Introduction to Splay Trees

- Want to guarantee worst case time $O(n \log n)$ for $n$ dictionary operations, irrespective of what sequence the operations come in.
- Vanilla BST is embarrassing when data is in increasing order.
- AVL trees are easy to describe and do what we want but
  - Do not provide ‘easy wins’: If the same key is queried upon, we do not take advantage of the work we did in finding the key.
  - Need to have extra balancing information at each node, which needs to be constantly updated.
  - Other balanced or pseudo-balanced trees have similar problems.
- Splay trees are a neat alternative.
  - They are vanilla BST (no balancing, color information).
  - Adversary can cause bad behavior: A particular operation could cost $O(n)$ time.
  - But cannot cause this repeatedly.
Intuition Behind Splay Tree

A simple idea: Rotate to root

Does not work! Cost is $n + \sum_{i=2}^{n} i \in \Theta(n^2)$
**Basic Splay Primitives**

A splay($x$) is implemented using three simple primitives counted as at most two rotations.

The zigzag case ($x$ moves two levels up)

![Diagram of zigzag case](image1)

The zig case ($p$ is the root and has no parent)

![Diagram of zig case](image2)
Basic Splay Primitives

The zig-zig case can be thought of as two calls to the `restructure()` method. ($x$ moves two levels up)
Difference Between Splaying And Rotate-to-Root

Three zig-zigs, and 2 winds up halfway up the tree
A Complete Splay Example

(1) Zig–Zag

(2) Zig–Zig

(4) Final Tree

(3) Zig
Implementing Dictionary Operations

- A find(x) is performed by searching for x.
  - If x is found, then we issue splay(x)
  - If x is not found, then we splay the last non-null node. This is necessary, otherwise an adversary could defeat splaying by issuing multiple calls to $-\infty$ on a degenerate tree

- An insert(x) is performed as in vanilla BST. We then issue splay(x). This is necessary, otherwise an adversary could defeat splaying
Implementing Dictionary Operations

- A remove(x) is performed by searching for x. x is now at the root of the tree with two subtrees L and R
  - If L is empty, R becomes the new tree
  - We issue find(x) on L. Now root w of new L has no right child
  - We hook R as the right child and return w as the root

- deleteMin() and deleteMax() can also be easily implemented
Does Splaying Work?

- The *real cost* of a splay operation is proportional to the height of the tree. The adversary can force a high real cost.

- The *increase in balance* is the extent to which the splay helps adjusting the tree. If the real cost is high, the balance also increases, so in the future, high real cost is not possible.

- Consider the zigzag case and suppose we splay(x)
  - If $B$ is heavy (say more than half the elements are in $B$) the splay brings a lot of nodes to the top. Future real costs for a lot of nodes is going to be low.
  - If $B$ is light (we have more than half the nodes in $A$ or $D$) so we bypassed a lot of nodes. The real cost could not be high.
Amortized Analysis: Proving Splaying Works

- Consider the abstract data type (ADT) $MStack$ which supports the operations Push($k$) and Pop() and also supports MultiPop() (which pops all existing items in the stack)

- Consider the time $T$ for an arbitrary sequence of $n$ operations on MStack
  
  - If the $i$th operation was a MultiPop(), it would take $O(i)$ time, thus $T = \sum i \in O(n^2)$
  
  - While true this analysis is conservative since there can’t be more pops than pushes.

  - A more careful analysis would prove this formally. Let $M_0, \ldots M_{n-1}$ be the series of operations, and let $M_{i_0}, \ldots M_{i_{k-1}}$ be the MultiPop() calls. The running time of $M_{i_j}$ is $i_j - i_{j-1} - 1$ which when summed over all MultiPop() proves that $T \in O(n)$
The Potential Method

- The $O(n)$ behaviour is easy to prove for MStack, but in more complicated data structures, it is harder to visualize the interdependence of multiple operations.

- The potential method is a technique which glosses over fine details in the data structure in the attempt to prove a reasonable cost estimate.
  
  - Let $c(i)$ denote the running time of the $i$th operation.
  - Denote $\phi(i)$, termed potential, to be an arbitrary function.
  - Define the amortized time $c'(i) = \phi(i) - \phi(i - 1) + c(i)$.
  - Observe that $T = \sum c(i) = \sum c'(i) + \phi(n) - \phi(0)$.
  - The actual time $T$ is less than the total amortized time if the net change in potential is positive.

- For MStack, define $\phi(i)$ to be the number of items before the $i$th operation. Since $c'(i) \in O(1)$ no matter what operation, $T \in O(n)$.
More on Amortized Analysis

• The potential function must be carefully chosen so as to prove the result we are interested

• Consider an extendable array (of size $m$) that may be used, for example, in inserting items during hashing
  
  – Rule: If the load factor becomes half, double the size
  
  – Assume hash functions so that find() and remove() takes $O(1)$ time
  
  – An insert takes $O(1)$ time quite often. Except, of course, when the load factor equals 0.5. Then it takes $O(m)$ time to rehash the $0.5m$ keys

• In fact, starting with a size of 4, a sequence of $n = 2^p - 1$ arbitrary operations takes cumulatively $O(n)$ time

• Choosing $\phi(i)$ to be the number of items in the array doesn’t help us prove the result since amortized cost of a single insert is $\Theta(m)$
**Extendible Array Example**

We apply the previous rule on an array of size 4, and insert 15 \((p = 4)\) items and count only the effort in rehashing since the cost of an actual insert otherwise is \(O(1)\).

<table>
<thead>
<tr>
<th>Operation Number</th>
<th>Rehash Cost</th>
<th>Size of Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5–7</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>9–15</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

Notice that the actual total cost is 14. For \(2^p - 1\) inserts, actual total is \(2(2^{p-1} - 1) < 2^p\) which is ‘linear.’
Analyzing Zigzag primitive for Splay Trees

- A splay tree supports insert, successful and unsuccessful searches, and removes. The cost of each operation is governed by a splay()
- A splay operation, in turn, is made up of, in general $k_1$ zigzag steps, $k_2$ zigzig steps, and at most 1 zig step
- Consider computing the amortized cost of a single zigzag step

![Diagram of splay tree restructuring](image)

- Define potential of a splay tree to be $\phi = \sum_i^n \log S_i = \sum R_i$ where $S_i$ is the sum of the descendants of node $i$ and the rank $R_i = \log S_i$
- The change in potential of one zigzag step is thus $\Delta = R'_x + R'_p + R'_g - (R_x + R_p + R_g)$ where prime denotes the rank after the zigzag splay primitive has been applied
Analyzing Zigzag primitive for Splay Trees

• We want to show that the potential could not have increased too much
  - \( R_g = R'_x \) and \( R_p > R_x \) implies \( \Delta \leq R'_p + R'_g - 2R_x \)
  - We relate \( R'_p + R'_g \) to \( R'_x \)
  - Notice that \( S'_p + S'_g < S'_x \)
  - Thus \( R'_p + R'_g < 2R'_x - 2 \)
  - Thus \( \Delta \leq 2(R'_x - R_x) - 2 \)

• The amortized cost of a zigzag primitive is \( 2(R'_x - R_x) \)

• If the splay consisted only of zigzag steps which took \( x \) to the root, the amortized cost of the splay would be at most \( 2(R'_\text{root} - R_\text{leaf}) \)

• The amortized cost is at most \( 2 \log n \)

• The actual cost of \( m \) splay operations cannot exceed \( O(m \log n) \) if we start with an empty tree