CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 4 — January 26, 2023

Prof. Jelani Nelson *Scribe: Nishant Bhakar, Ryan Zhu*

1 Overview

In the last lecture we introduced blocking flows, but these require link-cut trees to operate efficiently. Before discussing these, we'll motivate them by first discussing another data structure in this lecture, splay trees. (Sleator, Tarjan '85)

2 Binary Search Trees

In general, binary search trees store (key, value) pairs in such a way that keys are comparable. Binary search trees can be used to solve many problems, including the dictionary problem, the predecessor problem, and others.

Definition 2.1. A *dictionary* is a data structure that stores a set of (key, value) pairs and allows for the following operations:

1. *Insert*(k, v): Insert a key-value pair into the dictionary.
2. *Query*(k): Return the value associated with the given key in the dictionary.
3. *Delete*(k): Delete the corresponding key-value pair from the dictionary.

Definition 2.2. A *binary search tree* is a binary tree that satisfies the following properties:

1. Each node has a key.
2. Each node has a value.
3. All keys of the left subtree of a node are less than the key of the node.
4. All keys of the right subtree of a node are greater than the key of the node.

Definition 2.3. The *BST Model* is a model of computation that represents binary search trees. Suppose the current node is x. At each time step, we can perform the following operations:

1. *child(x)*: Go to the left or right child of x.
2. *parent(x)*: Go to the parent of x.
3. *rotate(x)*: Rotate x. Rotating a node to the left or right swaps the node with its child. For example, if we rotate the node x to the left, then x becomes the left child of its right child, and the right child of x becomes the left child of x's parent. If x is the root, then the new root is the right child of x. An example is shown in the figure below:

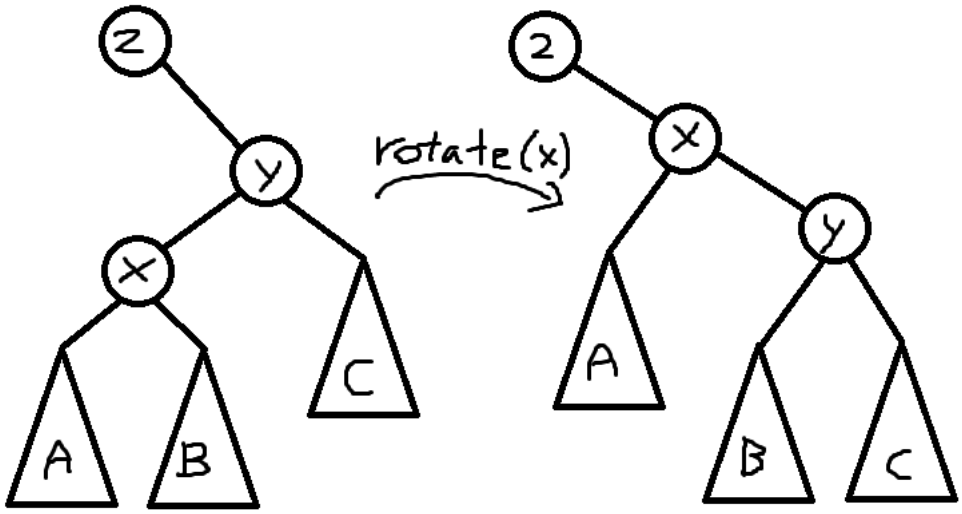

Figure 1: Rotating x.

The above operations relate to online algorithms. Online algorithms run in real time and receive operations sequentially. Most importantly, future operations are unknown.

3 Splay trees

We now turn the discussion from binary search trees in general to splay trees specifically.

Definition 3.1. The *Dynamic Optimality Conjecture* states that the cost of splaying is always within a constant factor of the optimal algorithm for performing searches. More specifically, suppose σ is a sequence of operations, and $\mathrm{OPT}(\sigma)$ is the minimum cost to service σ under the BST model. Then $\mathrm{cost}_{\mathrm{splay}}(\sigma) = \mathcal{O}(\mathrm{OPT}(\sigma))$

The best competitive ratio to the offline optimal BST among all online BSTs is $\mathcal{O}(\log \log n)$, achieved by Tango trees [1].

3.1 Notable Properties

3.1.1 $O(\log n)$ Amortized Operation Cost

Say the amortized cost of query/insert/delete are $t_q/t_i/t_d$. Then, for all operation sequences σ, we have that the total time to serve σ is upper bounded by $(\#\text{ of queries})\, t_q + (\#\text{ of inserts})\, t_i + (\#\text{ of deletes})\, t_d$.

3.1.2 Working Set

For item j, let t_j be the number of operations since j was last accessed, then the amortized cost of accessing j is $\mathcal{O}(t_j)$.

