6. Sequences and Comparison-Based Search Structures

If you are stranded on a desert island with only one data structure, this chapter will help convince you that the data structure of choice should be a *binary search tree*. Binary search trees and their relatives *skip lists* and *B-trees* (also discussed in this chapter) are extremely powerful and versatile structures. They are used for countless applications in practice, and they give us efficient ways to implement *sequences* and *dictionaries*, perhaps the most fundamental data structure types.

Sequences. An element in a dynamic sequence is inserted, deleted, or accessed according to its numeric index within the sequence. As we showed in Section 1.5, arrays and linked lists are the simplest data structures we have available for representing a sequence, but both have serious drawbacks: arrays take $\Theta(n)$ worst-case time to insert or delete elements in the middle of a sequence, and linked lists take $\Theta(n)$ worst-case time to scan to an element given its index. In this chapter, we will see how binary search trees and their relatives allow us to *access*, *modify*, *insert*, and *delete* elements anywhere in a sequence in only $O(\log n)$ time.

Dictionaries. In a dictionary, each record of data has an associated *key* that is used to access the record. For example, a set of student records might be keyed on name, allowing us to quickly look up the record for a student given their name. A dictionary might consist of nothing more than a set of individual elements (e.g., a literal dictionary, containing text strings). It can also serve as a fast indexing structure built on top of an existing database. For example, in our database of student records, we might build separate dictionary structures keyed on name and student ID, allowing fast lookups based on either field[1]. Dictionaries support the fundamental operations *insert*, *delete*, and *find*, where *find*(k) locates an element with key k. Binary search trees and the other structures we discuss in this chapter suppose these operations in $O(\log n)$ time.

Sets and Maps. A dictionary corresponds to notion of an abstract mathematical *set*, since it represents a set of keys and supports efficient set membership testing.

[1]As with many other data structures, we often don't store large multi-field data records directly inside a dictionary itself. Rather, each element in the dictionary contains a key and a pointer to the larger data record it represents. This involves less memory shuffling, and eliminates redundancy when we build multiple dictionaries to index the same larger database on different fields.

(a)

```
set<int> S; // S is a set of integers
int x;
while (cin >> x) {
 if (S.count(x) > 0) cout << "Value " << x << " duplicated.\n";
 else S.insert(x);
}
```

(b)

```
map<string, int> C; // C is a map from strings to integers
string s, most_freq = "";
while (cin >> s) {
 C[s] = C[s] + 1; // C[s] counts the occurrences of string s
 if (C[s] > C[most_freq]) most_freq = s;
}
cout << most_freq << " appears " << C[most_freq] << " times.\n";
```

FIGURE 6.1: Examples of sets and maps using the C++ Standard Template Library (which usually implements them using binary search trees under the hood): in (a), we check for duplicate integers appearing in the input, thereby solving the element uniqueness problem, and (b) determines a string appearing most frequently in our input. For those not accustomed to C++, `cin` and `cout` are used for input and output, and `S.count(x)` returns 1 if x is in S, 0 otherwise.

You will often hear dictionaries called sets; the two terms are more or less synonymous. By augmenting each element in a dictionary with an associated value, we get another common type of data structure called a *map*, which stores a collection of (key, value) pairs. Many programming languages provide built-in support for maps using array notation, where $A[k]$ refers to the value associated with key k. This allows us to pretend that we are indexing an "array" using strings or other complicated objects (sometimes this is known as an *associative array*). For example:

`grade["student name"] = 95.`

Due to its simple array-like appearance, novice programmers often fail to appreciate that underneath the hood, this statement is doing something very different from a standard array access — it is looking up the element with key "student name" in the dictionary named "grade", and assigning its associated value to 95. This mechanism also conveniently gives us the ability to emulate a large sparsely-filled array using memory proportional only to the number of elements actually stored in the array. For example, if we start with an empty structure and say

`is_prime[2305843009213693951] = true,`

this adds a single element to our structure, with key 2305843009213693951 and value true, taking dramatically less space than an actual physical array of length 2305843009213693951. For all practical purposes, however, we can still use this structure as if it were an actual array. Since binary search trees and their relatives support *find* in $O(\log n)$ time, accessing an element is only slightly slower than the $O(1)$ time you would get from a genuine array (and in the next chapter, we will see how to achieve constant access time using hash tables).

Additional prototypical examples of sets and maps in code are shown in Figure 6.1.

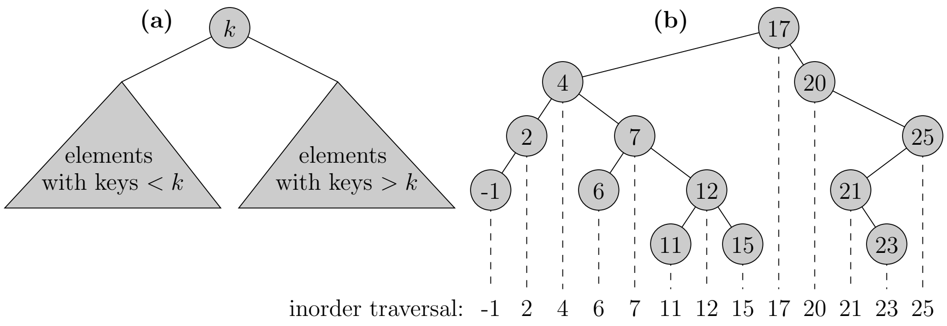


FIGURE 6.2: (a) The BST property, and (b) a BST above its inorder traversal, highlighting the way it is ordered from left to right.

Although a basic dictionary only needs to support *insert*, *delete*, and *find*, all of the structures in this chapter provide far more versatility, supporting many other useful operations:

- *Inexact search.* If *find*(k) fails to find a key of value k, we can still find elements in our dictionary with nearby keys.
- *Priority queue functionality.* These structures support *find-min* and *find-max*, and hence can also be used as priority queues (e.g., supporting *remove-min* by *find-min* plus *delete*).
- *Selection (rank-based access).* We can retrieve the element of any rank r. The special cases $r = 1$ and $r = n$ give the minimum and maximum, and $r = n/2$ gives the median.
- *Range queries.* We can quickly count or enumerate in sorted order the elements in any given range $[a, b]$.
- *Sorting.* We can output the contents of our dictionary in sorted order.

Of course, if all you need is a basic dictionary, these structures still work fine, although you may also want to consider using a *hash table*, the subject of the next chapter. Hash tables are extremely fast RAM dictionary structures (requiring integer keys) that support the basic *insert*, *delete*, and *find* operations in only constant time, but that lack much of the added versatility of binary search trees and their relatives.

6.1 The Binary Search Tree

As shown in Figure 6.2, a binary search tree (BST) is a binary tree whose elements satisfy the *binary search tree property*: for any element e in the tree (say, having key k), elements in e's left subtree have keys less than k, and elements in e's right subtree have keys greater than k. The BST property and the heap property (from the last chapter) are the two most common structural properties around which tree data structures are designed. We can think of the BST property as a "horizontal" property, with elements ordered from left to right, whereas the heap property is a "vertical" property, with elements ordered from top to bottom. To allow easy navigation, each element e in a BST typically maintains its key, $key(e)$, a parent pointer, $parent(e)$, and pointers $left(e)$ and $right(e)$ to its left and right children.

The BST property allows us to implement most BST operations in a straightforward recursive fashion. For example, in a BST with root element r:

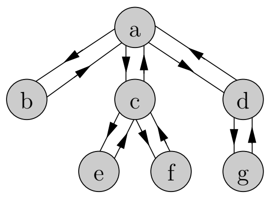
- To *find* an element with key k, the root is the answer if $k = key(r)$. Otherwise, we recursively call $find(k)$ on the left subtree of the root if $k < key(r)$, or on the right subtree of the root if $k > key(r)$. This explains how the binary search tree gets its name, since the recursive process of finding a key is analogous to the process of binary searching in a sorted array.
- To *insert* an element with key k, we recursively insert into the left subtree of the root if $k < key(r)$, or the right subtree of the root if $k > key(r)$.
- To *delete* an element e, we replace e with the merged contents of e's two subtrees using an operation called *join* with a simple recursive implementation that we will discuss shortly. Here (as in many other data structures), we define *delete* to take a pointer directly to an element of data, so we can separate its functionality from *find*. If all we know is the key k of the element we want to delete, we would instead call $delete(find((k))$.

These operations, as well as most other BST operations, run in $O(h)$ time on a BST of height h. This could be as bad as $\Theta(n)$ if the BST is nothing more than a long path of depth $n - 1$ (and we can easily end up with a BST in this shape, say if we insert its elements in sorted order). However, we will soon learn how to restructure a BST during insertions and deletions so it stays *balanced*. A balanced BST satisfies $h = O(\log n)$, so all fundamental operations run in $O(\log n)$ time. Note that a balanced tree does not need to be "perfectly" balanced, as, say, a binary heap, with every level completely filled in except the lowest. For now, we will continue quoting running times of $O(h)$ for our BST operations, but bear in mind that these will all become $O(\log n)$ once we learn how to maintain balance.

For simplicity, we assume that all keys in our BSTs (and other dictionary structures) are distinct. Duplicate keys are not particularly problematic, though. For example, we can maintain a single record in our structure for each distinct key value, which points to a linked list of all elements sharing that key. Sometimes we modify the BST property to specify where equal elements go, although this must be done with caution. For example, if we say that equal elements go in the left subtree, then a BST containing n equal elements would need to be shaped like a single left path, which is maximally unbalanced and hence very inefficient.

