6.851: Advanced Data Structures Spring 2007

Lecture 2 — 14 February, 2007

Prof. Erik Demaine *Scribes: Christopher Moh, Aditya Rathnam*

1 Overview

In the last lecture we covered linked lists and some results regarding the move-to-front heuristic. We discovered that although move-to-front was competitive in the single finger model, it was not competitive in a multi-finger model, when compared to the Order-By-Next-Request (OBNR) algorithm. In today's lecture we will talk about Binary Search Trees (BSTs) instead of lists, and in particular, splay trees.

BSTs store elements in internal nodes, and every node satisfies the following *symmetric* property: for every non-leaf node, the value of its left child (if present) is less than or equal to the value of the node, and the value of the right child (if present) is more than or equal to the value of the node.

We keep a pointer to the current element, initially at the root, and allow the following operations: move the pointer to the left, to the right, or to rotate the current node. Note that rotations are unambiguous as each node has at most one sibling. We can perform accesses by simply moving down the left or right pointers depending on how the value at the node compares with the search value. Unless stated otherwise (notably in the finger theorems stated below), we assume that the pointer starts at the root for every access.

1.1 One Extra Claim from Last Lecture

Before we start, we will make the following claim:

Claim 1. *The amortized cost of $OBNR(x)$, that is, the cost OBNR achieves when accessing element x is less than or equal to $1 + 4\lceil \log(w_x) \rceil$ where w_x is the number of distinct elements accessed since the last access to x, including x itself.*

The above bound is known as a *working set bound*: The running time of OBNR when all accesses are fairly close to each other is logarithmic in the size of this so-called working set, and not in the size of the entire set of elements.

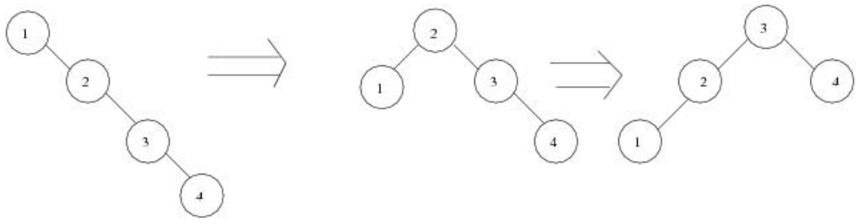
2 Cost Models for BSTs

We will examine the basic cost models but first some quick notation. A general access sequence is of the form $\langle x_1, x_2, x_3, \ldots, x_m \rangle$; where m denotes the total number of acesses. We use n to denote the number of elements in the BST.

For the static and dynamic models, we will examine the idea of competitiveness. In general, an online algorithm is said to be α-competitive if for all access sequences x, the cost of the algorithm on sequence x, $COST(x) \leq \alpha OPT(x)$ where $OPT(x)$ is the minimum possible cost of accessing sequence x. Note that under certain models, $OPT(x)$ can be computed offline, that is, if all accesses are known in advance. We say that an online algorithm is optimal if it is $O(1)$-competitive.

2.1 Worst case

In the worst case, the cost per access is $\Omega(\lg n)$ for a balanced BST like a red-black tree or an AVL tree. However, we hope to do better than that. We begin by presenting preliminary examples of sequences for which we can achieve $o(\lg n)$ bounds. 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:



2.2 Stochastic and Static models

In the stochastic model, the x_j's in the access sequence are i.i.d. random variables; element $i \in [n]$ is accessed with probability p_i. In a degenerate case where $\exists i : p_i \approx 1$, storing element i at the root allows us to achieve roughly $O(1)$ time per access.

In the static model, the x_j's are not necessarily independent. This minor change has significant implications: For example, we will see below that the Transpose algorithm is stochastically optimal but not statically optimal.

In general, the setup is similar to a Huffman tree, which is a data structure that stores data only at the leaves, and attempts to minimize the expected depth of an element chosen according to the given distribution. This expected depth matches the entropy of the distribution, up to an additive 1. The entropy is defined as:

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

The Huffman tree achieves the entropy bound $O(H)$, which from information theory is the optimal bound. 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 factor). We also note that it is possible for BSTs to slightly beat the entropy bound for some access sequences because unlike Huffman trees (which achieve the entropy bound) they can store elements in internal nodes.

In the stochastic and static models, we disallow rotations; that is, the tree is static and cannot be modified from its initial configuration. Under this model, we can find an optimal BST via dynamic programming (DP) and in this way for any sequence x calculate $OPT(x)$.

