1 Amortized Analysis

- Last time we discussed amortized analysis of data structures
  - A way of expressing that even though the worst-case performance of an operation can be bad, the total performance of a sequence of operations cannot be too bad.

- One way of thinking of amortized time is as being an “average”: If any sequence of \( n \) operations takes less than \( T(n) \) time, the amortized time per operation is \( T(n)/n \).

- We formally defined amortized time using the idea that we over-charge some operations and store the over-charge as credits/potential that can then help pay for later operations (*potential method*)
  - Consider performing \( n \) operations on an initial data structure \( D_0 \)
  - \( D_i \) is data structure after \( i \)th operation.
  - \( c_i \) is actual cost (time) of \( i \)th operation.
  - Potential function: \( \Phi : D_i \rightarrow \mathbb{R} \)
  - \( \tilde{c}_i \) amortized cost of \( i \)th operation: \( \tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \)
  - Given \( \Phi(D_0) = 0 \) and \( \Phi(D_i) \geq 0 \): \( \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \tilde{c}_i \)

- We also discussed two examples of amortized analysis
  - Stack with **MULTIPOP** (\( O(n) \) worst-case, \( O(1) \) amortized).
  - **INCREMENT** on binary counter (\( O(\log n) \) worst-case, \( O(1) \) amortized).

In both cases we could argue for \( O(1) \) amortized performance without actually doing potential calculation—we just think about potential/credits as being distributed on certain parts of the data structure and let operations put and take credits while maintaining some invariant (**accounting method**).
2 Splay trees

- We have previously discussed binary search trees and how they can be kept balanced \(O(\log n)\) height) during insert and delete operations (red-black trees).
  - Rebalancing rather complicated
  - Extra space used for the color of each node

- We also discussed skip lists which are a lot simpler than red-black trees
  - Only guarantee \(O(\log n)\) expected performance
  - No extra information is used for rebalance information though

- Splay trees are search trees that “magically” balance themselves (no rebalance information is stored) and have amortized \(O(\log n)\) performance.

- Recall search trees:
  - Binary tree with elements in nodes
  - If node \(v\) holds element \(e\) then
    - all elements in left subtree \(< e\)
    - all elements in left subtree \(> e\)

- Splay tree:
  - Normal (possibly unbalanced) search tree \(T\)
  - All operations implemented using one basic operation, \textsc{Splay}:

  \[
  \text{SPLAY}(x, T) \text{ searches for } x \text{ in } T \text{ and reorganizes tree such that } x \text{ (or min element } > x \text{ or max element } < x \text{) is in root}
  \]

  - \textsc{Search}(\(x, T\)): \textsc{Splay}(\(x, T\)) and inspect root
  - \textsc{Insert}(\(x, T\)): \textsc{Splay}(\(x, T\)) and create new root with \(x\)
- **DELETE**(\(x, T\)):
  - **SPLAY**(\(x, T\)) and remove root → tree falls into \(T_1\) and \(T_2\).
  - **SPLAY**(\(x, T_1\))
  - Make \(T_2\) right son of new root of \(T_1\) after splay

\[
\text{splay(x,T)} \\
\text{splay(x,T1)}
\]

\[
\text{All operations perform } O(1) \text{ SPLAY’s and use } O(1) \text{ extra time.}
\]

\[
O(\log n) \text{ amortized SPLAY gives } O(\log n) \text{ amortized bound on all operations.}
\]

- Implementation of **SPLAY**:
  - Search for \(x\) like in normal search tree
  - Repeatedly rotate \(x\) up until it becomes the root.
    We distinguish between three cases:
    1. \(x\) is child of root (no grandparent): **rotate**(\(x\))

\[
\text{e.g.}
\]

\[
\text{2. } x \text{ has parent } y \text{ and grandparent } z \text{ and both } x \text{ and } y \text{ left (right) children: } \text{rotate}(y) \text{ followed by } \text{rotate}(x)
\]

\[
\text{e.g.}
\]
3. $x$ has parent $y$ and grandparent $z$ and one of $x$ and $y$ is a left child and the other is a right child: \texttt{rotate}(x) \texttt{ followed by rotate}(x)

e.g.

\begin{center}
\begin{tikzpicture}
  \node (x) {x} child {node (z) {z} child {node (y) {y} child {node (a) {a} child {node (b) {b}}} child {node (c) {c}}} child {node (d) {d}}}} child {node (e) {e} child {node (f) {f} child {node (g) {g}}} child {node (h) {h}}};
\end{tikzpicture}
\end{center}

- Note:
  
  - A SPLAY can take $O(n)$ worst-case time (very unbalanced tree)
  - But Splay trees somehow seem to stay nicely balanced

Examples: SPLAY(1, $T$)

\begin{center}
\begin{tikzpicture}
  \node (x) {x} child {node (y) {y} child {node (z) {z} child {node (a) {a} child {node (b) {b}}} child {node (c) {c}}} child {node (d) {d}}}} child {node (e) {e} child {node (f) {f} child {node (g) {g}}} child {node (h) {h}}};
\end{tikzpicture}
\end{center}

