Efficient Binary Search Trees

COS 226 October 12, 2023

Robert E. Tarjan

Observations

Over the last 60 years, computer scientists have developed many beautiful and theoretically efficient algorithms and data structures.

But computer science is still a young field.

But we have often settled for the first (good enough) solution.

It may not be the best – the design space is rich.

Goal: simplicity + efficiency = “elegance”

Identify the simplest possible efficient methods to solve basic problems

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

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

Dictionary Problem

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

Access: find a given item, return its information
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 in-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).

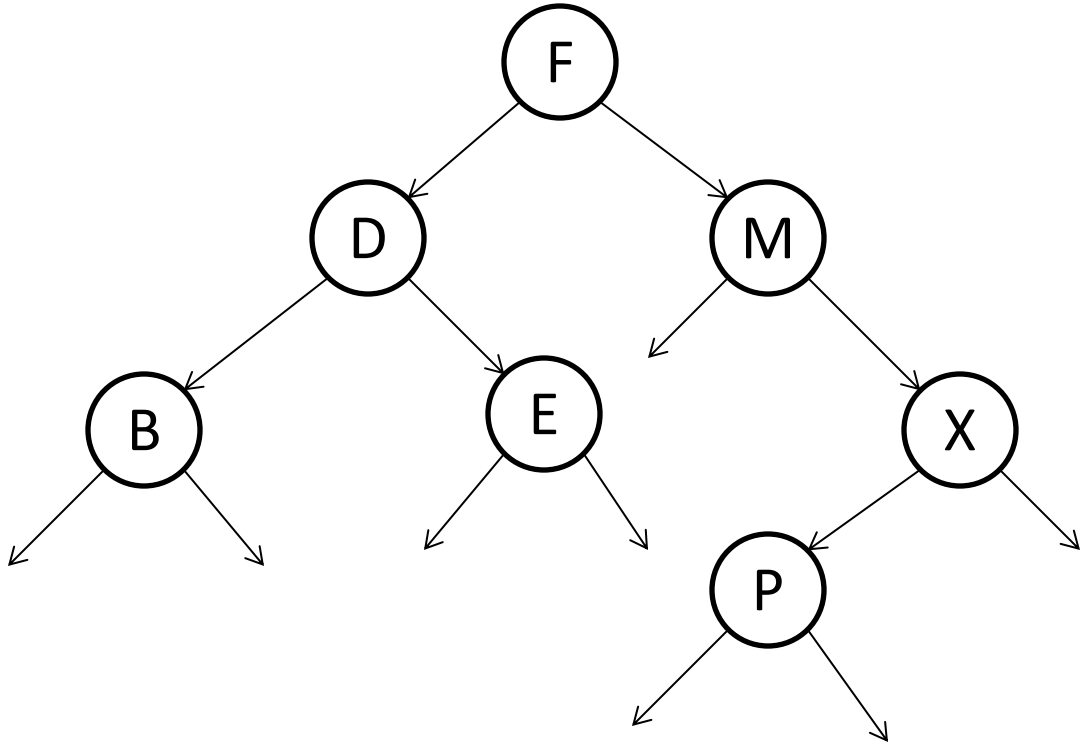
If S non-empty, compare x to some item y in S.

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

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

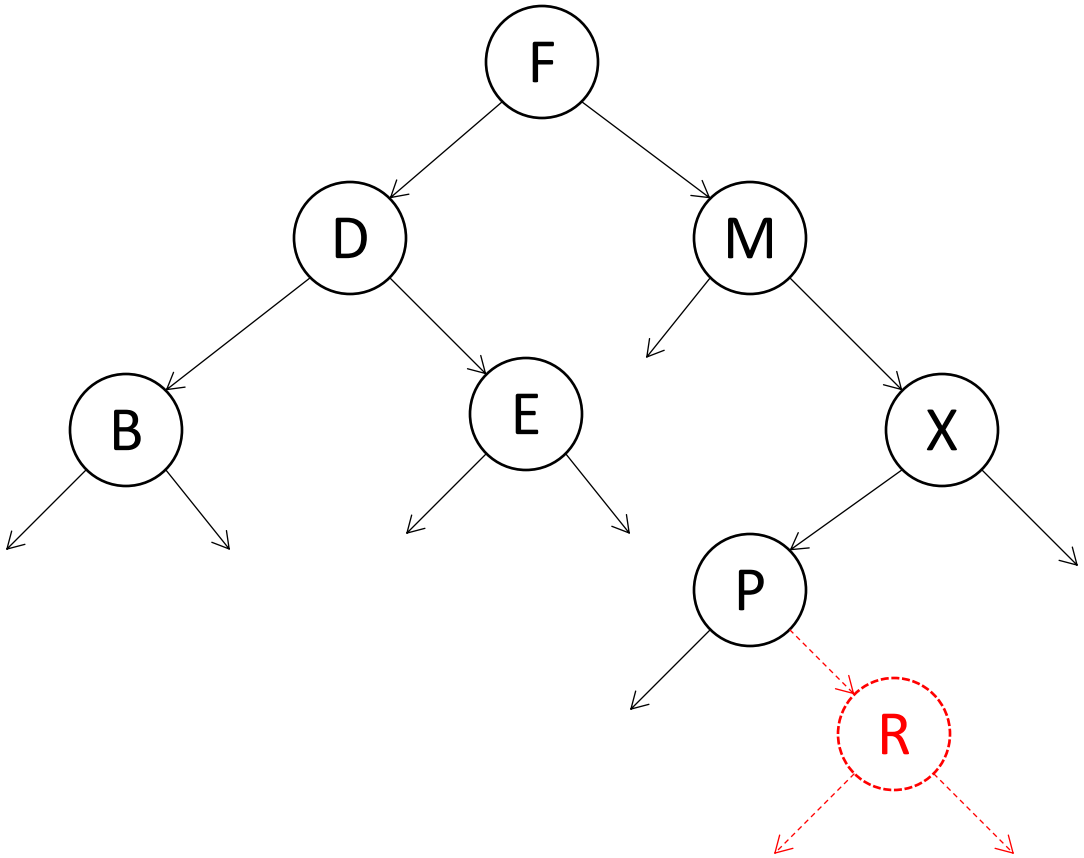
If $x > y$, search among elements in $S > y$

Data Structure:
Binary Search Tree



Insertion

Insert R



Deletion

Find item. Remove node. Repair tree.

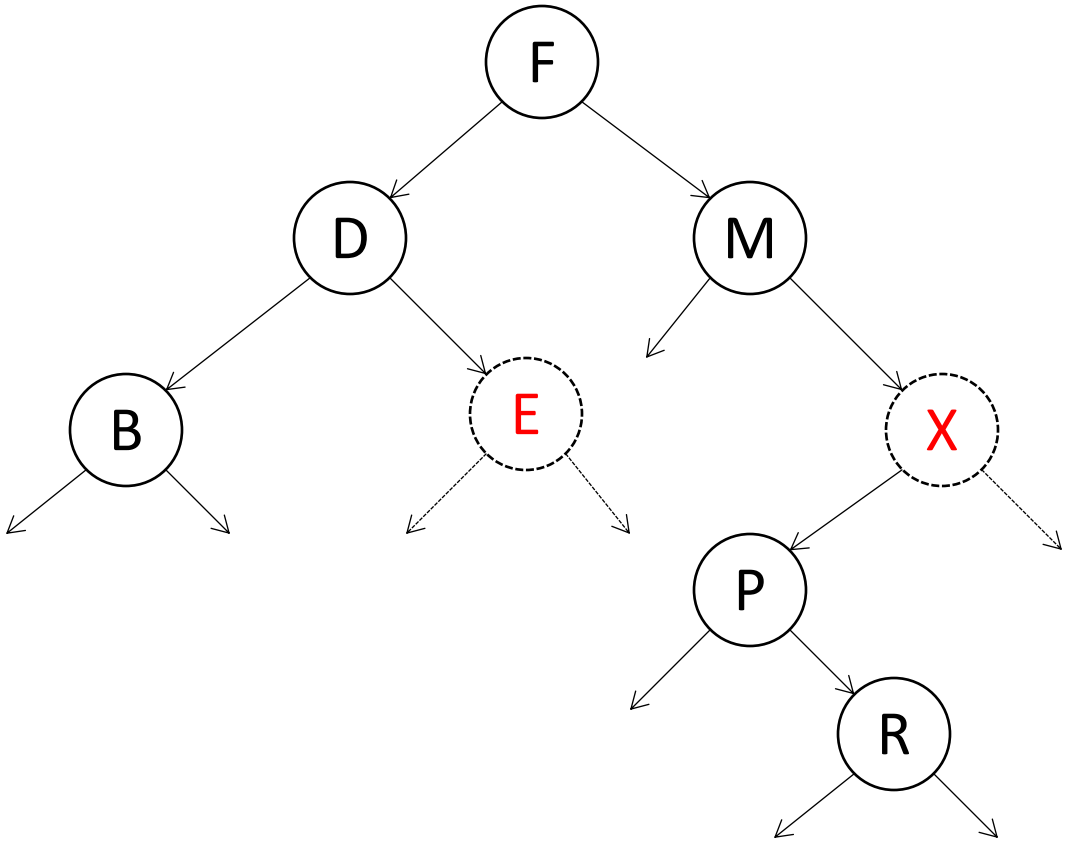
If leaf (no children), delete node.

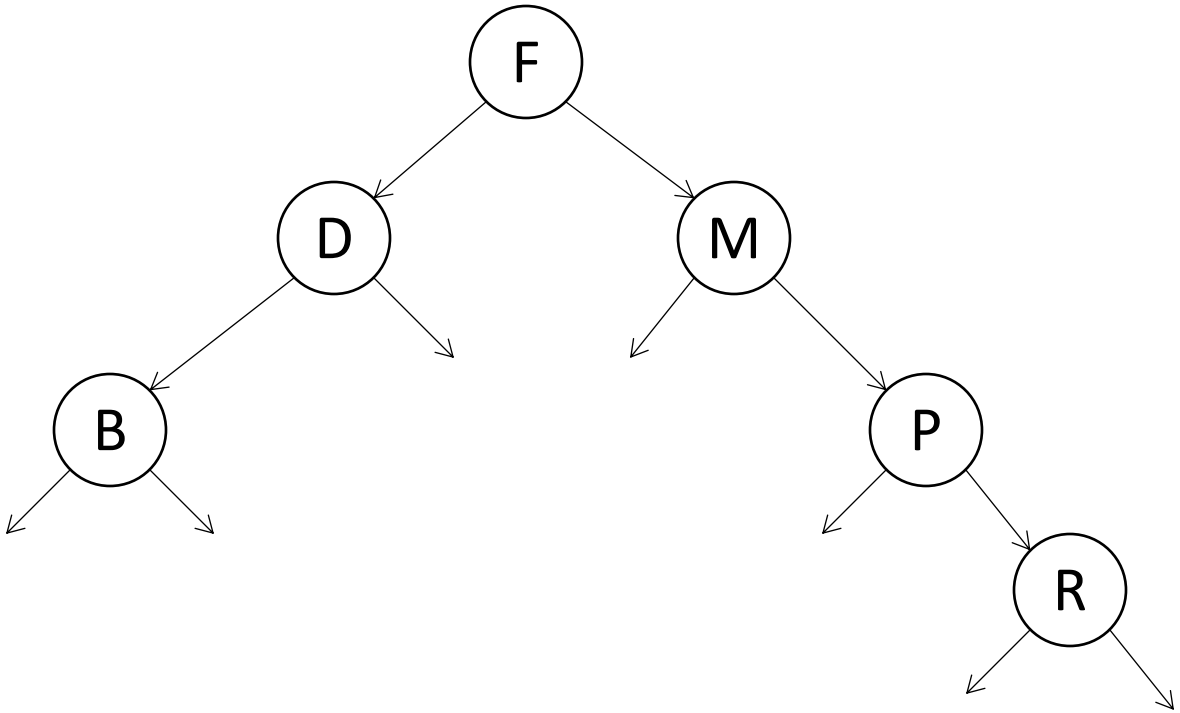
If unary (one child), replace by the other child.

If binary(two children)?