First observe that only the frequency of accessing each element matters; because the tree is static, temporal correlations are inconsequential. 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). 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.

$$Cost(r) = 1 + Pr(left) * Cost(left) + Pr(right) * Cost(right)$$

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 subproblem gives a dynamic program with run time $O(n^3)$. Knuth improved the DP to run in $O(n^2)$ [?]. Whether we can achieve a better result is an open problem (3-SUM hard?). In this static case, the optimal cost per access is bounded by:

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

From information theory, $O(H)$ is the statically optimal bound. This means that this (offline) BST is $O(1)$-competitive against any static tree. Here we have to know the p_i's in advance and thus the algorithm is offline. As we will see later, splay trees give us an online statically optimal algorithm.

2.3 Dynamic model

In linked lists, the constant-finger model proved to be better than the one-finger model. For trees however, we simply examine the one-finger model as the constant-finger model does not provide much added benefit. Building on the Sleator and Tarjan cost model from the last lecture, we pay 1 unit for each standard rotation we perform using a node x and its parent. We use the notation Rotate(x) to specify that we are rotating x with its parent. In the subsequent two sections, we will consider some algorithms for self-organizing trees and conclude by detailing the properties of splay trees.

There is no known algorithm to calculate the cost of the optimal dynamic binary search tree for all access sequences. In addition, it is not known whether any online BST is $O(1)$-competitive.

3 BST Access Algorithms

3.1 Transpose

The Transpose algorithm for BSTs takes the search element x and performs a standard rotation to move x one level closer to the root everytime it is requested. Unfortunately, this algorithm is bad for uniform searches because it generates all binary trees with equal probability [?]. This gives us an average search cost of $O(\sqrt{n})$, which is worse than a balanced binary tree.

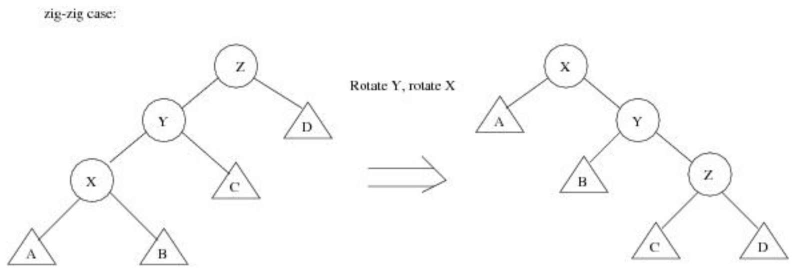
3.2 Move-to-root

The Move-to-root algorithm extends the Transpose idea by repeatedly rotating the search element x all the way to the root, every time it is requested. The repeated rotations result in better search performance - the search cost is within a factor of $2ln2 \approx 1.38$ of the statically optimal tree in the stochastic model. This is still not statically optimal however, as repeated rotations cause many nodes to get too deep into the tree, causing a performance hit when they are requested.

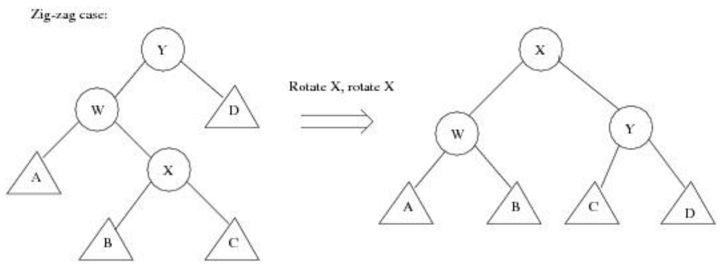
4 Splay Trees

Splay trees were first introduced by Sleator and Tarjan [?] and are an elegant self-organizing binary search tree. Splay trees search for an element x and then always bring x to the root. First, we locate x by walking down the tree and comparing with the elements. Then, depending on the placement of x, we perform one of two special rotations shown below until we bring x to the root. At the end, we may need to perform a regular rotation if x becomes the child of the root. Splay trees do not maintain any balance or auxiliary information but nonetheless, achieve a remarkable set of properties including static optimality and are conjectured to satisfy dynamic optimality as well.

The zig-zig case:



The zig-zag case:



4.1 Access Lemma

We state a fundamental result about Splay trees without proof. For details, consult [?].

Lemma 2. *For each element x_i in the tree, we give it an arbitrary positive weight w_i. For any node x, let W_x be $\sum_{i \in subtree(x)} w_i$. Let the potential Φ be denoted as $\sum_x \log W_x$. Then the amortized cost* *of a single rotation (zig-zig or zig-zag) is at most* $3(\log W_x^{new} - \log W_x^{old})$*, where the superscripts old and new denote the values of* W_x *before and after the rotation.*

Corollary 3. *From telescoping the sum above through all rotations needed in a splay step, the amortized cost of access*(x) *is at most* $O(\log W_{tree} - \log W_x^{old}) + 1$*, where* W_{tree} *be the value of* W_x *when* x *is the root of the tree.*

The above *access lemma* can be used to prove many theorems about splay trees, as we will see below. It counts rotations, instead of the path length needed to access the element from the root, but this is simply a constant factor difference, as the splay step rotates the accessed element all the way back to the root. To see that the potential function makes sense, consider a perfectly balanced tree and a perfectly unbalanced tree (i.e. a linked list), and assume that $w_i = 1$ for all elements. The potential of the perfectly balanced tree is $O(n)$, while that of the linked list is $O(n \log n)$. This confirms our intuition that unbalanced trees should have a higher potential.

