Splay Trees

Today's lecture will focus on a very interesting case study of amortized analysis, a powerful binary search tree (BST) data structure called the *Splay Tree*. Splay trees have a lot of nice practical properties, they make many problems that are tricky to solve using ordinary BSTs much easier. They also have many nice theoretical properties which show that they are often provably more efficient than other BSTs for several common use cases. We will focus mainly on the analysis, which makes heavy use of the potential method from the previous lecture.

Objectives of this lecture

In this lecture, we want to:

- Review binary search trees (BSTs)
- Understand the functionality of the Splay tree data structure
- Analyze the cost of the Splay tree data structure using the potential method

Recommended study resources

- `http://www.link.cs.cmu.edu/splay/`

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 are 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 and update a set of items, where there is an ordering on the items. 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.

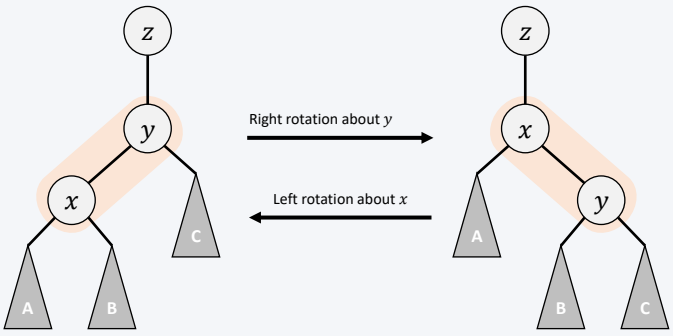
1.1 Rotations

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.

A rotation involves a pair of nodes x and y, where y is the parent of x. 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.

Key Idea: Rotations

A *left rotation* and a *right rotation* are defined by the following schematic:



In a right rotation, a left child takes the place of its parent, and vice versa.

In these notes, we are defining a rotation with respect to the *parent* node of the rotation, e.g., a right rotation about y means to move the left child of y up, and a left rotation about x means

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

to move the right child of x up. This is just a convention, and others exist. Some sources will define rotations with respect to the lower node, for example, they might call the two rotations above a rotation of x and a rotation of y respectively.

2 Splay Trees (self-adjusting search trees)

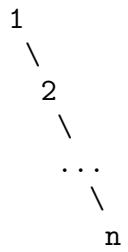
These notes just describe the *bottom-up splaying* algorithm, the proof of the access lemma, and a few applications. The key of idea of splay trees is *locality*. Locality is a common property of many real-world problems, which essentially says that data that was accessed recently is likely to be accessed again soon. How can we translate this idea into a binary search tree algorithm? Here's one way:

Key Idea: Move recently accessed nodes to the root

Whenever we access a node in the tree, move it to the root.

At a high level, every time a node is accessed in a splay tree, it is moved to the root of the tree. The exact way in which it is moved to the root of the tree turns out to be important, as doing so naively can be inefficient. The simplest way to move an element to the root would be to rotate with respect to its parent over and over again until it became the root. The following exercise should help you see why this is not very efficient. What would a worst-case sequence of accesses look like under this scheme?

Problem 1. Inefficient move-to-root algorithm Starting with a tree of height n consisting of the elements 1 through n:

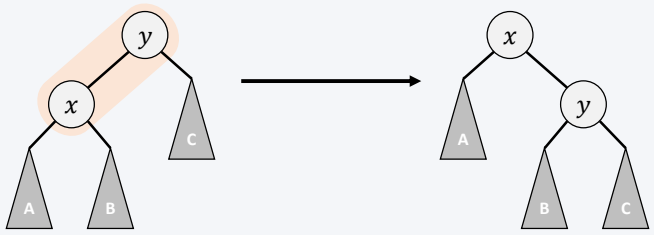


show what happens if you move n to the root by performing a sequence of left rotations. If we start with this tree and sequentially rotate n, then $n-1$, then $n-2$, etc. to the root, the tree that results is the same as the starting tree, but the total work is $\Omega(n^2)$, for an amortized lower bound of $\Omega(n)$ per operation.

We need to move the node to the root using a more sophisticated pattern to avoid this cost. This pattern is called *splaying*. We will show that the amortized cost of the splay operation is $O(\log n)$. We'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 (y), and x's grandparent (z) 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.

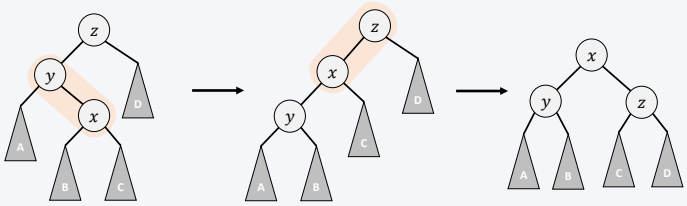
Definition: The zig rule

If we are splaying a node x, and its parent y is the root, we apply the zig rule. The zig rule is just a left or right rotation about y.



