Binary Search Trees

Robert E. Tarjan
Princeton University &
Intertrust Technologies

Heidelberg Laureate Forum
September 23, 2016

Observations

Over the last 50+ years, computer scientists have developed many beautiful and theoretically efficient algorithms.

But many such algorithms have yet to be used in practice. Some fail when used improperly, or are less efficient than simpler methods with worse theoretical efficiency.

Why?

Implementers, pressed for time, choose the simplest solution that works, or seems to.

Einstein: “Make everything as simple as possible, but not simpler.”

How should (we) theoreticians respond?

Develop and analyze simple methods. The analysis can be complicated, but the algorithm must be simple.

Apply theory to analyze and improve methods used or usable in practice.

Find the structure in empirical datasets and devise algorithms that exploit this structure.

My research goals

Develop and analyze reference algorithms:

algorithms from "the book"

a la "proofs from the book" (Erdős)

Algorithms as simple as possible,

with provable resource bounds

for important input classes,

and efficient in practice

via systematically exploring the design space

A tale of three data structures

- From binary search to binary search trees
- Rebalancing
- Weak AVL (wavl) trees
- Deletion without rebalancing: relaxed AVL (ravl) trees
- Biased access: splay trees
- Dynamic optimality?

Dictionary Problem: Support three operations on a set *S* of items:

Access: find a given item, return its info

Insert: add a new item

Delete: remove an item

Assume items are totally ordered, so that binary search is possible: store in a *binary search tree*: one item per node, in symmetric order

Can also do range queries & other order-based operations; can't use hashing

Binary Search

Maintain set S in sorted order.

To find x in S:

If S empty, stop (failure).

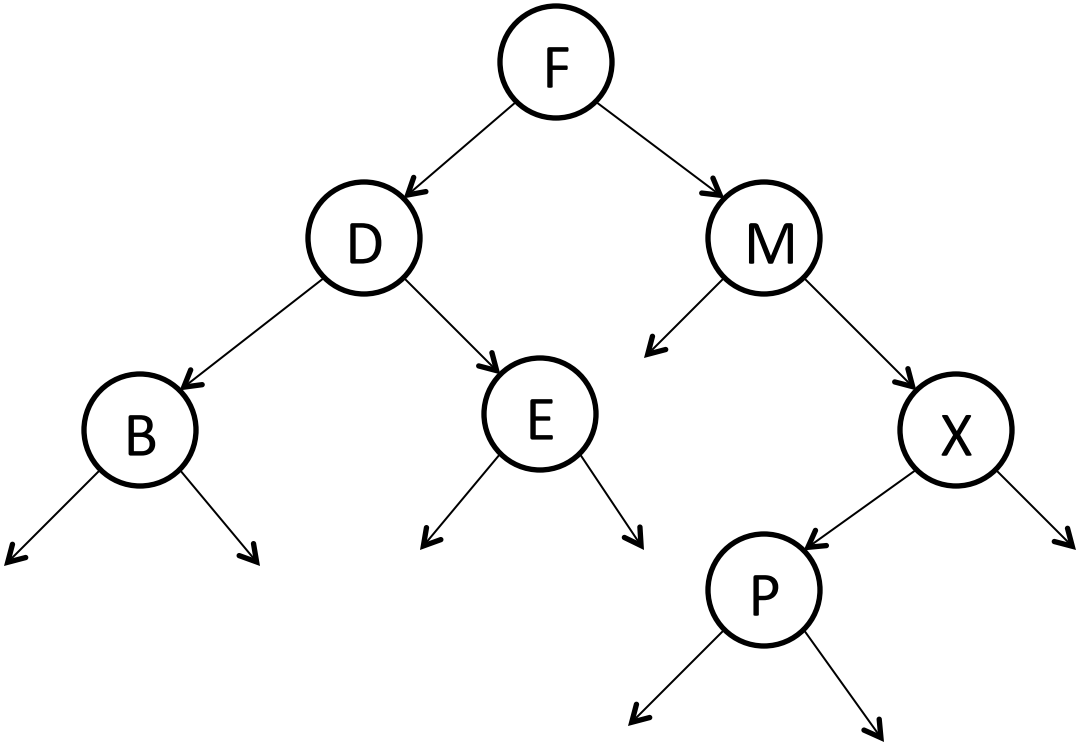
If S non-empty, compare x to some item y in S.

If $x = y$, stop (success).

If $x < y$, search among elements in $S < y$

If $x > y$, search among elements in $S > y$

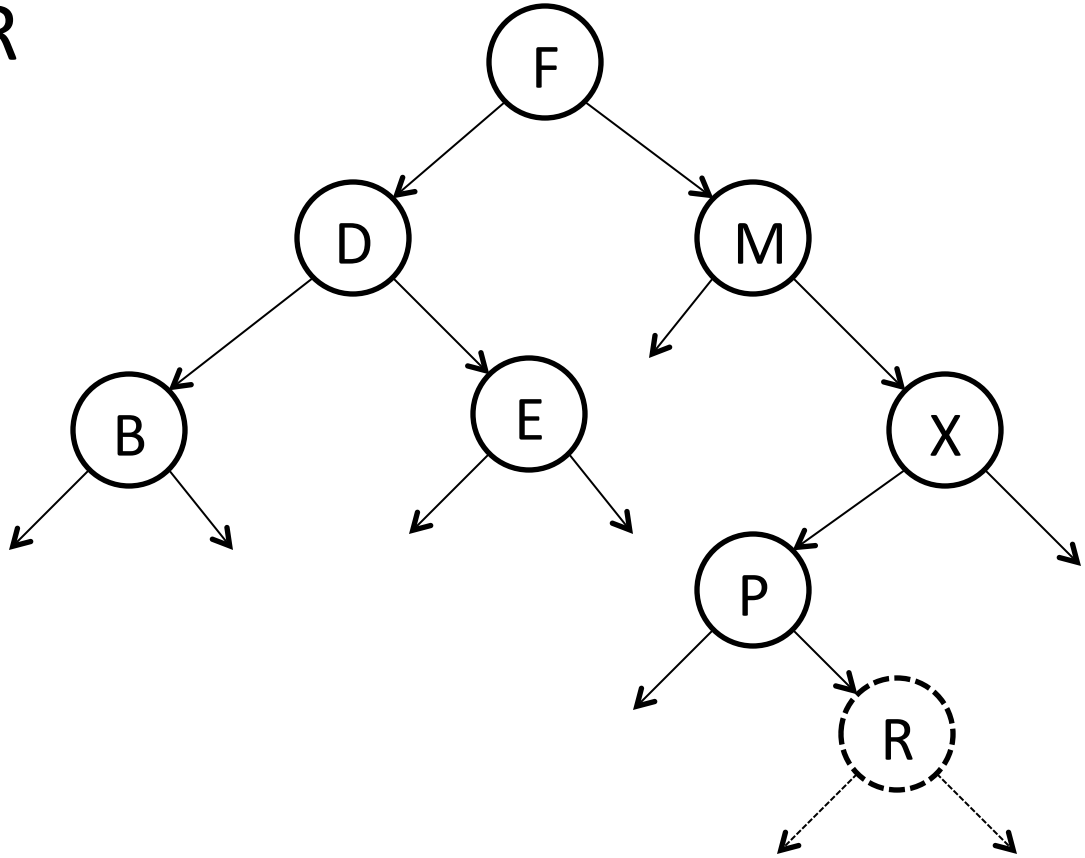
Data Structure:
Binary Search Tree



Insertion

Search. Replace missing node by item.

Insert R



Deletion

Find item. Remove node. Repair tree.

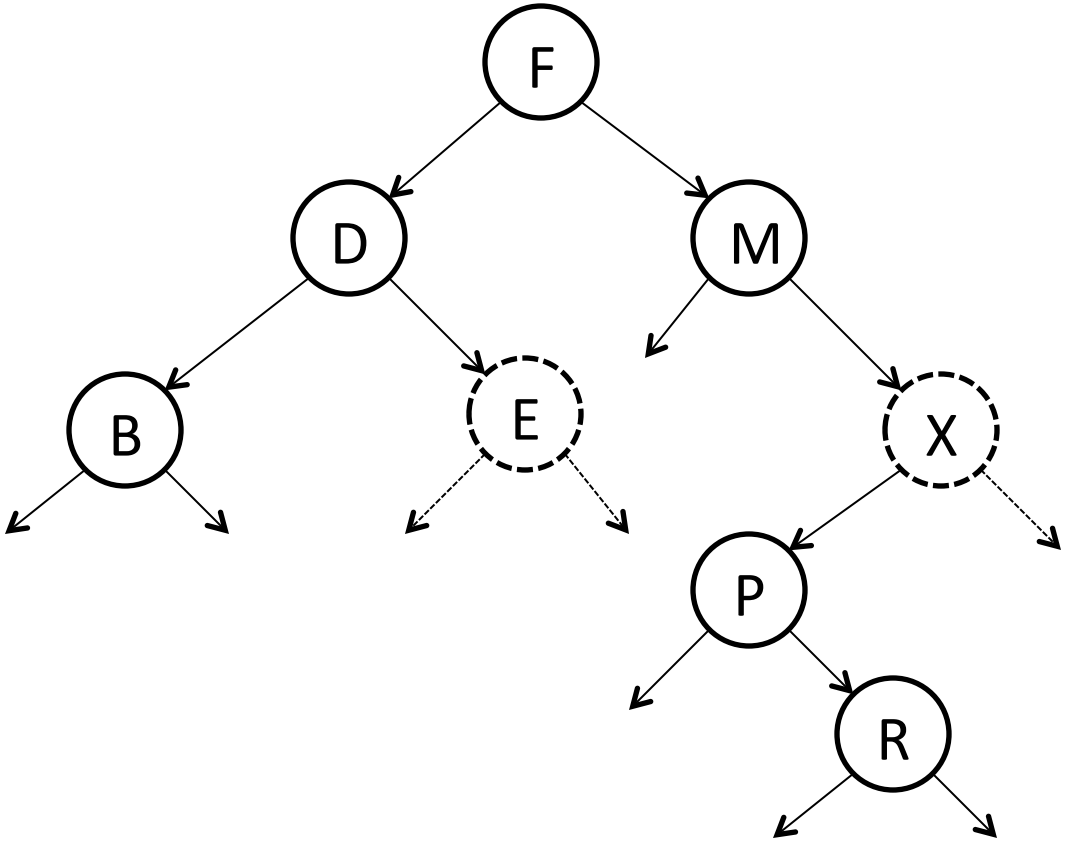
If leaf (no children), delete node.

