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 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 take 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 ith operation and let C_i denote the cost (running time) of the ith operation then the *amortized cost* of the ith operation is defined as

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

Note that this is simply a definition and, since Φ is arbitrary, A_i has nothing to do with the real cost

C_i. However, the the cost of a sequence of m operations is

$$\begin{aligned}\sum_{i=1}^{m} C_i &= \sum_{i=1}^{m} (C_i + \Phi(D_i) - \Phi(D_{i-1})) - \sum_{i=1}^{m} (\Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{m} A_i - \sum_{i=1}^{m} (\Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{m} A_i + \Phi(D_0) - \Phi(D_m) \ .\end{aligned}$$

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 decendants. 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:

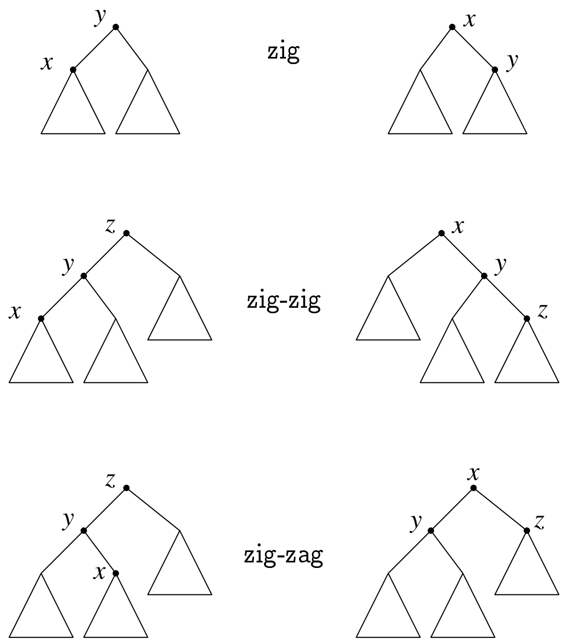

Figure 6.1: Splay tree.

Lemma 3. *(Access Lemma) 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 $a = t + \Phi_1(T) - \Phi_0(T)$ where t is the actual cost of the operation, and Φ_0 and Φ_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 $3(r_1(x) - r_0(x)) + 1$.

$$\begin{aligned} \text{amortized cost} &= 1 + r_1(x) + r_1(y) - r_0(x) - r_0(y) && (6.1)\\ &\leq 1 + r_1(x) - r_0(x) && (6.2)\\ &\leq 1 + 3(r_1(x) - r_0(x)) && (6.3) \end{aligned}$$

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.

$$\begin{aligned}
\text{amortized cost} &= 2 + r_1(x) + r_1(y) + r_1(z) - r_0(x) - r_0(y) - r_0(z) && (6.4)\\
&= 2 + r_1(y) + r_1(z) - r_0(x) - r_0(y) && (6.5)\\
&\le 2 + r_1(x) + r_1(z) - 2r_0(x)) && (6.6)\\
& && (6.7)
\end{aligned}$$

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 $r_1(x) \ge r_1(y)$ and $r_0(x) \le r_0(y)$. What remains to be shown is that $2 + r_1(x) + r_1(z) - 2r_0(x)) \le 3(r_1(x) - r_0(x))$.

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

$$\begin{aligned}
r_1(z) + r_0(x) - 2r_1(x) &= \log(s_1(z)/s_1(x)) + \log(s_0(x)/s_1(x)) && (6.8)\\
&= \log((s_1(z)/s_1(x)) * (s_0(x)/s_1(x))) && (6.9)\\
&\le \log(1/4) && (6.10)\\
&= -2 && (6.11)
\end{aligned}$$

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

$$\begin{aligned}
\text{amortized cost} &= 2 + r_1(x) + r_1(y) + r_1(z) - r_0(x) - r_0(y) - r_0(z) && (6.12)\\
&= 2 + r_1(y) + r_1(z) - r_0(x) - r_0(y) && (6.13)\\
&\le 2 + r_1(y) + r_1(z) - 2r_0(x)) && (6.14)\\
& && (6.15)
\end{aligned}$$

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, $r_0(x) \le r_0(y)$ since x is a child of y prior to the rotations. What remains to be shown is that $2+r_1(y)+r_1(z)-2r_0(x)) \le 3(r_1(x)-r_0(x))$. This is equivalent showing that $r_1(y) + r_1(z) + r_0(x) - 3r_1(x) \le -2$.

$$\begin{aligned}
r_1(y) + r_1(z) + r_0(x) - 3r_1(x) &= \log((s_1(y)/s_1(x)) * (s_1(z)/s_1(x))) + r_0(x) - r_1(x) \quad (6.16) \\
&\le \log(1/4) \quad (6.17) \\
&= -2 \quad (6.18)
\end{aligned}$$

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 $s_1(x) \ge s_1(y) + s_1(z)$ and $r_0(x) \le r_1(x)$.

Now, suppose that a splay operation requires $k > 1$ steps. Only the kth 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, starting with the most mundane.

