6.897: Advanced Data Structures Spring 2005

Lecture 3 — February 8, 2005

Prof. Erik Demaine *Scribe: Jeff Cohen*

1 Introduction

In the previous lectures we discussed hashing, presenting data structures capable of achieving excellent bounds for solving dictionary and membership problems. The model allowed us to compute hash functions; for instance, the elements can be integers or strings.

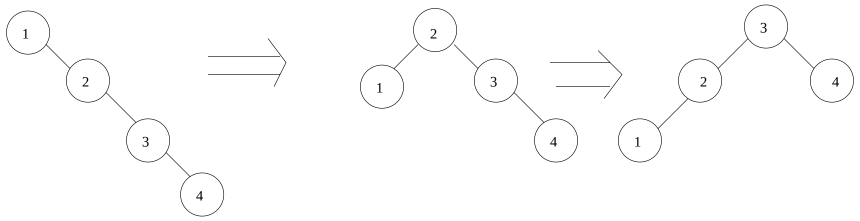
This week, we will work in a more restricted and abstract setting, namely the comparison model: elements can only be compared and the result is $<, =$ or $>$. Furthermore, we only consider Binary Search Trees. In this model of computation, we are given n keys placed in the n nodes of a binary tree in *symmetric order* (an inorder traversal of the tree gives the elements in sorted order). We are also given a pointer, which at the beginning of each operation is at the root. A unit-cost manipulation of the tree can consist of: moving the pointer to the left child, right child, or parent of the node to which it currently points, or performing a rotation.

When searching for x, our goal is to make the pointer point to the node x. This solves the dictionary problem, because then we could retrieve whatever data is associated with x. However, because of the restrictions in the model, BSTs can solve harder problems: searching implicitly determines the predecessor or successor of a given x, if x is not in the set. Because these operations are essentially equivalent to searching, it is customary to study BSTs in a simplified setting. We only care about a static set of n elements (though the BST representing them *can* change in shape), and we only consider successful searching.

2 The Cost of Searching

Note that in the worst case, the cost per access is $\Omega(\lg n)$, because, even in any tree, the deepest node is $\Omega(\lg n)$ deep. For some access sequences, however, we may be able to do better (at least in an amortized sense). This lecture is about which access sequences allow us to beat the $O(\lg n)$ bound, how we can beat it, and what bounds we can achieve.

We begin by presenting preliminary examples of sequences for which we can achieve $o(\lg n)$ bounds. A general access sequence is of the form $\langle x_1, x_2, x_3, \ldots, x_m \rangle$; we always use m for the total number of operations. This allows correlation among the accesses. For instance, if the sequence is $\langle 1, 2, ..., n \rangle$ $(m = n)$, we can achieve $O(1)$ cost per access by starting with a linear, sorted tree and rotating it after each access:



In the stochastic model, the x_j's are i.i.d. random variables: element $i \in [n]$ is accessed with probability p_i. In a degenerate case where $(\exists)j : p_j \approx 1$, storing j at the root allows us to achieve roughly $O(1)$ time per access. In general, this setup reminds one of Huffman trees, which are data structures that store data only at the leaves, and want to minimize the expected depth of an element chosen according to the given distribution. This expected depth matches the entropy of the distribution, upto an additive 1. The entropy is defined as:

$$H = \sum_i p_i \lg \frac{1}{p_j} + O(1)$$

BSTs are more constrained in that they must arrange nodes in symmetric order in the tree. However, we will see that it is still possible to match the entropy bound (up to an additive 3).

3 Competitiveness and Optimality

We define $OPT(x)$ to be the minimum cost to perform the sequence of accesses $x = \langle x_1, \ldots, x_m \rangle$, over all possible BST algorithms. Note that we can tune how the BST changes to the choice of the sequence x. In particular, the algorithm knows the future: it can handle x_i in a certain way, optimizing also for future accesses. So OPT is an offline optimum.

