For the *DeleteMin* operation, let R be the rank of the tree that contains the minimum element, and let T be the number of trees before the operation. To perform a *DeleteMin*, we once again split the children of a tree, creating an additional R new trees. Notice that, although this can remove marked nodes (by making them unmarked roots), this cannot create any additional marked nodes. These R new trees, along with the other T trees, must now be merged, at a cost of $T + R + \log N = T + O(\log N)$, by Lemma 11.3. Since there can be at most $O(\log N)$ trees, and the number of marked nodes cannot increase, the potential change is at most $O(\log N) - T$. Adding the actual time and potential change gives the $O(\log N)$ amortized bound for *DeleteMin*.

Finally, for the *DecreaseKey* operation, let C be the number of cascading cuts. The actual cost of a *DecreaseKey* is $C + 1$, which is the total number of cuts performed. The first (noncascading) cut creates a new tree and thus increases the potential by 1. Each cascading cut creates a new tree, but converts a marked node to an unmarked (root) node, for a net loss of one unit per cascading cut. The last cut also can convert an unmarked node (in Fig. 11.20 it is node 5) into a marked node, thus increasing the potential by 2. The total change in potential is thus at most $3 - C$. Adding the actual time and the potential change gives a total of 4, which is $O(1)$.

11.5. Splay Trees

As a final example, we analyze the running time of splay trees. Recall, from Chapter 4, that after an access of some item X is performed, a splaying step moves X to the root by a series of three operations: *zig*, *zig-zag*, and *zig-zig*. These tree rotations are shown in Figure 11.21. We adopt the convention that if a tree rotation is being performed at node X, then prior to the rotation P is its parent and G is its grandparent (if X is not a child of the root).

Recall that the time required for any tree operation on node X is proportional to the number of nodes on the path from the root to X. If we count each *zig* operation as one rotation and each *zig-zig* or *zig-zag* as two rotations, then the cost of any access is equal to 1 plus the number of rotations.

In order to show an $O(\log N)$ amortized bound for the splaying step, we need a potential function that can increase by at most $O(\log N)$ over the entire splaying step but that will also cancel out the number of rotations performed during the step. It is not at all easy to find a potential function that satisfies these criteria. A simple first guess at a potential function might be the sum of the depths of all the nodes in the tree. This does not work, because the potential can increase by $\Theta(N)$ during an access. A canonical example of this occurs when elements are inserted in sequential order.

A potential function Φ that does work is defined as

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

where $S(i)$ represents the number of descendants of i (including i itself). The potential function is the sum, over all nodes i in the tree T, of the logarithm of $S(i)$.

