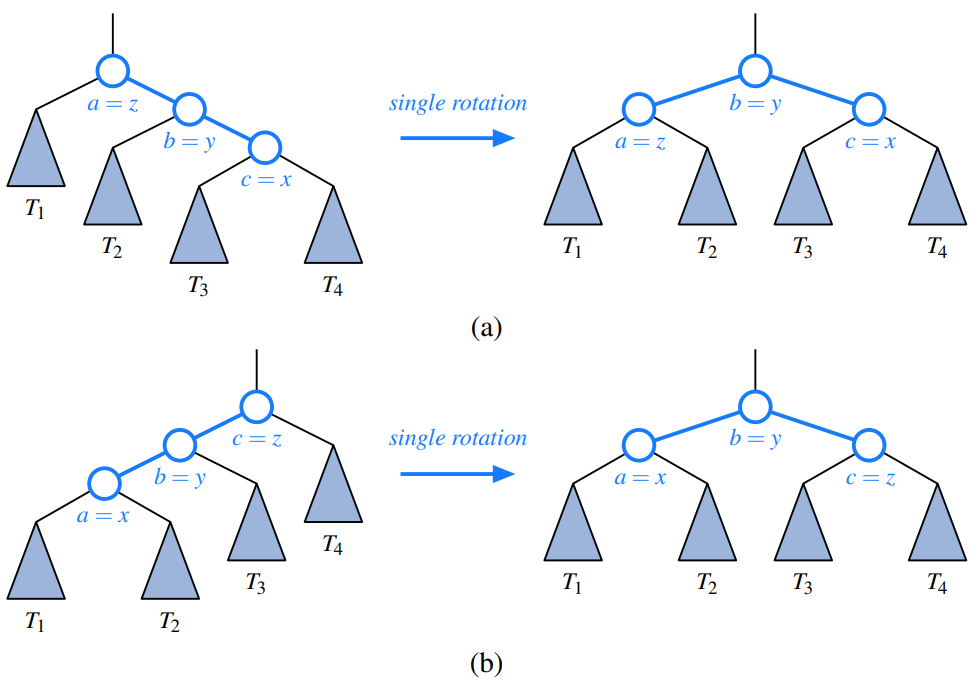


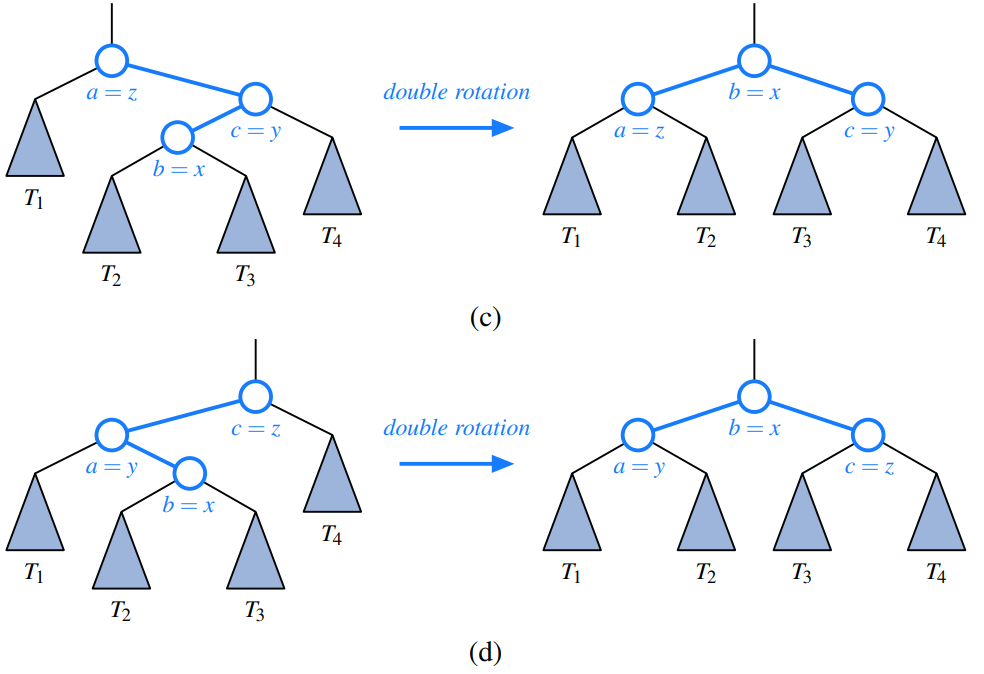
One or more rotations can be combined to provide broader rebalancing within a tree. One such compound operation we consider is a **trinode restructuring**. For this manipulation, we consider a position x, its parent y, and its grandparent z. The goal is to restructure the subtree rooted at z in order to reduce the overall path length to x and its subtrees.



