Binary Search Trees: Splay Trees

Daniel Kane

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

Slides of this chapter does not match the slides shown in the lecture

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.

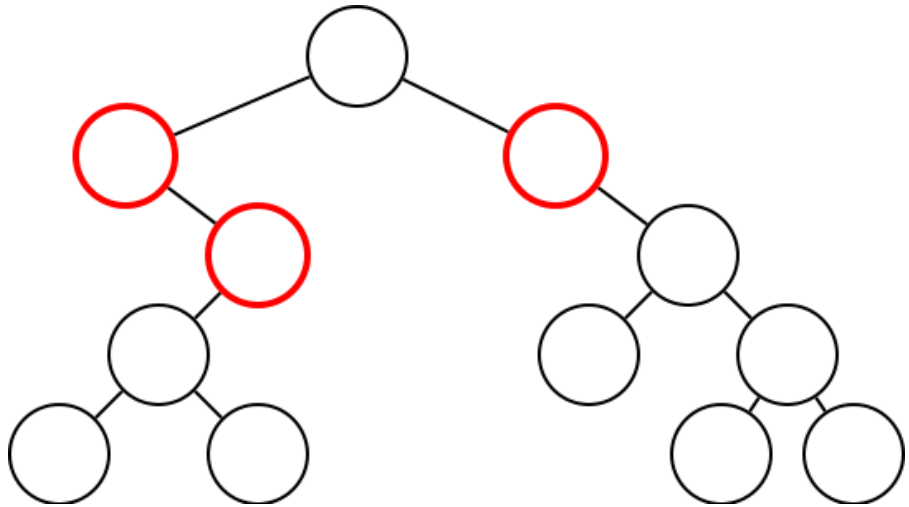
Outline

Non Uniform Inputs

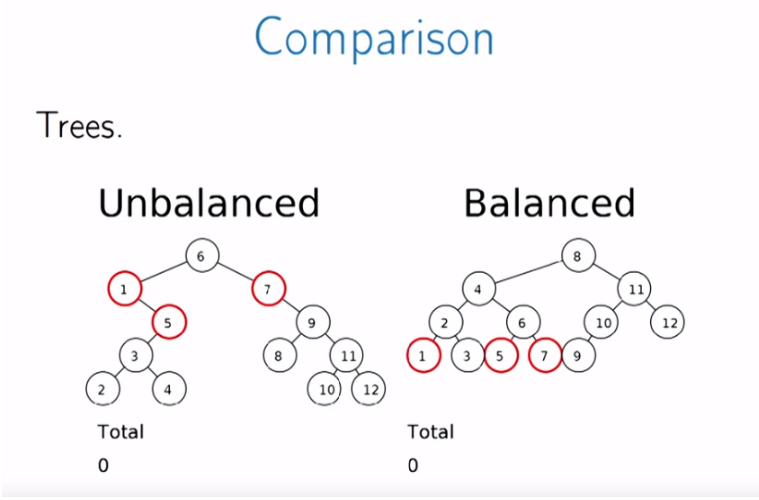
- Search for random elements $O(\log(n))$ best possible.

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.

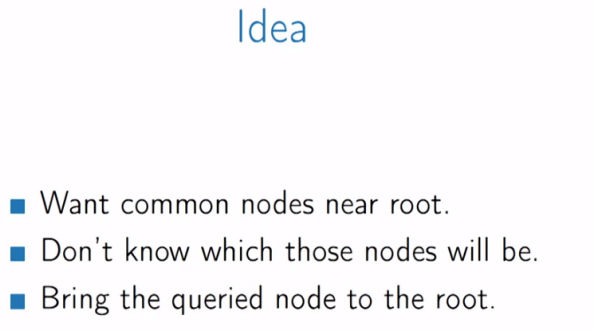


Idea



Bring query node to the root.

As you can see from the figure some nodes are faster to find in an unbalanced tree than balanced tree hence we can bring the frequent node to the root and reduce the access time

