

# Chapter 6

## Self-Adjusting Data Structures

Chapter 5 describes a data structure that is able to achieve an expected query time that is proportional to the entropy of the query distribution. The most interesting aspect of this data structure is that it does not need to know the query distribution in advance. Rather, the data structure adjusts itself using information gathered from previous queries.

In this chapter we study several other data structures that adjust themselves in the same way. All of these data structures are efficient only in an amortized sense. That is, individual operations may take a large amount of time, but over a sequence of many operations, the amortized (average) time per operation is still small.

The advantage of self-organizing data structures is that they can often perform “natural” sequences of accesses much faster than their worst-case counterparts. Examples of such natural sequences including accessing each of the elements in order, using the data structure like a stack or queue and performing many consecutive accesses to the same element.

In other sections of this book we have analyzed the amortized cost of operations on data structures using the accounting method, in which we manipulate credits to bound the amortized cost of operations. A different method of amortized analysis is the *potential method*. For any data structure  $D$ , we define a potential  $\Phi(D)$ . If we let  $D_i$  denote our data structure after the  $i$ th operation and let  $C_i$  denote the cost (running time) of the  $i$ th operation then the *amortized cost* of the  $i$ th operation is defined as

$$A_i = C_i + \Phi(D_i) - \Phi(D_{i-1}) .$$

Note that this is simply a definition and, since  $\Phi$  is arbitrary,  $A_i$  has nothing to do with the real cost  $C_i$ . However, the cost of a sequence of  $m$  operations is

$$\begin{aligned} \sum_{i=1}^m C_i &= \sum_{i=1}^m (C_i + \Phi(D_i) - \Phi(D_{i-1})) - \sum_{i=1}^m (\Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^m A_i - \sum_{i=1}^m (\Phi(D_i) - \Phi(D_{i-1})) \end{aligned}$$

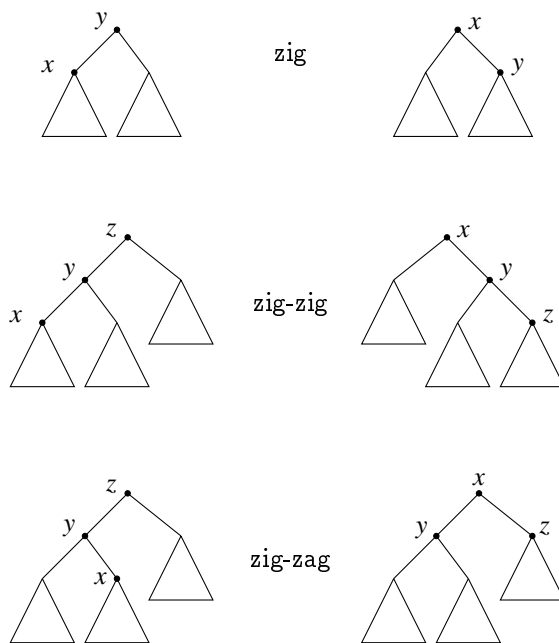


Figure 6.1: Splay tree.

$$= \sum_{i=1}^m A_i + \Phi(D_0) - \Phi(D_m) .$$

The quantity  $\Phi(D_0) - \Phi(D_m)$  is referred to as the *net drop in potential*. Thus, if we can provide a bound  $D$  on  $A_i$ , then the total cost of  $m$  operations is  $mD$  plus the net drop in potential.

The potential method is equivalent to the accounting method. Anything that can be proven with the potential method can be proven with the accounting method. (Just think of  $\Phi(D)$  as the number of credits the data structure stores, with a negative value indicating that the data structure owes credits) However, once a potential function is defined, mathematical manipulations that can lead to very compact proofs are convenient. Of course, the rub is in finding the right potential function.