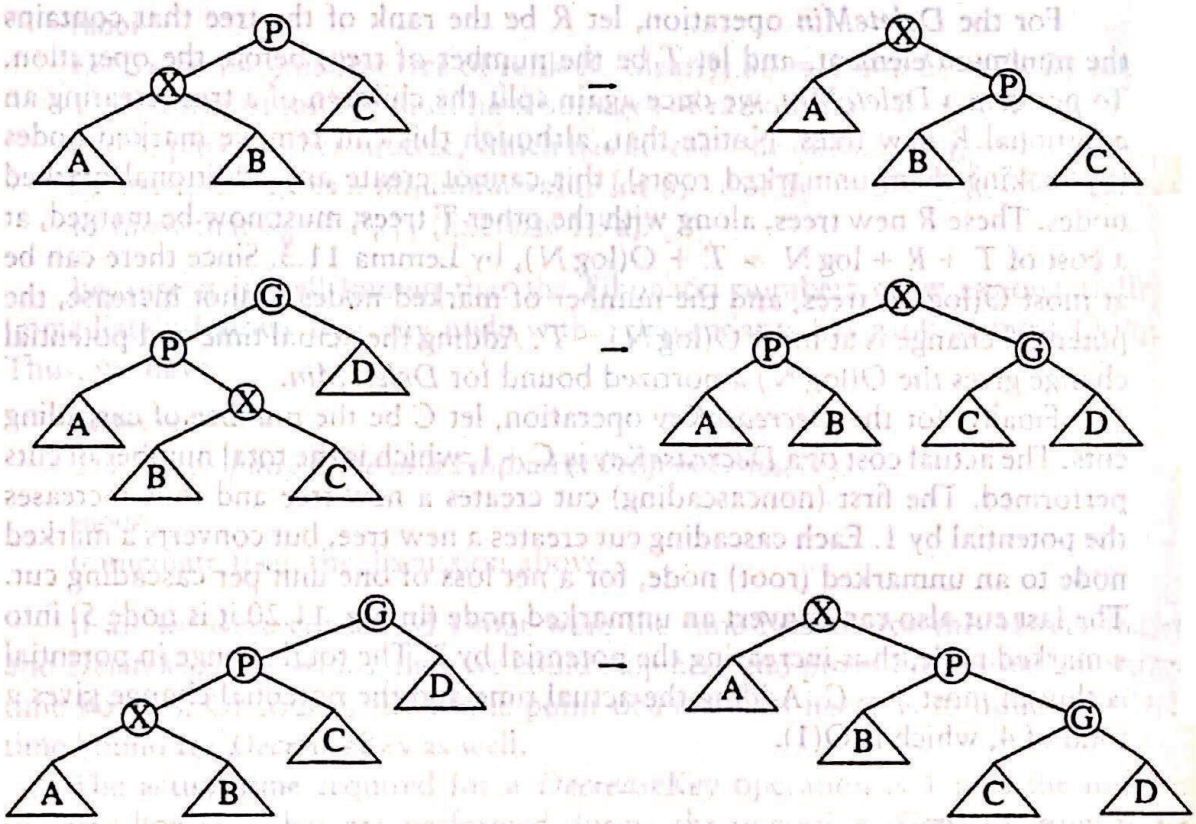


Figure 11.21 *zig, zig-zag,* and *zig-zig* operations; each has a symmetric case (not shown)

To simplify the notation, we will define

$$R(i) = \log S(i)$$

This makes

$$\Phi(T) = \sum_{i \in T} R(i)$$

$R(i)$ represents the *rank* of node i. The terminology is similar to what we used in the analysis of the disjoint set algorithm, binomial queues, and Fibonacci heaps. In all these data structures, the meaning of *rank* is somewhat different, but the rank is generally meant to be on the order (magnitude) of the logarithm of the size of the tree. For a tree T with N nodes, the rank of the root is simply $R(T) = \log N$. Using the sum of ranks as a potential function is similar to using the sum of heights as a potential function. The important difference is that while a rotation can change the heights of many nodes in the tree, only X, P, and G can have their ranks changed.

Before proving the main theorem, we need the following lemma.

LEMMA 11.4.

If $a + b \leq c$, and a and b are both positive integers, then

$$\log a + \log b \leq 2 \log c - 2$$

From Figure 11.21 we see that $S_f(X) = S_i(G)$, so their ranks must be equal. Thus, we obtain

$$AT_{\text{zig-zag}} = 2 + R_f(P) + R_f(G) - R_i(X) - R_i(P)$$

We also see that $S_i(P) \geq S_i(X)$. Consequently, $R_i(X) \leq R_i(P)$. Making this substitution gives

$$AT_{\text{zig-zag}} \leq 2 + R_f(P) + R_f(G) - 2R_i(X)$$

From Figure 11.21 we see that $S_f(P) + S_f(G) \leq S_f(X)$. If we apply Lemma 11.4, we obtain