3.1.3 Static Optimality

Suppose the optimal T (found by dynamic programming) put the item i at level ℓ_i in T. Then, the amortized cost of accessing i is $\mathcal{O}(\ell_i)$. This is different from dynamic optimality because it does not take into account the order of operations. If we did know the ordering, we could be more efficient by rotating up frequently accessed items.

3.1.4 Static Finger

Definition 3.2. A *finger* is a pointer to a node in a tree. A *static finger* is a finger that points to a node in a tree, and the finger is not updated after the node is accessed. The amortized cost of accessing a node with a static finger is $\mathcal{O}(\log|f - i + 1|)$, where f is the node pointed to by the finger and i is the node being accessed.

3.1.5 Dynamic Finger

Definition 3.3. A *dynamic finger* is a finger that points to a node in a tree, and the finger is updated after the node is accessed. For instance, if the sequence $\sigma = \{x_1, x_2, x_3, \cdots\}$ then the amortized cost of the ith query is $\mathcal{O}(\log|x_i - x_{i-1} + 1|)$.

3.2 Implementation

We've spent a lot of time talking about properties. Now it's time to finally explain what a splay tree actually is.

Definition 3.4. A *splay tree* is a binary search tree that satisfies the following additional properties in addition to the binary search tree properties:

1. The tree is balanced.
2. The tree is splayed.

3.2.1 Operations

Supported operations on a splay tree include:

1. *query(k)*: Find the node with key k by following pointers and then splay(k).

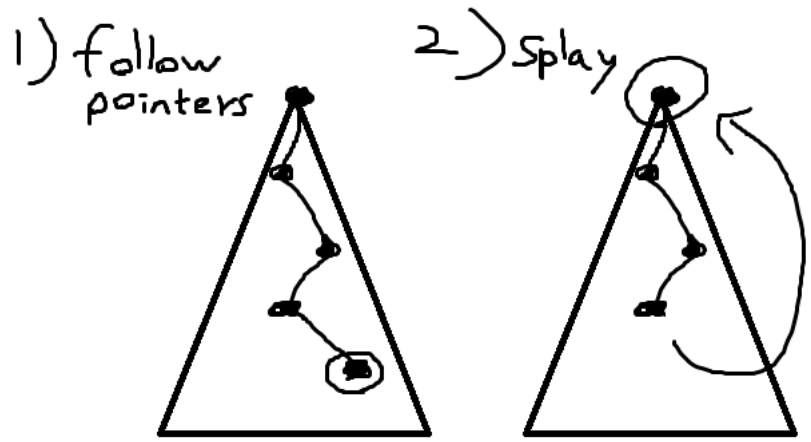

Figure 2: Querying a splay tree.

2. *insert*(k, v): Insert a node with key k and value v into the tree. Begin by following pointers to where the node with key k should be. Insert it a new node with key k and value v, then splay the newly created node to bring it to the root.

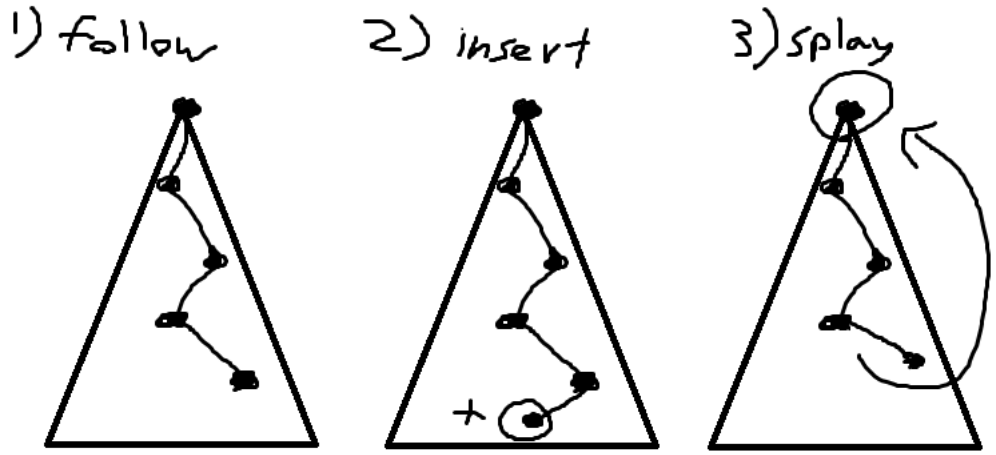

Figure 3: Inserting into a splay tree.

3. *delete*(k): Delete the node with key k from the tree. Begin by following pointers to find where the node with key k is in the tree. Bring it up to the root by splaying it. Then, delete the node, creating two subtrees A and B. Next, splay the node with the largest key in A to the root, and attach B as that node's new right child to join the two subtrees together.

