Recap of this week’s lectures:

- Amortization (Binary Counter, Dictionary)
- Splay Trees

**Binary Counter Revisited:** Suppose we are incrementing a binary counter, but instead of each bit flip costing 1, suppose flipping the $i^{th}$ bit costs us $2^i$. (Flipping the lowest order bit $A[0]$ costs $2^0 = 1$, the next higher order bit $A[1]$ costs $2^1 = 2$, the next costs $2^2 = 4$, etc.) What is the amortized cost per operation for a sequence of $n$ increments, starting from zero?

**Solution:** $O(\log n)$. The idea is simple. We flip $A[0]$ each time, so pay $n$ over $n$ operations. We flip $A[1]$ every other time, so pay $\leq 2 \times n/2 = n$ over $n$ operations, and so on, until $A[\lceil \log_2 n \rceil]$ which gets flipped once for a cost of at most $n$. Hence $O(n \log n)$ in total, or $O(\log n)$ per operation.
(Bonus) Another Dictionary Data Structure: A “dictionary” data structure supports fast insert and lookup operations into a set of items. Note that a sorted array is good for lookups (binary search takes time only $O(\log n)$) but bad for inserts (takes linear time), and a linked list is good for inserts (takes constant time) but bad for lookups (takes linear time). Here is a simple method that takes $O(\log^2 n)$ search time and $O(\log n)$ amortized cost per insert.

Here, we keep a collection of arrays, where array $i$ has size $2^i$. Each array is either empty or full, and each is in sorted order. However, there will be no relationship between the items in different arrays. The issue of which arrays are full and which are empty is based on the binary representation of the number of items we are storing. For example, if we had 11 items (where $11 = 1 + 2 + 8$), then the arrays of size 1, 2, and 8 would be full and the rest empty, and the data structure might look like this:

\[
\begin{align*}
A0: & \quad [5] \\
A1: & \quad [4, 8] \\
A2: & \quad \text{empty} \\
A3: & \quad [2, 6, 9, 12, 13, 16, 20, 25]
\end{align*}
\]

**Lookups.** How would you do a lookup in $O(\log^2 n)$ worst-case time?

**Solution:** Just do binary search in each occupied array. In the worst case, this takes time $O(\log(n) + \log(n/2) + \log(n/4) + \ldots + 1) = O(\log^2 n)$.

**Inserts.** How would you do inserts? Suppose you wanted to insert an element, you will have 12 items and $12 = 8 + 4$, you want to have two full arrays in $A_2$ and $A_3$ and the rest empty. Suggest a way that, if you insert an element 11 into the example above, gives:

\[
\begin{align*}
A0: & \quad \text{empty} \\
A1: & \quad \text{empty} \\
A2: & \quad [4, 5, 8, 11] \\
A3: & \quad [2, 6, 9, 12, 13, 16, 20, 25]
\end{align*}
\]

(Hint: merge arrays!)

**Solution:** Create an array of size 1 that just has this inserted number in it. We now look to see if $A0$ is empty. If so we make the new array be $A0$. If not we merge our array with $A0$ to create a new array (which in the example would be [5, 11]) and look to see if $A1$ is empty. If $A1$ is empty, we make this array be $A1$. If not, we merge this with $A1$ to create a new array and check to see if $A2$ is empty, and so on.

**Cost of Inserts:** Suppose the cost of creating an array of length 1 costs 1, and merging two arrays of length $m$ costs $2m$. So, the above insert had cost $1 + 2 + 4$. Inserting another element would cost 1, and the next insert would cost $1 + 2$.

What is the amortized cost of $n$ inserts?
**Solution:** With this cost model defined above, it’s exactly the same as the binary counter with cost $2^k$ for the $k^{th}$ bit. So the amortized cost is $O(\log n)$.
Cyclic Splaying: Starting from a tree $T_0$ of $n$ nodes a sequence of $\ell \geq 1$ splay operations is done. It turns out that the initial tree $T_0$ and the final tree $T_\ell$ are the same. Let $k$ be the number of distinct nodes splayed in this sequence. (Clearly $k \leq \ell$.) Below is an example where $k = \ell = 4$ and $n = 6$.