## 6.1 Splay Trees

One of the first-ever self-modifying data structures is the *splay tree*. We begin our discussion of splay trees with the definition and analysis of the splay operation. A splay tree is a binary search tree that is augmented with the splay operation. Given a binary search tree  $T$  rooted at  $r$  and a node  $x$  in  $T$ , splaying node  $x$  is a sequence of left and right rotations performed on the tree until  $x$  is the root of  $T$ . This sequence of rotations is governed by the following rules: (Note: a node is a left (respectively, right) child if it is the left (respectively, right) child of its parent) In what follows,  $y$  is the parent of  $x$  and  $z$  is the parent of  $y$  (should it exist).

1. **Zig step:** If  $y$  is the root of the tree, then rotate  $x$  so that it becomes the root.
2. **Zig Zig step:** The node  $y$  is not the root. Both  $x$  and  $y$  are left (or right) children, then first rotate  $y$ , so that  $z$  and  $x$  are children of  $y$ , then rotate  $x$  so that it becomes the parent of  $y$ .
3. **Zig Zag step:** Node  $y$  is not the root. If  $x$  is a right child and  $y$  is a left child or vice-versa, rotate  $x$  twice. After one rotation it is the parent of  $y$  and after the second rotation, both  $y$  and  $z$  are children of  $x$ .

We are now ready to define the potential of a splay tree. Let  $T$  be a binary search tree on  $n$  nodes and let the nodes be numbered from 1 to  $n$ . Assume that each node  $i$  has a fixed positive weight  $w(i)$ . Define the size  $s(x)$  of a node  $x$  to be the sum of the weights of all the nodes in the subtree rooted at  $x$ , including the weight of  $x$ . The rank  $r(x)$  of a node  $x$  is  $\log(s(x))$ . By definition, the rank of a node is at least as large as the rank of any of its descendants. The potential of a given tree  $T$  is  $\Phi(T) = \sum_{i=1}^n \log(s(i))$ .

All of the properties of a splay tree are derived from the following lemma:

**Lemma 3.** *The amortized time to splay a binary search tree  $T$  with root  $t$  at a node  $x$  is at most  $3(r(t) - r(x)) + 1 = O(\log(s(t))/\log(s(x)))$*

*Proof.* If node  $x$  is the root, then the lemma holds trivially. If node  $x$  is not the root, then a number of rotations are performed according to the rules of splaying. We first show that the lemma holds after each step (zig, zigzig or zigzag) is performed. Let  $r_0(i)$ ,  $r_1(i)$  and  $s_0(i)$ ,  $s_1(i)$  represent the rank and size of node  $i$  before and after the first step, respectively. Recall that the amortized cost of an operation  $\alpha = t + \Phi_1(T) - \Phi_0(T)$  where  $t$  is the actual cost of the operation, and  $\Phi_0$  and  $\Phi_1$  are the potentials before and after the step.

1. **Zig step:** In this case,  $y$  is the root of the tree and a single rotation is performed to make  $x$  the root of the tree. In the whole splay operation, this step is only ever executed once. The actual cost of the rotation is 1. We show that the amortized cost is at most  $3(r_1(x) - r_0(x)) + 1$ .

$$\text{amortized cost} = 1 + r_1(x) + r_1(y) - r_0(x) - r_0(y) \tag{6.1}$$

$$\leq 1 + r_1(x) - r_0(x) \tag{6.2}$$

$$\leq 1 + 3(r_1(x) - r_0(x)) \tag{6.3}$$

Equation (6.1) follows from the definition of amortized cost and the fact that the only ranks that changed are the ranks at  $x$  and  $y$ . The ranks of all other nodes in the tree remain the same since the nodes in their subtrees remain the same. Therefore, the change in potential is determined by the change in the ranks of  $x$  and  $y$ . Equation (6.2) holds because prior to the rotation, node  $y$  was the root of the tree which implies that  $r_0(y) \geq r_1(y)$ . Equation (6.3) follows since after the rotation, node  $x$  is the root the tree, which means that  $r_1(x) - r_0(x)$  is positive.