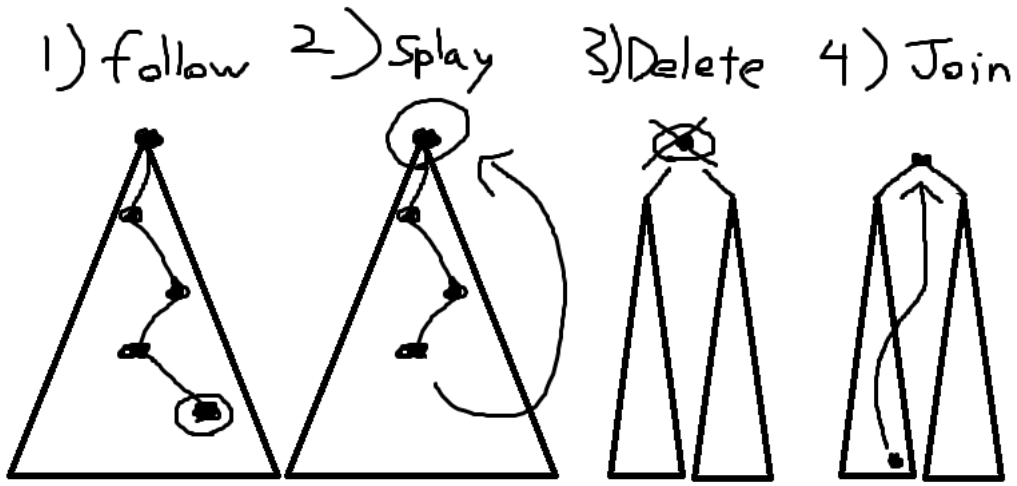

Figure 4: Deleting from a splay tree.

4. *splay(k)*: Rotate the node with key k to the root of the tree. Suppose $y = \text{parent}(x)$ and $z = \text{parent}(y)$. Then there are three cases:

   (a) $z = \text{nil}$: Rotate x.

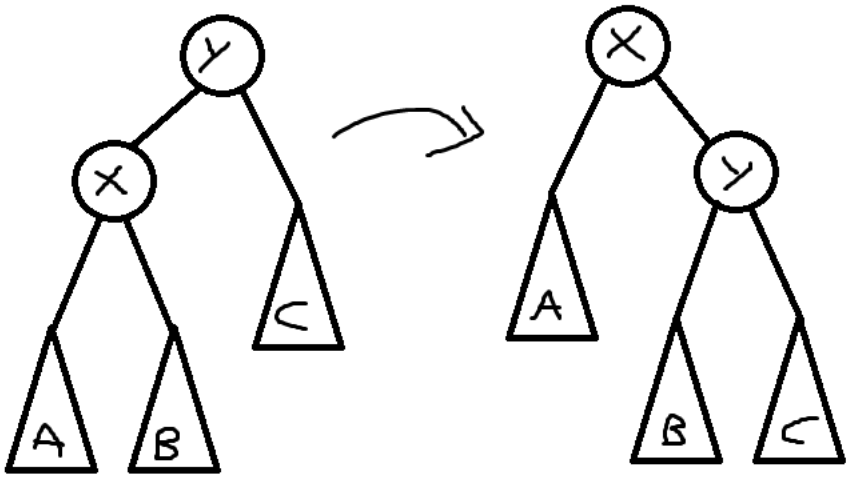

Figure 5: Splaying, case 1

   (b) $x = y$.left and $y = z$.left or $x = y$.right and $y = z$.right: Rotate y and then rotate x.