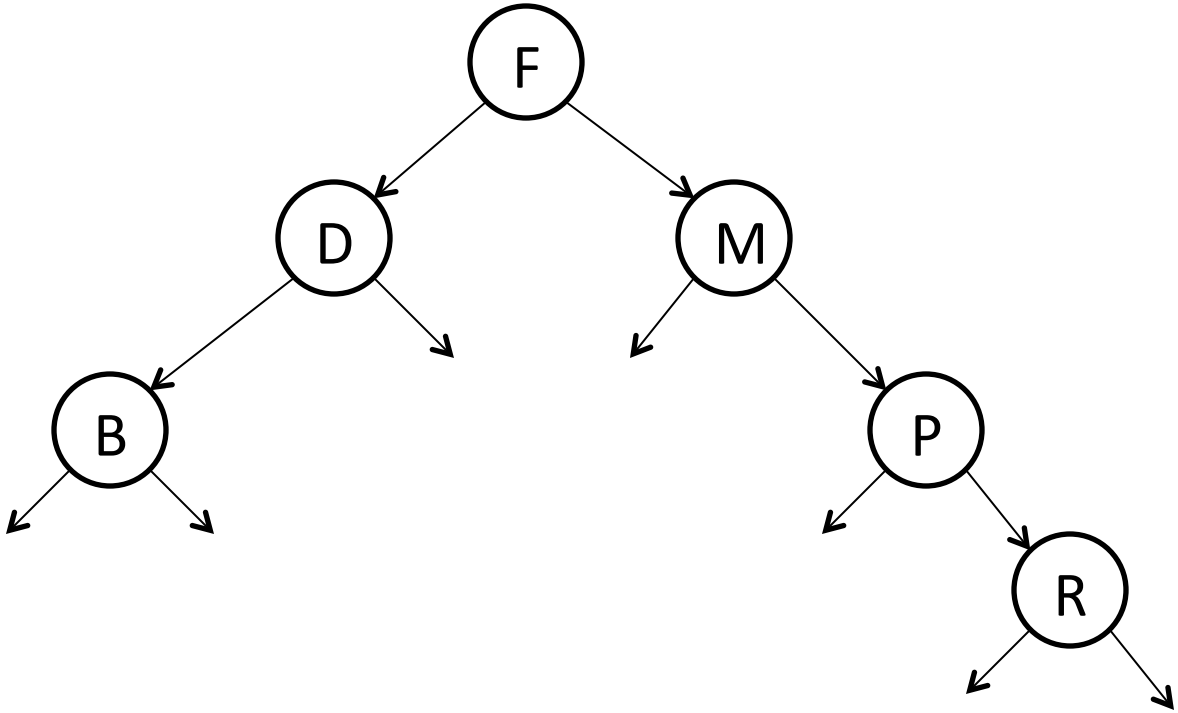
If unary (one child), replace by other child.

If binary?

Delete E

Delete X



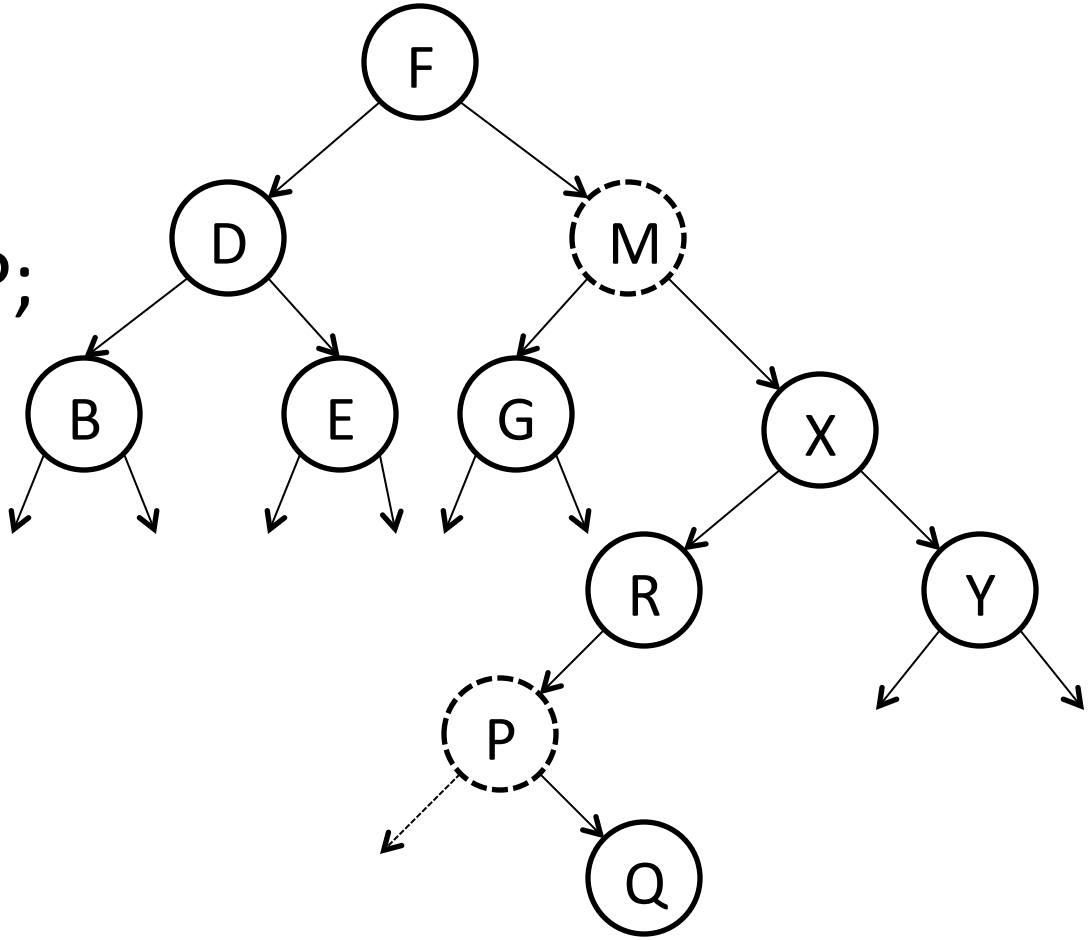

F
D
M
B
P
R

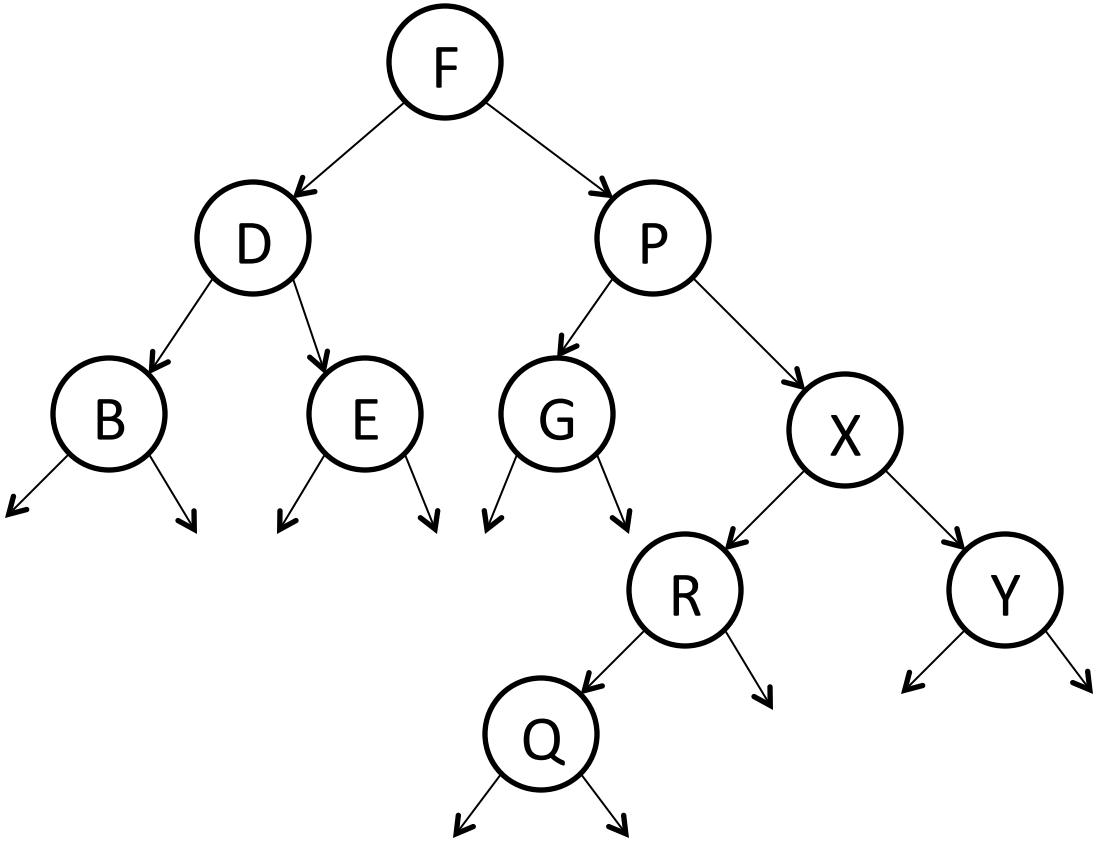
If binary, swap with next item. Now in leaf or unary node; delete. To find next item, follow left path from right child.

Delete M:

Swap with P;

delete.




