Lecture 7 — March 5, 2003

Prof. Erik Demaine *Scribe: Siddhartha Sen*

1 Overview

In the last lecture, we analyzed the Move-To-Front heuristic for searching in a self-organizing linked-list. In particular, we showed that Move-To-Front is both statically and dynamically optimal (as good as any dynamic list) up to a factor of 2, provided we work under Sleator & Tarjan's cost model. We also introduced an omniscient algorithm called Order-By-Next-Request [Mun00], and proved that it is exponentially faster than Move-To-Front or any other online algorithm for a particular request sequence (listing the elements in order).

In this lecture, we will continue our discussion of self-organizing lists by showing that Munro's Order-By-Next-Request strategy is not only better than all online algorithms, but amortized it achieves the "entropy" bound, which is the best possible performance for any comparison-based search structure in the stochastic model. Then, we will move on to a new topic: trees. First, we will discuss optimal binary search trees; then, we will cover self-organizing trees and splay trees, explain how they work, and describe some of the bounds known about them.

2 Self-Organizing Lists (continued)

We finish our discussion of Munro's Order-By-Next-Request (ONR) strategy [Mun00] by showing that the amortized cost of the algorithm achieves the entropy bound. Recall that ONR is an omniscient algorithm, in that we know for each element the time that it will be accessed. So, upon request to an element at position i, we do the following:

1. Continue scan to position $\lceil\lceil i\rceil\rceil$, where $\lceil\lceil i\rceil\rceil = 2^{\lceil \lg i\rceil}$ is the hyperceiling of i.
2. Reorder (sort) these elements according to when they will next be requested.

The total cost for a permutation of the elements was $\Theta(n \lg n)$. By contrast, Move-To-Front (or any online strategy) can cost $\Theta(n^2)$ (with the startup model, or with a nasty permutation of the list matching its initial order). Per element, the costs are $\Theta(\lg n)$ versus $\Theta(n)$, which is an exponential discrepancy.

While ONR may seem "unfair" because it requires "knowing the future" (i.e., knowing when each element will be accessed), it is important to realize that this assumption is sometimes realistic. For example, when generating a minimum spanning tree, it is possible to predict the order of the nodes that are going to be visited—so in this sense we "know" the future.

2.1 Amortized Cost of ONR

Theorem 1. *The amortized cost of ONR to access an element is* $\leq 1 + 4\lceil \lg r \rceil$, *where* $r =$ *number of distinct elements accessed since this element was last accessed (including the element itself).*

Corollary 2. *The total cost of ONR is* $\leq m + 4\sum_{i=1}^{n} \lceil \lg \frac{m}{f_i} \rceil$, *where* $f_i =$ *number of occurences of* i. *Let* $p_i = \frac{f_i}{m}$, *the probability of accessing element* i. *Then the following statements are true:*

1. *The amortized cost for element* i *is* $O(\lg \frac{1}{p_i})$.
2. *The "expected" cost for a random element is* $O(\sum_{i=1}^{n} p_i \lg \frac{1}{p_i})$.
3. *The total cost is* $O(m \sum_{i=1}^{n} p_i \lg \frac{1}{p_i})$.

The bound in statement 2 above represents the *entropy* of the probability distribution characterized by the p_i. This is promising because it shows that the expected cost of ONR for a random element approaches the information-theoretic lower bound.

Now let's prove the above theorem.

Proof: We begin with two failed attempts (at calculating the cost of ONR) before demonstrating the correct approach.

Proposition 1: *Charge the elements in the last block (i.e. the largest block).*

This is a bad idea because most of these elements might never be requested.

Proposition 2: *Charge the elements in the front (i.e. the small blocks, for some notion of small).*

Unfortunately, this is still a bad idea because there aren't enough elements in these blocks; most of the elements (a constant fraction) are actually in the two largest blocks.

Proposition 3: *Soak the middle class: charge a cost of* $\lceil\lceil i \rceil\rceil$ *to the penultimate block* b *of size* $\frac{\lceil\lceil i \rceil\rceil + 1}{4}$ *(unless the request is for the first element, in which case we charge the first element).*

Claim 3. *An element* i *is charged at most once in any block, between two requests for* i.

