Self-adjusting Data Structures

Robert E. Tarjan, Princeton University

Paris, 15 June 2022

What is a data structure?

A way to store information so that queries and updates are fast

What is the best structure for the required operations?

Simple operations, but many of them

Dictionary

Store a set of items (keys), each with some data

Operations:

access(x): Find key x and return its data

insert(x): Insert key x with its data

delete(x): Delete key x and its data

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 allow “out of balance” structures, as long as they are “in balance” often enough.

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?

Two examples

Self-adjusting search trees

Self-adjusting heaps

Dictionary

Store a set of items (keys), each with some data

Operations:

access(x): Find key x and return its data

insert(x): Insert key x with its data

delete(x): Delete key x and its data

Dictionary data structures

List accessed by sequential search

Direct-access array: array indices are keys

Hash table: array indices are functions of the keys

Array ordered by key, accessed by binary search

Search tree, accessed by binary search

Binary search

Universe of keys is totally ordered, allowing binary comparisons

Binary search: Store *keys* in S in sorted order

To access x in S:

If S empty, stop (x not in S).

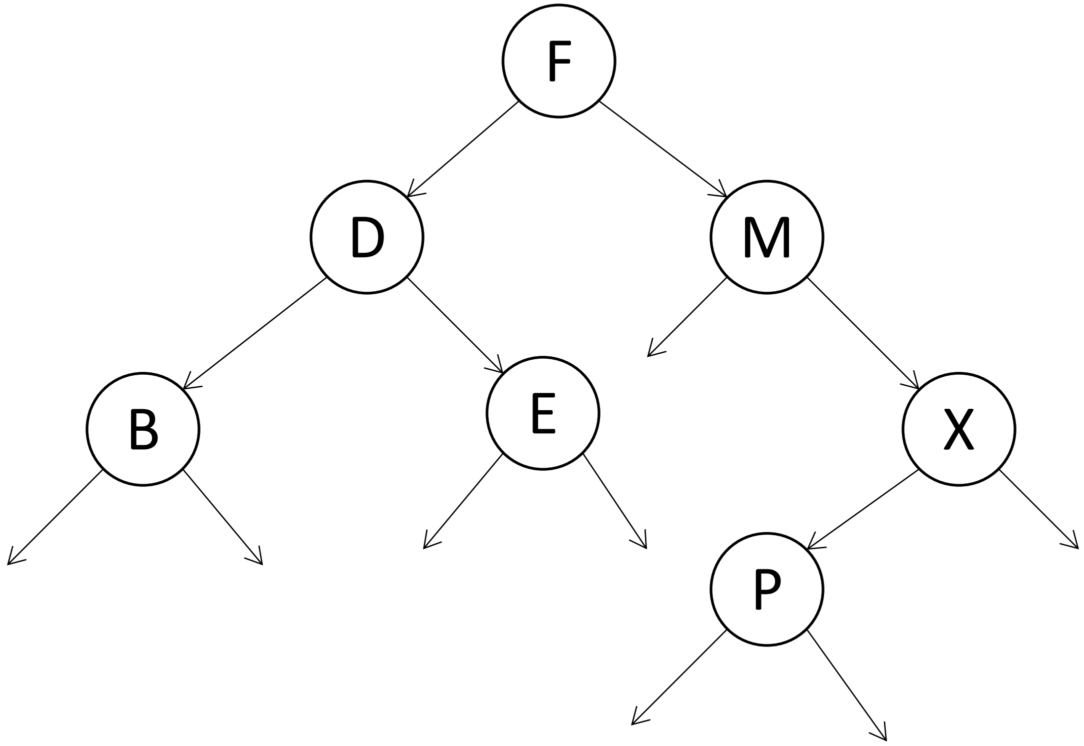
If S non-empty, compare x to some item y in S.

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

If $x < y$, search in $\{z \text{ in } S \mid z < y\}$.

If $x > y$, search in $\{z \text{ in } S \mid z > y\}$.

Binary search tree



Binary tree

Each node has a left child and a right child, either or both of which can be missing (null)

Each except one, the root, is a child of exactly one node, its parent

Each node has pointers to its children

Left (right) subtree of a node contains all nodes reachable from its left (right) child

Binary search tree

Each node holds a key and its data

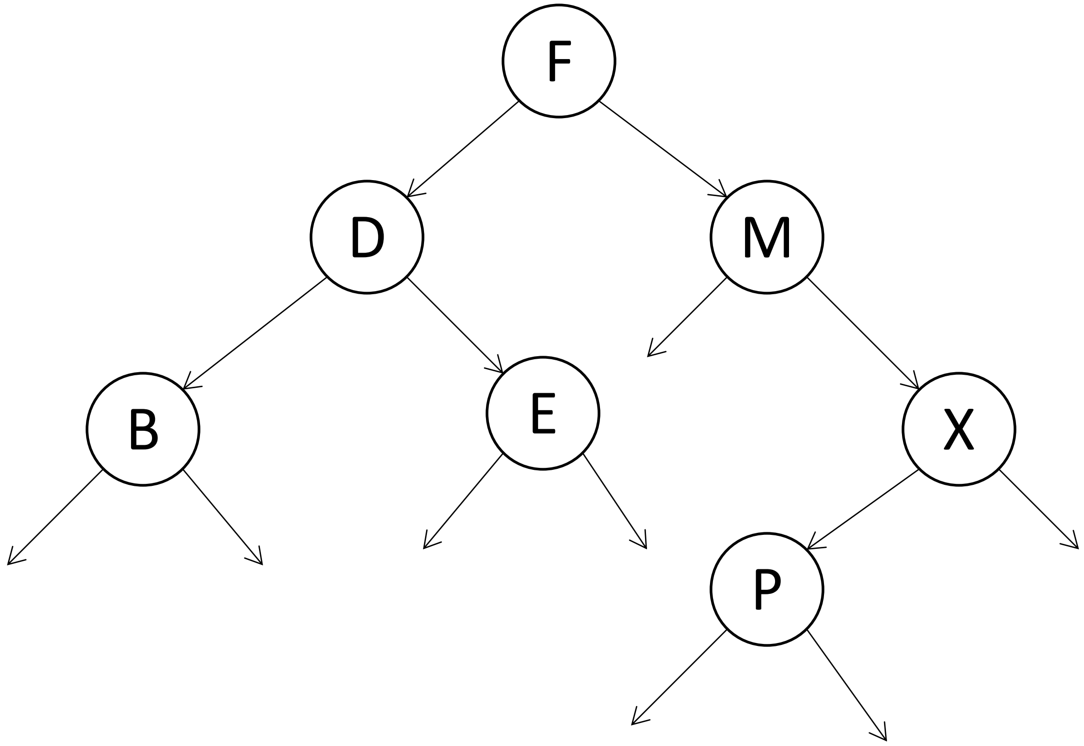
Keys are in symmetric order: keys in left (right) subtree of *x* are less than (greater than) *x*

Access is by binary search from the root

To access a key takes O(*d*) time, where *d* = number of nodes on path from root

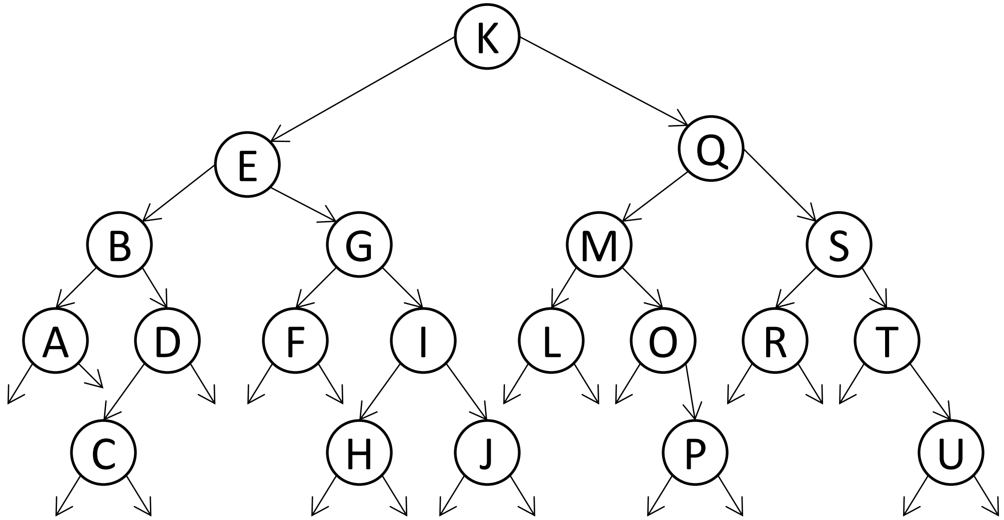
To make accesses fast, make paths short

Binary search tree



Best case

All leaves have depths within 1: depth $\lfloor \lg n \rfloor$. (*lg*: base-two logarithm)



