CMSC 651 Advanced Algorithms — Lecture 1
Lecturer: Samir Khuller

Original notes by Hsiwei Yu.

1 Overview of Course

The course will cover many different topics. We will start out by studying various combinatorial algorithms together with techniques for analyzing their performance. We will also study linear programming and understand the role that it plays in the design of combinatorial algorithms. We will then go on to the study of NP-completeness and NP-hard problems, along with polynomial time approximation algorithms for these hard problems.

1.1 Amortized Analysis

Typically, most data structures provide absolute guarantees on the worst case time for performing a single operation. We will study data structures that are unable to guarantee a good bound on the worst case time *per* operation, but will guarantee a good bound on the *average* time it takes to perform an operation. (For example, a sequence of m operations will be guaranteed to take $m \times T$ time, giving an average, or *amortized time* of T per operation. A single operation could take time more than T.)

Example 1 (not covered in class): Consider a STACK with the following two operations: `Push(x)` pushes item x onto the stack, and `M-POP(k)` pop's the top-most k items from the stack (if they exist). Clearly, a single `M-POP` operation can take more than $O(1)$ time to execute, in fact the time is $\min(k, s)$ where s is the stack-size at that instant.

It should be evident, that a sequence of n operations however runs only in $O(n)$ time, yielding an "average" time of $O(1)$ per operation. (Each item that is pushed into the stack can be popped at most once.)

There are fairly simple formal *schemes* that formalize this very argument. The first one is called the **accounting method**. We shall now assume that our computer is like a vending machine. We can put in \$ 1 into the machine, and make it run for a *constant* number of steps (we can pick the constant). Each time we push an item onto the stack we use \$ 2 in doing this operation. We spend \$ 1 in performing the push operation, and the other \$ 1 is stored *with* the item on the stack. (This is only for analyzing the algorithm, the actual algorithm does not have to keep track of this money.) When we execute a multiple pop operation, the work done for each pop is paid for by the money stored with the item itself.

The second scheme is the **potential method**. We define the potential Φ for a data structure D. The potential maps the current "state" of the data structure to a real number, based on its current configuration.

In a sequence of operations, the data structure transforms itself from state D_{i-1} to D_i (starting at D_0). The *real cost* of this transition is c_i (for changing the data structure). The potential function satisfies the following properties:

- $\Phi(D_i) \geq 0$.
- $\Phi(D_0) = 0$

We define the *amortized cost* to be $c'_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, where c_i is the true cost for the i^{th} operation.

Clearly,

$$\sum_{i=1}^{n} c'_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

Thus, if the potential function is always positive and $\Phi(D_0) = 0$, then the amortized cost is an upper bound on the real cost. Notice that even though the cost of each individual operation may not be constant, we may be able to show that the cost over any sequence of length n is $O(n)$. (In most applications, where data structures are used as a part of an algorithm; we need to use the data structure for over a sequence

of operations and hence analyzing the data structure's performance over a sequence of operations is a very reasonable thing to do.)

In the stack example, we can define the potential to be the number of items on the stack. (Exercise: work out the amortized costs for each operation to see that Push has an amortized cost of 2, and M-Pop has an amortized cost of 1.)

Example 2: The second example we consider is a k-bit counter. We simply do INCREMENT operations on the k-bit counter, and wish to count the total number of bit operations that were performed over a sequence of n operations. Let the counter be $= < b_k b_{k-1} \dots b_1 >$. Observe that the least significant bit b_1, changes in every step. Bit b_2 however, changes in every alternate step. Bit b_3 changes every 4^{th} step, and so on. Thus the total number of bit operations done are:

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \dots \leq 2n.$$

A potential function that lets us prove an amortized cost of 2 per operation, is simply the number of 1's in the counter. Each time we have a cascading carry, notice that the number of 1's decrease. So the potential of the data structure falls and thus pays for the operation. (Exercise: Show that the amortized cost of an INCREMENT operation is 2.)

Original notes by Mark Carson.

2 Splay Trees

Read [21] Chapter 4.3 for Splay trees stuff.