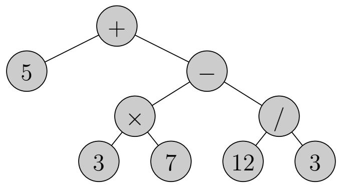
6.1.1 Traversals and Sorting

A *traversal* is a walk through a tree that enumerates its elements in some well-specified ordering. A particularly common BST traversal is the *inorder* traversal,



Euler tour traversal: a b a c e c f c a d g d a

(a)



Inorder traversal: $(5 + ((3 \times 7) - (12 \ / \ 3)))$
Preorder traversal: $+ \ 5 \ - \ \times \ 3 \ 7 \ / \ 12 \ 3$
Postorder traversal: $5 \ 3 \ 7 \ \times \ 12 \ 3 \ / \ - \ +$

(b)

FIGURE 6.3: Different tree traversals: (a) an Euler tour traversal, and (b) inorder, preorder, and postorder traversals of a binary tree that represents a mathematical expression; observe that these correspond to the infix, prefix, and postfix representations of the expression (so parsing an expression into a tree plus a traversal can be used to transform between these representations). For clarity, we have included parentheses in the inorder traversal.

which recursively traverses the left subtree of the root, then prints the root, then recursively traverses the right subtree of the root. This takes only $\Theta(n)$ time, and prints the contents of a BST in sorted order, as a consequence of the BST property.

The inorder traversal gives us yet another $O(n \log n)$ algorithm for sorting, sometimes called *BST sort*: first build a balanced BST on n elements by inserting each one in $O(\log n)$ time, then enumerate its contents in sorted order with a $\Theta(n)$ inorder traversal. Since the BST is comparison-based (along with every other data structure in this chapter), the $\Omega(n \log n)$ worst-case lower bound for comparison-based sorting implies a worst-case lower bound of $\Omega(\log n)$ on the time required to insert an element into a BST.

It is useful to think of the inorder traversal of a BST as a sequence of n elements "encoded by" the BST. Since we are currently using the BST to implement a dictionary, this sequence is the sorted ordering of the elements in the dictionary. We will revisit this viewpoint in a few pages when we learn how to encode an arbitrary sequence within the inorder traversal of a BST.

Traversals for Non-Binary Trees. As opposed to the inorder traversal, which only makes sense in binary trees, several other common types of traversals, shown in Figure 6.3, are used in both binary and non-binary trees:

- A *preorder* traversal prints the root, and then recursively visits the subtrees of the root from left to right. Preorder traversals are often used in tree computations that work downward from the root to the leaves.

- A *postorder* traversal recursively visits the subtrees of the root from left to right, then prints the root. Postorder traversals are often used in tree computations that work their way up from the leaves to the root.

- An *Euler tour*, or *Eulerian* traversal, walks "around the perimeter" of a tree as it prints the root, then the first subtree of the root, then the root again, then the second subtree of the root, and so on, ending with the root. The traversal is so-named because it corresponds an Eulerian cycle of the tree with its edges doubled (in fact, it is almost more of a traversal of the edges in a tree than its nodes). Each edge is followed twice (down and later up), so the Euler tour traversal of an n-node tree has length $2n - 1$. We could also call this a *depth-first* traversal, since it visits the same sequence of nodes as *depth-first search*, a common graph traversal algorithm we will learn in Chapter **??**.

All three of these traversals run in $\Theta(n)$ time on a rooted n-node tree. We will see another non-binary generalization of the inorder traversal similar to the Euler tour traversal when we study B-trees later in the chapter.

Problem 93 (Identifying Traversals). Given a sequence of n elements, devise an algorithm that tests in $\Theta(n)$ time whether this sequence corresponds to the inorder, preorder, postorder, or Euler tour traversal of some BST. You may assume the tree contains only distinct elements. [Solution]

6.1.2 Min and Max, and the BST as a Priority Queue

To find the minimum element in a BST in $O(h)$ time, we walk down the "left spine" of the tree, starting from the root and repeatedly following left child pointers until we reach an element with no left child. Similarly, we can compute the maximum element in a BST in $O(h)$ time by walking down its right spine. In a balanced BST, this allows us to implement the operations *find-min* and *find-max* in $O(\log n)$ time.

Since it supports *find-min* and *find-max*, the balanced BST can function as a priority queue whose asymptotic performance matches the binary heap. It can perform *insert* and *remove-min* (*find-min* followed by *delete*), as well as every other fundamental priority queue operation, in $O(\log n)$ time. However, the balanced BST is probably not the data structure of choice if all we need is a priority queue, since the binary heap is far simpler to implement. Furthermore, the balanced BST cannot match the performance of the fancier priority queues discussed in the last chapter (e.g., it cannot perform *decrease-key* in $O(1)$ amortized time like a Fibonacci heap).

6.1.3 Predecessor, Successor, and Inexact Search

The predecessor of element e, *pred*(e), is the element right before e in the inorder traversal (i.e., the next-smallest element in the BST). The successor of e, *succ*(e), is the element right after e in the inorder traversal (i.e., the next-largest element in the BST). Given a pointer to e, we can easily locate *pred*(e) and *succ*(e) in $O(h)$ time, as explained in Figure 6.4.

In addition to *pred*(e) and *succ*(e), it is also easy to implement the related operations *pred*(k) and *succ*(k) that return the closest element before or after a given key value k (if k is present in the BST, they return the element with this key, just like *find*). Implementing *pred*(k) and *succ*(k) in $O(h)$ time is quite simple: an unsuccessful call to *find*(k) will terminate on an element e that is either *pred*(k) or *succ*(k), after which a single call to either *pred*(e) or *succ*(e) can be used to locate the other of these two elements.

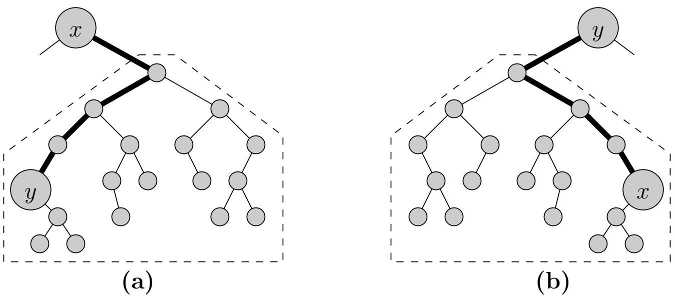


FIGURE 6.4: To compute *succ*$(x) = y$, we either (a) take the minimum element in x's right subtree, if x has a right child, or otherwise (b) find the first "right parent" we encounter walking upward from x. Similarly, to compute *pred*$(y) = x$, we either (b) take the maximum element in y's left subtree, if y has a left child, or otherwise (a) find the first "left parent" we encounter walking upward from y. Observe that these cases are completely symmetric; for example, in (a), if x is the first "left parent" of y, then y is the minimum in x's right subtree.

The operations above allow a BST to perform *inexact searches*, finding elements whose keys are close in value to a key we want. For example, when searching for a partial book title in a library catalog, we can find the closest matches using *pred*(k) or *succ*(k), then we can scroll through a list of nearby matches using repeated calls to *pred*(e) or *succ*(e), until we find the book we want. We can also answer a *range query* of the form "tell me all the elements in the range $[a, b]$ in sorted order" by first finding $e =$ *succ*(a), and then by repeatedly stepping from e to *succ*(e) until *key*$(e) > b$. As we see in the next problem, this actually runs quite fast.

Problem 94 (Successive Successors). *Pred*(e) and *succ*(e) have good amortized performance. Please show that if we start at the minimum element in a BST, only $\Theta(n)$ time is required for $n - 1$ successive calls to *succ*(e) (and that this essentially performs an inorder traversal of the tree). Furthermore, show that performing k repeated calls to *succ*(e) starting from any element in a BST (say, in response to a range query asking us to output all the elements in some interval $[a, b]$) requires only $O(h + k)$ time. This doesn't quite fit our standard definition of amortized running time, but we could say repeated calls to *succ* (or repeated calls to *pred*) run in $O(1)$ amortized time as long as we pay a one-time penalty of $O(h)$ up front. [Solution]

Deletion After Swapping with a Predecessor or Successor. Our preferred method for deleting an element in a BST is by replacing it with the *join* of its two subtrees (we discuss the join operation in a moment). However, another common method for deletion involves the use of *pred* and *succ*. It is trivial to delete an element with no children (i.e., a leaf), and also easy to delete an element with only one child, since we can just replace the element with its child. To delete an element e with two children, we note that both *pred*(e) and *succ*(e) can have at most one child, so we can first swap e with one of these elements, and then delete e from its new location, where it now has at most one child. Replacing e with *pred*(e) or *succ*(e) is "safe", since this does not create any violation of the BST property.

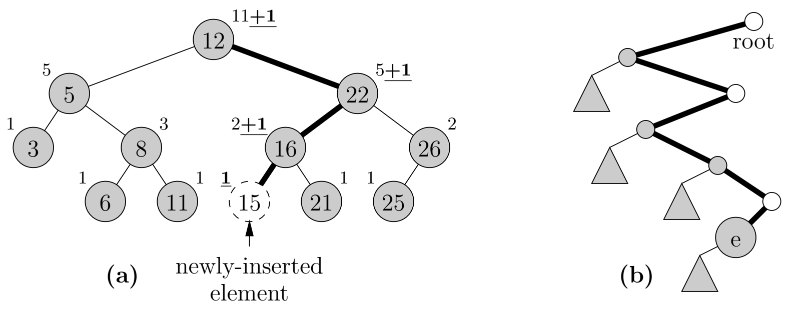


FIGURE 6.5: (a) Incrementing subtree sizes along the path down to a newly-inserted element, and (b) the set of all elements (shaded) with keys no larger than *key*(e); the number of such elements is the rank of e.