[Watch videos to understand these concepts.]

Schematic illustration of a trinode restructuring operation: (a and b) require a single rotation; (c and d) require a double rotation.

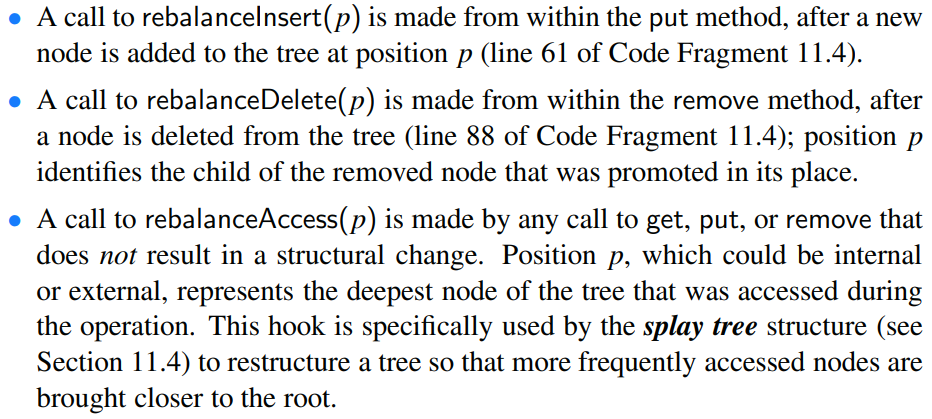




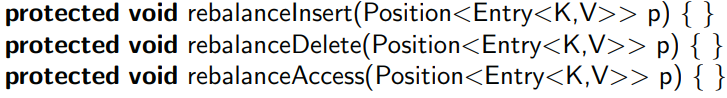
Java Framework for Balancing Search Trees