Finally, we note that the above lemma only holds when the number of accesses is $\Omega(n \log n)$. This is because unlike standard amortized analysis, the initial potential is non-zero. We can only ignore this initial potential (so-called "startup cost") if the number of accesses is large enough. Since $\Phi_{max} - \Phi_{min} = O(n \log n)$, where Φ_{max} and Φ_{min} denote the maximum and minimum possible potential, we need the number of accesses to be $\Omega(n \log n)$.

5 Results and Conjectures about Splay trees

5.1 Results following from the Access Lemma

We will give four results that follow directly (with some ingenuity) from the access lemma.

5.1.1 Logarithmic Access Cost

For every element in the tree, we give it weight $w_i = 1$. Then $W_{tree} = n$ and the cost of access is $O(\log n - \log 1) = O(\log n)$. Intuitively, this lemma is pleasing because it confirms that splay trees do no worse than balanced BSTs.

5.1.2 Static Optimality Theorem

Let each element i have a fixed probability of access $p_i = \frac{f_i}{m}$, where m is the total number of accesses and f_i the frequency of accessing element i. Set $w_i = p_i$. Since the sum of probabilities is 1, then $W_{tree} = 1$, and the cost of accessing element i is $O(\log 1 - \log p_i) = O(-\log p_i)$.

This result shows that splay trees achieve the entropy bound and thus do as well as the best offline static tree. The most amazing thing about this is that splay trees are completely online and have no idea what p_i is, yet they still achieve this bound. Although we used p_i in the analysis, the splay tree is completely ignorant of it.

5.1.3 Static Finger Theorem

Suppose we fix one of the elements f in the tree as a "finger". That is to say, all searches start from this "finger" instead of from the root. Set $w_i = \frac{1}{1+(i-f)^2} < \frac{1}{(i-f)^2}$ when $i \neq f$. Then the weight of the tree $W_{tree} \leq 2\sum_{d=1}^{n+1} \frac{1}{d^2} \leq 2\sum_{d=1}^{\infty} \frac{1}{d^2} = \frac{\pi^2}{3} = O(1)$. Then the cost of access is $O(\log O(1) - \log \frac{1}{1+(x-f)^2}) = O(1 + \log(1 + (x-f)^2)) = O(\log[2 + |x - f|])$.

This theorem shows that regardless of what finger is chosen, a splay tree does as well as a BST with that finger, and runs in time logarithmic to the distance of the element from the finger. Thus, if you access some element near the finger, the amortized cost of accessing that element is cheap. This is remarkable because splay trees have no idea where the finger is.

5.1.4 Working Set Theorem

Let $t_i(x)$ be the number of distinct elements accessed since the last access of element x before time i. We set the potential of a key w_x at time i to be $\frac{1}{t_i(x)^2}$. We can show, using analysis similar to that of the static finger theorem, that $W_{tree} \leq \frac{\pi^2}{6} = O(1)$. After applying the access lemma, the cost of accessing element x is $O(1 + \log t_i(x_i)$. This shows that the cost of accessing elements within a working set is logarithmic in the size of that set and not in the size of the entire set of elements. Equivalently, it says that if you accessed something you accessed recently, the cost of accessing it is cheap.

The observant reader might note something fishy here: We are changing the potential after every access! In particular, after accessing element x, w_x increases to 1. If during the reweighting of the entire tree, the total potential increases, then we have obtained "free" potential from the reweighting and the above analysis is flawed.

However, we will show that this is not the case. Note that for all elements $y \neq x$, w_y cannot increase: It either decreases, if x was accessed for the first time since y was last accessed, and stays the same otherwise. Hence the only potential increase that can come from the reweighting is from the increase in w_x. But after each access, x goes to the root, and therefore the only subtree where the potential increases is the root, and the only W_i value affected is W_{tree}. But we already know that W_{tree} is upper bounded, and we have used this upper bound, so the potential can never increase after re-weighting. Therefore, this potential change is justified and our analysis holds.

5.2 Other Results

The results here are cited without proof.