Suppose i is in block b. There are two cases to consider:

Case 1: i gets charged in block b and either stays in b or moves forward. This is shown by the solid arrow in Figure 1. Elements from the new position of i to the end of block $b+1$ serve as a buffer because we know they will remain in the same relative order until i is accessed (i.e. they will only be accessed after i is accessed first). In this case, i can't be charged unless it moved to block $b+1$ and the resulting buffer did not extend to fill block $b+2$ (and then an element in block $b+2$ was requested).

Case 2: i moves to block $b+1$. This is shown by the dashed arrow in Figure 1. In this case, i may get charged by a request in block $b+2$, but again this will only happen once between two requests for i (see Claim 3).

Now we ask the following question: how many blocks can i possibly be in? We know that i is charged at most once per block. Let r be the number of distinct elements accessed between two

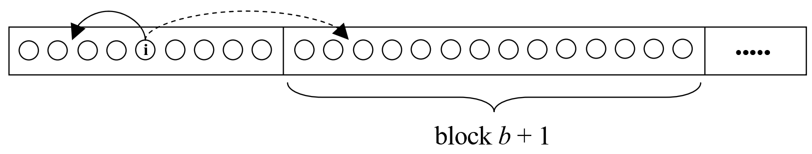

Figure 1: Possible movement of element i after a request for i. Case 1 is illustrated by the solid arrow; Case 2 is illustrated by the dashed arrow.

accesses to i. Then the farthest right i can possibly move is position r. This leads us to the following two conclusions:

1. i can be in at most $\lceil \lg r \rceil$ blocks, so there will be $\leq \lceil \lg r \rceil$ charges attributed to i.
2. The amortized cost to access element i will therefore be $\leq 4\lceil \lg r \rceil + 1$, where the additional 1 represents the base charge of accesses to the front of the list.

This ends our proof of Theorem 1.

3 Trees

We now move on to a new topic: trees. We will start by discussing optimal binary search trees in the Stochastic model. Then we will move on to self-organizing trees and splay trees, explain how they work, and describe some theorems and bounds known about them.

3.1 Binary Search Trees (BSTs)

We can ask all the same questions with Binary Search Trees (BSTs) as we did with linear search data structures. For BSTs, it is easy to achieve a worst case search time of $\Theta(\lg n)$ for both successful and unsuccessful searches. The cost model here counts the number of nodes inspected, starting at the root. We now discuss both static and dynamic optimality of BSTs in the Stochastic model, as we did for lists. We will find that dynamic optimality is much harder to achieve with trees than it is for lists.

3.1.1 Stochastic Model: Optimal BST [GM59]

First, we define some probability terms for optimal BSTs in the Stochastic model (see [CLRS01]). Let p_i = probability of searching for element i in a BST with n elements, where $1 \leq i \leq n$. Let q_i = probability of searching between element i and element $i+1$, where $0 \leq i \leq n$. Then, the probabilities associated with element accesses in the BST can be written out as a sequence: $q_0, p_1, q_1, p_2, q_2, \ldots, q_{m-1}, p_m, q_m$.

Recall that the cost of a search in the BST is equivalent to the number of nodes traversed. Define $\texttt{root}[i, j]$ to be the root of the optimal BST on $q_{i-1}, p_i, q_i, \ldots, p_j, q_j$. Now let $\texttt{cost}[i, j]$ = cost of the tree rooted at `root`$[i,j]$ $\times$ probability of entering this tree. How do we compute the entries of the cost function? We can use a method called *dynamic programming*, which takes a divide-and-conquer approach to solve problems by remembering and re-using solutions to smaller subproblems (see [CLRS01] for a more thorough discussion of dynamic programming). We define the base case and recursive step of this dynamic program as follows:

1. *Base Case*:
   (a) `root`$[i,i] = i$ (this is a tree on one element)
   (b) `cost`$[i,i] = q_{i-1} + p_i + q_i$ (this is the probability of entering the tree multiplied by 1, because we charge 1 upon entrance into the tree)
   (c) `cost`$[i,i-1] = 0$ (this is a special zero-probability case)
2. *Recursive Step*:
   (a) $\texttt{cost}[i,j] = \min_{i \le r \le j}(\underbrace{q_{i-1} + p_i + q_i + \ldots + p_j + q_j}_{\text{cost to access root}} + \underbrace{\texttt{cost}[i,r-1]}_{\text{left subtree}} + \underbrace{\texttt{cost}[r+1,j]}_{\text{right subtree}})$
   (b) `root`$[i,j]$ = min argument (r) of the `cost` function above

There are a total of $O(n^2)$ subproblems (entries to compute in the `cost` function), each of which takes $O(n)$ time to complete—giving us a total time of $O(n^3)$. We can speed this up if we change the range of the min function (in the recursive step above) to use the following range instead:

$$\texttt{root}[i,j-1] \le \texttt{root}[i,j] \le \texttt{root}[i+1,j] \qquad \text{[Knu71]}$$

With this modification, the total time is reduced to $O(n^2)$. There are now n different sizes of intervals $[i,j]$ (or "levels"), and each interval is covered at most twice (see Figure 2). Because there are $O(n)$ options at each level (i.e. for a given interval size k), the total time becomes $O(n^2)$.

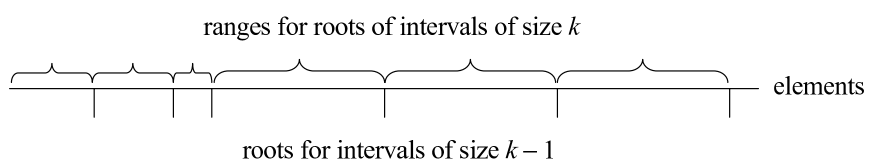

Figure 2: Interval ranges for modified dynamic programming solution.

OPEN: Is there an $o(n^2)$ time solution, or $o(n^2)$ space and polynomial-time solution, to this dynamic-programming problem? Or can it be shown that the problem is 3SUM-hard [DMO03, Problem 11], and therefore is likely to require $\Omega(n^2)$ time?

Having computed the entries of the `cost` and `root` functions, we now state the cost of a search in the optimal BST. The search cost actually approaches the entropy bound (H) of the search probabilities p_i and q_i of the optimal BST. If we define the entropy bound as

$$H = \sum_{i=1}^{n} p_i \lg \frac{1}{p_i} + \sum_{i=1}^{n} q_i \lg \frac{1}{q_i},$$

then our search cost falls within the following range:

$$H - \lg H - \lg e + 1 \leq \text{search cost} \leq H + 3$$

Because the search cost approaches the entropy bound, the BST is statically optimal.

3.2 Self-Organizing Trees (Preliminaries)

Following the format of our discussion on lists, we now consider the dynamic counterpart of static trees: self-organizing trees. We begin by describing two variants of self-organizing trees, and explain why they fail to achieve the static optimality (entropy) bound. In Section 3.3, we discuss a third variant—splay trees—that is able to achieve static optimality.

All of the self-organizing tree structures described in these notes have unknown distributions.

3.2.1 Transpose Analog [AM78]

The Transpose analog for trees basically takes a static tree and rotates a search element one level closer to the root every time it is requested. For ease of discussion, we will refer to this rotation function as `Rotate(`n`)`, where n is the element being requested. Figure 3 shows an example tree and the rotation that results from requesting the element x in that tree (or `Rotate(`x`)`).

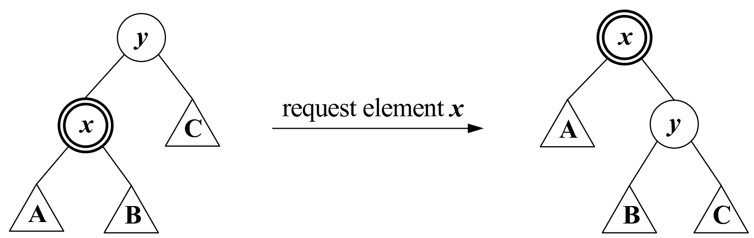

Figure 3: Element rotation upon request in a Tranpose analog tree.