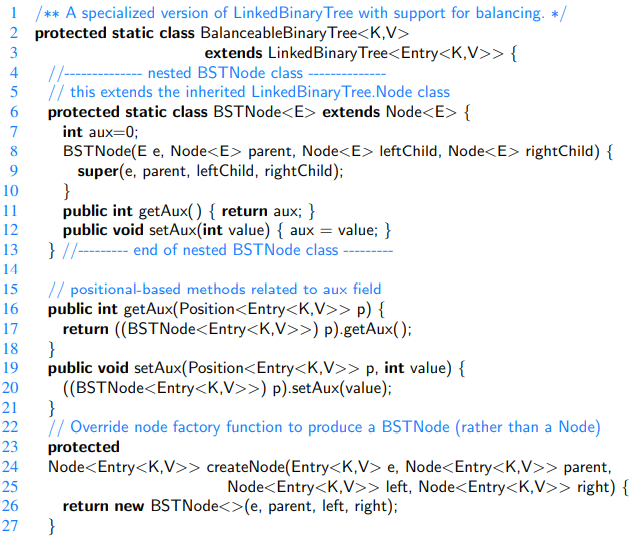
Hooks for Rebalancing Operations

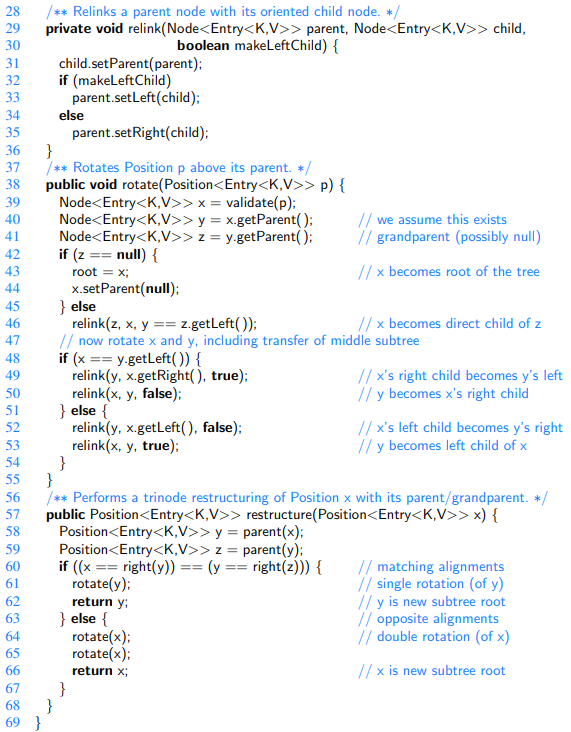
Our implementation of the basic map operations includes strategic calls to three nonpublic methods that serve as hooks for rebalancing algorithms:



This is another example of the template method design pattern.



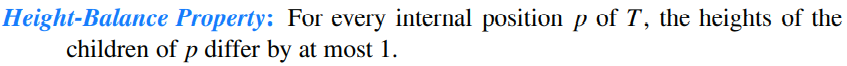




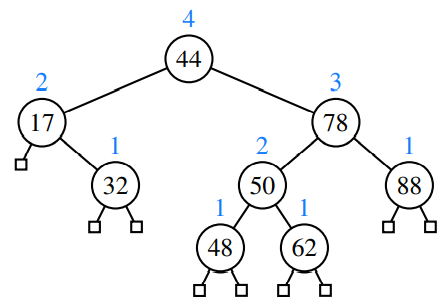
AVL Trees

Recall that we defined the height of a subtree rooted at position p of a tree to be the number of edges on the longest path from p to a leaf. By this definition, a leaf position has height 0.

In this section, we consider the following **height-balance property**, which characterizes the structure of a binary search tree T in terms of the heights of its nodes:



Any binary search tree T that satisfies the height-balance property is said to be an **AVL tree**. Example:



The keys of the entries are shown inside the nodes, and the heights of the nodes are shown above the nodes (all leaves have height 0).

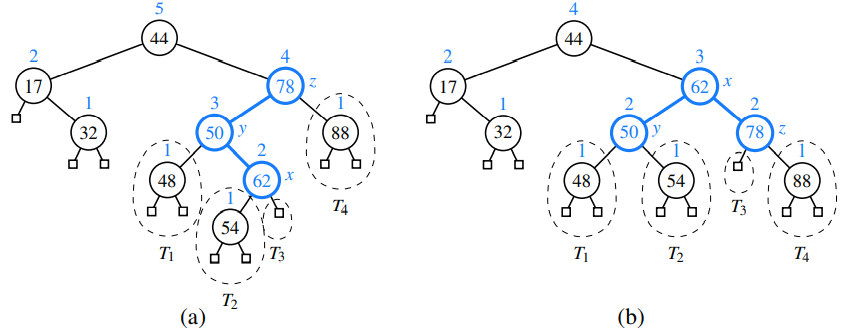
An immediate consequence of the height-balance property is that a subtree of an AVL tree is itself an AVL tree. The height-balance property also has the important consequence of keeping the height small, as shown in the following proposition.



