CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 5: Dynamic Data Structures and Binary Search Trees

Harvard SEAS - Fall 2024 *2024-09-17*

1 Announcements

- PS2 to be posted tomorrow
- PS1 due tomorrow
- SRE this Thursday
- Anurag OH after class in SEC 3.323 (11:30-12:30).
- Gabe and Katherine OH on zoom, Wednesday.
- Late policy reminder: 8 late days total. 3 can be used on a single pset.

2 Recap/Loose-Ends from Lec 4

- Defs of a static data structure problem vs. a solution to such a problem. (cf. CS51: abstract data type vs. implementation.)
- Example: StaticPredecessors, which has many applications (enabling "range select" in relational databases, NoSQL data stores, ML systems).
- Solving StaticPredecessors using Sorted Arrays.
- Arrays vs. Linked Lists.

Q: How quickly can we implement other operations when using a sorted array to store a multiset S of key-value pairs?

1. `min`(): return the smallest key K in S.
2. `rank`(K): return the *number* of pairs $(K', V') \in S$ such that $K' < K$.
3. `insert`(K, V)
4. `delete`(K, V)

3 Dynamic Data Structures

As you might have been wondering for the Interval Scheduling Problem, it is often the case that we do not get all of our input data at once, but rather it comes in through incremental updates over time. This gives rise to *dynamic* data-structure problems.

Definition 3.1. A *dynamic data structure problem* is a quintuple $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{U}, \mathcal{Q}, f)$ where:

- $\mathcal{I}$ is a (sometimes infinite) set of possible initial inputs x, and $\mathcal{O}$ is a (sometimes infinite) set of possible outputs y.
- $\mathcal{U}$ is a set of *updates*,
- $\mathcal{Q}$ is a set of *queries*, and
- for every $x \in \mathcal{I}$, $u_0, u_1, \ldots, u_{m-1} \in \mathcal{U}$, and $q \in \mathcal{Q}$, $f(x, u_0, \ldots, u_{m-1}, q) \subseteq \mathcal{O}$ is a set of *valid answers*.

Often we take $\mathcal{I} = \{\epsilon\}$, where ϵ is the empty input (e.g. a length 0 array), since the inputs can usually be constructed through a sequence of updates. For example:

Updates :

- `insert`(K, V) for $K \in \mathbb{R}$: add one copy of (K, V) to the multiset S of key-value pairs being stored. (S is initially empty.)
- `delete`(K) for $K \in \mathbb{R}$: delete one key-value pair of the form (K, V) from the multiset S (if there are any remaining).

Queries :

- `search`(K) for $K \in \mathbb{R}$: output $(K', V') \in S$ such that $K' = K$.
- `next-smaller`(K) for $K \in \mathbb{R}$: output $(K', V') \in S$ such that $K' = \max\{K'' : \exists V''\ (K'', V'') \in S,\ K'' < K\}$.

Data-Structure Problem Dynamic Predecessors

A *multiset* is like a set but can contain more than one copy of an element. The multiset S appearing in the above definition is only used to define the functionality of the data structure, namely how queries should be answered. How this set is actually maintained is up to the particular solution.

To solve a dynamic data structure problem, we need to now come up with algorithms that also implement the updates.

The definition of what it means to implement a dynamic data structure with algorithms gets a bit cumbersome (and we don't expect you to remember it), but we include it here in the notes for completeness:

Definition 3.2. For a dynamic data structure problem $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, \mathcal{U}, f)$, an *solution* is a triple of algorithms (Preprocess, EvalQ, EvalU) such that for all $x \in \mathcal{I}$, $u_0, u_1, \ldots, u_{m-1} \in \mathcal{U}$ and $q \in \mathcal{Q}$,

if $f(x, u_0, u_1, \ldots, u_{m-1}, q) \neq \emptyset$, then $y_m \in f(x, u_0, u_1, \ldots, u_{m-1}, q)$, where $y_0, \ldots, y_m$ are defined inductively as follows:

- $y_0 = \text{Preprocess}(x)$, and
- $y_i = \text{EvalU}(y_{i-1}, u_i)$ for $i = 1, \ldots, m$.

If $f(x, u_0, u_1, \ldots, u_{m-1}, q) = \emptyset$, then we require that $y_m = \perp$.