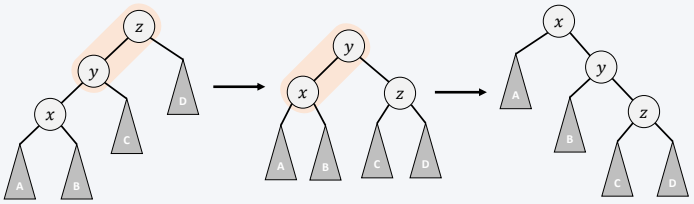
Definition: The zig-zag rule

If we are splaying a node x whose parent is y and grandparent is z, and x and y are not both left children or both right children, we apply the zig-zag rule. The zig-zag rule is two consecutive rotations with respect to x's parent (i.e., a rotation with respect to y followed by a rotation with respect to z).



Definition: The zig-zig rule

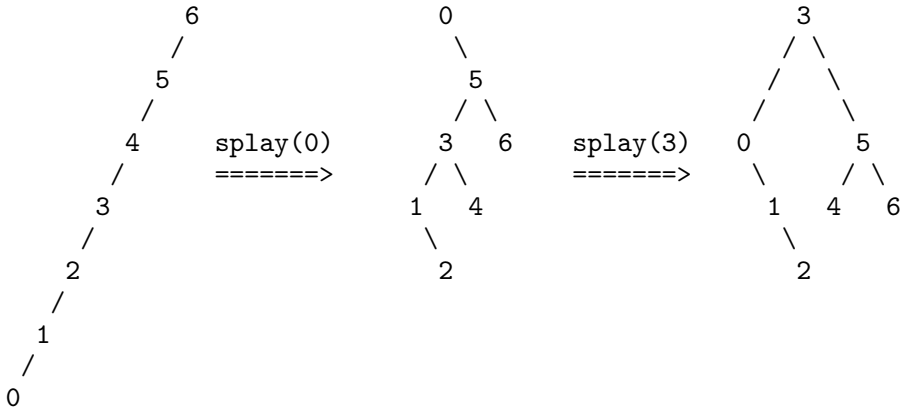
If we are splaying a node x whose parent is y and grandparent is z, and x and y are both left or both right children, we apply the zig-zig rule. The zig-zig rule is a rotation with respect to z followed by a rotation with respect to y.



Each of these rules has an exact mirror image variant which we do not depict. Note that the zig rule is just an ordinary rotation that we saw earlier, while the zig-zag and zig-zig rules are compositions of two separate rotations. The zig-zag case just rotates about x's parent twice, which is exactly the same thing that we argued doesn't always work. The magic idea that makes splay trees work is the zig-zag step. Instead of rotating about x's parent twice, it first rotates about x's grandparent, and then about x's parent. This tiny subtle change turns out to make

all the difference in making it into an efficient algorithm.

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. 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. Here are some examples:



Observe that unlike in the naive move-to-root algorithm, splaying 0 here actually decreased the height of the tree, making it more balanced.

3 Standard BST Operations Using Splaying

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 ever splaying.)

Split Say we are given a key x which exists in the tree, and we want to split the tree into two trees, such that the first contains all keys in the tree that are at most x, and the second contains all keys that are larger than x. To do so, we splay the node containing x to the root, then remove its right subtree.

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.

4 Analysis of Splaying

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.)

4.1 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 defined as

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

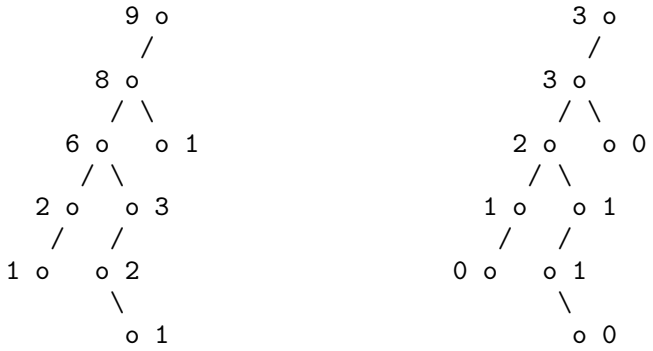
Now we define the *rank of a node* x as

$$r(x) = \lfloor \log_2(s(x)) \rfloor$$

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. (Note however that the ranks could be negative if the weights are less than 1.)