Splay trees are a powerful data structure developed by Sleator and Tarjan [20], that function as search trees without any explicit balancing conditions. They serve as an excellent tool to demonstrate the power of amortized analysis.

Our basic operation is: $splay(k)$, given a key k. This involves two steps:

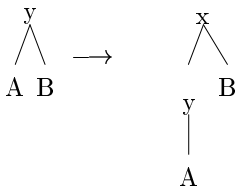
1. Search through out the tree to find node with key k.
2. Do a series of rotations (a splay) to bring it to the top.

The first of these needs a slight clarification:
If k is not found, grab the largest node with key less than k instead (then splay this to the top.)

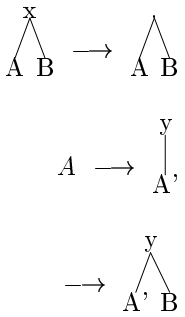
2.1 Use of Splay Operations

All tree operations can be simplified through the use of splay:

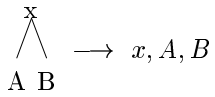
1. $Access(x)$ - Simply splay to bring it to the top, so it becomes the root.
2. $Insert(x)$ - Run $splay(x)$ on the tree to bring y, the largest element less than x, to the top. The insert is then trivial:



3. $Delete(x)$ - Run $splay(x)$ on the tree to bring x to the top. Then run $splay(x)$ again in x's left subtree A to bring y, the largest element less than x, to the top of A. y will have an empty right subtree in A since it is the largest element there. Then it is trivial to join the pieces together again without x:



4. *Split*(x) - Run *splay*(x) to bring x to the top and split.



Thus with at most 2 splay operations and constant additional work we can accomplish any desired operation.

2.2 Time for a Splay Operation

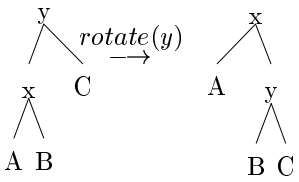
How much work does a splay operation require? We must:

1. Find the item (time dependent on depth of item).
2. Splay it to the top (time again dependent on depth)

Hence, the total time is $O(2\times$ depth of item).

How much time do k splay operations require? The answer will turn out to be $O(k \log n)$, where n is the size of the tree. Hence, the amortized time for one splay operation is $O(\log n)$.

The basic step in a splay operation is a *rotation*:



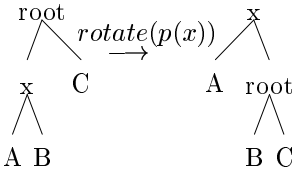
Clearly a rotation can be done in $O(1)$ time. (Note there are both left and right rotations, which are in fact inverses of each other. The sketch above depicts a right rotation going forward and a left rotation going backward. Hence to bring up a node, we do a right rotation if it is a left child, and a left rotation if it is a right child.)

A splay is then done with a (carefully-selected) series of rotations. Let $p(x)$ be the parent of a node x. Here is the splay algorithm:

Splay Algorithm:

while $x \neq root$ **do**

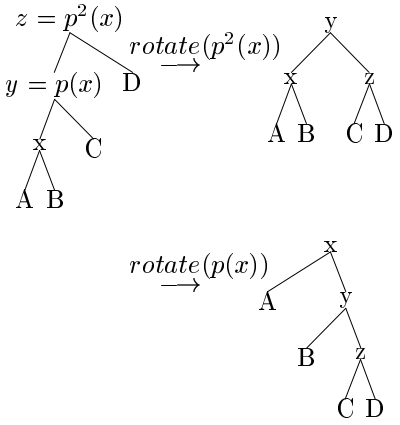
if $p(x) = root$ **then** *rotate*($p(x)$)



else if both x and $p(x)$ are left (resp. right) children, **do** right (resp. left) rotations: **begin**

rotate($p^2(x)$)
rotate($p(x)$)

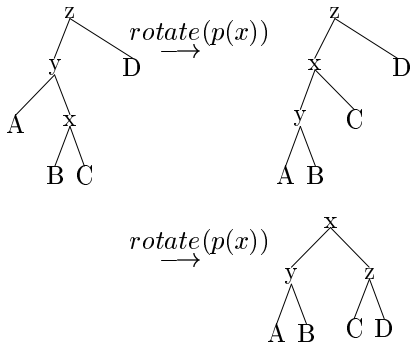
end



else /* x and $p(x)$ are left/right or right/left children */ **begin**

$rotate(p(x))$
$rotate(p(x))$ /* note this is a new $p(x)$ */

end



fi

od

When will accesses take a long time? When the tree is long and skinny.
What produces long skinny trees?