6.1.4 Augmenting a BST, Select, and Rank

The BST can be modified to support *rank-based* access, via the following operations:

- *Select*(r). Returns the element having the rth largest key (i.e., the element of rank r, or equivalently the element at index r within an inorder traversal). The minimum element has rank $r = 1$, the maximum has rank $r = n$, and the median has rank $r = n/2$.

- *Rank*(e). Returns the rank of element e — that is, the index of e within an inorder traversal (this is the "inverse" of the *select* operation).

To implement these both in $O(h)$ time, we first need to augment each element e in our BST with its subtree size, *size*(e). We keep this information up to date by modifying *insert* and *delete*. When a new element e is inserted, we increment the subtree sizes along the path from the root down to e (as shown in Figure 6.5(a)). Deletion of e decrements these values. Note that these changes preserve the asymptotic $O(h)$ running times of *insert* and *delete*.

The rank of e is the number of BST elements e' with *key*(e') $\leq$ *key*(e). As shown in Figure 6.5(b), these elements e' include all the "left parents" of e along the path up to the root, as well as their left subtrees. To implement *rank*(e) in $O(h)$ time, we count these while walking from e up to the root. The size of a subtree is counted quickly by inspecting its root, thanks to our augmented information.

To implement *select*(r) in $O(h)$ time, we compute the rank R of the root by adding one to its left subtree size. If $r = R$, the root is the answer. Otherwise, if $r < R$, we recursively call *select*(r) in the left subtree of the root, and if $r > R$ we recursively call *select*($r - R$) in the right subtree of the root. This approach is directly analogous to the quickselect algorithm for selection from an array, so we can view the BST augmented with subtree sizes to support *select* and *rank* (sometimes known as an *order statistic tree*) as a dynamic version of quickselect.

Select and *rank* are just two of many extended operations we can build by carefully augmenting a BST. As another simple example, by augmenting each element

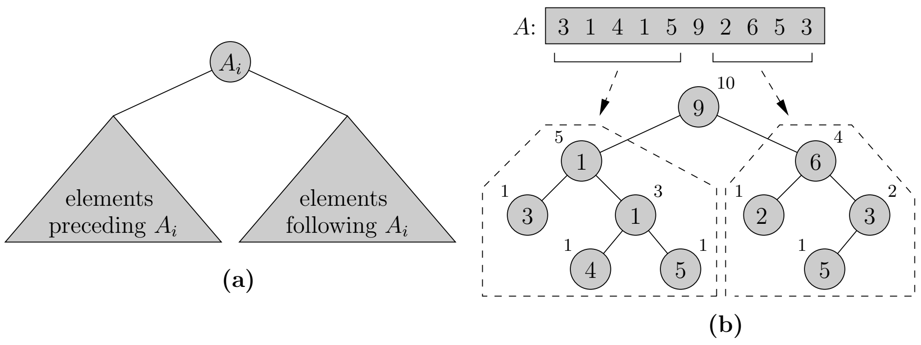


FIGURE 6.6: Encoding a sequence $A_1 \ldots A_n$ as the inorder traversal of a BST.

with direct pointers to its predecessor, successor, and the minimum and maximum elements in its subtree, we get $O(1)$ worst-case implementations of *find-min*(e), *find-max*(e), *pred*(e), and *succ*(e) (note that we can maintain this extra information without compromising the $O(h)$ running time of *insert* or *delete*).

Problem 95 (Randomly Sampling from a BST). If each element e in a BST has an associated frequency count f_e, how can we augment our BST to support the ability to sample a random element e with probability proportional to f_e in $O(h)$ time, assuming we can generate a random number in $[0, 1]$ in $O(1)$ time? See also problem 114. [Solution]

Problem 96 (Inversion Counting with a BST). Please show how to count the number of inversions in a length-n array in $O(n \log n)$ time using a BST (see also problems 61 and 122). For simplicity, please assume the BST is balanced, so every BST operation runs in $O(\log n)$ time (we will learn how to achieve balance shortly). [Solution]

6.1.5 Encoding a Sequence in a BST

The BST is quite effective as a dictionary data structure. In this section, we discuss the second major application of the BST: representing an arbitrary n-element sequence $A_1 \ldots A_n$. As shown in Figure 6.6, the natural modification of the "BST property" in this case ensures that the sequence corresponds to the inorder traversal of the tree. We store some element A_i at the root, we store $A_1 \ldots A_{i-1}$ in the left subtree of the root, and we store $A_{i+1} \ldots A_n$ in the right subtree of the root.

Since our elements are not sorted from left to right, the *find* operation is now meaningless and no longer used. Instead, we access elements in the sequence by index using *select* and *rank* (so it is necessary to augment the tree with subtree sizes). *Select*(r) returns the element at index r within the sequence, and *rank*(e) returns the index of element e. We can also use *pred* and *succ* to step between consecutive sequence elements, and *find-min* and *find-max* to jump directly to the beginning or end of the sequence. Elements are inserted by rank (index) as well, using *select* to find the proper location within the tree to attach a new element.

Recall from Section 1.5 that arrays and linked lists both have significant drawbacks when representing a dynamic sequence. The array can access elements by index in

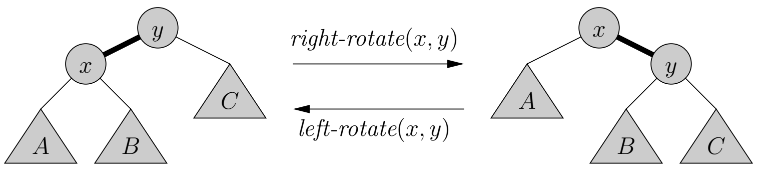


FIGURE 6.7: The *left-rotate* and *right-rotate* operations about an edge from x to y. Both directions preserve the inorder traversal of the BST: contents of A, x, contents of B, y, contents of C.

only $O(1)$ time, but it takes potentially $\Theta(n)$ time to insert and delete elements in the middle of the sequence. The linked list has the opposite trade-off: $\Theta(n)$ time to access an element by index, but then $O(1)$ time to insert or delete. The balanced BST balances these two extremes, supporting all operations in $O(\log n)$ time.

Since we no longer search by key when we encode a sequence, it makes slightly less sense to call this usage of our data structure a "binary search tree", and some would argue that "order statistic tree" is more appropriate in this context. However, we use the term "binary search tree" for both cases, since it really is the same data structure playing both roles, and all of our ensuing discussion (e.g., with maintaining balance) applies in both cases. In both roles, the BST is perhaps best viewed as a structure that fundamentally encodes a sequence from left to right within its inorder traversal. When used as a dictionary, this sequence is the sorted ordering of the elements in the dictionary, enabling key-based access as well as rank-based access.

6.1.6 Rotations

Soon, we will discuss methods to keep a BST balanced, by strategically moving elements during *insert* and *delete* via some mechanism that preserves the BST property. The most common such mechanism is a *rotation* about some edge in the tree. Rotations come in two symmetric flavors, *left rotations* and *right rotations*. As seen in Figure 6.7, the inorder traversal of a BST is unchanged after either direction of rotation. The reader can probably guess how these might be used to maintain balance. For example, if subtree A in Figure 6.7 becomes too deep, a right rotation may improve the overall balance of the tree.

We occasionally use rotations to pull an element to the root of a tree (by repeatedly rotating with its parent), or down to a leaf (by repeatedly rotating with a child). For example, we could delete an element in $O(\log n)$ time with high probability by removing it after first re-locating it to a leaf by repeatedly rotating it with a random child — effectively following a random null path, as we did back in Section 5.4.1.

Problem 97 (Rotation Distance). The *rotation distance* between two n-element binary trees T_1 and T_2 is the minimum number of rotations required to transform T_1 so that it has the same "shape" as T_2. The question of how efficiently one can compute the exact rotation distance between two trees is actually an open problem. Here, see if you can devise an algorithm that performs $O(n)$ rotations to transform between any pair of n-element trees provided as input. [Solution]

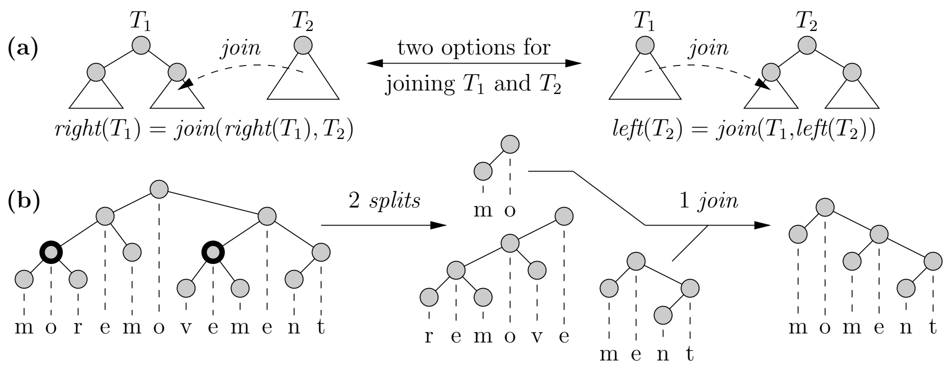

(a)
T_1
T_2
join
two options for joining T_1 and T_2
T_1
T_2
join
$right(T_1) = join(right(T_1), T_2)$
$left(T_2) = join(T_1, left(T_2))$
(b)
2 splits
1 join
m o r e m o v e m e n t
m o
r e m o v e
m e n t
m o m e n t

FIGURE 6.8: (a) Recursively joining two BSTs, and (b) using split and join to cut a substring out from the sequence encoded by a BST. The bold elements in the leftmost BST are the ones on which we call *split*.

