In today's lecture, we will discuss:

- binary search trees in general
- definition of splay trees
- analysis of splay trees

The analysis of splay trees uses the potential function approach we discussed in the previous lecture. It seems to be required.

1 Binary Search Trees

These lecture notes assume that you have seen binary search trees (BSTs) before. They do not contain much expository or backtround material on the basics of BSTs.

Binary search trees is a class of data structures where:

1. Each node stores a piece of data
2. Each node has two pointers to two other binary search trees
3. The overall structure of the pointers is a tree (there's a root, it's acyclic, and every node is reachable from the root.)

Binary search trees are a way to store a update a set of items, where there is an ordering on the items. I know this is rather vague. But there is not a precise way to define the gamut of applications of search trees. In general, there are two classes of applications. Those where each item has a key value from a totally ordered universe, and those where the tree is used as an efficient way to represent an ordered list of items.

Some applications of binary search trees:

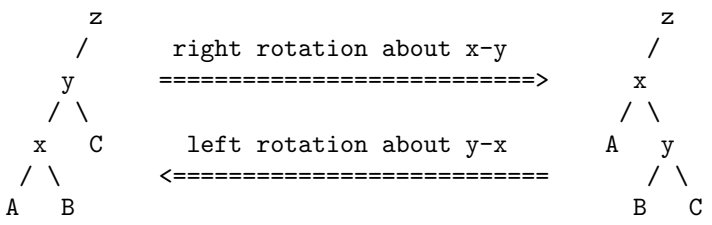
- Storing a set of names, and being able to lookup based on a prefix of the name. (Used in internet routers.)
- Storing a path in a graph, and being able to reverse any subsection of the path in $O(\log n)$ time. (Useful in travelling salesman problems).
- Being able to quickly determine the rank of a given node in a set.
- Being able to split a set S at a key value x into S_1 (the set of keys $< x$) and S_2 (those $> x$) in $O(\log n)$ time.

Given a tree[1] T an *in-order* traversal of the tree is defined recursively as follows. To traverse a tree rooted at a node x, first traverse the left child of x, then traverse x, then traverse the right child of x. When a tree is used to store a set of keys from a totally ordered universe, the in-order traversal will visit the keys in ascending order.

A *rotation* in binary search tree is a local restructuring operation that preserves the order of the nodes, but changes the depths of some of the nodes. Rotations are used by many BST algorithms to keep the tree "balanced", thus assuring that the depth is logarithmic.

[1] In this lecture "tree" is used synonymously with "BST".

A rotation involves a pair of nodes `x` and its parent `y`. After the rotation `x` is the parent of `y`. The update is done in such a way as to guarantee that the in-order traversal of the tree does not change. So if x is initially the left child of y, then after the rotation y is the right child of x. This is known as a *right rotation*. A *left rotation* is the mirror image variant.

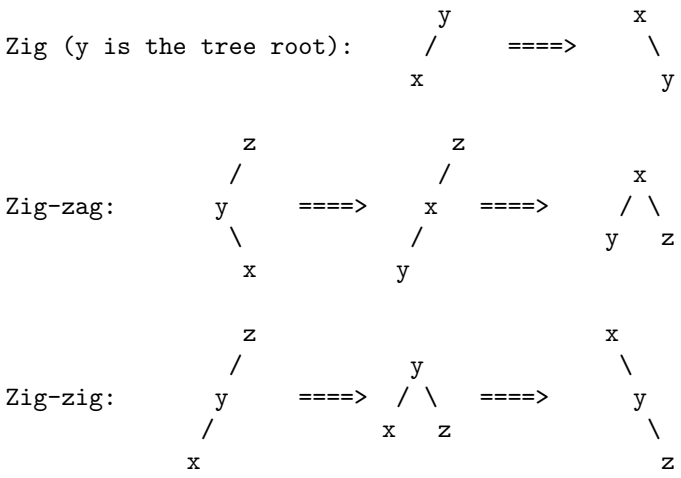


2 Splay Trees (self-adjusting search trees)

These notes just describe the *bottom-up splaying* algorithm, the proof of the access lemma, and a few applications. (See `http://www.link.cs.cmu.edu/splay/` for more information along with an interactive splay tree demo.)

