ICS 691: Advanced Data Structures Spring 2016

Lecture 3

Prof. Nodari Sitchinava *Scribe: Ben Karsin*

1 Overview

In the last lecture we started looking at self-adjusting data structures, specifically, MOVE-TO-FRONT querying of un-sorted lists, as introduced in [ST85a]. Using amortized analysis, we proved that using MOVE-TO-FRONT is 4-competitive with the optimal solution.

In this lecture we focus the discussion on SPLAY TREES, as detailed in [ST85b]. In Section 2, we discuss Binary Search Trees in the context of optimal layout for performing m queries. In Section 3 we introduce the SPLAY tree and illustrate the 3 operations: ZIG, ZIG-ZAG, and ZIG-ZIG. We perform amortized analysis on these 3 operations in Section 3.3 and show that it is c-competitive with the optimal solution.

2 Binary Search Trees

Balanced BINARY SEARCH TREES (BST) implement dictionary APIs in $O(\log N)$ time for insert/delete/search. If access frequencies, $f(x_i)$ are known for all m queries (i.e., item x_i is accessed $f(x_i)$ times, we can consider the optimal BST. The method of constructing this optimal off-line BST in $O(n^3)$ is given in chapter 15 of [CLRS09]. The work to create this optimal BST can be further reduced to $O(n^2)$, as discussed in exercise 15-3 of [CLRS09].

If we define $p(x) = \frac{f(x)}{m}$, we can consider the cost of m queries, C_m as:

$$\begin{aligned} C_m &= \sum_{i=1}^{m} f(x) \cdot \log \frac{1}{p(x)} \\ &= \sum_{i=1}^{m} p(x) \cdot m \cdot \log \frac{1}{p(x)} \\ &= m \cdot \sum_{i=1}^{n} p(x) \cdot \log \frac{1}{p(x)} \\ &= m \cdot H_n \end{aligned}$$

Where $H_n = \sum_{i=1}^{n} p(x) \log \frac{1}{p(x)}$ is the entropy of the query set. Hence, if all queries are equally likely, $H_n = \log n$ and the total cost is $m \cdot \log(n)$.

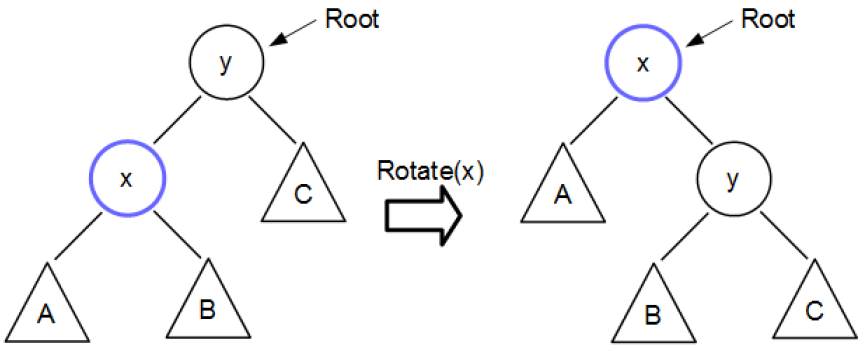

Figure 1: A case where the ZIG operation is used to move target node x to the root. This is only used when the target node is the direct child of the root node.

3 Self-adjusting BSTs: Splay Trees

SPLAY TREES, detailed in [ST85b] are self-adjusting BSTs that:

- implement dictionary APIs, like the MOVE-TO-FRONT list,
- are simply structured with limited overhead,
- are c-competitive with best offline BST, i.e., when all queries are known in advance,
- are conjectured to be c-competitive with the best online BST, i.e., when future queries are not know in advance

3.1 Splay Tree Operations

Simply, SPLAY TREES are BSTs that move search targets to the root of the tree. This is accomplished by 3 simple rotate rules that are applied repeatedly until the search target reaches root. The 3 rotate rules are ZIG, ZIG-ZAG, and ZIG-ZIG, and are detailed below:

3.1.1 Zig

The ZIG operation is applied if the element being moved, x, is the child of the root, as seen in Figure 1. When this is the case, the element, x, is simply rotated with the root node, making it the new root.

