Self-Adjusting Data Structures

Robert E. Tarjan

November 9,2021

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

Data structures

We need to do a sequence of simple operations

Amortization

When doing a sequence of operations, we may not care about the cost of individual operations. Our goal is to minimize the total cost of the sequence.

We can afford expensive operations if there are enough cheap ones.

Can we use this idea in the design and analysis of data structures?

Amortize: to liquidate a debt by installment payments.

From Medieval Latin: to reduce to the point of death.

In analysis of algorithms: to pay for the total cost of a sequence of operations by charging each operation an equal (or appropriate) amount.

Beyond the worst case

By allowing some expensive operations if they are balanced by many cheap ones, we expand the design space.

We can consider “out of balance” structures,as long as they are “in balance” often enough.

Beyond the worst case

Sometimes, the “worst case” is rare. If the sequence of operations has some structure, we would like to exploit this.

Are there “self-adjusting” data structures, which adapt to the way they are used?

We have already seen an example of a self-adjusting data structure.

What is it?

Union-Find

Is weighted quick-union self-adjusting?

Union-Find

Is weighted quick-union self-adjusting?

No. By itself, it reduces the worst-case time per find to logarithmic.

Union-Find

Is path compression self-adjusting?

Union-Find

Is path compression self-adjusting?

Yes. By compressing each find path (paying a constant factor), it speeds up later finds. (At least that is the hope…)

Does it work?

Tree representation of disjoint sets

Represent each set by a rooted tree, whose nodes are the elements in the set, with each node x having a pointer to its parent; if x is a root, x points to itself. Store information about the set (such as its name) in the root.

To *find* an element, follow the path of parent pointers to the root, return the root.

To *unite* the sets containing two elements, find the roots of their sets, make one point to the other.

The shape of each tree is *arbitrary*. The shape is determined by the execution of the operations.

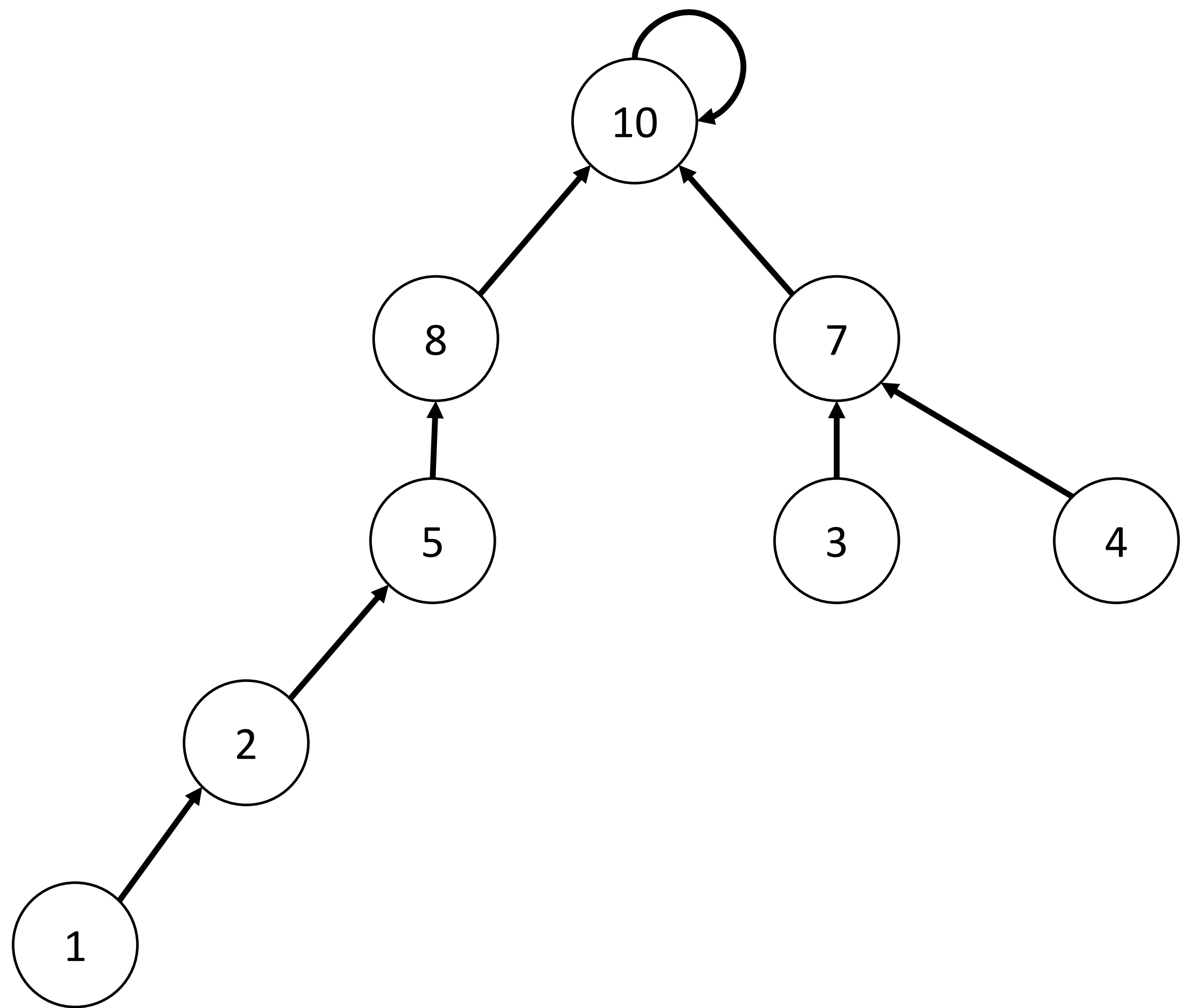
Weighted quick union

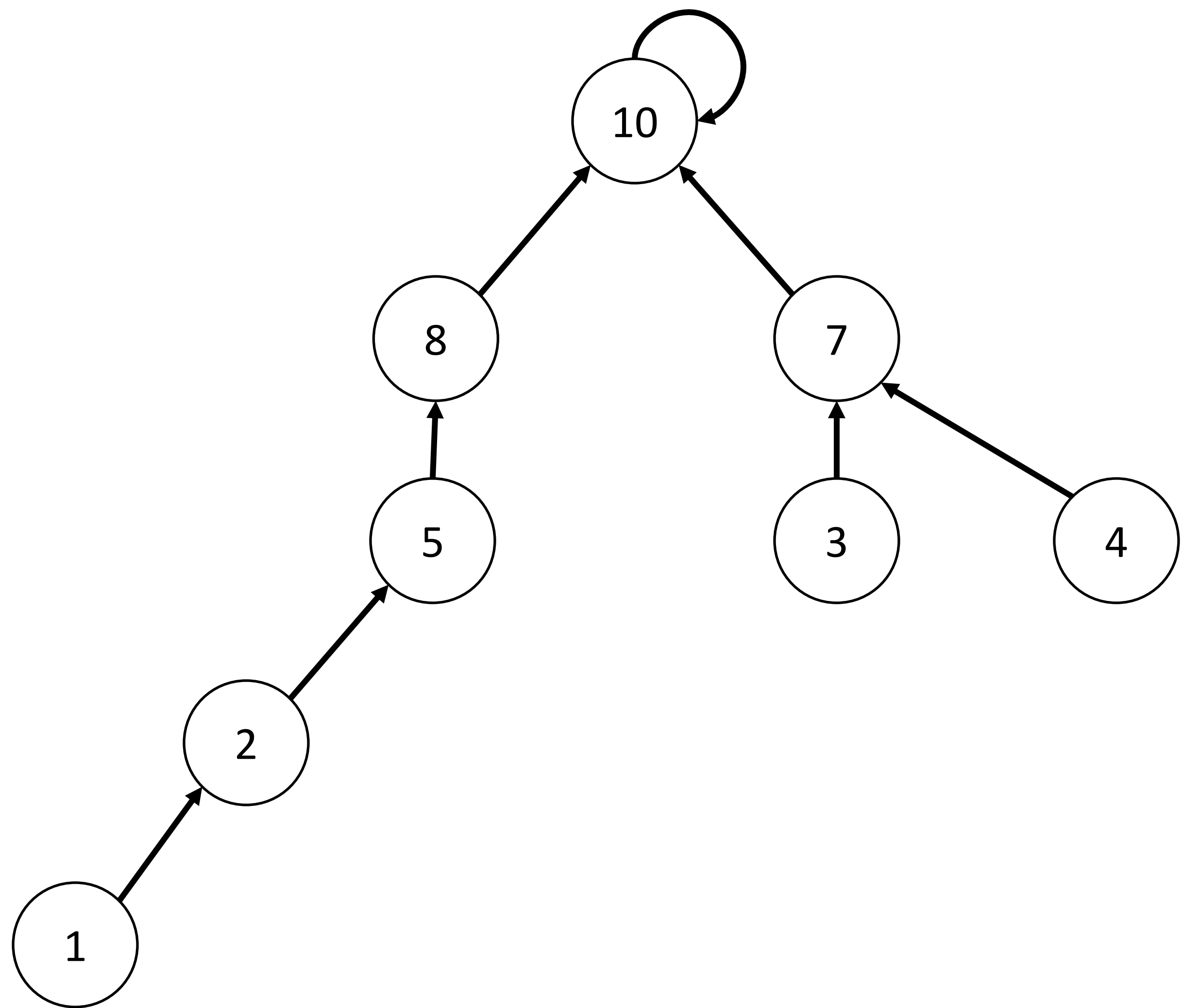
Store the number of nodes in each tree in its root. When uniting the sets containing two elements, make the root of the smaller tree point to that of the larger tree, breaking a tie arbitrarily.