At a high level, every time a node is accessed in a splay tree, it is moved to the root of the tree. We will show that the amortized cost of the operation is $O(\log n)$. Just moving the element to the root by rotating it up the tree does not have this property. (To see this, start with a tree of nodes labeled $0, 1, 2, \ldots, n-1$ that is just a long left path down from the root, with 0 being the leftmost node in the tree. If we sequentially rotate 0, then 1, then 2, etc. to the root, the tree that results is the same as the starting tree, but the total work is $\Omega(n^2)$, for an amortized lower bound of $\Omega(n)$ per operation.) Splay trees perform movements in a very special way that guarantees this logarithmic amortized bound.

I'll describe the algorithm by giving three rewrite rules in the form of pictures. In these pictures, `x` is the node that was accessed (that will eventually be at the root of the tree). By looking at the structure of the 3-node tree defined by `x`, `x`'s parent, and `x`'s grandparent we decide which of the following three rules to follow. We continue to apply the rules until `x` is at the root of the tree:



Some notes:

1. Each arrow in the diagram above represents a rotation. Zig case does one rotation, the other cases do two rotations.

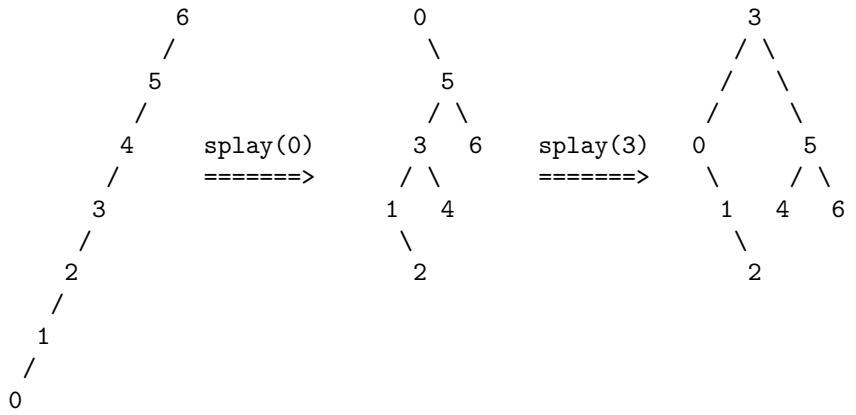
2. A *splay step* refers to applying one of these rewrite rules. Later we will be proving bounds on number of splay steps done. Each such step can be executed in constant time.

3. Each rule has a mirror image variant, which I have not shown for brevity. So for example, if `x` is the right child of `y` (the root), then the mirror image of the Zig case applies. In this case a left rotation would be done to put `x` above `y`, instead of the right rotation show above

4. The Zig-zig rule is the one that distinguishes splaying from just rotating `x` to the root of the tree.

5. There are a number of alternative versions of the algorithm, such as top-down splaying, which may be more efficient than the bottom-up version described here. Code for the top-down version is available here `http://www.link.cs.cmu.edu/splay/`.

Here are some examples:



To analyze the performance of splaying, we start by assuming that each node x has a weight $w(x) > 0$. These weights can be chosen arbitrarily. For each assignment of weights we will be able to derive a bound on the cost of a sequence of accesses. We can choose the weight assignment that gives the best bound. By giving the frequently accessed elements a high weight, we will be able to get tighter bounds on the running time. Note that the weights are only used in the analysis, and do not change the algorithm at all. (A commonly used case is to assign all the weights to be 1.)