- a series of inserts in ascending order 1, 3, 5, 7, ...
- each insert will take $O(1)$ steps – the splay will be a no-op.
- then an operation like $access(1)$ will take $O(n)$ steps.
- *HOWEVER* this will then result in the tree being balanced.
- Also note that the first few operations were very fast.

Therefore, we have this general idea – splay operations tend to balance the tree. Thus any long access times are "balanced" (so to speak) by the fact the tree ends up better balanced, speeding subsequent accesses.

In potential terms, the idea is that as a tree is built high, its "potential energy" increases. Accessing a deep item releases the potential as the tree sinks down, paying for the extra work required.

Original notes by Mark Carson.

3 Amortized Time for Splay Trees

Read [21] Chapter 4.3 for Splay trees stuff.

Theorem 3.1 *The amortized time of a splay operation is* $O(\log n)$.

To prove this, we need to define an appropriate potential function.

Definition 3.2 *Let* s *be the splay tree. Let* $d(x)$ *= the number of descendants of* x *(including* x*). Define the rank of* $x, r(x) = \log d(x)$ *and the potential function*

$$\Phi(s) = \sum_{x \epsilon s} r(x).$$

Thus we have:

- $d(\text{leaf node}) = 1$, $d(\text{root}) = n$
- $r(\text{leaf node}) = 0$, $r(\text{root}) = \log n$

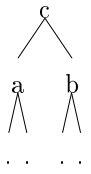
Clearly, the better balanced the tree is, the lower the potential Φ is. (You may want to work out the potential of various trees to convince yourself of this.)

We will need the following lemmas to bound changes in Φ.

Lemma 3.3 *Let* c *be a node in a tree, with* a *and* b *its children. Then* $r(c) > 1 + min(r(a), r(b))$.

Proof:

Looking at the tree, we see $d(c) = d(a) + d(b) + 1$. Thus we have $r(c) > 1 + min(r(a), r(b))$.



□

We apply this to

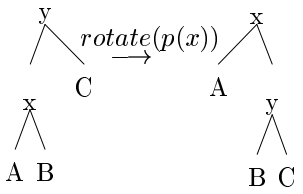
Lemma 3.4 (Main Lemma) *Let* $r(x)$ *be the rank of* x *before a rotation (a single splay step) bringing* x *up, and* $r'(x)$ *be its rank afterward. Similarly, let* s *denote the tree before the rotation and* s' *afterward. Then we have:*

1. $r'(x) \geq r(x)$
2. *If* $p(x)$ *is the root then* $\Phi(s') - \Phi(s) < r'(x) - r(x)$
3. *if* $p(x) \neq$ *root then* $\Phi(s') - \Phi(s) < 3(r'(x) - r(x)) - 1$

Proof:

1. Obvious as x gains descendants.

2. Note in this case we have

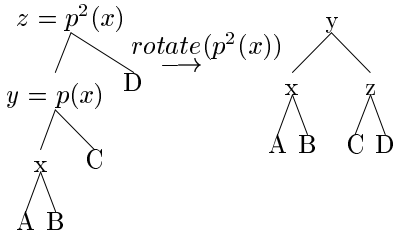


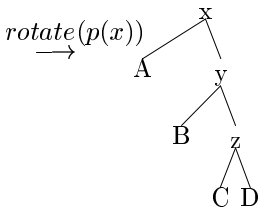
   so that clearly $r'(x) = r(y)$. But then since only x and y change rank in s',

$$\begin{aligned}\Phi(s') - \Phi(s) &= (r'(x) - r(x)) + (r'(y) - r(y)) \\ &= r'(y) - r(x) < r'(x) - r(x)\end{aligned}$$

   since clearly $r'(y) < r'(x)$.