2. **Zig Zig step:** The node  $y$  is not the root. Both  $y$  and  $z$  are left (or right) children. In this case, two rotations are made. The first rotation is at  $y$ , so that  $z$  and  $x$  are children of  $y$ . The second rotation is at  $x$  so that it becomes the parent of  $y$  and grandparent of  $z$ . Since two rotations are performed, the actual cost of this step is 2.

$$\text{amortized cost} = 2 + r_1(x) + r_1(y) + r_1(z) - r_0(x) - r_0(y) - r_0(z) \quad (6.4)$$

$$= 2 + r_1(y) + r_1(z) - r_0(x) - r_0(y) \quad (6.5)$$

$$\leq 2 + r_1(x) + r_1(z) - 2r_0(x) \quad (6.6)$$

$$(6.7)$$

After the two rotations, the only ranks that change are the ranks of  $x$ ,  $y$  and  $z$ . Therefore, the change in potential is defined by the change in their ranks. Equation (6.4) follows from the definition of amortized cost. Now, prior to the two rotations,  $x$  is child of  $y$  which is a child of  $z$  and after the two rotations,  $z$  is a child of  $y$  which is a child of  $x$ . This implies that  $r_1(x) = r_0(z)$ . Equation (6.5) follows. Similarly, (6.6) follows since  $r_1(x) \geq r_1(y)$  and  $r_0(x) \leq r_0(y)$ . What remains to be shown is that  $2 + r_1(x) + r_1(z) - 2r_0(x) \leq 3(r_1(x) - r_0(x))$ .

With a little manipulation, we see that this is equivalent to showing that  $r_1(z) + r_0(x) - 2r_1(x) \leq -2$ .

$$r_1(z) + r_0(x) - 2r_1(x) = \log(s_1(z)/s_1(x)) + \log(s_0(x)/s_1(x)) \quad (6.8)$$

$$= \log((s_1(z)/s_1(x)) * (s_0(x)/s_1(x))) \quad (6.9)$$

$$\leq \log(1/4) \quad (6.10)$$

$$= -2 \quad (6.11)$$

Equation (6.8) simply follows from the definition of ranks. Notice that  $s_1(x) \geq s_1(z) + s_0(x)$ . This implies that  $s_1(z)/s_1(x) + s_0(x)/s_1(x) \leq 1$ . If we let  $b = s_1(z)/s_1(x)$  and  $c = s_0(x)/s_1(x)$ , we note that  $b + c \leq 1$  and the expression in (6.9) becomes  $\log(bc)$ . Elementary calculus shows that given  $b + c \leq 1$ , the product  $bc$  is maximized when  $b = c = 1/2$ . This gives (6.10).

- 3. Zig Zag step:** Node  $y$  is not the root. If  $x$  is a right child and  $y$  is a left child or vice-versa, rotate  $x$  twice. After one rotation it is the parent of  $y$  and after the second rotation, both  $y$  and  $z$  are children of  $x$ . Again since two rotations are performed, the actual cost of the operation is 2. Let us look at what the amortized cost is.

$$\text{amortized cost} = 2 + r_1(x) + r_1(y) + r_1(z) - r_0(x) - r_0(y) - r_0(z) \quad (6.12)$$

$$= 2 + r_1(y) + r_1(z) - r_0(x) - r_0(y) \quad (6.13)$$

$$\leq 2 + r_1(y) + r_1(z) - 2r_0(x) \quad (6.14)$$

$$(6.15)$$

After the two rotations, the only ranks that change are the ranks of  $x$ ,  $y$  and  $z$ . Therefore, the change in potential is again defined by the change in their ranks. Equation (6.12) follows from the definition of amortized cost. Now, prior to the two rotations,  $z$  is the root of the subtree under consideration and after the two rotations,  $x$  is the root. Therefore, we have  $r_1(x) = r_0(z)$ . Equation (6.13) follows. Similarly,  $r_0(x) \leq r_0(y)$  since  $x$  is a child of  $y$  prior to the rotations. What remains to be shown is that  $2 + r_1(y) + r_1(z) - 2r_0(x) \leq 3(r_1(x) - r_0(x))$ . This is equivalent showing that  $r_1(y) + r_1(z) + r_0(x) - 3r_1(x) \leq -2$ .

$$r_1(y) + r_1(z) + r_0(x) - 3r_1(x) = \log((s_1(y)/s_1(x)) * (s_1(z)/s_1(x))) + r_0(x) - r_1(x) \quad (6.16)$$

$$\leq \log(1/4) \quad (6.17)$$

$$= -2 \quad (6.18)$$

The argument is similar to the previous case. Equation (6.16) simply follows from the definition of ranks and some manipulation of logs. The inequality (6.17) follows because  $s_1(x) \geq s_1(y) + s_1(z)$  and  $r_0(x) \leq r_1(x)$ .

