6.854 Advanced Algorithms

Lecture 4: September 21, 1999 Lecturer: David Karger

Scribes: Jaime Teevan, Arvind Sankar, Jason Rennie

Splay Trees

4.1 Motivation and Background

With splay trees we are leaving behind heaps and moving on to trees. Trees allow one to perform actions not typically supported by heaps, such as find element, find predecessor, find successor, and print nodes in order. Binary trees are a common and basic form of a tree. As long as a binary tree is balanced, it has logarithmic insert and delete time. The goal of a splay tree is to have the tree maintain a logarithmic depth in an amortized sense by adjusting its structure with each access. This creates a powerful data structure which can be proven to be, in the limit, as good as or better than any static tree optimized for a certain sequence of accesses.

4.1.1 Previous Work

After binary search trees were first proposed, a number of variants were developed to improve on their poor worst case behavior. These include AVL trees, 2-3 trees, red-black trees and many more. Each improved the performance of a simple binary search tree, but left something to be desired. Most require augmentation of the simple tree data structure and none can claim theoretical performance as good as that of splay trees.

The two basic elements of splay trees, self-adjustment and rotation, are hardly new. Many variants on binary trees use some form of rotation; self-adjusting linked lists and heaps had been introduced before splay trees. Splay trees were basically the right orchestration of ideas that were known for some time. While variants on binary search trees often aggressively pursue maintaining a balance, splay trees deal with the problem lazily, doing a little bit of work with each tree operation, but making little obvious effort to maintain a nicely balanced tree.

4.1.2 Intuition

This laziness is instantiated in the splay tree's self-modification with every access. It would be nice if we could show that the work done for accessing an element is at most $O(\log n)$. This isn't possible, so we do the next best thing, that is, we show that any additional work that we do (beyond $O(\log n)$) can be accounted for by our past laziness. In other words, if we do work $> O(\log n)$, we want to show that there were a lot of operations before this one where we did less work than we were allowed, and thus, we can amortize away the work that is $> O(\log n)$ by spreading it around to previously performed operations. As we all know by know, this technique is called amortization.

If the path to find a node is long, it implies that the tree is poorly balanced. Thus, as you travel from a node to a new node, most of the tree below is the new node. A double rotations distributes some of the weight below the queried node over its siblings. The total amount of imbalance, which is measured by the sum of ranks, is a potential function against which we charge the cost of long searches.

4.2 Tree Rotation

Tree rotation can be thought of as a way to move a node to a higher position in a binary search tree without affecting the ordering properties. The simplest rotations are called *single rotations*; they involve two nodes and their corresponding subtrees. Figure 4.1 displays such a simple rotation. When read from left to right, the rotation brings node x to the top of the tree.

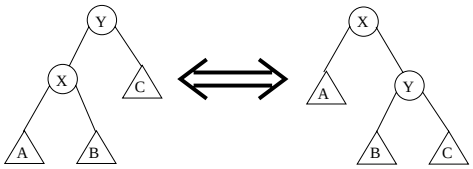

Figure 4.1: A zig rotation

4.2.1 Double rotations

A slightly more complication rotation is a *double rotation*. Here, three nodes and their subtrees are involved; a *double rotation* essentially performs two single rotations in sequence. Figure 4.2 is a depiction of what is known as a *zig-zig* rotation. When following the arrow from left to right, the node x is brought up two levels to the top of the subtree. Figure 4.3 is a depiction of what is known as a *zig-zag* rotation. Any weight below x is spread more evenly as x is brought to the root of the subtree.

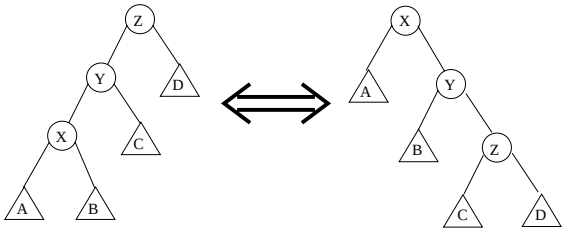

Figure 4.2: A zig-zig rotation

The *double rotation* is one of the key elements that make Splay Trees the success that they are. While single rotations can move a queried element to the top of a tree without affecting key ordering, only

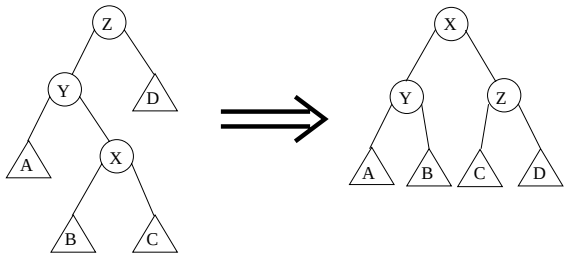

Figure 4.3: A zig-zag rotation

double rotations allow yield the balancing properties that give splay trees static optimality and $O(\log n)$ operations.