Unfortunately, this type of tree is bad for uniform searches, because it turns out that all binary trees are generated (through the rotations) with the same probability [AM78]. This gives us an average search cost of $\Theta(\sqrt{n})$, which is worse than a balanced binary tree.

3.2.2 Move-To-Root

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

3.3 Splay Trees [ST85]

Splay trees are a variation on Move-To-Root (and, consequently, a variation on the Transpose analog as well). In both the Transpose and Move-To-Root trees, the rotation function only moved the element being requested towards the root, while the other elements were generally moved downwards in the tree. In splay trees, however, the rotation function takes a different approach: instead of considering the search element's immediate parent, it looks one level higher and considers its grandparent. There are two cases that arise from this technique:

1. *Zig-Zig case*: This is shown in Figure 4a). The element being requested is x; the resulting rotation is analogous to `Rotate(`z`)` followed by `Rotate(`y`)`.
2. *Zig-Zag case*: This is shown in Figure 4b). The element being requested is y; the resulting rotation is analogous to `Rotate(`x`)` followed by `Rotate(`z`)`.

By repeatedly applying either the Zig-Zig or Zig-Zag rotations, the element being requested will eventually end up in the root or as a son of the root (in which case one final rotation will be necessary to move it to the root). The sequence of rotations applied to the search element is referred to as the *splay operation* on that element.

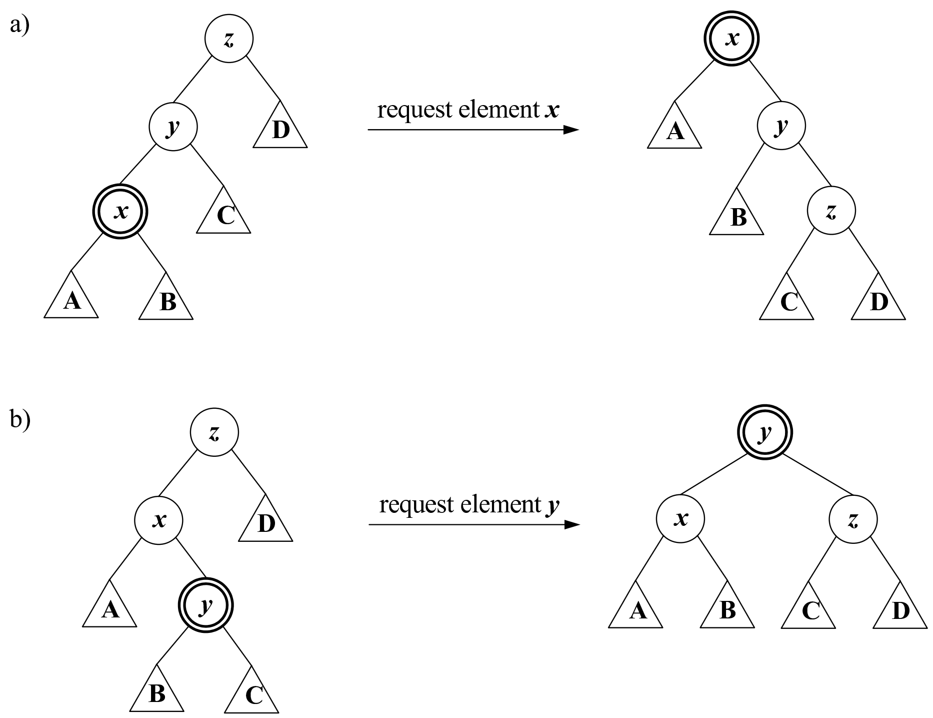

Figure 4: Element rotation upon request in a splay tree: a) a zig-zig rotation caused by a request to element x; b) a zig-zag rotation caused by a request to element y.

The rotation scheme for splay trees yields a much better performance than the scheme for Move-To-Root. The amortized cost per operation—for search as well as insert and delete—is $O(\lg n)$. Furthermore, this cost approaches the entropy bound, making splay trees statically optimal.