5 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

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

Proof. Throughout the proof, keep in mind that we are counting splay steps as our cost. To do the proof, we are going to analyze each splay step individually. We will show the following lemma:

Lemma: Cost of a splay step

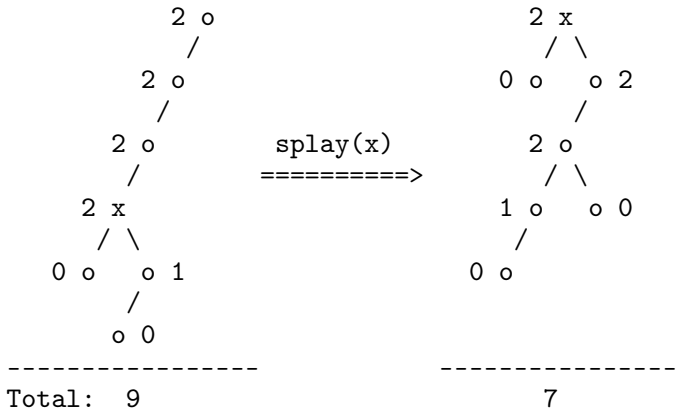
The amortized cost of a zig-zag or zig-zig splay step is $3(r(z) - r(x))$. The cost of the zig splay step is $3(r(y) - r(x)) + 1$.

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 $r'(x)$ is the rank of x after the splay step, and $r(x)$ represents the rank of x before the splay 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, we see 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.

We will need one more lemma, called the Rank Rule.

Lemma: The 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:

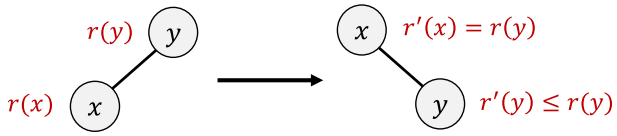
```
   x
  / \        Then x >= r+1
 r   r
```

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. 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 need to show an amortized cost of at most $3(r(y)-r(x))+1$ to pay for the step.



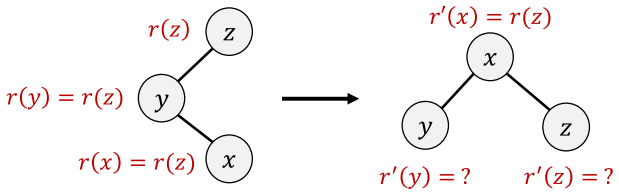
The actual cost is 1. Now, since x is the new root, $r'(x) = r(y)$. Furthermore, since y used to be the root, its rank can not have increased, so $r'(y) \leq r(y)$. So the sum of the new ranks of x and y is at most $2r(y)$, and the sum of their old ranks is $r(y)+r(x)$, hence the difference in potential is at most

$$2r(y)-(r(y)+r(x)) = r(y)-r(x).$$

Hence the amortized cost is at most $(r(y)-r(x))+1 \leq 3(r(y)-r(x))+1$.

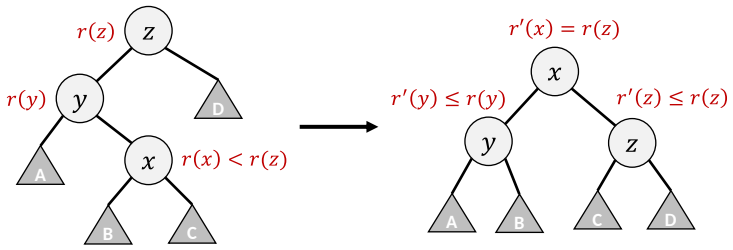
The Zig-zag Case: We'll split it further into 2 cases:

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



We know that $r'(x) = r(z)$, but by the rank rule, either $r'(y) < r(z)$ or $r'(z) < r(z)$. Therefore the sum of ranks is at least one lower than it was before the splay, so the potential has decreased by at least one. This pays for the actual cost of 1 of the splay, and hence the amortized cost of this step is zero, which is at most $3(r(z) - r(x)) = 0$.

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



First, observe that after the splay step, y and z both have a strict subset of their previous descendants, so their ranks can't have increased, i.e., $r'(y) \leq r(y)$ and $r'(z) \leq r(z)$. Since $r(x) < r(z)$, and since the ranks are integers, it must be true that $r(z) \geq r(x) + 1$.