Delete E

Delete X



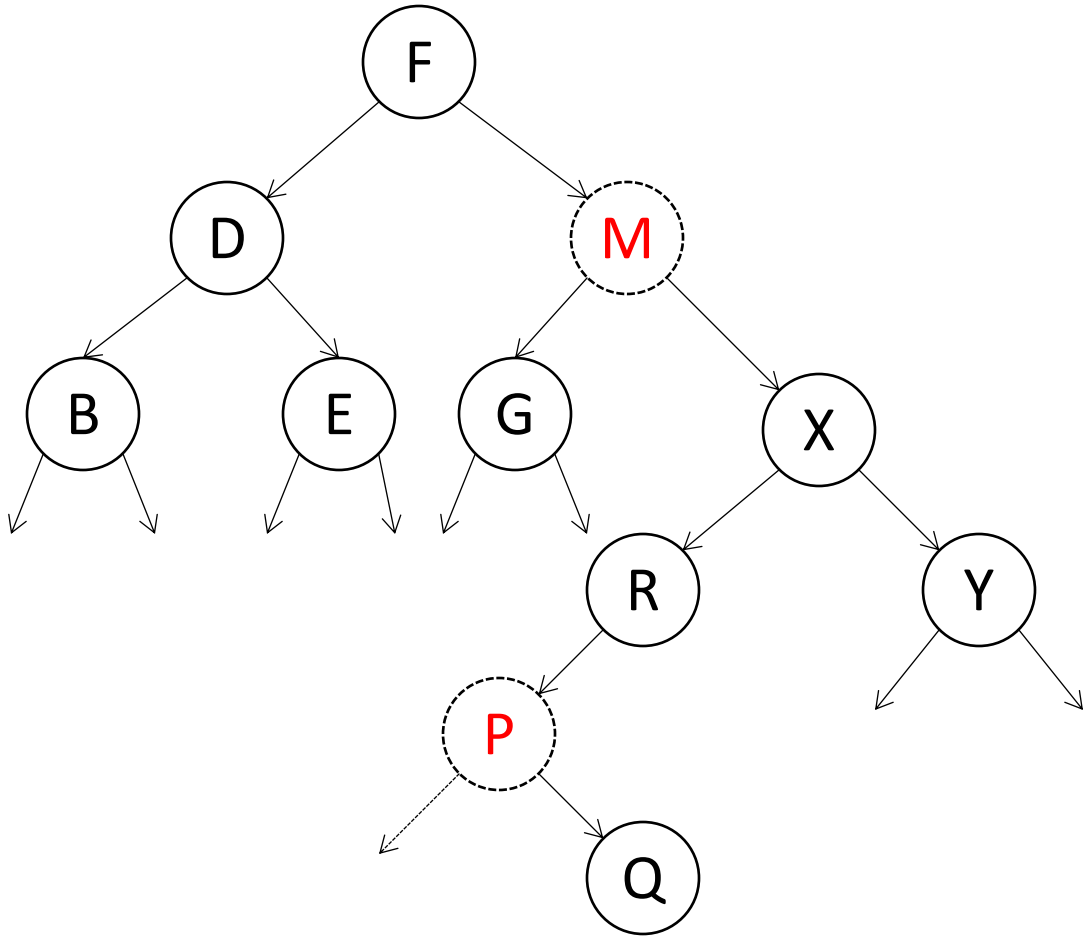

F
D
M
B
P
R

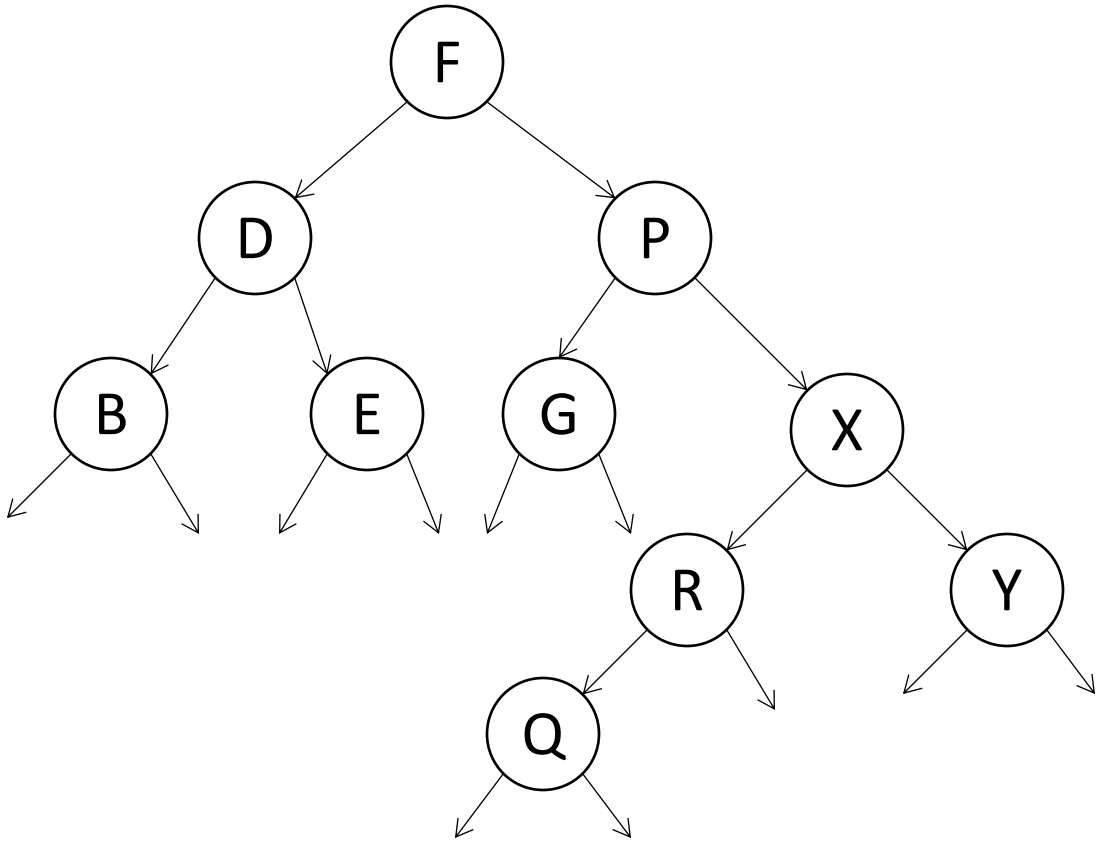
If binary, swap with next item. Now in leaf or unary node; delete. To find next item, follow left path from right child (Hibbard deletion).

Delete M:

Swap with P;

delete.




F
D
P
B
E
G
X
R
Y
Q

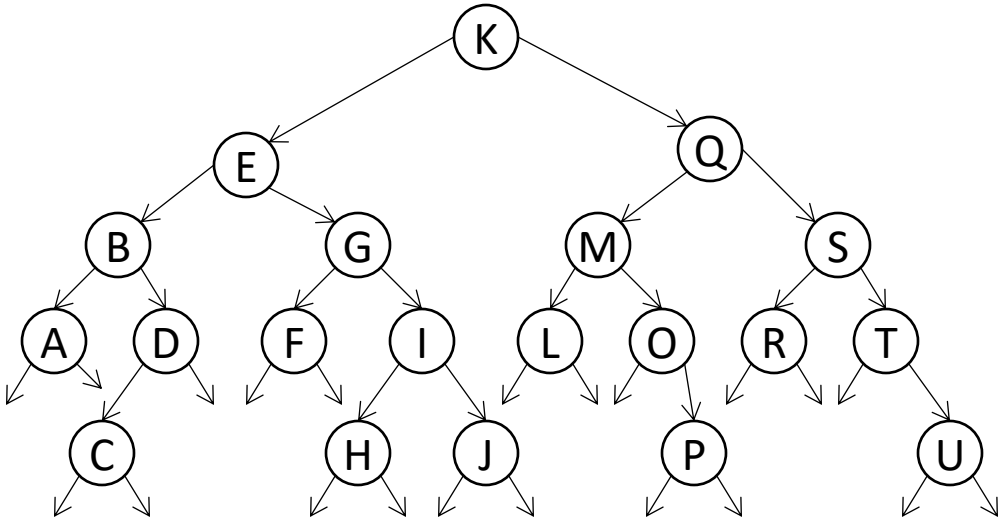
Time Per Operation

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

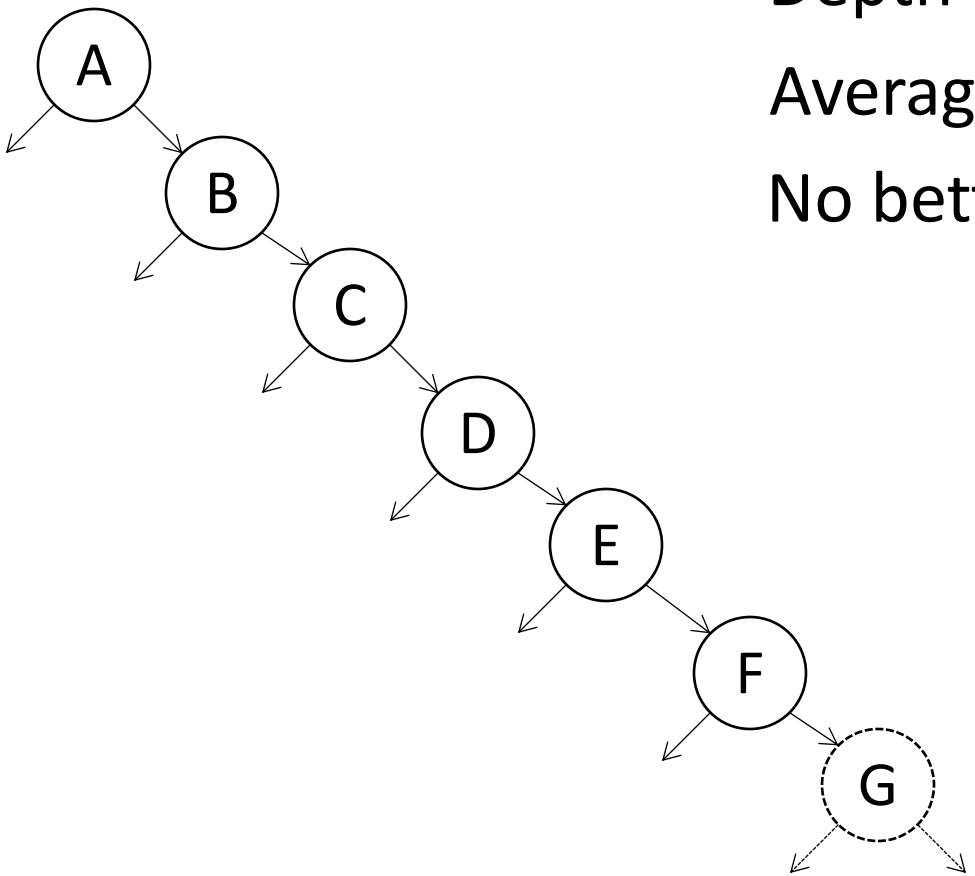
Starting with an empty tree, if n items are inserted in uniformly random order, expected tree depth (access time) is $O(\log n)$

The analysis is the same as that of quicksort with pivot chosen uniformly at random

Worst Case

Natural but bad insertion order: sorted.

Insert A, B, C, D, E, F, G,…



Depth of tree is *n*

Average access time is ~*n*/2

No better than a list!

Ways to improve efficiency

- Balance: AVL trees, (left-leaning) red-black trees, weak AVL trees…
- Randomization: *Zip* trees
- Self-adjustment: Splay trees

Zip Trees

The height of a node is the maximum number of links on a path from it to a leaf.

Idea: On insertion, choose a height for an item and insert it at the given height, or close to it. Definition:

Choose heights like those in a best-case BST: ½ of the nodes at height 0, ¼ at height 1, 1/8 at height 2…
Choose the heights randomly.

We cannot choose heights exactly.

Instead, for each node to be inserted we choose a rank, as follows: flip a fair coin and count the number of heads before the first tail.* The rank of a node does not change while it is in the tree.

The rank of a node has a geometric distribution: a node has rank k with probability $1/2^{k+1}$.

We want the height of a node to be within a constant factor of its rank.

*Can do in O(1) machine instructions.

*Zip** Tree

A binary search tree in which each node has a rank chosen randomly on insertion, with nodes in in-order by key and max-heap-ordered by rank, breaking rank ties in favor of smaller key:

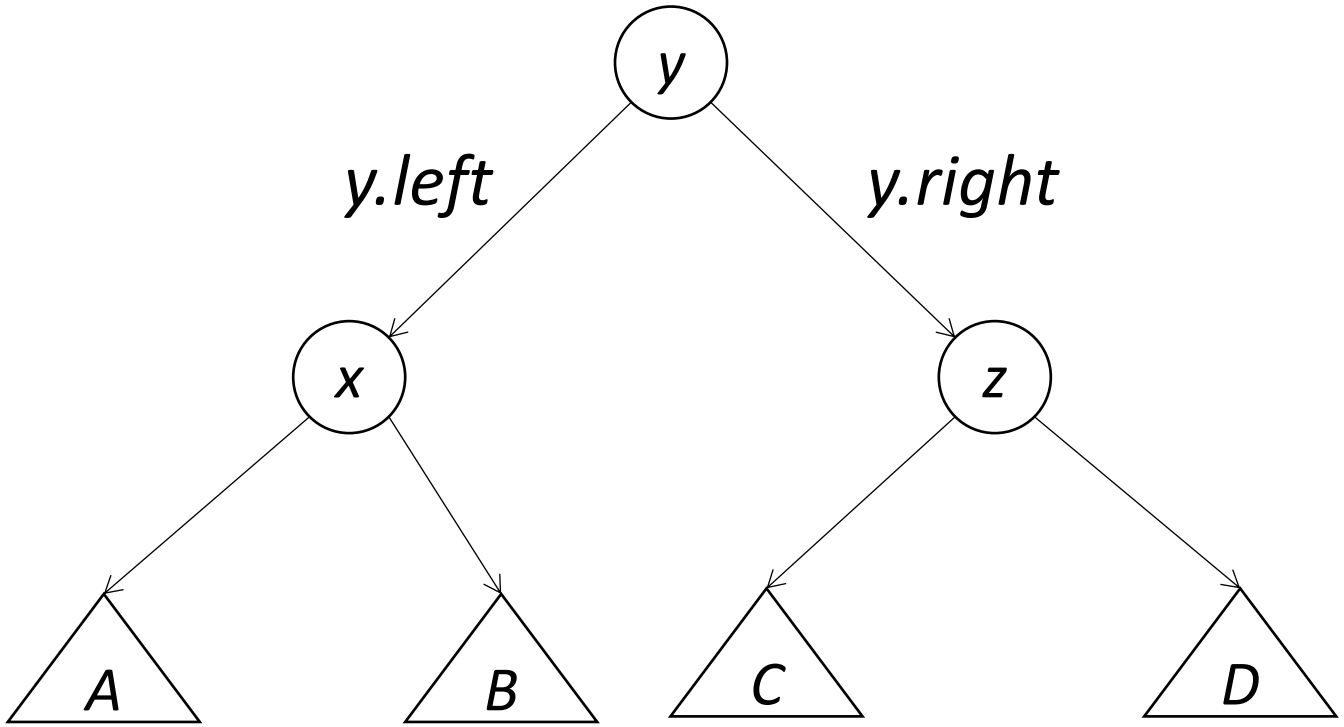
$$x.left.key < x.key < x.right.key$$
$$x.left.rank < x.rank$$
$$x.right.rank \leq x.rank$$