Now, we would like both EvalU and EvalQ to be extremely fast. Dynamic data structures can also be conveniently implemented using `class`es in Python, similarly to as shown in Lecture 4, with the update operations also implemented as methods.

4 Binary Search Trees

We can solve the Dynamic Predecessors problem by maintaining a sorted array, which achieves $O(n)$ runtime in insertions and deletions. Table 1 collects the list of these runtimes. While this is a reasonable runtime, we would ideally like to achieve updates and queries in a runtime of $O(\log n)$. The data structure that achieves this desired runtime is a binary search tree.

Data structure runtimes	
Queries or Updates	Sorted Array
`search`	$O(\log n)$
`next-smaller`	$O(\log n)$
`next-larger`	$O(\log n)$
`min`	$O(1)$
`max`	$O(1)$
`rank`	$O(\log n)$
`insert`	$O(n)$
`delete`	$O(n)$

Table 1: List of query and update runtimes for sorted arrays.

4.1 Binary Search Tree Definition and Intro

Definition 4.1. A *binary search tree (BST)* is a recursive data structure. The *empty BST* has no vertices. Every nonempty BST has finitely many vertices, one of which is the root vertex r, and every vertex v has:

- a key K_v
- a value V_v
- a left subtree, which is either the empty BST or is rooted at a vertex denoted v_L, which we call the *left-child* of v.
- a right subtree, which is either the empty BST or is rooted at a vertex denoted v_R, which we call the *right-child* of v.

We require that the sets of vertices contained in the subtrees rooted at v_L and v_R are disjoint from each other and don't contain v. Furthemore, we require that every non-root vertex has exactly one parent, and the root vertex has no parents.

Crucially, we also require that the keys satisfy:

> *The BST Property: If v has a left-child v_L, then the keys of v_L and all its descendants are no larger than K_v, and similarly, if v has a right-child, then the keys of v_R and all of its descendants are no smaller than K_v.*

The *multiset* S stored by a BST T is the multiset of key-value pairs occurring at vertices of T.

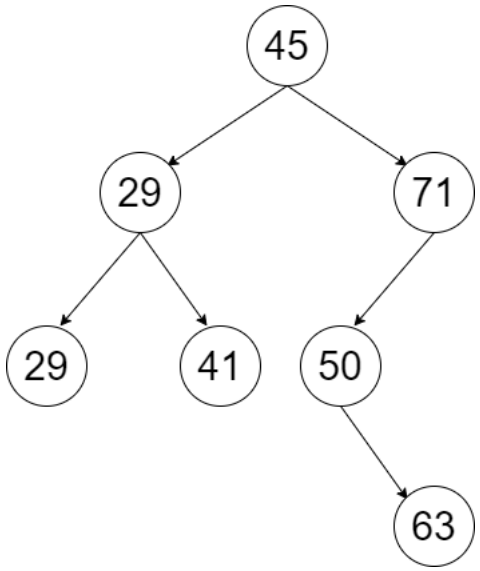


Figure 1: Example of a binary search tree.

Remarks:

- **A BST and its root:** We will blur the distinction between a BST T and its root. Thus T_L (respectively, T_R) will denote the BST rooted at the left-child (resp., right-child) of T.
- **Pseudocode notations:** In Python and some of our pseudocode, we will write the empty BST as `None`. We will also use distinct fonts in the pseudocode (as well as proof of correctness) for left subtree (`T.left` instead of T_L), right subtree (`T.right` instead of T_R) etc.
- **Definitions in other texts:** The description of binary search trees in the Roughgarden and the CLRS texts differs slightly from ours. First, instead of considering the data associated with a key to be a value stored at the same vertex, they treat the vertex itself as the data associated with a key. So, for example, to replace one data element with another, in our formulation, we could just rewrite the key and value attributes of the corresponding vertex, whereas the texts would require removing the vertex from the tree and rewiring pointers to the new vertex.