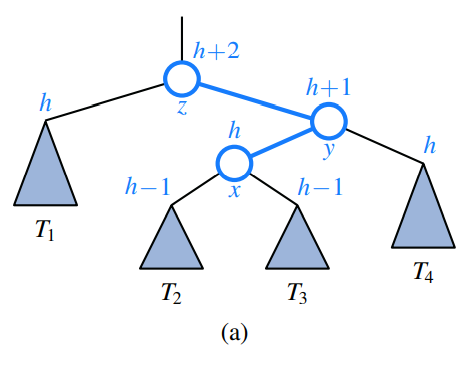
Update Operations

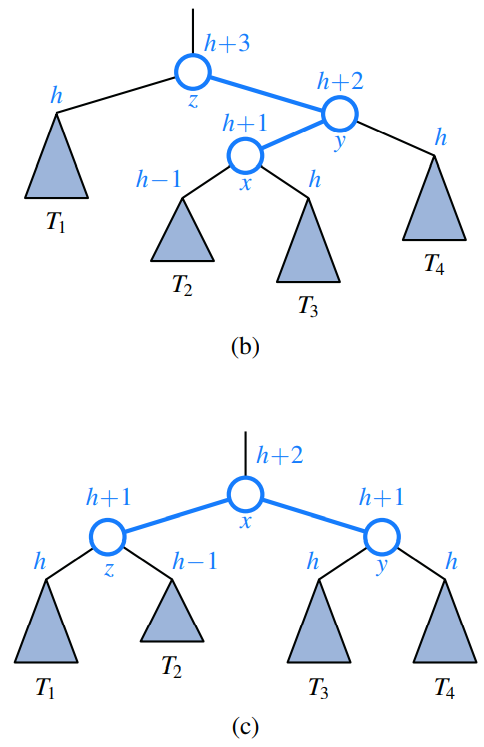
Given a binary search tree T, we say that a position is **balanced** if the absolute value of the difference between the heights of its children is at most 1, and we say that it is **unbalanced** otherwise.

Below: An example insertion of an entry with key 54 in the AVL tree: (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 above them, and we identify the nodes x, y, and z and subtrees T1, T2, T3, and T4 participating in the trinode restructuring.



Below: Rebalancing of a subtree during a typical insertion into an AVL tree: (a) before the insertion; (b) after an insertion in subtree T3 causes imbalance at z; (c) after restoring balance with trinode restructuring. Notice that the overall height of the subtree after the insertion is the same as before the insertion.

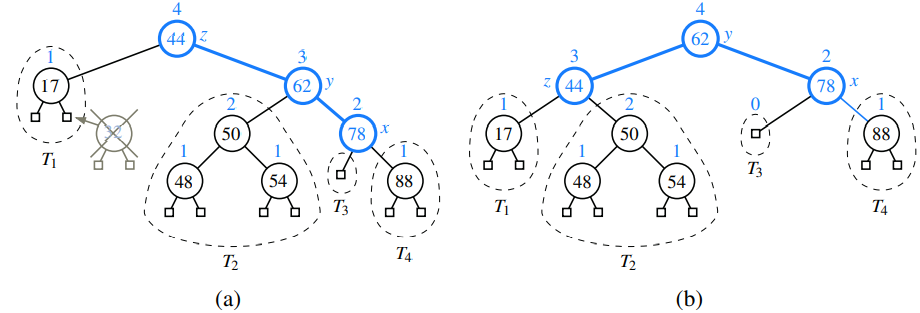




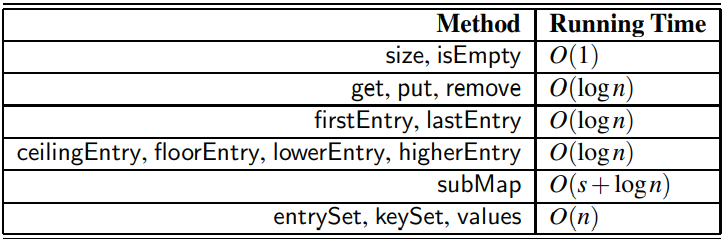
Deletion

Recall that a deletion from a regular binary search tree results in the structural removal of a node having either zero or one internal children. Such a change may violate the height-balance property in an AVL tree. In particular, if position p represents a (possibly external) child of the removed node in tree T, there may be an unbalanced node on the path from p to the root of T. In fact, there can be at most one such unbalanced node.