(a) Use some setting of node weights to show that the average number of splaying steps in this cycle (i.e., the average per splay operation) is at most $1 + 3 \log_2 n$. Make use of the Access Lemma for splay trees covered in lecture yesterday.

(b) (Extra material, for you to do at home.) Now use a different setting of node weights to show that the average number of splaying steps in this cycle (per splay operation) is at also most $1 + 3 \log_2 k$.

Solution: Start with equation (2) of the lecture on amortized analysis:

$$\sum_i c_i = \left( \sum_i ac_i \right) + \Phi(s_{\text{initial}}) - \Phi(s_{\text{final}})$$

Note that since the initial and final trees are the same, the initial and final potentials are equal. So the above equation shows that the total cost is the same as the total amortized cost.

By the Access Lemma, the amortized number of splaying steps in each splay is $3(r(t) - r(x)) + 1 \leq 3r(t) + 1$. (Since all the weights are positive.) So the total amortized cost, which equals the total actual number of splay steps, is at most $\ell(3r(t) + 1)$, and the average number of splay steps per splay is at most $(3r(t) + 1)$.

a. Set all the weights of each node to 1. We know $r(t)$, or the rank of the root, is $\lfloor \log(s(t)) \rfloor$, where $s(t) = \sum_{y \in T(t)} w(y)$. Since all $n$ nodes are in the subtree rooted at the root, $s(t) = n$ and $r(t) = \lfloor \log(n) \rfloor$. We also know $r(x) \geq 0$, so $3(r(t) - r(x)) + 1 \leq 3\lfloor \log(n) \rfloor + 1$.

b. Let the weights of the $k$ elements we access be 1 and the others be $\epsilon > 0$. The rank of the root is then $\lfloor \log(k + (n-k)\epsilon) \rfloor$. If we choose $\epsilon$ to be small enough so that $(n-k)\epsilon$ is smaller than 1, then $\lfloor \log(k + (n-k)\epsilon) \rfloor = \lfloor \log(k) \rfloor \leq \log(k)$. This is because the function $f(x) = \lfloor \log(x) \rfloor$ only changes when $x$ crosses an integer value. Again by the Access Lemma, we know the average number of splaying steps is $\leq 3(\log(k) - r(x)) + 1 \leq 3 \log(k) + 1$. 
Balls in Bins There are $n$ balls and an infinite number of bins. A bin can have 0 or more balls in it. A move consists taking all the balls of some bin and putting them into distinct bins. The cost of a move is the number of balls moved. Define the potential of a state of this system as the sum of the potentials of all the bins. The potential of a bin with $k$ balls in it is:

$$\Phi(k) = \max(0, k - z)$$

Where for convenience $z = \lfloor \sqrt{n} \rfloor$.

1. Prove that the amortized cost of a move is at most $2z$.

   **Solution:** Consider one move in which $k$ is the number of balls in the bin vacated. Let $ac$ be the amortized cost of the move. Let $X$ be the set of bins whose count increases where the final number of balls is $> z$. (These are precisely the bins causing the potential to increase.) Breaking the analysis into two cases we get:

   $$ac = \begin{cases} 
   k + |X| & \text{if } k \leq z \\
   k - (k - z) + |X| & \text{otherwise}
   \end{cases}$$

   Clearly $|X| \leq k$, so in the first case $ac \leq 2k \leq 2z$. We also know that each bin of $X$ has at least $z + 1$ balls in it at the end. So we know that $|X| \times (z + 1) \leq n$. So $|X| \leq z$. (If $|X| \geq z + 1$ then $|X| \times (z + 1) > n$, a contradiction.) So in the 2nd case we have:

   $$ac = k - (k - z) + |X| \leq k - (k - z) + z = 2z$$

   which finishes the proof. 

2. Show a sequence of moves which achieves $\Omega(z)$ per move.

   **Solution:** Let $k = \lfloor \sqrt{2n} \rfloor$. With this choice $(k - 1) + \ldots + 1 \leq n$. Move balls around until you’ve created a bin with $k$ balls, one with $k - 1$ balls, etc., down to a bin with 1 ball.

   Now take the bin with $k$ balls in it, and move those balls to a bin with $k - 1$ balls, $k - 2$ balls, etc., plus one ball to an empty bin. The new arrangement is isomorphic to the previous one. So this move, which costs $k$, can be repeated forever.