F
D
P
B
E
G
X
R
Y
Q

Time Per Operation

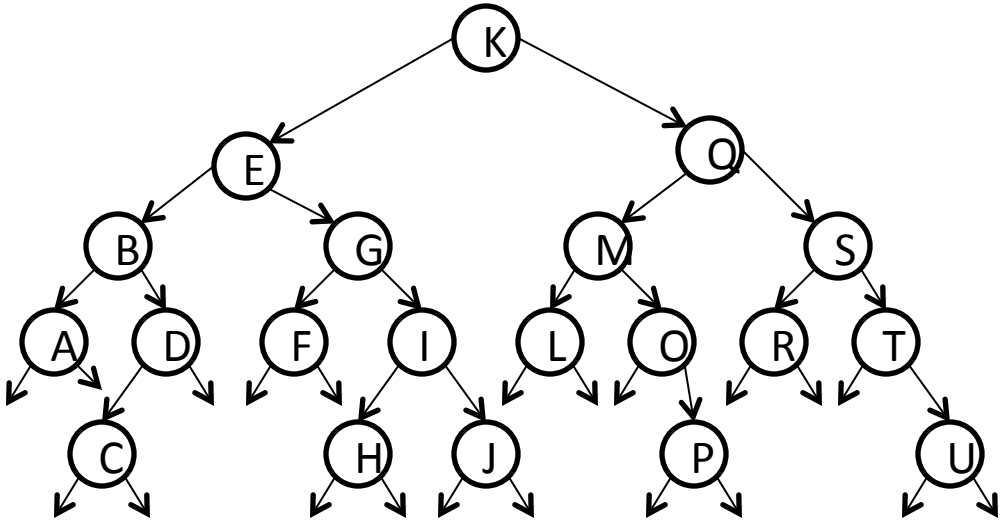
Proportional to depth of deepest node reached during operation (length of path from root)

Goal: minimize tree depth

Best Case

All leaves have depths within 1: depth $\lfloor \lg n \rfloor$

(*lg*: base-two logarithm) *n* = #items



Can achieve if tree is static (insertion order chosen by implementation, no deletions)

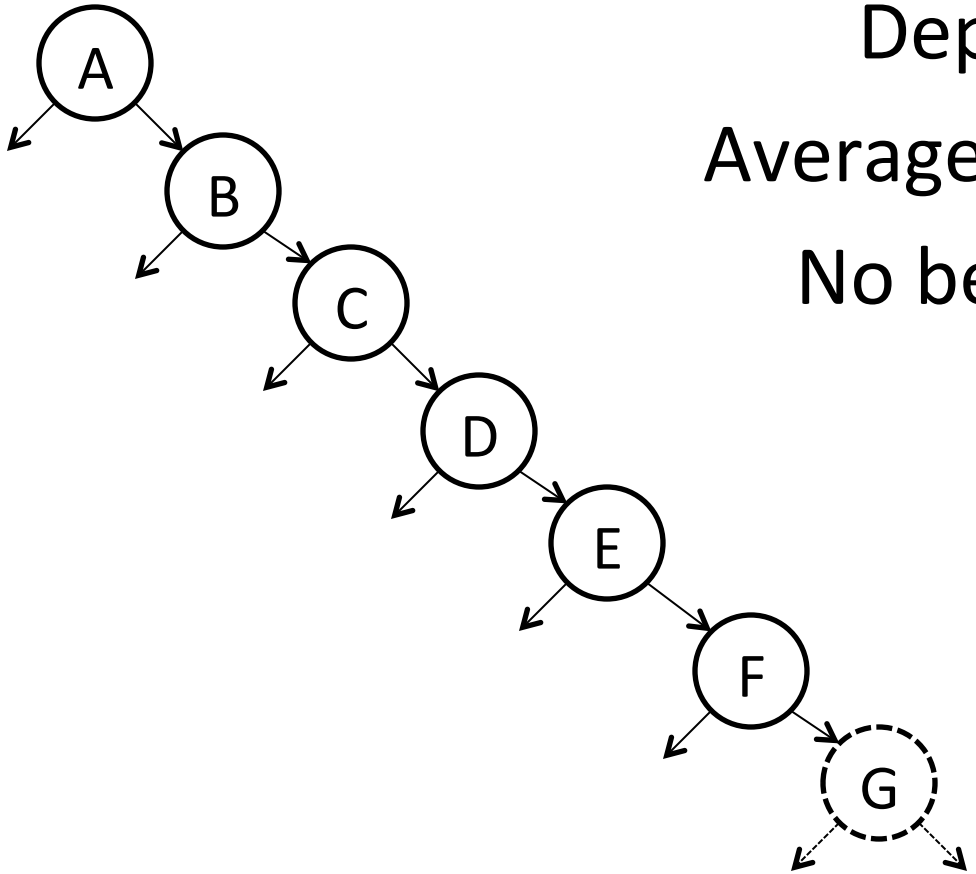
Average Case

Starting with an empty tree, if n items are inserted in random order, expected tree depth (access time) is $O(\log n)$

Worst Case

Natural but bad insertion order: sorted.

Insert A, B, C, D, E, F, G,…



Depth of tree is $n - 1$.

Average access time is ~n.

No better than a list!

Balanced Trees

Rebuild tree after each insert/delete? O(n) time

Want update times as well as search times to be O(lg$n$).

Can't keep all leaves within 1 in depth. Need more flexibility.

- How to define balance?
- How to restore balance after an insertion or deletion?

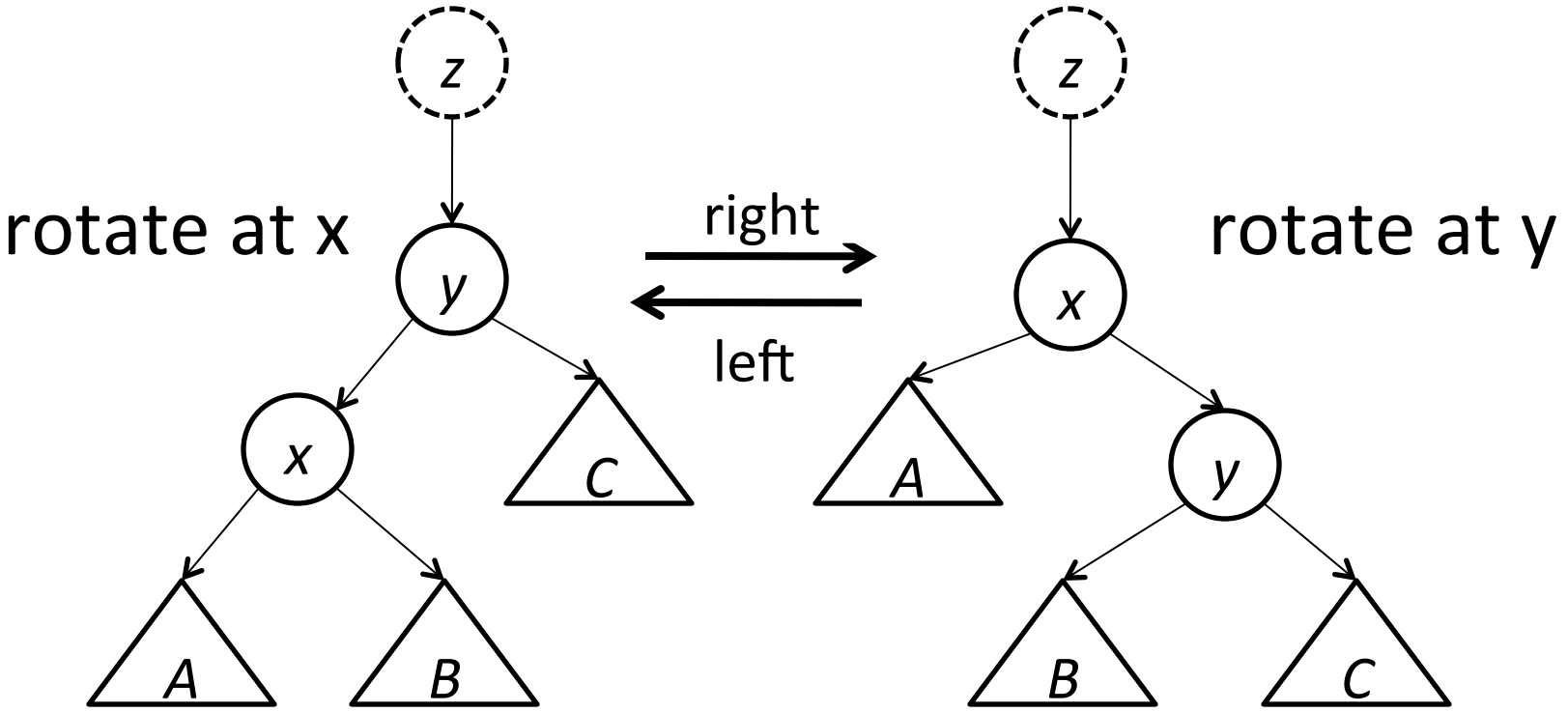
Restructuring primitive: *Rotation*

Preserves symmetric order (searchability).

Changes some depths.

Complete: can transform any tree into any other tree on the same set of items.

Local: takes O(1) time.


rotate at x
z
y
x
A
B
C
right
left
z
x
A
y
B
C
rotate at y

Balance

Keep siblings similar.

Height balance: keep heights of siblings not too far apart (constant difference or constant ratio).

Weight balance: keep sizes of siblings (#nodes is subtrees) not too far apart (constant ratio).

AVL Trees

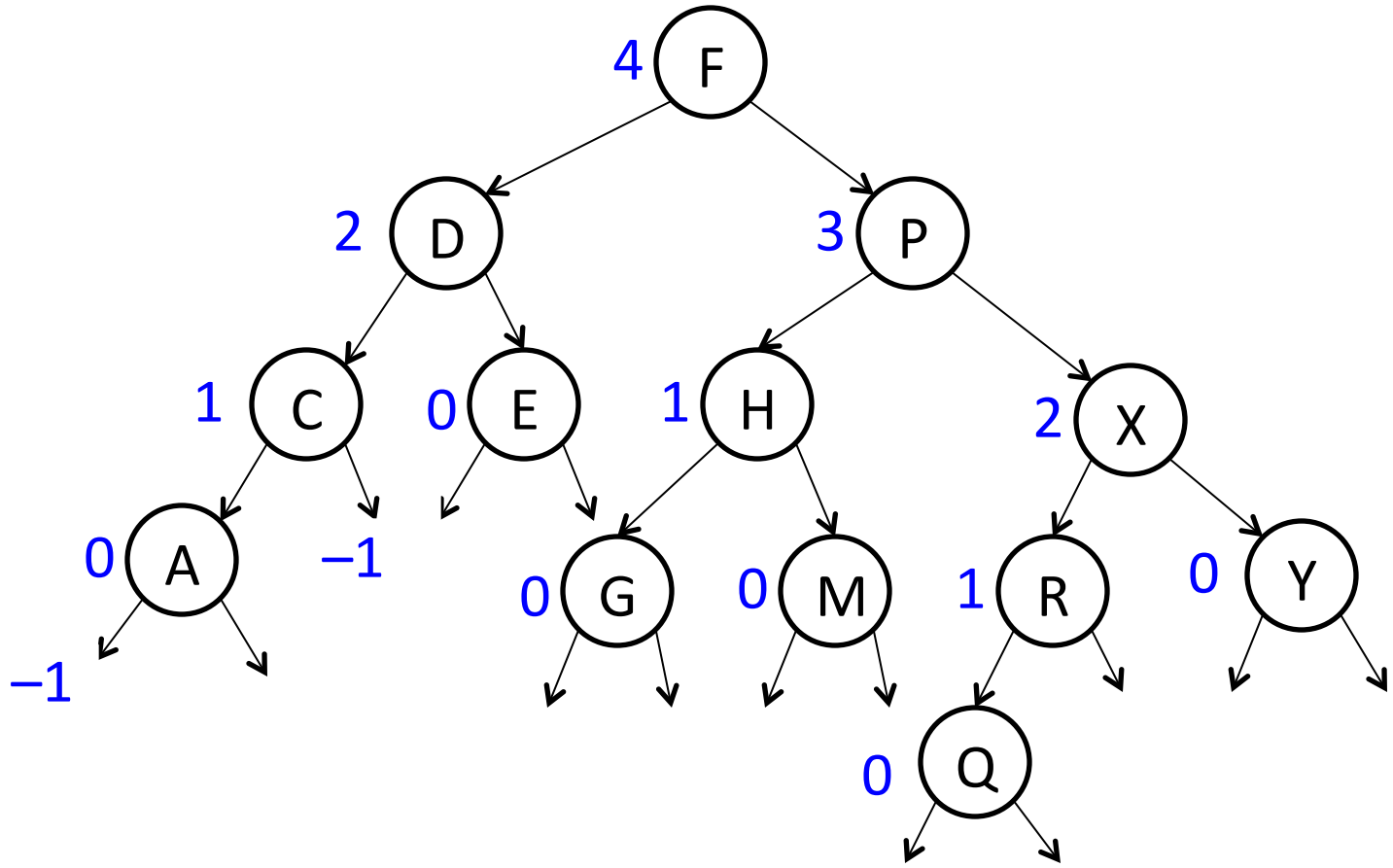
Adelson-Velskey & Landis, 1961:

Any two siblings differ in height by at most 1

height $\leq \log_{\varphi} n < 1.44043 \lg n$ ($\varphi = (1 + \sqrt{5})/2$)

After insertion, rebalance by walking up access path updating heights and doing rotations (at most two).