**Zip*: “to move very fast”

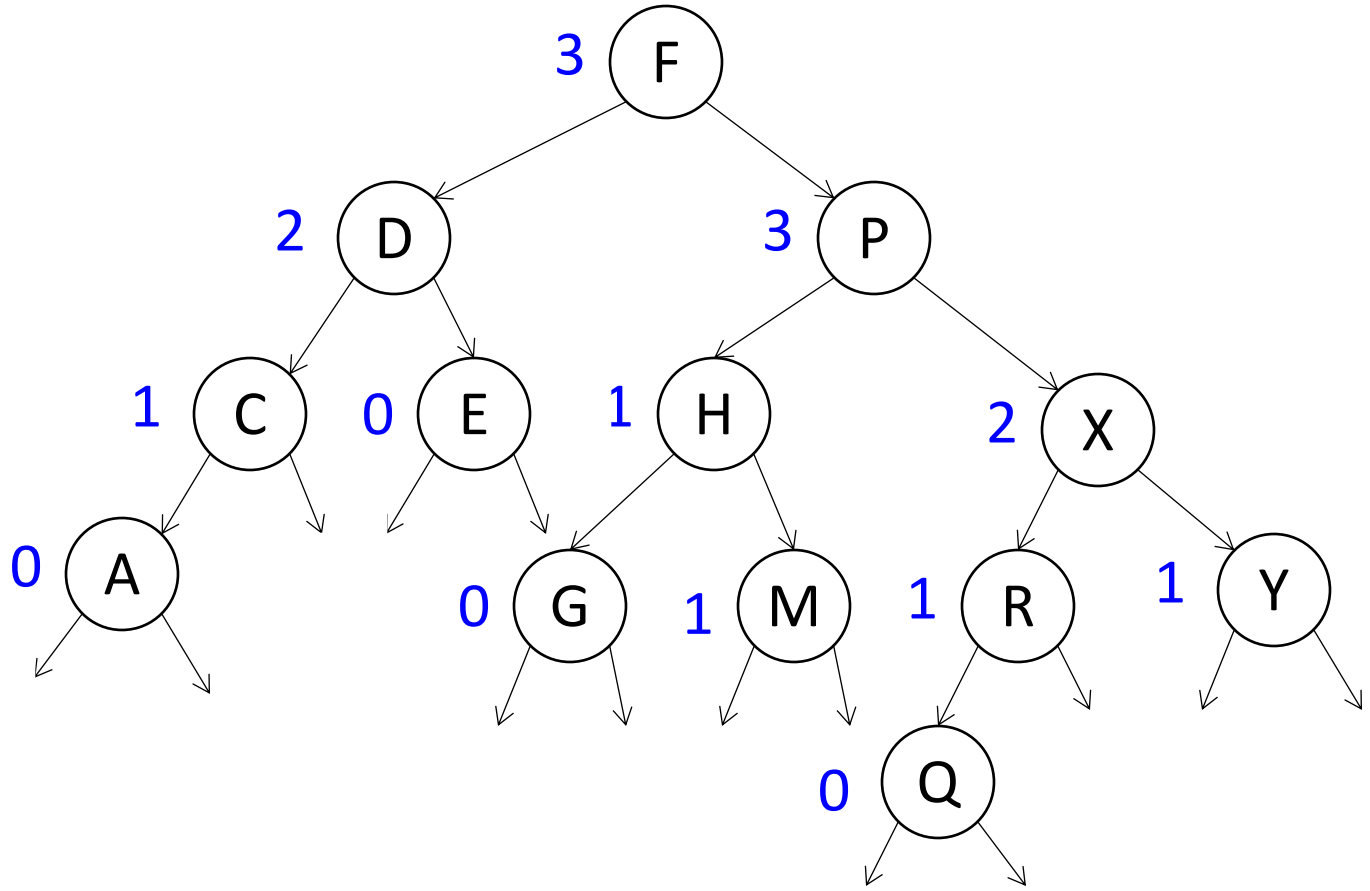
In-order by key: key of *y* greater than keys in subtree of *x*, smaller than keys in subtree of *z*

Max-heap order by rank: rank of *y* no smaller than that of *x*,

greater than that of *z*



A *Zip* Tree

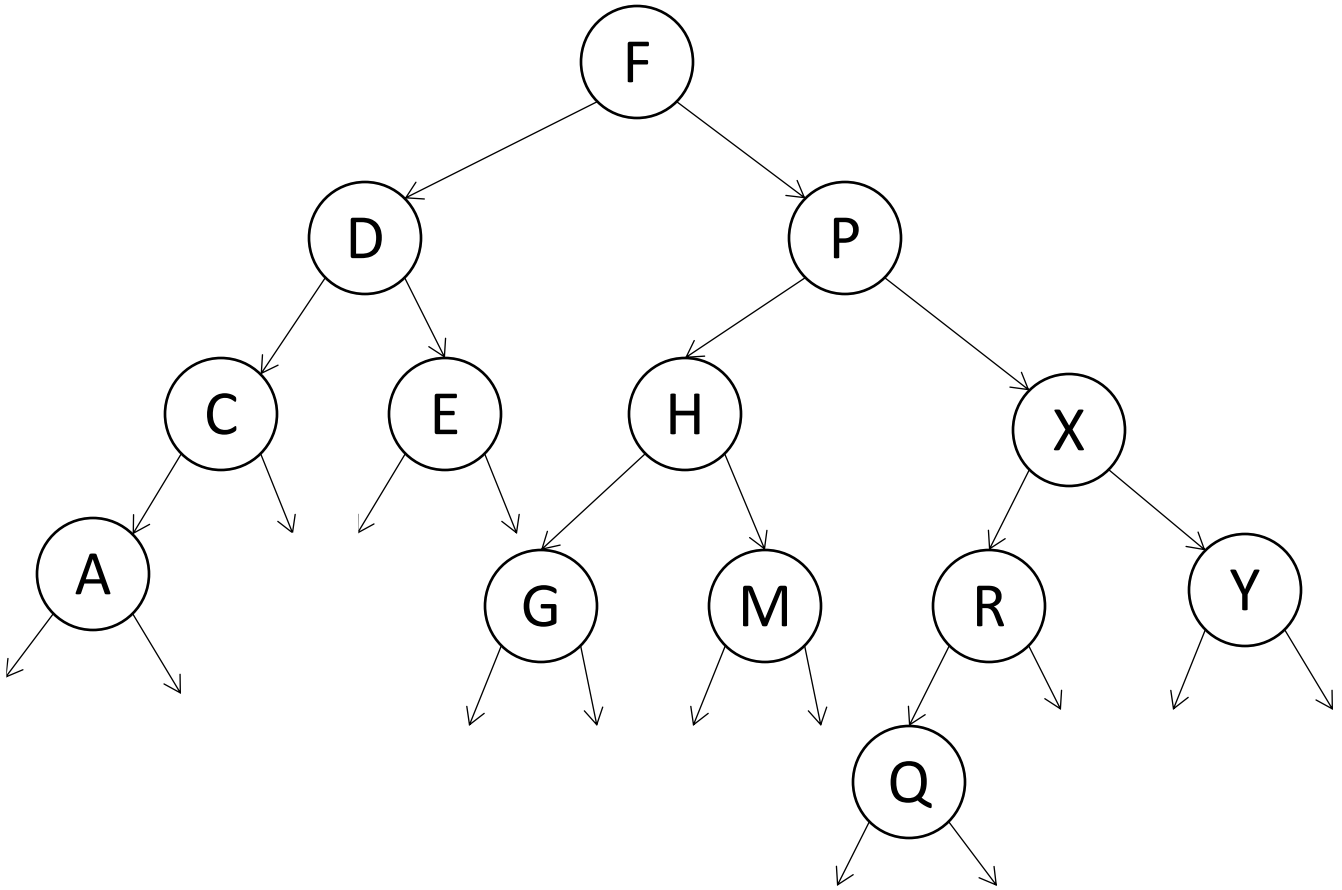


Zip tree insertion?

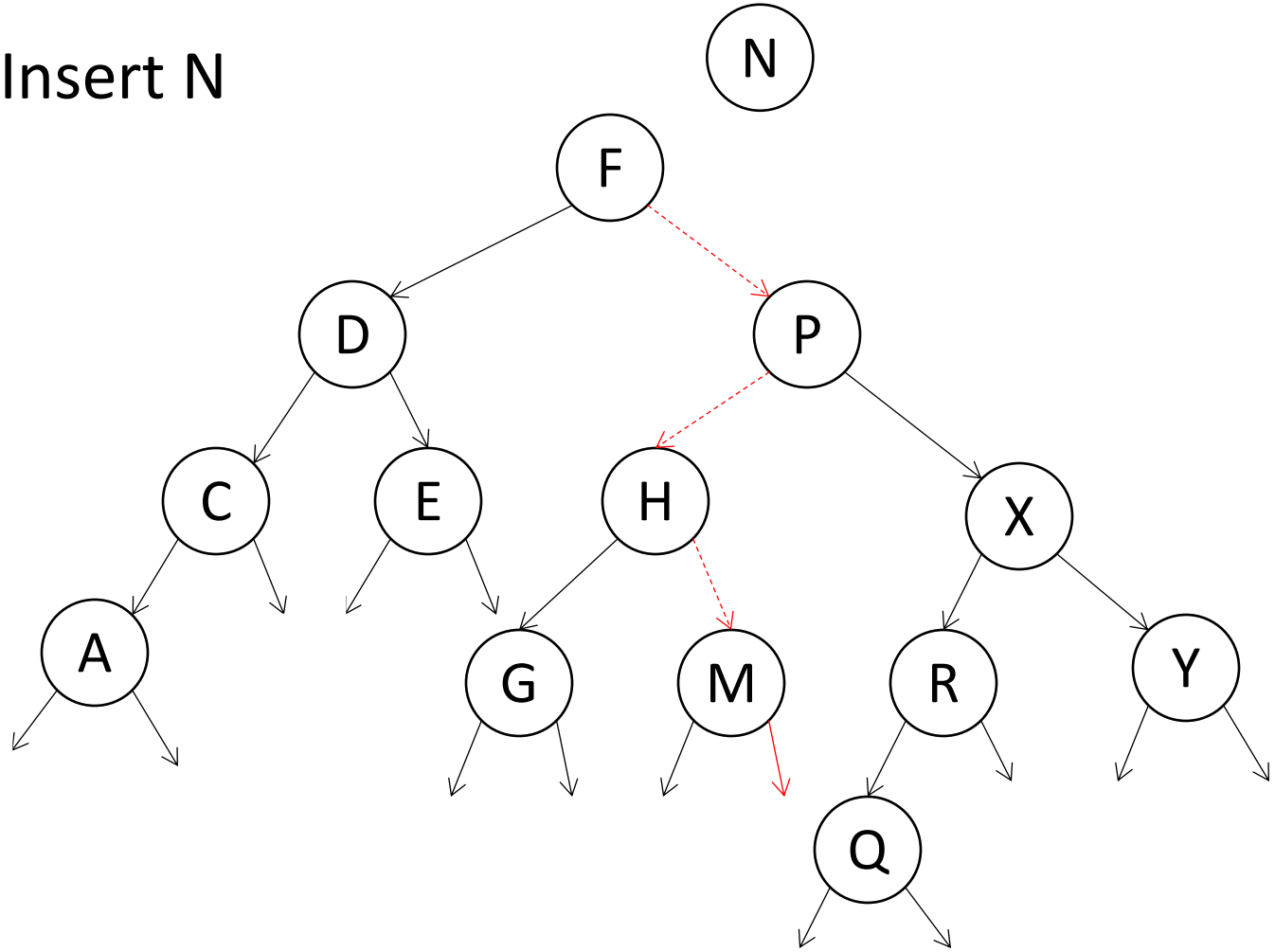
Root insertion (Stephenson 1980)

Let *x* be the item to be inserted. Follow the search path for *x*, unzipping it by splitting it into a path *P* of nodes with keys less than that of *x* and a path *Q* of nodes with keys greater than that of *x*. Make the top node of *P* the left child of *x* and the top node of *Q* the right child of *x*.