$SPLAY(5, T)$

\begin{center}
\begin{tikzpicture}
  \node (x) {x} child {node (y) {y} child {node (z) {z} child {node (a) {a} child {node (b) {b}}} child {node (c) {c}}} child {node (d) {d}}}} child {node (e) {e} child {node (f) {f} child {node (g) {g}}} child {node (h) {h}}};
\end{tikzpicture}
\end{center}
• Analysis:
  - We will use **accounting method** to show that all operations (SPLAY) takes \(O(\log n)\) amortized time.
    * We will imagine that each node in tree has credits on it
    * We will use some credits to pay for (part of) rotations during a splay
    * We will see that we only have to place \(O(\log n)\) new credits (on root) when performing an **Insert** or **Delete**
  - Note that we will ignore cost of searching for \(x\), since the rotations cost at least as much as the search (⇒ if we can bound amortized rotation cost we also bound search cost).
  - Let \(T(x)\) be tree rooted at \(x\). We will maintain the **credit invariant** that each node \(x\) holds \(\mu(x) = \lfloor \log |T(x)| \rfloor \) credits.
  - We will prove the following lemma:

    | Less than or equal to \(3(\mu(T) - \mu(x) + O(1))\) credits are needed to perform SPLAY \((x, T)\) operation and maintain credit invariant

  - Using this lemma we get that a SPLAY operation uses at most \(3\lfloor \log n \rfloor + O(1) = O(\log n)\) credits (time).
  - As an **Insert** or a **Delete** requires us to insert at most \(O(\log n)\) extra credits (on the root) more than the ones used on the SPLAY, we get the \(O(\log n)\) amortized bound.

• Proof of lemma:
  - Let \(\mu\) and \(\mu'\) be the value of \(\mu\) before and after a rotate operation in case 1, 2, or 3.
  - During a SPLAY operation we perform a number of, say \(k \geq 0\), case 2 and 3 operations and possibly a case 1 operation.
  - Next time we will show that the cost of one operation is:
    * Case 1: \(3(\mu'(x) - \mu(x) + O(1))\)
    * Case 2: \(3(\mu'(x) - \mu(x))\)
    * Case 3: \(3(\mu'(x) - \mu(x))\)

\[
\downarrow
\]
When we sum over all \(\leq k + 1\) operations in a splay we get \(3(\mu(T) - \mu(x) + O(1))\) where \(\mu(x)\) is the number of credits on \(x\) before the SPLAY.
Note that it is important that we only have the \(O(1)\) term in case 1.

• Case 1:
  - We have: \(\mu'(x) = \mu(y), \mu'(y) \leq \mu'(x)\) and all other \(\mu\)'s are unchanged.
  - To maintain invariant we use: \[
\begin{align*}
\mu'(x) + \mu'(y) - \mu(x) - \mu(y) &= \mu'(y) - \mu(x) \\
&\leq \mu'(x) - \mu(x) \\
&\leq 3(\mu'(x) - \mu(x))
\end{align*}
\]
  - To do actual rotation we use \(O(1)\) credits.
Case 2:

- We have $\mu'(x) = \mu(z)$, $\mu'(y) \leq \mu'(x)$, $\mu'(z) \leq \mu'(x)$, $\mu(y) \geq \mu(x)$ and all other $\mu$'s are unchanged.

- To maintain invariant we use:

$$\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z) = \mu'(y) + \mu'(z) - \mu(x) - \mu(y)$$
$$= (\mu'(y) - \mu(x)) + (\mu'(z) - \mu(y))$$
$$\leq (\mu'(x) - \mu(x)) + (\mu'(x) - \mu(x))$$
$$= 2(\mu'(x) - \mu(x))$$

- This means that we can use the remaining $\mu'(x) - \mu(x)$ credits to pay for rotation, unless $\mu'(x) = \mu(x)$ (can happen since $\mu(x) = \lceil \log |T(x)| \rceil$).

- We will show that if $\mu'(x) = \mu(x)$ then $\mu'(x) + \mu'(y) + \mu'(z) < \mu(x) + \mu(y) + \mu(z)$ which means that the operation actually releases credits we can use for the rotation:

* Assume $\mu'(x) = \mu(x)$ and $\mu'(x) + \mu'(y) + \mu'(z) \geq \mu(x) + \mu(y) + \mu(z)$

* We have $\mu(x) = \mu'(x) = \mu(y)$

$$\mu(x) = \mu'(x) = \mu(y)$$

and

$$\mu'(x) + \mu'(y) + \mu'(z) \geq \mu(x) + \mu(y) + \mu(z)$$
$$= 3\mu(x)$$
$$= 3\mu'(x)$$

- Since $\mu'(y) \leq \mu'(x)$ and $\mu'(z) \leq \mu'(x)$ we get $\mu'(x) = \mu'(y) = \mu'(z)$

* Since $\mu(x) = \mu'(x)$ we have $\mu(x) = \mu(y) = \mu(z) = \mu'(x) = \mu'(y) = \mu'(z)$ which cannot be true (and thus our initial assumption cannot be true):

Let $a$ be $|T(x)|$ before rotations $(a = |T1| + |T2| + 1)$

Let $b$ be $|T(z)|$ after rotations $(b = |T3| + |T4| + 1)$

Since $\mu(x) = \mu'(z) = \mu'(x)$ we have $\lceil \log a \rceil = \lceil \log b \rceil = \lceil \log (a + b + 1) \rceil$ but then we have the following contradiction:

- if $a \leq b$: $\lceil \log (a + b + 1) \rceil \geq \lceil \log 2a \rceil = 1 + \lceil \log a \rceil > \lceil \log a \rceil$
- if $a > b$: $\lceil \log (a + b + 1) \rceil \geq \lceil \log 2b \rceil = 1 + \lceil \log b \rceil > \lceil \log b \rceil$

Case 3:

- Can be proved analogously to case 2.