An AVL Tree. Numbers left of nodes are heights.
missing nodes have $h = -1$ (two shown)



AVL Trees: Deletion?

After deletion, walk up access path and rebalance by updating heights and doing rotations.

But: ~log*n* rotations per deletion, worst-case

Expensive insertions and deletions can alternate: each update takes ~log*n* time

Weak AVL (wavl) Trees

Idea: weaken balance to speed up rebalancing

(Haeupler, Sen, & Tarjan, 2009)

nodes have ranks (proxy for heights)

rank of a missing node rank = –1

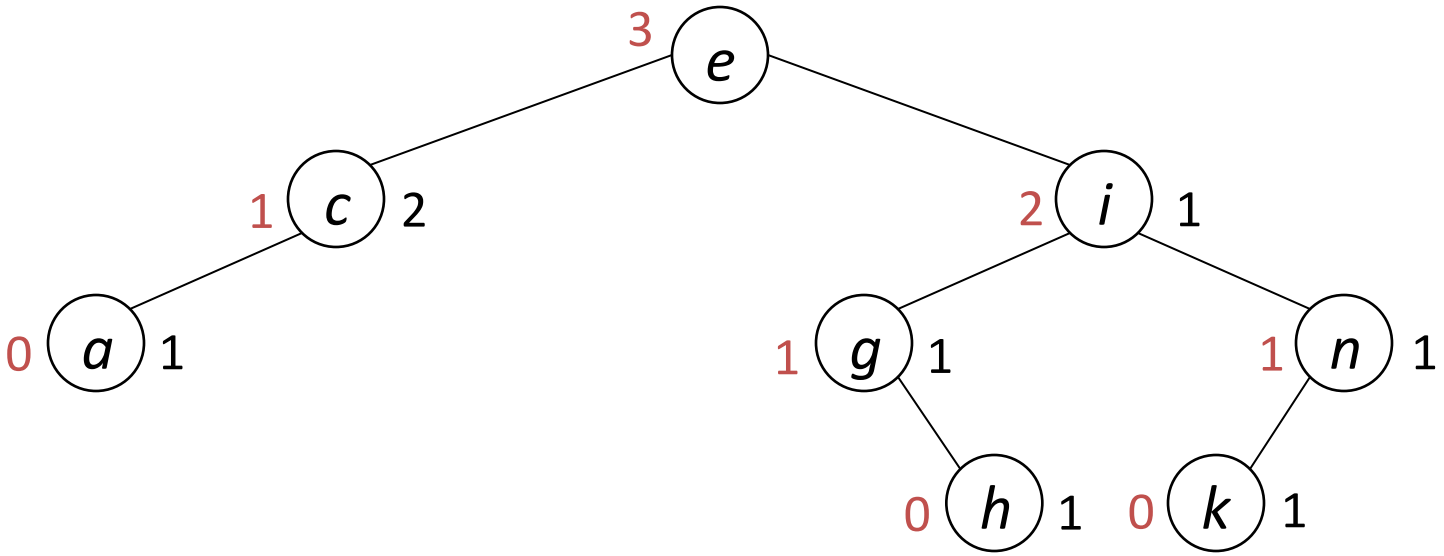
rank of a leaf = 0

$1 \le \text{rank}(\text{parent of } x) - \text{rank}(x) \le 2$

$h \le 2 \lg n$

Example of a wavl tree

ranks in red, rank differences in black

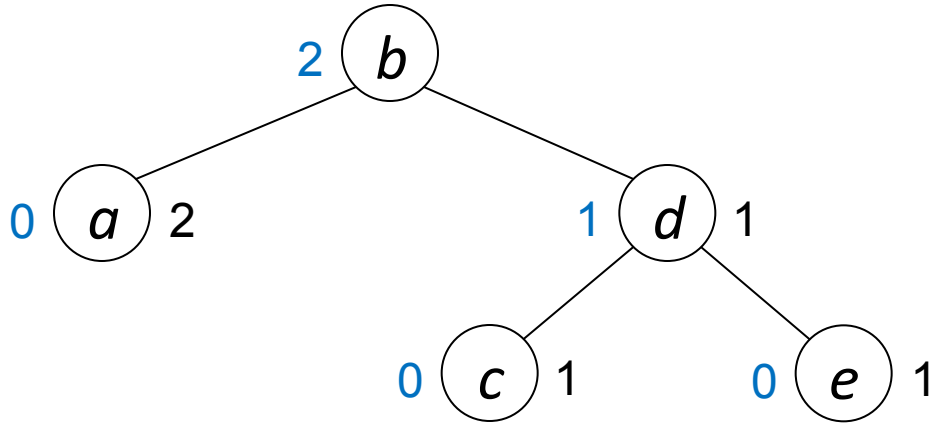


(Weak) AVL Tree Insertion

Give a new node (leaf) rank = 0.

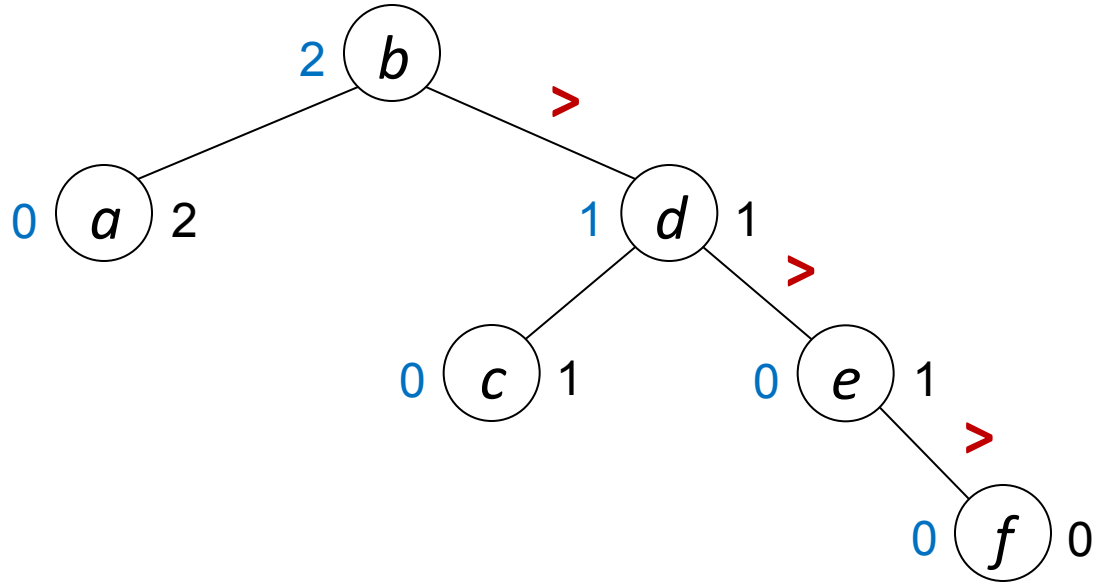
While some node has rank equal to that of its parent, do a rebalancing step (6 cases)