Insert N

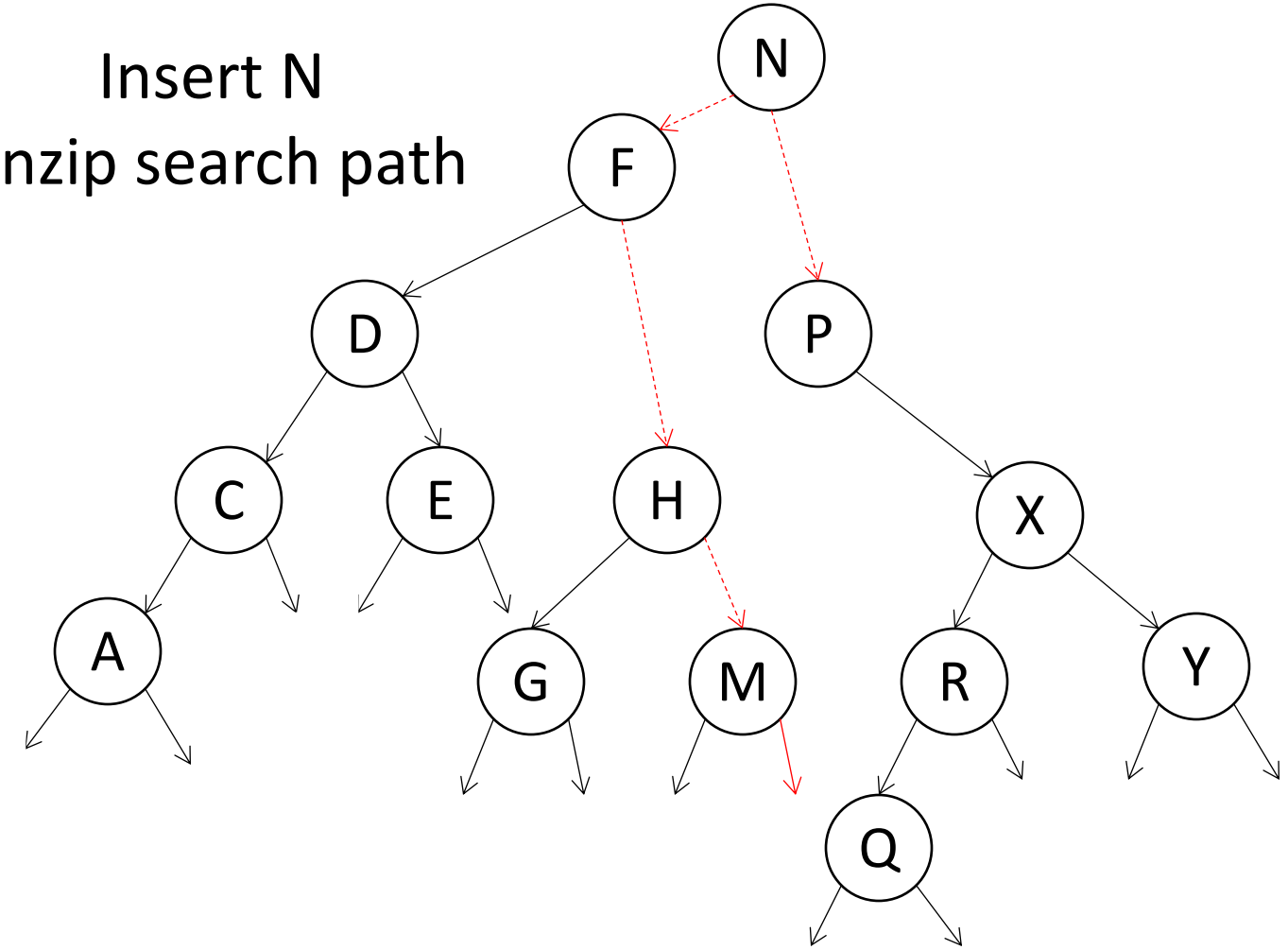


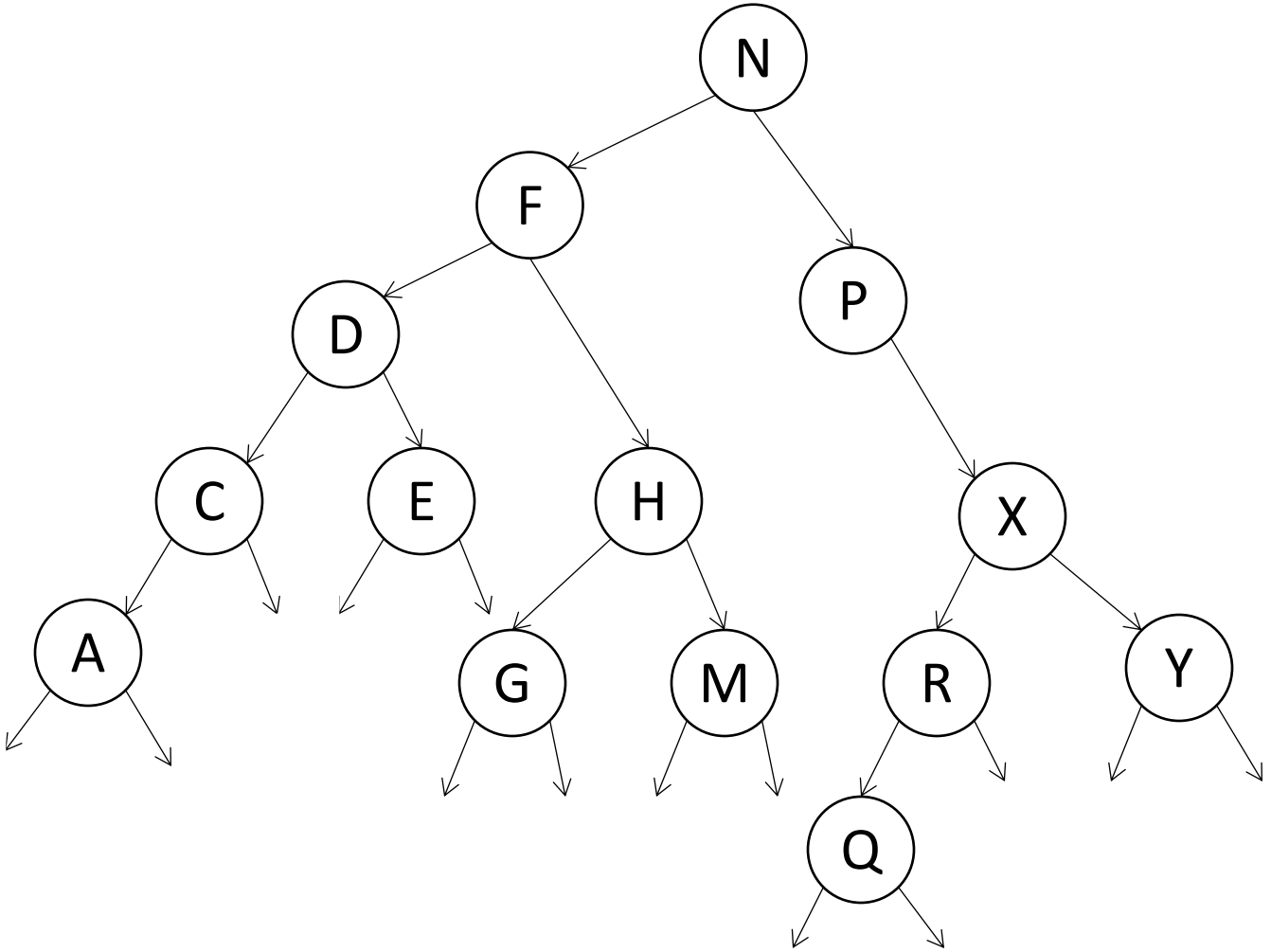
Insert N



Insert N

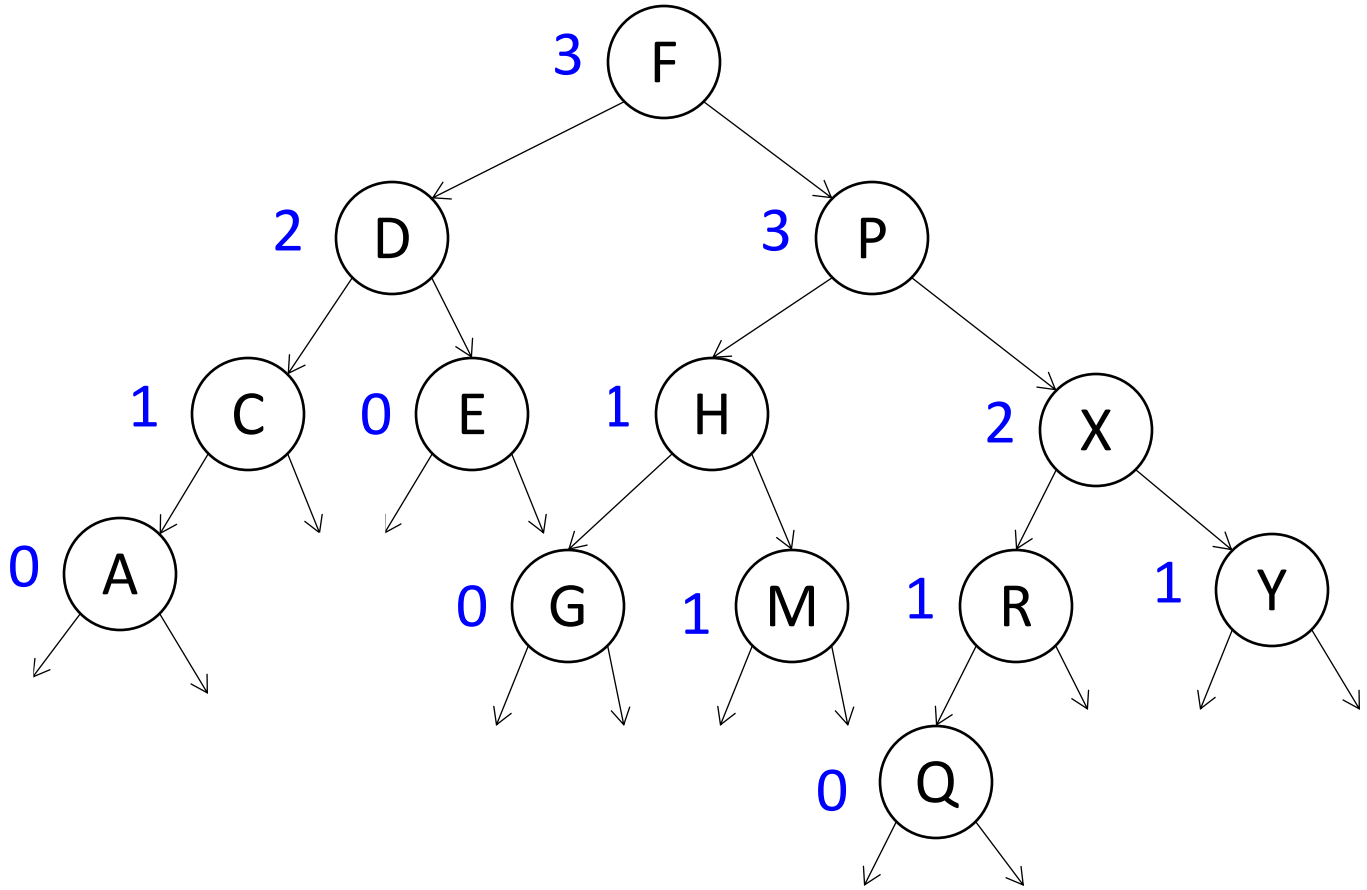
Unzip search path



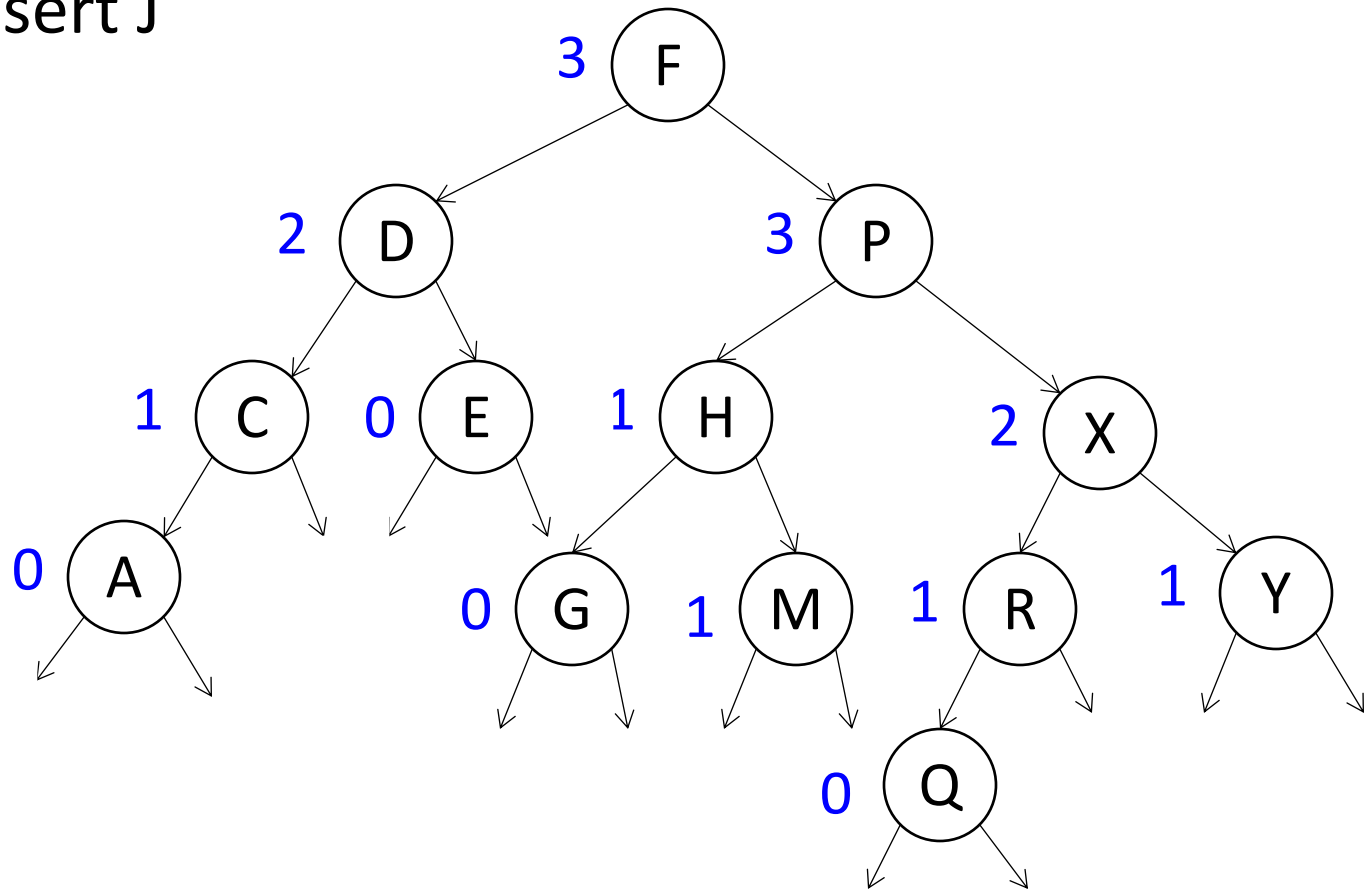

N
F
P
D
C
E
H
X
A
G
M
R
Y
Q

Zip tree insertion: hybrid of leaf & root insertion

Let *x* be the item to be inserted. Choose its rank. Follow the search path for *x* until reaching the node y that *x* should replace. Follow the remaining search path for *x*, *unzipping* it by splitting it into a path *P* of nodes with keys less than that of *x* and a path *Q* of nodes with keys greater than that of *x*. Make the top node of *P* the left child of *x* and the top node of *Q* the right child of *x*. Replace y as a child of its parent by *x*.

