

# Binary Search Trees: Splay Trees

Daniel Kane

Department of Computer Science and Engineering  
University of California, San Diego

Data Structures  
Data Structures and Algorithms

# Learning Objectives

- Implement a splay tree.
- Understand the ideas behind the runtime analysis.
- Know some other properties of splay tree runtimes.

# Outline

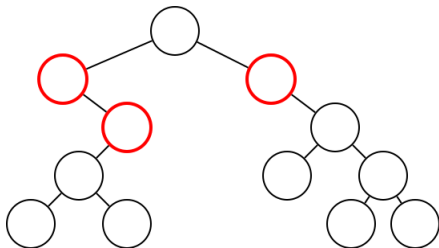
- 1 Non-Uniform Input Sequences
- 2 Analysis
- 3 Operations
- 4 Other Properties

# Non Uniform Inputs

- Search for random elements  $O(\log(n))$   
best possible.

# Non Uniform Inputs

- Search for random elements  $O(\log(n))$  best possible.
- If some items more frequent than others, can do better putting frequent queries near root.



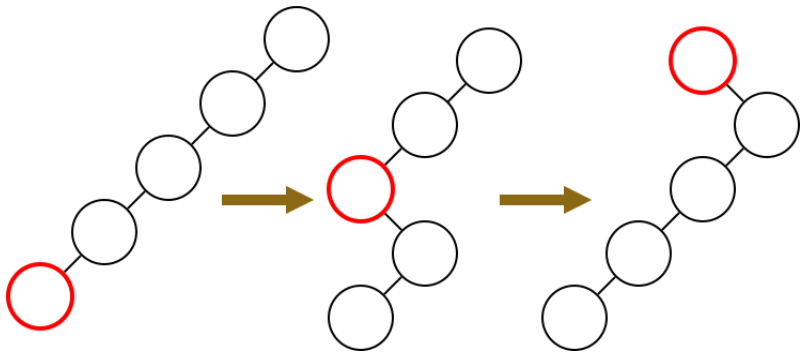
# Idea

Bring query node to the root.

# Simple Idea

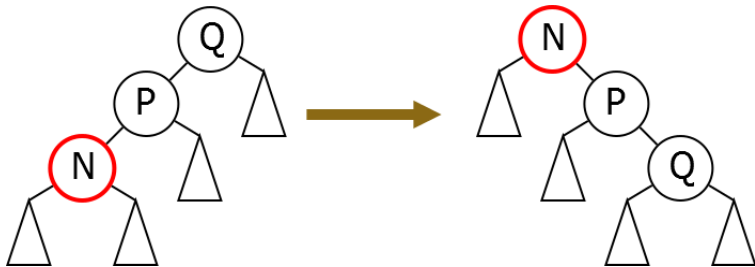
Just rotate to top.

Doesn't work.



# Modification

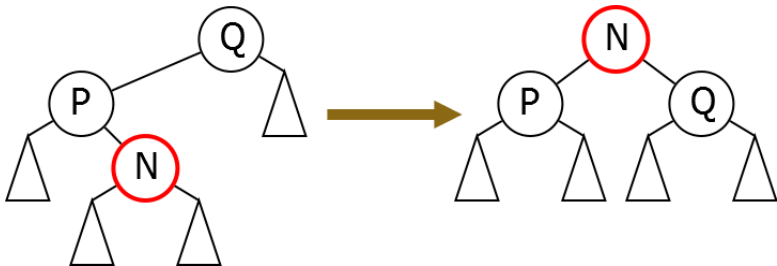
Zig-Zig





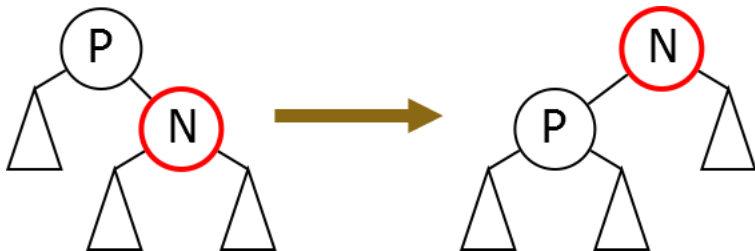
# Modification

Zig-Zag



# Modification

If just below root:  
Zig



# Splay

**Splay( $N$ )**

Determine proper case

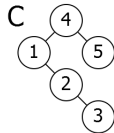
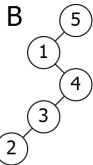
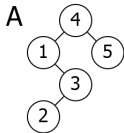
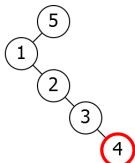
Apply Zig-Zig, Zig-Zag, or Zig as appropriate

