Splay Trees

\[ v \]

\[ 3 \]

\[ 4 \]

\[ 6 \]

\[ 8 \]
Splay Trees are Binary Search Trees

**BST Rules:**
- entries stored only at internal nodes
- keys stored at nodes in the left subtree of \( v \) are less than or equal to the key stored at \( v \)
- keys stored at nodes in the right subtree of \( v \) are greater than or equal to the key stored at \( v \)

**An inorder traversal will return the keys in order**

Note that two keys of equal value may be well-separated.
Searching in a Splay Tree: Starts the Same as in a BST

Search proceeds down the tree to found item or an external node.

Example: Search for time with key 11.
Example Searching in a BST, continued

- search for key 8, ends at an internal node.
Splay Trees do Rotations after Every Operation (Even Search)

- **new operation:** *splay*
  - splaying moves a node to the root using rotations

- **right rotation**
  - makes the left child \( x \) of a node \( y \) into \( y \)'s parent; \( y \) becomes the right child of \( x \)

- **left rotation**
  - makes the right child \( y \) of a node \( x \) into \( x \)'s parent; \( x \) becomes the left child of \( y \)

(a right rotation about \( y \))

(a left rotation about \( x \))

(structure of tree above \( y \) is not modified)

(structure of tree above \( x \) is not modified)
Splaying:

- “$x$ is a left-left grandchild” means $x$ is a left child of its parent, which is itself a left child of its parent.
- $p$ is $x$’s parent; $g$ is $p$’s parent.

Start with node $x$.

- Is $x$ the root? If yes, stop.
  - If no, is $x$ a child of the root? If yes, is $x$ the left child of the root?
    - If yes, right-rotate about the root.
    - If no, left-rotate about the root.
  - If no, is $x$ a left-left grandchild? If yes, right-rotate about $g$, then right-rotate about $p$.
  - If no, is $x$ a right-right grandchild? If yes, left-rotate about $g$, then left-rotate about $p$.
  - If no, is $x$ a right-left grandchild? If yes, left-rotate about $p$, then right-rotate about $g$.
  - If no, is $x$ a left-right grandchild? If yes, right-rotate about $p$, then left-rotate about $g$.

Splay Trees
Visualizing the Splaying Cases

zig-zig

zig-zag

zig
Splaying Example

let $x = (8,N)$
- $x$ is the right child of its parent, which is the left child of the grandparent
- left-rotate around $p$, then right-rotate around $g$

1. (before rotating)

2. (after first rotation)

3. (after second rotation)

$x$ is not yet the root, so we splay again
Splaying Example, Continued

1. (before applying rotation)

- now \( x \) is the left child of the root
- right-rotate around root

2. (after rotation)

- \( x \) is the root, so stop
Example Result of Splaying

- The tree might not be more balanced
- e.g. splay (40,X)
  - before, the depth of the shallowest leaf is 3 and the deepest is 7
  - after, the depth of shallowest leaf is 1 and deepest is 8
Splay Tree Definition

- A splay tree is a binary search tree where a node is splayed after it is accessed (for a search or update)
  - Deepest internal node accessed is splayed
  - Splaying costs $O(h)$, where $h$ is height of the tree
    - Which is still $O(n)$ worst-case
      - $O(h)$ rotations, each of which is $O(1)$
Splay Trees & Ordered Dictionaries

which nodes are splayed after each operation?

<table>
<thead>
<tr>
<th>method</th>
<th>splay node</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(k)</td>
<td>if key found, use that node</td>
</tr>
<tr>
<td></td>
<td>if key not found, use parent of ending external node</td>
</tr>
<tr>
<td>put(k,v)</td>
<td>use the new node containing the entry inserted</td>
</tr>
<tr>
<td>remove(k)</td>
<td>use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)</td>
</tr>
</tbody>
</table>
Amortized Analysis of Splay Trees

- Running time of each operation is proportional to time for splaying.
- Define rank(v) as the logarithm (base 2) of the number of nodes in subtree rooted at v.
- Costs: zig = $1, zig-zig = $2, zig-zag = $2.
- Thus, cost for playing a node at depth d = $d.
- Imagine that we store rank(v) cyber-dollars at each node v of the splay tree (just for the sake of analysis).
Cost per zig

Doing a zig at $x$ costs at most $\text{rank}'(x) - \text{rank}(x)$:
- $\text{cost} = \text{rank}'(x) + \text{rank}'(y) - \text{rank}(y) - \text{rank}(x) \\ < \text{rank}'(x) - \text{rank}(x).$
Cost per zig-zig and zig-zag

Doing a zig-zig or zig-zag at $x$ costs at most
$$3(\text{rank}'(x) - \text{rank}(x)) - 2$$
Cost of Splaying

Cost of splaying a node $x$ at depth $d$ of a tree rooted at $r$:
- at most $3(\text{rank}(r) - \text{rank}(x)) - d + 2$:
- Proof: Splaying $x$ takes $d/2$ splaying substeps:

$$
cost \leq \sum_{i=1}^{d/2} \text{cost}_i
$$

$$
\leq \sum_{i=1}^{d/2} (3(\text{rank}_i(x) - \text{rank}_{i-1}(x)) - 2) + 2
$$

$$
= 3(\text{rank}(r) - \text{rank}_0(x)) - 2(d/d) + 2
$$

$$
\leq 3(\text{rank}(r) - \text{rank}(x)) - d + 2.
$$
Performance of Splay Trees

- Recall: rank of a node is logarithm of its size.
- Thus, amortized cost of any splay operation is $O(\log n)$
- In fact, the analysis goes through for any reasonable definition of $\text{rank}(x)$
- This implies that splay trees can actually adapt to perform searches on frequently-requested items much faster than $O(\log n)$ in some cases