Lecture 23: Binary Search Trees and Dynamic Optimality

Objectives of this lecture

- Define operations on a binary search tree.
- Introduce the online algorithm problem of binary search trees.
- Prove that splay trees have amortized $O(\log n)$ update time.

1 Philosophy

In these notes, we study binary search trees from the perspective of serving a long sequence of accesses. We first define the BST model and compare the static setting, where the tree shape is fixed, with the dynamic setting, where we are allowed to change the tree using rotations. We then introduce splay trees, which use a simple self-adjusting rule: whenever a node is accessed, move it to the root by repeated rotations.

This leads to the broader question of whether there is an online BST algorithm that is always nearly as good as the best possible dynamic strategy for the entire access sequence. This is the motivation for the dynamic optimality conjecture. Finally, we prove the basic guarantee for splay trees: although a single operation can be expensive, the total cost over many operations is only $O((n+q)\log n)$ by amortized analysis.

2 Binary Search Trees

A binary search tree (BST) consists of nodes with keys (values), such that there is a root, every node has at most two children (left and right), and the following ordering property holds: for every node, every key in its left subtree is smaller than the node's key, and every key in its right subtree is larger than the node's key. Thus, given a value x, we can decide whether x appears in the tree by walking down the tree.

The basic operations are search, insert, and delete. Search follows the unique comparison path from the root. Insert searches for the appropriate empty child position and places the new key there. Delete is slightly subtler: if the node has two children, we may swap it with its predecessor or successor and then delete a node with at most one child. In this note, we focus on the

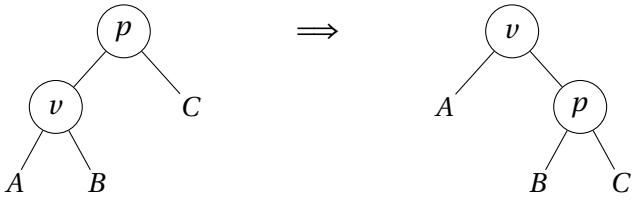
setting where the tree is already initialized but there are many searches (and there are ways to update the structure of the tree over time).

3 Online Sequence of Accesses

Consider a BST T with keys numbered 1 to n. Consider a sequence of *accesses* $x_1, x_2, \ldots, x_q$ where each $x_i \in \{1, 2, \ldots, n\}$. Think of this as saying that you want to go touch the nodes $x_1, \ldots, x_q$ in the BST T in that order.

Static BST. If the BST does not change, then the cost of accessing $x_1, \ldots, x_q$ is the sum of the depths of the x_i's in the tree T. Finding a BST that minimizes this is exactly the *optimal BST* problem that we studied at the beginning of the course (whose solution used dynamic programming).

Dynamic BST. During this sequence of accesses, we might want to allow a way to update the BST. This is done through a sequence of operations called *rotations*. A rotation is most easily described by a picture. Specifically, if v is a node that is not the root, a rotation changes the tree as follows. The picture below is the case where v is a left child; the right-child case is symmetric.



You can check that the resulting tree is still a valid BST.

Computing the dynamic cost. We need a model to explain what the cost is for a dynamic BST. This is most clearly stated as follows. Initialize a pointer at the root. You can do the following operations, which all cost 1.

1. Move the pointer to a child or parent.
2. Rotate the node the pointer is at upward.
3. You can exit the algorithm when the pointer is at the node with the desired key value x.

Then the total cost of a sequence of accesses is just the total cost to do all the accesses (where cost is defined by the items above).

Given a sequence, is it easy to compute the minimal cost? Is it even easy to approximate the minimum cost up to a constant factor? This is a wide open problem. Motivated by this, Sleator and Tarjan stated the dynamic optimality conjecture. However, we first need to introduce splay trees, which are a specific type of dynamic BST.

4 Splay Trees and Dynamic Optimality

Sleator and Tarjan conjectured that the *splay tree* has cost within a constant factor of the minimal cost (of a dynamic tree). Here is how a splay tree works when a node x is accessed.

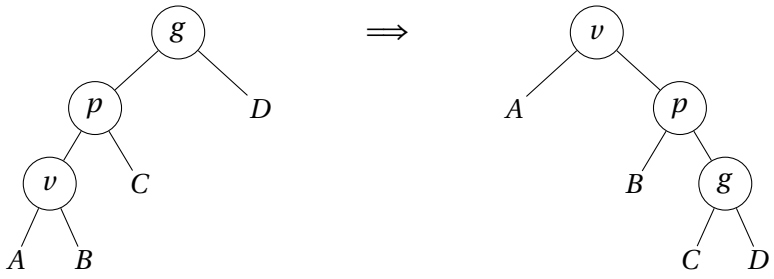
1. Go to node x.
2. *Splay* x up to the root.

Here is how a splay works. Let v be the current node, with parent $p(v)$ and grandparent $p(p(v))$ whenever these exist. We have the following cases.

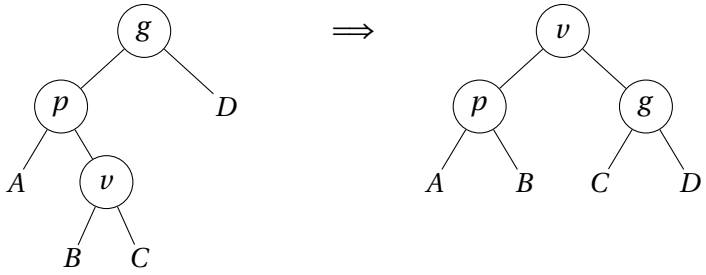
- (Zig) If $p(v)$ is the root, rotate v up once and stop.
- (Zig-zig) If $p(v) \to v$ and $p(p(v)) \to p(v)$ are both left children or both right children, then rotate up $p(v)$ and then rotate up v.
- (Zig-zag) If $p(v) \to v$ and $p(p(v)) \to p(v)$ are in opposite directions, rotate v up twice.