2.1 Sizes and Ranks

Before we can state our performance lemma, we need to define two more quantities. Consider any tree T — think of this as a tree obtained at some point of the splay tree algorithm, but all these definitions hold for any tree. The *size of a node* x in T is the total weight of all the nodes in the subtree rooted at x. If we let $T(x)$ denote the subtree rooted at x, then the size of x is denoted by

$$s(x) = \sum_{y \in T(x)} w(y)$$

Now we define the *rank of a node* x as

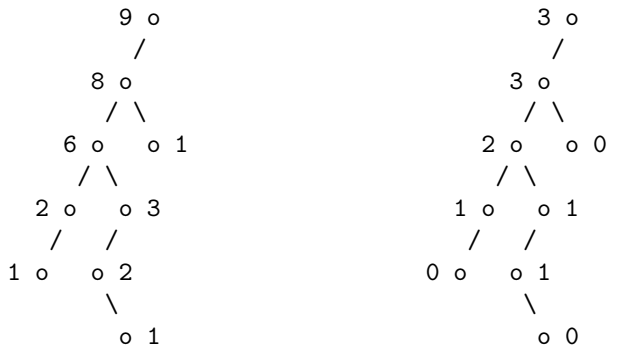
$$r(x) = \lfloor \log(s(x)) \rfloor$$

For each node x, we'll keep $r(x)$ tokens on that node if we are using the banker's method. (Alternatively, the *potential function* $\Phi(T)$ corresponding to the tree T is just the sums of the ranks of

all the nodes in the tree.)

$$\Phi(T) := \sum_{x \in T} r(x).$$

Here's an example to illustrate this: Here's a tree, labeled with sizes on the left and ranks on the right.



Notes about this potential Φ:

1. Doing a rotation between a pair of nodes x and y only effects the ranks of the nodes x and y, and no other nodes in the tree. Furthermore, if y was x's parent before the rotation, then the rank of y before the rotation equals the rank of x after the rotation.
2. Assuming all the weights are 1, the potential of a balanced tree is $O(n)$, and the potential of a long chain (most unbalanced tree) is $O(n \log n)$.
3. In the banker's view of amortized analysis, we can think of having $r(x)$ tokens on node x.

3 The Amortized Analysis

Lemma 1 (Access Lemma) *Take any tree T with root t, and any weights $w(\cdot)$ on the nodes. Suppose you splay node x; let T' be the new tree. Then*

$$\begin{aligned} \textit{amortized number of splaying steps} &= \textit{actual number of splaying steps} + (\Phi(T') - \Phi(T)) \\ &\leq 3(r(t) - r(x)) + 1. \end{aligned}$$

Proof: As we do the work, we must pay one token for each splay step we do. Furthermore we must make sure that we always leave $r(x)$ tokens on node x as we do our restructuring. We are going to allocate $3(r(t) - r(x)) + 1$ tokens to do the splay. Our job in the proof is to show that this is enough.

First we need the following observation about ranks, called the Rank Rule.

> **Rank Rule:** Suppose that two siblings have the same rank, r. Then the parent has rank at least $r + 1$.

This is because if the rank is r, then the size is at least 2^r. If both siblings have size at least 2^r, then the total size is at least 2^{r+1} and we conclude that the rank is at least $r + 1$. We can represent this with the following diagram:

```
  x
 / \       Then x >= r+1
r   r
```

Conversly, suppose we find a situation where a node and its parent have the same rank, r. Then the other sibling of the node must have rank $< r$. So if we have three nodes configured as follows, with these ranks:

```
  r
 / \       Then x < r
x   r
```