$$\log S_f(P) + \log S_f(G) \leq 2 \log S_f(X) - 2$$

By the definition of *rank*, this becomes

$$R_f(P) + R_f(G) \leq 2R_f(X) - 2$$

Substituting this, we obtain

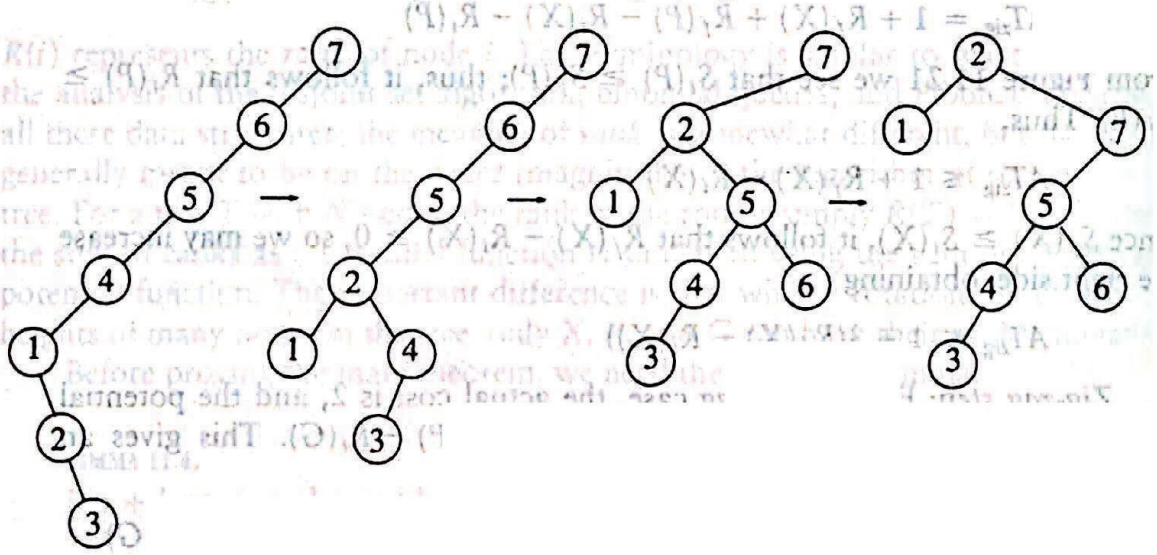
$$\begin{aligned} AT_{\text{zig-zag}} &\leq 2R_f(X) - 2R_i(X) \\ &\leq 2(R_f(X) - R_i(X)) \end{aligned}$$

Since $R_f(X) \geq R_i(X)$, we obtain

$$AT_{\text{zig-zag}} \leq 3(R_f(X) - R_i(X))$$

Zig-zig step: The third case is the *zig-zig*. The proof of this case is very similar to the *zig-zag* case. The important inequalities are $R_f(X) = R_i(G)$, $R_f(X) \geq R_f(P)$, $R_i(X) \leq R_i(P)$, and $S_i(X) + S_f(G) \leq S_f(X)$. We leave the details as Exercise 11.8.

Figure 11.22 The splaying steps involved in splaying at node 2



From Figure 11.21 we see that $S_f(X) = S_i(G)$, so their ranks must be equal Thus, we obtain

$$AT_{\text{zig-zag}} = 2 + R_f(P) + R_f(G) - R_i(X) - R_i(P)$$

We also see that $S_i(P) \geq S_i(X)$. Consequently, $R_i(X) \leq R_i(P)$. Making this substitution gives

$$AT_{\text{zig-zag}} \leq 2 + R_f(P) + R_f(G) - 2R_i(X)$$

From Figure 11.21 we see that $S_f(P) + S_f(G) \leq S_f(X)$. If we apply Lemma 11.4, we obtain

$$\log S_f(P) + \log S_f(G) \leq 2 \log S_f(X) - 2$$

By the definition of *rank*, this becomes

$$R_f(P) + R_f(G) \leq 2R_f(X) - 2$$

Substituting this, we obtain

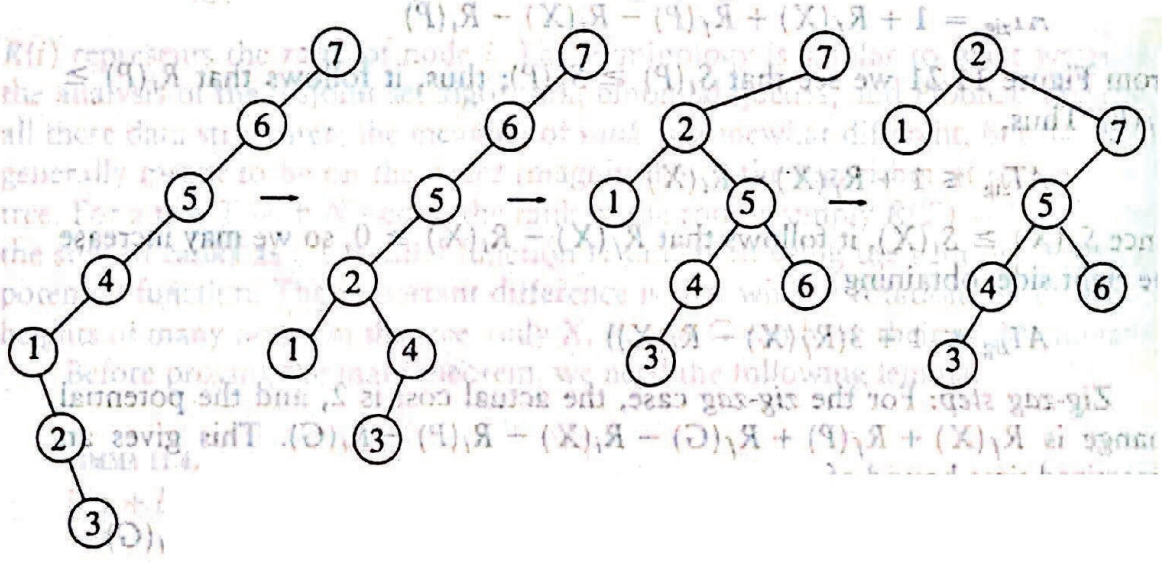
$$\begin{aligned} AT_{\text{zig-zag}} &\leq 2R_f(X) - 2R_i(X) \\ &\leq 2(R_f(X) - R_i(X)) \end{aligned}$$