We say that a BST algorithm is α-competitive if $COST(x) \leq \alpha OPT(x)$ for all x. In this case, the BST algorithm should be online: it must handle x_i without knowing what future accesses $x_{i+1}, \ldots, x_m$ will be. We say that a BST is dynamically optimal if it is $O(1)$-competitive. Dynamic optimality has a profound meaning: knowing the future can only improve your bound by constant factor. Thus, there does not exist any tree which is better than a dynamically optimal tree, not even on a single sequence of accesses. Whether there exists a dynamically optimal BST is a long-standing open problem, but an $O(\lg \lg n)$-competitive one has been found [3].

If we disallow rotations (we force the tree to be static), we can find an optimal BST via dynamic programming. First observe that only the frequency of accessing each element matters; because the tree is static, temporal correlations are inconsequential. The DP looks at each element r, and tries to place r at the root. For each r, we have two subproblems: finding an optimal left subtree and an optimal right subtree. In general, the subproblems are finding the optimal tree for an interval of the elements, so there are $O(n^2)$ subproblems. Trying all roots for each one gives a dynamic program running in time $O(n^3)$. This can be improved to $O(n^2)$, using a technique of Knuth [6]. Whether we can achieve a faster result is an open problem.

In this static case, the optimal cost per access is bounded by:

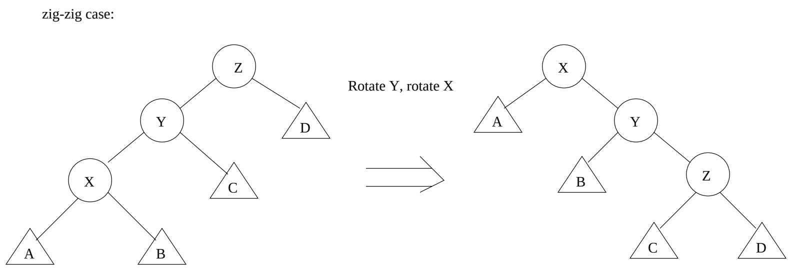
$$H - \lg H - \lg e + 1 \le \frac{OPT}{access} \le H + 3$$

Here $H = \sum_i p_i \lg \frac{1}{p_j} + O(1)$. If we are in the stochastic model p_i's are actual probabilities, and the "cost" is the expected cost of accessing an element from that distribution. If we have a deterministic sequence of accesses, p_i is the number of appearances of i in our sequence divided by m (the empirical probability). A BST is said to be statically optimal (or alternatively to have an entropy bound) if its cost is $O(H)$. This means that the BST is $O(1)$-competitive against any static tree. It is useful to achieve this without knowing the sequence of accesses in advance (i.e. give an online algorithm). Note that our optimal construction is offline, because the p_i's have to be known in advance. Splay trees can achieve this, and other remarkable bounds, online.

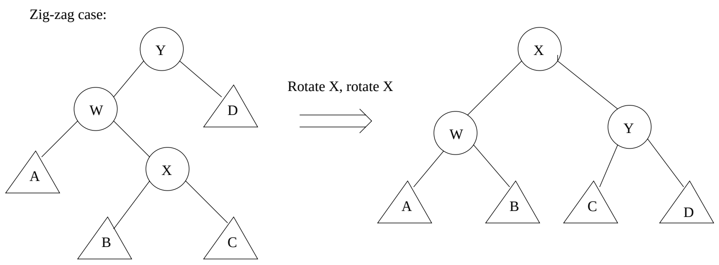
4 Splay Trees

Splay trees were introduced in a famous paper by Sleator and Tarjan [7]. They handle a search for an element x by always bringing x to the root. First, we locate x by walking down the tree, and comparing with the elements. Then, depending on the order of x, its parents, and its grandparent, we do one of the following two cases:

The zig-zig case:



The zig-zag case:



At the end of this, we may need to do one more rotation, if x is the son of the root. Splay trees are an unusual BST, in that they do not explicitly maintain any balance, or any auxiliary information. However, they have a series of remarkable properties, such as static optimality; there is also a rich set of conjectures, the most important of which is dynamic optimal.

