Splay Trees

Self-Adjusting Binary Search Trees

(Prep for Implementation)

Part 0: Key Vocabulary

Core Concepts

- **splay** = "spread out." The operation that moves a node to the root.
- **rotation** = a local restructuring that preserves BST order but changes depths
- **zig** = one rotation (used when parent is the root)
- **zig-zig** = two rotations in the same direction (node and parent are both left children, or both right children)
- **zig-zag** = two rotations in opposite directions (node is left child of right child, or vice versa)
- **locality** = recently accessed data is likely to be accessed again soon

For the Analysis

- **size** $s(x)$ = number of nodes in the subtree rooted at x (including x)
- **rank** $r(x) = \lfloor \log_2 s(x) \rfloor$ = how "big" the subtree is (logarithmically)
- **potential** $\Phi(T) = \sum_{x \in T} r(x)$ = sum of all ranks in the tree
- **amortized cost** = actual cost + change in potential

The Big Idea

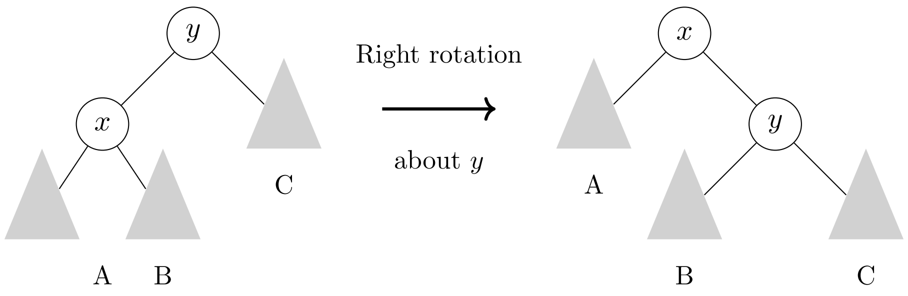
Every time you access a node, move it to the root.
Why? Two benefits:

1. **Temporal locality:** If you access node x again soon, it's already near the top!
2. **Self-balancing:** The splay operation tends to make the tree more balanced over time.

The "magic" is *how* we move it to the root. Naive repeated rotations don't work well. The specific zig-zig pattern is what makes splay trees efficient.

Part 1: Understanding Rotations

A rotation is like a "pivot" that swaps a parent-child relationship while keeping the BST property.



What Happens in a Right Rotation

1. x (left child) becomes the new parent
2. y (old parent) becomes x's right child
3. x's old right subtree (B) becomes y's new left subtree
4. Subtrees A and C don't move relative to their parents

Key insight: The in-order traversal (A, x, B, y, C) stays the same!

Left Rotation is the Mirror Image

In a **left rotation** about x:

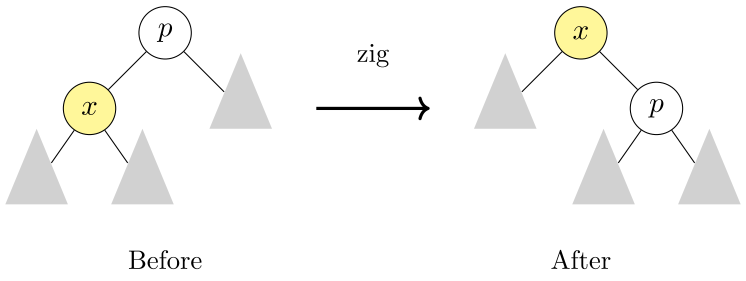
- y (right child) becomes the new parent
- x (old parent) becomes y's left child
- y's old left subtree becomes x's new right subtree

Part 2: The Three Splay Cases

When splaying node x to the root, we look at x, its parent p, and its grandparent g (if it exists).