Problem 98 (Balancing a Tree with Rotations). A BST with $n = 2^x - 1$ elements can be restructured so as to be "perfectly" balanced, with every level filled in completely. We could do this in $\Theta(n)$ time with divide and conquer: start with an inorder traversal, place the middle element at the root, then recursively build perfectly balanced subtrees on the left and right. Here, we develop a more "in place" approach, based on rotations. First, show how to transform any BST into a right path in $\Theta(n)$ time. Next, show how to make multiple passes down this path while performing left rotations (each time roughly halving the path length), so that in $\Theta(n)$ time we end with a perfectly-balanced BST. If you are feeling ambitious, show how this approach can be extended to BSTs of arbitrary size, so that $|size(left(e)) - size(right(e))| \leq 1$ holds for every element e in the final balanced tree (these trees are similar to *Braun trees* — problem 90). [Solution]

6.1.7 Split and Join

Two elegant but often under-appreciated BST operations are *split* and *join*, which are defined as follows in the context of a dictionary:

- *Split*(e). Break a BST into two BSTs, one containing keys at most *key*(e), the other containing keys greater than *key*(e).
- *Join*(T_1, T_2). Join two BSTs T_1 and T_2 (with all keys in T_1 being less than those in T_2) into a single BST.

Split is easy to implement in $O(h)$ time by rotating e up to the root and detaching its right subtree. Equivalently, we can use recursion: detach e's right subtree if e is the root, or otherwise recursively split the subtree containing e. To *join* T_1 and T_2, we could rotate the maximum element of T_1 to its root and attach T_2 as its right child, or alternatively rotate the minimum of T_2 to its root and attach T_1 as its left child. One can also implement *join* recursively (the author's preference), as shown in Figure 6.8(a), by taking T_1 and recursively joining it with the left subtree of T_2, or by taking T_2 and recursively joining it with the right subtree of T_1. Both approaches run in time bounded by the combined heights of T_1 and T_2.

Insert and *delete* can be easily built using *split* and *join*. To delete element e, we replace e with the *join* of its two subtrees. To insert element e with key k, we can split our tree on $pred(k)$, then join the resulting two pieces back together as children of e. Whereas the "standard" BST *insert* procedure inserts an element as a leaf, this method inserts a new element at the root.

Cutting and Linking BSTs and Sequences. When performing a range query for all elements in some range $[a, b]$, we can use two calls to *split* to obtain a BST with keys smaller than a, a BST with keys in $[a, b]$, and a BST with keys larger than b. By joining the first and last BSTs back together, we have effectively pulled the BST representing the answer to our query completely out of the original tree. If our trees represent sequences instead of dictionaries, this same approach allows us to efficiently *cut* a smaller sequence out of the original (shown in Figure 6.8(b)), after which we can *paste* it back in at any location with one call to *split* and two calls to *join*. We will build on this idea further when we introduce *dynamic tree* data structures in Section 8.5.

6.1.8 Maintaining Augmented Data

One issue to keep in mind when we perform rotations (or any other operation that modifies the structure of a BST, such as *insert*, *delete*, *split*, or *join*) is how to keep augmented information in the tree (e.g., subtree sizes) up to date. This is a general concern with data structure design — we often augment a structure to give it extra functionality, but on the other hand, we now need to spend extra work maintaining this augmented data whenever the structure is modified.

Most common types of BST augmentations are *locally* recomputable, where we can recompute the augmented information attached to e in $O(1)$ time in response to a change in the augmented information of one of e's children. For example, $size(e) = 1 + size(left(e)) + size(right(e))$ can be updated in $O(1)$ time if $size(left(e))$ or $size(right(e))$ changes. As a general rule, any locally-recomputable augmentation can be easily maintained without affecting the asymptotic running time of any standard operation that changes the structure of a BST (e.g., *insert*, *delete*, *split*, or *join*)[2]. The intuition behind this is as follows: a modification originating at some element e could cause a chain of augmentation updates, each in $O(1)$ time, propagating from e up to the root (say, along a path of length p). However, since the current operation is modifying e to begin with, it presumably has already walked down this path to reach e, so it has already spent $O(p)$ time.

6.2 Balanced Binary Search Trees

The key to good performance with a BST is maintaining balance, so $h = O(\log n)$. Many balancing approaches appear in the literature and in practice, and we have chosen several these to present here, since each one highlights useful and elegant techniques for data structure design.

[2]Moreover, this will remain true even when we study B-trees later in the chapter, even though nodes in a B-tree can have many more than two children.

Balancing mechanisms fall into three groups. We begin this section with *worst-case* balancing mechanisms, which maintain $h = O(\log n)$ always and which provide $O(\log n)$ worst-case guarantees for all standard BST operations. We then discuss *randomized* mechanisms, where balance is always maintained with high probability, and all operations therefore run in $O(\log n)$ time with high probability. Finally, we discuss *amortized* mechanisms, where strict balance is not always maintained, but any sequence of k operations still takes $O(k \log n)$ worst-case time.

6.2.1 Height-Balanced (AVL) Trees

A tree is *height-balanced* if for every element e, the heights of e's left and right subtrees differ by at most one. Height-balanced trees are also known as *AVL trees* after their inventors Adel'son-Vel'skiĭ and Landis. An easy induction proof shows that $h = O(\log n)$ for any height-balanced tree. [Details]

To implement an AVL tree, we augment each element with the height of its subtree; this is a locally-recomputable augmentation, so it does not affect the asymptotic running time of *insert* or *delete* to maintain. Every time we insert or delete an element, we check to make sure the resulting tree remains height-balanced. If not, we can restore height balance in $O(\log n)$ time by performing a small number of carefully chosen rotations. [Details]

6.2.2 Balancing Based on Subtree Size

Let α be any constant in $[1/2, 1)$. A tree is α-balanced if for every element e, both the left and right subtrees of e contain at most $\alpha \cdot size(e)$ elements. Since our subtree size drops by a constant factor every step down the tree we take, it is easy to see that $h = O(\log n)$ for any α-balanced tree.

Balancing mechanisms based on this idea in the literature are known as "weight-balanced" trees or trees of "bounded balance" (sometimes called BB[α] trees). As with the AVL tree, they restore balance after an insertion or deletion breaks the α-balance property by performing a small number of rotations in $O(\log n)$ time (for appropriate values of α). The details are somewhat messy[3], however, and are omitted from our present discussion. Instead, we will focus on the use of this property as part of a simple amortized balancing scheme described in a few pages.

6.2.3 Red-Black and Path-Balanced Trees

Perhaps due to their prominence in a number of well-known algorithms texts, *red-black* trees seem to be particularly popular in practice. We color every element in a BST either red or black such that two invariants are maintained: the number of black elements on every root-to-leaf path must be the same, and we cannot have an adjacent (parent-child) pair of red elements. A simple induction argument shows that this invariant implies that $h = O(\log n)$. [Details]

[3]The details are messy enough that the original journal paper describing this balancing mechanism contained subtle errors in its analysis.

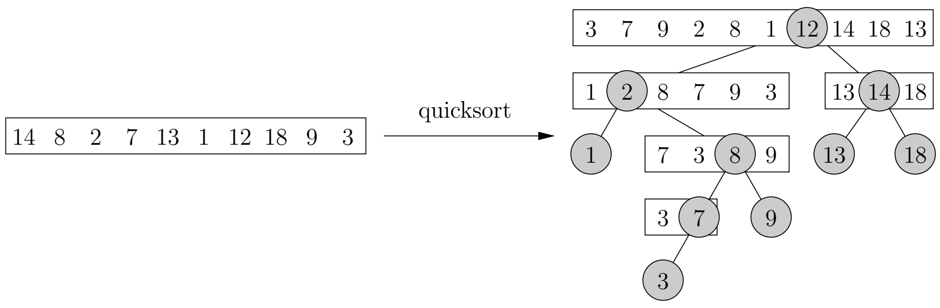


FIGURE 6.9: The tree on the right shows the recursive process of quicksorting an array, with pivots circled. For example, the top row shows the array after partitioning about the first pivot, 12. The pivots form the BST we would get by inserting elements in the order they were selected as pivots; for example, 12 (the first pivot) is inserted first, and so becomes the root. Moreover, the same comparisons happen during quicksort and the process of building this BST, just in a different order: all elements are compared to 12 during the initial partition operation in quicksort, while during BST construction they are all compared to 12 as they are inserted and make their way down the tree.

After our invariant breaks due to an insertion or deletion, we can restore it in $O(\log n)$ time by rotations and by recoloring elements. We omit the somewhat tedious details of how this is done, since the red-black balancing mechanism turns out to be equivalent to the conceptually simpler mechanism used to balance a “2-3-4 tree”, which we cover later in the chapter.

A nice mathematical generalization of the red-black tree goes by the name of a *path-balanced* tree, where edges are assigned numeric weights such that all root-to-leaf paths have the same sum. For the specific case of a red-black tree, an edge leading down into a red node has weight zero and an edge leading into a black node has weight one. Rebalancing in this setting involves a mixture of edge reweightings and rotations in order to achieve some desired property of the weight sequences (e.g., no two adjacent zeros, for the red-black tree), although as with the red-black tree, the details are often somewhat cumbersome. At the end of this the chapter, however, we highlight a simple randomized mechanism (equivalent to a *skip list*, a close relative of the BST) that keeps a tree path-balanced, with all operations running in $O(\log n)$ time with high probability.