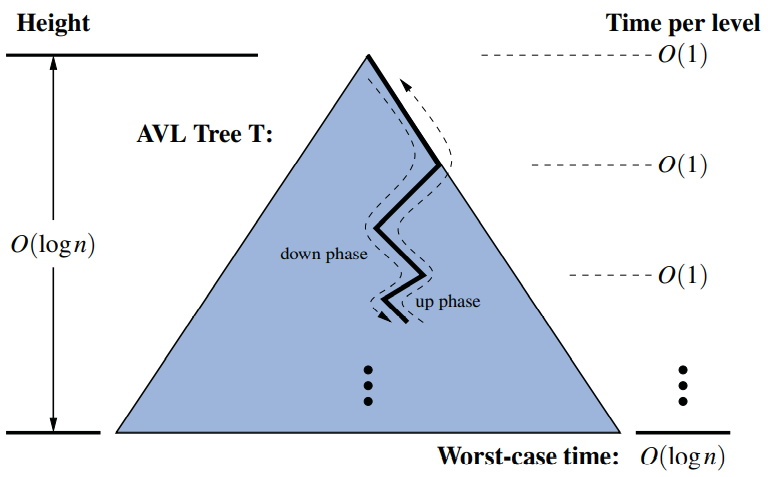
Below: Deletion of the entry with key 32 from the AVL tree: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a trinode restructuring of x, y, and z restores the height-balance property.