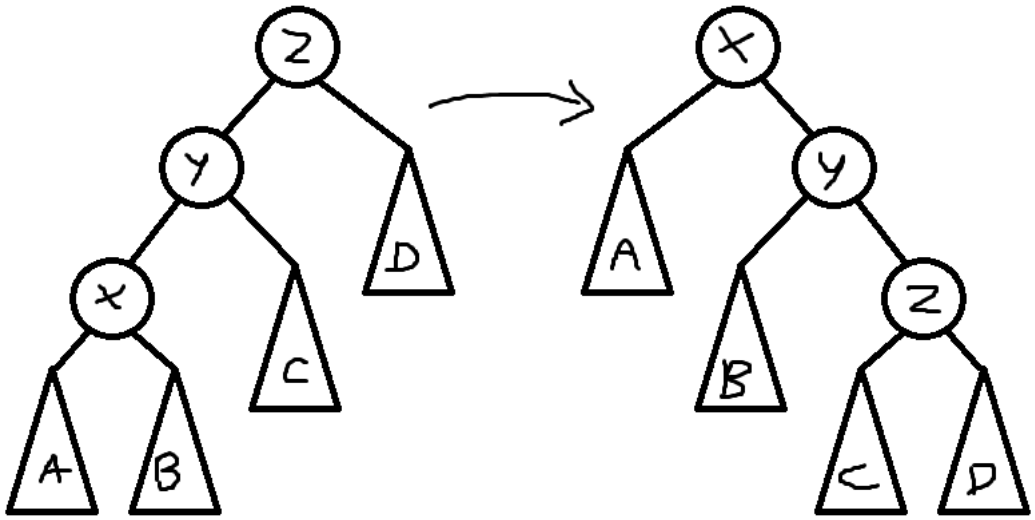

Figure 6: Splaying, case 2

(c) All other cases: Rotate x and then rotate x.

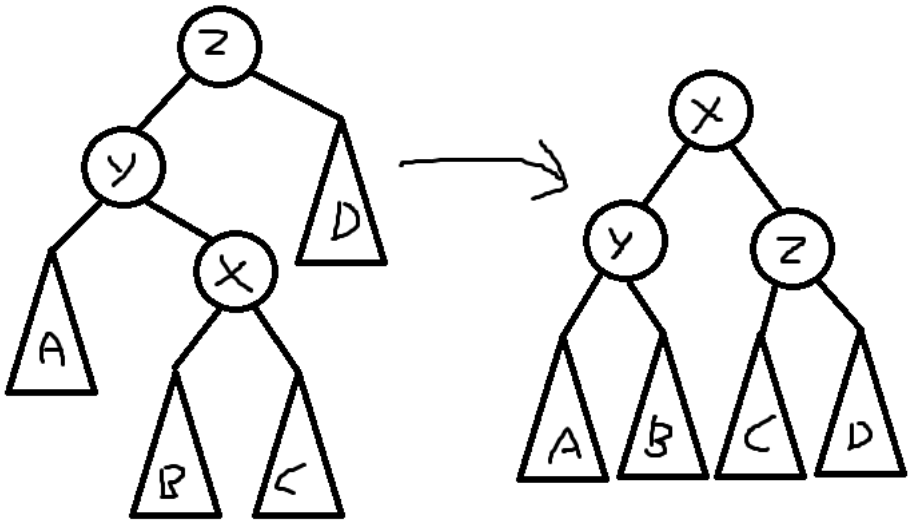

Figure 7: Splaying, case 3

3.2.2 Operation Properties

Properties of operations include:

Claim 3.5. The amortized cost of splay(k) is $\mathcal{O}\left(\log(\frac{W}{s(k)})+1\right)$. For each k in the database, give it weight $w(k)$ and define $s(k) := \sum_{u \in \text{subtree}(k)} w(u) \, | W = \sum_{k \in \text{databasae}} w(k) = s(\text{root})$. Additionally, $r(k) := \log(s(k))$.

Proof. For some state of our data structure, we assign a potential $\Phi\left\{\text{states}\right\} \to \mathcal{R}$. Note Φ can be any function. The potential cost of any operation is true cost $+ \Delta\Phi$ where $\Delta\Phi = \Phi_{\text{after operation}} - \Phi_{\text{before operation}}$. Now splay $\Phi = \sum_k r\left(k\right)$. We want to demonstrate splay$(k) \leq 3\left(r\left(\text{root}\right) - r\left(x\right)\right)+1$. There are the following cases:

1. Case 1: potential cost = true cost $+ \Delta\Phi \leq \left(3r'\left(x\right) - r\left(x\right)\right) + 1$.
2. Cases 2-4: potential cost = true cost $+ \Delta\Phi \leq \left(3r'\left(x\right) - r\left(x\right)\right)$.
3. Note potential cost $= 2 + \Delta\Phi = 2 + \left(r'\left(x\right) + r'\left(y\right) + r'\left(z\right)\right) - \left(r\left(x\right) + r\left(y\right) + r\left(z\right)\right)$. Note $r'\left(x\right) = r\left(z\right)$ and $r'\left(y\right) \leq r'\left(x\right)$ and $r\left(y\right) \geq r\left(x\right)$. All this meaning potential cost $\leq 2 + r'\left(x\right) + r'\left(z\right)$. Note by the AM-GM inequality, $\frac{a+b}{2} \geq \sqrt{ab} \Longrightarrow \frac{r(x)+r'(z)}{2} \leq \log\left(\frac{s(x)+s'(z)}{2}\right) \leq \log\left(\frac{s'(x)}{2}\right) = r'\left(x\right) - 1$, leading to our desired result.

□

References

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