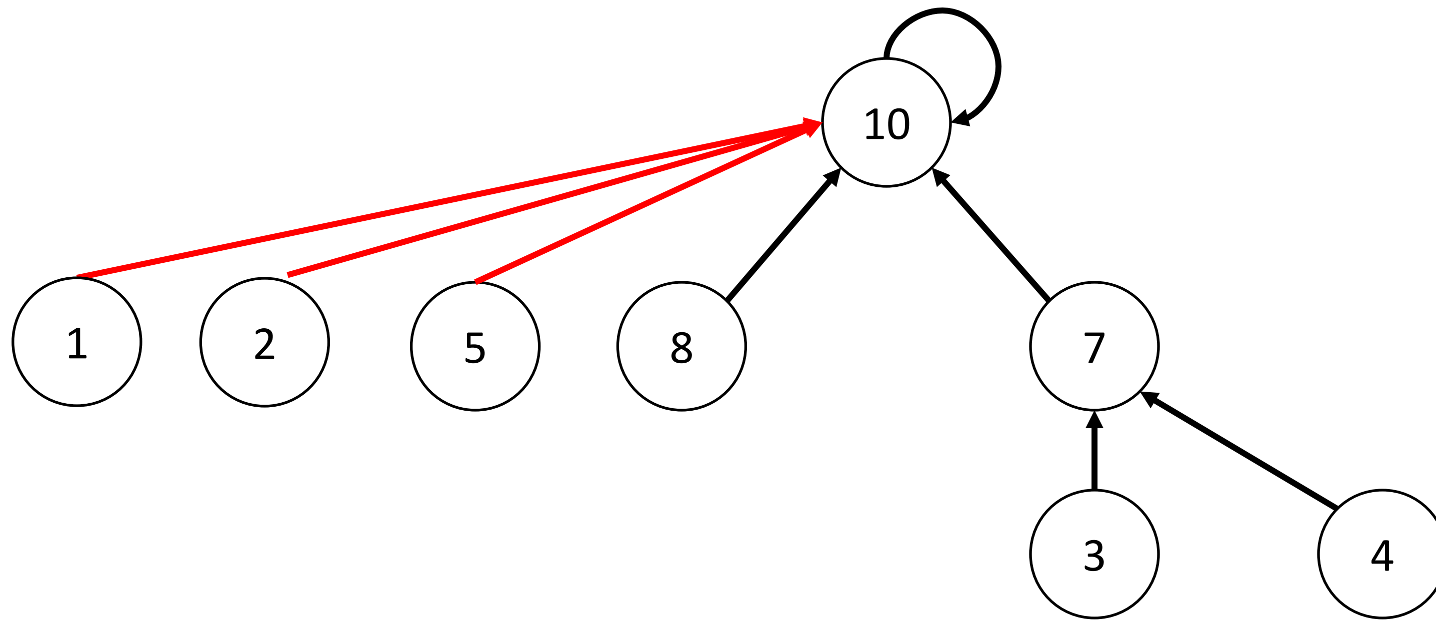
Find with path compression

After finding the root of the tree containing a given element, traverse the path again, making each node on the path point directly to the root.

A set of trees with finds done with path compression is a self-adjusting data structure


10
8
7
5
3
4
2
1


10
8
7
5
3
4
2
1



Path compression

How efficient are weighted quick union and finds with path compression when used together?

History of bounds (amortized time per *find*)

m = number of finds, n = number of elements

1971 O(1) (incorrect proof)

1972 $O(\lg\lg n)$ M. Fischer

1973 $O(\lg^* n)$ Hopcroft & Ullman

1975 $\Theta(\alpha(n, m/n))$ Tarjan

lower bound analysis is top-down

upper bound analysis is bottom-up

later $\Omega(\lg\lg n)$ (incorrect proof)

2005 $O(\alpha(n, m/n))$ Seidel & Sharir (top-down)

Ackermann's function (one version)

$A_1(n) = 2n$

$A_k(0) = 1$ if $k > 1$

$A_k(n) = A_{k-1}(A_k(n-1))$ if $k > 1$, $n > 0$

$= A_{k-1}^{(n)}(1)$

$= A_{k-1}$ applied to 1, n times

$A_2(n) = 2^n$, $A_3(n)$ = tower of n 2's, $A_4(n)$ grows very fast

$\alpha(n, d) = \min\{k > 0 \mid A_k(\lceil d + 2 \rceil) > n\}$

Another example: binary search trees

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 for these.

Binary Search

Maintain set S in sorted order.

To find x in S:

If S empty, stop (failure).

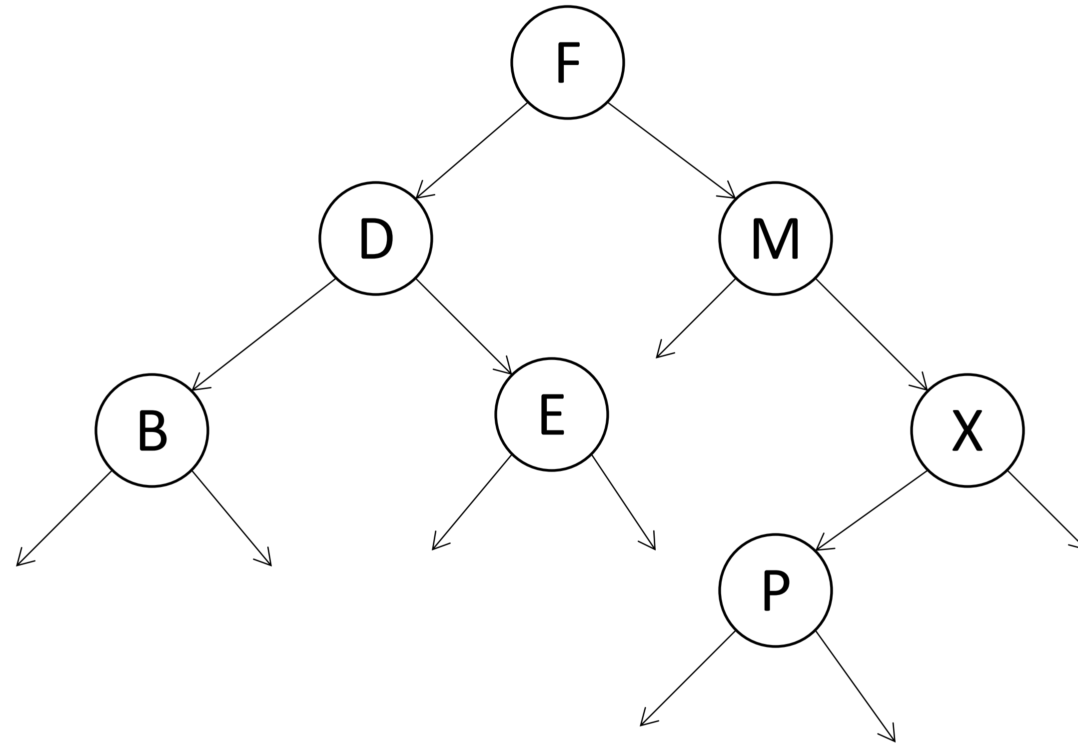
If S non-empty, compare x to some item y in S.