Insertion example



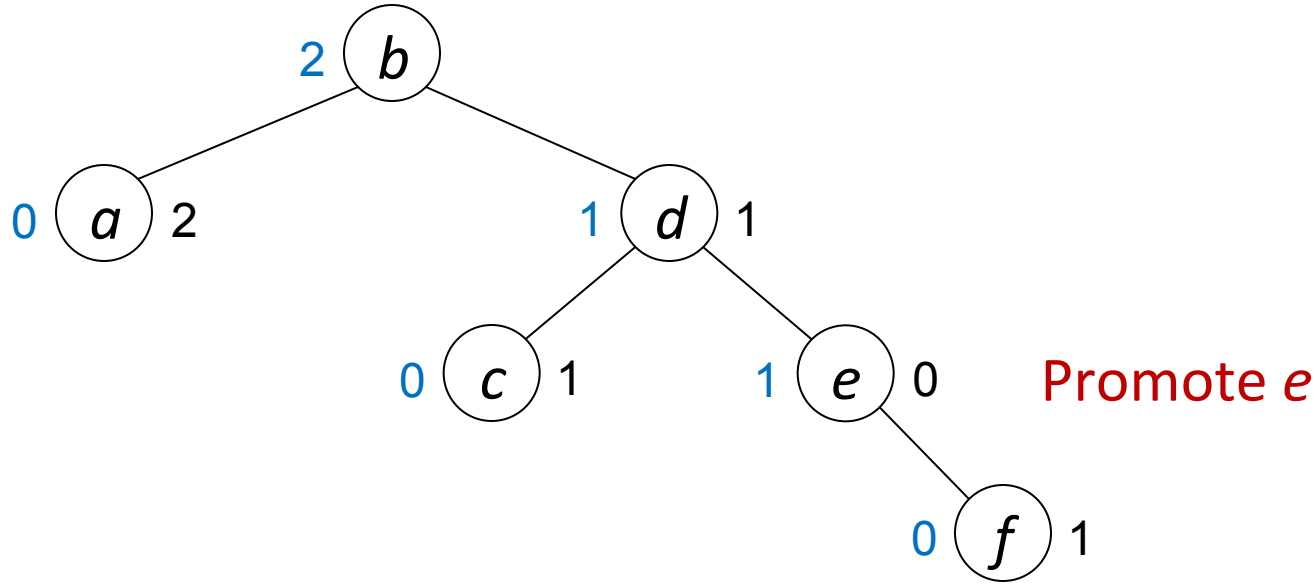
Insert *f*

Insertion example



Insert *f*

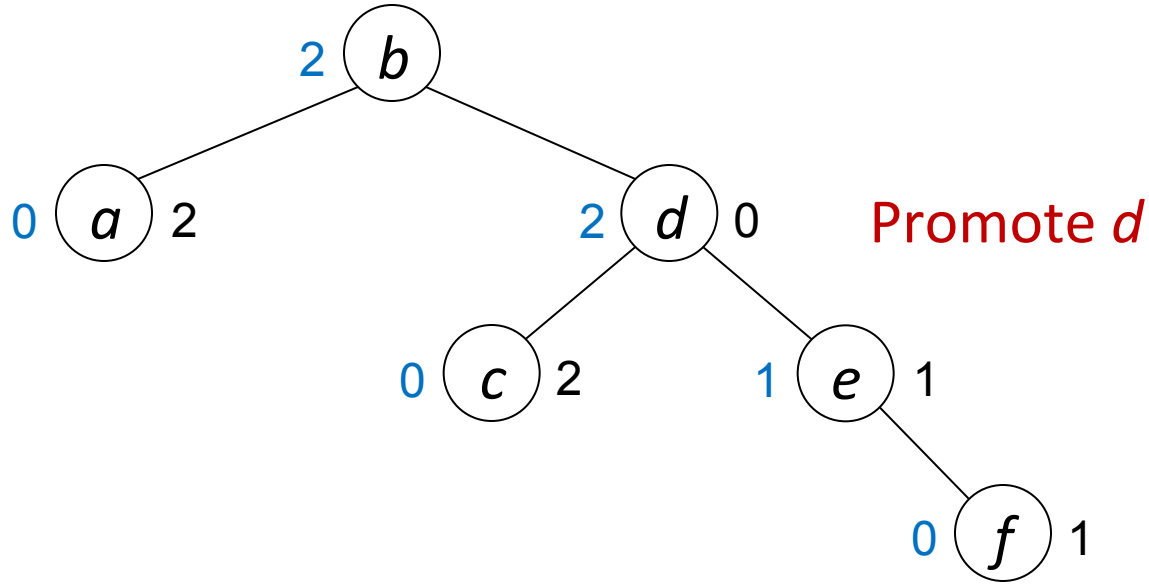
Insertion example



Promote *e*

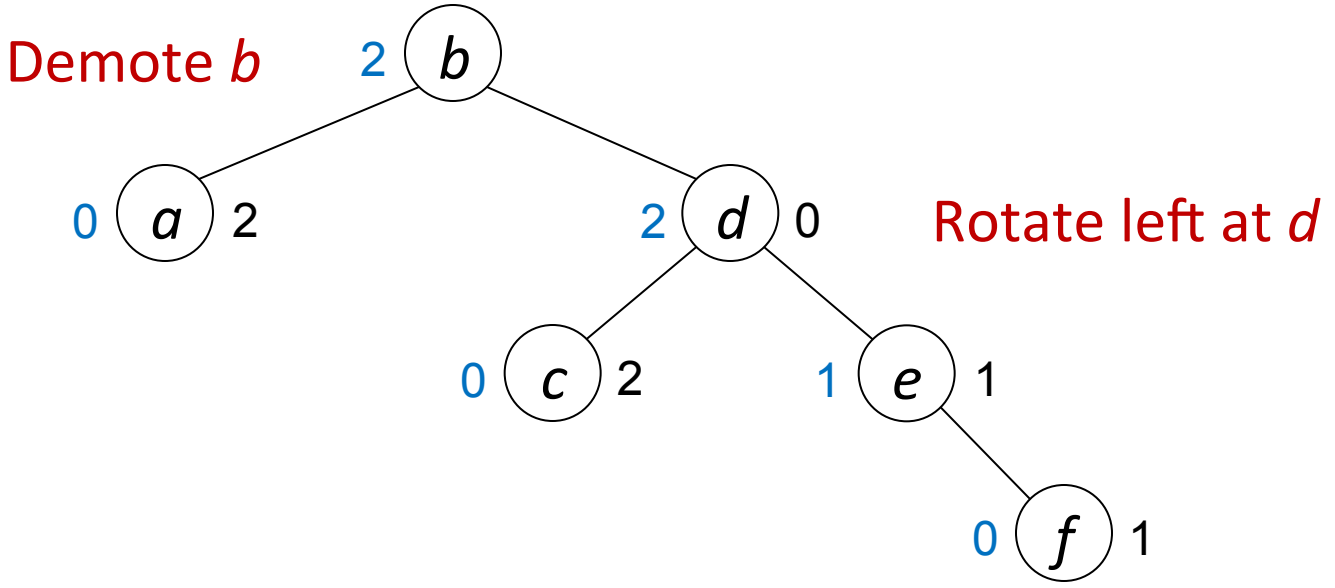
Insert *f*

Insertion example



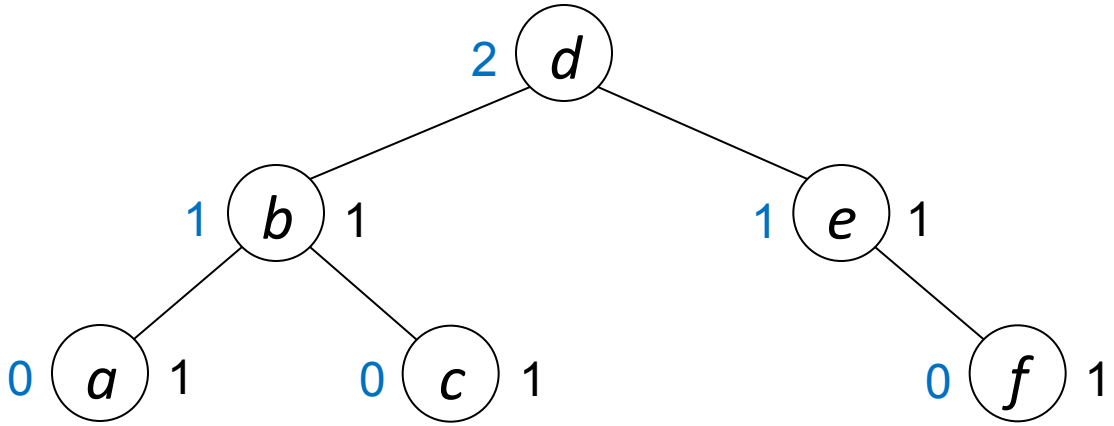
Insert *f*

Insertion example



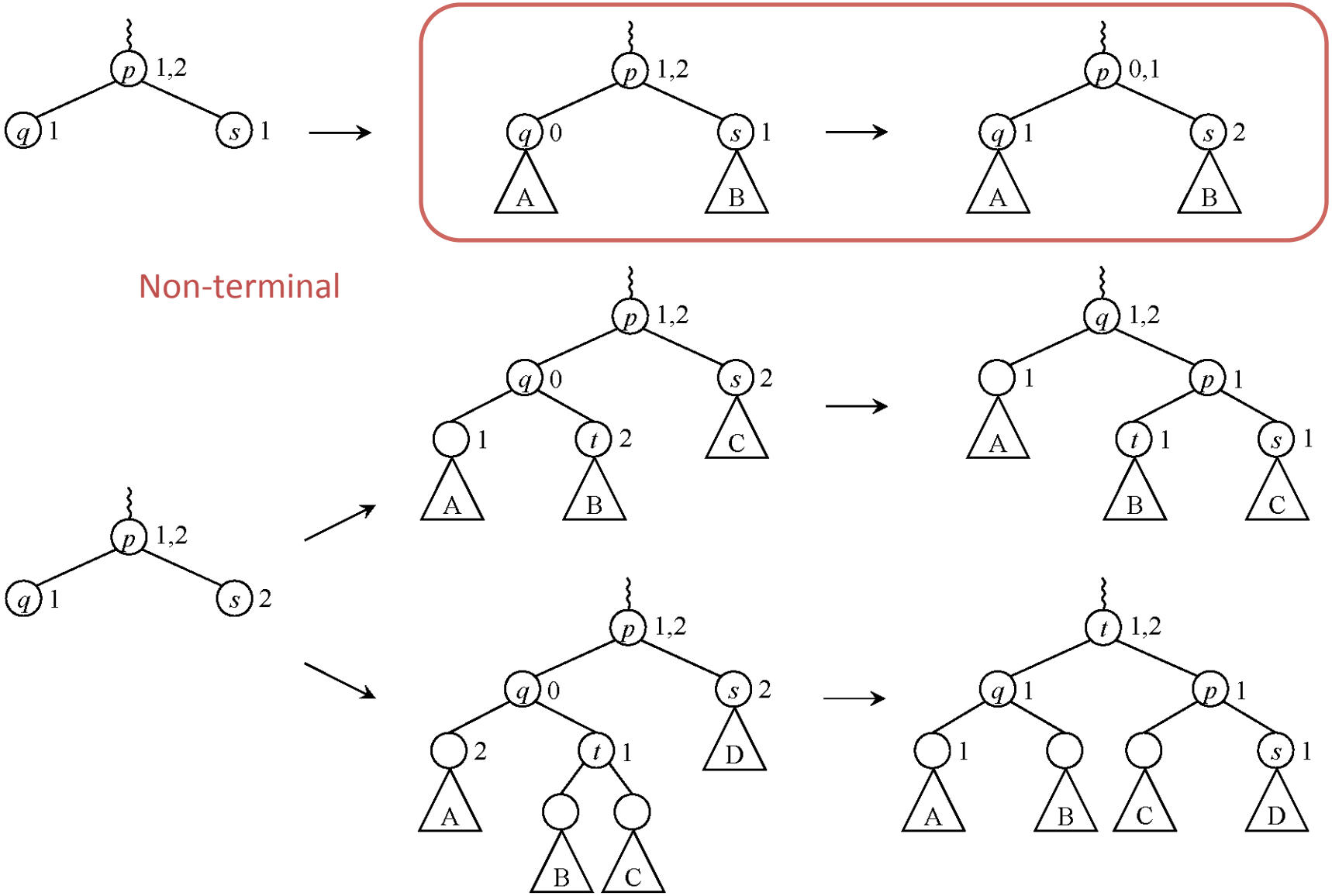
Insert *f*

Insertion example

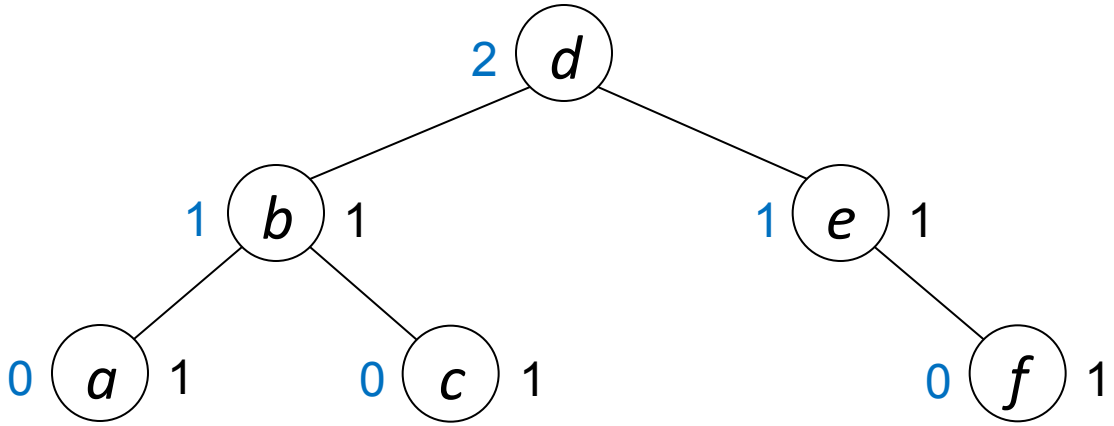


Insert *f*

Rebalancing: insertion

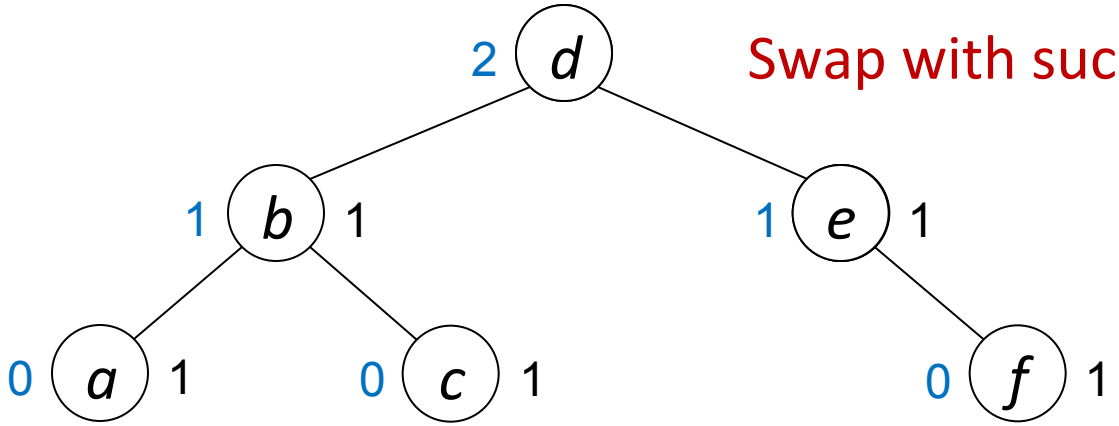


Deletion example



Delete *d*

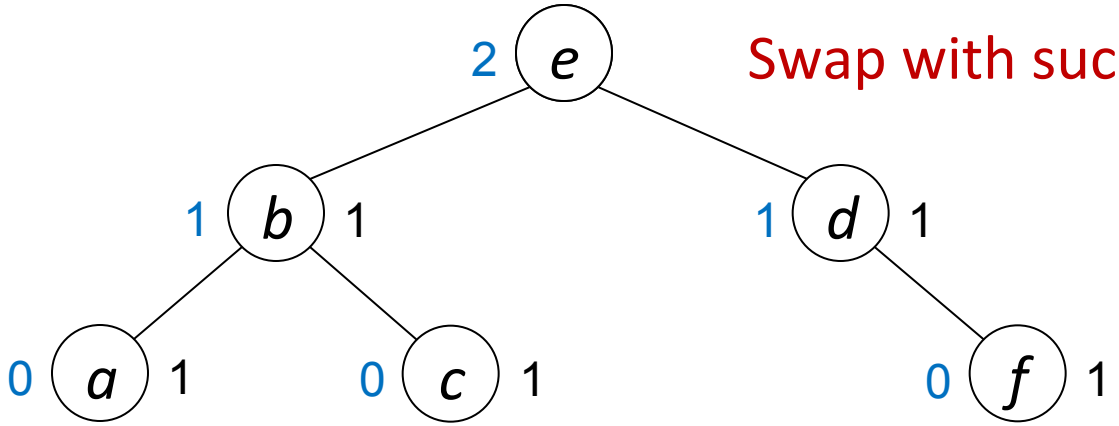
Deletion example



Swap with successor

Delete *d*

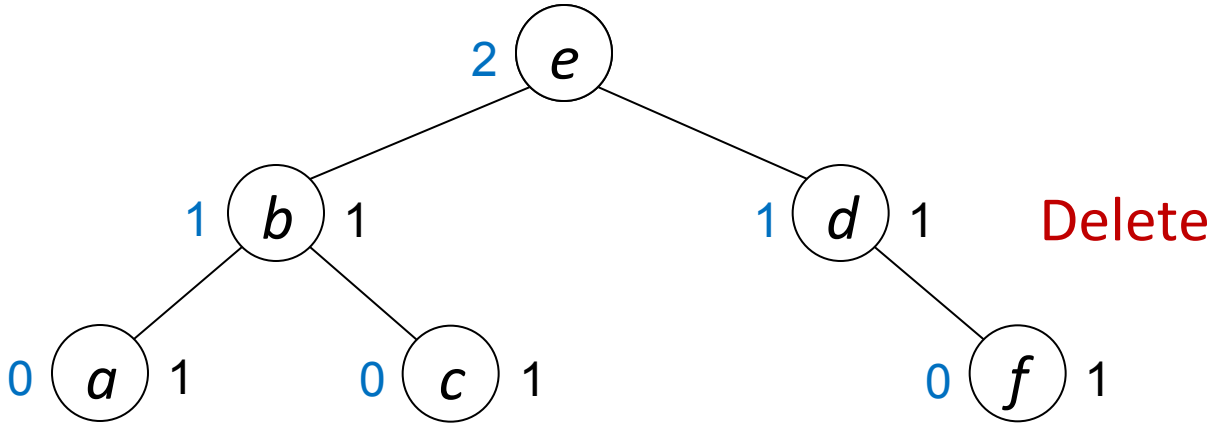
Deletion example



Swap with successor

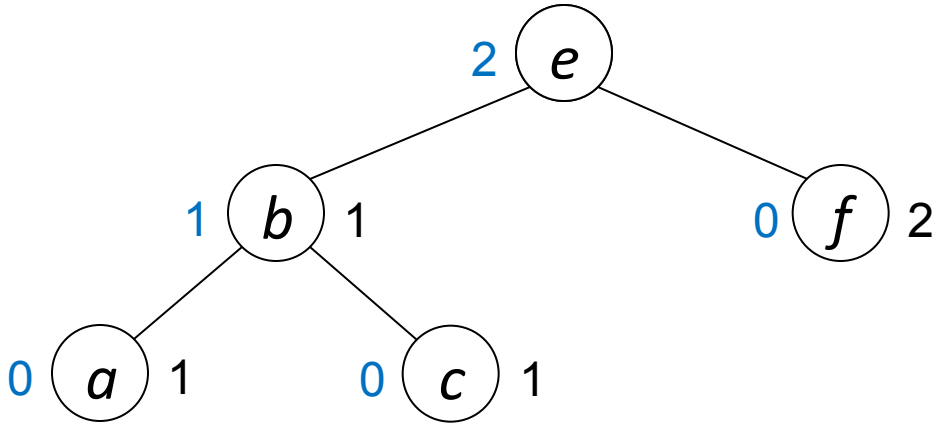
Delete *d*

Deletion example



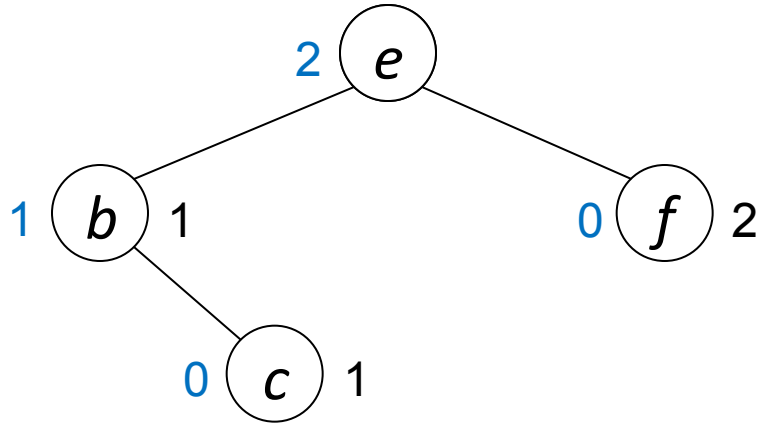
Delete *d*

Deletion example



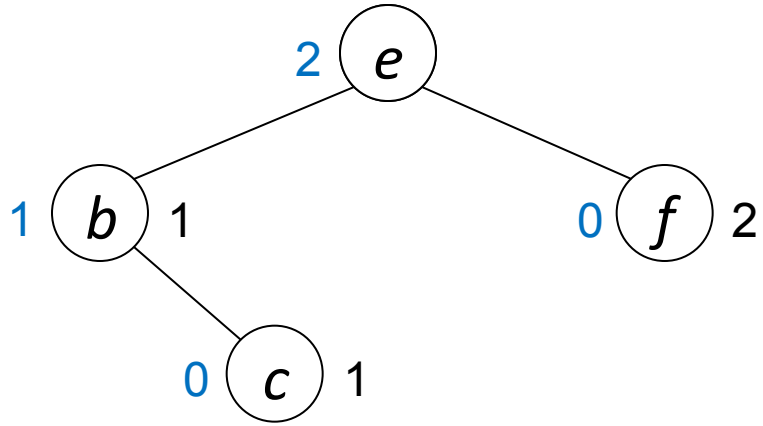
Delete *a*

Deletion example



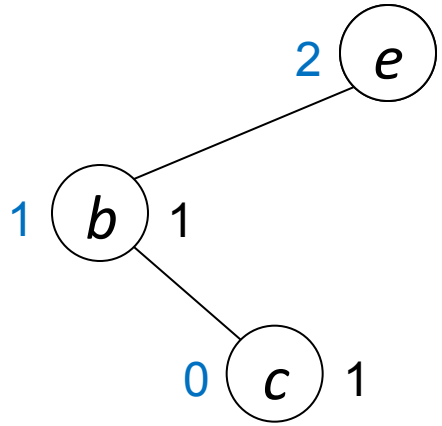
Delete *a*

Deletion example



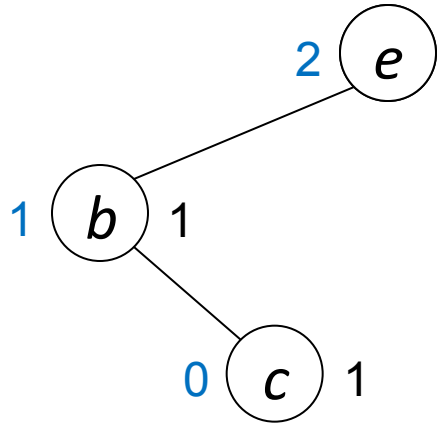
Delete *f*

Deletion example



Delete *f*

Deletion example