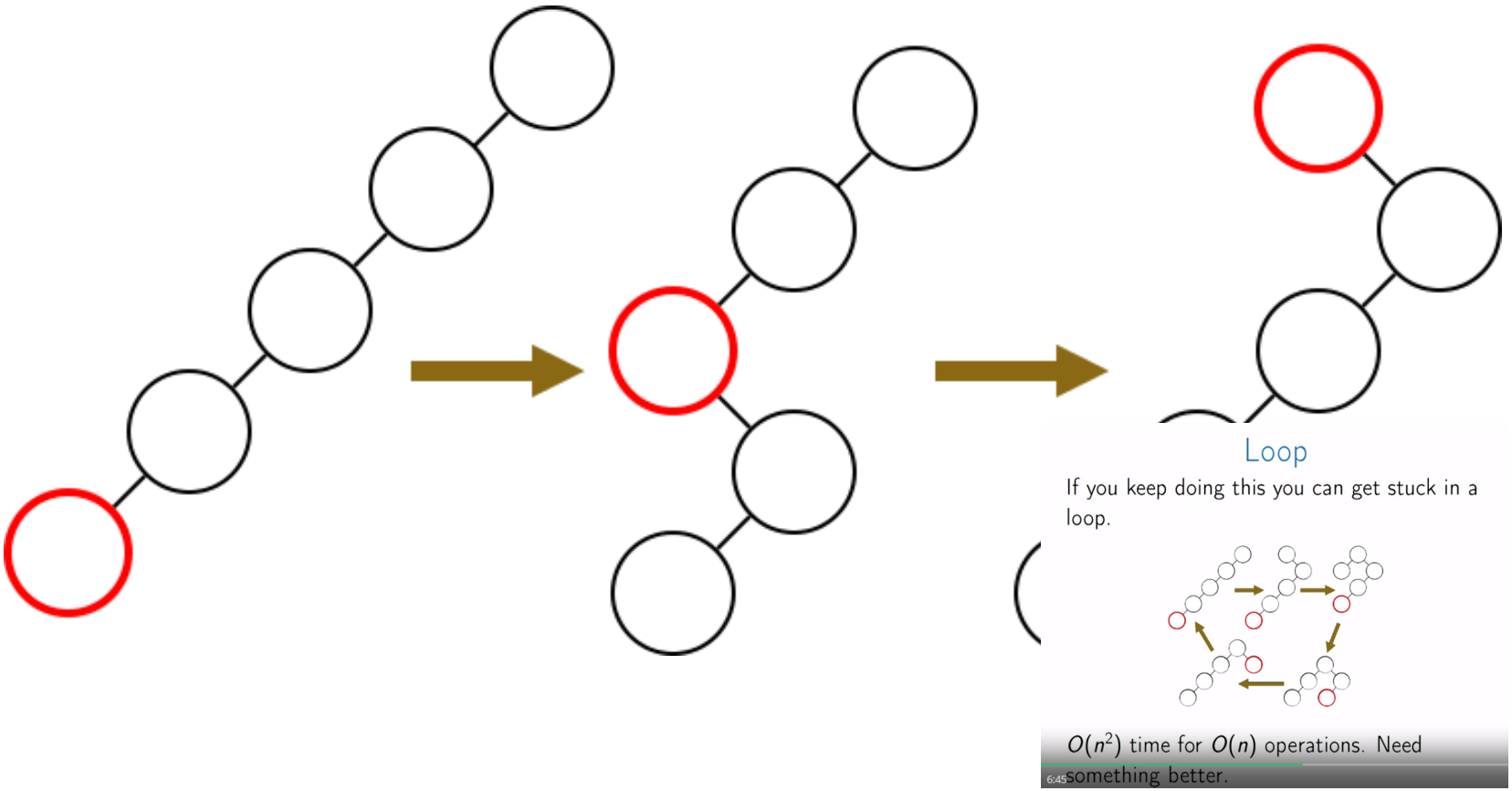


Simple Idea

Just rotate to top.

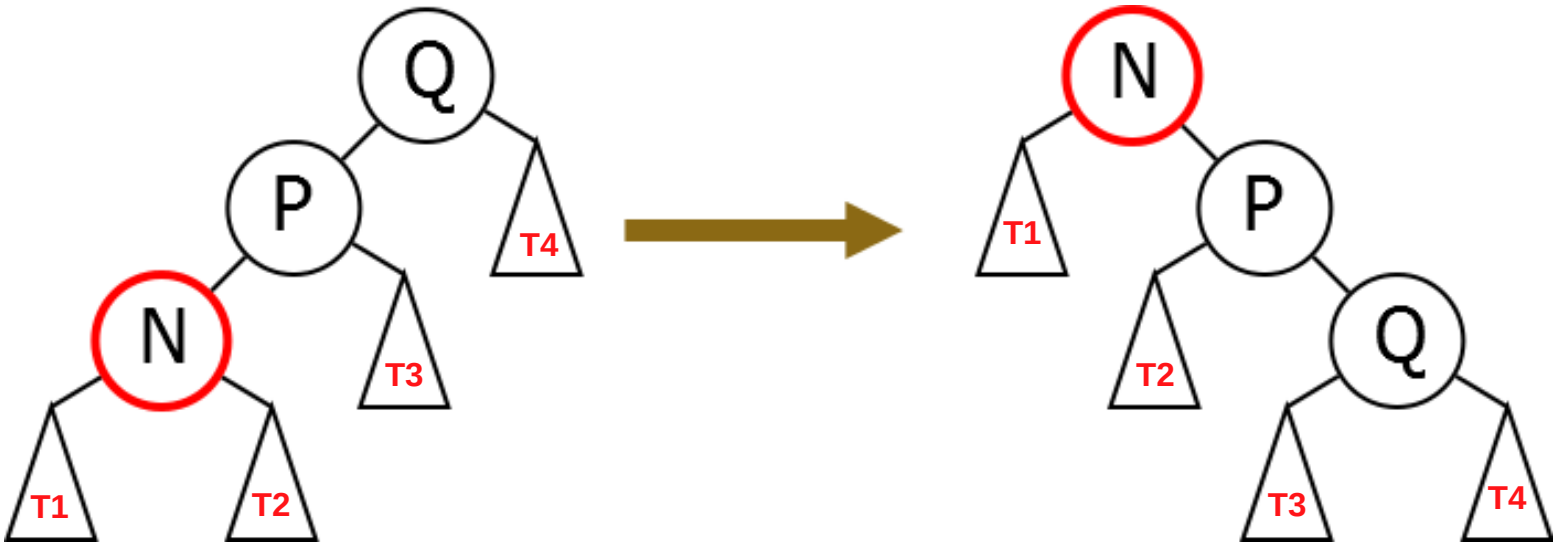
Doesn't work.