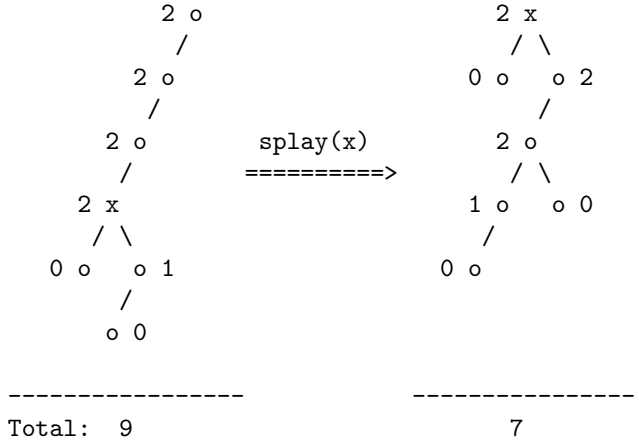
Now we can go back to proving the lemma. The approach we take is to show that $3(r(z) - r(x))$ tokens are sufficient to pay for a zig-zag or a zig-zig step. (Here z refers to x's grandparent, as in the definition of splaying.) And that $3(r(y) - r(x)) + 1$ is sufficient to pay for the zig step.

We can express this a little bit differently. Take the zig-zag amortized cost $3(r(z) - r(x))$ mentioned above. We can write this as $3(r'(x) - r(x))$, where $r'() = r_{T'}()$ is the rank function after the step, and $r() = r_T()$ represents the rank function before the step. This identity holds because the rank of x after the step is the same as the rank of z before the step.

Using this notation it becomes obvious that when we sum these costs to compute the amortized cost for the entire splay operation, they telescope to:

$$3(r(t) - r(x)) + 1.$$

Note that the +1 comes from the zig step, which can happen only once. Here's an example, the labels are ranks:

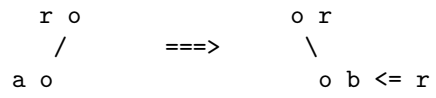


We allocated: 3(2-2)+1 = 1 tokens that are supposed to pay for the splay of x. There are two more tokens in the tree before than are needed at the end. (The potential decreases by two.) So we have a total of 3 tokens to spend on the splay steps of this splay.

But there are 2 splay steps. So $3 > 2$, and we have enough.

It remains to show these bounds for the individual steps. There are three cases, one for each of the types of splay steps.

The Zig Case: The following diagram shows the ranks of the two nodes that change as a result of the zig step. We've allocated $3(r - a) + 1$ to pay for the step. We'll show that this is sufficient.



The actual cost is 1, so we spend one token on this. We take the tokens on a and augment them with another $r - a$ (resulting in a pile of r tokens) and put them on b. Thus the total number of tokens needed is $1 + r - a$. This is at most $1 + 3(r - a)$.

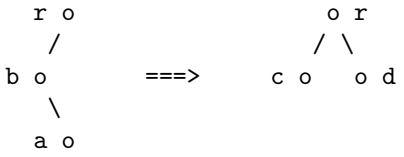
The Zig-zag Case: We'll split it further into 2 cases:

Case 1: The rank does not increase between the starting node and ending node of the step.

By the Rank Rule, one of a or b must be $< r$, so one token is released from the data structure. we use this to pay for the work.

Thus, our allocation of $3(r - r) = 0$ is sufficient to pay for this.

Case 2: The rank does increase between the starting node and ending node of the step.



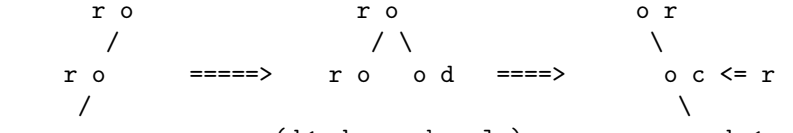
Note that $r - a > 0$. So use $r - a$ (which is at least 1) to pay for the work. Use $r - a$ to augment the tokens on a to create a pile of r tokens which can supply c. Finally use $r - b$ ($\leq r - a$) tokens to add to the pile on b to make a pile of r tokens, which are enough to cover d.

Thus $3(r - a)$ tokens are sufficient, which completes the zig-zag case.

(We actually only need $2(r - a)$ because it is easily shown that $c \leq b$.)