Case 1: Zig (Parent is Root)

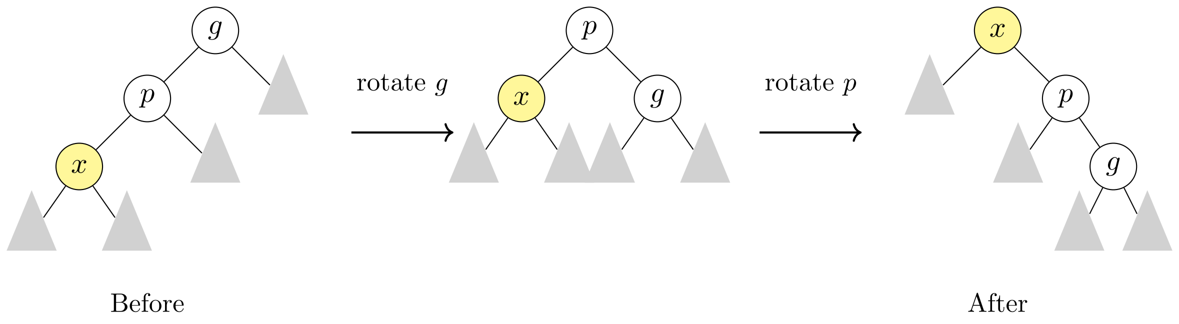
If p is the root, just do one rotation.



This is just a single rotation. x is now the root.

Case 2: Zig-Zig (Same Direction)

x and p are both left children (or both right children).

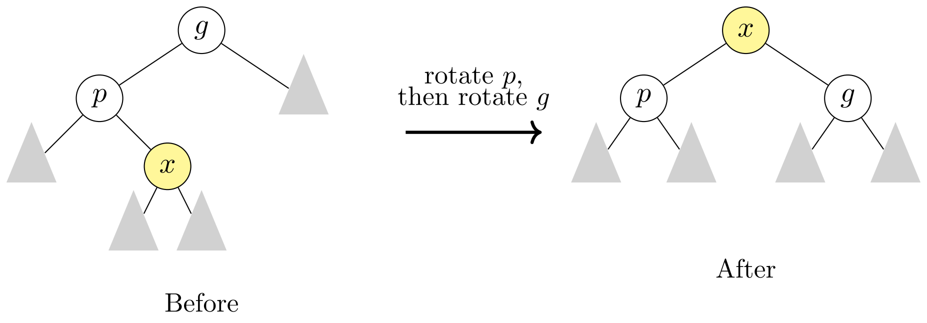


Critical: Rotate Grandparent First!

In zig-zig, we rotate g first, then p. This is NOT the same as rotating p twice!
The order matters for the amortized analysis to work out.

Case 3: Zig-Zag (Opposite Direction)

x is a right child of p, and p is a left child of g (or the mirror).



In zig-zag, we rotate p first, then g. Node x ends up with p and g as its two children.

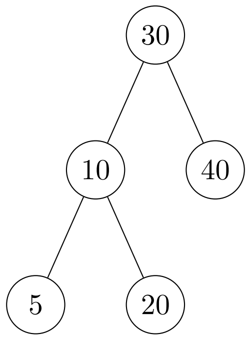
Summary: How to Decide Which Case

Given node x to splay:

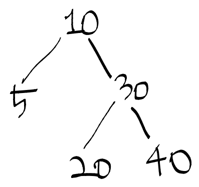
1. If x's parent is the root $\Rightarrow$ **Zig** (one rotation)
2. Else look at x, parent p, grandparent g:
   - If x and p are both left children OR both right children $\Rightarrow$ **Zig-Zig**
   - If x and p are on opposite sides $\Rightarrow$ **Zig-Zag**
3. Repeat until x is the root

Problems

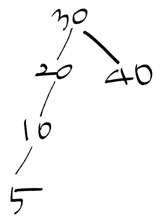
Problem 1 (Rotation Practice)**.** Consider this BST:



Part (a): Perform a **right rotation** about node 30. Draw the resulting tree.



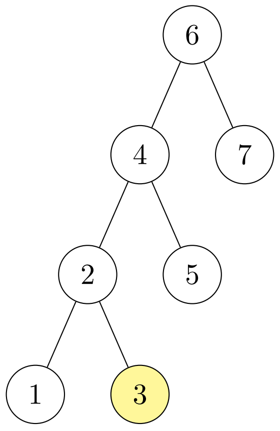
Part (b): Starting from the ORIGINAL tree, perform a **left rotation** about node 10. Draw the resulting tree.



Part (c): In both cases, verify that the in-order traversal (5, 10, 20, 30, 40) is preserved.

Both cases are same as original.