if  $N.\text{Parent} \neq \text{null}$ :

    Splay( $N$ )

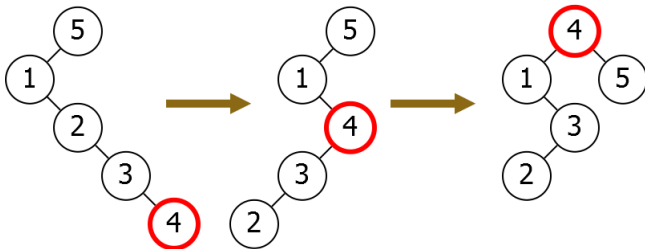
# Problem

Which of the following is the result of playing the highlighted node?



# Problem

Which of the following is the result of splaying the highlighted node?

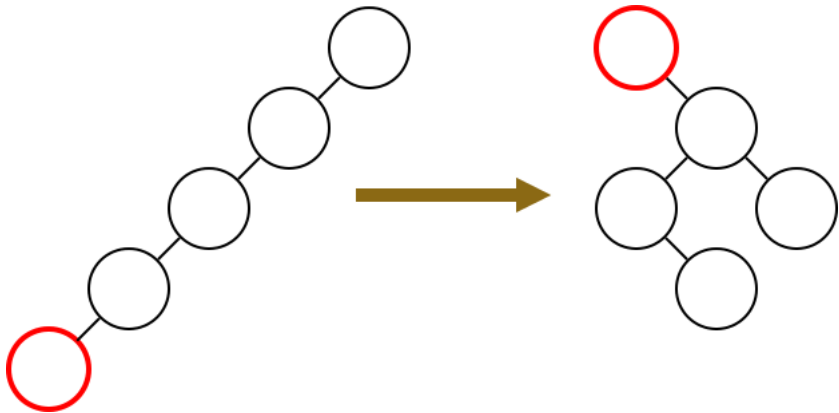


# Outline

- 1 Non-Uniform Input Sequences
- 2 Analysis
- 3 Operations
- 4 Other Properties

# Sometimes Slow

Splay operation is sometimes slow:



# Amortized Analysis

Need to amortize. Pick correct potential function.



# Rank

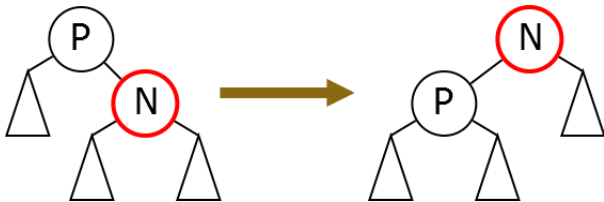
$R(N) = \log_2(\text{Size of subtree of } N).$

Potential function

$$\Phi = \sum_N R(N).$$

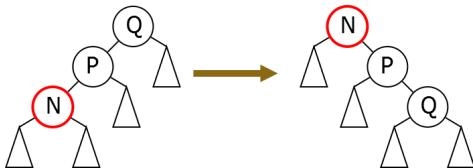
# Zig Analysis

$$\begin{aligned}\Delta\Phi &= R'(N) + R'(P) - R(N) - R(P) \\ &= R'(P) - R(N) \\ &\leq R'(N) - R(N).\end{aligned}$$



