Splay Trees I

Recap: Lots of search trees
- Unbalanced BSTs
- AVL Trees
- 2-3, Red-black, AA Trees
- Treaps + Skip lists

→ **Focus:** Worst-case or randomized expected case

Other/Better Criteria?

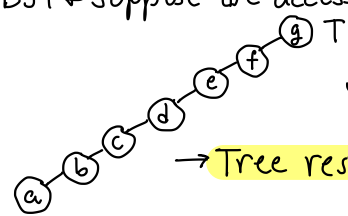
Expected case: Some keys more popular than others

Self-adjusting: Tree adapts as popularity changes

How to design/analyze?

Splay Tree: A self-adjusting binary search tree
- **No rules!** (yay anarchy!)
  - No balance factors
  - No limits on tree height
  - No colors/levels/priorities
- **Amortized efficiency:**
  - Any single op - slow
  - Long series - efficient on avg.

Intuition: Let T be an unbalanced BST + suppose we access its deepest key

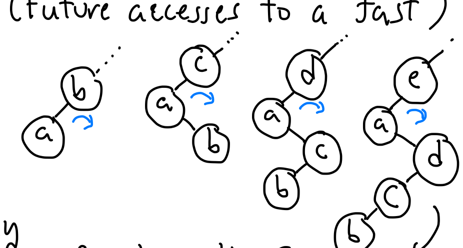


find("a")

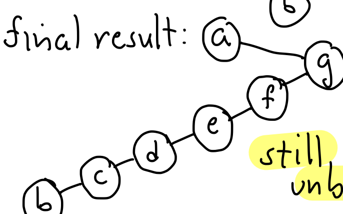
ugh!

→ **Tree restructures itself**

Idea I: Rotate "a" to top (Future accesses to "a" fast)

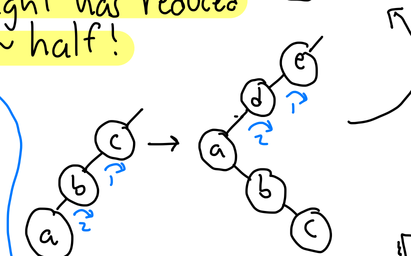


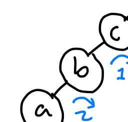
.... final result:



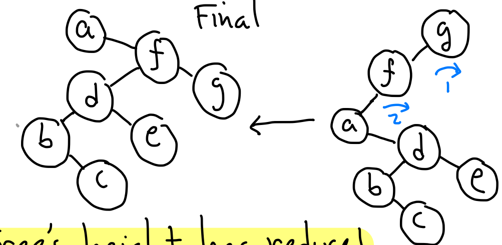
still unbalanced!

Idea II: Rotate 2 at a time - upper + lower





Final

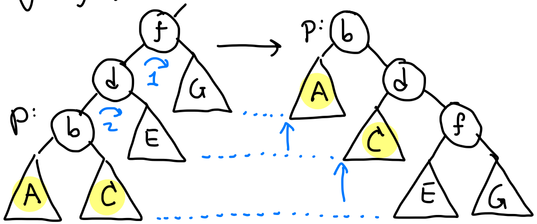


Tree's height has reduced by ~ half!

Lesson: Different combinations of rotations can:
- bring given node to root
- significantly change (improve?) tree structure.

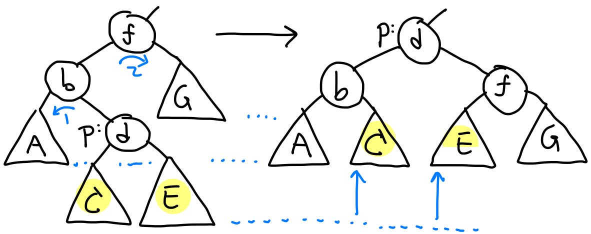
Splay Trees II

ZigZig(p): [LL case]



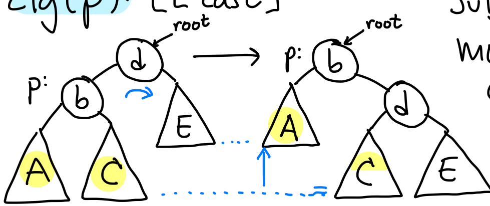
Subtrees A,C move up↑

ZigZag(p): [LR case]



Subtrees C, E of p move up↑

Zig(p): [L case]