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

Proof. Assign a weight of 1 to each node. The maximum value that $s(x)$ can have for any node $x \in T$ is n since the sum of the weights is n. The minimum value of $s(x)$ is 1. Therefore, the amortized access cost for any element is at most $3(\log(n) - \log(1)) + 1 = 3\log(n) + 1$. The maximum potential $\Phi_{max}(T)$ that the tree can have is

$$\sum_{i=1}^{n} \log(s(i)) \le \sum_{i=1}^{n} \log(n) = n \log n$$

The minimum potential $\Phi_{min}(T)$ that the tree can have is

$$\sum_{i=1}^{n} \log(s(i)) \ge \sum_{i=1}^{n} \log(1) = 0 \ .$$

Therefore, the maximum drop in potential over the whole sequence is $\Phi_{max}(T) - \Phi_{min}(T) = n \log n$. Thus, the total cost of accessing the sequence is $O(m \log n + n \log n)$, as required. □

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. Next we prove something much more interesting.

Theorem 15 (Static Optimality Meta-Theorem). *Fix some static binary search tree* T^* *and let* $d(x)$ *denote the depth of the node containing* x *in* T^*. *For any access sequence* $a_1, \ldots, a_m$ *consisting of elements in* T^*, *the cost of accessing* $a_1, \ldots, a_m$ *using a splay tree is at most* $O(n \log n + m + \sum_{i=1}^{m} d(a_i)$.

Proof. If $d(x)$ is the depth of node x in T^*, then we assign a weight of

$$w(x) = 1/n + 1/3^{d(x)}$$

to the node containing x in the splay tree. Now, note that the size of the root (and hence any node) is at most

$$\sum_{x \in T} \left(1/n + 1/3^{d(x)}\right) = 1 + \sum_{x \in T} 1/3^{d(x)} \leq 1 + \sum_{i=0}^{\infty} 2^i/3^i = 4 \ ,$$

since T^* has at most 2^i nodes with depth j and these each have weight $1/3^j$.

Now, applying the Access Lemma, we find that the amortized cost of accessing node x is at most

$$\begin{aligned} 3(\log 4 - \log(s(x))) + 1 &= 7 - 3\log(1/n + 1/3^{d(x)}) \\ &\leq 7 - 3\log(1/3^{d(x)}) \\ &= 7 + 3\log(3^{d(x)}) \\ &= 7 + (3\log 3)d(x) \end{aligned}$$

All that remains is to bound the net drop in potential, $\Phi_{max} - \Phi_{min}$. We have already determined that $s(x) \leq 4$ for any node x, so $r(x) \leq 2$ and $\Phi_{max} \leq 2n$. On the other hand, $s(x) \geq w(x) \geq 1/n$ for every node x, so $r(x) \geq -\log n$ and $\Phi_{min} \geq -n \log n$. Putting everything together, we get that the total cost of accessing $a_1, \ldots, a_m$ using a splay tree is

$$\begin{aligned} C(a_1, \ldots, a_m) &\leq \Phi_{max} - \Phi_{min} + \sum_{i=1}^{m} (7 + (3\log 3)d(a_i)) \\ &= 2n - (-n \log n) + \sum_{i=1}^{m} (7 + (3\log 3)d(a_i)) \\ &= n\log n + 2n + 7m + (3\log 3)\sum_{i=1}^{m} d(a_i) \\ &= O\left(n \log n + m + \sum_{i=1}^{m} d(a_i)\right) \ . \end{aligned}$$

□

The Static Optimality Meta-Theorem allows us to compare the performance of a splay tree with that of *any* fixed binary search tree. For example, letting T^* be a perfectly balanced binary search tree on n items—so that $d(x) \leq \log_n$ for every x—and applying the Static Optimality Theorem gives the Balance Theorem.

As another example, in Chapter 5, we saw that when searches are drawn independently from some distribution, then there exists a binary search tree whose expected search time is proportional to the entropy of that distribution. This implies the following corollary of the Static Optimality Meta-Theorem:

Corollary 1. *(Static Optimality Theorem) If* $a_1, \dots, a_m$ *are drawn from a set of size* n*, each independently from the a probability distribution with entropy* H*, then the expected time to access* $a_1, \dots, a_m$ *using a splay tree is* $O(n \log n + m + mH)$*.*

As another example, we obtain the *static finger property*. As an exercise, show how, given any integer $f \in \{1, \dots, n\}$ to design a static tree that stores $1, \dots, n$ and such that the time to search for x is $O(1 + \log|f - x|)$. (Hint: Try the special case $f = 1$ first.) The existence of this tree implies the following result:

Corollary 2. *(Static Finger Theorem) Let* $a_1, \dots, a_m$ *be a sequence of values in* $\{1, \dots, n\}$*. Then, for any* $f \in \{1, \dots, n\}$*, the time to access* $a_1, \dots, a_m$ *using a splay tree is* $O(n \log n + m + \sum_{i=1}^{m} \log|a_i - f|)$*.*

Of course, Corollary 2 is not really restricted to trees that store $\{1, \dots, n\}$; these are just stand-ins for the ranks of elements in any totally ordered set. We finish with one additional property of splay trees that is not covered by the Static Optimality Theorem:

Theorem 16. *(Working-Set Theorem) For any access sequence* $a_1, \dots, a_m$ *the cost of accessing* $a_1, \dots, a_m$ *using a splay tree is at most* $O(n \log n + m + \sum_{i=1}^{m} \log w_i(a_i))$*, where*

$$w_i(x) = \begin{cases} \min\{j \geq 1 : a_{i-j} = x\} & \textit{if } x \in \{a_1, \dots, a_{i-1}\} \\ n & \textit{otherwise.} \end{cases}$$

Proof. Assign the weights $1, 1/4, 1/9, \dots, 1/n^2$ to the values in the splay tree in the order of their first appearance in the sequence $a_1, \dots, a_m$, so that $w(a_1) = 1$ and, if $a_2 \neq a_1$, then $w(a_2) = 1/4$, and so on. Immediately after an accessing and splaying an element a_i, whose weight is $1/k^2$, we will *reweight* some elements as follows: a_i will be assigned weight 1, the item with weight 1 will be assigned weight $1/4$, the item with weight $1/4$ will be assigned weight $1/9$ and, in general, any item with weight $1/j^2$, $j < k$ will be assigned weight $1/(j+1)^2$.

Notice that, with this scheme, the sum of all the weights, and hence the size of the root is

$$\sum_{i=1}^{n} 1/i^2 \leq \sum_{i=1}^{\infty} 1/i^2 = \pi^2/6 \ .$$

(The second, infinite, sum is the subject of the Basel Problem, first solved by Euler in 1735; the solution can be found in most calculus textbooks.) This initial assignment of weights along with the reassignment of weights ensures that the weight of a_i at the time it is accessed is $1/(w_i(a_i))^2$. The Access Lemma then implies that the amortized cost to access a_i is at most

$$3(\log(\pi^2/6) - \log(1/w_i(a_i)^2)) + 1 = 3(\log(\pi^2/6) + 2\log w_i(a_i)) + 1$$

This is all great, except that we also have to account for any increase in potential caused by the reweighting. The reweighting increases the weight of a_i from $1/(w_i(a_i))^2$ to 1. For values other than a_i, the weights only decrease or remain unchanged. Since, by the time we do reweighting a_i is at the root, the reweighting does not increase the rank of a_i. On the other hand, it can only decrease the ranks of other nodes, so overall it can not result in an increase in potential.

To finish, we need only bound Φ_{max} and Φ_{min}. We already have an upper bound of $\log(\pi^2/6)$ on the rank of the root, which gives an upper bound of $\Phi_{max} \leq n \log(\pi^2/6)$. For the lower-bound, note that every node has weight at least $1/n^2$, so it's rank is at least $-2\log n$ and $\Phi_{min} \geq -2n\log n$. To summarize, the total cost of accessing $a_1, \ldots, a_m$ is

$$\begin{aligned} C(a_1, \ldots, a_m) &\leq \Phi_{max} - \Phi_{min} + \sum_{i=1}^{m} 3\left(\log \pi^2/6 + 2\log w_i(a_i)\right) \\ &= (\pi^2/6)n + 2n\log n + 3\log(\pi^2/6)m + \sum_{i=1}^{m} 6\log w_i(a_i) \\ &= O\left(n\log n + m + \sum_{i=1}^{m} \log w_i(a_i)\right) \ . \end{aligned}$$

□

6.2 Pairing Heaps

John Iacono apparently has a mapping between splay trees and pairing heaps that makes all the analyses carry over.

6.3 Queaps

Next, we consider an implementation of priority queues. These are data structures that support the insertion of real-valued priorities. At any time, the data structure can return a pointer to a node storing the minimum priority, and given a pointer to a node, that node (and it's corresponding priority) can be deleted.

For the data structure we describe, the cost to insert an element is $O(1)$ and the cost to extract the element x is $O(\log(q(x))$, where $q(x)$ denotes the number of elements that have been in the data structure longer than x. Thus, if the priority queue is used as a normal queue (the first element inserted is the first element deleted) then all operations take constant time. We say that a data structure that has this property is *queueish*.

The data structure consists of a balanced tree T and a list L. Refer to Figure 6.3. The tree T is 2-4 tree, that is, all leaves of T are on the same level and each internal node of T has between 2 and 4 children. From this definition, it follows immediately that the ith level of T contains at least 2^i nodes, so the height of T is logarithmic in the number of leaves. Another property of 2-4 trees that is not immediately obvious, but not difficult to prove, is that it is possible to insert and delete a leaf in $O(1)$ amortized time. These insertions and deletions are done by splitting and merging nodes (see Figure 6.2), but any sequence of n insertions and deletions results only in $O(n)$ of these split and merge operations.

It is important to note that T is not being used as a search tree. Rather, the leaves of T will store priorities so that the left to right order of the leaves is the same as the order in which the corresponding priorities were inserted.

The nodes of T also contain some additional information. The *left wall* of T is the path from the root to the leftmost leaf. Each internal node v of T that is not part the left wall stores a pointer