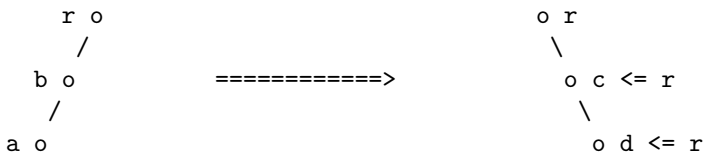
The Zig-zig Case: Again, we split into two cases.

Case 1: The rank does not increase between the starting node and the ending node of the step.



As in the zig-zag case, we use the token gained because of $d < r$ to pay for the step. Thus we have $3(r - r) = 0$ tokens and we need 0.

Case 2: The rank does increase between the starting node and the ending node of the step.



We can use $r - a$ (which is at least 1) to pay for the work, use $r - a$ to boost the tokens on a to cover those needed for d, and use $r - a$ to boost the tokens on b to cover those needed for c.

Summing these three gives $3(r - a)$, which completes the analysis of the zig-zig case.

So in every case we have shown that $3(r'(x) - r(x))$ is sufficient to pay for the splay step (modulo the extra $+1$ needed for the zig case). Thus these amortized costs telescope giving an amortized number of splay steps of $3(r(t) - r(x)) + 1$.

■

4 Balance Theorem

Using the Access Lemma we can prove the bound on the amortized cost of splaying: this is captured in the Balance Theorem.

Theorem 2 (Balance Theorem) *A sequence of m splays in a tree of n nodes takes time*

$$O(m \log n + n \log n).$$

Proof: Suppose all the weights equal 1. Then the access lemma says that if we splay node x in tree T to get the new tree T'

$$\begin{aligned} \text{actual number of splaying steps } + \big(\Phi(T') - \Phi(T)\big) &\leq 3(\log_2 n - \log_2 |T(x)|) + 1 \\ &\leq 3 \log_2 n + 1. \end{aligned}$$

Hence, if we start off with a tree T_0 of size at most n and perform any sequence of m splays to it

$$T_0 \xrightarrow{splay} T_1 \xrightarrow{splay} T_2 \xrightarrow{splay} \cdots \xrightarrow{splay} T_m,$$

repeatedly using this inequality m times shows:

$$\text{actual total number of splaying steps } + (\Phi(T_m) - \Phi(T_0)) \leq m(3 \log_2 n + 1).$$

In any tree T with unit weights, each $s(x) \leq n$ so each $r(x) \leq \log_2 n$ so $\Phi(T) \leq n \log_2 n$; also $\Phi(T) \geq 0$. Rearranging, we get

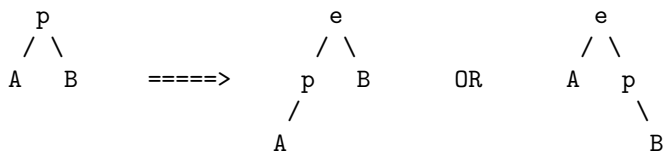
$$\begin{aligned} \text{actual total number of splaying steps } &\leq m(3 \log_2 n + 1) + (\Phi(T_0) - \Phi(T_m)) \\ &\leq O(m \log n) + O(n \log n). \end{aligned}$$

This proves the Balance Theorem. ■

5 Using Splaying with Searching and Updates

Searching: Since the keys in the splay tree are stored in in-order, the usual BST searching algorithm will suffice to find a node (or tell you that it is not there). However with splay trees it is necessary to splay the last node you touch in order to pay for the work of searching for the node. (This is clear if you think about what would happen if you repeatedly searched for the deepest node in a very unbalanced tree without every splaying.)

Insertion: Say we want to insert a new key `e` into the tree. Proceed like ordinary binary search tree insertion, but stop short of linking the new item `e` into the tree. Let the node you were about to link `e` to be called `p`. We splay `p` to the root. Now construct a new tree according to one of the following pictures:



The choice of which pattern to use on the right is determined by whether `e` is less than `p` or greater than it. Note that after this operation one of `p`'s children will be empty.