because after the rotation the tree is highly unbalanced and also it is very time consuming as it takes O(n^2) time for O(n) operations hence we use the modifications zig-zig, zig-zag or zig as appropriate



Modification

Zig-Zig

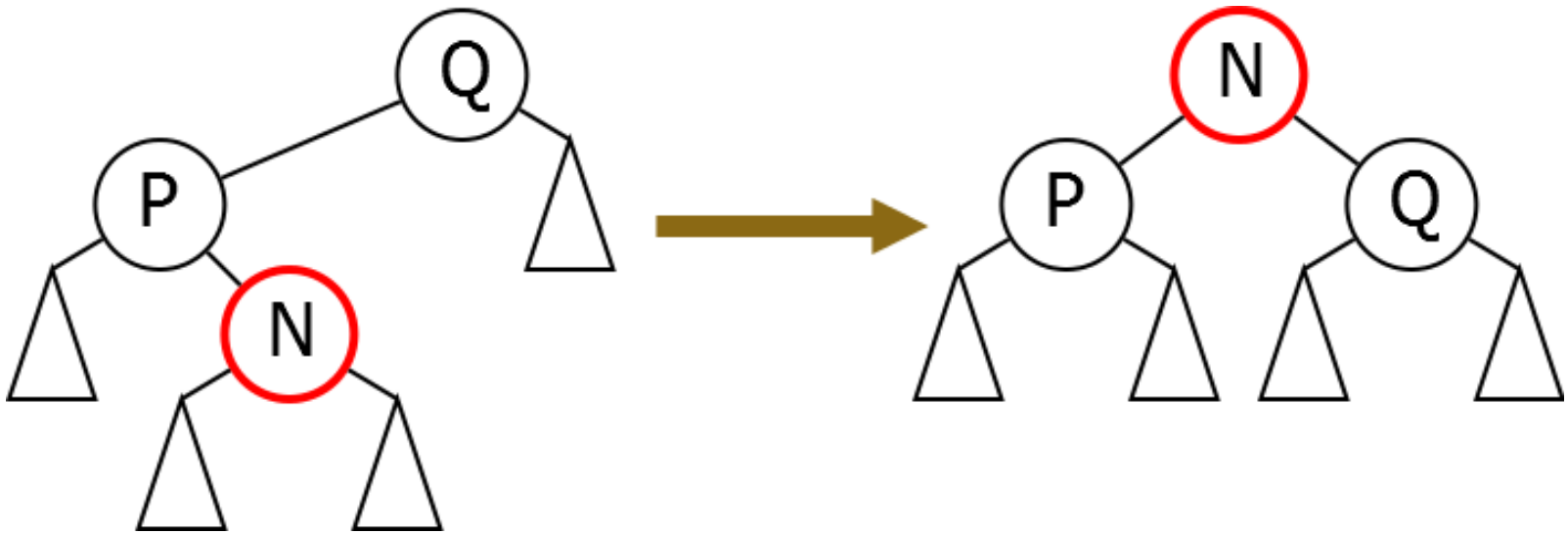


*** In Zig-Zig. Relationship of parent with son is the same as that of grand-parent with parent.**

*** Also notice that the tree won't necessarily get balanced after the splay tree operation.**

Modification

Zig-Zag



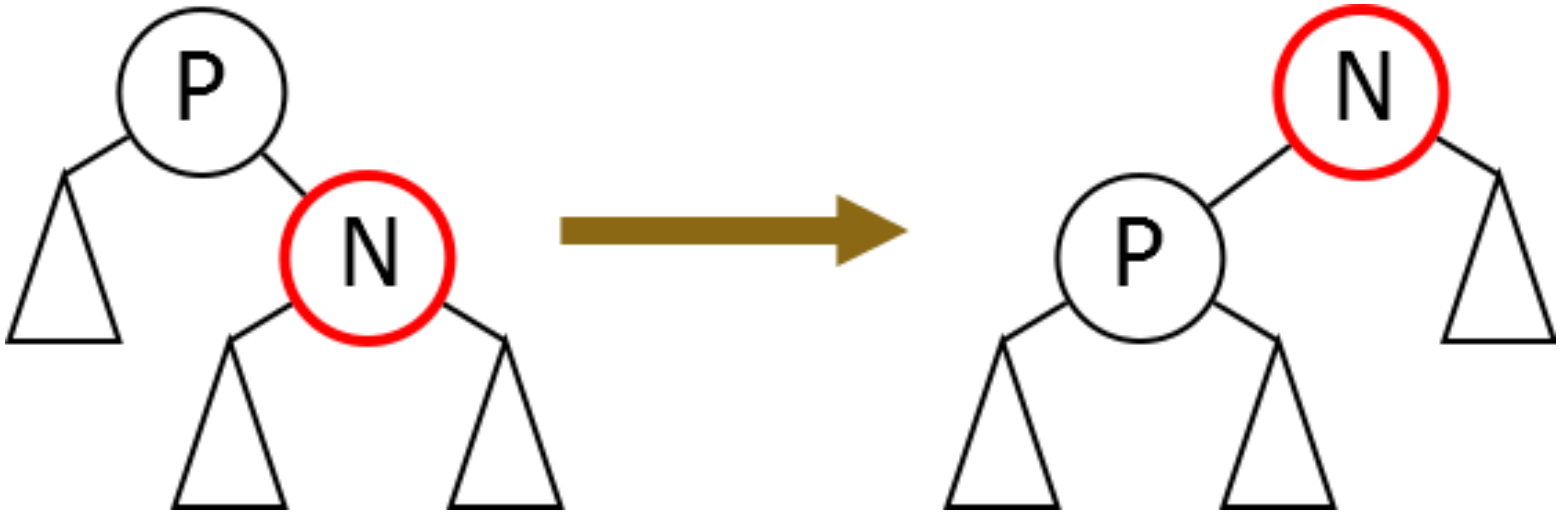