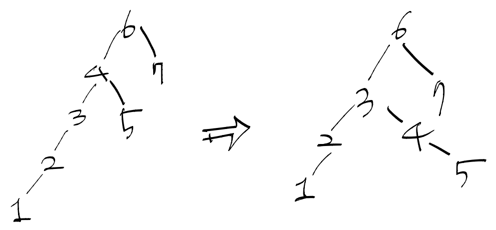
Problem 2 (Splay Step by Step)**.** Consider this BST. We want to `splay(3)` (move node 3 to the root).



Part (a): Identify the first splay case.

- Node to splay: $x = 3$
- Parent: $p =$ 2
- Grandparent: $g =$ 4
- Is p the root? no (root is 6)
- Are x and p on the same side (both left or both right)? no
- Therefore, the case is: zig-zag (zig / zig-zig / zig-zag)

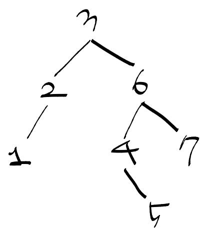
Part (b): Draw the tree after performing this splay step.



1. rotate 2

2. rotate 4

Part (c): Identify and perform the next splay step(s) until node 3 is at the root. Show each intermediate tree.



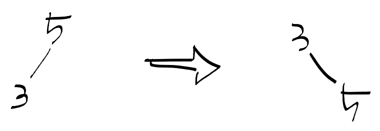
Problem 3 (Splay Tree Operations)**.** Starting with an empty splay tree, perform the following operations in order. After each operation, draw the resulting tree.

Part (a): `insert(5)`

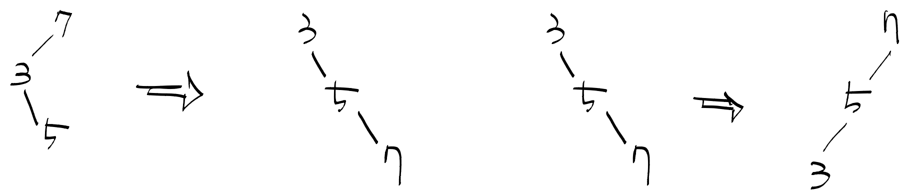
(Hint: Insert like a normal BST, then splay the inserted node to the root. Since it's the only node, it's already the root.)

Part (b): `insert(3)`

(Insert 3 as left child of 5, then splay 3 to root.)

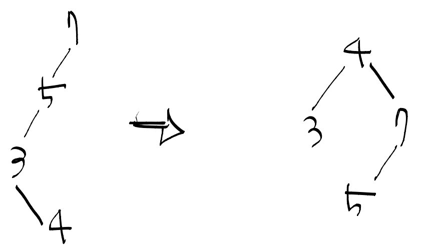


Part (c): `insert(7)`



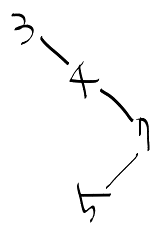
Part (d): `insert(4)`

(This will require a zig-zag!)

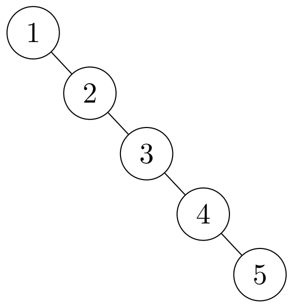


Part (e): `find(3)`

(Search for 3, then splay it to the root.)

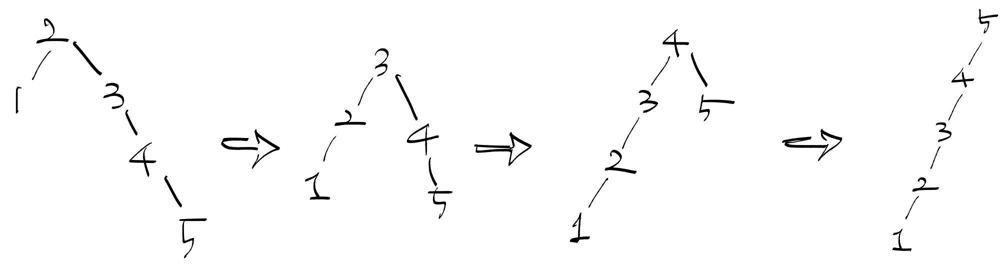


Problem 4 (Why Naive Move-to-Root Fails)**.** Consider a degenerate tree (a "stick"):



The **naive** approach: to move node 5 to the root, just rotate about 4, then about 3, then about 2, then about 1.