6.2.4 Randomly-Structured Binary Search Trees

When we studied quicksort, we alluded to its strong similarities with the BST, which essentially encodes the tree of all recursive subproblems generated by quicksort. As explained in Figure 6.9, the process of building a BST performs exactly the same comparisons as quicksort, just in a different order, allowing us to deduce properties of BSTs given what we already know about quicksort. For example, the recursion depth in quicksort corresponds to the height of its associated BST. This is particularly useful for *randomly-structured* BSTs — built from an initially-empty

BST by inserting n elements in random order — since this process is analogous to randomized quicksort. Given this direct correspondence, we can now leverage our knowledge of randomized quicksort to conclude that the following all hold in expectation and with high probability:

Randomized quicksort		Randomly-structured BSTs
Spends $O(\log n)$ comparisons on any specific array element	$\leftrightarrow$	Depth of any specific element is $O(\log n)$
Spends $O(\log n)$ comparisons per element on every array element	$\leftrightarrow$	Height of entire tree is $O(\log n)$
$O(n \log n)$ total comparisons	$\leftrightarrow$	$O(n \log n)$ total time to build

Unfortunately, even though a randomly-built BST starts out balanced, subsequent insertions and deletions (which are not random, since they are controlled by the user) may lead to imbalance over time. However, we can cleverly modify *insert* and *delete* to keep the tree always randomly structured, as if we had just built it randomly from scratch. This approach, which we call the *randomized BST* (RBST), yields a tree that always satisfies $h = O(\log n)$ with high probability.

To *insert* an element e into an n-element tree while preserving its random structure, we flip a biased coin and with probability $\frac{1}{n+1}$, we insert e at the root[4], either using *split* as described in Section 6.1.7, or equivalently by inserting e at a leaf as in a standard unbalanced BST, then rotating it up to the root. Otherwise, we recursively insert e into either the left or right subtree of the root, based on its key or intended rank (this recursive call again flips a coin to determine if e should be inserted at the root of its subtree, and so on). [Proof that this keeps the tree randomly structured]

To *delete* element e, we replace e by the result of joining its two subtrees, using a biased version of the recursive *join* operation (Figure 6.8(a)) that preserves random structure. It takes two randomly-structured trees T_1 and T_2 (with elements of T_1 preceding those in T_2) and joins them into a single randomly-structured tree. Suppose $n = n_1 + n_2$, where n_1 and n_2 denote the number of elements in T_1 and T_2. With probability n_1/n we recursively join T_2 with the right subtree of T_1's root and otherwise (with probability n_2/n), we recursively join T_1 with the left subtree of T_2's root[5]. [Proof that this keeps the tree randomly structured]

6.2.5 Treaps

The *treap* is a curious hybrid between a binary search *tree* and a binary *heap*. As shown in Figure 6.10, each element in a treap contains two keys, a "BST" key and "heap" key. The BST keys satisfy the BST property, and the heap keys satisfy the heap property. If you ignore the heap keys, the structure therefore looks like a BST, and if you ignore the BST keys, the structure looks like a heap.

The treap gives us another simple randomized mechanism for maintaining balance in a BST by keeping it in a randomly-structured state. It is actually equivalent to the RBST mechanism we just described, even though it may look quite different (in the

[4]This seems reasonable since if we had built the resulting $(n+1)$-element tree randomly from scratch, there would be a $\frac{1}{n+1}$ chance of e being inserted first, and hence located at the root.

[5]These bias factors seem reasonable, since we want every element in the joined tree to have an equal probability of being its root. If T_1 has twice as many elements as T_2, it should therefore be twice as likely that T_1's root (i.e., a random element in T_1) stays at the root of the joined tree.

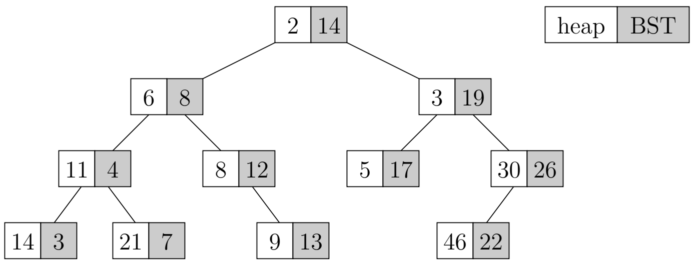


FIGURE 6.10: An example of a treap.

literature, treaps were proposed several years before RBSTs). If we take a BST and assign each element a random distinct heap key, then there is only one valid "shape" the resulting treap can assume, and this is the shape of a randomly-structured tree. The element with minimum heap key (effectively a random element) must reside at the root, and then the BST property dictates which elements must belong to the left and right subtrees. The roots of these subtrees are again uniquely determined by the heap property, and so on. The resulting tree is the same we would get by inserting all elements into a BST in order by their heap keys, which is random order. Therefore, a treap with randomly-chosen distinct heap keys is balanced with high probability. The same is true even if heap keys are not necessarily distinct, as long as they are randomly chosen from a sufficiently large range.

Insertion and deletion in a treap requires maintaining both the BST and heap properties. To *insert* a new element e, we assign it a random heap key and then use the standard BST insertion procedure to insert e as a leaf in our tree, ignoring heap keys for the moment. This satisfies the BST property, but we may now violate the heap property between e and $\mathit{parent}(e)$. To fix this, we use the standard *sift-up*(e) heap operation (Section 5.3.2), which we now implement using rotations so it maintains the BST property. Deletion of an element e can be done several ways; perhaps the cleanest is to replace e with the join of its two subtrees T_1 and T_2, where now the usual choice between "recursively merge T_1 into the left subtree of T_2" or "recursively merge T_2 into the right subtree of T_1" (Figure 6.8(a)) is determined by whichever of T_1 and T_2 has the smaller heap key at its root. Alternatively, we could remove e trivially if it is a leaf or has only one child, or otherwise, just as in a standard BST, we swap e with $\mathit{pred}(e)$ or $\mathit{succ}(e)$ and then remove e from the tree. We then repair violations of the heap property (since we have replaced e with an element having a different heap key) by calling *sift-up* or *sift-down*, which are again implemented using rotations so they preserve the BST property.

A BST can be used to encode a dictionary or a sequence. Accordingly, we can also encode a sequence in the "BST" part of a treap. In this context, we often use the elements of the sequence themselves as the heap keys (rather than a separate set of keys), leading to a data structure known as a *Cartesian tree* that we will study in Chapter 8. In the literature, one sometimes finds the term "Cartesian tree" used as a synonym of "treap". In this book, however, we treat the Cartesian tree as a relative of treap where the "BST" part of the structure is used to encode a sequence, rather than a dictionary. Although they are closely related to treaps, Cartesian trees look somewhat different since their elements each have only a single key, and since they are typically not balanced. In addition to Cartesian trees, Chapter 8 introduces yet another close relative of the treap, known as a *priority search tree*, which is used for encoding a set of points in the 2D plane, where x coordinate plays the role of a BST key and y coordinate serves as a heap key.

Problem 99 (Searching for Nearby Elements). In some variants of BSTs, it is faster to walk to an element e from a previously-located element e' close to e in rank space, than to search for e starting from scratch at the root. Suppose we walk from e' to e in a treap (or an RBST, being equivalent in structure). Please show that the path we follow has expected length $O(\log \Delta)$, where $\Delta = |rank(e) - rank(e')| + 1$. As a hint, use linearity of expectation, in much the same way we analyzed the expected running time of randomized quicksort by counting expected comparisons between elements. [Solution]

Problem 100 (Writes Versus Reads). Please argue that the process of building a treap on n elements involves only $\Theta(n)$ expected individual memory writes (versus $\Theta(n \log n)$ expected memory reads). This can be advantageous since writes are often less efficient than reads due to caching overhead. Note that this is equivalent to showing $\Theta(n)$ expected rotations, so you may want to proceed by linearity of expectation, letting X_{ij} be an indicator random variable taking the value 1 if the element of rank i is rotated upward to unseat the element of rank j. If you can show that $\mathbf{E}[X_{ij}] = \Theta(1/r^2)$, with $r = |i-j|+1$, then $\mathbf{E}[\sum X_{ij}]$ will be $\Theta(n)$, as desired. Why does the result of this problem not hold if we use n calls to *insert* in an RBST instead of a treap? [Solution]

6.2.6 Amortized Rebalancing and Scapegoat Trees

In addition to $size(e)$, let us attach an "imbalance counter", $count(e)$, to each element e, giving an upper bound on $|size(left(e)) - size(right(e))|$. We increment $count(e)$ any time an insertion or deletion occurs in e's subtree, since any such operation can at worst increase this imbalance count by one. Specifically, when we *insert* or *delete* an element, we walk from the root down to the point of insertion/deletion, doing the following for each element e along the way:

1. Increment (for *insert*) or decrement (for *delete*) the value of $size(e)$.

2. Increment $count(e)$.

3. If $count(e) > \lceil \frac{1}{3} size(e) \rceil$, we stop the insertion/deletion process and rebuild e's subtree from scratch (taking into account the inserted or deleted element) in $\Theta(size(e))$ time so it is "perfectly" balanced, say, as discussed in problem 98. During the process, we reset $count(e')$ to $|size(left(e')) - size(right(e'))|$ for every element e' in e's subtree. Note that this ensures that $count(e') \leq \lceil \frac{1}{3} size(e') \rceil$ for every such e', so these elements will not trigger this rebalancing step again until they take part in subsequent insertions or deletions.

