Splay Trees

• Goals:
  – Able to implement Splay Tree insertion, search, and deletion, in Java.
  – Able to state the \textit{amortized} asymptotic time cost of splay tree operations.
  – Exposure to amortized cost of a series of operations.
  – (Not in scope: ability to prove the bounds on time-cost of splay tree operations.)
Splay Trees

- A “lazy” self-balancing binary search tree.
- With each insertion, search and deletion, the tree adjusts its shape, via a “splay” operation.
  - A “splay” (at node $x$) is a series of rotations that brings $x$ to the root position. (Details soon.)
  - No balance information is maintained: tree might have a tall shape.
    - Yet, due to splaying, time-cost of $m$ ops in an $n$-node tree is $O(m \log n)$.
    - This is known as an “amortized” op time-cost of $O(\log n)$.
Amortized Analysis

- How is it practical for restaurants to offer unlimited soda refills?
  - Why don't they go bankrupt?
Amortized Analysis

• How is it practical for restaurants to offer unlimited soda refills?
  – Why don't they go bankrupt?
• It's fine, as long as the *price* of a soda covers the *average* soda expense,
  – even if a customer or two is very thirsty.
Amortized Time: Array Append

- Arrays with variable size:
  - One physical array
  - One integer to give physical size
  - One integer to give current active size

- How to append to the array?
Amortized Time: Array Append

- Attempt 1: Resize by 1 at a time
  - Cost of inserting $n$ elements?
Amortized Time: Array Append

- Attempt 1: Resize by 1 at a time
  - Cost of inserting $n$ elements?

- Attempt 2: Resize by 16 at a time
  - Cost of inserting $n$ elements?

$$\sum_{i=1}^{n} i = O(n^2)$$
Amortized Time: Array Append

- **Attempt 1**: Resize by 1 at a time
  - Cost of inserting $n$ elements?

- **Attempt 2**: Resize by 16 at a time
  - Cost of inserting $n$ elements?

- **Attempt 3**: Double the array each time
  - Cost of inserting $n$ elements?

\[
\sum_{i=1}^{n} i = O(n^2) \\
\sum_{i=1}^{n/16} 16i = O(n^2)
\]
Array Append, Doubling

- Cost So Far
- 2n
Array Append, in Summary

- Individual operations very expensive
  \( O(n) \)

- Expensive operations very rare

- **Total cost** for any sequence of insertions:
  \( O(n) \)

- **Amortized cost** of one insertion:
  \( O(1) \)
Amortized Time

- **Amortized** time cost gives *long-term* time information
  - Some individual operations maybe be quite costly (i.e., slow);
  - other operations might be low-cost (i.e., quick).
  - An amortized analysis determines the total, which can be used to express the *average cost per operation*.

- Enlarging a hash table by doubling: an example of *amortized constant time*. 

Calculating Amortized Time

\[ t_{\text{amortized}} = \frac{t_{\text{total}}}{\text{number of ops}} \]
Amortized Analysis: The Accounting Method

- Assign a **price** for each operation
  - You “charge” your user a price for each operation, i.e., each input, each search, etc.
  - Price might depend on $n$, or might not.
- Expenses: your code must **pay** for the implementation of the ADT operations.
- If your data structure never goes bankrupt, even with worst-case input, then you have a valid amortized analysis.
Accounting Method: Hash table

- **Price:** the caller must “pay” $7 per insertion (constant cost).

- **Expenses:**
  - $1 to insert into a chain, when the table isn't full.
  - $1 to scan each location in the table
  - $1 to transfer each record in a chain

- **Each ordinary insertion builds up a small surplus that can be “spent” when enlarging the table.**
Amortized Time Summary

• Amortized time = total time / number of ops
  – Some operations are cheap: save up for later
  – Some ops are expensive, using up stored value

• How to make an amortized analysis:
  – Accounting method
  – Potential method (out of scope)
Accounting Method Summary

- The accounting method uses *imaginary* dollars to prove bounds on the average cost.
  - Users pay a dollar price per call to a method,
  - Data structure incurs dollar expense carrying it out,
  - Income must cover the expenses.