3.1.2 Zig-zag

The ZIG-ZAG operation is applied when the target element being moved, x, needs to be moved up in alternate directions, as illustrated in Figure 2. When a node being moved up the tree falls on the opposite direction of its parent as its parent falls from its grandparent, we employ the ZIG-ZAG

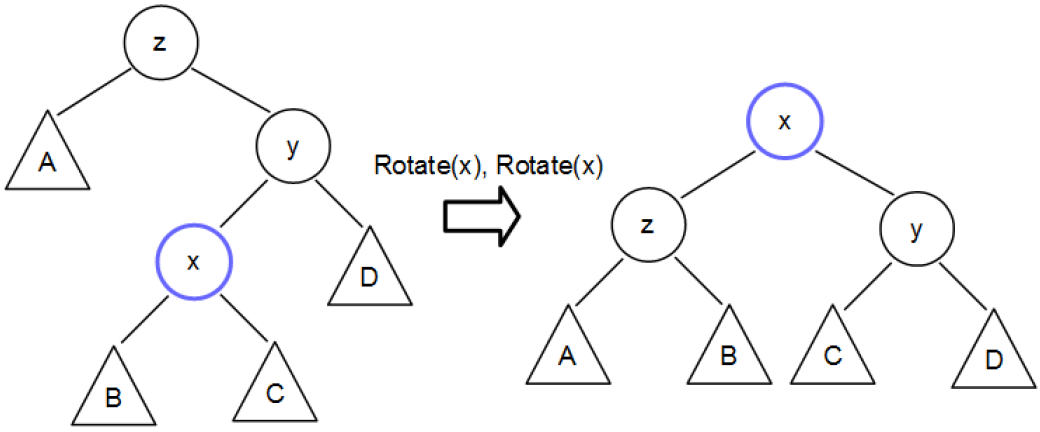

Figure 2: A case where the ZIG-ZAG operation is employed to move target node x up the tree. The ZIG-ZAG operation would also be used in the symmetric case (i.e., where x is the right child of y and y is the left child of z).

operation. The ZIG-ZAG operation involves rotating x with its parent (y), and then rotating again with its new parent (z).

3.1.3 Zig-zig

The final operation, ZIG-ZIG, is used when both the node being moved up falls on the *same* side of its parent as its parent does (from its grandparent). When this occurs, we first perform *Rotate* on the targets parent node, followed by *Rotate* on the target node itself. Figure 3 illustrates how the ZIG-ZIG operation is performed on an example tree.

3.2 Splay Tree Example

We consider the worst-case initial BST and look at how the splay tree and its operations would perform. Clearly, our worst-case BST would be a tree where each node has only 1 child, and the other is NULL. In this case, searching the leaf node would result in an $O(n)$ time search. However, using a splay tree, each search would result in the SPLAY operation moving the searched (leaf) node to the root, thereby moving lots of other elements. Figure 4 illustrates how a splay tree would perform when repeatedly searching such a target.

We see in Figure 4 that, after just 3 searches on a worst-case tree, we have a tree that has optimal height. All subsequent searches will be $O(\log n)$, and would further adjust the tree as needed. This example shows how SPLAY TREES quickly self-balance and achieve optimal or near-optimal performance quickly. Amortized over a large number of queries, the cost of searching (and SPLAYING a node to the root) is $O(\log n)$.

3.3 Amortized Analysis

We use the potential method of amortized analysis and first define several variables:

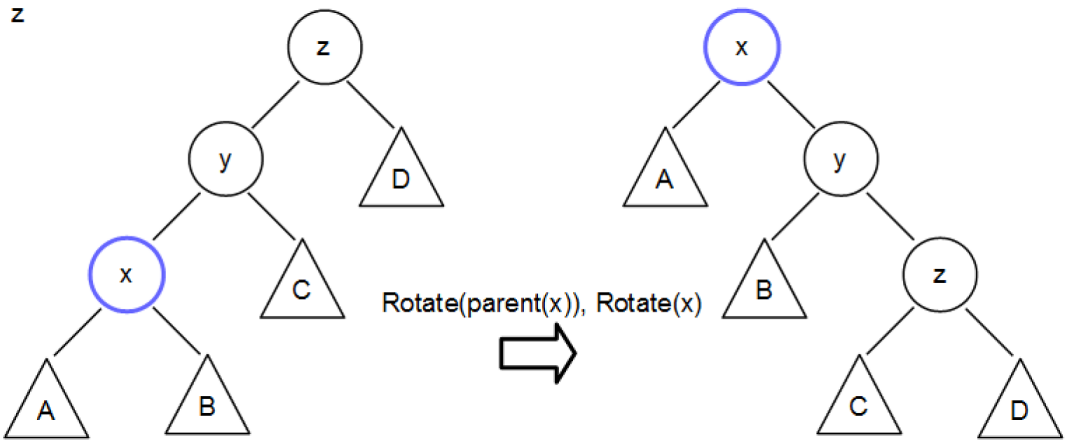

Figure 3: An example of how the ZIG-ZIG operation is performed by a SPLAY TREE. First, y is rotated upwards, then x is rotated up to the root. This is only employed when both the target (x) and its parent y fall on the same side of their respective parent nodes.