Double demote *e*

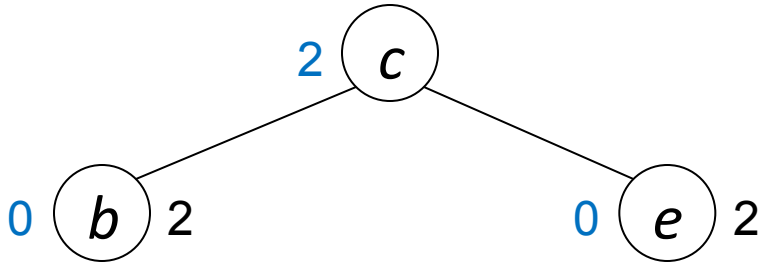
Demote *b*

Delete *f*

Double rotate at *c*

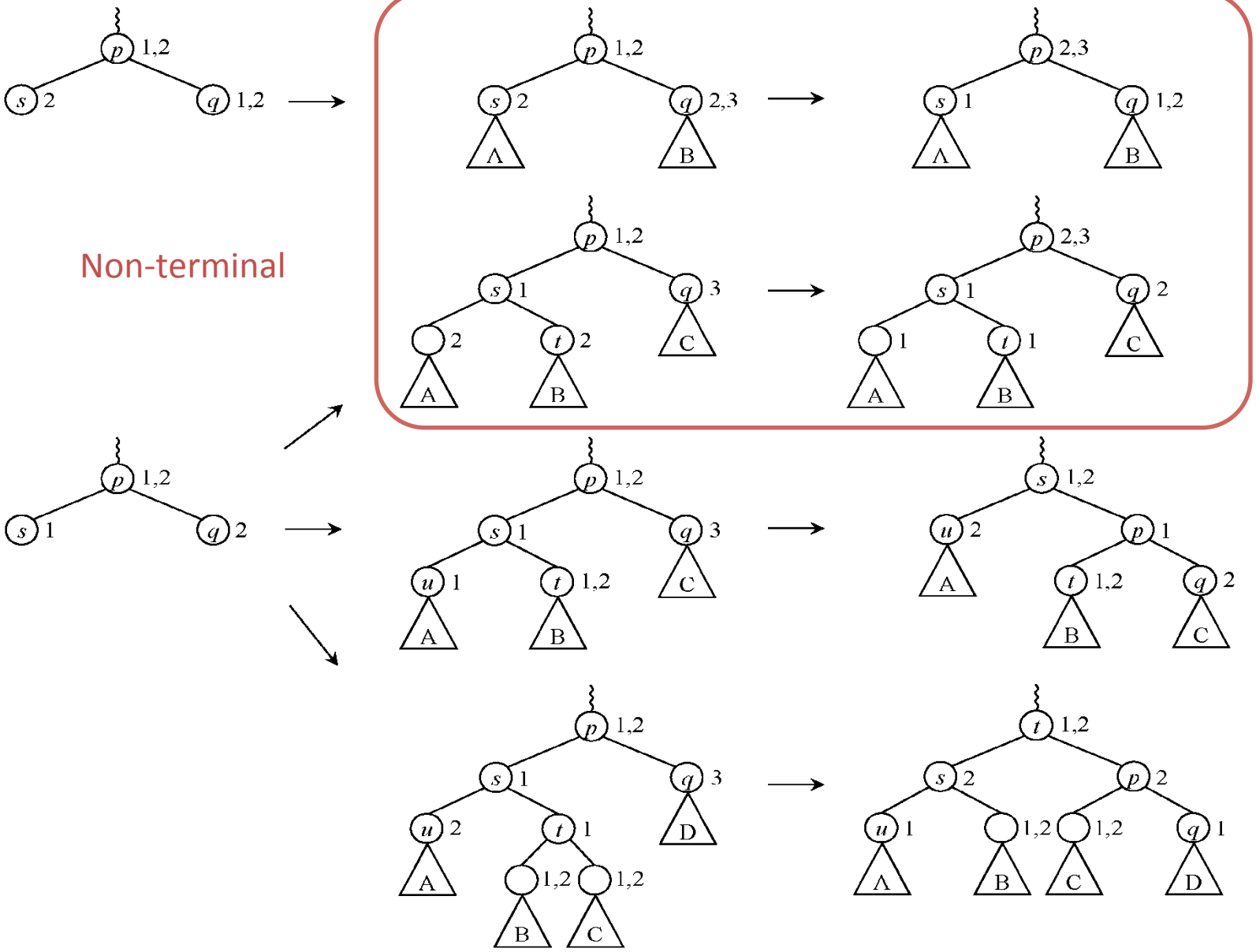
Double promote *c*

Deletion example



Delete *f*

Rebalancing: deletion



Wavl Tree Rebalancing

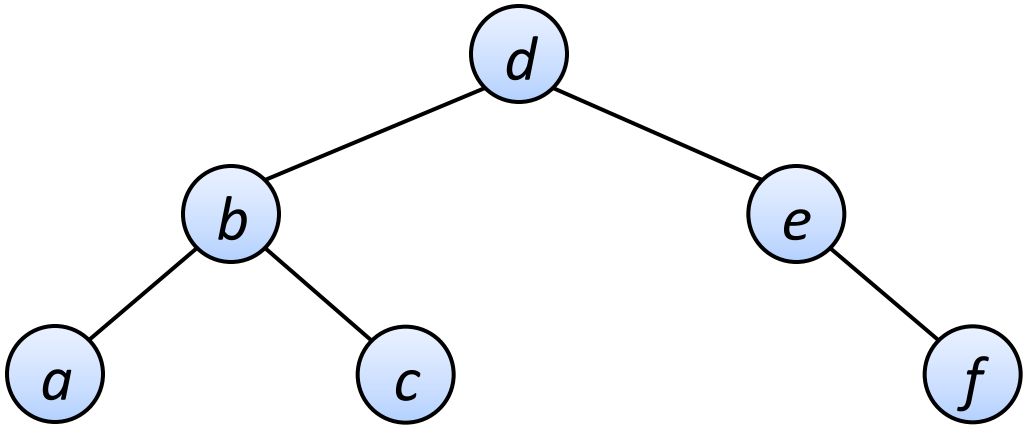
≤2 rotations per update (insert or delete)

m updates take O(*m*) rebalancing steps

Frequency of rank-*k* rebalancing steps is exponentially small in *k*: most rebalancing is near the bottom of the tree

If no deletions, wavl = AVL

Simpler Updates?



Rebalance after insertions, ~~deletions~~

Why avoid deletion rebalancing?

Eliminate more than half of the rebalancing code

Complicated (textbooks often omit)

High synchronization overhead

Some production B-tree implementations don't do it!

- WiredTiger, BerkeleyDB

Story time…

Deletion Without Rebalancing

Relaxed AVL (ravl) trees (Sen & Tarjan, 2009):

all rank differences are positive

Insertion rebalancing as in (weak) AVL trees

Deletion: no rebalancing, but preserve ranks

Properties of ravl trees

Tree can evolve to have arbitrary structure, but only slowly