Can achieve if tree is static (or insertion order is known off-line)

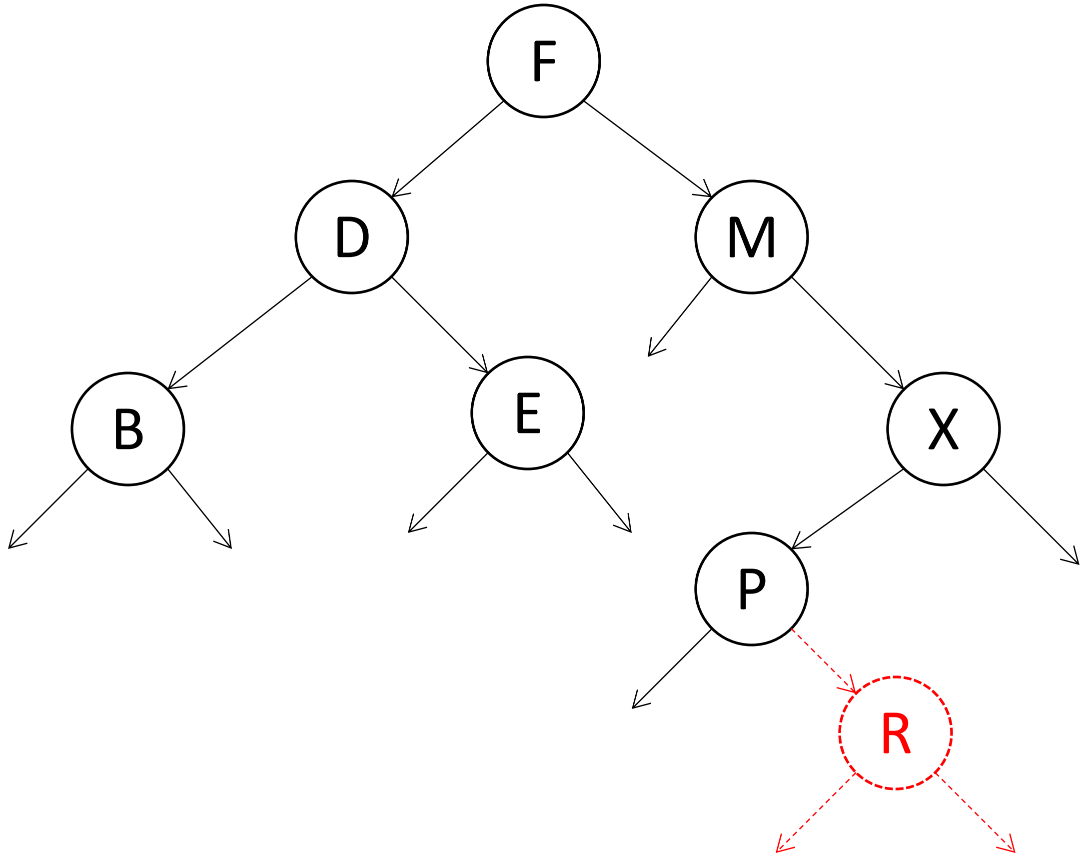
How to do inserts, deletes?

First, find any method, then refine or modify it to make it fast

Leaf insertion: follow the access path, insert key in a new node attached where the search falls off the bottom of the tree

Leaf Insertion

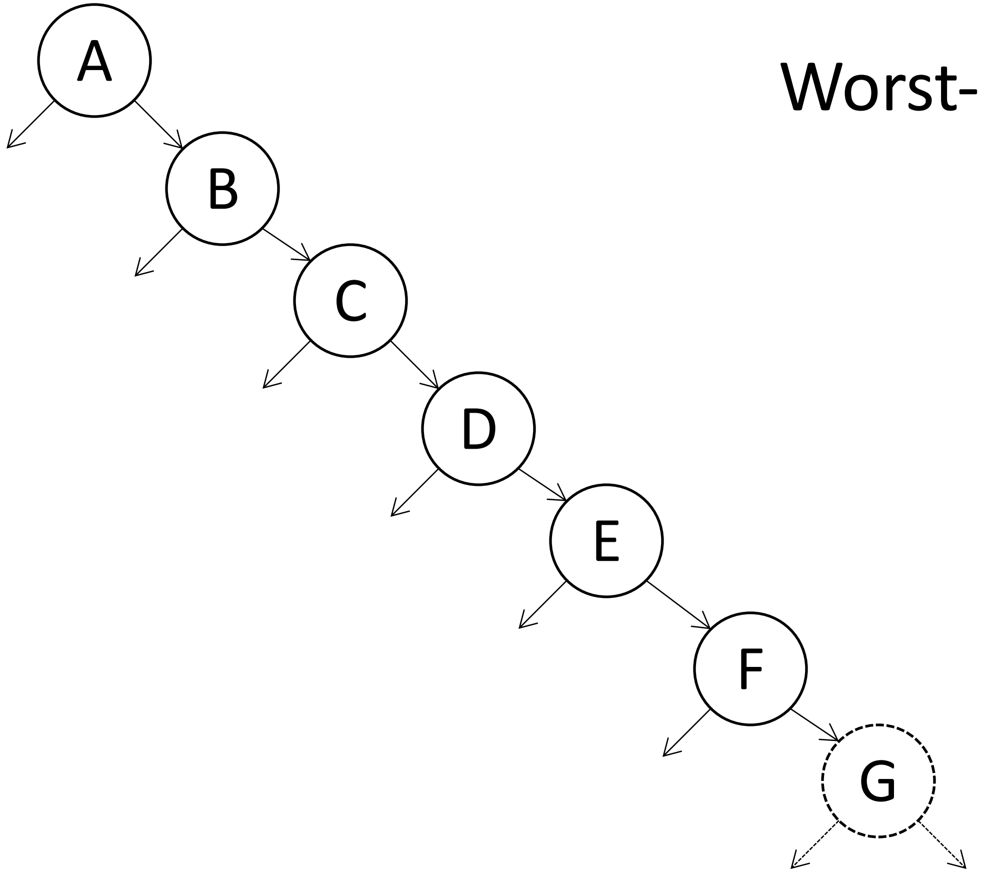
Insert R



Worst case

A natural but bad insertion order: sorted.

Insert A, B, C, D, E, F, G,…



Worst-case access cost is *n*.

= list!

Classic solution: keep the tree *balanced*

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

AVL trees: Adelson-Velsky, Landis 1971

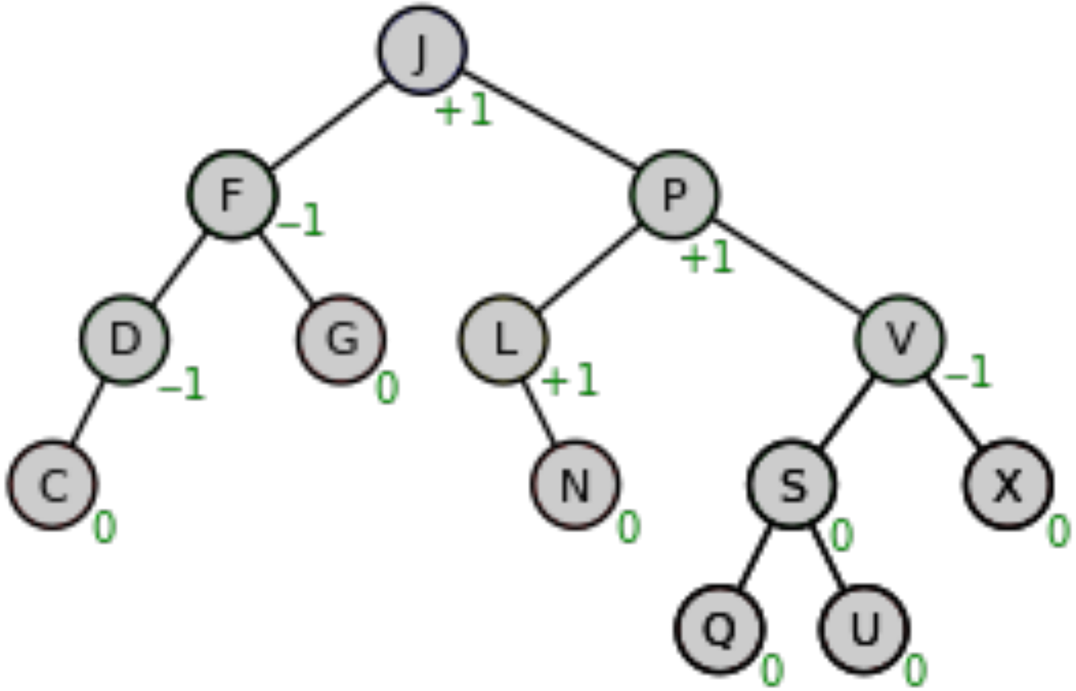
red-black trees: Bayer 1972, Guibas and Sedgewick 1978