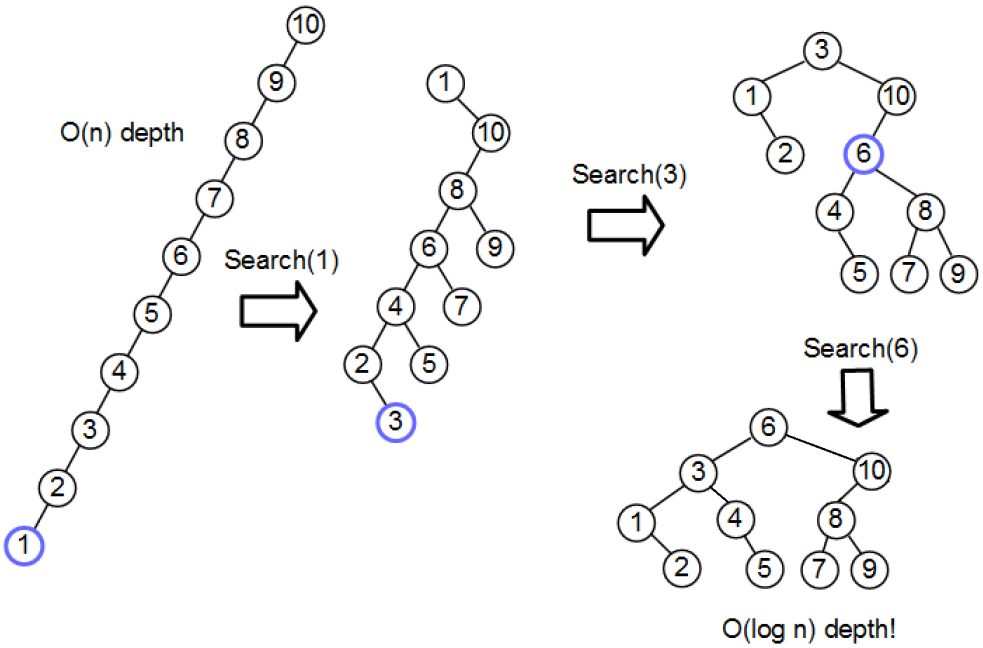

Figure 4: An example worst-case tree and result of repeatedly searching (and using the SPLAY operation) it. We see that, after searching just 3 times, we have a tree of optimal height for the input size.

Definition 1. *Let $w(x)$ be the* weight *associated with node x.*

Definition 2. *Let $s_i(x)$ be the* size *of node x after performing the ith query, defined formally as:*

$$s_i(x) = \sum_{v \in subtree\ rooted\ at\ x} w(v)$$

Definition 3. *Let $r_i(x)$ be the* rank *of x:*

$$r_i(x) = \log\left(s_i(x)\right)$$

Note that if we set $w(x) = 1$ for all x, then $s_i(x)$ is just the number of nodes in the subtree and $r_i(x)$ is the minimum levels in the subtree rooted at x.

Using these definitions, we define our potential function to perform the amortized analysis as:

$$\Phi(T_i) = \sum_{x \in T_i} r_(x),$$

where T_i is the entire splay tree structure after the ith query. To perform amortized analysis, we look at the 3 individual splay tree operations defined in Section 3.1. Recall that the amortized cost using the potential method is defined as:

$$\hat{c_i} = c_i + \Delta\Phi_i$$

Consider the ZIG-ZIG operation defined in Section 3.1 and the 2 rotations it involves. The cost of performing a rotation is 1, therefore:

$$\hat{c_i} = 2 + \Delta\Phi_i = 2 + (\Phi(T_i) - \Phi(T_{i-1}))$$

Where T_{i-1} and T_i are the potential function values before and after performing the ZIG-ZIG operation, respectively. We see in Figure 3 that the potential of subtrees A,B,C, and D don't change, so they cancel out in $\Delta\Phi_i$ and we only need to consider the 3 nodes involved in the rotations: the target node (x), its parent (y), and its grandparent (z). Therefore, we expand the above formula to the changes in potential for each of the nodes x, y, and z:

$$\hat{c_i}(x) = 2 + r_i(x) + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) - r_{i-1}(z)$$

We note, that, as seen in Figure 3:

$$\begin{aligned} r_{i-1}(z) &= r_i(x) \\ r_i(y) &\leq r_i(x) \\ r_{i-1}(y) &\geq r_{i-1}(x) \end{aligned}$$

Therefore, our amortized cost is:

$$\hat{c_i}(x) \leq 2 + r_i(x) + r_i(z) - r_{i-1}(x) - r_{i-1}(x)$$

We can further simplify this using the fact that $\frac{\log a + \log b}{2} \leq \log \frac{a+b}{2}$ and that $s_i(z) + s_{i-1}(x) \leq s_i(x)$:

$$\begin{aligned} r_i(z) &= \log s_i(z) \\ r_{i-1}(x) &= \log s_{i-1}(x) \\ \log s_i(z) + \log s_{i-1}(x) &\leq 2 \log \frac{s_i(z) + s_{i-1}(x)}{2} \\ &\leq 2(\log s_i(x) - 1) = 2(r_i(x) - 1) \end{aligned}$$

Using this simplification, our amortized cost becomes:

$$\begin{aligned} \hat{c_i}(x) &\leq 2 + r_i(x) + (2r_i(x) - 2 - r_{i-1}(x)) - r_{i-1}(x) - r_{i-1}(x) \\ &= 3(r_i(x) - r_{i-1}(x)) \end{aligned}$$

We can use a similar analysis to show that the ZIG-ZAG operation has amortized cost $\hat{c_i}(x) \leq 3(r_i(x) - r_{i-1}(x))$.