This approach is a prototypical example of the "lazy" amortized design philosophy, where we leave parts of a data structure relatively unattended until they become sufficiently modified so as to warrant wholesale rebuilding. By rebuilding a subtree when its imbalance count exceeds $\frac{1}{3}$ of its size[6], our tree stays $\frac{2}{3}$-balanced, so its

[6]We use $\frac{1}{3}$ since it makes the math work out nicely, but any constant in $(0, 1)$ would also work.

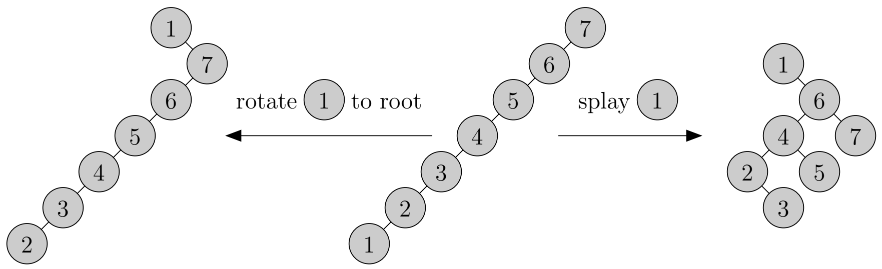


FIGURE 6.11: The impact of using single rotations to pull the lowest element in a degenerate path-shaped tree to the root (on left), and splaying the lowest element (on right). As a side effect, splaying roughly halves the depth of the tree.

height is $O(\log n)$ following our earlier analysis of α-balanced trees. Large batch rebuilds may cause *insert* and *delete* to run slowly in the worst case, but they happen with sufficiently low frequency that we can prove $O(\log n)$ amortized time bounds for both operations. [Details]

The amortized rebalancing idea above leads to another related mechanism, sometimes called either a *scapegoat tree* or a *general balanced tree*, which maintains $O(\log n)$ *worst-case* height while remarkably storing no augmented information at all! The main idea here is that if we notice an insertion occurring at too large a depth (at depth exceeding $\lfloor \log_{3/2} n \rfloor$), then we know one of its ancestor nodes is to blame (i.e., is a "scapegoat") for being too unbalanced, so we walk up the tree, rebalancing each subtree as we go, until the issue has been corrected, with the depth of our tree no longer exceeding $\lfloor \log_{3/2} n \rfloor$. With a similar analysis as above, we can show that *insert* and *delete* both still run in $O(\log n)$ amortized time. [Details]

Problem 101 (Rank-Sensitive Priority Queues). In the comparison model, any data structure (e.g., a priority queue) that always knows its minimum element must require $\Omega(\log n)$ worst-case time for either *insert* or *delete*. However, this bound only really applies when we are inserting or deleting elements of low rank (close to the minimum). Here, we build a so-called *rank-sensitive* priority queue, supporting *find-min* in $O(1)$ time and *insert* and *delete* in $O(\log(n/r))$ time (the best possible in the comparison model), where r is the rank of the element being inserted or deleted. Such data structures are ideal if we only occasionally need the functionality of a priority queue; otherwise, an average element takes only $O(1)$ expected time to insert or delete. We show two elegant ways to obtain rank-sensitive priority queues by modifying different types of balanced BSTs. Note that unlike BSTs, which can find any element quickly, these structures only support an efficient *find-min* operation just like any other standard priority queue.

(a) Suppose we relax the implementation of a BST with amortized rebalancing described above, so that every right subtree remains "unbuilt", much like with a radix heap (Section 5.5.3). Please show how such a structure can be used to implement *insert* and *delete* in $O(\log(n/r))$ amortized time. [Solution]

(b) Another way to implement a rank-sensitive priority queue is to modify a treap. Typically, we store the "actual" elements in the BST keys of a treap, setting the heap keys randomly to ensure balance. Suppose instead that we store the actual elements in our structure in the heap keys, and choose the BST keys at random. Please show how to implement such a structure so that *insert* and *delete* run in $O(\log(n/r))$ expected time. As a hint, try not to store the BST keys explicitly. [Solution]

6.2.7 Splay Trees

If we rotate an element to the root every time it is accessed, then frequently-accessed elements should end up near the root, making them faster to access over time. Unfortunately, this heuristic alone does not seem to give any provable guarantees with respect to balance, but a slight generalization using "double rotations" remarkably does. When rotating element e to the root, we look two steps ahead, at e's parent and grandparent:

- If the parent and grandparent are *out-of-line* (e.g., e is a left child but *parent*(e) is a right child), then we rotate e upward twice, just as before.
- If e is *in line* with its parent and grandparent (e.g., e and *parent*(e) are both left children), we perform the next two rotations in reverse order, first rotating along the parent-grandparent edge, and then along the edge from e to its parent.

We continue rotating e up the tree in two-step increments in this fashion, until either e reaches the root, or e lands one step away from the root, in which case we perform a single rotation to place e at the root. This process is known as *splaying* e, and as we see in Figure 6.11, it can be much more effective at "flattening out" a tree as a side effect than single rotations alone. A *splay tree* is a BST in which an element is splayed every time it is accessed (say, with *find* or *select*).

An element is inserted into a splay tree just as in a standard BST, then splayed. To delete an element, we splay it and then replace it with the *join* of its two subtrees. Splay trees are particularly adept at implementing *split* and *join*. To execute *split*(e), we splay e and detach its right subtree. To *join* trees T_1 and T_2, we can either splay the maximum element of T_1 and attach T_2 as its right child, or splay the minimum element of T_2 and attach T_1 as its left child.

Like skew heaps (Section 5.4.2), splay trees are known as *self-adjusting* data structures, since all they do is blindly apply simple local restructuring rules, without relying on any augmented information such as subtree sizes. Despite this apparent lack of sophistication, one can show that any sequence of k operations in a splay tree (starting from an empty tree) takes $O(k \log n)$ time, so even though they do not necessarily adhere to any strict notion of balance, splay trees support all standard BST operations in $O(\log n)$ amortized time. [Detailed analysis]

There is nothing particularly special about the specific restructuring rule above we use for splaying, since many similar multi-rotation local restructuring rules have now been shown in the literature to give similar performance bounds. The rule above tends to be the most popular, however, since it comes from the original paper by Sleator and Tarjan that introduced splay trees.

Properties of Splay Trees. Splay trees are known to satisfy many impressive properties, and they are conjectured to satisfy many others. Consider an *access* *sequence* S in which we access m elements $e_1, e_2, \ldots, e_m$ one after the other in an n-element splay tree.

- **The Dynamic Finger Property.** Just as with RBSTs and treaps (problem 99), splay trees support efficient searching for an element e close to a previously-accessed element e'. If $\Delta_j = |rank(e_j) - rank(e_{j-1})| + 1$, then a splay tree can execute S in $O(m + n + \sum_j \log \Delta_j)$ time, thereby effectively accessing e_j in only $O(\log \Delta_j)$ amortized time. Note also that this implies that it takes only $\Theta(n)$ time to access all n elements in a splay tree in order, matching the performance of a traditional inorder traversal (and also note the distinction between an inorder traversal and the process of accessing — and therefore splaying — all the elements of a tree in order).
- **The Working Set Property.** Let D_j denote the number of distinct elements accessed prior to e_j starting from the last time we accessed this same element. A splay tree can execute S in $O(m + n \log n + \sum_j \log D_j)$ time, thereby effectively accessing each element e_j in amortized time $O(\log D_j)$. If we only access elements in a small "working set" whose size k is much smaller than n, all accesses therefore only take $O(\log k)$ amortized time.
- **The Static Optimality Property.** Even though a splay tree doesn't know S in advance, the time it takes to execute S is at worst a constant factor times that of a *static* BST (whose shape cannot change over time) designed to execute S in a minimal amount of time. Chapter 11 shows how to compute such an optimal static BST in $O(n^2)$ time using dynamic programming.

Most of these are easy to prove by modifying the basic amortized analysis of splay trees, with the notable exception of the dynamic finger property, proved in a pair of journal papers totaling 85 pages! [Proofs of the remaining properties]

A prominent open question in the field of data structures today is whether or not splay trees satisfy the *dynamic optimality conjecture*, which claims that splay trees take only a constant factor more time than any tree that can dynamically reconfigure itself with rotations, even one that knows the access sequence in advance (further elaboration appears in the endnotes). If true, this would be quite remarkable!

Problem 102 (A Hybrid FIFO Queue and Priority Queue). In Chapter 4, we discuss several implementations of a *min-queue*, supporting *enqueue* and *dequeue* in $O(1)$ amortized time and the ability to find, but not extract, the minimum in $O(1)$ time. Here, we add the ability to *delete* arbitrary elements (including the minimum) in $O(\log n)$ amortized time. It is conceivable that a splay tree can achieve these bounds if we simply store the contents of our FIFO queue as a sequence within the splay tree, and augment each element with a pointer to the minimum element in its subtree — see the section of the endnotes on the *deque conjecture* for splay trees for more information. However, in this problem, we use an approach that is simpler to analyze by using a pair of splay trees. In addition to the constant amortized bounds for *enqueue* and *find-min*, please show how to achieve deletion of an arbitrary element e in $O(\log m)$ amortized time, where m is the number of elements from e onward in the queue. For example, *dequeue* should run in $O(1)$ amortized time, since $m = 1$, and if we *enqueue* an element and then subsequently remove it, then $m = n$. Such a data structure has been called a *queap* in the literature, being a hybrid between a queue and a heap. As a hint, use the dynamic finger property. [Solution]

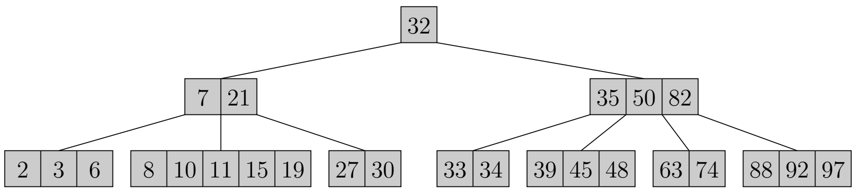


FIGURE 6.12: An example of a B-tree with $B = 3$.