4.3 Running time

Splay trees run all basic operations in $\log(n)$ time. We can prove this through manipulation of weights and potential function. We define $w(x)$ to be the weight of node x. Then:

$$\begin{aligned} s(x) &= \sum_{y \in \text{descendants}} w(y) \\ r(x) &= \log_2 s(x) \\ \Phi(DS) &= \sum w(x) \end{aligned}$$

$s(x)$ is the sum of the weights of all descendants of x, including the weight of x itself. $r(x)$ is called the rank of x. Φ is the potential function that we use for proving properties about splay trees. DS represents the splay tree data structure. We can use this notation as a basis for proving the theorem that underlies the $\log(n)$ running time of splay trees:

Theorem 1 (Access Theorem) *The amortized time to access x from root t is at most* $3(r(t) - r(x)) + 1$.

Proof: Using the simple lemma that we prove below, we will show that the amortized cost of a double rotation is $\leq 3(r(t) - r(x))$ and the amortized cost of a single rotation is $\leq 3(r(t) - r(x)) + 1$. We will show that a sequence of rotations yields a telescoping sum that results in the bound described in the Access Theorem.

Lemma 1 *Given b as a root with two children, a and c,* $r(a) + r(c) - 2r(b) + 2 \leq 0$.

Proof: Consider the equivalent inequality among the sizes of the three nodes

$$4s(a)s(c) \leq s(b)^2$$

We know that

$$s(b) = s(a) + s(c) + w(b)$$

by definition. Since the weights are non-negative, we obtain

$$s(b) \geq s(a) + s(c)$$

Hence

$$(s(a) + s(c))^2 - 4s(a)s(c) \leq s(b)^2 - 4s(a)s(c)$$

or

$$(s(a) - s(c))^2 \leq s(b)^2 - 4s(a)s(c)$$

Since squares are always non-negative, we get the desired inequality

$$0 \leq s(b)^2 - 4s(a)s(c)$$

Consider the amortized cost of performing one rotation. Let's say that z is the root of a subtree and that node x is either a child or grand child of z. If we can show that the amotized cost of any double rotation $\leq 3(r(z) - r(x))$ and the amortized cost of any single rotation is $\leq 1 + 3(r(z) - r(x))$, then the amortized cost of splaying an element x_0 to root x_k is

$$\begin{aligned} C &= 1 + 3(r(x_k) - r(x_{k-1})) + 3(r(x_{k-1}) - r(x_{k-2})) + \ldots + 3(r(x_1) - r(x_0)) && (4.1) \\ &= 1 + 3(r(x_k) - r(x_0)) && (4.2) \end{aligned}$$

where the x_i's $(i \geq 1)$ are roots of subtrees that are rotated.

Now we will show that the amortized time for each rotation involved in the splaying of node x is at most $3(r'(x) - r(x))$, and the time for a single rotation is at most $1 + 3(r'(x) - r(x))$, where r' denotes the rank of a node after the rotation. There are three cases to be considered:

ZIG This is the rotation shown in figure 4.1. The nodes that change ranks are x and y. So the amortized cost is

$$1 + r'(x) + r'(y) - r(x) - r(y)$$

But $r'(x) = r(y)$, and $r'(y) \leq r'(x)$. So the cost is at most

$$1 + r'(x) - r(x) \leq 1 + 3(r'(x) - r(x))$$

ZIG-ZAG This is the double rotation shown in figure 4.3. Now three nodes change rank, namely x, y and z. The amortized cost is

$$2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$

We note that $r'(x) = r(z)$, and $r(y) \geq r(x)$, so that the cost is at most

$$2 + r'(y) + r'(z) - 2r(x)$$

which we rewrite as

$$2(r'(x) - r(x)) + r'(y) + r'(z) - 2r'(x) + 2$$

which by the lemma is at most

$$2(r'(x) - r(x)) \leq 3(r'(x) - r(x))$$

ZIG-ZIG The other kind of double rotation is shown in figure 4.2. Again three nodes x, y and z change ranks. The amortized cost is therefore

$$2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$

We have $r'(x) = r(z)$, so the cost simplifies to

$$2 + r'(y) + r'(z) - r(x) - r(y)$$

Since $r'(x) \geq r'(y)$, and $r(y) \geq r(x)$, this expression is at most

$$2 + r'(x) + r'(z) - 2r(x)$$

We want this to be less than or equal to $3(r'(x) - r(x))$, so we need to show that

$$r(x) + r'(z) - 2r'(x) + 2 \leq 0$$

This looks somewhat like the inequality we proved in the lemma, so we try to see if that can be applied. The proof depended only on the fact that the sum of the sizes of the two subtrees was at most the size of the entire tree. Clearly, that remains true:

$$s(x) + s'(z) \leq s'(x)$$

So we can still apply the lemma, and get the desired inequality.

Now we can sum over all the rotations that are needed to splay x to the root. The sum telescopes, and we get the desired bound.

