AVL Trees: Are They Worth It?

Advantages
• Rotations are cool!

Disadvantages
• Wouldn’t want to meet one in a dark alley at night

Splay What?
• Blind adjusting version of AVL trees
  – Why worry about balances? Just rotate anyway!
• Amortized time for all operations is $O(\log n)$
• Worst case time is $O(n)$
• Insert/Find always rotates node to the root!

Idea

You’re forced to make a really deep access.

Since you’re down there anyway, fix up a lot of deep nodes?

Zig-Zag* 

Yes, the original 1985 paper actually uses this terminology!
Zig

Splaying Example

Still Splaying 6

Almost There, Stay on Target*

Splay Again

Example Splayed Out
Why Splaying Helps

- If a node $n$ on the access path is at depth $d$ before the splay, it's at about depth $d/2$ after the splay
  - Exceptions are the root, the child of the root, and the node splayed
- Overall, nodes which are below nodes on the access path tend to move closer to the root
- Splaying gets amortized $O(\log n)$ performance.

Splay Operations: Find

- Find the node in normal BST manner
- Splay the node to the root

Splay Operations: Insert

- Insert the node in normal BST manner
- Splay the node to the root

Splay Operations: Remove

Now what?

Join

- Join(L, R): given two trees such that L < R, merge them
  - Splay on the maximum element in L then attach R

Insert Example

Proof by analogy is not a valid proof technique!
**Delete Example**

Delete(4)

**Nifty Splay Operation: Splitting**

- Split(T, x) creates two BSTs L and R:
  - all elements of T are in either L or R (\( T = L \cup R \))
  - all elements in L are \( \leq x \)
  - all elements in R are \( \geq x \)
  - L and R share no elements (\( L \cap R = \emptyset \))

*How do we split a splay tree?*

**Splay Tree Summary**

- All operations are in amortized \( O(\log n) \) time
- Splaying can be done top-down; better because:
  - only one pass
  - no recursion or parent pointers necessary
- Splay trees are very effective search trees
  - Relatively simple
  - No extra fields required
  - Excellent locality properties: frequently accessed keys are cheap to find

**Interlude: Amortized Analysis**

- Consider any sequence of operations applied to a data structure
  - Your worst enemy could choose the sequence!
- Some operations may be fast, others slow
- Goal:
  - Show that the average time per operation is still good

\[
\text{total time for } n \text{ operations} = \frac{\text{total time}}{n}
\]
Stack ADT

- Stack operations
  - push
  - pop
  - isEmpty
- Stack property: if x is on the stack before y is pushed, then x will be popped after y is popped

What is biggest problem with an array implementation?

Stretchy Stack Implementation

```c
int * data;
int maxsize;
int top;

Push(e){
    if (top == maxsize){
        temp = new int[2*maxsize];
        for (i=0;i<maxsize;i++) temp[i]=data[i];
        delete data;
        data = temp;
        maxsize= 2*maxsize; }
    else { data[++top] = e; }
}
```

Best case Push ∈ O(  )
Worst case Push ∈ O(  )

Stretchy Stack Amortized Analysis

- Consider sequence of n operations
  - push(3); push(19); push(2); …
- What is the max number of stretches?
- What is the total time?
  - Let’s say a regular push takes time a, and stretching an array contains k elements takes time bk.

- Amortized time =

Series

- Arithmetic series: \[ \sum_{i=1}^{n} f(i) = \frac{n(N+1)}{2} \]
- Geometric series: \[ \sum_{i=0}^{n} A^i = \frac{A^{n+1}-1}{A-1} \]

\[ \sum_{i=0}^{\log_2 n} 2^i = \frac{2^{\log_2 n+1} - 1}{2-1} = 2^{\log_2 n+1} - 1 = 2n - 1 \]

Moral of the Story

- For more complicated analyses, this procedure is formalized with the idea of credit
- We said that a splay was O(\( \log n \))
  - Always invest \( \log n \) per splay
  - For an easy splay, bank the leftover money
  - For a hard splay, use money from the bank
  - Prove there’s always enough money in the bank for any operation

Bug Brian or Hannah for more references on amortized analysis!