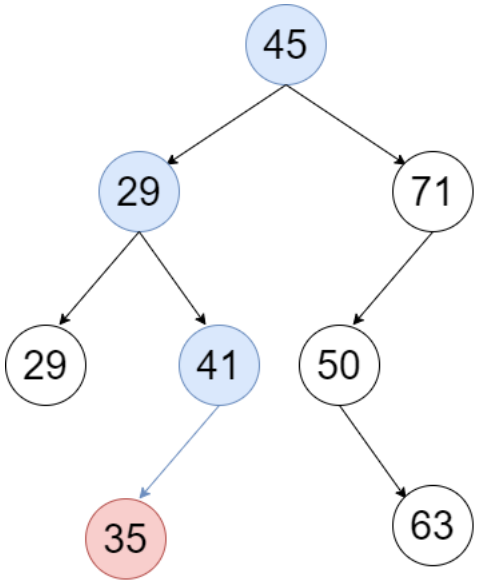
- **Parent pointers:**

  Relatedly, the texts (and our pset 0) also include a parent pointer at every vertex. It's not necessary for the current lecture, but it can be useful in implementations, as you may have discovered in pset 0 (so feel free to include it if you prefer).

- **Higher fanout:**

  It is also worth noting that in practice, trees with fanout (the number of children, aka maximum outdegree) greater than 2 are often used for greater efficiency in terms of memory accesses. Specifically, in large data systems, the fanout is often chosen so that the size of each vertex is a multiple of the size of a "page" in memory or on disk.

Our original motivation to introduce the binary search tree data-structure was to reduce the cost of insertions and deletions in comparison to a sorted array. Lets see this with the example in Figure 1 (the 'values' attributes are chosen to be empty). If we need to insert 35, then the vertices that we compare 35 to appear colored in the following image:



The number of such vertices is the length of the path from the root to the leaf vertex that holds 45.

This motivates the definition of the tree's height, which captures the worst-case cost of insertion.

Definition 4.2 (Height of a tree)**.** The *height* h of a BST is the length of the longest path from the root vertex to any leaf node. The height of an empty tree is defined to be -1.

The reason it makes sense to define the height of an empty tree to be -1 is so that the height of a tree is always one plus the maximum of the heights of its left-subtree and and right-subtree.

Theorem 4.3. *Given a binary search tree of height (equivalently, depth) h, all of the dynamic predecessors+successors operations (queries and updates), as well as min and max queries, can be performed in time $O(h)$. That is, the queries are correctly answered according to the multiset stored by the tree, and the updates correctly modify the multiset stored by the tree while maintaining all the requirements of a BST (in particular, the BST Property).*

Since $O(h)$ only refers to sufficiently large h, note that we spend $O(1)$ time even when $h = -1$ or $h = 0$, even though $c \cdot h \leq 0$ in these cases.

Proof.

- **Insertions:**

```
1  Insert(T,(K,V))
2  if T = None then
3  |   Let T = new BST with key K, value V, no children
4  |   return T
5  Let v be the root of T.
6  if K ≤ K_v then
7  |   T.left = Insert(T.left,(K, V))
8  |   return T
9  else
10 |   T.right = Insert(T.right,(K, V))
11 |   return T
```

Runtime: The runtime of this operation is $O(h)$ for the same reason as in Pset 0. More formally, let $T_{\texttt{insert}}(h)$ be the runtime maximum for trees of height at most h. The Lines 2-5 in the algorithm cost $O(1)$ runtime. Only one of Lines 6 or 9 succeeds, which recurses the algorithm on a tree of height at most $h - 1$ and then adds $O(1)$ runtime to it. Then for $h > 0$, we have

$$T_{\texttt{insert}}(h) = T_{\texttt{insert}}(h-1) + O(1),$$

which implies

$$T_{\texttt{insert}}(h) = O(h).$$

Proof of Correctness:

Our proof of correctness uses induction on the height of the tree. Specifically, we prove the following statement by induction on h:

> (*) for every BST T of height at most h storing multiset S, Insert$(T, (K, V))$ returns a valid BST storing multiset $S \cup \{(K, V)\}$.

Our base case is $h = -1$, i.e. an empty tree. In this case, we return a tree containing exactly $(\texttt{K}, \texttt{V})$, where no vertex has any children, so the BST property is automatically satisfied.