MANY others…

We need:

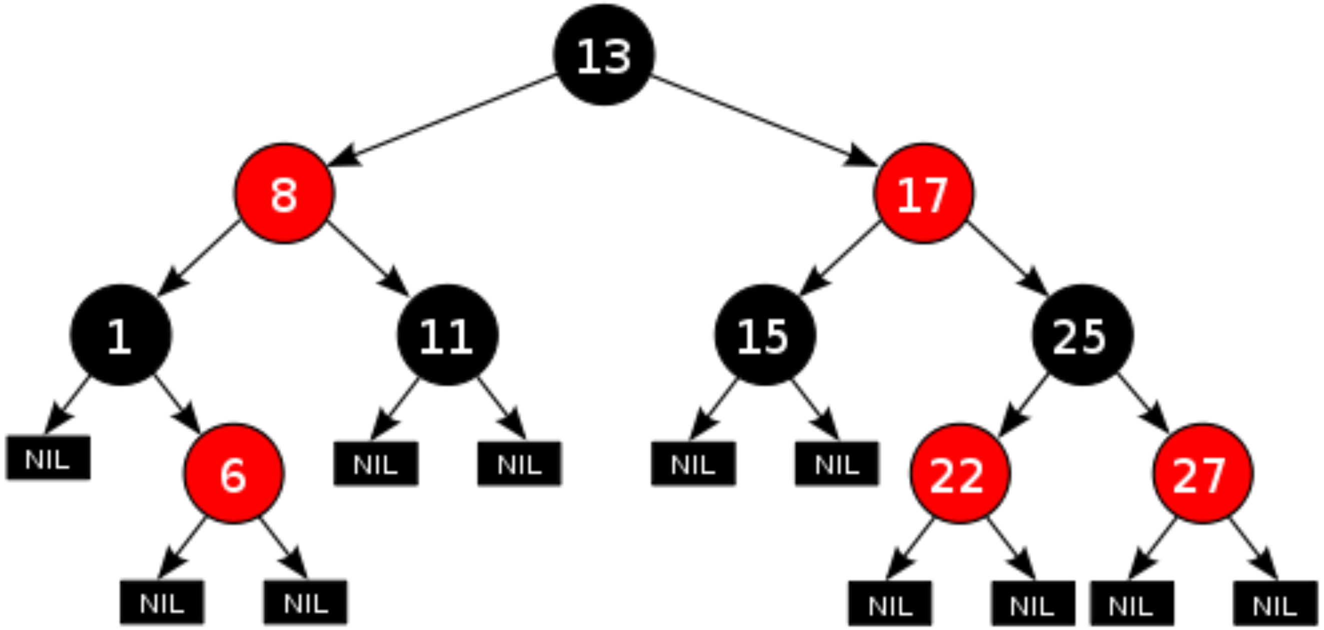
- A balance condition
- A way to restructure the tree during an update to maintain balance

An AVL tree
(image from Wikipedia)

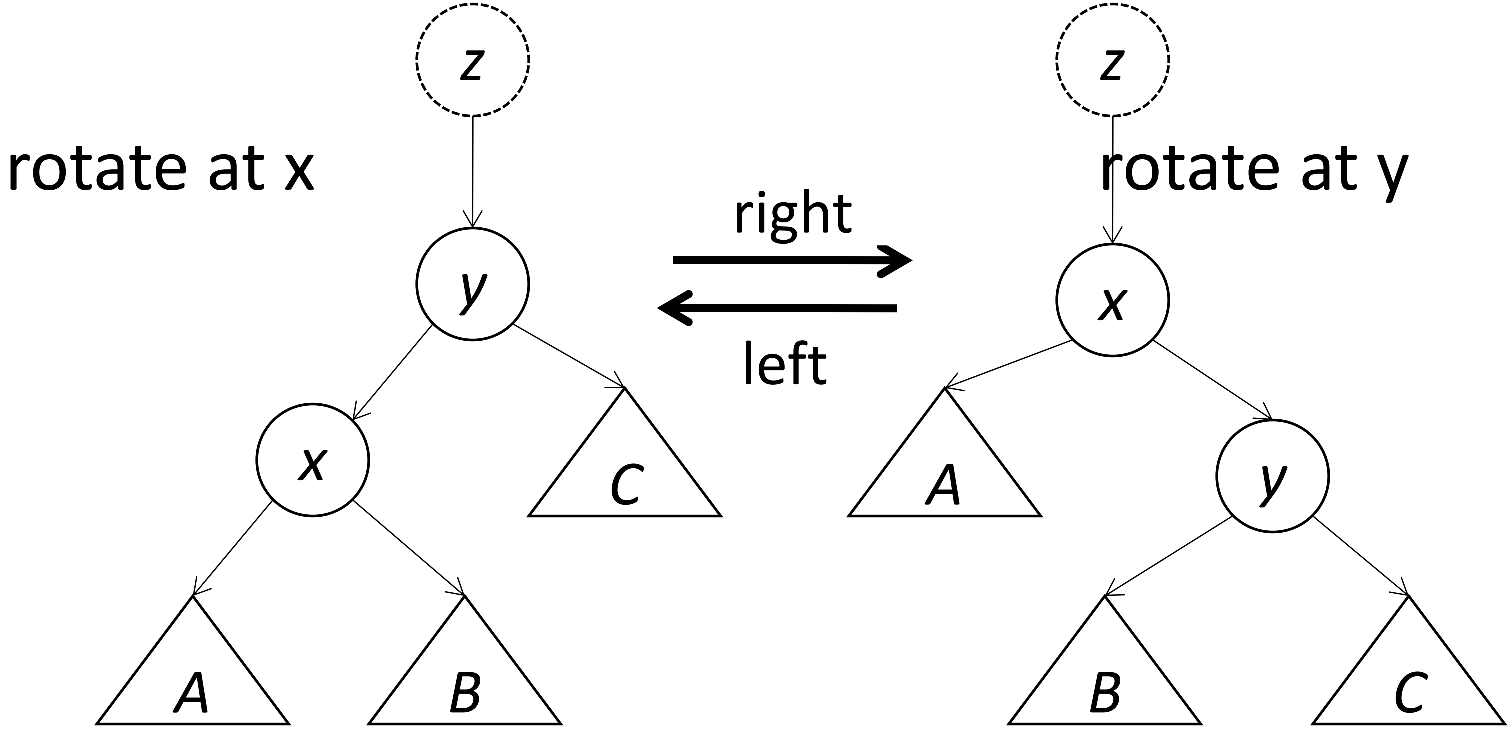


A red-black tree

(image from Wikipedia)



Restructuring primitive: rotation



Rebalancing

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

AVL tree insertion: rebalance bottom-up on access path after insertion

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 data structure

During each operation, including accesses, restructure to make future operations faster

Measure speed by total time of all operations

not worst-case time per operation

Goal: small amortized time = worst-case total time/#operations

Can we design such data structures?

Can we prove that they are fast?