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

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

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

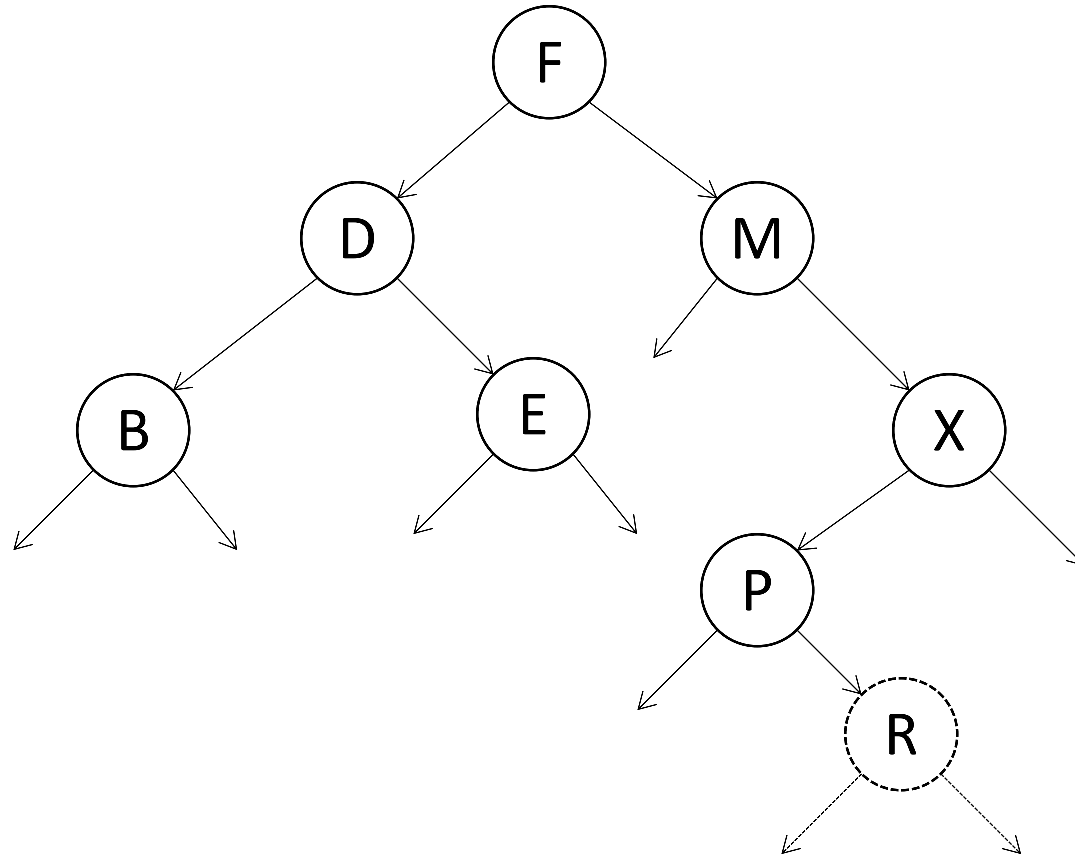
Data Structure:
Binary Search Tree



Insertion

Search. Replace missing node by item.

Insert R



Deletion

Find item. Remove node. Repair tree.

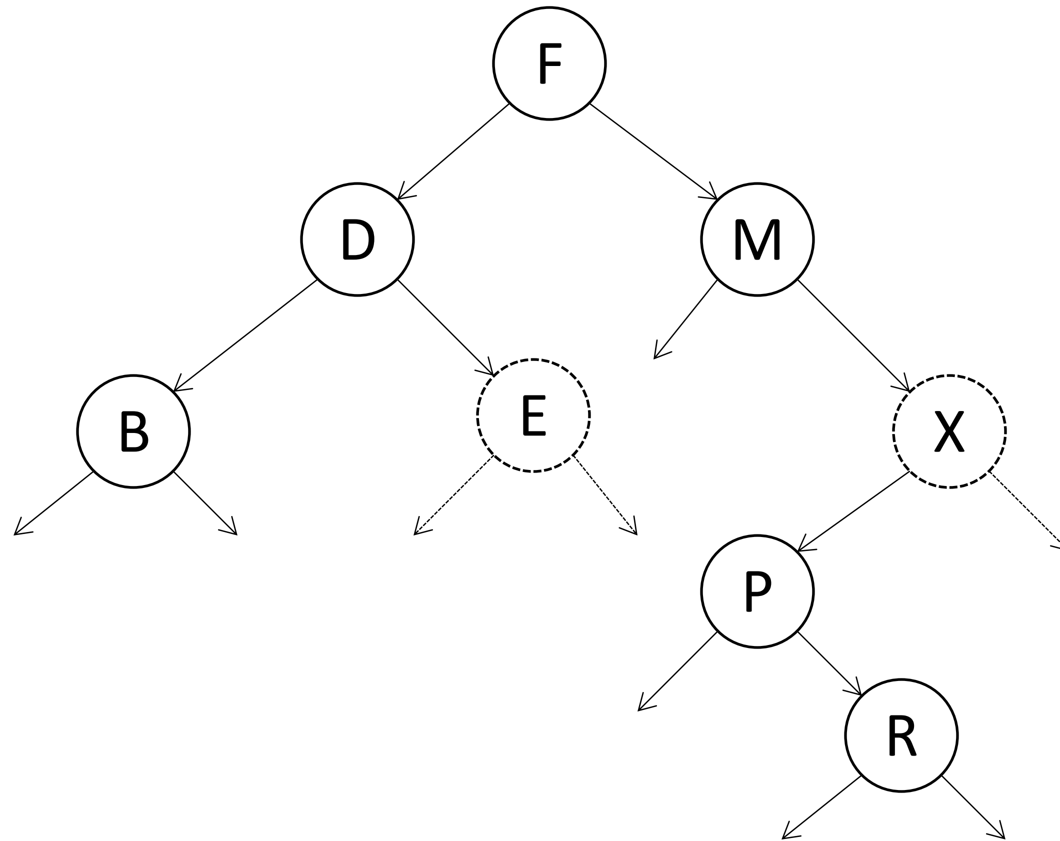
If leaf (no children), delete node.