6.3 B-Trees

We now leave the world of *binary* trees behind as we introduce the *B-tree*, where nodes contain multiple elements and may have multiple children. A non-leaf node with k elements has $k + 1$ children, as shown in Figure 6.12, from which one can easily see how the BST property generalizes in this situation. Just like a BST, the B-tree also encodes a sequence from left to right that we can recover by a traversal similar to an Euler tour traversal. When used as a dictionary, the sequence is sorted. B-trees are "complete" in the sense that all leaves have the same depth.

Every B-tree is parameterized by a number $B > 1$ describing the number of elements a node may contain. Every node except the root must have between $B-1$ and $2B-1$ elements, inclusive (and therefore $B \ldots 2B$ children). The root is special, having between 1 and $2B - 1$ elements (i.e., there is no lower bound). The B-tree with $B = 2$ is called a *2-3-4 tree*, since every node may have 2, 3, or 4 children. As we mentioned earlier in the chapter, there is a direct correspondence between 2-3-4 trees and red-black trees, so we can generally avoid discussion of the somewhat involved mechanism for balancing a red-black tree by instead looking at the same process from the simpler viewpoint of balancing a 2-3-4 tree. [Further details]

In practice, B is usually quite large (say, in the thousands), making the height of the tree (at most $\lceil \log_B n \rceil$) quite small. B-trees are often used for large datasets stored on slow block-transfer media like disk, where accessing data is very slow, but we can transfer an entire "page" of data into main memory all at once. By setting B just large enough that a page from disk contains an entire node, this allows us to perform only $O(\log_B n)$ disk accesses for each B-tree operation (versus $O(\log_2 n)$ for a BST). For example, if $B = 2^{10} = 1024$, a B-tree needs to access the disk roughly 10 times less often than a BST. If we keep the top two or three levels of the B-tree in main memory, we can usually reach any element with only one or two additional disk accesses.

Locating an element in a B-tree by its key or its rank (if we have augmented nodes with subtree sizes) is straightforward. Just as with a BST, we walk down the tree guided by the BST property or subtree sizes. Along the way, we examine $O(\log_B n)$ nodes, each containing $O(B)$ elements, for a total running time of $O(B \log_B n)$. However, remember that in the common use of the B-tree in an external memory setting, the "$O(\log_B n)$" part usually matters much more than the "$O(B)$" part.

Insertion and Deletion in a *B*-Tree. B-trees support a notion of rotation just like BSTs, as shown in Figure 6.13(a). By rotating elements through a parent node,

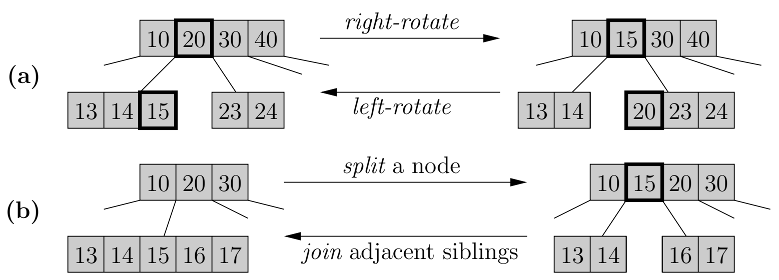


FIGURE 6.13: Operations on a B-tree: (a) rotating left and right through a parent node to trade elements among adjacent siblings, and (b) splitting a node by donating its median element to its parent, and the inverse operation of joining two adjacent siblings by stealing an intermediate element from the parent.

a node can transfer elements to or from its adjacent siblings. Other fundamental B-tree operations include splitting a node into two adjacent siblings, and the reverse operation of joining two adjacent siblings together, shown in Figure 6.13(b). We only split nodes that are at least full (with at least $2B - 1$ of elements), since otherwise we risk creating nodes that are "underfull", containing fewer than $B - 1$ elements. Similarly, we only join adjacent siblings if both contain at most the minimum possible number $B - 1$ of elements, since otherwise we risk creating a node that is too full. Rotations and the *split* and *join* operations all require $O(B)$ time if the elements in a node are stored in an array, as is typical.

In order to *insert* into a B-tree, we first search to find a leaf node into which our new element should be placed. If this leaf node contains fewer than $2B-1$ elements we can simply insert the new element. If not, we have two alternatives: if one of our siblings is not full, we free up space for the new element by donating one of our elements to the sibling with a rotation. Otherwise, we can insert the new element and then split the current node. Since splitting a node donates an element to its parent, this may in turn cause the parent to donate a key to a sibling via rotation or be split. If this chain of splits propagates far enough, we may end up splitting the root, and this is the only way our B-tree increases in height.

Deletion is completely symmetric, possibly resulting in a propagating chain of *join* operations that, if it reaches the root, can decrease the overall height of the tree. We generally only delete from leaf nodes. To delete an element e from a non-leaf node, we can first swap e with its predecessor or successor, much like deletion from a standard BST. Both of these elements will belong to leaf nodes if e does not.

Both *insert* and *delete* run in $O(B \log_B n)$ time, since they access $O(\log_B n)$ nodes each of size $O(B)$. This asymptotic running time is unaffected even if we need to maintain augmented information in the tree (e.g., subtree sizes), as long as the augmented information is locally-recomputable.

One can improve the running time of every standard B-tree operation in theory by storing the $O(B)$ elements in each node not as an array, but instead as a tiny balanced BST. This reduces the time required to interact with each node (e.g., searching a node, or splitting / joining nodes) from $O(B)$ to $O(\log B)$, improving the overall running time for each B-tree operation to $O(\log B \log_B n) = O(\log n)$. This would probably not sensible in practice, though, since B is usually not large enough to make $O(\log B)$ with a larger hidden constant a big win over $O(B)$ with a small hidden constant.

Problem 103 (Constant Amortized Memory Writes). Assuming $B = O(1)$ for simplicity, insertion and deletion in a B-tree require $O(\log n)$ individual memory reads but only $O(1)$ amortized memory writes[7], saving time in most computing environments where writes are more costly than reads. Please prove this amortized constant bound. As a hint, you should determine the amount of "credit" associated with a node in 4 states: underfull by one element, minimally full, maximally full, and overfull by one element (the credit in all other cases can be set to zero). [Solution]

Leaf-Oriented Trees. Sometimes we design BSTs or B-trees with elements stored only in leaves, known as *leaf-oriented* trees. If used as a dictionary, we would typically augment interior (non-leaf) nodes so they store the minimum and maximum keys present in their subtrees; this information is sufficient to allow *find* and all other standard tree operations to work as efficiently as a non-leaf-oriented structure. A leaf-oriented B-tree whose leaves are all connected in a long doubly-linked list (to facilitate enumeration and range queries) is known as a B+-tree, and it is perhaps the most popular of several variants / extensions of B-trees found in practice.

6.4 Skip Lists

The downside of implementing a dictionary using a sorted doubly-linked list is the excessive time it takes to scan to a particular location in the list (say, to locate an element based on its key or rank). We can alleviate this problem, however, by introducing an "express" list — just like an express train or bus, this list makes fewer stops, getting you close to your destination much faster. The *skip list* is a data structure built on this idea, and it can be made to support all fundamental operations on dynamic sequences and dictionaries in $O(\log n)$ time with high probability, just like with an RBST or treap.

Figure 6.14 illustrates the structure of a skip list, a collection of doubly-linked lists stacked on top of each-other, where the lowest (level zero) list contains every element (in sorted order, for a dictionary), and each successive level contains a subset of roughly half the elements on the level beneath it. We maintain separate records in memory for an element on each of the levels in which it is present, connected together by "vertical" pointers in a doubly-linked list. Each list is preceded by a

[7]One must be slightly careful with the definition of a "B-tree" for this result to apply. The original definition in the literature allowed each node to have only between B and $2B-1$ children, which in fact may initially seem more natural since splitting an overfull node now results in exactly two minimally-filled nodes, and joining a minimally-filled node with an underfull node now results in a node exactly at its upper capacity. However, this variant behaves poorly from an amortized perspective since an *insert* can set off a cascade of splits going all the way to the root, after which a *delete* of the same element would effectively undo all of these with joins, putting the tree back in its original state. An alternating series of such operations therefore requires $\Theta(\log n)$ memory writes per operation, whereas if we just add a small amount of extra slack, allowing nodes to have up to $2B$ children, then we get instead a constant amortized bound. If we relax the upper bound further, allowing maximum node sizes above $2B$, this only helps from an amortized perspective.