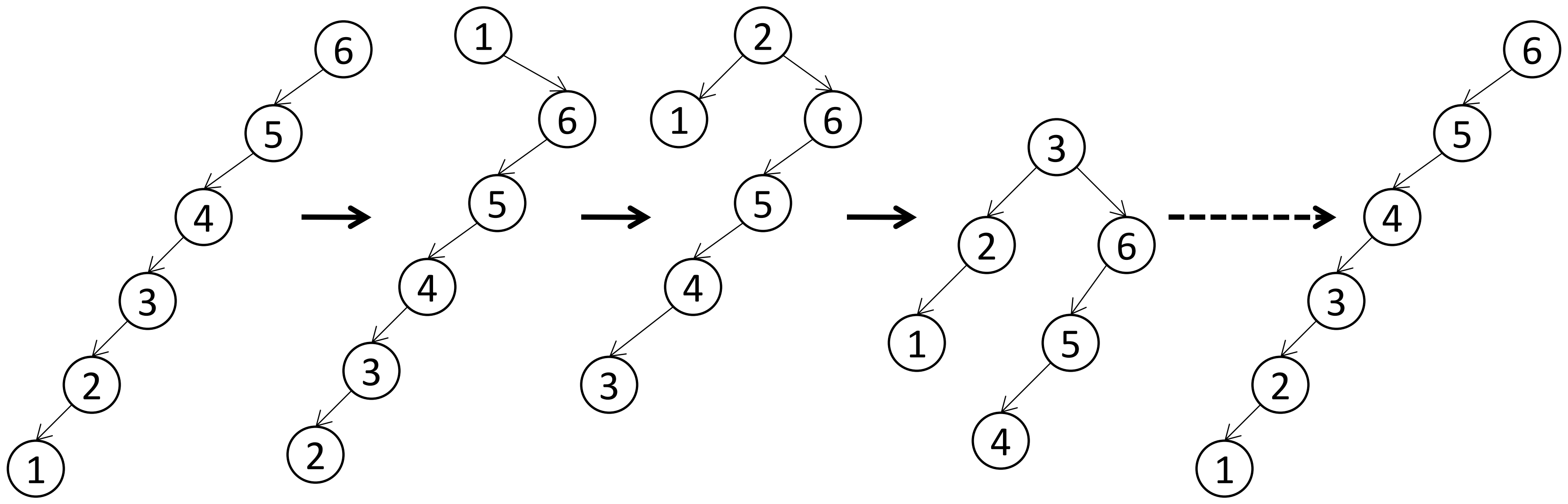
Self-adjusting binary search tree

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** $x.p \neq$ 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**

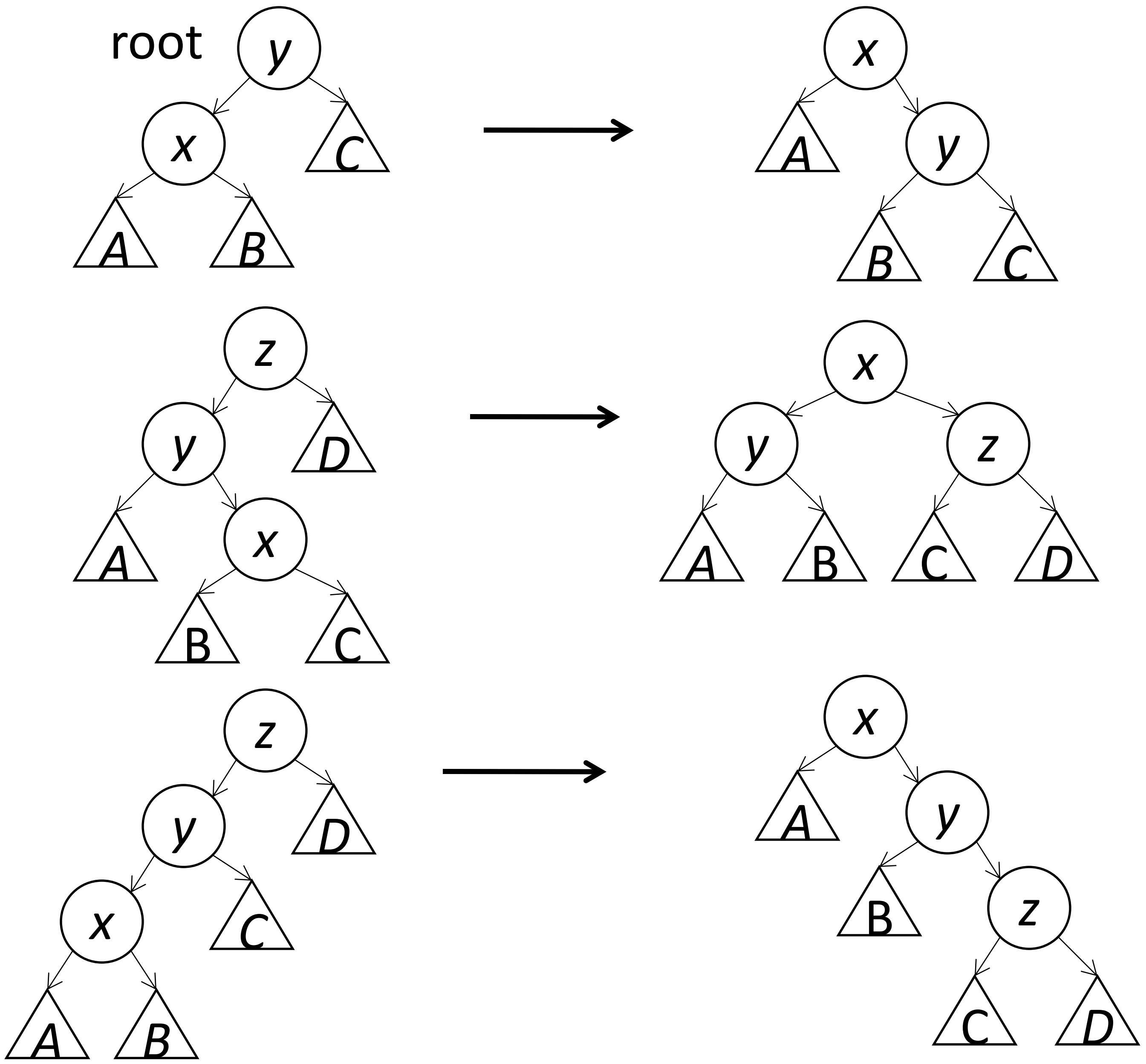
    $x.p$ is *left* then {*rotate*(x), *rotate*(x)} *zig-zag*