*** In Zig-Zag. Relationship of parent with son is the different than that of grand-parent with parent.**

*** Also notice that the tree won't necessarily get balanced after the splay tree operation.**

Modification

If just below root:

Zig



There is no grandparent in this structure

Splay

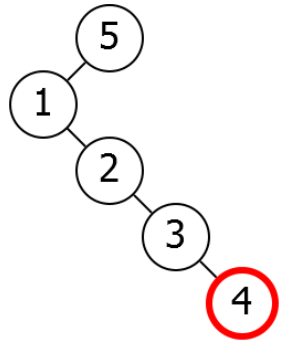
Splay(N)

```
Determine proper case
Apply Zig-Zig, Zig-Zag, or Zig as
appropriate
if N.Parent ≠ null:
  Splay(N)
```

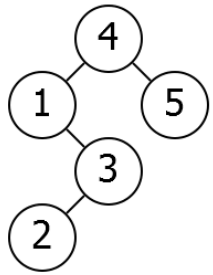
using zig-zig, zig-zag you bring the node near the root

Problem

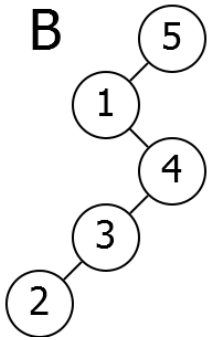
Which of the following is the result of splaying the highlighted node?