Can we do better than static optimality? Currently, there is a big conjecture that splay trees are also dynamically optimal. In the subsections that follow, we outline a brief analysis of splay trees and list some of the consequences that arise from this analysis, including the dynamic optimality conjecture (which has not been proven to date).

3.3.1 Analysis of Splay Trees

We present a brief sketch of the analysis for splay trees, in what is called the "access theorem".

Suppose we assign arbitrary weights $w(i)$ to each element i in the splay tree. We define the size of a node x, or $s(x)$, to be the total weight of the nodes in the subtree rooted at x:

$$s(x) = \sum \{w(i) \mid i \text{ is in subtree rooted at } x\}$$

Similarly, we define the rank of node x, or $r(x)$, to be the following:

$$r(x) = \lg s(x)$$

The rank of x represents the "ideal depth" of the subtree rooted at x. Now we state (without full proof) the Access Theorem, which was a significant result in [ST85]:

Theorem 4. *If a splay tree currently has root t, then the amortized cost to splay a given node x is at most*

$$3(r(t) - r(x) + 1) = O\left(\lg \frac{s(t)}{s(x)}\right) \qquad \text{[ST85]}$$

Proof idea: The proof of this theorem is similar to the Move-To-Front proof from the previous lecture. It uses the potential function $\Phi = \sum\{r(x) \mid x \text{ is a node}\}$ and assumes that all elements are accessed. Basically, the proof shows that the amortized cost of a double rotation in the splay tree is $\leq 3(r(t) - r(x))$ and the amortized cost of a single rotation is $\leq 3(r(t) - r(x)) + 1$. Then, it shows that a sequence of rotations yields a telescoping sum that results in the bound stated above.

3.3.2 Consequences of Splay Tree Analysis

There are several results that follow directly from our splay tree analysis above. We list them here without full discussion.

- $\lg n$ *Bound*: If we assign a weight of 1 to all nodes in the tree—that is, $w(i) = 1$ for all i—then it follows that $s(t) = n$. From the Access Theorem, the amortized cost to splay a node in this tree is $O(\lg \frac{n}{k})$, where $1 \leq k \leq n$; hence we have our $O(\lg n)$ amortized bound.
- *Static Optimality*: If we let $w(i) = p_i = \frac{f_i}{m}$, i.e. the probability of accessing node i in the tree, then it follows that $s(t) = 1$. Because $s(x) \geq w(x)$, an access to node x will cost $O(\lg \frac{1}{k})$, where $w(x) \leq k \leq 1$, or equivalently, $O(\lg \frac{1}{p_x})$ (again from the Access Theorem). This gives

us a total cost of $O(m \sum_{i=1}^{n} p_i \lg \frac{1}{p_i}) = O(mH)$, where H is the entropy bound; hence we achieve static optimality.

- *Static Finger Theorem*: The Static Finger Theorem can be stated as follows:

  Theorem 5. *Let f be a specific node in the tree called the "finger" (i.e., fix your finger on the node f). Then the following is a bound on the amortized cost of accessing a node i:*

  $$O(\lg(1 + |i - f|))$$

  where $|i - f|$ is the distance in the total order of the keys stored in the finger f and the node i.

  Proof: Assign the weights $w(i) = \frac{1}{(1+i-f)^2}$ to each node i. Then, it follows that $s(t) \leq \sum_{k=-\infty}^{+\infty} \frac{1}{k^2} = \frac{\pi^2}{3}$; that is, the sum of all the weights is bounded by a constant. From the Access Theorem, we have that the cost to splay node i is therefore $O(\lg \frac{O(1)}{w(i)}) = O(\lg(1 + i - f)^2) = O(\lg(1 + |i - f|))$; hence the bound above.

3.3.3 More Properties of Splay Trees

We state (without proof) a few more theorems relevant to our discussion of splay trees. All of these results are nontrivial; the last one is extremely difficult.