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

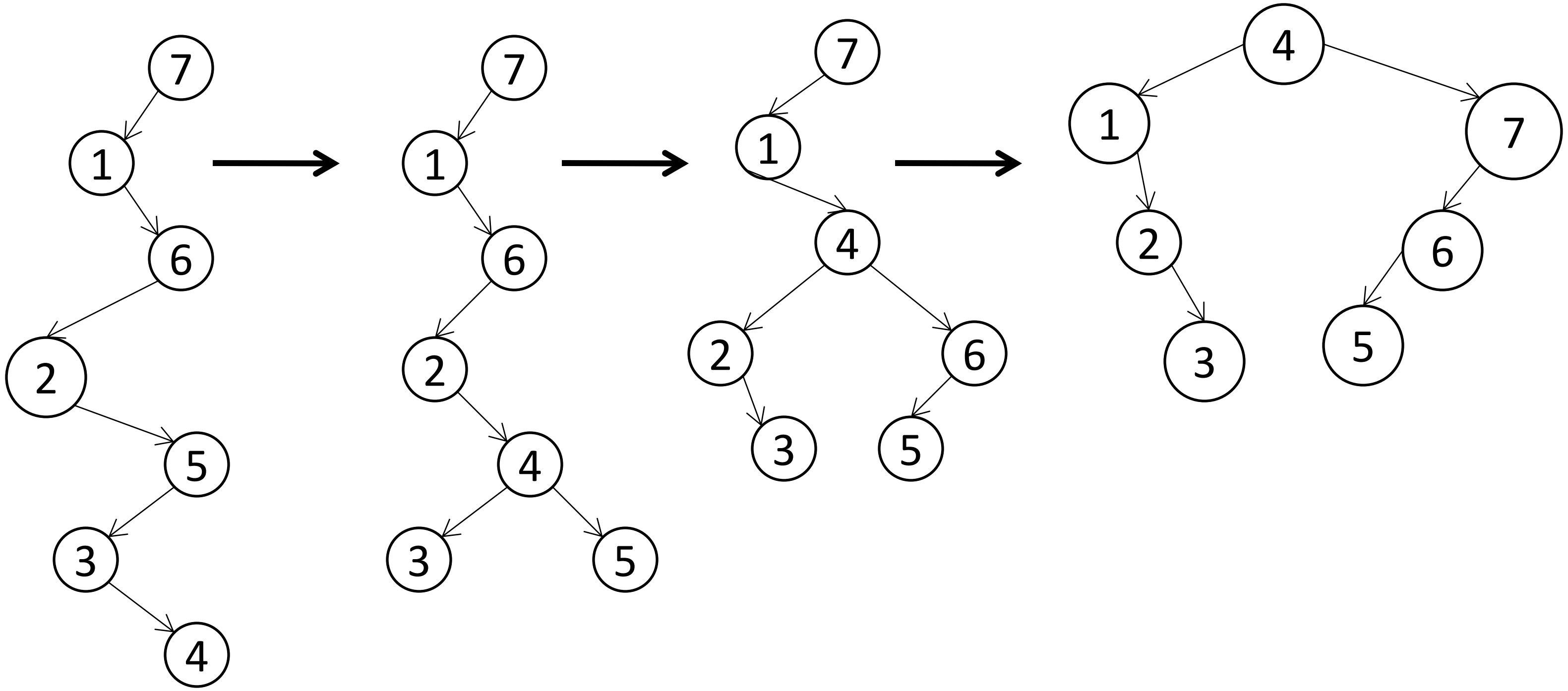
zig

zig-zag

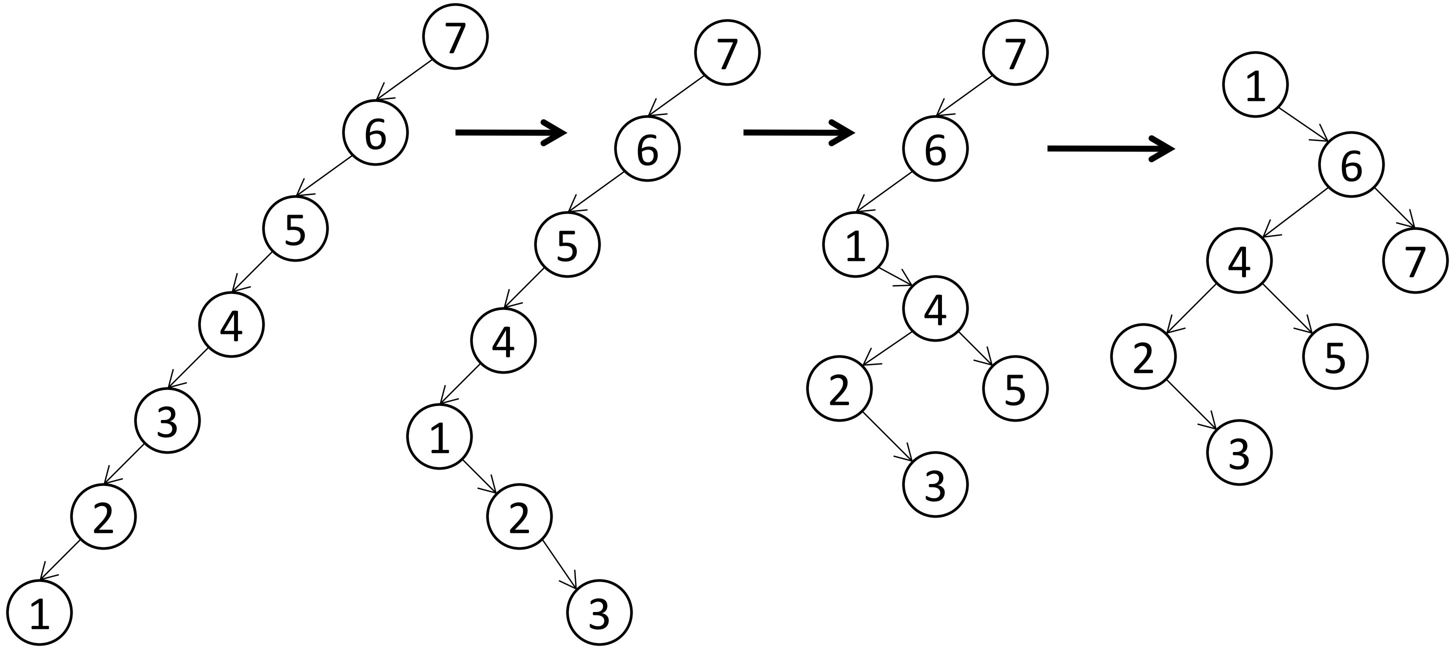
zig-zig



Splay: pure zig-zag



Splay: pure zig-zig



Depth halving

If y is on the path to x, splay(x) roughly halves the depth of y

No node increases in depth by more than two

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*)

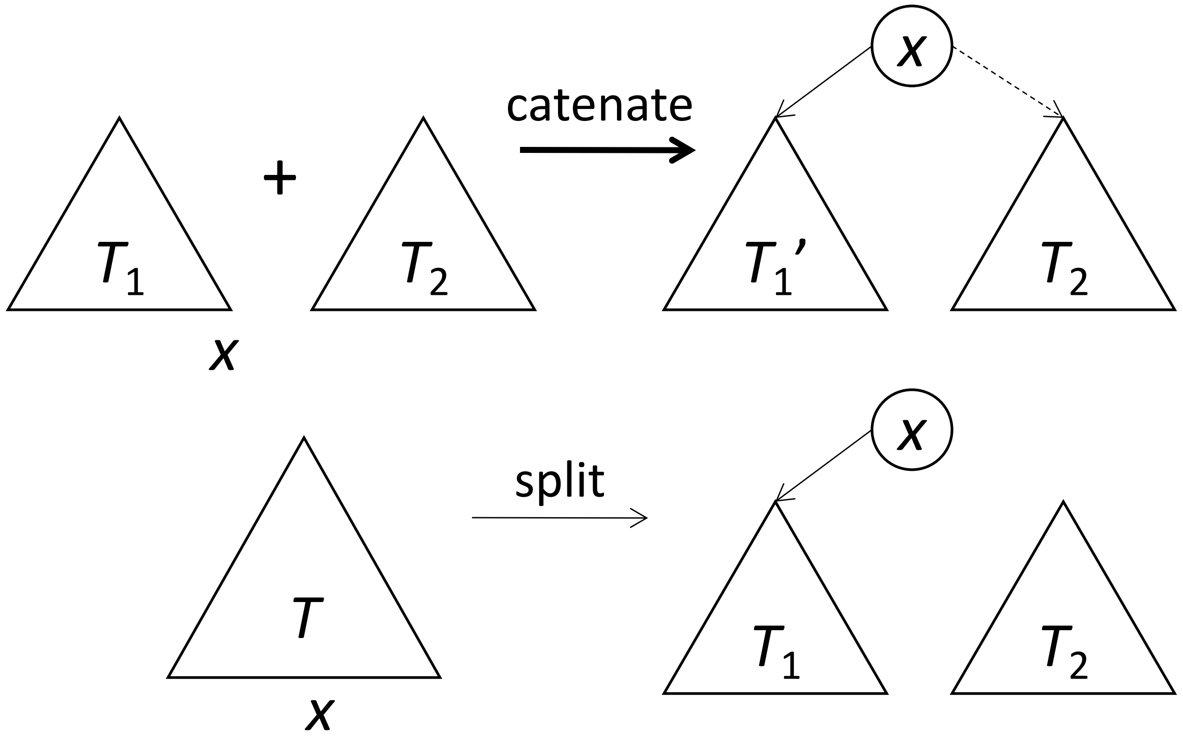
Delete *x*: follow access path to *x*, swap with successor if *x* is in a node with two children, delete *x*, splay at old parent of *x*

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)

Why think the conjecture is true?