4.1 Access Theorem

This theorem implies many interesting bounds on splay trees. Suppose we assign an arbitrary weight w_i to each element i in a splay tree. Define $s(x)$ by:

$$s(x) = \sum_i \{w_i \mid \text{i is in the subtree rooted at x}\}$$

Define a potential function ϕ (for amortized analysis) by:

$$\phi = \sum_x \lg s(x)$$

An intuition behind this is that ϕ is low if the tree is balanced and high if it is unbalanced.

Lemma 1. *The amortized cost of one splay step is bounded by:*

$$cost \leq 3(\lg s^{new}(x) - \lg s^{old}(x))$$

The proof of this is simply a calculation of the effect of a splay step.

Theorem 2 (access theorem). *The amortized cost of accessing x is bounded by:*

$$cost \leq 3(\lg s^{new}(x) - \lg s^{old}(x)) + 1$$

This follows from the previous lemma, because summing the costs of every splay step gives a telescoping sum. The $+1$ is added for the final rotation that we may have to do. Observe the $s^{new}(x)$ includes the weight of the entire tree, because x is now at the root.

As always in amortized analysis, we need to ensure that the potential cannot vary too much, because otherwise we would have a high additive cost to pay for a large store of initial potential, and our analysis would break down. We do this by bounding the minimum and maximum potential:

$$\begin{aligned}
\min \phi &\geq \sum_i \lg w_i \\
\max \phi &\leq n \lg \sum_i w_i \\
\max \phi - \min \phi &\leq \sum_i (\lg \sum_j w_j - \lg w_i)
\end{aligned}$$

In all applications from below, this implies a startup cost of $O(n)$ or $O(n \lg n)$. This is just an additive constant per operation if, say, $m = \Omega(n \lg n)$.

4.2 Consequences of the Access Theorem

Log Amortized. If $w_i = 1$ for all i, then $s(\text{root}) = n$, and the amortized cost of an operation is $O(\lg \frac{n}{k})$ for some $k \geq 1$, so the overall bound for the speed of an operation is $O(\lg n)$.

Static Optimality. If $w_i = p_i = \frac{f_i}{m}$, where f_i is the number of accesses to i in our sequence of m accesses, then $s(\text{root}) = 1$ and $s(x) \geq w_x = p_x$, so that the time for an access is $O(\lg \frac{1}{k})$ for some $k \geq p_x$, so the time is $O(\lg \frac{1}{p_x})$. Note that adding up $COST(x)$ for all x gives entropy (amortized, and to within a constant factor). Remarkably, we don't need to know the values of p_i to execute the algorithm; we only used them in the analysis. Thus, we can get static optimality, even with an online algorithm.

Static Finger Theorem. If you fix your finger (pointer) on some node f, the distance from f to a node i is $|i-f|$ (this is the distance in *rank space*). Choose $w_i = \frac{1}{(i-f)^2}$, and $w_f = 1$. Then $s(\text{root}) \leq 1+2\sum_{k=1}^{\infty} \frac{1}{k^2} = 1+\frac{\pi^2}{3}$. The cost of an operation is thus $O(\lg \frac{O(1)}{w_i}) = O(\lg{(i-f)^2}) = O(\lg{(|i-f|)})$ (except for $i = f$, when it is $O(1)$). This theorem means that if you access something near the finger, the access is cheap (logarithmic in the rank distance). Again, this is a truly remarkable fact, because splay trees do not know where the finger is. Thus, they achieve this bound for any possible finger simultaneously (but the finger must still be static).

Working-Set Theorem. Let $t_i(y)$ be the number of distinct elements (including y) accessed since the last time y was accessed before time i. Then, the amortized cost to access x_i is $O(1+\lg t_i(x_i))$. This bound says that if accesses concentrate on a smaller set of elements (the working set), the cost is the logarithm of this set, not of n. Such a working set phenomenon is frequently observed and used, e.g., by caches. Equivalently, the bound says that if you access something you accessed recently, the access is cheap.