Again, this is easiest to see with pictures. For simplicity let write $p(v)$ as p and $p(p(v))$ as g (where g means grandparent).

Zig-zig.



Zig-zag.



Each splay step decreases the depth of v, so the process eventually terminates with v at the root.

Problem: Dynamic optimality conjecture

The cost of a splay tree is within an $O(1)$ factor of the optimal cost for any sequence of accesses.

This conjecture remains open. We will not discuss the best known partial results here; instead, we prove the much easier theorem that splay trees have logarithmic amortized cost.

5 Amortized Analysis of Splay Trees

In this section, we prove the following theorem.

Theorem: Splay Trees

The cost of q accesses $x_1, \dots, x_q$ implemented on a splay tree with n nodes is at most $O((n+q)\log n)$.

All logarithms in this section are base 2. For a node u, let $s(u)$ denote the size of the subtree rooted at u, and define its *rank* by

$$r(u) = \log s(u).$$

Define the potential of the whole tree to be

$$\Phi(T) = \sum_u r(u).$$

Since every subtree has size at most n, we always have

$$0 \le \Phi(T) \le n \log n.$$

We first analyze the cost of the splaying phase alone.

Lemma: Access Lemma

Suppose we splay a node x to the root of a tree T. If R is the number of rotations performed during this splay, then

$$R + \Delta\Phi \le 3\big(r(\text{root of } T) - r(x)\big) + 1 \le 3\log n + 1.$$

Proof. Write Φ and Φ' for the potential before and after a single splay step. Only the nodes involved in that step can change rank.

Zig step. Here the actual cost is 1. Let p be the parent of x. After the rotation, x becomes the parent of p, so $r'(p) \le r'(x)$. Also, x was in the subtree of p before the rotation, so $r(p) \ge r(x)$.

Hence

$$\begin{aligned}1+(\Phi'-\Phi) &= 1+r'(x)+r'(p)-r(x)-r(p)\\ &\le 1+2r'(x)-2r(x)\\ &\le 1+3\big(r'(x)-r(x)\big).\end{aligned}$$

Zig-zig and zig-zag steps. Here the actual cost is 2. Let p and g be the parent and grandparent of x before the step. After the step, both p and g become children of x, so their new subtrees are disjoint and satisfy

$$s'(p)+s'(g)\le s'(x).$$

By AM-GM,

$$r'(p)+r'(g)=\log\big(s'(p)s'(g)\big)\le 2\log\left(\frac{s'(p)+s'(g)}{2}\right)\le 2r'(x)-2.$$

Also, before the step, the node x lies in the subtrees of both p and g, so $r(p)\ge r(x)$ and $r(g)\ge r(x)$. Therefore

$$\begin{aligned}2+(\Phi'-\Phi) &= 2+r'(x)+r'(p)+r'(g)-r(x)-r(p)-r(g)\\ &\le 2+r'(x)+(2r'(x)-2)-3r(x)\\ &= 3\big(r'(x)-r(x)\big).\end{aligned}$$

Now sum these inequalities over all splay steps. The rank terms telescope, and the final rank of x is exactly the rank of the root after the splay. Thus

$$R+\Delta\Phi\le 3\big(r(\text{root after splay})-r(x)\big)+1.$$

But after the splay, x is the root of the entire tree, so

$$r(\text{root after splay})=\log n.$$

This proves the lemma. □

Proof of the theorem. For the ith access, let R_i be the number of rotations used in the splay of x_i, and let Φ_i be the potential after that access. By the access lemma,

$$R_i+(\Phi_i-\Phi_{i-1})\le 3\log n+1.$$

Summing over all i gives

$$\sum_{i=1}^{q} R_i\le q(3\log n+1)+\Phi_0-\Phi_q\le q(3\log n+1)+n\log n.$$

So the total number of rotations over the whole sequence is $O((n+q)\log n)$.

Now compare this with the full access cost. If x_i is at depth d_i before its splay, then searching for x_i from the root uses exactly d_i pointer moves. Also, every rotation in the splay decreases

the depth of x_i by exactly 1, so the splay performs exactly d_i rotations. Hence the search cost for the ith access equals R_i, and the total cost of that access is at most

$$R_i \text{ (search)} + R_i \text{ (rotations)} + 1 \text{ (exit)} \le 2R_i + 1.$$

Therefore the total access cost is at most

$$\sum_{i=1}^{q} (2R_i + 1) \le 2 \sum_{i=1}^{q} R_i + q = O((n+q)\log n).$$

□

Corollary: Main corollary

Search, insert, and delete each take amortized $O(\log n)$ time in a splay tree.

Proof. The theorem gives the search bound. For insertion, we do an ordinary BST insertion and then splay the inserted node. For deletion, we splay the node to be deleted to the root, remove it, and then join the two remaining subtrees by splaying the maximum node of the left subtree. Each update uses only a constant amount of extra work on top of a constant number of splays, so the amortized cost is still $O(\log n)$. □