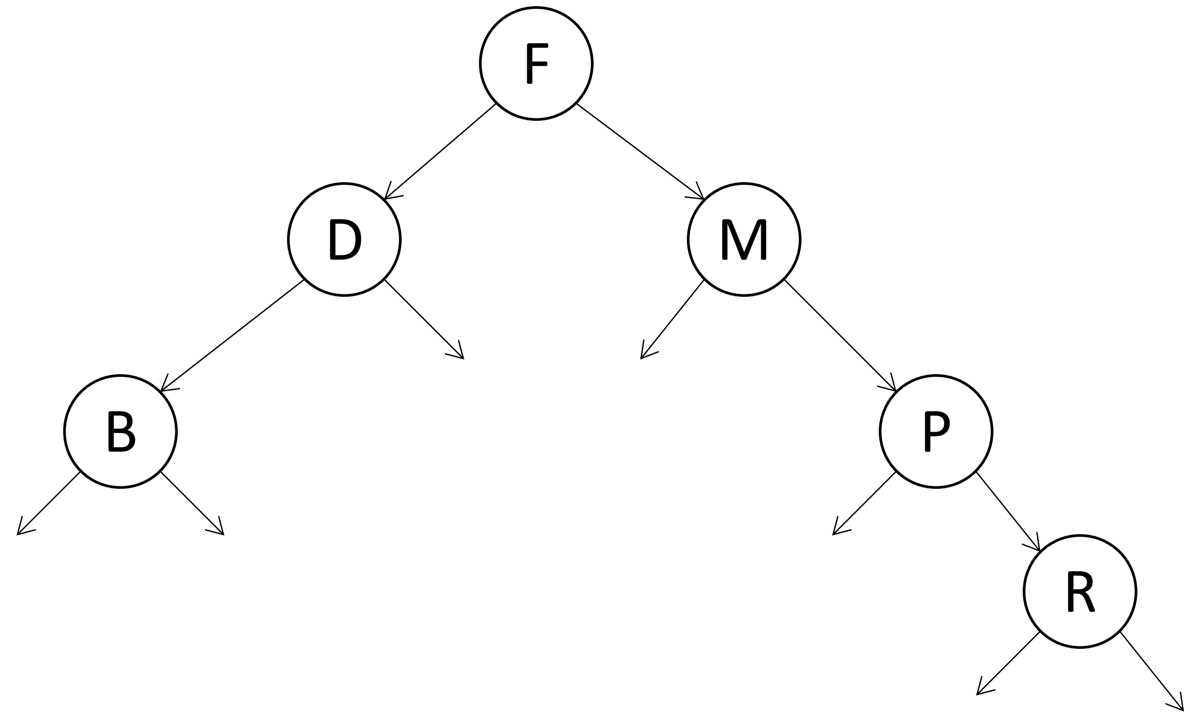
If unary (one child), replace by other child.

If binary?

Delete E

Delete X



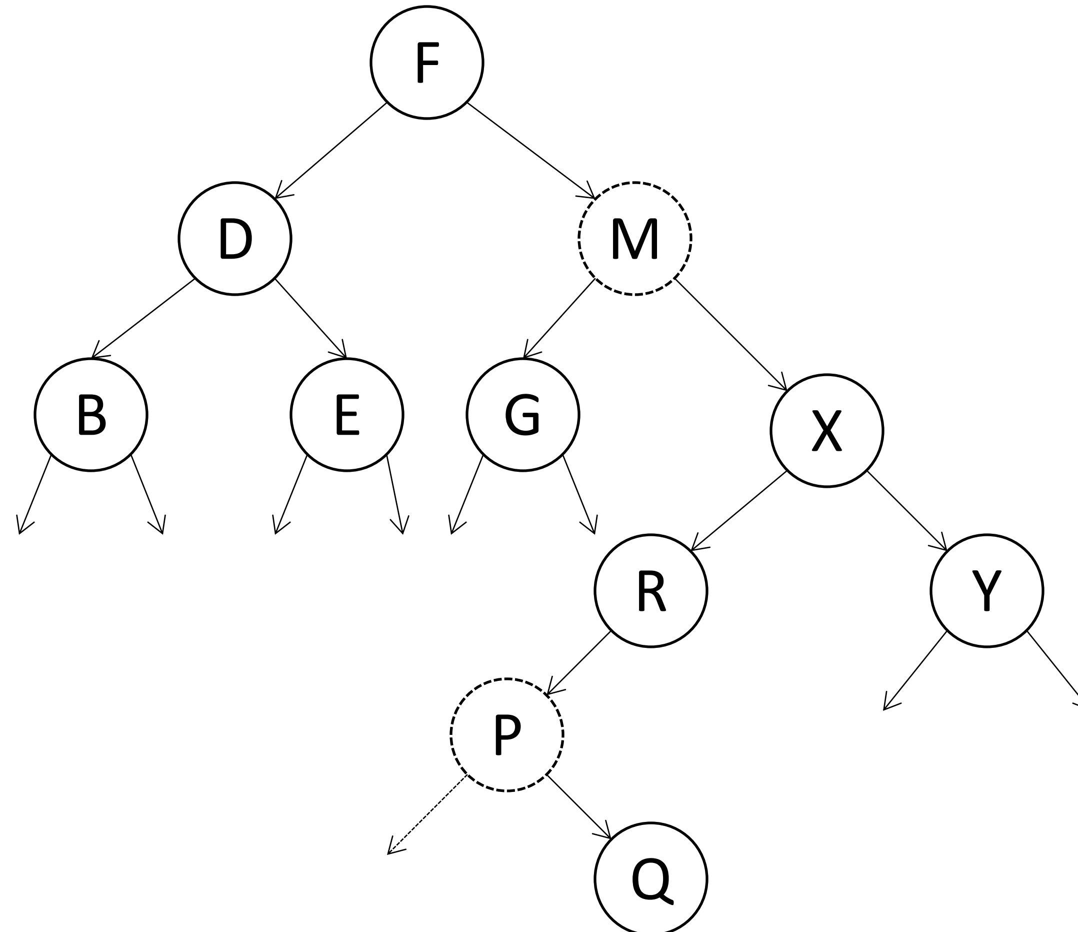

F
D
M
B
P
R

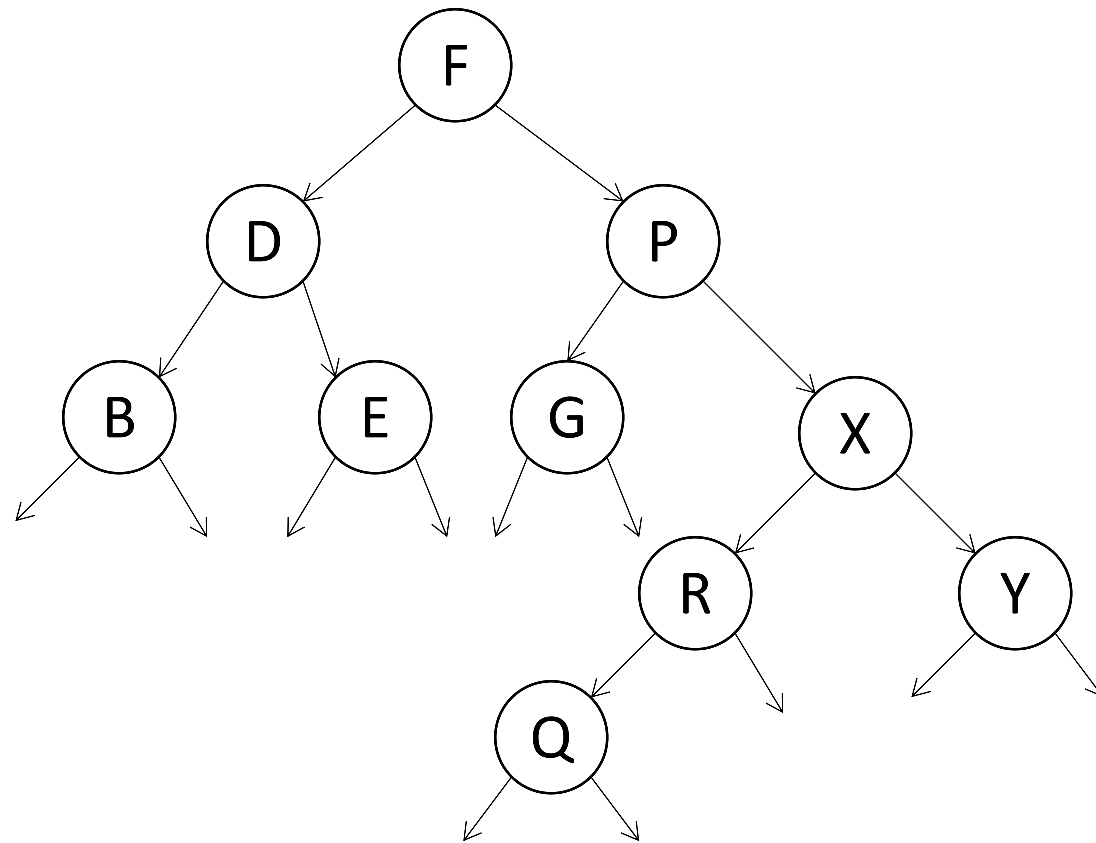
If binary, swap with next item. Now in leaf or unary node; delete. To find next item, follow left path from right child.