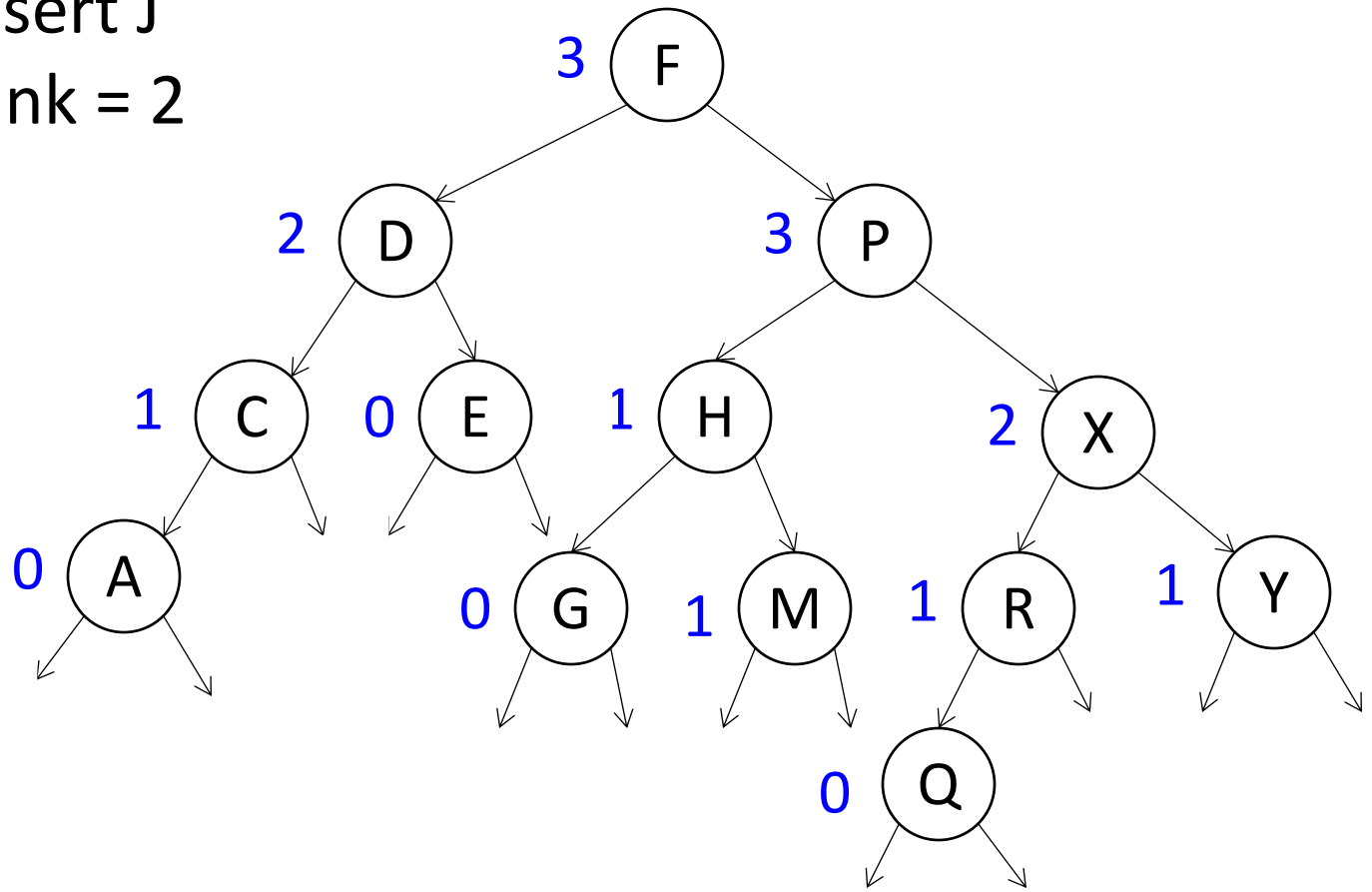

3
F
2
D
3
P
1
C
0
E
1
H
2
X
0
A
0
G
1
M
1
R
1
Y
0
Q

Insert J



Insert J

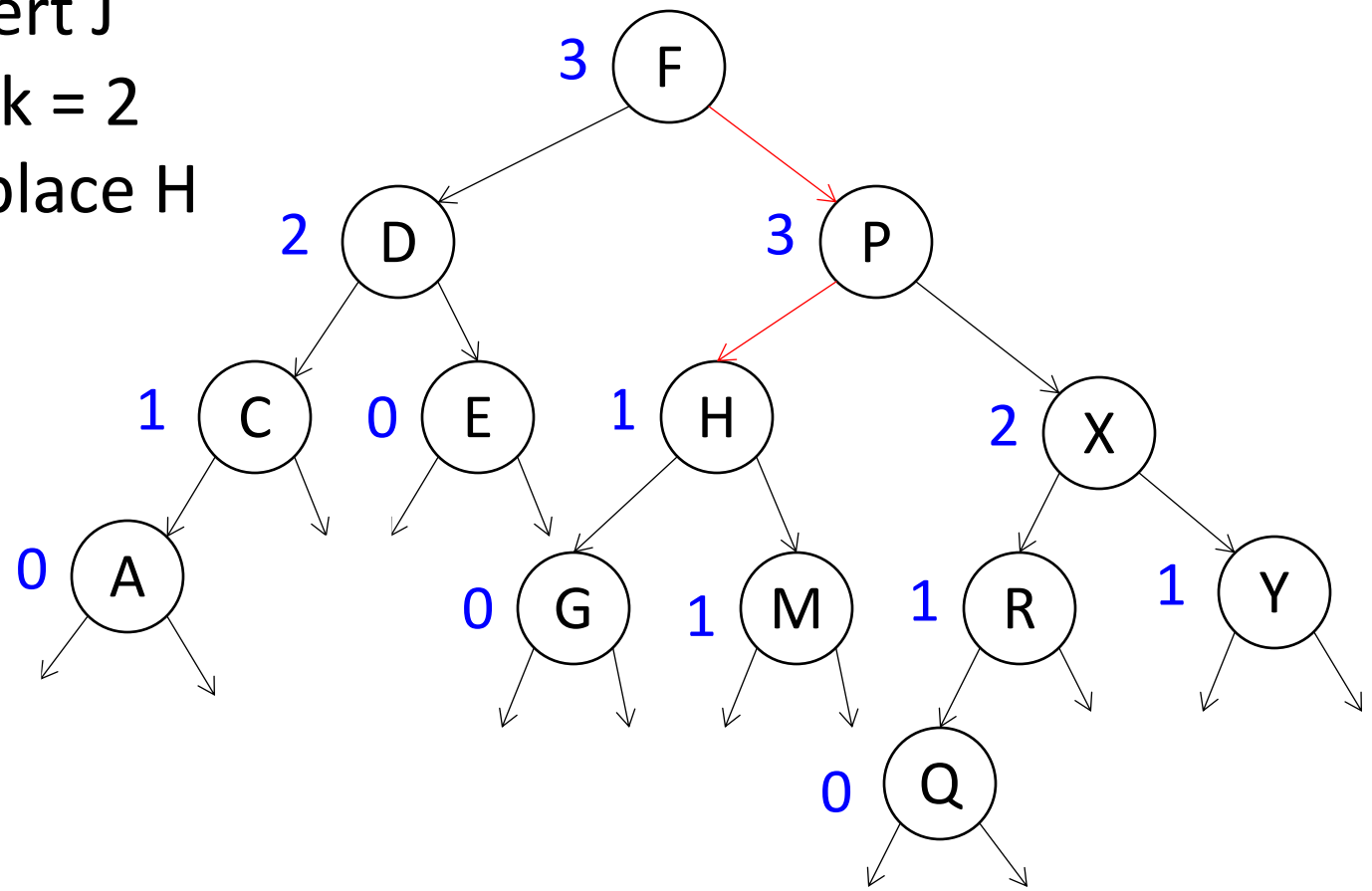
rank = 2



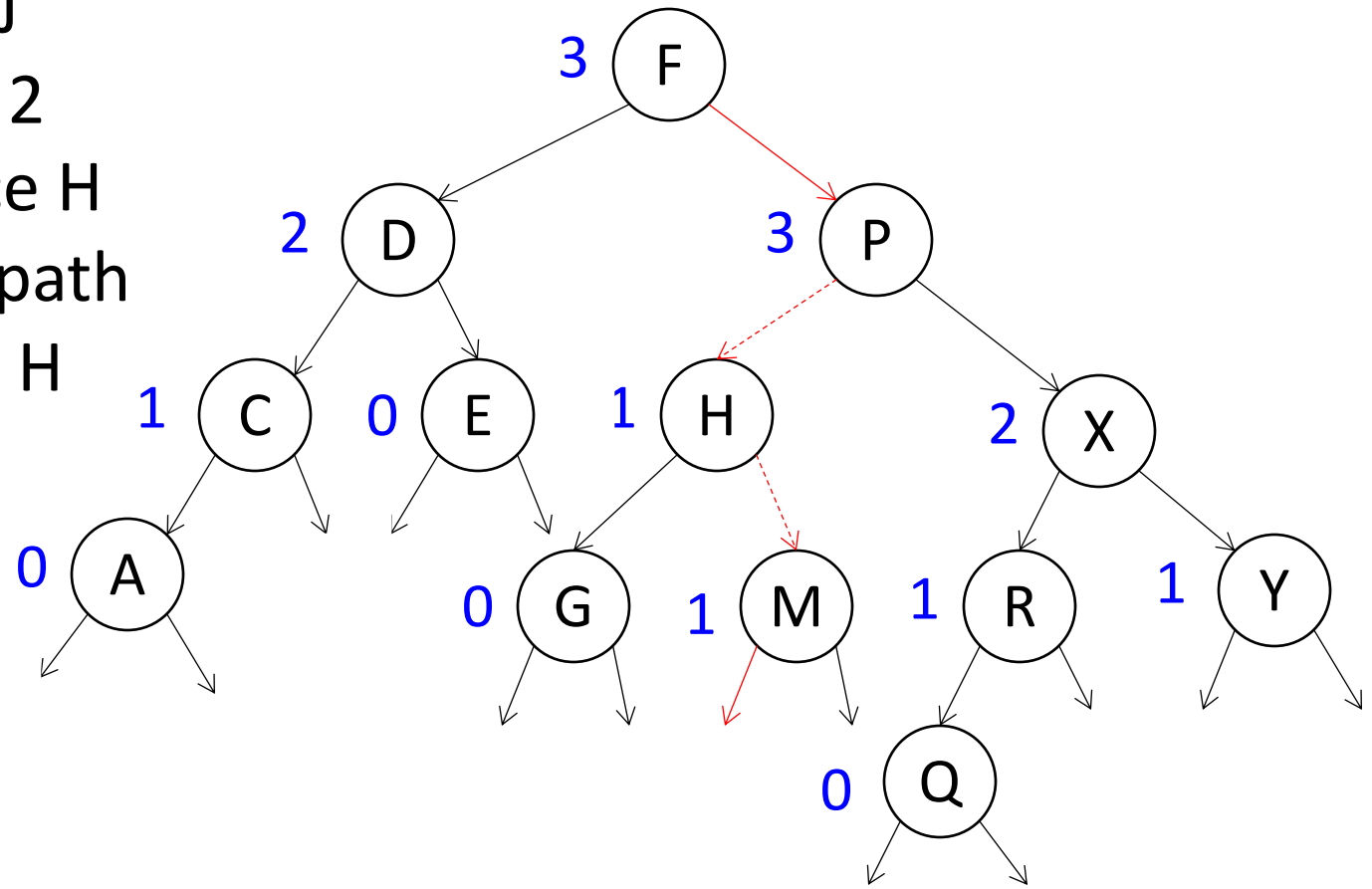
Insert J

rank = 2

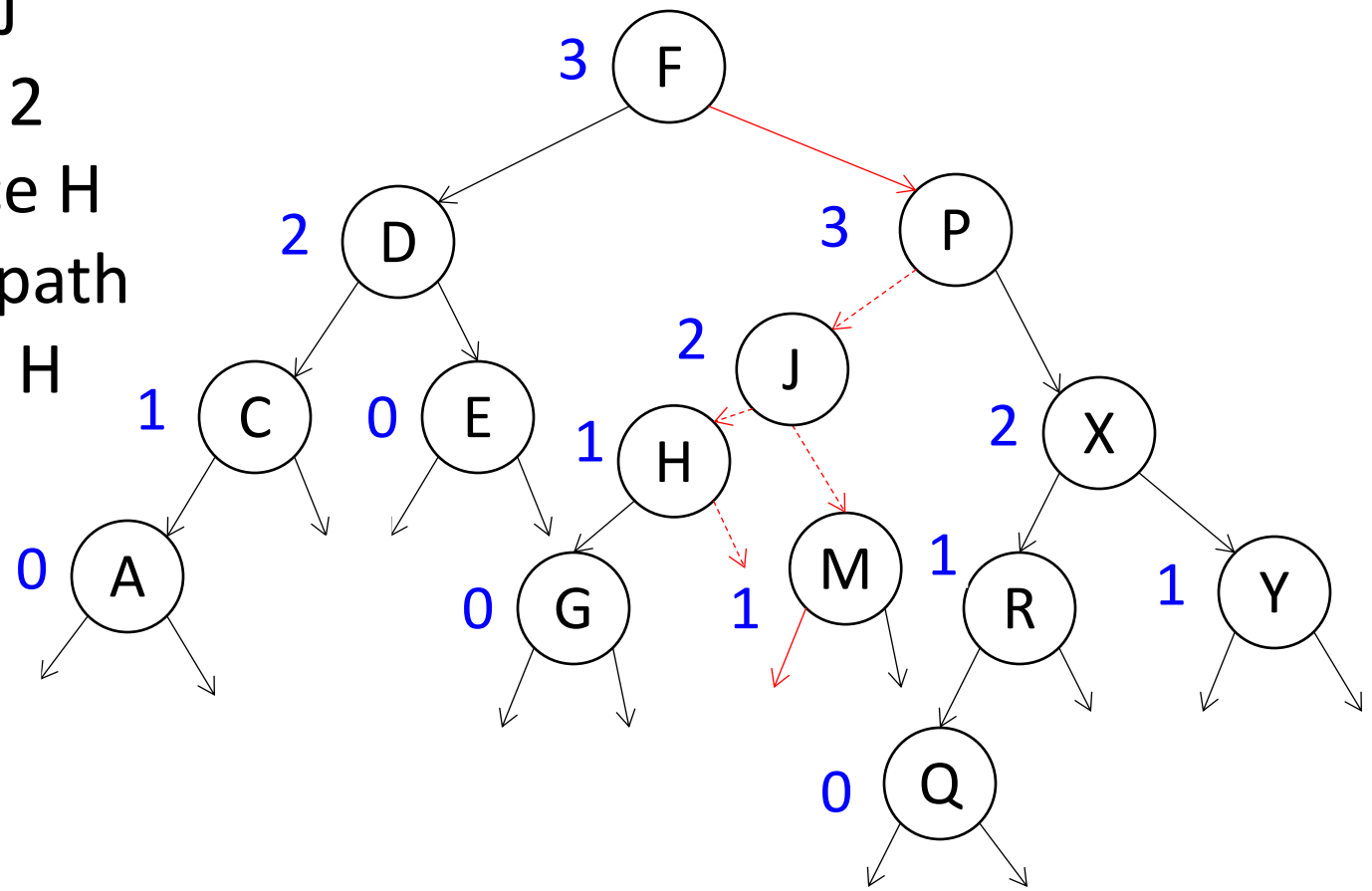
Replace H

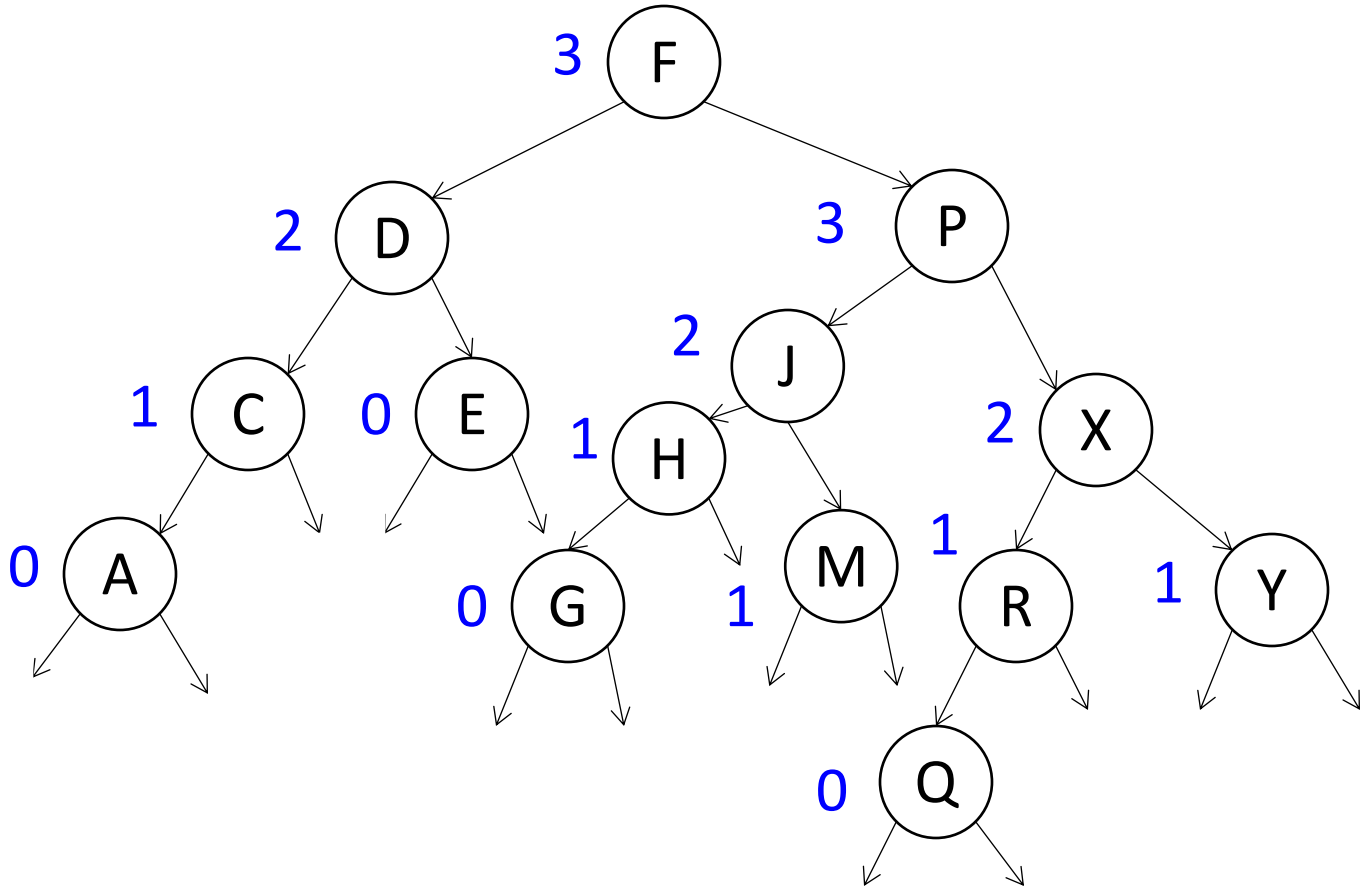


Insert J
rank = 2
Replace H
Unzip path
from H



Insert J
rank = 2
Replace H
Unzip path
from H

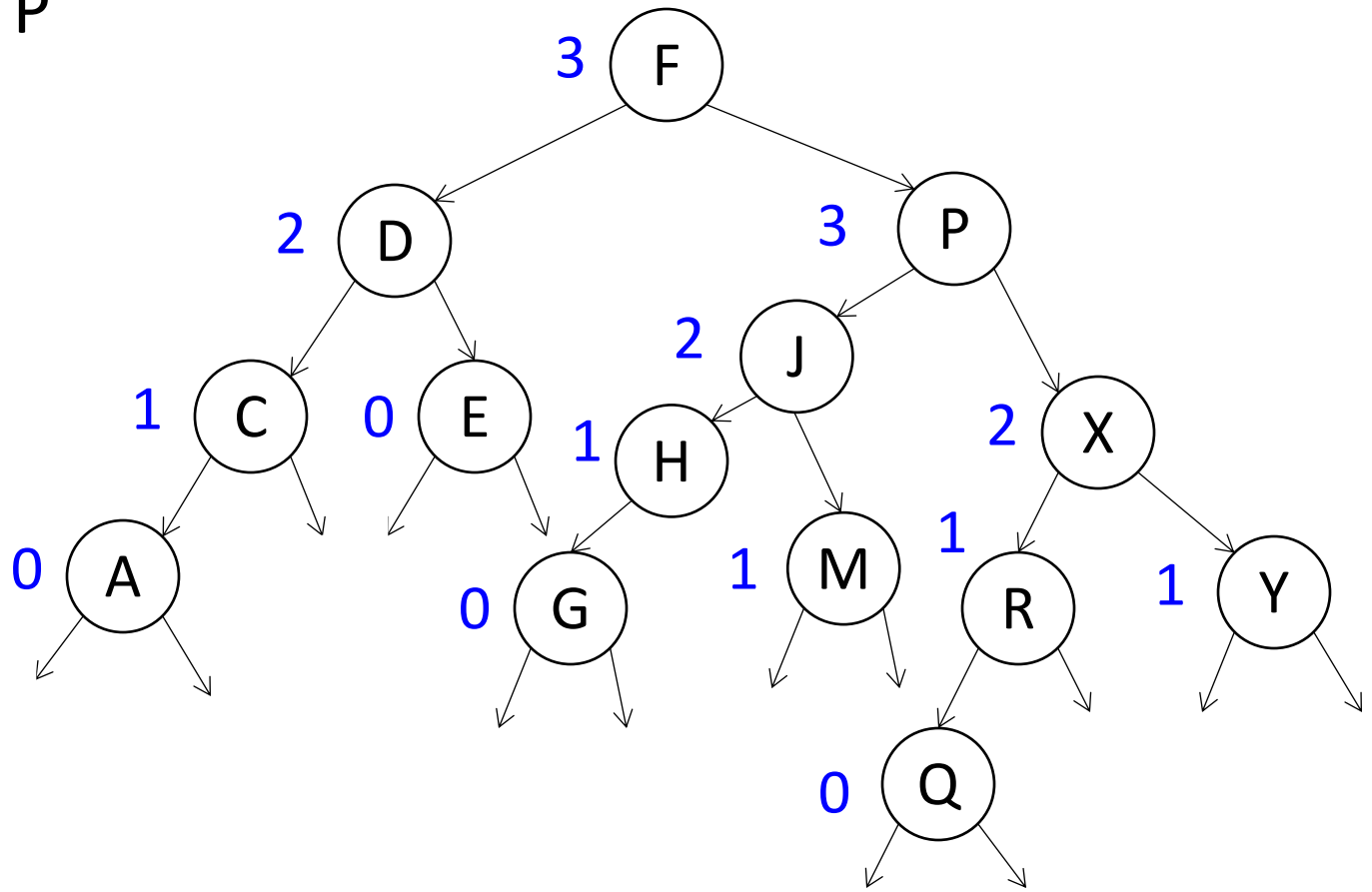