Any optimum algorithm is monotone: deleting any access in the access sequence does not increase the total cost

Splay is monotone to within a constant factor if and only if the conjecture is true: Levy and T 2019

Monotone to within a constant factor: deleting any subset of accesses in a sequence increases the cost by at most a constant factor

Proof idea (one direction)

Given an optimum algorithm *A* for a given sequence *S*, one can simulate the behavior of *A* on *S* applying splaying to a super-sequence *S'* of *S*

Each splay does O(1) rotations

$|S'| = O(|S|)$

New goal

Prove that splay is monotone to within a constant factor

Heap (priority queue)

Store a set of items, each with a numeric key

Operations:

insert(x, H): Insert item x with its key into heap H

delete-min(H): Delete and return an item with minimum key in heap H

Heap with *decrease-key*

Store a set of items, each with a numeric key

Operations:

make-heap(): Return an empty heap

insert(x, H): Insert item x with its key into heap H

delete-min(H): Delete and return an item with minimum key in heap H

decrease-key(x, k, H): Replace by k the key of item x in heap H
k must be no larger than the key of x

Meldable heaps

Store a collection of item-disjoint heaps

Operations:

make-heap(): Return an empty heap

insert(x, H): Insert item x with its key into heap H

delete-min(H): Delete and return an item with minimum key in heap H

decrease-key(x, k, H): Replace by k the key of item x in heap H
k must be no larger than the key of x

meld(H_1, H_2): Combine item-disjoint heaps H_1 and H_2 into a single heap, and return it

Notes

decrease-key(x, H) is given a pointer to the location of item x in heap H

n inserts followed by n delete-mins will sort n items by key, so any binary-comparison-based method requires $\Omega(n\log n)$ time for n operations

Applications

Priority-based scheduling and allocation

Discrete event simulation

Network optimization:

- Shortest paths
- Minimum spanning trees
- Maximum weight matching

Dijkstra’s shortest path algorithm

Single source, non-negative arc lengths

Use a heap whose items are vertices, with key equal to length of shortest path found so far

n inserts, n delete-mins, m decrease-keys n = #vertices m = #arcs

If O(log$n$) time per operation, total time is O($m$log$n$)

Can we do better?

O(1) decrease-key would give O(m + $n$log$n$) for Dijkstra’s algorithm

Our goal

O(lg*n*) amortized time for *delete-min* and *delete*

O(1) amortized time, or at least o(log*n*), for all other operations, including meld

Heap as a binary search tree

Need parent pointers for *decrease-key*, *delete*

Do a *decrease-key* as a *delete* followed by an *insert*

All operations except *meld* take O(log*n*) time, worst-case if tree is balanced, amortized if self-adjusting (splay tree)

Binary search tree is too rigid

We need a more flexible structure

Alternative: heap-ordered tree

Heap order: $x.p.k \leq x.k$ for all items x.

$x.k$ = key of x, $x.p$ = parent of x

Heap order is defined for all rooted trees, not just binary trees: nodes can have any number of children

Heap order → item in root has min key

→ *find-min* takes O(1) time

What tree structure? How to implement heap operations?

Heap-ordered tree of non-constant degree

link: combine two trees by comparing the keys of their roots, making the root with smaller key the parent of the other

This increases the degree of the new root, hence non-constant degree

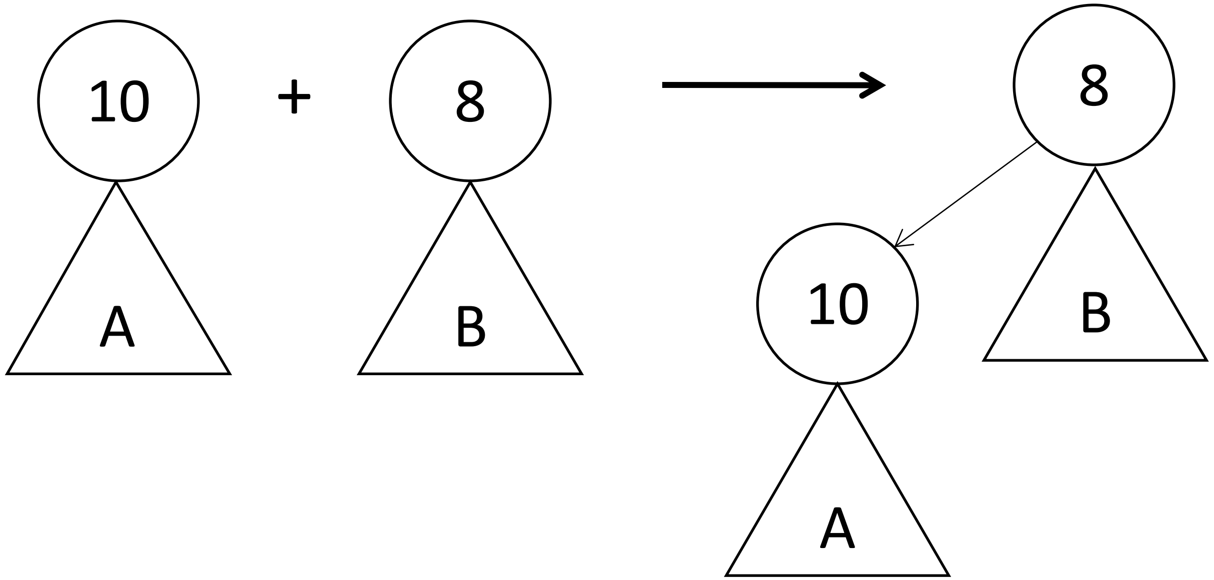
The new root is the *winner* of the link, the new child is the *loser* of the link.

We will build all operations out of links and cuts (breaking links)

A link

One comparison, O(1) time

8 is the winner, 10 is the loser



Heap operations

find-min: return item in root

make-heap: return a new, empty tree

insert: create a new, one-node tree, link with existing tree

meld: link two trees

decrease-key: change key, break link with parent, link with root *delete-min*: Delete root, link trees rooted at its children.

Time is proportional to number of children: need to link in a way that keeps #children small.

How?

Balanced heap

Only link trees whose roots have the same degree (#children)

Keeps each tree size logarithmic in root degree

Handling decrease-key requires careful tree pruning (or equivalent)

A heap is a set of heap-ordered trees, not just one (this can be fixed)

Must store node degrees: ranks

Fibonacci heaps and many related structures use these ideas, achieve desired bounds: $O(\log n)$ delete-min, $O(1)$ other operations

Self-adjusting heap

Do not store ranks

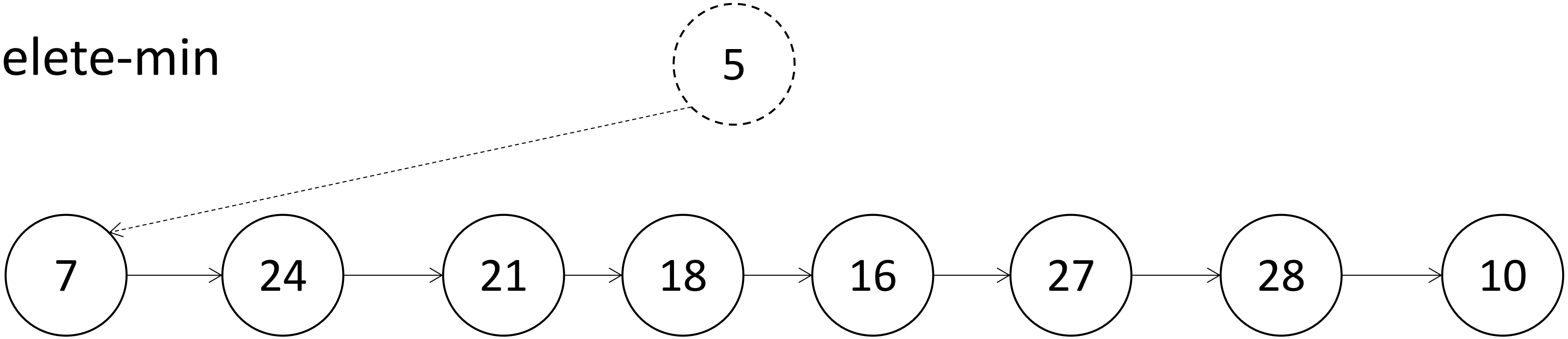
Do links during delete-min based on position in list of new roots

Pairing heap
Fredman, Sedgewick,Sleator, T 1986

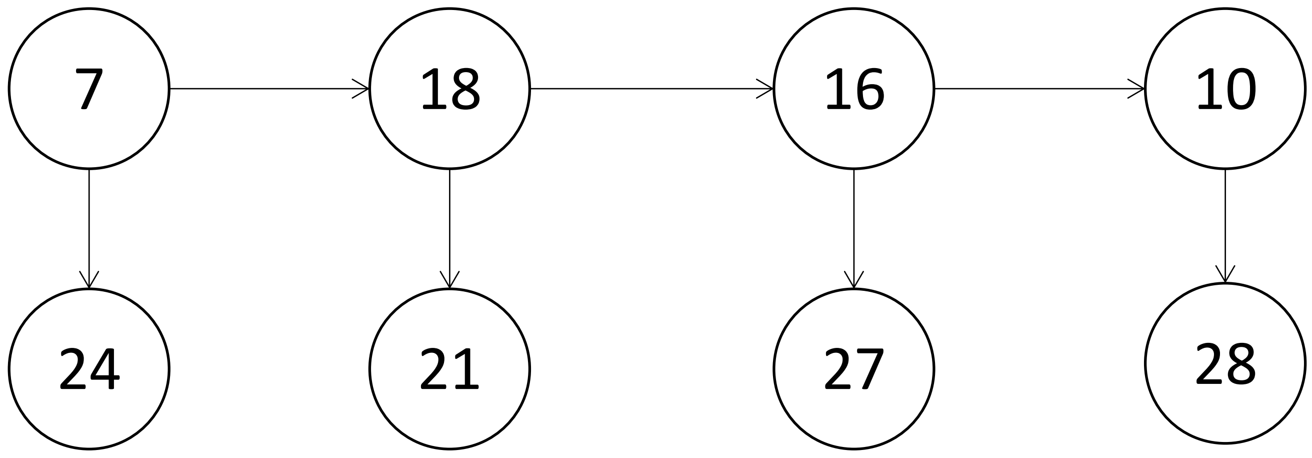
Delete-min: after deleting root, do two linking passes through the list of new roots