Delete M:

Swap with P;

delete.




F
D
P
B
E
G
X
R
Y
Q

Time Per Operation

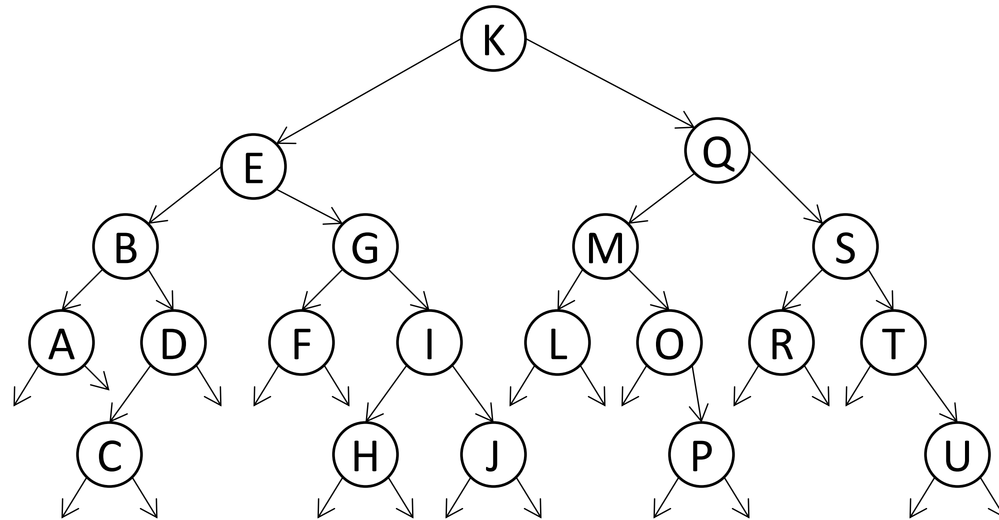
Proportional to depth of deepest node reached
during operation (length of path from root)

Goal: minimize tree depth

Best Case

All leaves have depths within 1: depth $\lfloor \log_2 n \rfloor$

n = number of items



Can achieve if tree is static

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

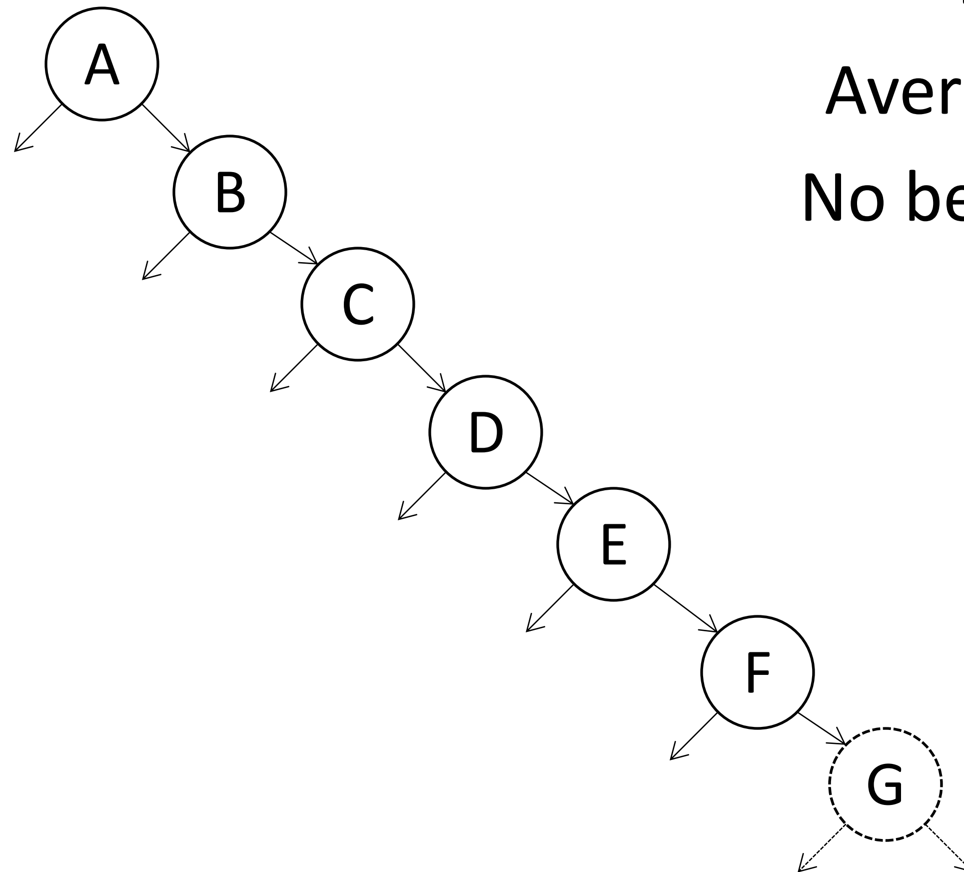
Natural but bad insertion order: sorted.

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(log$n$).

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?