3
F
2
D
3
P
1
C
0
E
2
J
2
X
0
A
1
H
0
G
1
M
1
R
1
Y
0
Q

Zip tree deletion: Inverse of insertion

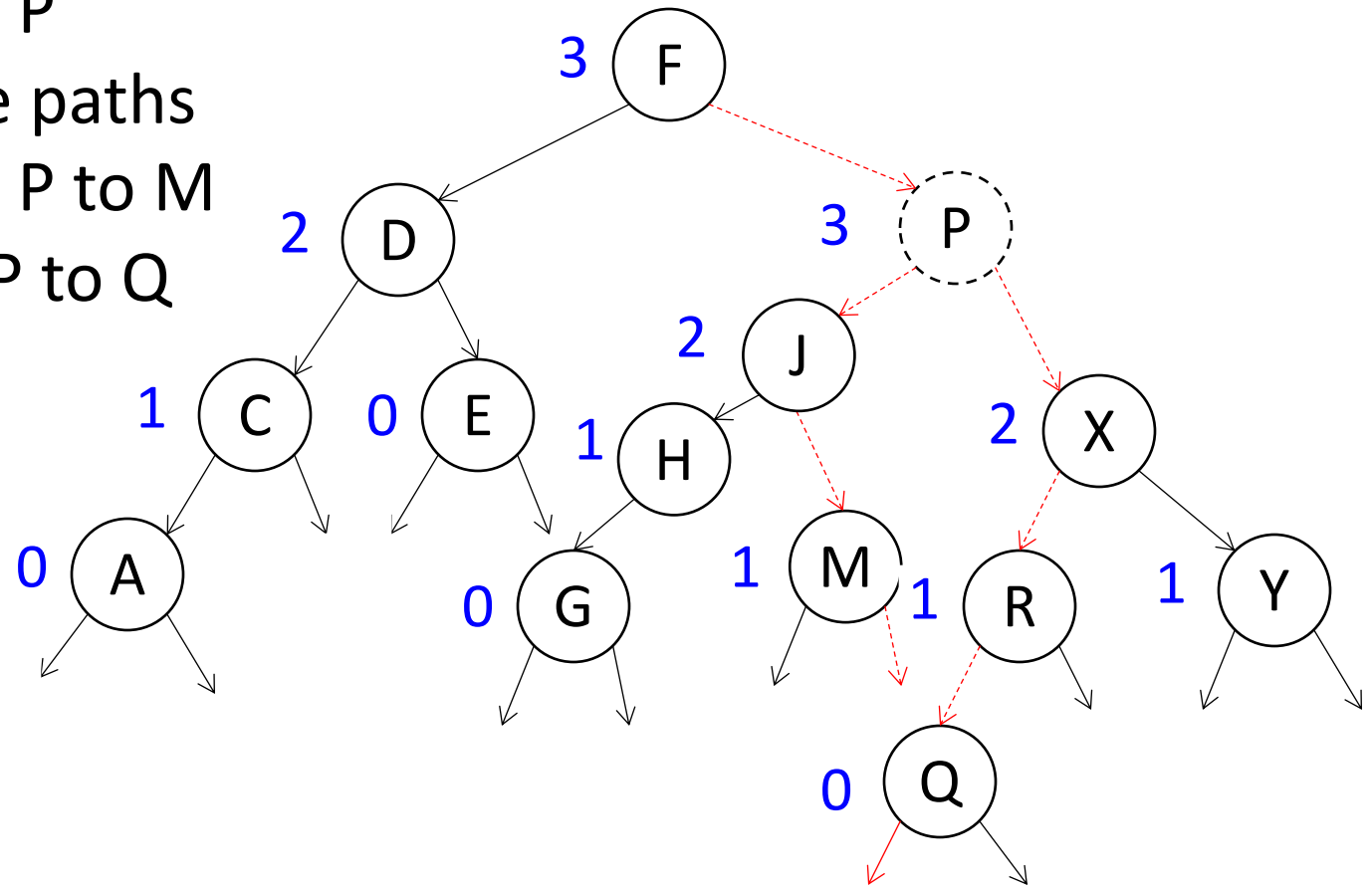
Search for the node *x* to be deleted. *Zip* the path from *x* to its predecessor (in key order) with the path from *x* to its successor (in key order), by merging them in decreasing rank order, breaking ties in favor of smaller keys, to form a single path *P*. Replace *x* as a child of its parent by the top node of *P*.