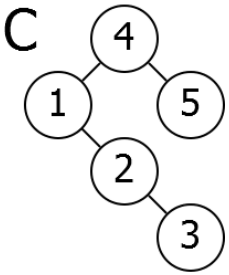
A



B

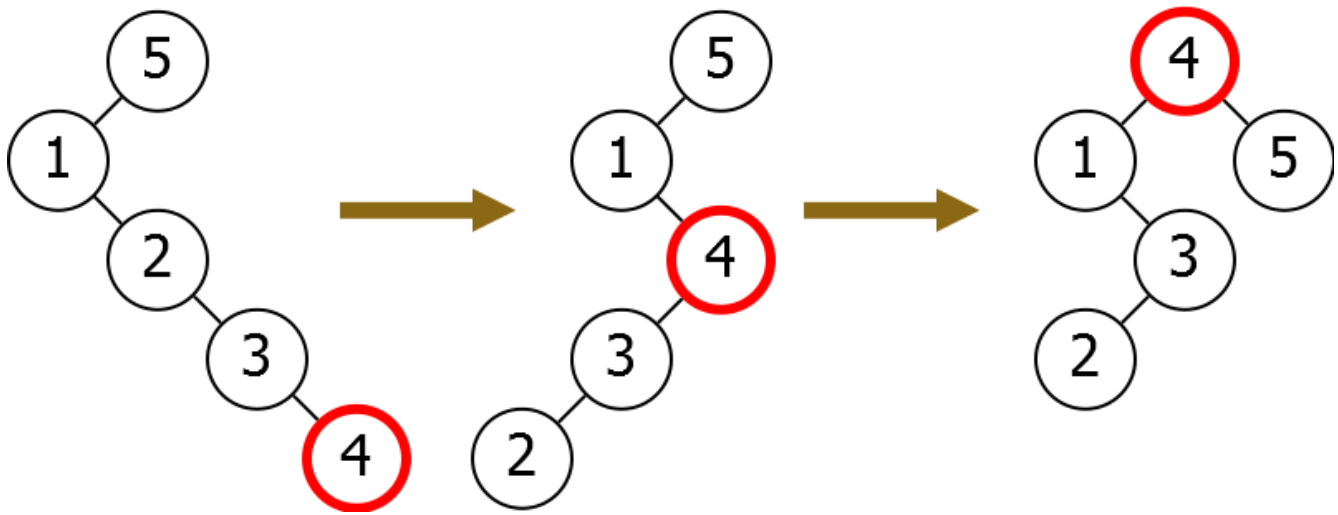


C



Problem

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

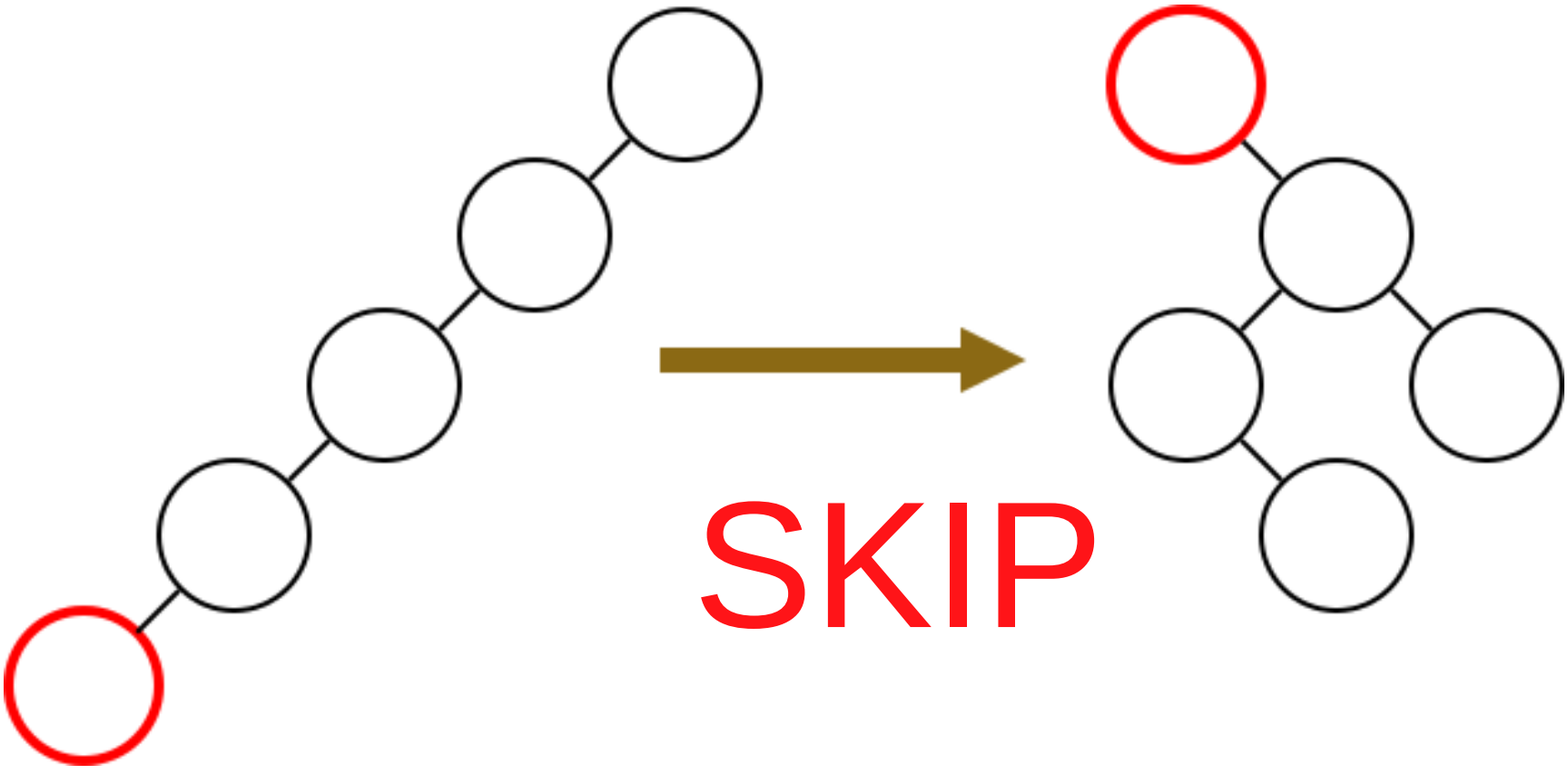


Outline

IGNORE ANALYSIS

Sometimes Slow

Splay operation is sometimes slow:



Amortized Analysis

Need to amortize. Pick correct potential function.