Theorem 6. *(Working Set Theorem) The amortized cost of accessing an element in a splay tree is* $O(\lg(\text{number of distinct accesses since the last access}))$*. (For the first access to an element, we charge $O(\lg n)$.) In particular, if only k elements are used, the amortized cost per element is $O(\lg k)$ (after startup).*

Theorem 7. *(Scanning Theorem) Splaying all elements in a tree one-by-one in order costs $O(n)$, where n is the total number of elements, starting from any tree configuration* [Tar85].

Theorem 8. *(Dynamic Finger Theorem) If element x of a splay tree was just accessed, then the following is a bound on the amortized cost of accessing element y:*

$$O(\lg(1 + |x - y|)) \qquad \text{[CMSS00, Col00]}$$

3.3.4 The Unified Property and the Dynamic Optimality Conjecture

The consequences and properties described in Sections 3.3.2 and 3.3.3 have various implications, in the sense that any structure possessing one property necessarily possesses another property. See Figure 5 for a summary of all such implications. In addition, there is a "unified" property we will see next lecture that implies both the working-set property and dynamic-finger property [Iac01].

There are still several questions about splay trees that remain unanswered. By far the most famous of these questions is the Dynamic Optimality conjecture, which is actually a generalization of the Scanning Theorem and Dynamic Finger Theorem above. The conjecture can be stated as follows:

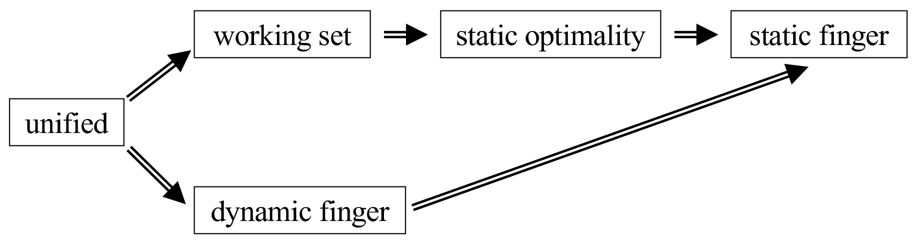

Figure 5: Summary of splay tree consequences and properties ($\Rightarrow$ indicates a logical implies relationship).

Conjecture 9. *(Dynamic Optimality Conjecture) Let T_0 be a binary search tree with n nodes. Consider a class of algorithms that access nodes in the tree by initiating a search at the root; in other words, the search cost is 1 plus the depth of the node. After an access, arbitrary rotations are also allowed, at a cost of 1 per rotation. For a given sequence σ, let $A(T_0, \sigma)$ be the cost of the optimal algorithm in this class for processing the sequence σ starting from tree T_0. Then the cost of splaying this sequence (from any starting tree configuration) is $O(A(T_0, \sigma) + n)$.*

If the above conjecture is true, then it follows that splay trees are the best self-organizing tree structure possible (to a constant factor). Moreover, it means that splay trees actually perform as well as *any* search tree on *any* given request sequence (even if the search tree is optimized for the given sequence).

Only restricted cases of the Dynamic Optimality Conjecture have been proven so far, e.g., the Scanning Theorem (Theorem 7 above). We'll see several attacks on this conjecture in the next lecture.

References

[AM78] B. Allen and J. I. Munro. Self-organizing binary search trees. *Journal of the ACM (JACM*, 25(4):526–535, 1978.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[CMSS00] 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 on Computing*, 30(1):1–43, 2000.

[Col00] R. Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[DMO03] E. D. Demaine, J. S. B. Mitchell, and J. O'Rourke. *The Open Problems Project*, 2003. Available online: http://cs.smith.edu/˜orourke/TOPP/.

[GM59] E. N. Gilbert and E. F. Moore. Variable-length binary encoding. *Bell System Tech. Journal*, 38:933–968, 1959.

[Iac01] J. Iacono. Alternatives to splay trees with $o(\log n)$ worst-case access times. In *Symposium on Discrete Algorithms*, pages 516–522, 2001.

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[Mun00] J. I. Munro. On the competitiveness of linear search. In *Proceedings of the 8th Annual European Symposium on Algorithms*, volume 1879 of *Lecture Notes in Computer Science*, pages 338–345, Saarbrücken, Germany, September 2000.

[ST85] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, July 1985.

[Tar85] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.