Delete P



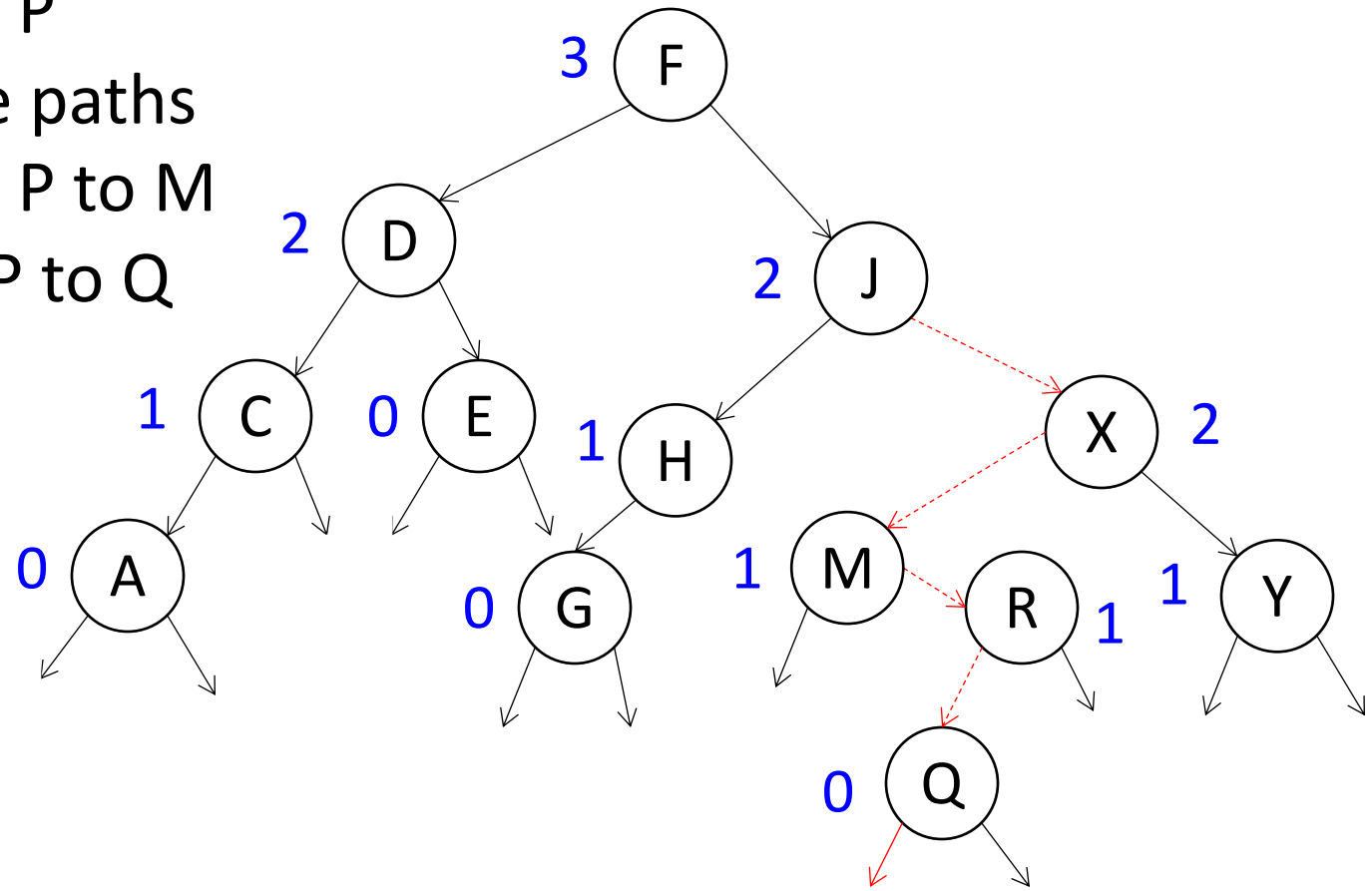
Delete P

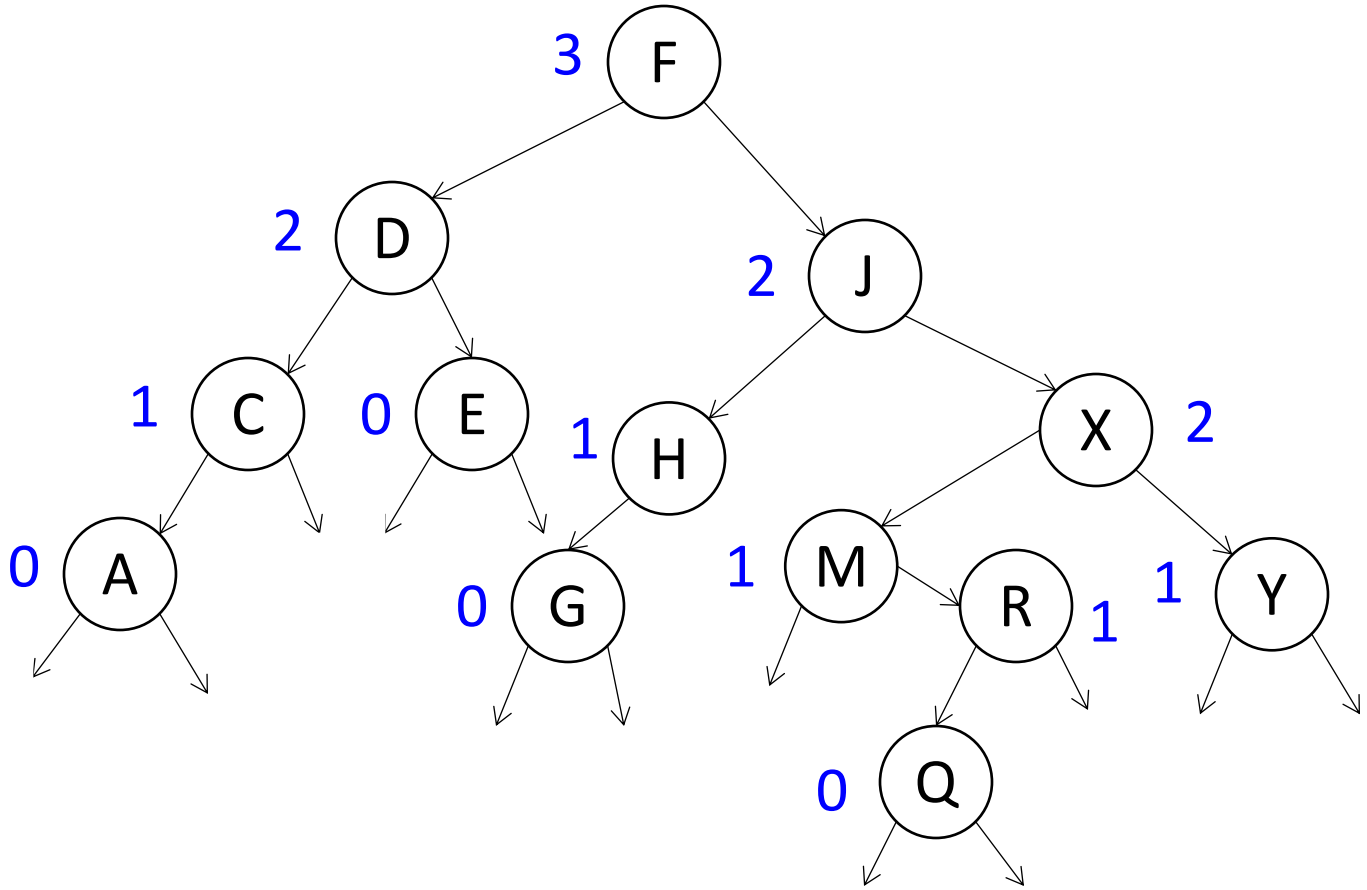
Zip the paths
from P to M
and P to Q



Delete P

Zip the paths
from P to M
and P to Q




3
F
2
D
2
J
1
C
0
E
1
H
X
2
0
A
0
G
1
M
R
1
1
Y
0
Q

A *Zip* Tree

(image from Street Art on Pinterest)

Zip Trees

- Tree height is O(log n) with high probability, making insertions and deletions efficient as well as simple
- Inserts and deletes can be implemented to proceed purely top-down
- Inserts and deletes do O(1) expected restructuring: Most of the changes are in the bottom of the tree
- Can modify to support frequency-biased access
- https://arxiv.org/abs/1806.06726

Classic way to make BST's efficient: keep the tree *balanced*

Maintain a local balance condition so that all path lengths are O(log *n*)

AVL trees: Adelson-Velsky, Landis 1971

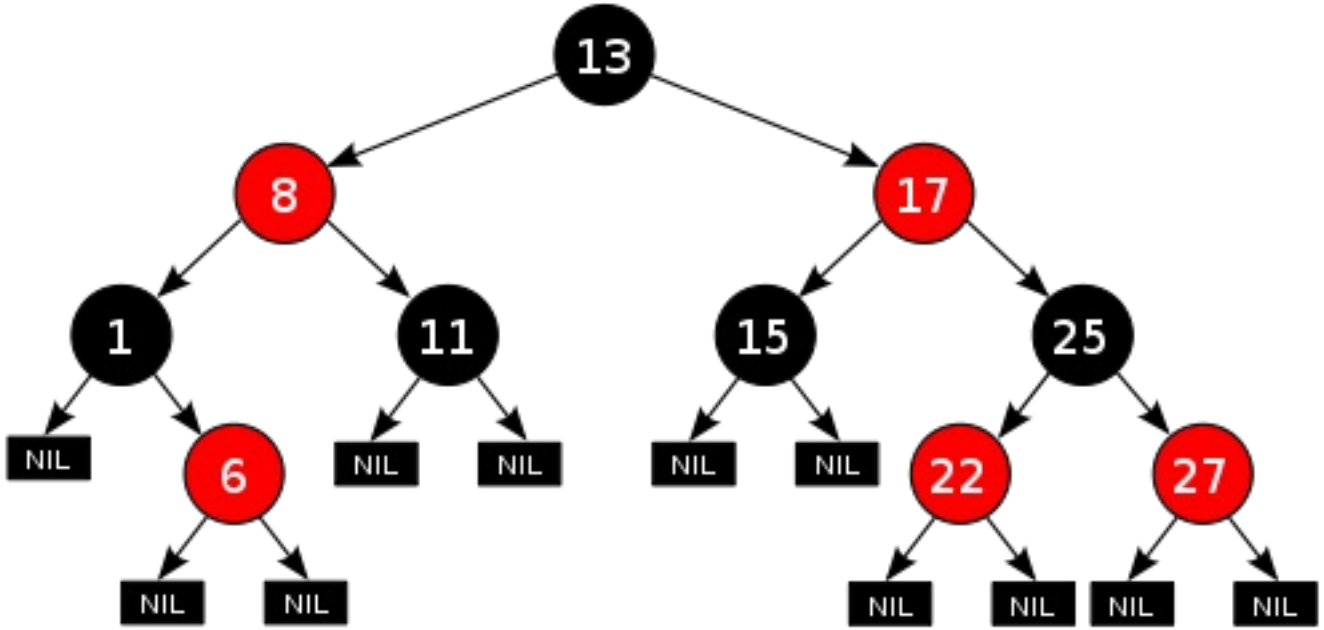
red-black trees: Bayer 1972, Guibas and Sedgewick 1978

MANY others…

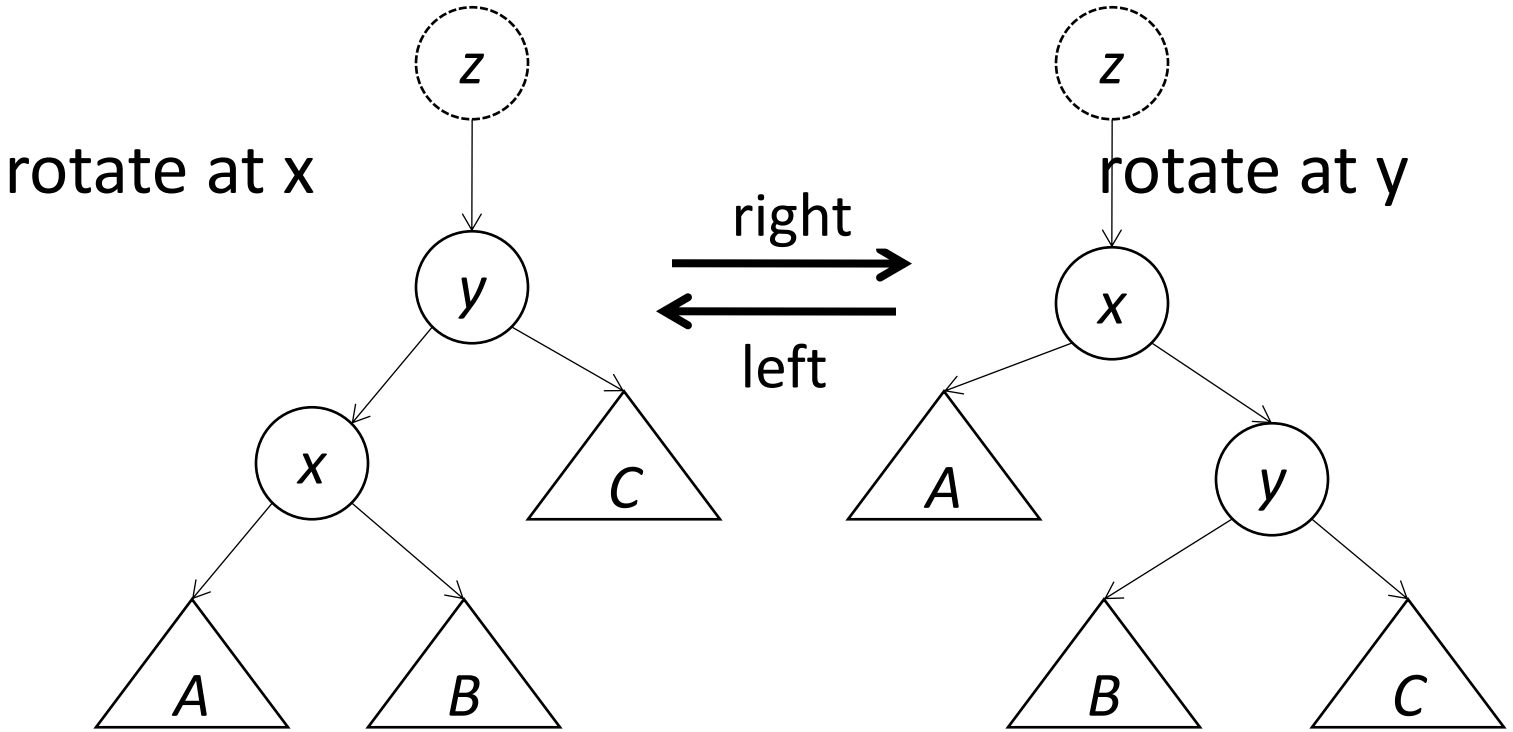
Need:

- A balance condition
- A way to restructure the tree during an update to maintain balance

A red-black tree
(image from Wikipedia)



Restructuring primitive: rotation



Rebalancing

During an insertion, do rotations and update balance data to restore balance

Red-black tree insertion: can rebalance either bottom-up after insertion or top-down during the access

Guarantees O(log *n*) access, insertion (and deletion) time

Balanced tree drawbacks

Rebalancing algorithms have many cases

typically 6 for insert, 8 for delete

Must store balance data (but maybe only 1 or 2 bits)

In practice, access is not uniform

Is there a way to take advantage of non-uniform access?

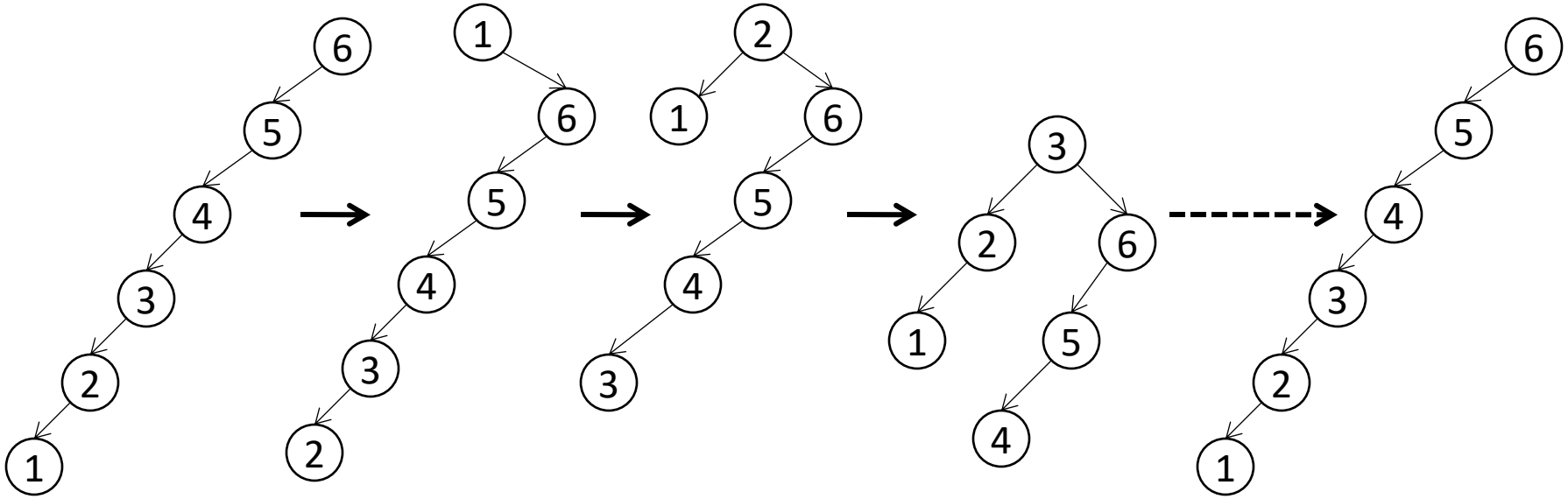
Self-adjusting binary search tree

Idea: move each accessed key to the root, via rotations

If the key is accessed again soon, this access will be fast

First try: move to root via bottom-up rotations

Bad example: access in order



n accesses in sorted order take $n^2/2$ node visits

and reproduce the original tree!