For the induction step, assume that (*) is true for trees T of height at most h, and we'll prove it for trees of height at most $h + 1$. Let T be an arbitrary tree of height $h + 1 \geq 0$, so T has a root v. Given a search key K, we split into casework. First consider the case where $\texttt{K} \leq \texttt{K}_v$. Note that T.left is a tree of height at most h, so by induction Insert(T.left, (K, V)) is a tree containing the key-value pairs in T.left plus (K, V) such that every vertex in it satisfies the BST property. Also, v itself satisfies the BST property: its right child hasn't changed, so it still satisfies the BST property. Every key K.left in its left child is either a key in T.left, which was no larger than key_v because T satisfied the BST property, or is K itself, which is at most $\texttt{K}_v$ by the assumption of this case.

The other case, where $\texttt{K} > \texttt{K}_v$, is similar.

- **Search:**

```
1 Search(T,K)
2 Let v be the root of T.
3 if K_v = K then
4 |   return (K_v, V_v)
5 if K > K_v and T.right ≠ ∅ then
6 |   return Search(T.right,K)
7 if K ≤ K_v and T.left ≠ ∅ then
8 |   return Search(T.left,K)
9 return ⊥
```

Proof of correctness: Our proof of correctness is similar in spirit to the previous case of Insertion - it uses induction on the height of the tree.

Specifically, we prove the following statement by induction on h:

> (*) for every BST T of height at most h, if T contains a vertex with key K, then $\texttt{Search}(T, K)$ returns a pair $(\texttt{K}_w, \texttt{V}_w)$ such that $\texttt{K}_w = \texttt{K}$, and otherwise $\texttt{Search}(T, K)$ returns $\perp$.

Our base case is height 0, i.e. a tree whose only vertex is the root v. In this case, T contains K if and only if $\texttt{K} = \texttt{K}_v$, which is exactly the condition under which $\texttt{Search}(T, K)$ returns $(\texttt{K}_v, \texttt{V}_v)$. Otherwise, $\texttt{Search}(T, K)$ returns $\perp$, since $\texttt{T.right} = \texttt{T.left} = \emptyset$. So $\texttt{Search}(T, K)$ is correct.

For the induction step, assume that (*) is true for trees T of height at most h, and we'll prove it for trees of height at most $h + 1$. Given an arbitrary tree T of height $h + 1$ and a search key K, we split into casework. Letting v be the root of T, if $\texttt{K}_v = \texttt{K}$ we are successful because we return $(\texttt{K}_v, \texttt{V}_v)$. If $\texttt{K}_v < \texttt{K}$ then by the BST property K is in T if and only if K is in the right subtree T.right (which is of height at most h), so the recursive call $\texttt{Search}(T.right, K)$ succeeds by induction, and the left call is similar.

Runtime: The analysis is similar to that of Insertion and gives a runtime of $O(h)$.

- **Minimum (or Maximum):**
The algorithm is to move to the left child up until one reaches a vertex with no left child, and then output the key at that vertex.

```
1 MinT(T)
2 Let v be the root of T.
3 if v.left = None then
4 |   return (K_v, V_v)
5 return MinT(T.left)
```

The proof of correctness follows from induction and the BST property and the runtime is $O(h)$.

Similar algorithm holds for finding the maximum, where one moves to the right subtree.

- **Next-smaller (or Next-larger):**
  This query is similar in nature to the search query, and hence the algorithm is quite similar to the search algorithm. We provide it below for completeness.

```
1  NextSmaller(T,q)
2  Let v be the root of T.
3  if q ≤ K_v and T.left ≠ ∅ then
4  |  return NextSmaller(T.left,K)
5  if q ≤ K_v and T.left = ∅ then
6  |  return ⊥
7  if q > K_v and T.right ≠ ∅ then
8  |  W =NextSmaller(T.right,q)
9  |  if W ≠ ⊥ then
10 |  |  return W
11 |  else
12 |  |  return (K_v, V_v)
13 return (K_v, V_v)
```

  We highlight a crucial modification in comparison to the search algorithm. Line 7 considers the case where the queried q is larger than the key at the root. In this case, one has to consider the possibility that the next-smaller key is not in the right subtree of the root, in which case the key-value pair at the root must be returned. Thus, Line 7 checks for the relevant conditions before recursing into the right subtree.

  The proof of correctness can similarly be obtained using induction and the BST property. The runtime is $O(h)$.

- **Deletions:** The algorithm for deletion is the subject of the upcoming SRE.

□

The above exercise shows that Table 1 can be updated to give the Table 2 with corresponding runtimes for BSTs.