Part (a): Perform the naive move-to-root for node 5 (four left rotations). Draw the final tree.

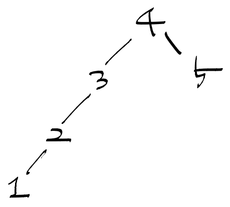


Part (b): What is the shape of the resulting tree? Is it more balanced or less balanced than before?

Still stick in the opposite direction.

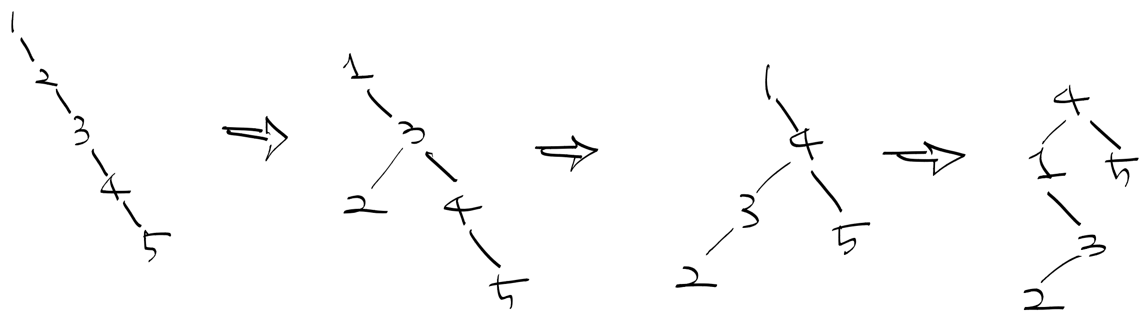
No improvement in balance.

Part (c): Now perform proper `splay(4)` on the result. What splay cases do you use?



zig

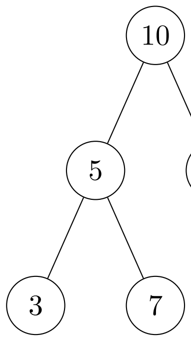
Part (d): After splaying 4, is the tree more balanced? This illustrates why the zig-zig rule (rotate grandparent first) is crucial.



Problem 5 (Splay Tree Potential Analysis)**.** For the analysis, we use:

- $s(x)$ = size of subtree rooted at x (number of nodes)
- $r(x) = \lfloor \log_2 s(x) \rfloor$ = rank of node x
- $\Phi(T) = \sum_{x \in T} r(x)$ = potential of tree T

Consider this tree:

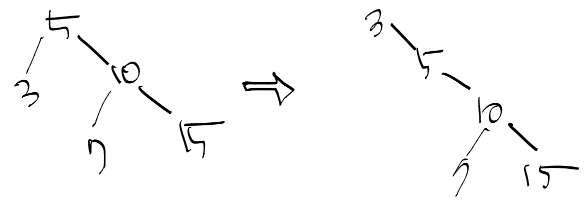


Part (a): Compute $s(x)$ for each node.

Node	$s(x)$	$r(x) = \lfloor \log_2 s(x) \rfloor$
3	1	0
7	1	0
5	3	1
15	1	0
10	5	2

Part (b): Compute $\Phi(T) = \sum r(x)$ for this tree.
$\Phi(T) = 0+0+1+0+2 = 3$

Part (c): Now splay node 3 to the root (this requires zig-zig then zig). Draw the final tree.



Part (d): Compute $s(x)$ and $r(x)$ for each node in the new tree, and find $\Phi(T')$.

	s(x)	r(x)
3	5	2
7	1	0
5	4	2
15	1	0
10	3	1

$\phi(T') = 2+0+2+0+1 = 5$ $\therefore \phi(T') = 5$

Part (e): Did the potential increase or decrease? The Access Lemma says the amortized cost is $O(\log n)$ per splay. How does the potential change help "pay for" the rotations?

Potential increased from 3 to 5 ($\Delta\phi = +2$). But the Access Lemma shows that over many operations, the amortized cost still O(log n) per operation. When we access deep nodes, we pay high actual cost but often decrease potential. When accessing shallow nodes, actual cost is low but potential may increase. It's balances out.