SKIP

Rank

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

Potential function

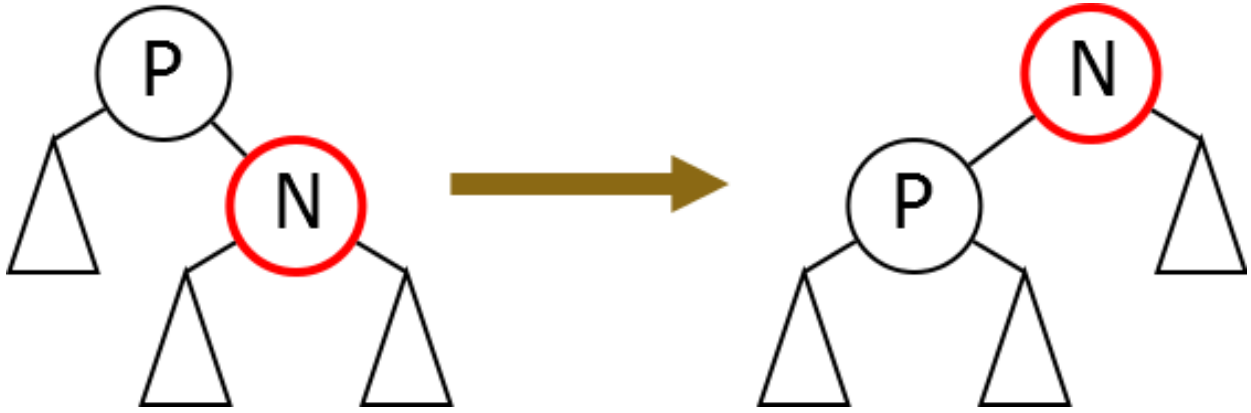
$$\Phi = \sum_N R(N).$$

SKIP

Zig Analysis

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

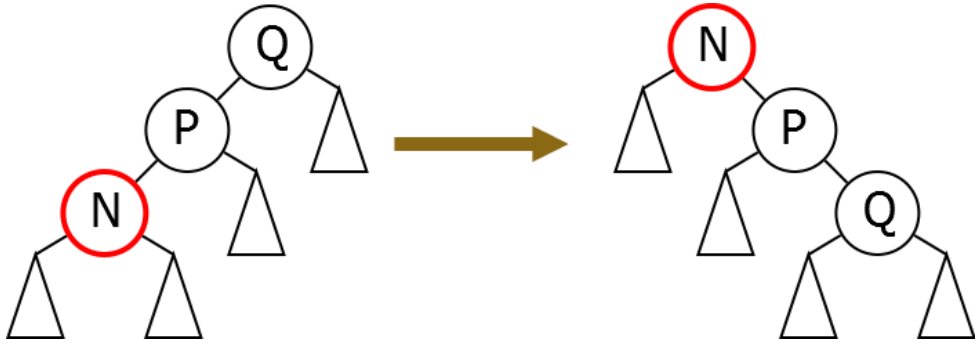
SKIP



Zig-Zig Analysis

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

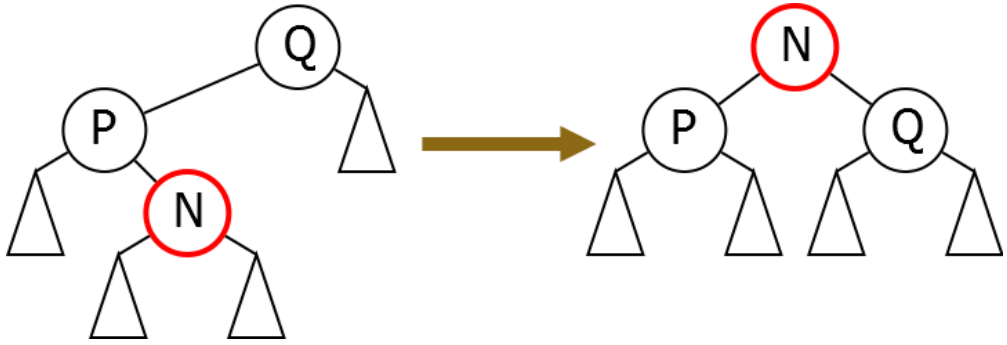
SKIP



Zig-Zag Analysis

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

SKIP



Total Change

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

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

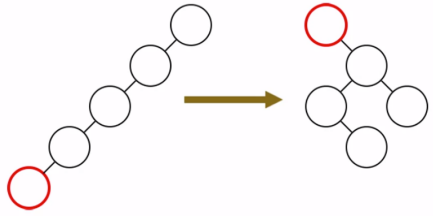
Outline

Last Time

- Idea: bring each query node to the root.
- Introduced splay operation to bring node to root.