5 Balanced Binary Search Trees

So far we have seen that a variety of operations (see Table 2)can be performed on Binary Search trees in time $O(h)$, where h is the height of the tree. Thus, we are now motivated to figure out how we can ensure that our binary search trees remain shallow (e.g. of height $O(\log n)$ where n is the number of items currently in the dataset). While doing so, we need to be sure that we retain the BST property.

There are several different approaches for retaining balance in binary search trees. We'll focus on one, called AVL trees, after mathematicians named Adelson-Velsky and Landis.

Definition 5.1 (AVL Trees)**.** An *AVL Tree* is a binary search tree in which:

- Every vertex has an additional attribute containing the *height* of the tree rooted at that vertex. ("data structure augmentation")

Data structure runtimes		
Queries	Sorted Array	Binary Search Tree
`search`	$O(\log n)$	$O(h)$
`next-smaller`	$O(\log n)$	$O(h)$
`next-larger`	$O(\log n)$	$O(h)$
`min`	$O(1)$	$O(h)$
`max`	$O(1)$	$O(h)$
`rank`	$O(\log n)$	$O(n)$ ($O(h)$ with "size-augmentation")
`insert`	$O(n)$	$O(h)$
`delete`	$O(n)$	$O(h)$

Table 2: List of query runtimes for sorted arrays and binary search trees. Note that if $h = O(\log n)$ for binary search trees, they have exponentially better runtime for insertion and deletion than a sorted array.

- Every pair of siblings in the tree have heights differing by at most 1 (where this includes the case where one sibling is the empty tree and has height -1). ("height-balanced")

Lemma 5.2. *Every AVL Tree with n vertices has height (=depth) at most $2\log_2 n$.*

Proof.
Let $n(h)$ be defined as the min number of vertices in an AVL tree of height $\geq h$. Then, by the second AVL property, for $h \geq 2$, $n(h) \geq n(h-1) + n(h-2) + 1$. Since $n(h-1) \geq n(h-2)$ by definition, we can simplify this equation to

$$n(h) \geq n(h-1) + n(h-2) + 1 \geq 2n(h-2).$$

Now we are at a good place for unrolling the expression for even h (a similar argument holds for odd h):

$$\begin{aligned} n(h) &\geq 2n(h-2) \\ &\geq 4n(h-4) \\ &\geq 8n(h-6) \\ &\vdots \\ &\geq 2^{h/2} n(0) \\ &= 2^{h/2}. \end{aligned}$$

This shows that $h \leq 2\log n(h)$, which proves the lemma. □

Thus, we can apply all of the operations in Table 2 on an AVL Tree in time $O(\log n)$. But recall that we need to maintain a dynamic data structure, and insertion or deletion operation can ruin the AVL property. To fix this, we need additional operations that allow us to move vertices around while preserving the binary search tree property. One important example is *rotation*.

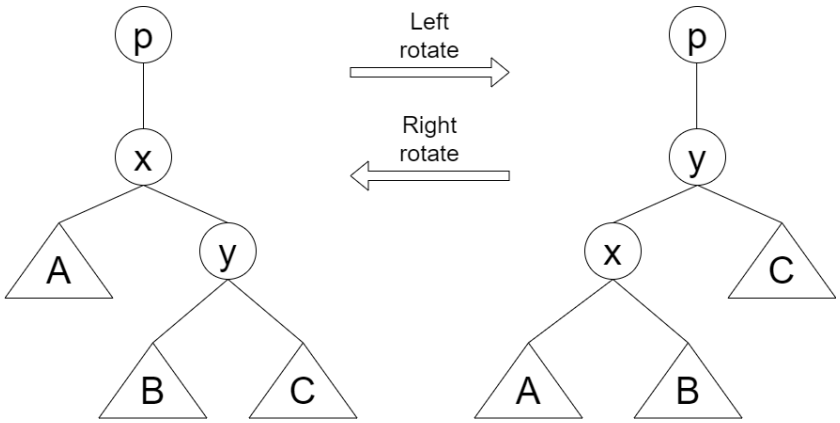


Figure 2: Rotation operations on a BST. On the left, all keys in A are $\leq K_x$, all keys in B are $\in [K_x, K_y]$, and all keys in C are $\geq K_y$. The left rotate operation preserves the BST property.