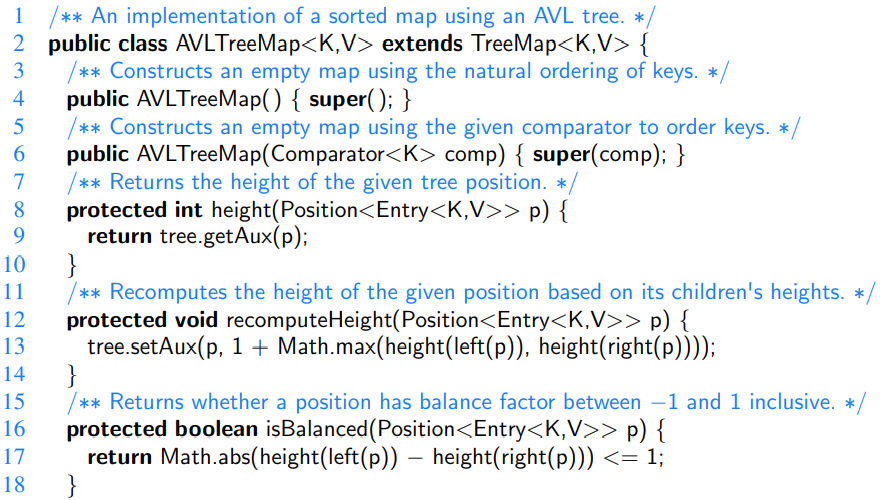
Performance of AVL Trees

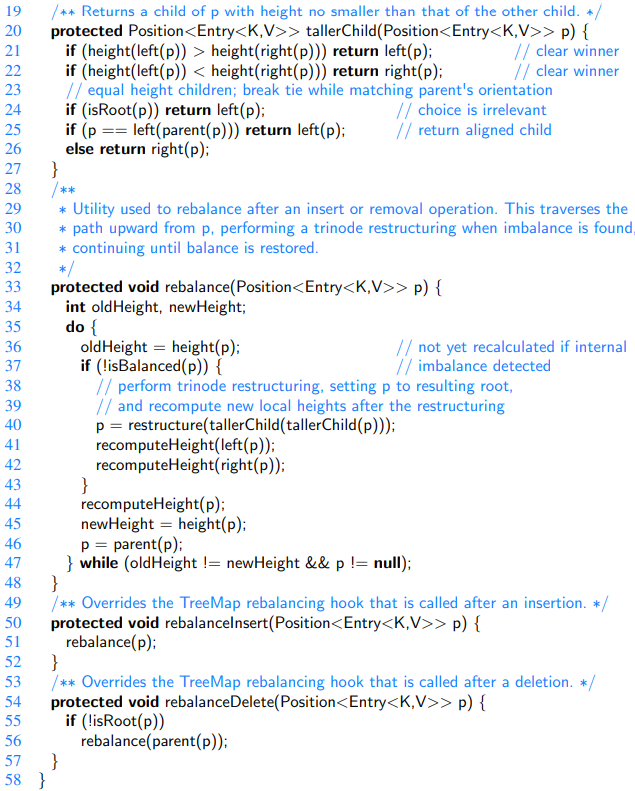


Below: Illustrating the running time of searches and updates in an AVL tree. The time performance is O(1) per level, broken into a down phase, which typically involves searching, and an up phase, which typically involves updating height values and performing local trinode restructurings (rotations).



Java Implementation





(2, 4) Trees

In this section, we will consider a data structure known as a **(2,4) tree**. It is a particular example of a more general structure known as a **multiway search tree**, in which internal nodes may have more than two children.

Map entries stored in a search tree are pairs of the form (k,v), where k is the **key** and v is the **value** associated with the key.

Definition of a Multiway Search Tree

Let w be a node of an ordered tree. We say that w is a **d-node** if w has d children. We define a multiway search tree to be an ordered tree T that has the following properties:

• Each internal node of T has at least two children. That is, each internal node is a d-node such that d ≥ 2.

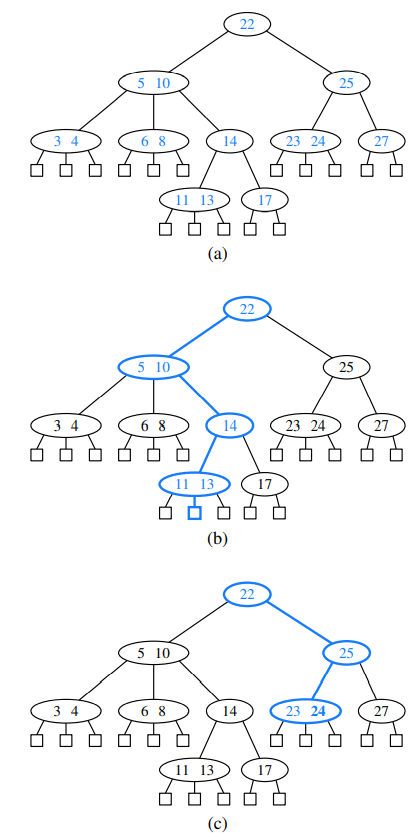
• Each internal d-node w of T with children c1,...,cd stores an ordered set of d −1 key-value pairs (k1,v1),..., (kd−1,vd−1), where k1 ≤ ··· ≤ kd−1.

• Let us conventionally define k0 = −∞ and kd = +∞. For each entry (k,v) stored at a node in the subtree of w rooted at ci, i = 1,...,d, we have that ki−1 ≤ k ≤ ki.

The above properties illustrated:

1. A multiway search tree T;
2. search path in T for key 12 (unsuccessful search);

(c) search path in T for key 24 (successful search).



Searching in a Multiway Tree

[Watch a video about it]

(2,4) Tree Operations

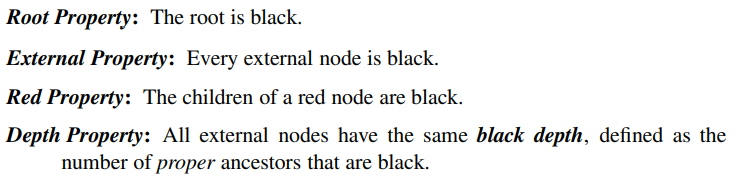
[Book 503(521)-510(528)]

[The slides seem to be better!]