Sometimes Slow

Splay operation is sometimes slow:



Splay operation is sometimes slow. When first time you search the node operation takes O(depth) time. however each time you splay a node (and node just rotate to top) you make tree little balanced and hence after sometime operations can be done in log(n) amortized (Average) time

Amortized Analysis

Need to amortize.

Theorem

The amortized cost of doing $O(D)$ work and then splaying a node of depth D is $O(\log(n))$ where n is the total number of nodes.

We will prove this later, but using it we can implement all our operations.

Find

Important Point

Even if you fail to find k, you must still splay the closest node you found. Otherwise your operation did $O(D)$ work with nothing to amortize against.

STFind(k, R)

$N \leftarrow$ Find(k, R)
Splay(N)
return N

Runtime

Suppose node at depth D.

- $O(D)$ time to find N.
- Splay N.
- Amortized cost $O(\log(n))$.

If the node is deep you have to do lot of work but we will not keep on doing lot of work as we are bringing the node closer to the root by splaying it.

Insert

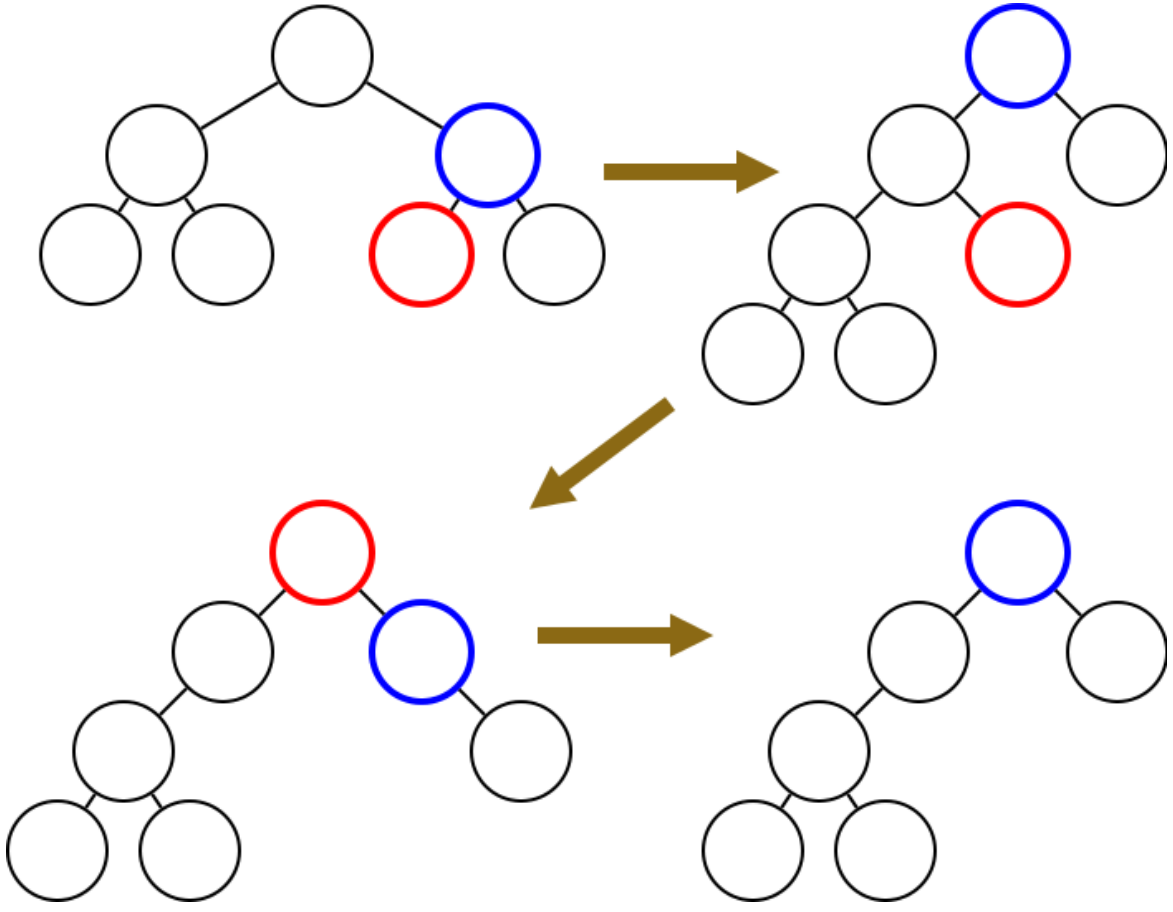
Insert, then splay

`STInsert`(k, R)

`Insert`(k, R)
`STFind`(k, R)

Delete

Bring N and successor to top. Deletes easily.