The rotation operations are depicted in Figure 2. The operation from the left to right is denoted by `Left.rotate(x)`, which takes as input the vertex x. From right to left the operation is denoted as `Right.rotate(y)`.

Runtime of the rotation: As depicted in Figure 2, a rotation only takes $O(1)$ time since it only needs to change a constant number of pointers. This brings us to the following theorem which shows that insertions in AVL tree can be achieved with $O(\log n)$ cost.

Theorem 5.3. *We can insert a new key-value pair into an AVL Tree T while preserving the AVL Tree property in time $O(\log n)$.*

Proof. (sketch)

The algorithm is informally described as follows:

- Perform the `insert` operation on T, adding a new leaf ℓ.
- Since AVL tree is height augmented, update the heights of all the vertices from ℓ to the root. Note that this only requires $O(h) = O(\log n)$ operations.
- If the AVL property is violated, rotations are used to restore the property.

The first two steps require $O(h) = O(\log n)$ operations. The third step also requires $O(h) = O(\log n)$ operations since the AVL property can only be violated at the vertices in the path from the root to ℓ. In total, the algorithm requires $O(\text{height}) = O(\log n)$ runtime.

□

Example: We depict the algorithm with an example, where the values attributes are chosen to be empty. Consider the AVL tree in Figure 3.

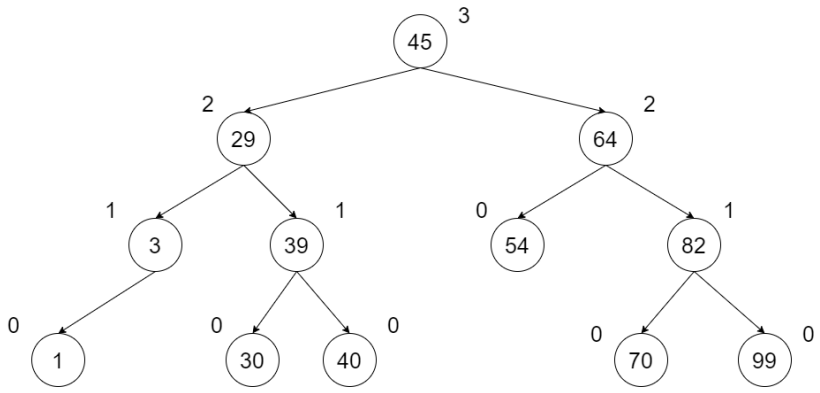


Figure 3: An example of AVL tree.

We insert 65 by creating a new leaf ℓ, resulting in the tree in Figure 4. We update the height of the corresponding vertices (rectangles in blue). Next we check the AVL property, which is violated since vertex 51 has height 0 while vertex 82 has height 2, so their difference is more than 1 (marked in red).

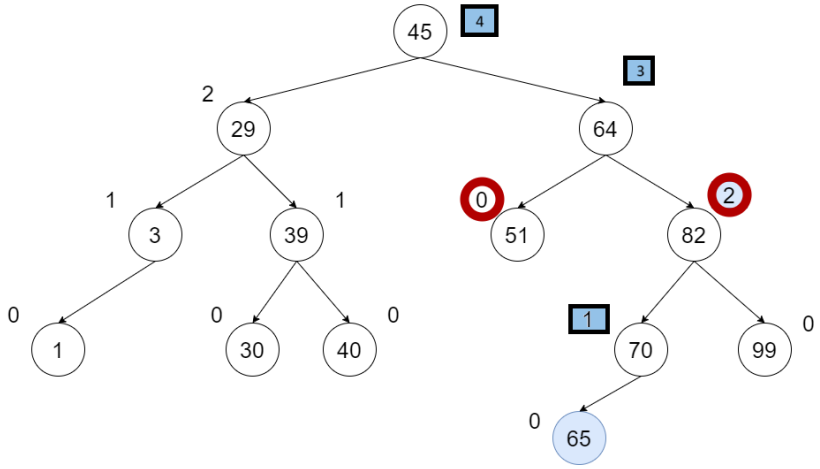


Figure 4: A new leaf is created with key 65. The height attributes are updated.

To fix this, we then use AVL rotations. Let y be the vertex of key 64, and z be the vertex of key 82. We apply `Right.rotate(z)` followed by `Left.rotate(y)`.