Now, suppose that a splay operation requires  $k > 1$  steps. Only the  $k$ th step will be a zig step and all the others will either be zigzig or zigzag steps. From the three cases shown above, the amortized cost of the whole splay operation is

$$3r_k(x) - 3r_{k-1}(x) + 1 + \sum_{j=1}^{k-1} (3r_j(x) - 3r_{j-1}(x)) = 3r_k(x) - 3r_0(x) + 1 .$$

□

If each node is given a weight of 1, the above lemma suggests that the amortized cost of performing a splay operation is  $O(\log n)$  where  $n$  is the size of the tree. However, Lemma 3 is surprisingly much more powerful. Notice that the weight of a node was never really specified. We only stated that it should be a positive constant. The power comes from the fact that the lemma holds for **any** assignment of positive weights to the nodes. By carefully choosing the weights, we can prove many interesting properties of splay trees. Let us look at some examples.

**Theorem 14** (Balance Theorem). *Given a sequence of  $m$  accesses into an  $n$  node splay tree  $T$ , the total access time is  $O((m + n) \log n + m)$ .*

*Proof.* Assign a weight of  $1/n$  to each node. The maximum value that  $s(x)$  can have for any node  $x \in T$  is 1 since the sum of the weights is 1. The minimum value is  $1/n$ . Therefore, the amortized access cost for any element is at most  $3(\log(1) - \log(1/n)) + 1 = 3 \log(n) + 1$ . The maximum potential  $\Phi_{\max}(T)$  that the tree can have is  $\sum_{i=1}^n \log(s(i)) \leq \sum_{i=1}^n \log(1) = 0$ . The minimum potential  $\Phi_{\min}(T)$  that the tree can have is  $\sum_{i=1}^n \log(s(i)) \geq \sum_{i=1}^n \log(1/n) = n - n \log n$ . Therefore, the maximum drop in potential over the whole sequence is  $\Phi_{\max}(T) - \Phi_{\min}(T) = n \log n - n$ . □

This means that starting from **any** initial binary search tree, if we access  $m > n$  keys, the cost per access will be  $O(\log n)$ . This is quite interesting since we can start with a very unbalanced tree, such as an  $n$ -node path, and still achieve good behaviour.

**Theorem 15** (Optimality Theorem). *Given a sequence of  $m$  access into an  $n$  node splay tree  $T$ , where every node is accessed at least once, then the total access time is  $O(m + \sum_{i=1}^n q(i) \log(m/q(i)))$ , where  $q(i)$  is the access frequency of node  $i$ .*

*Proof.* Assign a weight of  $q(i)/m$  to each node. The maximum value that  $s(x)$  can have for any node  $x \in T$  is 1 since the sum of the weights is 1. The minimum value is  $q(i)/m$ . Therefore, the amortized access cost for any element is at most  $3(\log(1) - \log(q(i)/m)) + 1 = 3 \log(m/q(i)) + 1$ . The maximum potential  $\Phi_{\max}(T)$  that the tree can have is  $\sum_{i=1}^n \log(s(i)) \leq \sum_{i=1}^n \log(1) = 0$ . The minimum potential  $\Phi_{\min}(T)$  that the tree can have is  $\sum_{i=1}^n \log(s(i)) \geq \sum_{i=1}^n \log(q(i)/m)$ . Therefore, the maximum drop in potential over the whole sequence is  $\Phi_{\max}(T) - \Phi_{\min}(T) = \sum_{i=1}^n \log(q(i)/m)$ . □

This shows that access time in splay trees achieves the entropy bound to within a constant.