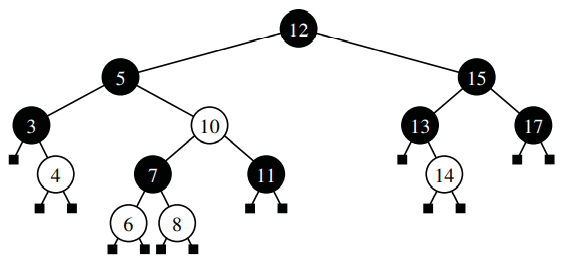
Red-Black Trees

[Although AVL trees and (2,4) trees have a number of nice properties, they also have some disadvantages. For instance, AVL trees may require many restructure operations (rotations) to be performed after a deletion, and (2,4) trees may require many split or fusing operations to be performed after an insertion or removal. The data structure we discuss in this section, the red-black tree, does not have these drawbacks; it uses O(1) structural changes after an update in order to stay balanced.]

Red-black tree properties:



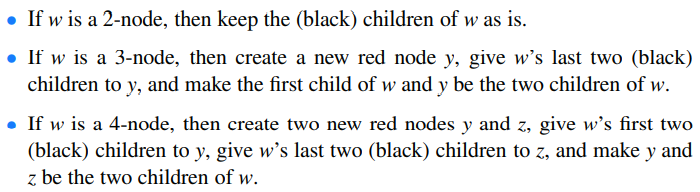
Example (with black-depth 3):



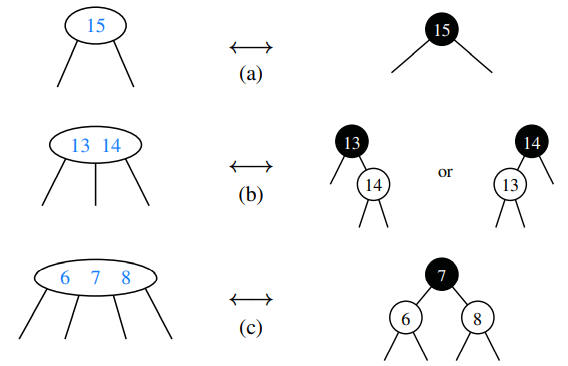
Given a red-black tree, we can construct a corresponding (2,4) tree by merging every red node w into its parent, storing the entry from w at its parent, and with the children of w becoming ordered children of the parent.

[Example for this in the book, bottom of page 510(528) tells where to find.]

Conversely, we can transform any (2,4) tree into a corresponding red-black tree by coloring each node w black and then performing the following transformations, as illustrated below:



[Notice that a red node always has a black parent in this construction.]





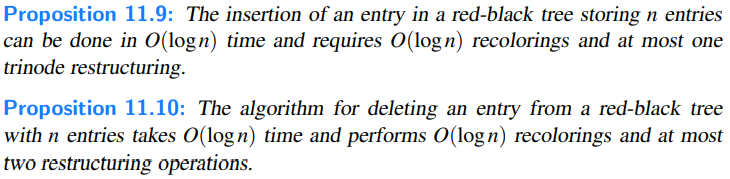
[Justification in book 512(530)]

Red-Black Tree Operations

[From book page 512(530) to 520(538).]

[Watch videos of it on youtube]

Performance of Red-Black Trees



Java Implementation

In this section, we will provide an implementation of a RBTreeMap class that inherits from the standard TreeMap class and relies on the balancing framework described earlier. In that framework, each node stores an auxiliary integer that can be used for maintaining balance information. For a red-black tree, we use that integer to represent color, choosing to let value 0 (the default) designate the color black, and value 1 the color red; with this convention, any newly created leaf in the tree will be black.