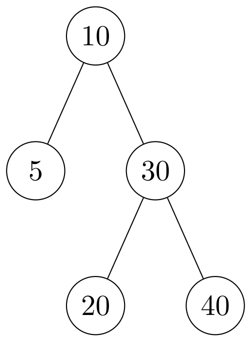
Solutions

Solution to Problem 1 (Rotation Practice)

Solution.

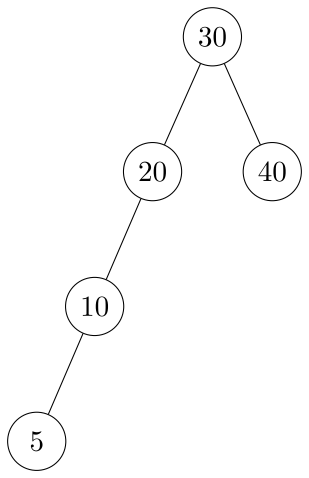
Part (a): Right rotation about 30

Node 10 (left child of 30) becomes the new root.



Part (b): Left rotation about 10

Node 20 (right child of 10) becomes 10's parent.



Part (c): In-order traversal of both results: 5, 10, 20, 30, 40. Same as original!

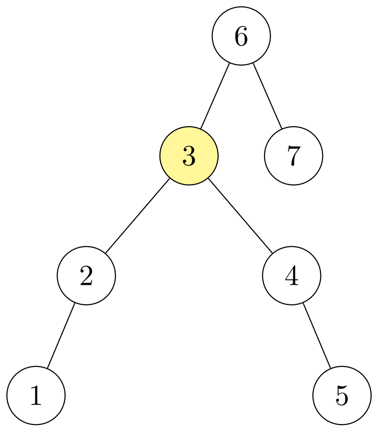
Solution to Problem 2 (Splay Step by Step)

Solution.

Part (a):

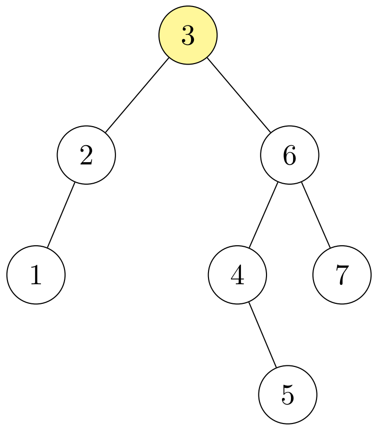
- $x = 3$, $p = 2$, $g = 4$
- Is p the root? No (root is 6)
- x is right child of p, p is left child of g $\Rightarrow$ opposite sides
- Case: **Zig-Zag**

Part (b): After zig-zag (rotate 2, then rotate 4):



Part (c): Next step:

- $x = 3$, $p = 6$ (the root)
- Since p is the root, this is a **Zig** case
- Right rotation about 6:

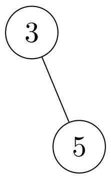


Node 3 is now the root. Done!

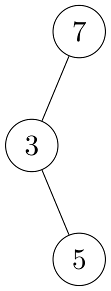
Solution to Problem 3 (Splay Tree Operations)

Solution.

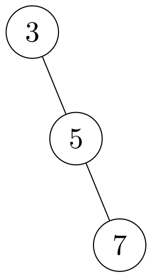
(a) insert(5): Just 5 at root.
(b) insert(3): Insert 3 as left child of 5, then splay (zig):



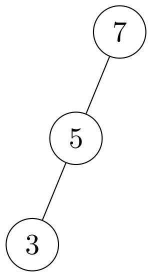
(c) insert(7): Insert as right child of 5, splay (zig-zag):



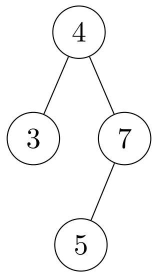
Wait, let me redo this. After inserting 7:



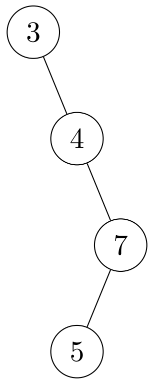
Splay 7: This is zig-zig (7 and 5 both right children). Result:



(d) insert(4): Insert between 3 and 5, splay:
Insert path: 7 → 5 → 3 → right child. After zig-zag then zig:



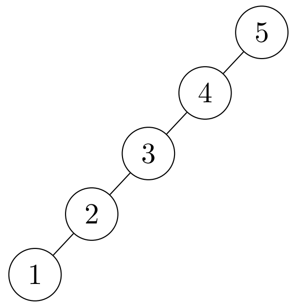
(e) find(3): Search finds 3 (left child of root), splay (zig):



Solution to Problem 4 (Why Naive Move-to-Root Fails)

Solution.

Part (a): Naive move-to-root for node 5 (four left rotations):
After all rotations, we get:



Part (b): It's still a stick! Just in the opposite direction. No improvement in balance.
Part (c): Splay(4) on the new tree:

- 4 is left child of 5 (root), so this is **Zig**
- After zig, 4 is root, 5 is right child

Then we'd need to continue if there were more nodes below...
Part (d): With proper splaying using zig-zig (grandparent first), the tree becomes more balanced. The key insight: zig-zig rotates the grandparent first, which "folds" the long path in half, reducing depth much faster than naive rotations.

Solution to Problem 5 (Potential Analysis)

Solution.

Part (a): Computing sizes and ranks:

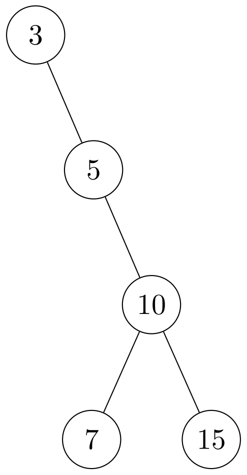
Node	$s(x)$	$r(x) = \lfloor \log_2 s(x) \rfloor$
3	1	0
7	1	0
5	3	1
15	1	0
10	5	2

Part (b): $\Phi(T) = 0 + 0 + 1 + 0 + 2 = 3$

Part (c): Splay(3): First zig-zig (3-5-10), then... wait, let's check:

- $x = 3$, $p = 5$, $g = 10$ (root)
- 3 is left child of 5, 5 is left child of 10 $\Rightarrow$ both left $\Rightarrow$ zig-zig

After zig-zig (rotate 10, then rotate 5), node 3 is at the root:



Part (d): New sizes and ranks:

Node	$s(x)$	$r(x)$
7	1	0
15	1	0
10	3	1
5	4	2
3	5	2

$\Phi(T') = 0 + 0 + 1 + 2 + 2 = 5$

Part (e): Potential increased from 3 to 5 ($\Delta\Phi = +2$). But the Access Lemma shows that over many operations, the amortized cost is still $O(\log n)$ per operation. When we access deep nodes, we pay high actual cost but often decrease potential. When accessing shallow nodes, actual cost is low but potential may increase. It balances out!

Implementation Hints

Data Structure

Each node needs:

- `key` (and optionally `value`)
- `left`, `right` pointers
- `parent` pointer (makes rotations easier!)

Key Functions to Implement

1. `rotateLeft(x)` and `rotateRight(x)` — basic rotations
2. `splay(x)` — move node x to root using zig/zig-zig/zig-zag
3. `find(key)` — BST search, then splay the last visited node
4. `insert(key)` — BST insert, then splay the new node
5. `delete(key)` — find and splay, then remove (tricky!)

Delete is Tricky!

One approach for `delete(key)`:

1. `find(key)` to splay it to root
2. If found, remove the root
3. If root has two children: splay the max of left subtree, then attach right subtree

Pseudocode for Splay

```
splay(x):
    while x.parent != null:
        p = x.parent
        g = p.parent

        if g == null:           // Zig case
            if x == p.left:
                rotateRight(p)
            else:
                rotateLeft(p)

        else if (x == p.left) == (p == g.left):  // Zig-zig
            if x == p.left:
                rotateRight(g)
                rotateRight(p)
            else:
                rotateLeft(g)
                rotateLeft(p)

        else:                   // Zig-zag
            if x == p.left:
                rotateRight(p)
                rotateLeft(g)
            else:
                rotateLeft(p)
                rotateRight(g)

    root = x
```