Pairing pass: link roots in adjacent pairs left-to-right (first to last)

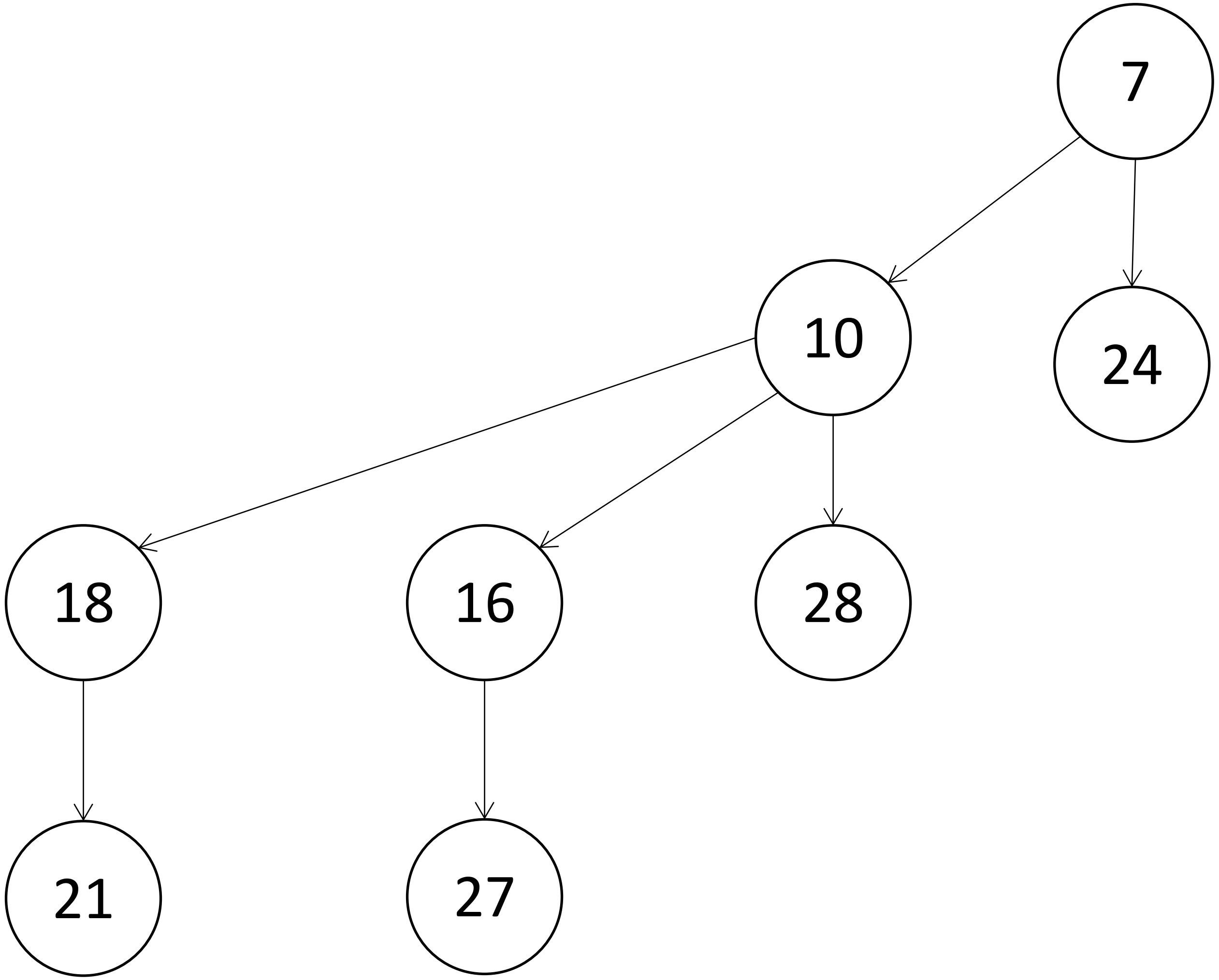
Assembly pass: Repeatedly link last two roots until only one remains delete-min