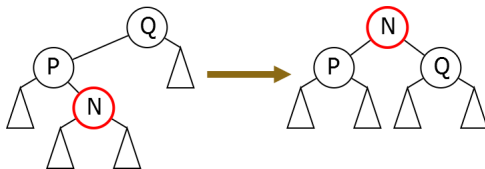
# Zig-Zig Analysis

$$\begin{aligned}\Delta\Phi &= R'(N) + R'(P) + R'(Q) \\ &\quad - R(N) - R(P) - R(Q) \\ &= (R'(P) - R(P)) + (R'(Q) - R(N)) \\ &\leq 3(R'(N) - R(N)) - 2\end{aligned}$$



# Zig-Zag Analysis

$$\begin{aligned}\Delta\Phi &= R'(N) + R'(P) + R'(Q) \\ &\quad - R(N) - R(P) - R(Q) \\ &= (R'(P) - R(P)) + (R'(Q) - R(N)) \\ &\leq 2(R'(N) - R(N)) - 2\end{aligned}$$



# Total Change

$$\begin{aligned}\Delta\Phi &\leq 3(R_k(N) - R_{k-1}(N)) - 2 \\ &\quad + 3(R_{k-1}(N) - R_{k-2}(N)) - 2 + \dots \\ &= 3(R'(N) - R(N)) - \Omega(\text{Depth}(N)) \\ &= O(\log(n)) - \text{Work}\end{aligned}$$

Amortized cost of Find+Splay is  $O(\log(n))$ .

# Outline

- 1 Non-Uniform Input Sequences
- 2 Analysis
- 3 Operations
- 4 Other Properties

# Find

STFind( $k, R$ )

$N \leftarrow \text{Find}(k, R)$

Splay( $N$ )

return  $N$

# Insert

Insert, then splay

$\text{STInsert}(k, R)$

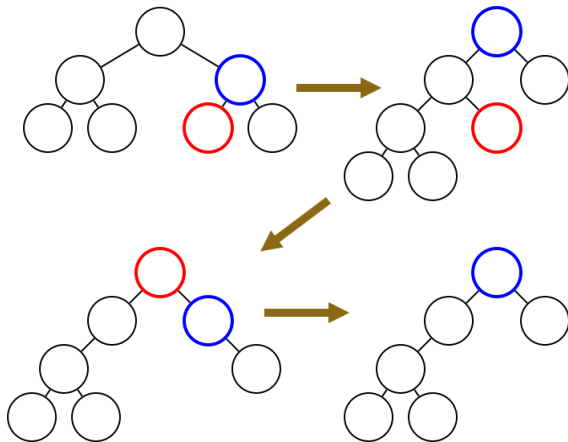
$\text{Insert}(k, R)$

$\text{STFind}(k, R)$



# Delete

Bring  $N$  and successor to top. Deletes easily.



# Delete

STDelete( $N$ )

Splay(Next( $N$ ))

Splay( $N$ )

Delete( $N$ )

# Split

$\text{STSplit}(R, x)$

$N \leftarrow \text{Find}(x, R)$

$\text{Splay}(N)$

split off appropriate subtree of  $N$

# Merge

STMerge( $R_1, R_2$ )

$N \leftarrow \text{Find}(\infty, R_1)$

Splay( $N$ )

$N.\text{Right} \leftarrow R_2$

# Summary

Performs all operations in  $O(\log(n))$  amortized time.

# Outline

- 1 Non-Uniform Input Sequences
- 2 Analysis
- 3 Operations
- 4 Other Properties

# Other Bounds

Splay trees have many other wonderful properties.

# Weighted Nodes

If you assign **weights** so that

$$\sum_N \text{wt}(N) = 1,$$

accessing  $N$  costs  $O(\log(1/\text{wt}(N)))$ .



# Dynamic Finger

Cost of accessing node  $O(\log(D + 1))$  where  $D$  is distance between last access and current access.

# Working Set Bound

Cost of accessing  $N$  is  $O(\log(t + 1))$  where  $t$  is time since  $N$  was last accessed.

# Dynamic Optimality Conjecture

It is conjectured that for any sequence of binary search tree operations that a splay tree does at most a constant factor more work than **the best** search tree for that sequence.

# Conclusion

## Splay Trees

- Easy to implement.
- $O(\log(n))$  time per operation.
- Can be much better if queries have structure.