Note that $r(t) = \log s(t)$ and $r(x) = \log s(x)$, so $3(r(t) - r(x)) = O(r(t) - r(x)) = O(\log(s(t)) - \log(s(x))) = O(\log(\frac{s(t)}{s(x)}))$. In the future, we will denote $s(t)$ as W, which is the sum of all of the weights in the tree rooted at t. We also use w_x to denote the weight of a single node x. Note that $w_x \leq s(x)$. As a result, the amortized time to splay a node, x, to the root, t, is $O(\log(\frac{W}{w_x}))$.

Now we will show that the basic tree operations insert and delete are $O(\log n)$. We will do this by defining two operations, split and join, that can be used to easily implement insert and delete.

Join(T_1, x, T_2) is a function of trees T_1, and T_2 and element x where $t_1 < x < t_2$ $\forall t_1 \in T_1, t_2 \in T_2$. Join($T_1$, x, T_2) returns a tree where x is the root with T_1 as its left subtree and T_2 as its right subtree. Join(T_1, x, T_2) has amortized cost $O(\log n)$. Note that we can also join two trees $T_1 < T_2$ can be joined by splaying the rightmost element of T_1 and making T_2 its right subtree. This has amortized cost $O(log n)$.

Split(T, x) is a function of a tree T and an element x. Split(T, x) returns two trees $T_1 \leq x < T_2$. Note that x simply defines a boundary between the two trees; x is not necessarily contained in T_1. Split(T, x) is implemented by splaying the greatest element less than x to the root. We remove the right subtree and call it T_2. The remaining tree is named T_1.

We are now prepared to implelement insert and delete with $O(\log n)$ amortized cost:

Delete(T, x) is a function of a tree T and an element x. We perform split(T, x) to yield two subtrees, T_1 and T_2. Note that since x was an element of T, it must now be the root of T_1. Since the root of

T_1 does not have a left subtree, we may remove x in constant time. We then join the two trees T_1 and T_2. Since we have done a constant number of $O(\log n)$ operations, the amortized cost for delete is $O(\log n)$.

Insert(T, x) is a function of a tree T and an element x. We perform split(T, x) to yield two subtrees, T_1 and T_2. We then join(T_1, x, T_2) to yield our resultant tree. The amortized cost for insert is $O(\log n)$.

4.4 Applications

Corollary 1 *In n-item tree, access time is $O(\log n)$ per operation. Let $w_x = 1$.*

$$W = \sum_{x \in nodes} w_x = n$$

This means that $O(\log \frac{W}{w_x}) = O(\log \frac{n}{1}) = O(\log n)$.

Corollary 2 *Splaying is "competitive" against any fixed binary tree. Imagine you have a the ideal fixed binary tree. Every item in that tree is assigned a depth d. Let $w_x = 3^{-d}$.*

$$W = \sum_{x \in nodes} w_x \leq \sum_{x} 2^x 3^{-d} \leq 1$$

The amortized cost to get a depth d item in a splay tree is $O(\log \frac{W}{w_x}) = O(\log \frac{1}{3^{-d}}) = O(d)$.

Corollary 3 *Static Optimality Theorem. Splaying is competitive against the best possible tree. m is the total number of accesses to a tree. p_x is the fraction of times that x will be accessed, making $p_x * m$ the access frequency for item x. Optimal access time is*

$$\Omega(m \sum_{x \in nodes} p_x \log \frac{1}{p_x})$$

Let the weight for item x be $w_x = p_x$. Amortized cost for x is $O(\log \frac{W}{w_x}) = O(\log \frac{1}{p_x})$

Any information theorist will recognize this sum as the entropy of a probability distribution. If we think of code lengths for items instead of node depths, then the above equation is the the average code length required to send information about m items given the underlying probability distribution. As one might guess, we can use information theory to determine the optimal static tree for a given access pattern.

Corollary 4 *Static Finger Theorem. Suppose when you search for an element, you leave a finger, f, and begin the next search from f. We show that a splay tree performs as well.*

$$\begin{aligned} w_x &= \frac{1}{(1+(x-f))^2} \\ \sum_{x \in nodes} w_x &= O(\sum_x \frac{1}{x^2}) = O(1) \end{aligned}$$

Amortized cost for x is $O(\log \frac{W}{w_x}) = O(\log \frac{1}{\frac{1}{(1+(x-f))^2}}) = O(\log |x-f|)$.

Corollary 5 *Working Set Theorem. Access item x_j at time j. Let t_j be the number of distinct items since previous x_j access. Thus the amortized cost of an access is* $O(\log t_j)$.

4.5 Practicalities

A major drawback of splay trees is that they require modification on every access. Modern computer systems normally use a cache to optimize for very fast read operations. Frequent changes to the data structure means frequent cache invalidations and an ineffective cache. There are several tricks that can be used to overcome this obstacle. One is to splay for a while, and then to stop. This only works well when the access frequency of each node remains relatively constant. Another trick is to only splay on operations that require $\geq \log n$ units of work. This allows splay tree accesses to remain $O(\log n)$ while reducing the amount of tree modification that must be performed.