- Examples:
  - Inserting into a hash table (including enlarging): amortized constant cost.
  - Splay tree insertion, search, deletion: amortized logarithmic cost. (Proof on D2L.)
Splay Trees

- Amortized time
- **Splay Trees**
Splay Trees

- Splay trees are **not always short**!
- Instead of keeping the tree short, we “splay” a node to the root for each basic operation:
  - Insert
  - Search
  - Deletion
- *Amortized* cost $O(\log n)$ per operation.
- Insight: Recently used nodes are likely to be used again, soon.
Splay Trees

Here's where it gets weird:

- Splay trees may have $O(n)$ height, but...
- Splay trees still display $O(\log n)$ amortized performance
- (Amortization makes this possible, but the analysis is difficult.)
Splaying (on Insert)

1. Start with an ordinary tree
2. Insert new node
3. Splay the node up to the root
Splaying (on Search)

1. Start with an ordinary tree.
2. Find the key in the tree.
3. Splay that node up to the root.

(If the key is absent, splay the leaf at which the search fails.)
Splaying (on Delete)

- Splay node to root
  - Like search, if not found, splay parent
- If 0 or 1 children, delete node like normal BST

*see next slide for Case 3 – (two children)*
Splaying (on Delete, Case 3)

- If two children:
  - Delete root (creates two sub-trees)
  - Splay predecessor (max of left) to root
  - Join sub-trees

- Equivalent:
  - Splay predecessor to root of left subtree (not overall root)
  - Delete Case 3 like normal
Splaying (on Deletion)

1. Start with an ordinary tree.

2. Splay target to root (like search)

3. If 0 or 1 children, delete as normal.
Splaying (on Deletion, Case 3)

3. If 2 children, delete root node
4. Splay predecessor
5. Join trees

In step 4, can we always assume that P has only a left child, and no right child? Why or why not?
Splaying Rotations

• Three rotations to know:
  “Zig” - plain rotation (only used at the root)
  “Zig-Zig” - two-level rotation, both in same direction
  “Zig-Zag” - two-level rotation, in opposite directions

• Start at the splayed node $x$, and while $x$ has a grandparent, do two-level rotations toward the root; then, if necessary, do a 1-level at the end.
Zig (our old friend, with new name)

NOTES:
- A will always represent the node we're splaying
- We'll leave off all of the sub-trees for simplicity
Order matters here. Rotate at C first. Otherwise, we cannot guarantee $O(\log n)$ performance.
Zig-Zag
Splay Trees In Practice: Insertion

Insert 1 into the tree.
Splay Trees In Practice: Insertion

Insert 2 into the tree.

Normally, 2 would be a child of 1...but we splay it up to the root.
The process continues as we insert more nodes...it never balances.
Splay Trees In Practice: Insertion
Splay Trees In Practice: Insertion
Splay Trees In Practice: Insertion
Splay Trees In Practice: Insertion
So what happens if we search for 1 here?

Let's track the steps of splaying it up to the root...
This is a zig-zig step.

This part of the subtree is reversed...things look bad, don't they?
This is also a zig-zig step.

But you see how the tree is starting to collapse?
Splay Trees In Practice: Search

This is also a zig-zig step.
Do you see how the tree is starting to collapse?
The last step is a zig step, since we're at the root.

The tree is much shorter.
We arrived at this configuration by doing:

- Lots of very cheap insertions
- A single expensive search

Intuition:
Cheap operations “save up” for later expensive ones.

Thus, we use amortized time.
Splay Trees: Discussion

- In what sense is a splay tree fast?
- What sorts of programs would work well with a splay tree? What would not?
- Even in “slow” cases, cost of \( m \) operations, starting from empty, is \( O(m \log n) \):
  - Amortized cost per operation is \( O(\log n) \).
  - Analysis is lengthy (on D2L, optional).