Second try: Splay Trees (Sleator and T 1983)

Splay: to spread out

splay(*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

x.p = parent of node *x*

splay(*x*): **while** *x.p* ≠ null **do**

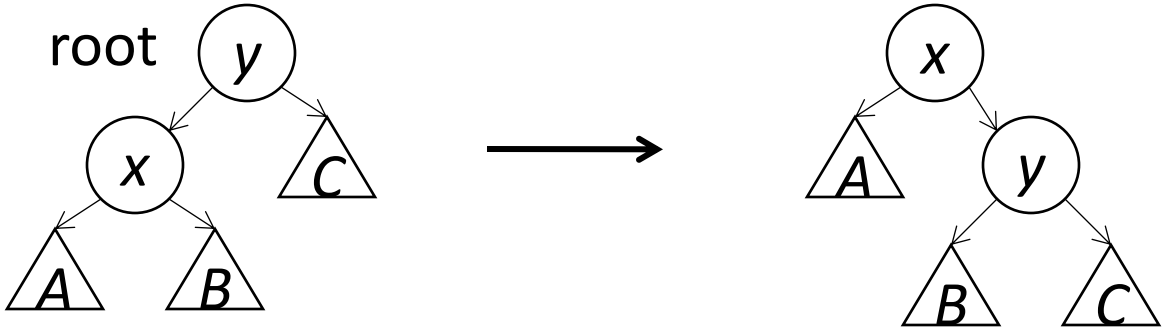
if *x.p.p* = null then *rotate*(*x*) *zig*

else if *x* is *left* **and** *x.p* is *right* **or** x is *right* **and**

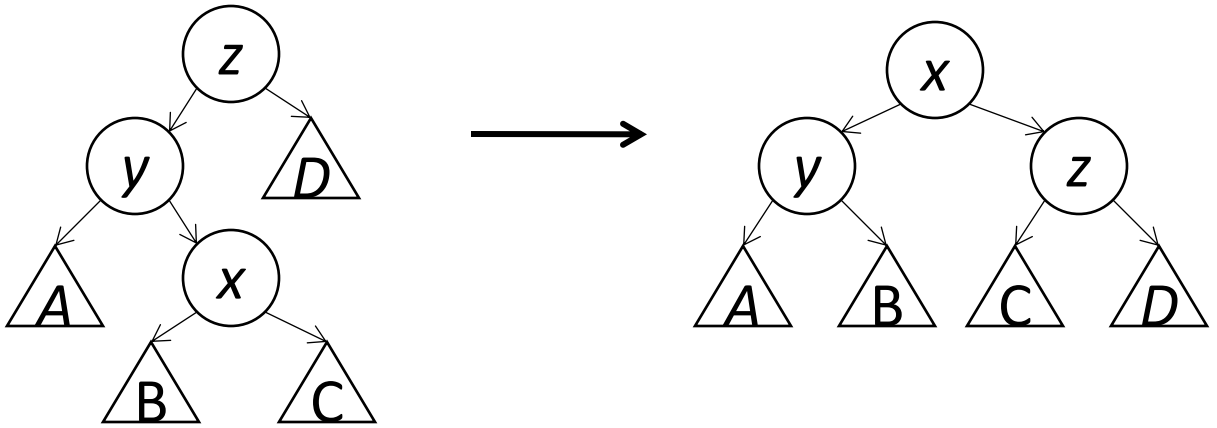
x.p is *left* then {*rotate*(*x*), *rotate*(*x*)} *zig-zag*

else {*rotate*(*x.p*), *rotate*(*x*)} *zig-zig*

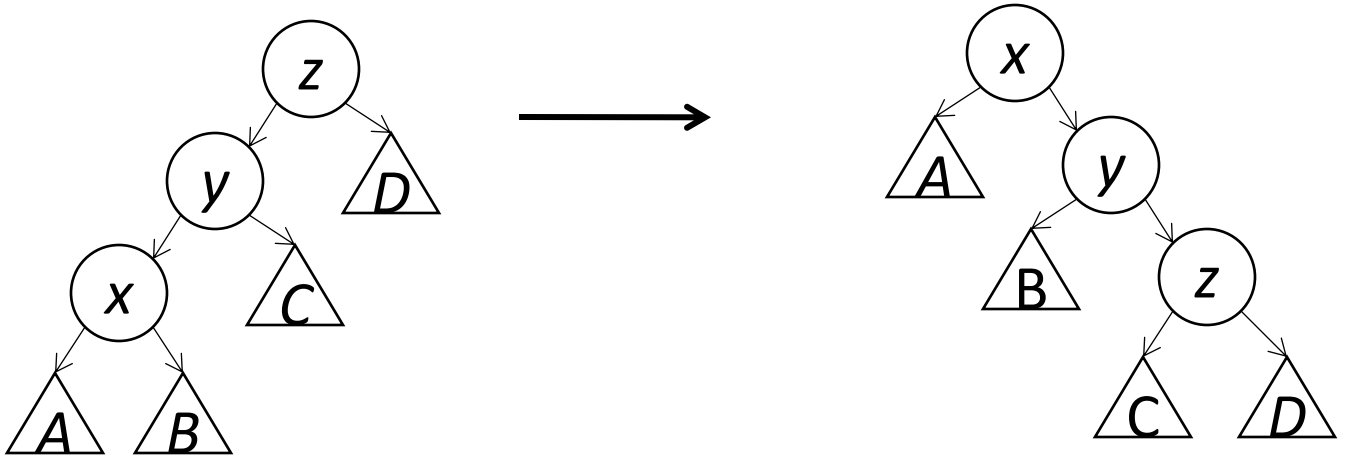
zig



zig-zag



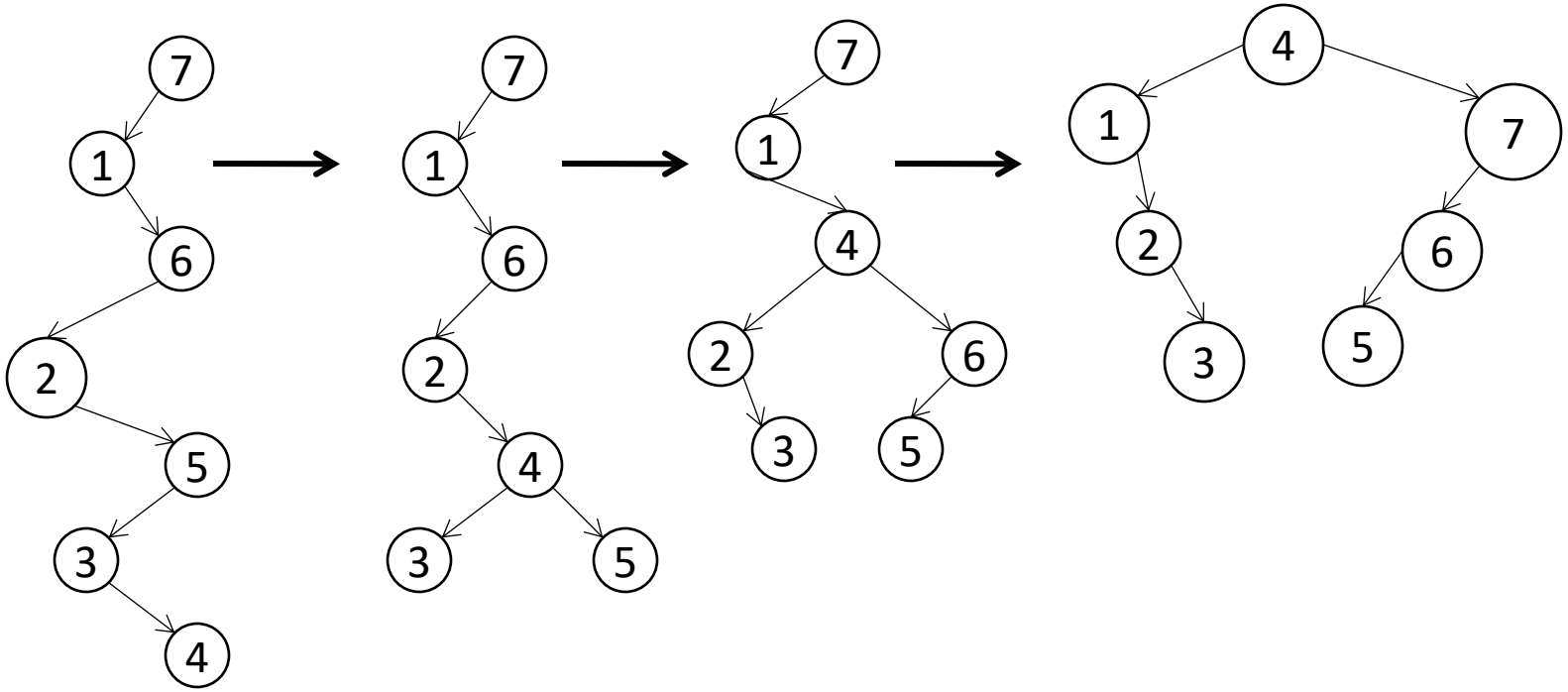
zig-zig



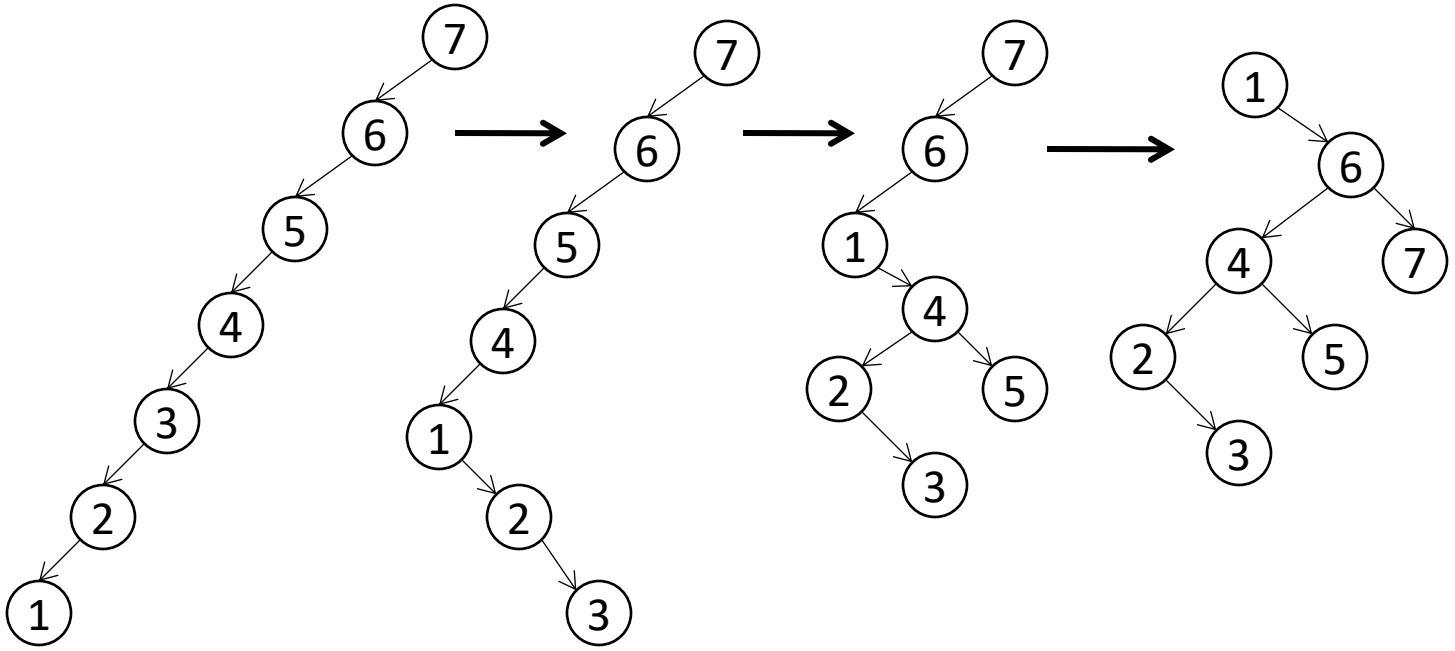
A Self-Adjusting Tree

(image by Jorge Stolfi)

Splay: pure zig-zag



Splay: pure zig-zig



Operations on splay trees

Access *x*: follow access path to *x*, then *splay*(*x*)

Insert *x*: follow access path to null, replace by *x*, *splay*(*x*)

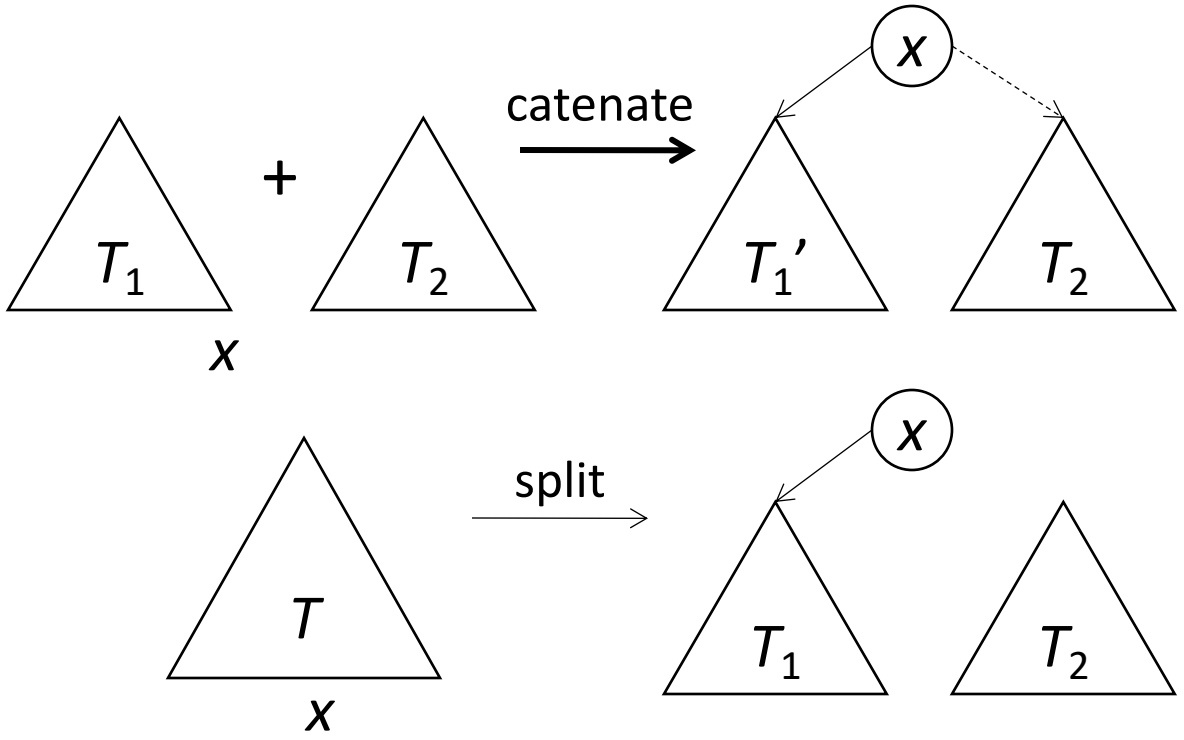
Delete *x*: follow access path to *x*, swap with successor if *x* is in a node with two children, delete *x*, splay at old parent of *x*

Time for an operation is proportional to number of nodes on access path, including one rotation per node on path (except root)

Catenate(T_1, T_2) (all items in T_1 < all items in T_2):

splay at last node x in T_1; $x.right \leftarrow root(T_2)$.

Split(T, x): *splay*(x); detach $x.right$ = root of tree containing all items > x.



Efficiency of Splay Trees

One operation can take many steps, even *n*

But long sequences of operations are fast:

m operations take O(*m*log*n*) time: amortized time per operation is O(log*n*)

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

Splaying exploits space or time locality just as well as complicated customized data structures (to within a small constant factor)

Just how good is splaying?

Dynamic optimality conjecture:

> Given an initial tree and any access sequence, splaying is as fast (to within a constant factor) as the best BST algorithm for the given sequence, even an algorithm that knows the entire sequence in advance

(Each access must be done by moving the accessed item to the root via rotations, at a cost of one plus the number of rotations)

For more, take COS 423!