To show this bound, we will change the potential function dynamically. At time i, let $w_y = \frac{1}{(t_i(y))^2}$. It can be seen that $t_i(y)$ is unique for each y, so $w(root) = \sum_1^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$. When we access x_i, the amortized cost is $O(1 + \lg \frac{O(1)}{t_i(x_i)^{-2}}) = O(\lg t_i(x_i))$. After the operation, we change $t_i(x_i)$ to 1 and increment t_i for every node accessed since the previous access to x_i; that is, we increment $t_i(y)$ for all y such that $t_i(y) < t_i(x_i)$ before we set $t_i(x_i)$ to 1. This increases w_{x_i} to 1 and decreases w_y slightly for all such y. The only node whose weight increases (and could cause a problem by increasing ϕ) is the new root, so it now appears in just one term of ϕ. It can be shown through direct calculation that the potential can only decrease as the result of changing the weights in this way.

4.3 Other Bounds for Splay Tress

There are other interesting bounds that do not follow from the access theorem, and which were discovered after the original work of Sleator and Tarjan [7].

Scanning Theorem. This states that accessing all the elements of a splay tree in order, beginning from any point, takes $O(n)$ time [9].

Dynamic Finger Theorem. This states that accessing element x_i costs $O(\lg(1 + |x_i - x_{i-1}|))$ amortized. Thus, we have finger which is moved dynamically with new accesses (note that this implies the static finger bound). This was a conjecture from the original paper by Sleator and Tarjan [7], until it was finally proved by Cole [2] through a very complicated analysis.

Deque Conjecture. This is a special case of having two dynamic fingers, one on the left and the other on the right. At any step, we can move any of these two fingers by 1 element to the left or to the right, simulating a doubly ended queue (deque). The total cost is known to be $O((m+n)\alpha(m+n))$ [8] and conjectured to be $O(m+n)$ [9].

Traversal Conjecture. This states that accessing elements in order, from any starting tree for which we know a preorder traversal, takes O(n) time [7].

Unified Conjecture. This conjecture states that the time to access x_i is:

$$O(\lg \min_y [t_i(y) + |x_i - y| + 2])$$

Note that $t_i(y) + |x_i - y|$ is the sum of temporal displacement from previous accesses and spatial displacement between nodes. This conjecture means that if you access something near something you accessed recently, the access is cheap – a generalization of the working set and finger bounds. This is proven to be achievable on a pointer machine [1, 4], but it is not known if it is achievable with a BST.

4.4 Implications of Bounds

There are a number of implications relating the bounds on splay trees [4, 5]. The Unified Conjecture implies both the Working Set Theorem and the Dynamic Finger Theorem. The Working Set Theorem implies the Static Optimality bound. The Static Optimality bound and the Dynamic Finger Theorem each imply the Static Finger Theorem.

References

[1] M. Badoiu, E. Demaine, *A Simplified, Dynamic Unified Structure*, Latin American Theoretical INformatics (LATIN), p. 466-473, 2004.

[2] R. Cole, *On the dynamic finger conjecture for splay trees. Part II: The proof*, SIAM Journal on Computing 30(1), p.44-85, 2000.

[3] E. Demaine, D. Harmon, J. Iacono, M. Patrascu, *Dynamic Optimality–Almost*, Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2004), p. 484-490, October 17-19, 2004.

[4] J. Iacono, *Alternatives to splay trees with O(log n) worst-case access times*, Symposium on Discrete Algorithms (SODA), p. 516-522, 2001.

[5] J. Iacono, *New upper bounds for pairing heaps*, Scandinavian Workshop on Algorithm Theory (SWAT), p. 32-45, 2000.

[6] D. Knuth, *Optimal binary search trees*, Acta Informatica 1, p. 14-25, 1971.

[7] D. Sleator, R. Tarjan, *Self-Adjusting Binary Search Trees*, Journal of the ACM 32, p. 652-686, 1985.

[8] R. Sundar, *On the Deque conjecture for the splay algorithm*, Combinatorica 12(1), p. 95-124, 1992.

[9] R. Tarjan, *Sequential access in splay trees takes linear time*, Combinatorica 5(5), p. 367-378, November 4, 1985.