The amortized cost of the ZIG operation, described in Section 3.1:

$$\begin{aligned} \hat{c_i}(x) &= 1 + \Delta\Phi_i \\ &= 1 + r_i(x) + r_i(y) - r_{i-1}(x) - r_{i-1}(y) \end{aligned}$$

Where x is our target node being moved up and y is its parent (and the root node of the tree). We note from Figure 1 that:

$$\begin{aligned} r_i(y) &\leq r_i(x) \\ r_{i-1}(y) &= r_i(x) \end{aligned}$$

Therefore, we can simplify our amortized cost to:

$$\begin{aligned}\hat{c}_i(x) &\leq 1 + r_i(x) + r_i(x) - r_{i-1}(x) - r_i(x) \\ &= 1 + r_i(x) - r_{i-1}(x)\end{aligned}$$

And because $r_i(x) \geq r_{i-1}(x)$, it follows that $\hat{c}_i(x) \leq 1 + 3(r_i(x) - r_{i-1}(x))$

With these amortized cost values for each rotation operation, we compute the total cost to SPLAY a node x from its position to the root as:

$$\begin{aligned}\hat{c}_{splay} &= \sum_{i=1}^{k} \hat{c}_i(x) \\ &\leq \hat{c_k}(x) + \sum_{i=1}^{k-1} \hat{c}_i(x)\end{aligned}$$

Since we perform either ZIG-ZAG or ZIG-ZIG operations for all steps from $i = 1$ to $k-1$ and at most one final ZIG operation to move x to the root, our amortized cost becomes:

$$\begin{aligned}\hat{c}_{splay} &\leq 1 + 3(r_k(x) - r_{k-1}(x)) + \sum_{i=1}^{k-1} 3(r_i(x) - r_{i-1}(x)) \\ &\leq 1 + \sum_{i=1}^{k} 3(r_i(x) - r_{i-1}(x)) \\ &\leq 1 + 3(r_k(x) - r_0(x)) \\ &\leq 1 + 3(\log s_{root} - \log s_0(x)) \\ &= 1 + 3\frac{\log s_{root}}{s_0(x)} \\ &= O\left(\log \frac{s_{root}}{s_0(x)}\right)\end{aligned}$$

Theorem 4. *The amortized cost of searching a* SPLAY *tree is* $O(\log n)$

Proof. Setting $w(x) = 1$, we get $s_{root} = n$, $s_0 \leq 1$, so the amortized cost of a splay operation is bounded by:

$$\hat{c}(x) \leq O(\log \frac{n}{1}) = O(\log n)$$

□

Using other values for node weights, we can prove other theorems about splay trees:

Theorem 5. *(Static optimality) Given a sequence S of queries with known access frequences $f(x_i)$, splay trees are $O(1)$-competitive with the best offline* BST *for S.*

Proof. Let $w(x) = p(x) = \frac{f(x)}{m}$, therefore our amortized cost is:

$$\hat{c} = \sum_{x \in S} \hat{c}(x) = \sum_{x \in S} O(\log \frac{s_{root}}{s(x)})$$

Where S is the sequence of queries being performed. Note that, since s_{root} is the sum of weights on all nodes:

$$\begin{aligned} s_{root} &= \sum_{x \in T} w(x) = \sum_{x \in T} p(x) = \sum_{x \in T} \frac{f(x)}{m} \\ &= 1 \end{aligned}$$

Furthermore, since $s(x)$ includes the weight of x, $s(x) = \sum w(x) \geq p(x)$. Using these, our amortized cost can be written as:

$$\begin{aligned} \hat{c} &\leq \sum_{i=1}^{m} \log \frac{1}{p(x_i)} = \sum_{x \in S} f(x) \log \frac{1}{p(x)} \\ &= O(m \cdot \sum p(x) \cdot \log \frac{1}{p(x)}) \\ &= O(m \cdot H_n) \end{aligned}$$

Where H_n is the entropy of the sequence of queries, S on n keys. Recall from Section 2 that the cost of m queries on the optimal static BST is $C_m = m \cdot H_n$. □

References

[CLRS09] T. Cormen, C. Lieserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, third edition, 2009.

[ST85a] D.D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[ST85b] D.D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.