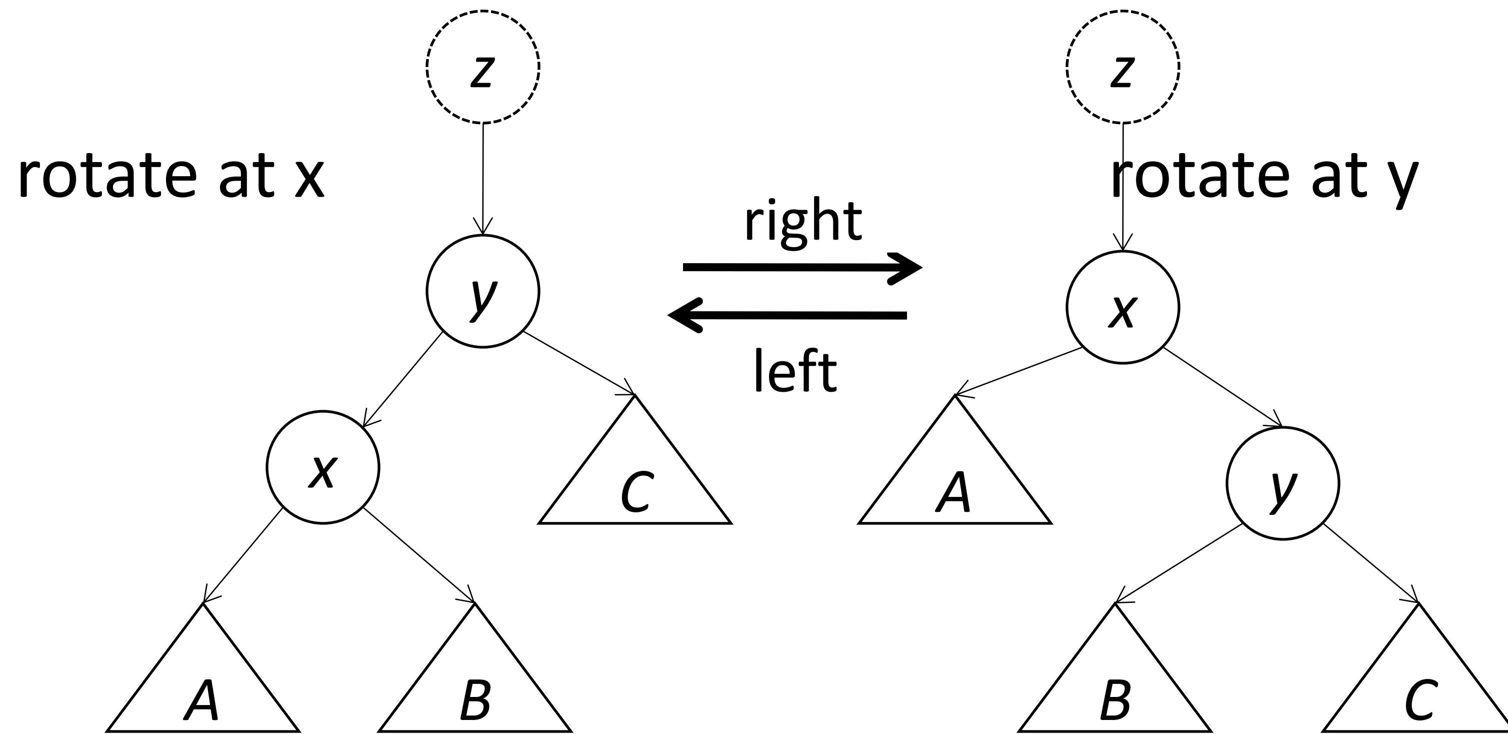
Restructuring primitive: *Rotation*

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

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 (number of nodes in subtrees) not too far apart (constant ratio).

(Left leaning) red-black trees

- Each link is red or black
- All paths from root to a missing node have the same number of black links (link to a missing node is black)
- No two red links in a row on a path from root to a missing node
- (Left leaning) Every red link is a left link

During inserts (and deletes), color flips and rotations are done along the access path to restore balance (restore the color rules)

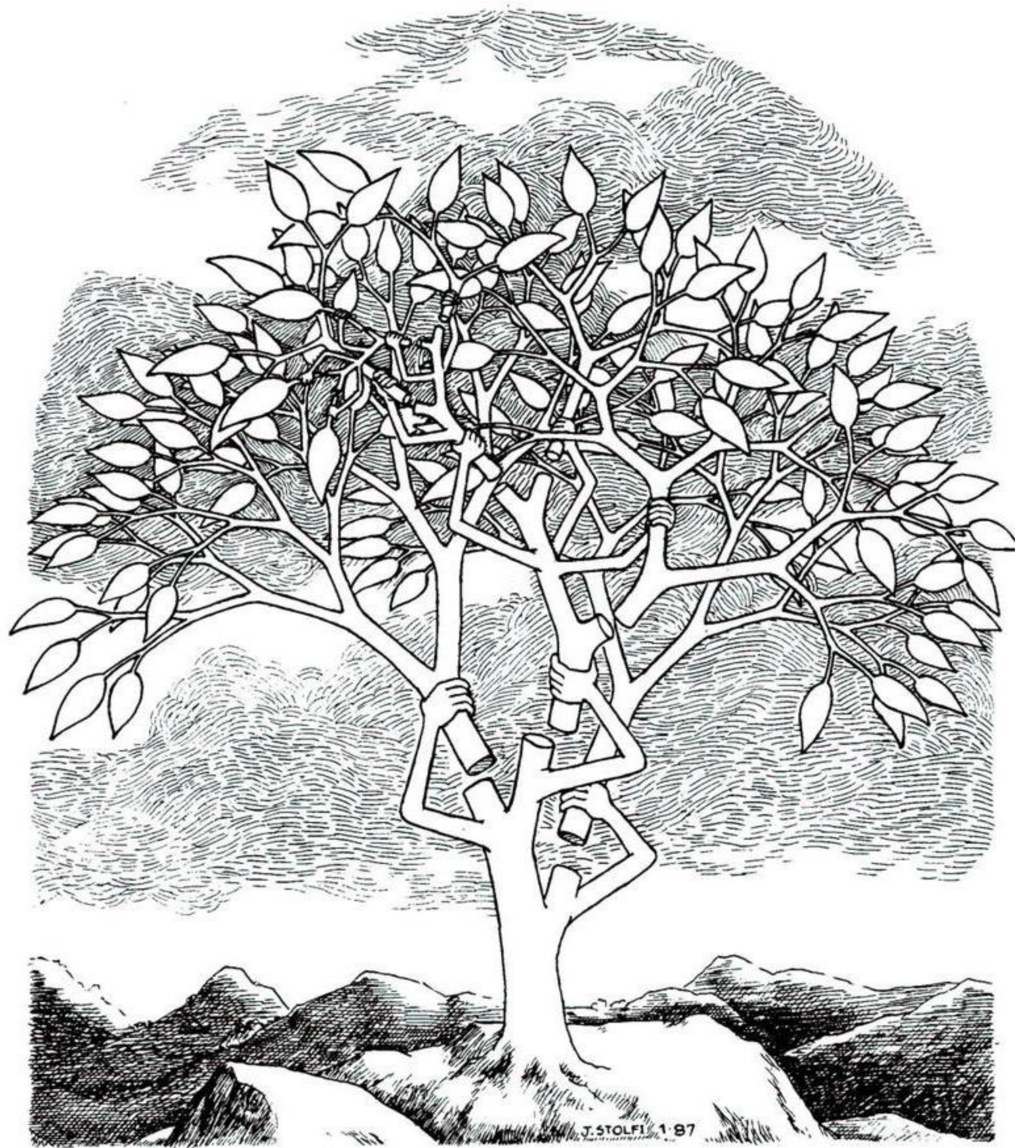
Balanced trees minimize worst-case access time to within a constant factor, but what if accesses are not uniform?

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?

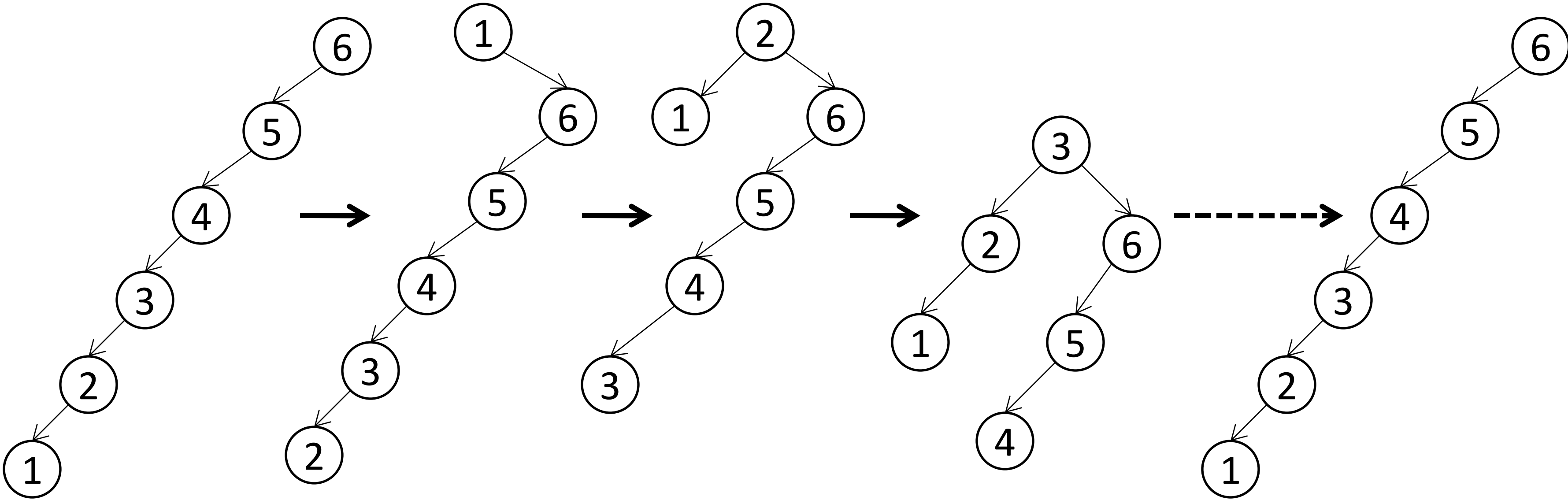
Is there a simple way to adjust a tree to match its usage?