5.2.1 Scanning Theorem

The Scanning Theorem, demonstrated by Tarjan [**?**], states the following:

Theorem 4. *If elements of a splay tree are accessed sequentially in order, then the total running time is linear, regardless of the initial structure of the splay tree.*

5.2.2 Dynamic Finger Theorem

The Dynamic Finger Theorem (formerly conjecture) which was first conjectured in [**?**] and finally proven by Cole et al.[**?**], [**?**] after a lengthy analysis, states:

Theorem 5. *The amortized cost of accessing the i^{th} element x_i is $O(\log[1 + |x_i - x_{i-1}|])$.*

The intuition behind this theorem is the static finger theorem: suppose instead of having a fixed finger, the finger could move after every access (we have to pay a cost for moving the finger around). In this case, it is always optimal to move the finger to the next element accessed: since we have to touch the next element anyway, why not take the fastest route to it, instead of returning to the original finger (or root) position? This theorem therefore shows that splay trees run within a constant factor of this optimal bound.

5.3 Conjectures

In this section, we show results that are conjectured to be true about splay trees but have not been proved.

5.3.1 Dynamic Optimality Conjecture

The Dynamic Optimality conjecture is one of the biggest open problems in the field of self-adjusting data structures. It was first stated in [**?**]:

Conjecture 6. *Consider an optimal offline algorithm OPT that is allowed to modify the tree via rotations between accesses. The cost of access is the same i.e.* 1 *plus the depth of the accessed node. In addition, each rotation costs $O(1)$. The conjecture is that the run time of splay trees is within a constant factor of OPT for any access sequence.*

If the dynamic optimality conjecture is correct, then splay trees are $O(1)$-competitive even in a general dynamic model. This is a very powerful result: splay trees are as good as any other tree on any given request sequence, even if the other tree is specially designed for the given request sequence. We note as an aside that splay trees can be easily shown to be at least $O(\log n)$-competitive dynamically.

It is an open question as to whether there exists *any* BST (not just splay trees) that satisfy the dynamic optimality conjecture. In the next few lectures, we will see recent results on this problem that have reduced the competitive gap from $O(\log n)$ to $O(\log \log n)$ [**?**].

5.3.2 The Unified Conjecture

In the lecture, we have shown several properties of splay trees, but how do they all compare to each other? We note the following:

- The Working set theorem implies the static optimality theorem, which in turn implies the static finger theorem.

- The Dynamic finger theorem implies the static finger theorem.

In addition, Iacono [?] has proposed the following *unified conjecture*:

Conjecture 7. *The amortized cost to access element x_i is upper bounded by:*

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

Intuitively, this suggests that element x_i is cheap to access if there exists some different element y that is both spatially (in terms of the structure of the tree) and temporally (in terms of access time) close to it. The unified conjecture, if true, would imply both the working set and dynamic finger theorems. It is not known whether the unified conjecture holds for a BST, but Badoiu and Demaine [?] have shown that it can be done on a pointer machine.

5.3.3 Traversal conjecture

Conjecture 8. *Supposing we have two splay trees, each with the same elements. Take a preorder traversal of one of the splay trees to obtain an access sequence. The conjecture is that the cost of accessing elements in this access sequence in order in the other splay tree is $O(n)$ [?].*

5.3.4 Deque conjecture

Supposing now we have two fingers, and we can choose to start our search pointer from either of them during an access. After each access, we are allowed to move either finger by one element, thus simulating a double-ended queue (deque). We now have the following conjecture:

Conjecture 9. *The cost of m accesses in splay tree of n elements is $O(m + n)$ [?].*

As an aside, it is known that the cost is at most $O((m + n)\alpha(m + n))$ [?].

5.3.5 Removing elements from the access sequence

Finally, an unpublished but simple conjecture states:

Conjecture 10. *Consider a sequence of accesses on a splay tree. If any of these accesses is removed, the resulting sequence of accesses is cheaper than the original sequence of accesses.*

This conjecture seems obvious, but has so far remained resistant to attacks.

References

[1] B. Allen and J.I. Munro, *Self-organizing binary search trees*, Journal of the ACM, 25, 546-535, 1978.

[2] M. Badoiu and E. Demaine, *A Simplified, Dynamic Unified Structure*, Latin America Theoretical Informatics, 466-473, 2004.

[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] R. Cole, B. Mishra, J. Schmidt, and A. Siegel, *On the dynamic finger conjecture for splay trees. Part I: Splay sorting* $\log n$*-block sequences*, SIAM Journal of Computing, 30(1), 1-43, 2000.

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

[6] J. Iacono, *Alternatives to Splay Trees with* $o(\log n)$ *worst case access times*, Symposium on Discrete Algorithms, 516-522, 2001.

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

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

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

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