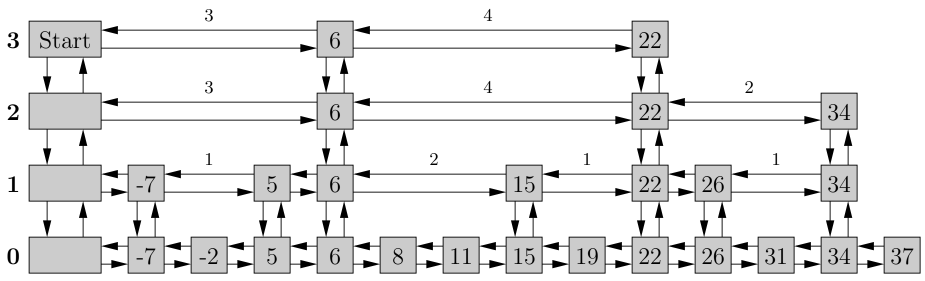


FIGURE 6.14: A skip list. Each horizontal link is augmented with its "skip count" (only nonzero skip counts are shown above).

special "start" element that we ensure is present in every level.

To *find* an element based on its key or rank, we start at the beginning of the top-level list and scan to the right, moving downward whenever we realize that the next link would skip too far ahead. Rank-based access requires augmenting each horizontal pointer with its "skip count" (much like augmenting a BST with subtree sizes), so we can keep track of our accumulated rank as we move through the structure. All other extended BST operations, such as *find-min* / *find-max*, *pred* / *succ*, *split* / *join*, range queries, and inorder traversals, can be translated in a straightforward manner to the skip list.

The *insert* operation employs randomness in a clever way: to insert an element, we *find* the appropriate location for it in the level-0 list and insert a copy there. We then repeatedly flip a fair coin and keep inserting copies into successively higher levels as long as we flip heads. That is, whenever we insert an element at level k, we flip a fair coin and with probability 1/2, we also insert it at level $k + 1$. This ensures that roughly half the elements appear in the level 1 list, roughly one quarter appear in the level 2 list, and so on. One can show that the total number of levels is $O(\log n)$, and that the total space occupied by the structure is $\Theta(n)$ (both bounds holding with high probability). To *delete* an element, we first find it and then remove all copies from every level on which it exists. With a simple analysis based on randomized reduction, we can show that all standard BST operations run on a skip list in $O(\log n)$ time with high probability. [Analysis of skip lists]

Skip Lists Versus BSTs. Skip lists are elegant from a theoretical perspective and efficient and easy to implement in practice. They also provide a nice alternative to the tree-based data structures discussed earlier in the chapter. In fact, they were originally introduced as a simpler alternative to the balanced BST. Interestingly, with a bit of care, one can turn the skip list "on its side" and view it as yet another type of BST with a randomized balancing mechanism involving upward rotation based on random coin flips whose complexity is roughly on par with that of RBSTs and treaps. The resulting BST has edge weights that all sum to the same value along every root-to-leaf path (owing to the fact that all elements in the bottom row of a skip list live at the same depth relative to the starting point), so this structure could also be characterized as a randomly-balanced type of path-balanced tree. [Interpreting a skip list as a BST].

One final advantage of skip lists is that by inserting n elements into a skip list and then reading them out in sorted order, we can obtain a comparison-based sorting algorithm that makes $O(n \log n)$ memory reads but only $\Theta(n)$ memory writes (both bounds holding with high probability). Among the other data structures described in this chapter, only the treap, B-tree, and red-black tree (equivalent to a B-tree for $B = 2$) offer this feature (see problems 100 and 103). The fact that we make only a small number of memory writes per data structure update is particularly useful with the persistence technique described in Section 4.7, since the size of a persistent data structure is directly determined by the number of memory writes we make over the lifetime of the structure. Skip lists also support the ability (like treaps, RBSTs, and splay trees) to move from a previous element e' to a new element e in (expected) time proportional to the logarithm of the rank difference between the two.

In Section 7.5.3 in the next chapter, we will see how to use ideas inspired by skip lists to route information efficiently through a *distributed hash table*, much like in the following problem.

Problem 104 (Level Ancestors). The *level ancestor* problem involves preprocessing a static n-node rooted tree (not necessarily a binary tree, and not necessarily balanced in any way) so that we can quickly answer queries of the form "what is element e's ancestor at depth d in the tree?". In other words, given any element e we need to be able to quickly jump an arbitrary "distance" upward along the path from e to the root. Level ancestors are easy to compute on a path (i.e., a non-branching tree), since we can simply store the path in an array and jump to the appropriate ancestor index in $O(1)$ time. In this problem, we investigate fast data structures for this problem on arbitrary trees; many of these structures are motivated in part by the same general ideas as skip lists.

(a) Suppose we augment every node in the tree with $\log n$ pointers that send us upward 1, 2, 4, 8, etc. nodes in the tree. Show how to build such a data structure using $O(n \log n)$ time and space, and show how it can answer level ancestor queries in $O(\log n)$ time. Next, please show how to use skip lists to improve this result so that we spend only $\Theta(n)$ expected preprocessing time and space, such that queries can be answered in $O(\log n)$ time with high probability. [Solution]

(b) Suppose we decompose a tree into node-disjoint paths as follows: we first locate the longest root-to-leaf path and remove it from the tree. This potentially splits the tree into several disjoint trees, and on each of them we proceed to again locate and remove the longest root-to-leaf path, until eventually every node belongs to some path in our decomposition. Show how to build such a path decomposition in $\Theta(n)$ time/space. Now, we employ a neat trick: take each path of length L and extend it upward to a length of $2L$ by including the L most immediate ancestors above it in the tree (it is possible that fewer than L ancestors exist, if we are near the top of the tree). Our paths may now cease to be disjoint, but this is fine since they still occupy only linear space (the combined length of all extended paths is at most $2n$). Please show how we achieve $\Theta(n)$ preprocessing time/space and $O(\log n)$ query time with this structure. As a hint: show that you can move upward through the tree using steps of exponentially increasing size. [Solution]

(c) Finally, see if you can combine (a) and (b) to obtain $O(n \log n)$ preprocessing time and space, with only $O(1)$ query time (as mentioned in the endnotes, the preprocessing bounds can even be improved to $\Theta(n)$!). [Solution]

Problem 105 (Adaptive Sorting Algorithms). When we studied sorting algorithms in Chapter 3, we learned that insertion sort has a running time of $\Theta(n + I)$, where

I denotes the number of inversions present in our input sequence. If I is small (say, on the order of n), then insertion sort can run much faster than our favorite $\Theta(n \log n)$ sorting algorithms such as merge sort and quicksort. In fact, there must exist some "cutoff" value of I, below which it is preferable to use insertion sort, and above which it is preferable to use merge sort or quicksort. This discontinuity is somewhat unsightly — it would be nice to have only one sorting algorithm that does well for all values of I. Such a sorting algorithm is said to be *adaptive* with respect to I. Many researchers have studied adaptive sorting algorithms, some of which scale gracefully in terms of I and others that involve different measures of disorder in a sequence; see the endnotes for further references. Suppose we implement insertion sort by maintaining the sorted prefix of our array in a skip list. Show that this gives a running time of $O(n + n\log(1 + I/n))$ with high probability, which is always at least as fast as both $O(n + I)$ and $O(n \log n)$ for all values of I. [Solution]

Problem 106 (A "Skip List" Approach to Persistence). In this problem, we describe the high-level details of a randomized analog of the amortized technique introduced in Section 4.7 for making a pointer-based data structure persistent. Our approach works in exactly the same setting as the approach from Section 4.7: we start with an ephemeral (non-persistent) structure in which every element occupies $O(1)$ memory and has at most $k = O(1)$ incoming pointers from other elements (e.g., $k = 3$ for a BST), where every element is accessed by following a sequence of pointers starting from a designated root node. Every node x should be augmented with a time-sorted "modification list", where each entry contains a time stamp t as well as the complete contents of node x as of time t. Any pointer to node x in this new persistent structure actually points at an entry in a x's modification list, rather than at x directly. As opposed to the method in Section 4.7 in which modification lists were limited to $k + 1$ entries before they were split[8], here we store each modification list as an array or linked list that can grow without bound (the root list should be an array, to facilitate binary searching).

To modify node x as of time t, we add a new entry to the end of x's modification list (in only $O(1)$ time) with time stamp t. We then flip a biased coin: with probability $1 - p$, we do nothing more, and with probability p, we modify all of the nodes currently pointing at x so they now point to the new entry in x's modification list. Modification of these "parent" nodes is done just as when we modified x, by adding a new record to their modification lists, and flipping coins to see if the modification should be propagated backward yet further, much like in a skip list. Any time a modification propagates backward to the root, it stops at that point. Alternatively, we can allow propagation of a modification to continue backward from the root. In this case, we introduce a new root node pointing at the original one, and again keep flipping coins to see if the modification should propagate backward even more. This effectively makes the modification of the root have the character of a skip list, instead of keeping it as one large time-sorted list that we binary search. You may want to focus on the array version for this problem since its analysis is slightly simpler, owing to the fact that we can avoid the extra analysis involved with the root.

Please show how to choose an appropriate value of p that (a) adds only $O(1)$ expected time to the operation of modifying any node, and that (b) results in $O(Q + \log T)$ expected query time, where Q is an upper bound on the query time of the original ephemeral structure, and T is the number of original update operations we have invoked (equivalently, the number of points in time at which we have historical versions to track). Observe that these bounds match the amortized bounds from Section 4.7. You may need to fill in yet-unspecified details of the structure as part of your solution. [Solution]

[8]Now that we know B-trees, it is worth noting the strong resemblance between our partial persistence amortized splitting mechanism and the cascading split mechanism in a B-tree, which also exhibits good amortized behavior.