J. STOLFI 1·87

First try: move to root

After an access, move the accessed node to the root by repeatedly doing rotations.

Bad example: sequential access



n accesses in sequential order take $\sim n^2/2$ rotations
and recreate the original tree!

Splay trees (Sleator and Tarjan 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.

splay(x): **while** $x.p \neq$ null **do**

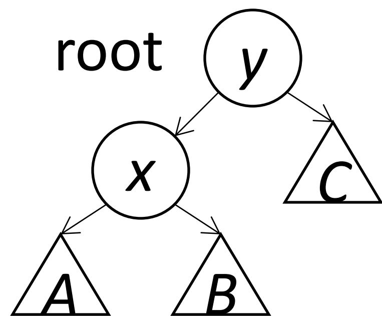
if $x.p.p$ = null then *rotate*(x) [*zig*]

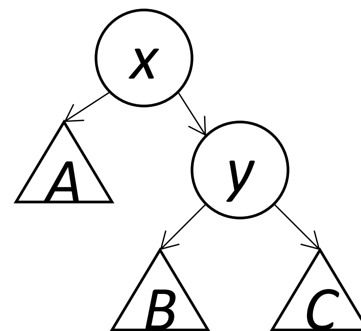
else if x is *left* **and** $x.p$ is *right* **or** x is *right* **and**

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

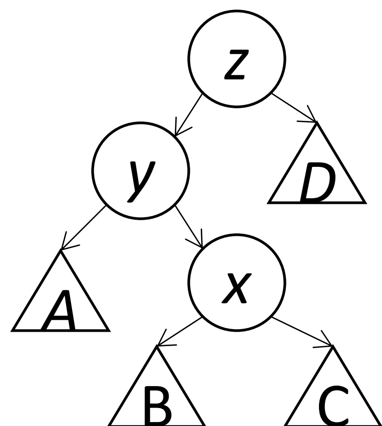
else {*rotate*($x.p$), *rotate*(x)} [*zig-zig*]