Height is logarithmic in *m*, number of insertions

Insertion/deletion take O(1) amortized rebalancing steps

Rebalancing affects nodes exponentially infrequently in height

Analysis is not straightforward

Can guarantee O(lg*n*) height via rebuilding, either all at once or incrementally: e.g. run a background process that deletes successive items and inserts them into a new tree.

Balanced trees minimize worst-case access time to within a constant factor, but what if accesses are not uniform?

Access locality:

- Different but fixed access probabilities
- Spatial locality: frequent accesses near certain positions: fixed or moving fingers, e.g. first, last
- Time locality: working set

“Self-adjusting” Trees?

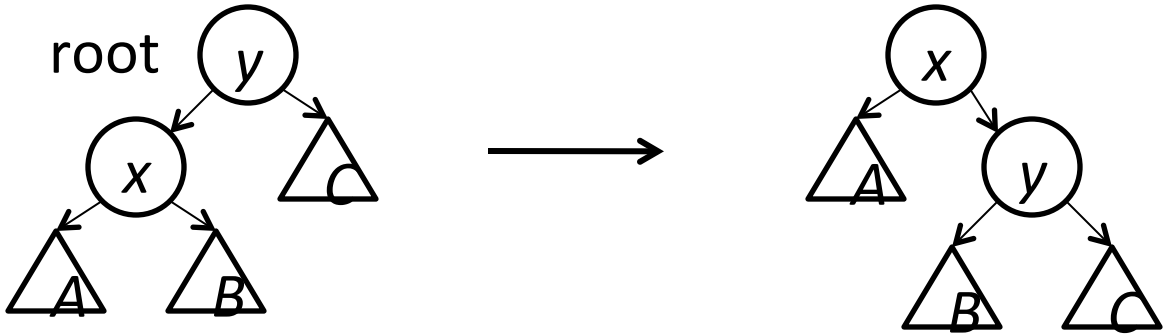
Is there a simple way to adjust a tree to match its usage?


J. STOLFI 1·87

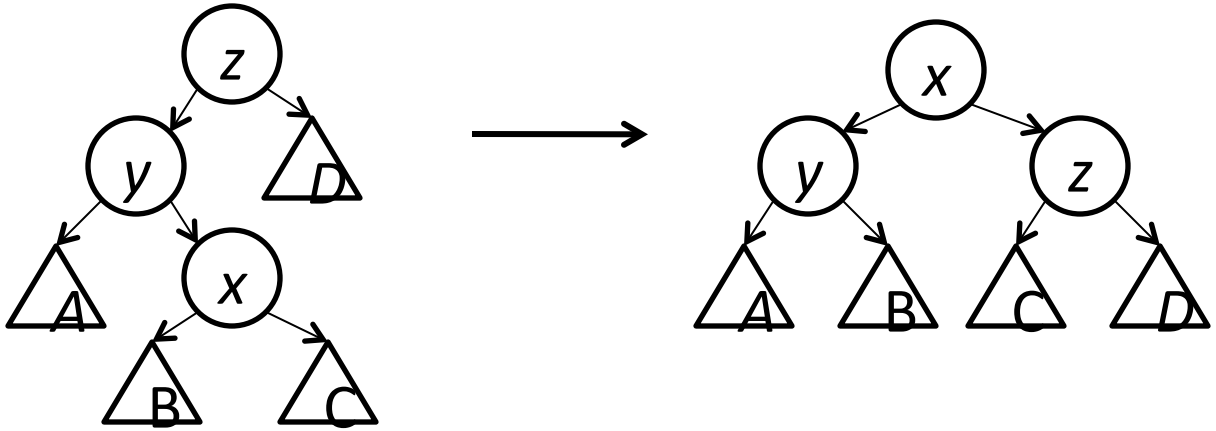
Splay Trees (Sleator & Tarjan, 1983)

Splay: to spread out. "*splay* at *x*" moves *x* to root via rotations, two at a time. Rotation order is generally bottom-up, but if the current node and its parent are both left or both right children, the top rotation is done first.