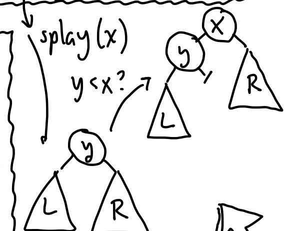
Subtree A moves up↑
C unchanged

Splay(Key x):

```
Node p ← find x by standard BST search
while (p≠root) {
    if (p == child of root) zig(p)
    else /* p has grand parent */
        if (p is LL or RR grand child) zigzig(p)
        else /* p is LR or RL gr. child */ zigzag(p)
```

insert(x):

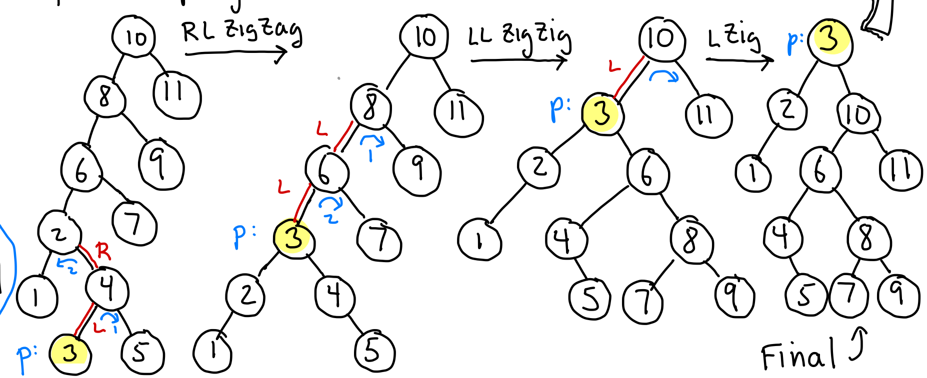
```
{ splay(x)                  ---> root.key ≠ x or error!
  q = new Node(x)
  if (root.key < x)
      x.left = root
      x.right = root.right
      root.right = null
  else ... symmetrical...
```



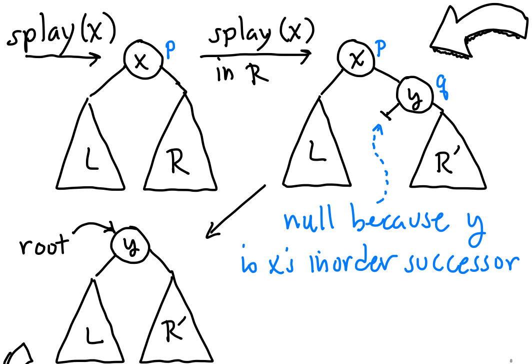
find(x):

```
splay(x)
if (root.key == x)
    found!
else not found
```

Example: splay(3)



SPLAY TREES III



delete(x):

splay(x) [x now at root]
p = root
if (p.key ≠ x) error!
splay(x) in p's right subtree
q = p.right [q's key is x's successor]
q.left = p.left [q.left == null]
root = q

Analysis:

- Amortized analysis
- Any one op might take $O(n)$
- Over a long sequence, average time is $O(\log n)$ each
- Amortized analysis is based on a sophisticated potential argument
- Potential: A function of the tree's structure
  Balanced ⇒ Low potential
  Unbalanced ⇒ High potential
- Every operation tends to reduce the potential

Splay Trees are Amazingly Adaptive!

Balance Theorem: Starting with an empty dictionary, any sequence of m accesses takes total time

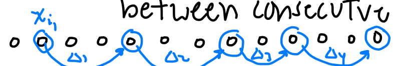
$$O(m \log n + n \log n)$$

where n = max. entries at any time.

Dynamic Finger Theorem:

Keys: $x_1 < \dots < x_n$. We perform accesses $x_{i_1}, x_{i_2}, \dots, x_{i_m}$

Let $\Delta_j = i_j - i_{j-1}$: distance between consecutive items



Thm: Total access time is

$$O\left(m + n \log n + \sum_{j=1}^{m} (1 + \lg \Delta_j)\right)$$

Static Optimality:

- Suppose key x_i is accessed with prob p_i $\left(\sum_{i=1}^{n} p_i = 1\right)$
- Information Theory: Best possible binary search tree answers queries in expected time $O(H)$ where $H = \sum_i p_i \lg 1/p_i$ ← Entropy

Static Optimality Theorem: Given a seq. of m ops. on splay tree with keys $x_1 \dots x_n$, where x_i is accessed q_i times. Let $p_i = q_i/m$. Then total time is

$$O\left(m \sum_i p_i \lg 1/p_i\right)$$