Since $R_f(X) \geq R_i(X)$, we obtain

$$AT_{\text{zig-zag}} \leq 3(R_f(X) - R_i(X))$$

Zig-zig step: The third case is the *zig-zig*. The proof of this case is very similar to the *zig-zag* case. The important inequalities are $R_f(X) = R_i(G)$, $R_f(X) \geq R_f(P)$, $R_i(X) \leq R_i(P)$, and $S_i(X) + S_f(G) \leq S_f(X)$. We leave the details as Exercise 11.8.

Figure 11.22 The splaying steps involved in splaying at node 2



The amortized cost of an entire splay is the sum of the amortized costs of each splay step. Figure 11.22 shows the steps that are performed in a splay at node 2. Let $R_1(2)$, $R_2(2)$, $R_3(2)$, and $R_4(2)$ be the rank of node 2 in each of the four trees. The cost of the first step, which is a *zig-zag*, is at most $3(R_2(2) - R_1(2))$. The cost of the second step, which is a *zig-zig*, is $3(R_3(2) - R_2(2))$. The last step is a *zig* and has cost no larger than $3(R_4(2) - R_3(2)) + 1$. The total cost thus telescopes to $3(R_4(2) - R_1(2)) + 1$.

In general, by adding up the amortized costs of all the rotations, of which at most one can be a *zig*, we see that the total amortized cost to splay at node X is at most $3(R_f(X) - R_i(X)) + 1$, where $R_i(X)$ is the rank of X before the first splaying step and $R_f(X)$ is the rank of X after the last splaying step. Since the last splaying step leaves X at the root, we obtain an amortized bound of $3(R_f(T) - R_i(X)) + 1$, which is $O(\log N)$.

Because every operation on a splay tree requires a splay, the amortized cost of any operation is within a constant factor of the amortized cost of a splay. Thus, all splay tree operations take $O(\log N)$ amortized time. By using a more general potential function, it is possible to show that splay trees have several remarkable properties. This is discussed in more detail in the exercises.

Summary

In this chapter, we have seen how an amortized analysis can be used to apportion charges among operations. To perform the analysis, we invent a fictitious potential function. The potential function measures the state of the system. A high-potential data structure is volatile, having been built on relatively cheap operations. When the expensive bill comes for an operation, it is paid for by the savings of previous operations. One can view potential as standing for *potential for disaster*, in that very expensive operations can occur only when the data structure has a high potential and has used considerably less time than has been allocated.

Low potential in a data structure means that the cost of each operation has been roughly equal to the amount allocated for it. Negative potential means debt; more time has been spent than has been allocated, so the allocated (or amortized) time is not a meaningful bound.

As expressed by Equation (11.2), the amortized time for an operation is equal to the sum of the actual time and potential change. Taken over an entire sequence of operations, the amortized time for the sequence is equal to the total sequence time plus the net change in potential. As long as this net change is positive, then the amortized bound provides an *upper bound* for the actual time spent and is meaningful.

The keys to choosing a potential function are to guarantee that the minimum potential occurs at the beginning of the algorithm, and to have the potential increase for cheap operations and decrease for expensive operations. It is important that the excess or saved time be measured by an opposite change in potential. Unfortunately, this is sometimes easier said than done.