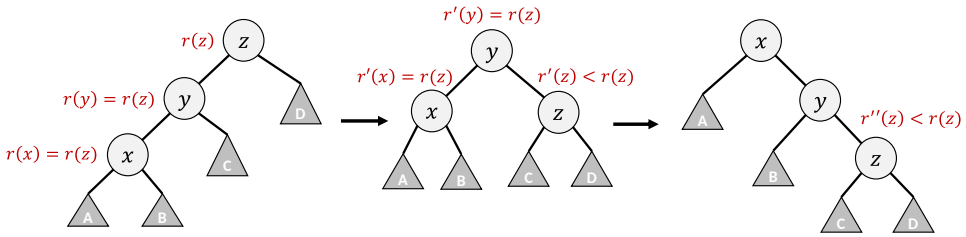
Now we write the difference in potentials as

$$\begin{aligned}
\Delta\Phi &= r'(z) + r'(y) + r'(x) - r(z) - r(y) - r(x) \\
&\leq r(z) + r(y) + r(z) - r(z) - r(y) - r(x) && \text{(use } r'(y) \leq r(y),\ r'(z) \leq r(z),\ r'(x) = r(z)\text{)} \\
&\leq 2r(z) - r(z) - r(x) && \text{(cancel and simplify)} \\
&\leq 2r(z) - (r(x) + 1) - r(x) && \text{(use } r(z) \geq r(x) + 1\text{)} \\
&= 2(r(z) - r(x)) - 1 && \text{(simplify)}
\end{aligned}$$

Adding the actual cost of 1, we get an amortized cost of at most $2(r(z) - r(x)) \leq 3(r(z) - r(x))$.

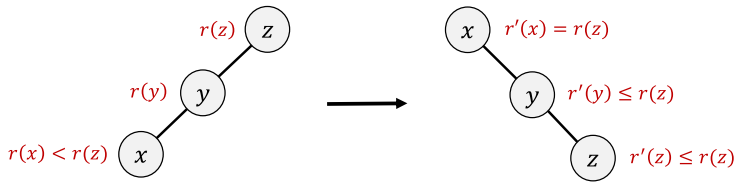
The Zig-zig Case: Again, we split into two cases.

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



Unlike the previous cases where we looked at only the starting and ending configurations, we will get some useful information from looking at the intermediate rotation. Since y is the root in the middle configuration, $r'(y) = r(z)$, and since x still has the same descendants it had before, $r'(x) = r(x) = r(z)$. By the rank rule, it must be true that $r'(z) < r(z)$. Then we look at the final configuration and notice that z still has the exact same descendants so $r''(z) = r'(z) < r(z)$. Therefore the total potential must decrease by at least 1, which pays for the actual cost of the splay, so we have $0 \leq 3(r(z) - r(x)) = 0$ as required.

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



Once again since we assume $r(x) < r(z)$ and the ranks are integers, we have $r(z) \geq r(x) + 1$. We always have $r'(z) \leq r(z)$ and $r'(y) \leq r(z)$ and $r'(x) = r(z)$. Since y started as an ancestor of x, we also have $r(y) \geq r(x)$ so we can write

$$\begin{aligned}
\Delta\Phi &= r'(z) + r'(y) + r'(x) - r(z) - r(y) - r(x) \\
&\leq r(z) + r(z) + r(z) - r(z) - r(y) - r(x) && \text{(use } r'(z) \leq r(z),\ r'(y) \leq r(z),\ r'(x) = r(z)\text{)} \\
&\leq 3r(z) - (r(x) + 1) - r(x) - r(x) && \text{(use } r(z) \geq r(x) + 1 \ \&\ r(y) \geq r(x)\text{)} \\
&= 3(r(z) - r(x)) - 1 && \text{(simplify)}
\end{aligned}$$

Adding the actual cost of 1, the amortized cost for this step is at most $3(r(z) - r(x))$.

So in every splay step, we have shown that the amortized cost is at most $3(r(z) - r(x))$ (plus the extra $+1$ needed for the zig case at the very end). Thus these amortized costs telescope giving an amortized number of splay steps of $3(r(t) - r(x)) + 1$. □

6 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 1: 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. □

7 Improvement and Applications of the Access Lemma

Optional content — Will not appear on the homeworks or the exams

The access lemma can be improved by using a different potential function. Instead of $r(x) = \lfloor \log_2(s(x)) \rfloor$ we use the *real rank*, defined as $rr(x) = \log_2(s(x))$. The potential function in this case will be the sum of the real ranks of all the nodes.

Lemma: Strong Access Lemma

Take any tree T with root t, and any weights $w(\cdot)$ on the nodes. Suppose you splay node x. Then using the real rank based potential function we have:

$$\text{amortized number of ROTATIONS } \leq 3(rr(t) - rr(x)) + 1.$$

Note that this lemma bounds the number of rotations, not splay steps. And the bound is almost the same as the access lemma. So this is roughly a factor of two stronger than the access lemma. We will not prove this lemma here.

One of the things that distinguishes splay trees from other forms of balanced trees is the fact that they are provably more efficient on 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: 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: 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: 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: 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: Dynamic Finger Theorem

Incorporation the notation from the Static Finger Theorem, the cost of an access sequence is bounded by

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