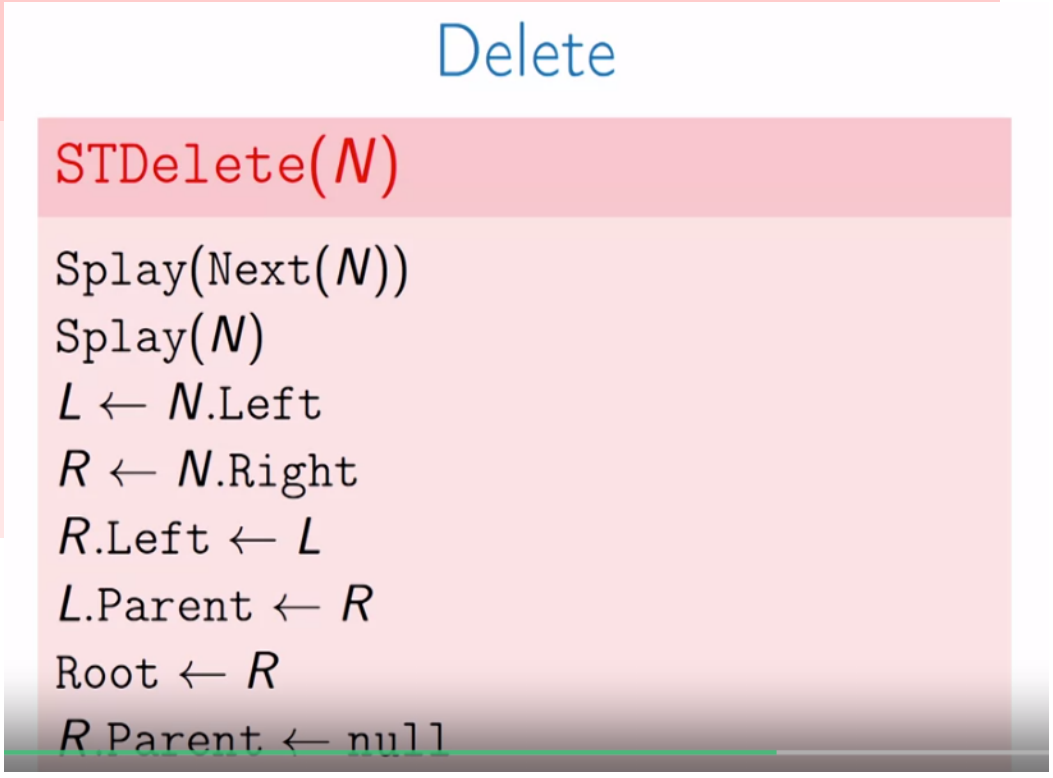


Delete

Consider this complete algo on the right discussed in the lecture

```
STDelete(N)
Splay(Next(N))
Splay(N)
Delete(N)
```

This does not work with the case when the node to be deleted is the largest key that is it does not have any successor. Figure that case on your own



Split

```
STSplit(R, x)
N ← Find(x, R)
Splay(N)
if N.Key > x:
  return CutLeft(R)
else if N.Key < x:
  return CutRight(R)
else
  return N.Left, N.Right
```

Split

CutLeft

```
CutLeft(N)
L ← N.Left
N.Left ← null
L.Parent ← null
return L, N.
```

STSplit(R, x)

$N \leftarrow$ Find(x, R)

Splay(N)

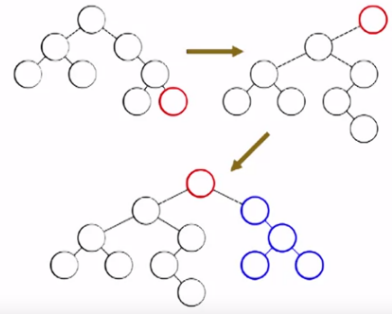
split off appropriate subtree of N

* If the node is found in the tree then split the tree into left and right directly from the node.
* First two cases N.key > x and N.key < x are used when the node we are looking for is not found in the tree in that case the algo will find the next nearest node and promote it to the root and then we will cut left or cut right.

Merge

Merge

Splay largest element of first tree, and stick second tree as child of the root.



Note after splay, root has no right child.

STMerge(R_1, R_2)

$N \leftarrow \texttt{Find}(\infty, R_1)$
$\texttt{Splay}(N)$
$N.\texttt{Right} \leftarrow R_2$

Merge

STMerge(R_1, R_2)

$N \leftarrow \texttt{Find}(\infty, R_1)$
$\texttt{Splay}(N)$
$N.\texttt{Right} \leftarrow R_2$
$R_2.\texttt{Parent} \leftarrow N$

Summary

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

Outline

Other Bounds

Splay trees have many other wonderful properties.

Weighted Nodes

If you assign weights so that

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

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

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.