after pairing pass

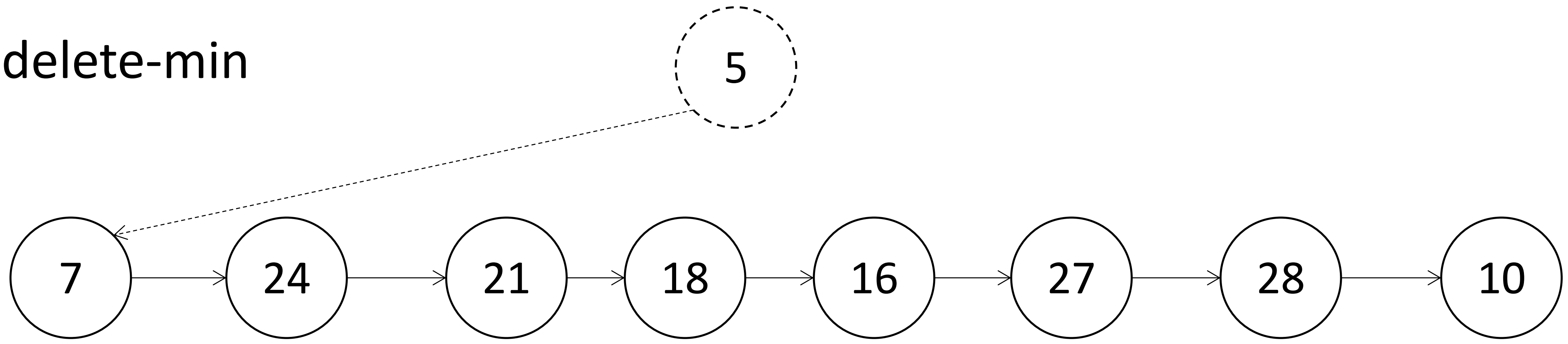


after assembly pass

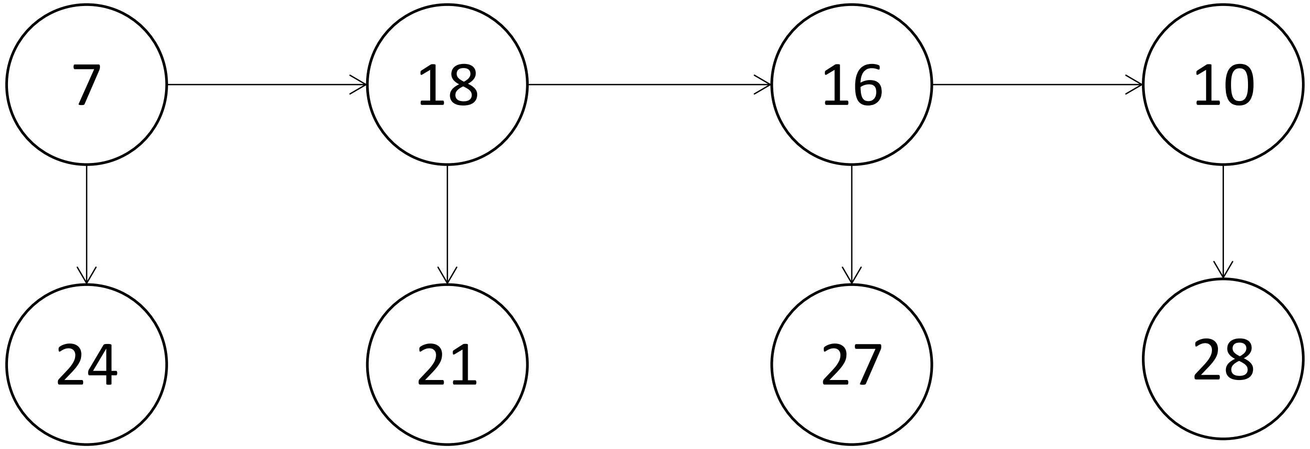


Multipass pairing heap

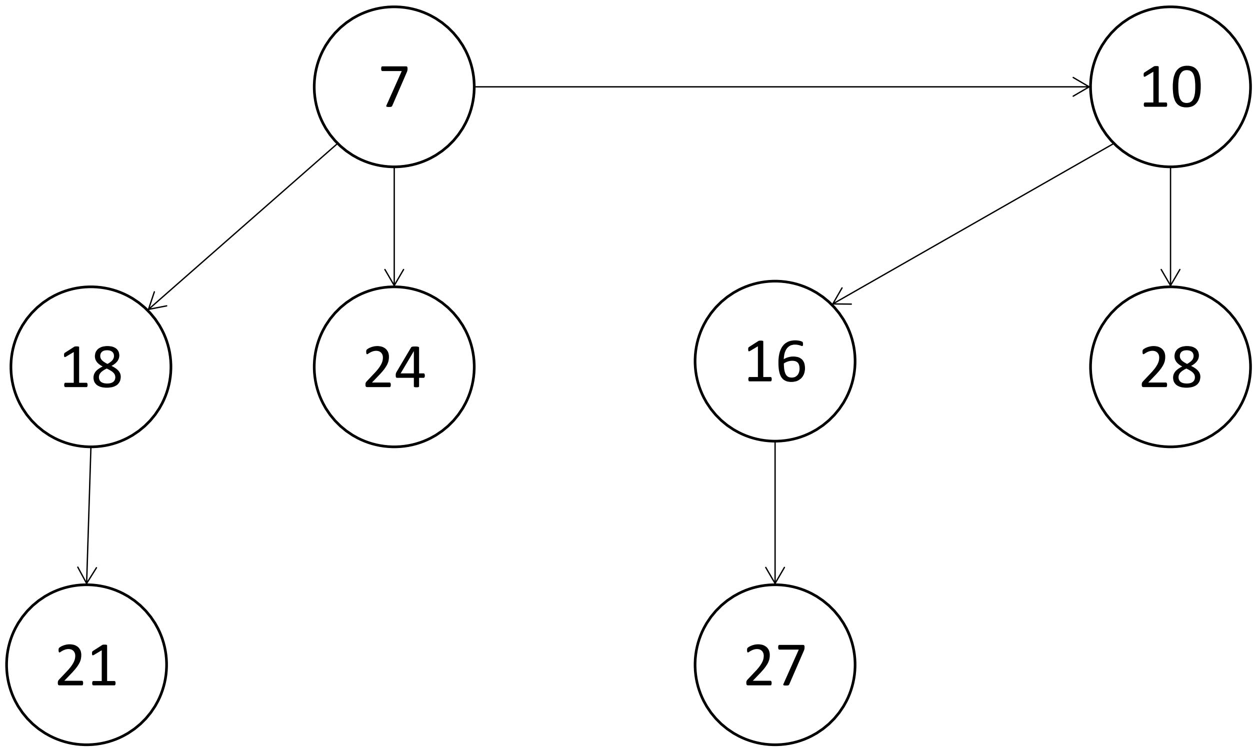
Delete-min: after deleting root, do pairing passes until one root remains delete-min



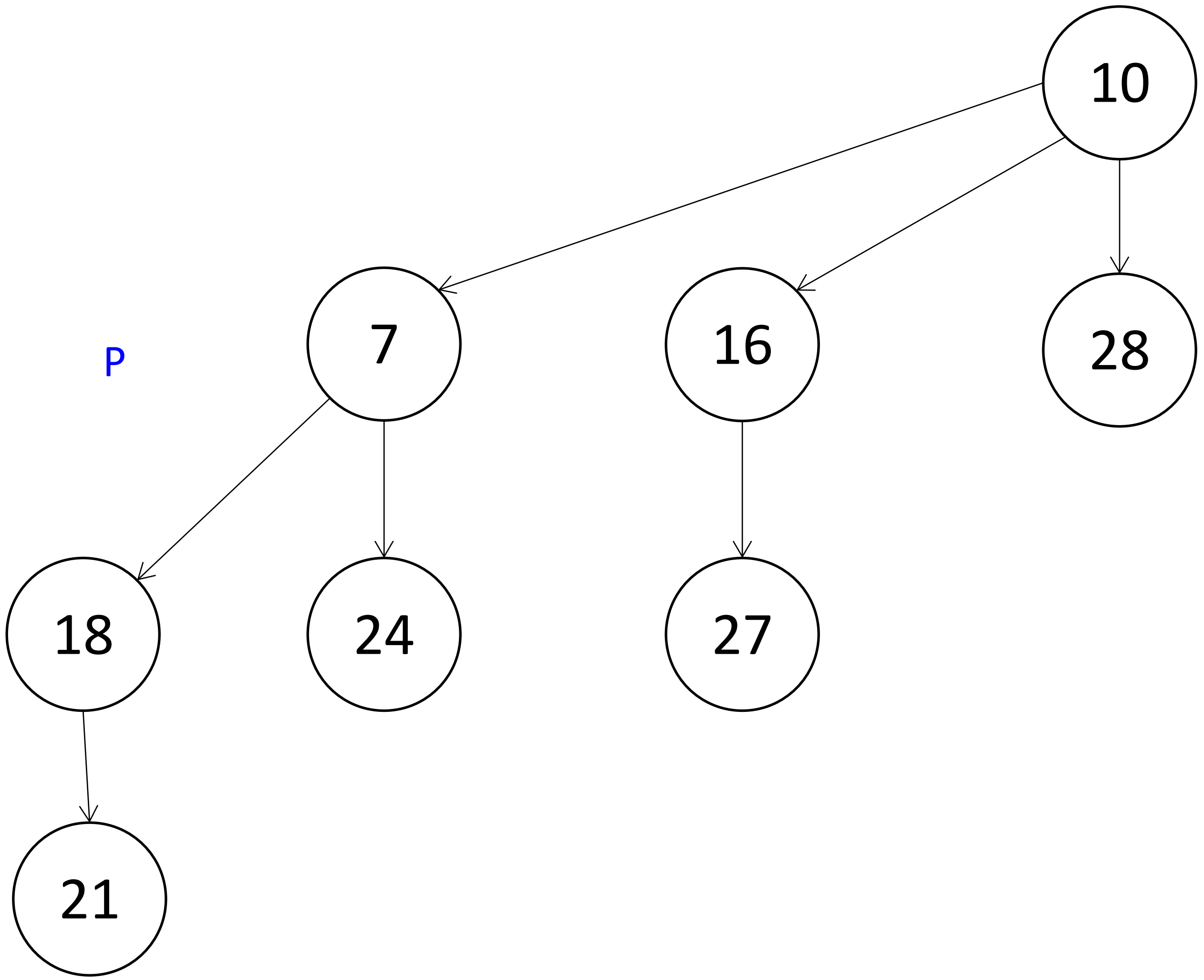
after first pairing pass



after second pairing pass



after third pairing pass



Why pairing?

The original analysis of splay trees applied to pairing heaps gives
O(log*n*) amortized time per heap operation

Pairing + assembly mimics repeated zig-zig on splay trees, if one ignores
the distinction between left and right children

Is O(log*n*) tight for decrease-key?

Lower bound

Any heap that stores no balance information needs $\Omega(\log\log n)$ time

for decrease-key Fredman 1999

Iacono and Ozkan 2014 have a similar result (with different restrictions)

New goal: $O(\log\log n)$ time for decrease-key in pairing heap or some other self-adjusting heap

New results Sinnamon (and T) 2022

New bounds for multipass pairing heaps: $O(\log n)$ time for delete-min

and delete, $O(\log\log n(\log\log\log n))$ for other operations

New bounds for slim and smooth heaps (other types of self-adjusting heaps):

$O(\log n)$ for delete-min and delete, $O(\log\log n)$ for other operations

In these data structures, can reduce insert and meld time to $O(1)$

with small changes to the data structure

What makes a self-adjusting heap fast?

During delete-min

Link adjacent roots in the list of roots

Try to minimize the number of new children of any node

Thanks!