3. Consider just the following case (the others are similar):





   Let r represent the ranks in the initial tree s, r'' ranks in the middle tree s'' and r' ranks in the final tree s'. Note that, looking at the initial and final trees, we have

$$r(x) < r(y)$$

   and

$$r'(y) < r'(x)$$

   so

$$r'(y) - r(y) < r'(x) - r(x)$$

   Hence, since only x, y and z change rank,

$$\begin{aligned}\Phi(s') - \Phi(s) &= (r'(x) - r(x)) + (r'(y) - r(y)) + (r'(z) - r(z)) \\ &< 2(r'(x) - r(x)) + (r'(z) - r(z))(*)\end{aligned}$$

   Next from Lemma 1, we have $r''(y) > 1 + min(r''(x), r''(z))$. But looking at the middle tree, we have

$$\begin{aligned}r''(x) &= r(x) \\ r''(y) &= r'(x)(= r(z)) \\ r''(z) &= r'(z)\end{aligned}$$

so that

$$r(z) = r'(x) > 1 + min(r(x), r'(z))$$

Hence, either we have

$$r'(x) > 1 + r(x), \quad so \quad r'(x) - r(x) > 1$$

or

$$r(z)) > 1 + r'(z), \quad so \quad r'(z) - r(z) < -1$$

In the first case, since

$$r'(z) < r(z) => r'(z) - r(z) < 0 < r'(x) - r(x) - 1$$

clearly

$$\Phi(s') - \Phi(s) < 3(r'(x) - r(x)) - 1$$

In the second case, since we always have $r'(x) - r(x) > 0$, we again get

$$\begin{aligned} \Phi(s') - \Phi(s) &< 2(r'(x) - r(x)) - 1 \\ &< 3(r'(x) - r(x)) - 1 \end{aligned}$$

□

We will apply this lemma now to determine the amortized cost of a splay operation. The splay operation consists of a series of splay steps (rotations). For each splay step, if s is the tree before the splay, and s' the tree afterwards, we have

$$at = rt + \Phi(s') - \Phi(s)$$

where at is the amortized time, rt the real time. In this case, $rt = 1$, since a splay step can be done in constant time.
Note: Here we have scaled the time factor to say a splay step takes one time unit. If instead we say a splay takes c time units for some constant c, we change the potential function Φ to be

$$\Phi(s) = c \sum_{x \epsilon s} r(x).$$

Consider now the two cases in Lemma 2. For the first case (x a child of the root), we have

$$\begin{aligned} at = 1 + \Phi(s') - \Phi(s) &< 1 + (r'(x) - r(x)) \\ &< 3\Delta r + 1 \end{aligned}$$

For the second case, we have

$$\begin{aligned} at = 1 + \Phi(s') - \Phi(s) &< 1 + 3(r'(x) - r(x)) - 1 \\ &= 3(r'(x) - r(x)) \\ &= 3\Delta r \end{aligned}$$

Then for the whole splay operation, let $s = s_0, s_1, s_2, \ldots, s_k$ be the series of trees produced by the sequence of splay steps, and $r = r_0, r_1, r_2, \ldots, r_k$ the corresponding rank functions. Then the total amortized cost is

$$at = \sum_{i=0}^{k} at_i < 1 + \sum_{i=0}^{k} 3\Delta r_i$$

But the latter series telescopes, so

$$at < 1 + 3(\ final\ rank\ of\ x - \ initial\ rank\ of\ x)$$

Since the final rank of x is $\log n$, we then have

$$at < 3 \log n + 1$$

as desired.

3.1 Additional notes

1. (Exercise) The accounting view is that each node x stores $r(x)$ dollars [or some constant multiple thereof]. When rotates occur, money is taken from those nodes which lose rank to pay for the operation.

2. The total time for k operations is actually $O(k \log m)$, where m is the largest size the tree ever attains (assuming it grows and shrinks through a series of inserts and deletes).

3. As mentioned above, if we say the real time for a rotation is c, the potential function Φ is

$$\Phi(s) = \sum_{x \in s} cr(x)$$