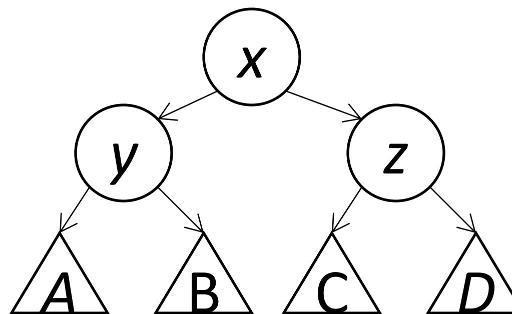
zig



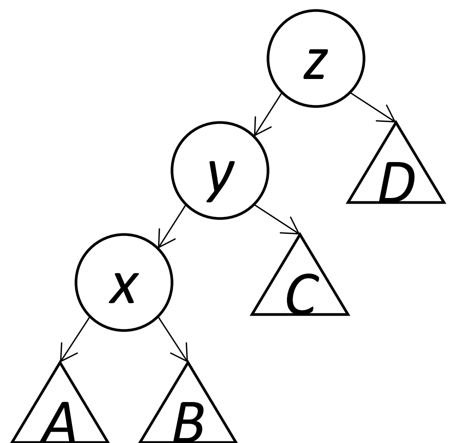


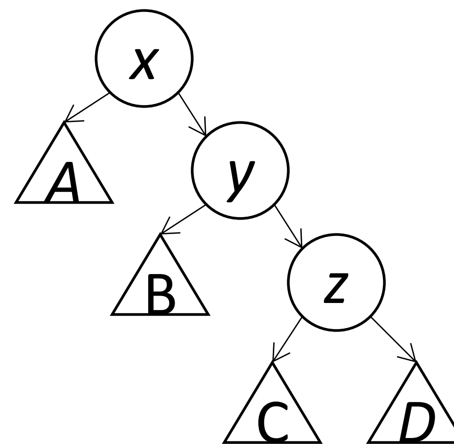
zig-zag



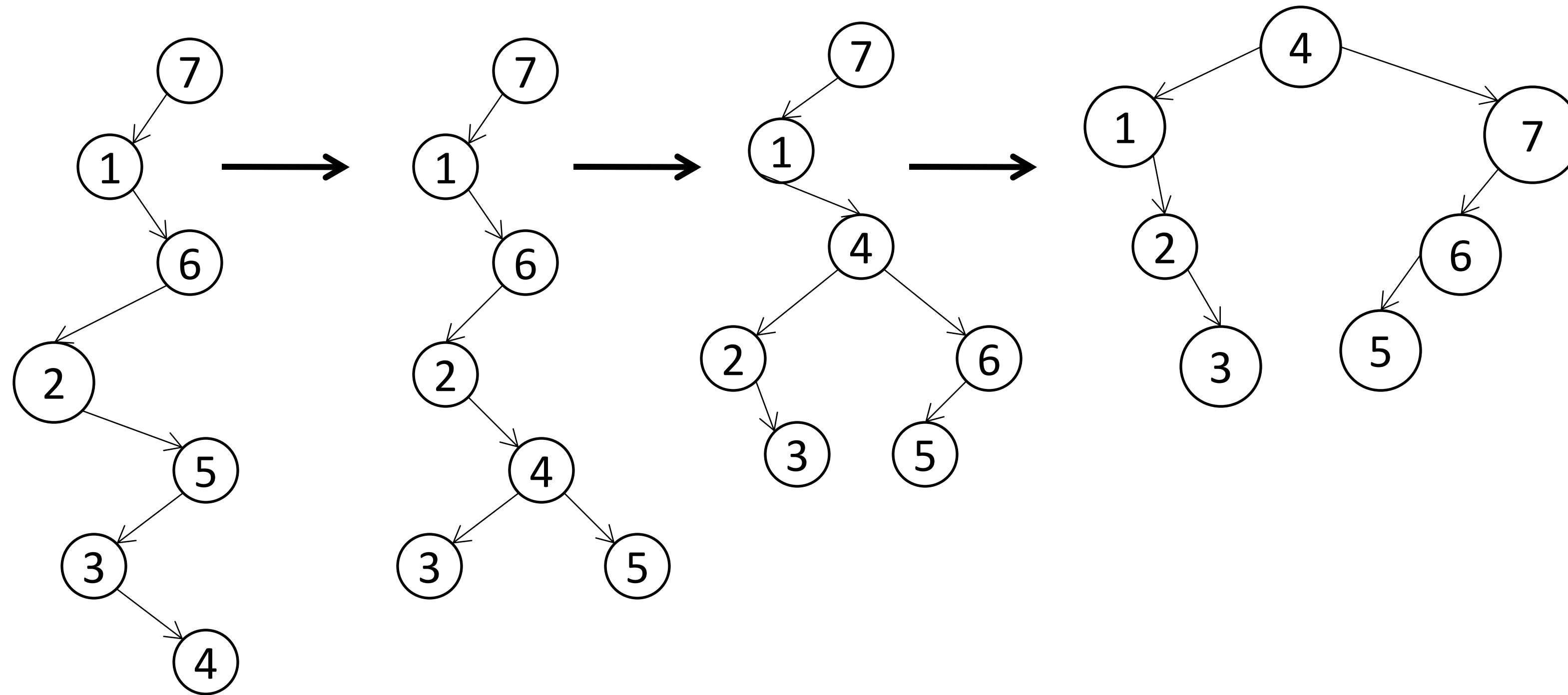


zig-zig

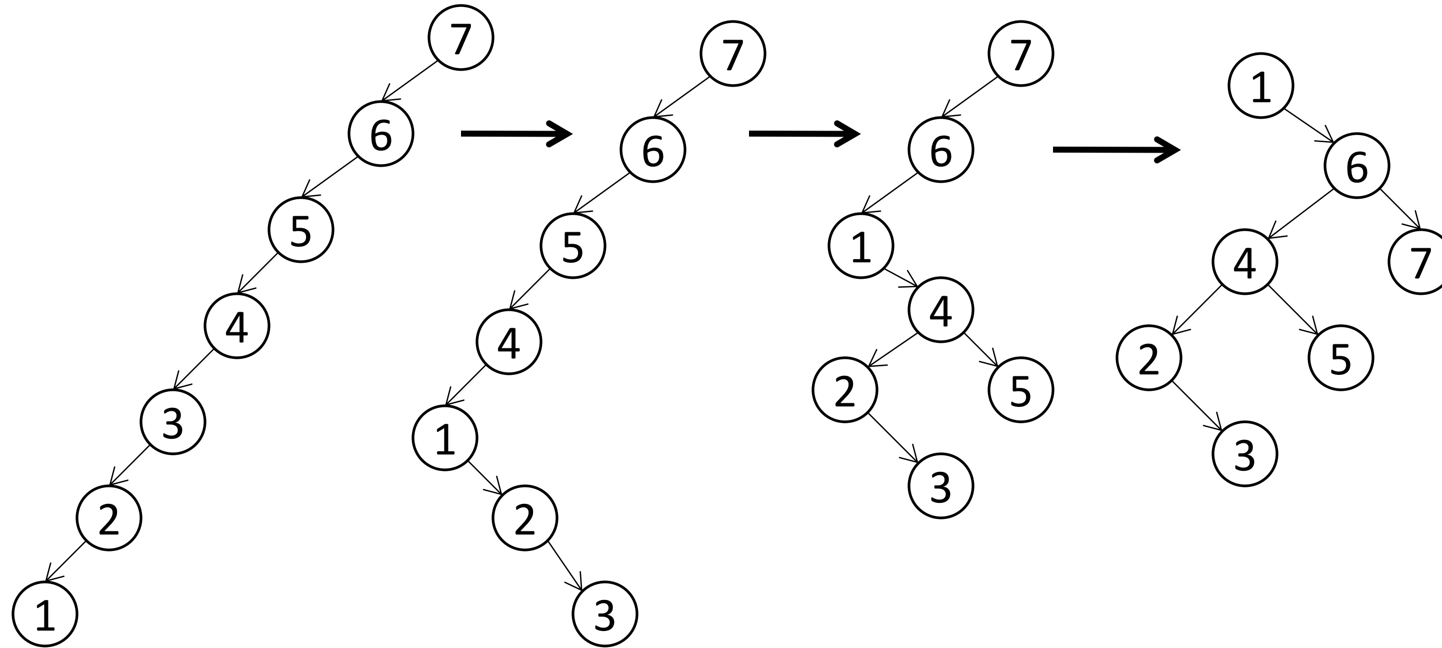




Splay: pure zig-zag



Splay: pure zig-zig



Demonstration

A great resource: "Data Structure Visualizations" by David Galles:

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

(Go to "Splay Trees")

Operations on splay trees

Access *x*: follow search path to *x*, then *splay*(*x*). Moves *x* to root.

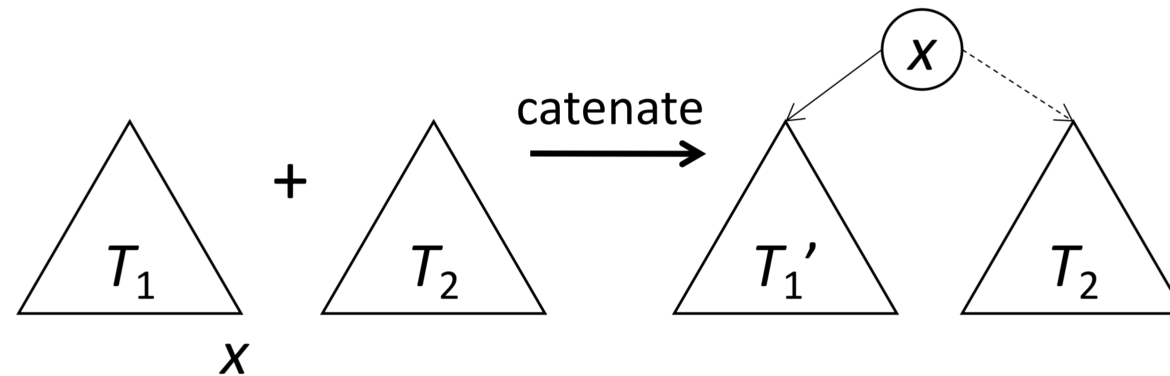
Insert *x*: follow search path to null, replace by *x*, *splay*(*x*).

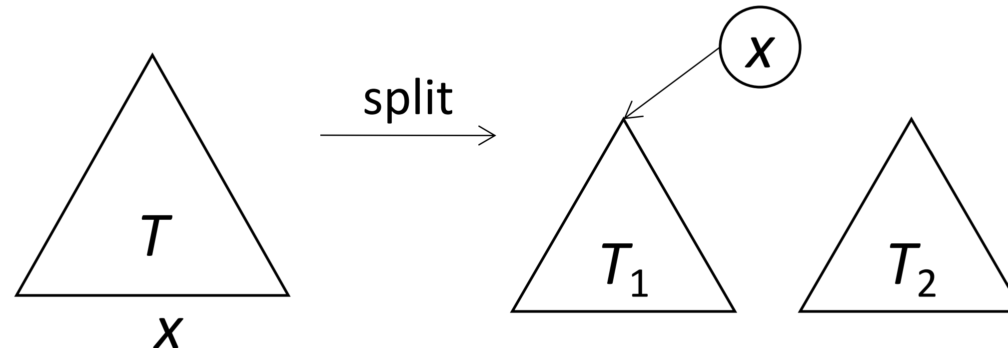
Delete *x*: follow search path to *x*, swap with successor if binary, delete *x*, splay at parent.

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* ← *root*(T_2).

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





Results

Balance Theorem: Given any initial tree of n nodes, performing m access operations takes $O((m + n)\log n)$ amortized time.

Balance Theorem with Insert/Delete: Starting from an empty tree, an arbitrary sequence of accesses, insertions, and deletions takes O(1 + log*n*) amortized time per operation, where *n* is the number of items in the current tree.

Static optimality theorem: Start from an arbitrary tree and do an arbitrary sequence of m accesses, with each item accessed at least once. Let $x.f$ = number of accesses of x, the amortized time to access x is $O(1 + \log(m/x.f))$.

→ competitive with static optimum tree for

given access frequencies

Working Set Theorem: Start with an arbitrary tree and do an arbitrary sequence of accesses, with each item accessed at least once. The amortized time to access x is $O(1 + \log(x.k))$, where $x.k$ is the number of distinct items accessed since the last time x was accessed.

Dynamic Finger Theorem (Cole et al. 1990): Start from an empty tree and do an arbitrary sequence of insertions, deletions, and accesses. The amortized time for an operation is $O(1 + \log t)$, where t is the number of nodes in symmetric order between the last node splayed and the current node splayed, inclusive.

True but the proof is long and complicated

Just how good is splaying?

Dynamic optimality conjecture: Given an initial tree and any access sequence, splaying does as well (to within a constant factor) 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)

Advantages of splay trees:

No balance information required.

Simple operations.

Take advantage of any exploitable pattern in the access sequence.

Disadvantage of splay trees:

Many rotations, even during accesses!

Reference

R. E. Tarjan, Data Structures and Network Algorithms, SIAM, 1983.

Thanks!