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 $a$.  

<table>
<thead>
<tr>
<th>Recap: Lots of search trees</th>
<th>Lesson: Different combinations of rotations can:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Unbalanced BSTs</td>
<td>- bring given node to root</td>
</tr>
<tr>
<td>- AVL Trees</td>
<td>- significantly change (improve) tree structure.</td>
</tr>
<tr>
<td>- 2-3, Red-black, AA Trees</td>
<td></td>
</tr>
<tr>
<td>- Treaps &amp; Skip lists</td>
<td></td>
</tr>
</tbody>
</table>

Focus: Worst-case or randomized expected case

Final

Tree's height has reduced by ~ half!

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

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

SPLAY TREES I

Intuition: Let $T$ be an unbalanced BST. Suppose we access its deepest key $a$. Find("a") ugh! 

→ Tree restructures itself
**ZigZig(p): [LL case]**

Node \( p \) ← find \( x \) by standard BST search
while \( (p \neq root) \)
  if \( (p \text{ is child of root}) \) Zig(p)
  else /* \( p \) has grand parent */
    if \( (p \text{ is LL or RR grand child}) \) ZigZig(p)
    else /* \( p \) is LR or RL grand child */ ZigZag(p)

**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 Trees II**

**Example:**

splay(3)

**Splay (Key x):**

Node \( p \) ← find \( x \) by standard BST search
while \( (p \neq root) \)
  if \( (p \text{ is child of root}) \) Zig(p)
  else /* \( p \) has grand parent */
    if \( (p \text{ is LL or RR grand child}) \) ZigZig(p)
    else /* \( p \) is LR or RL grand child */ ZigZag(p)

**find(x):**

root ← splay(\( x \))
if \( (root . key == x) \)
  return root . value
else return null

**insert(x):**

Node \( p \) ← splay(\( x \))
if \( (p . key == x) \) Error!!
\( q \) ← new Node(\( x \))
if \( (p . key < x) \)
  \( q . left \) ← \( p \)
  \( p . right \) ← \( p . right \)
else \( \) (symmetrical)...
  root ← \( q \)
splay\(x\) \quad \text{in } R

\text{null because } y
\text{is } x\text{'s inorder successor}

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 \(\Rightarrow\) Low potential
  Unbalanced \(\Rightarrow\) High potential
- Every operation tends to reduce the potential

**Dynamic Finger Theorem**

Keys: \(x_1, \ldots, x_n\). We perform accesses \(x_{i_1}, x_{i_2}, \ldots, x_{i_m}\).

Let \(\Delta_j = j_j - j_{j-1}\), distance between consecutive items.

Thm: Total access time is
\(O(m + n \log n + \sum_{j=1}^{m}(1 + \log \Delta_j))\)

Static Optimality:
- Suppose key \(x_i\) is accessed with prob \(p_i\). \(\sum_i p_i = 1\)
- Information Theory: Best possible binary search tree answers queries in expected time \(O(H)\) where
\(H = \sum_i p_i \log \frac{1}{p_i}\) = Entropy

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

**Splay Trees**
Splay Trees are Amazingly Adaptive!

**Balance Theorem:** Starting with an empty dictionary, any sequence of \(m\) accesses takes total time
\(O(m \log n + m \log n)\)
where \(n = \max\) entries at any time.

\(\text{delete}(x):\)

splay\(x\) \[x\text{ now at root}\]
\(p = \text{root}\)
if \((p, \text{key} \neq x)\) error!
splay\(x\) in \(p\)'s right subtree
\(q = p, \text{right} \quad [q\text{'s key is } x\text{'s successor}]\)
\(q, \text{left} = p, \text{left} \quad [q\text{'s left == null}]\)
root = \(q\)