Let's analyze the amortized cost of this opertion when all the weights are 1 and the tree initially has n nodes in it. The amortized cost of the splay is $O(\log n)$ according to the Access Lemma. This also pays for the work of finding node `p`. But we're not done yet, because attaching `e` to the tree changes the potential, and we have to take this into account. The number of tokens on `p` can only decrease (its size can not increase). However we have to populate `e` with tokens, the number of which is at most $\lfloor \log(n+1) \rfloor = O(\log n)$. Thus the total amortized cost of the insertion is $O(\log n)$.

An alternative insertion algorithm is to do the ordinary binary tree insertion (link `e` to `p`), and then simply splay `e` to the root. This also has efficient amortized cost, but the analysis is a bit more complex.

Join: Here we have two trees, say, A, and B. We want to put them together with all the items in A to the left of all the items in B. This is called the *Join* operation. This is done as follows. Splay the rightmost node of A. Now make B the right child of this node. The amortized cost of the operation is easily seen to be $O(\log n)$ where n is the number of nodes in the tree after joining. (As in the case of insertion, we are using the uniform weight assignment to do this analysis.)

Delete: To delete a node `x`, simply splay it to the root and then Join its left and right subtrees together as described above. The amortized cost is easily seen to be $O(\log n)$ where n is the size of the tree before the deletion.

6 Additional Applications of the Access Lemms

One of the things that distinguishes splay trees from other forms of balanced trees is the fact that they are provably more efficient various natural access patterns. These theorems are proven by playing with the weight function in the definition of the potential function. In this section we describe some of these theorems.

The following theorem shows that splay trees perform within a constant factor of any static tree.

Theorem 3 (Static Optimality Theorem) *Let T be any static search tree with n nodes. Let t be the number of comparisons done when searching for all the nodes in a sequence s of accesses. (This sum of the depths of all the nodes). The cost of splaying that sequence of requests, starting with any initial splay tree is $O(n^2 + t)$.*

The following theorem says that if we measure the cost of an access by the log of the distance to a specific element called the finger, then this is a bound on the actual cost (modulo an additive term). There is a counter-intuitive aspect to this, which is that the algorithm achieves this without knowing which element you've picked to be the finger.

An alternative interpretation is that if the accesses have spatial locality (they cluster around a spacific place) then the sequence is cheap to process.

Theorem 4 (Static Finger Theorem) *Consider a sequence of m accesses in an n-node splay tree. Let $a[j]$ denote the jth element accessed. Let f be a specific element called the finger. Finally*

let $|e - f|$ denote the distance (in the in-order list of the tree) between elements e and f. Then the following is a bound on the cost of splaying the sequence:

$$O(m + n \log n + \sum_{1 \le j \le m} \log(|f - a[j]| + 1))$$

If the previous theorem addresses spatial locality in the sequence, the following one address temporal locality. If I access an element, then shortly after this, I access it again, then the second access is cheap. (It's the log of the number of different items accessed between the two.)

Theorem 5 (Working Set Theorem) *In a sequence of m accesses in an n-node splay tree, consider the jth access, which is to an item x. Let $t(j)$ be the number of different items accessed between the current access to x and the previous one. (If there is no previous access, then $t(j) = n$ the size of the tree.) Then the total cost of the access sequence is*

$$O(m + n \log n + \sum_{1 \le j \le m} \log(t(j) + 1)$$

The following two theorems also addresses spatial locality, in a different way from the Static Finger Theorem. Their proofs do not follow from the Access Lemma.

Theorem 6 (Sequential Access Theorem) *The cost of the access sequence that accesses each of the n items in the tree in left-to-right order (in-order) is $O(n)$.*

The following theorem generalizes the Sequential Access Theorem. It says that the cost of splaying an element can be tied to the log of the distance between the current element being splayed and the one that was just splayed.

Theorem 7 (Dynamic Finger Theorem) *Incorporation the notation from the Static Finger Theorem, the cost of an access sequence is bounded by*

$$O(m + n + \sum_{2 \le j \le n} \log(|a[j-1] - a[j]| + 1))$$