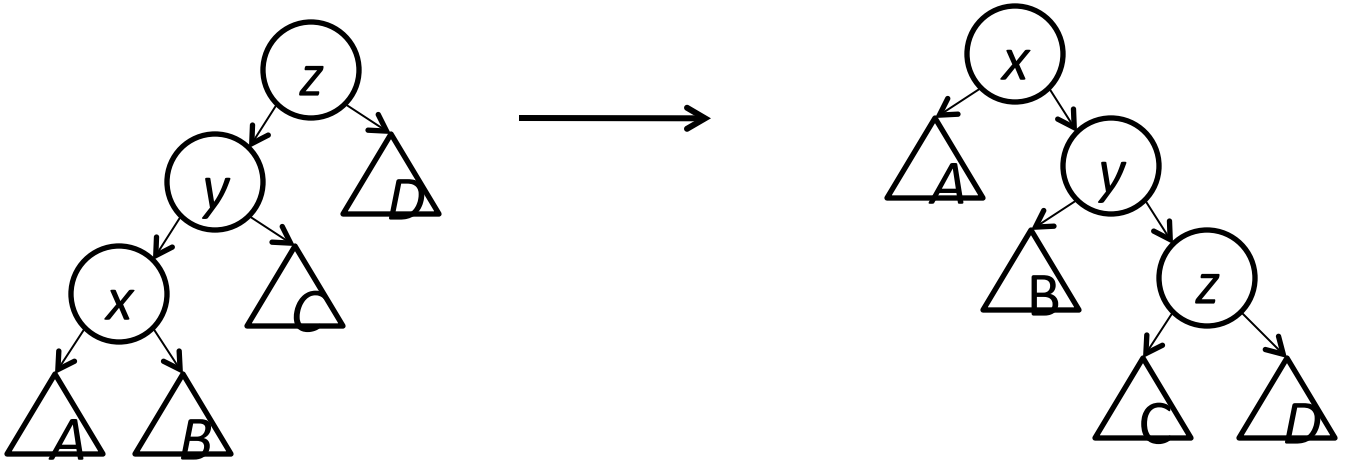
zig



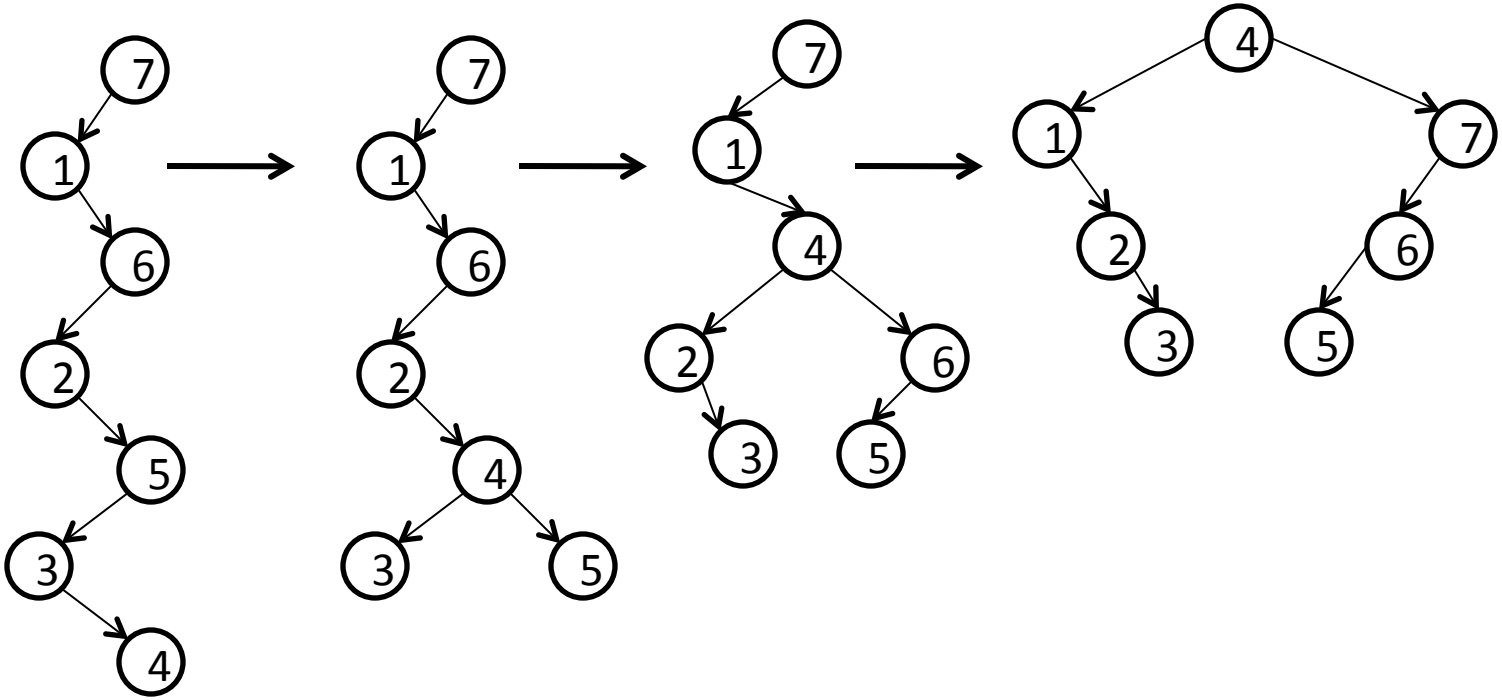
zig-zag



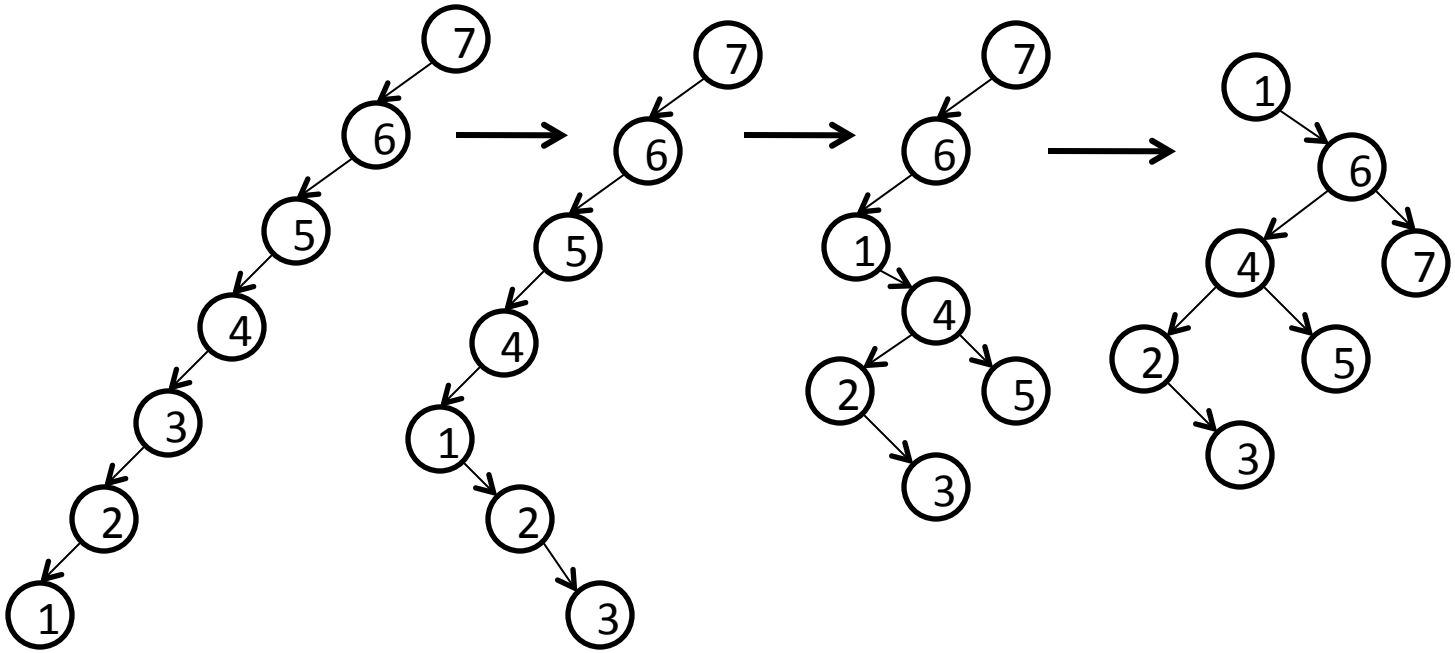
zig-zig



Splay: pure zig-zag



Splay: pure zig-zig



Operations on Splay Trees

Access *x*: follow search path to *x*, then splay at *x*.

Insert *x*: follow search path to missing node, replace by *x*, splay at *x*.

Delete *x*: follow search path to *x*, swap with successor if binary, delete *x*, splay at parent of *x* (before deletion).

Efficiency of Splay Trees

Individual operations can take many steps, even ~n

But long sequences of operations are fast:

m operations take O($m$log$n$) time

Fixed access probabilities: splaying matches the best static tree (to within a constant factor)

Splaying exploits space or time locality as well as complicated customized data structures (twacf)

Just how good is splaying?

Dynamic optimality conjecture: Given an initial tree and any access sequence, splaying does as well (twacf) as the best BST algorithm for the given sequence, even one that knows the entire sequence in advance.

(Each access must be done by moving the accessed item to the root via rotations. Each rotation costs 1)

Conclusions

(1) Both theory and practice are critical: each gives direction and insights to the other

(2) Classic problems still have rich secrets to yield, with practical benefits

References

B. Haeupler, S. Sen, & Robert E. Tarjan. **Rank-Balanced Trees.** ACM Trans. Algorithms 11(4): 30 (2015).

D. Kim, S. Sen, & R. E. Tarjan. **Deletion without Rebalancing in Binary Search trees.** ACM Trans. Algorithms 12(4): 57 (2016).

D. D. Sleator & R. E. Tarjan. **Self-Adjusting Binary Search Trees.** J. ACM 32(3): 652-686 (1985).

Thanks!