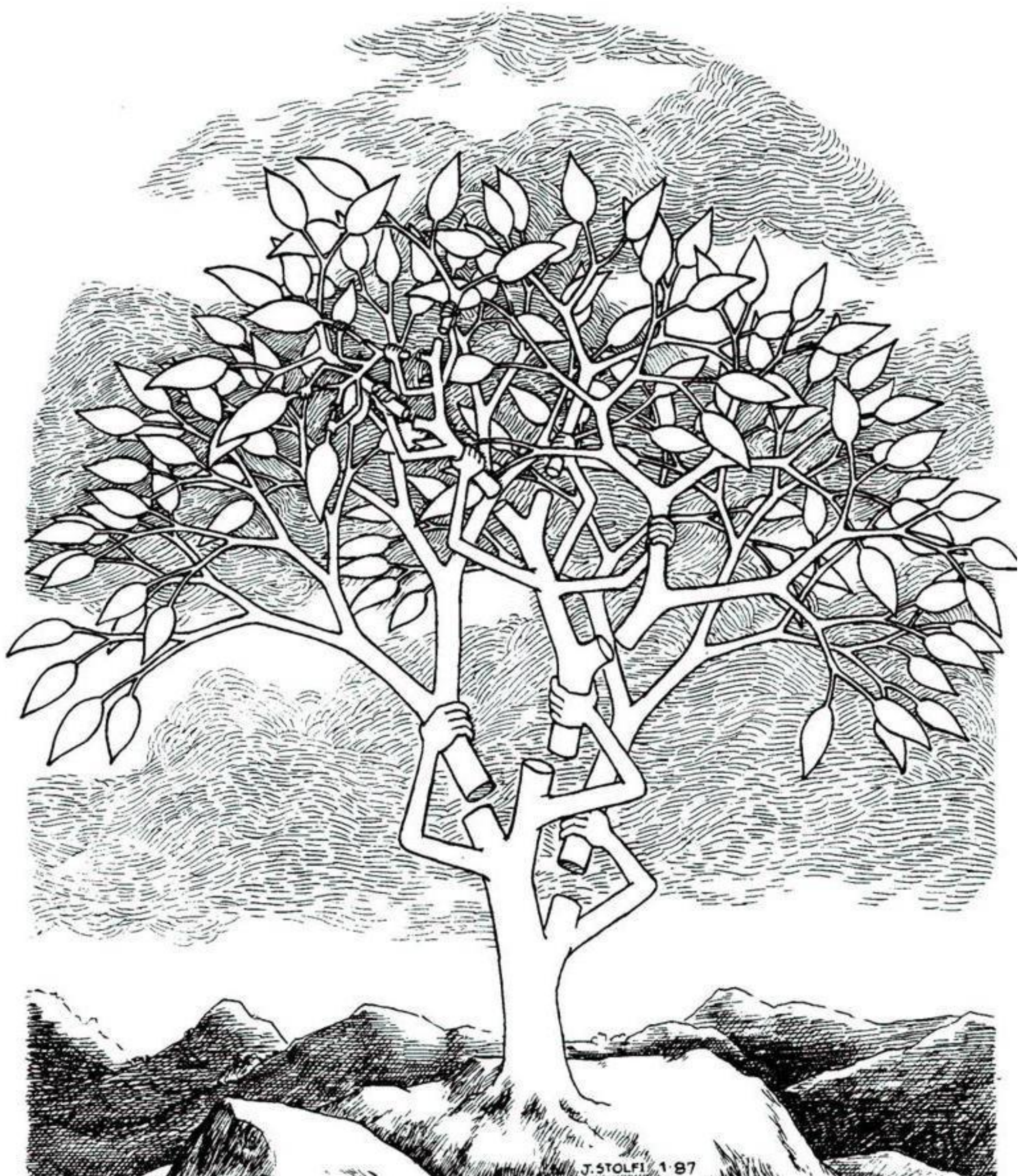
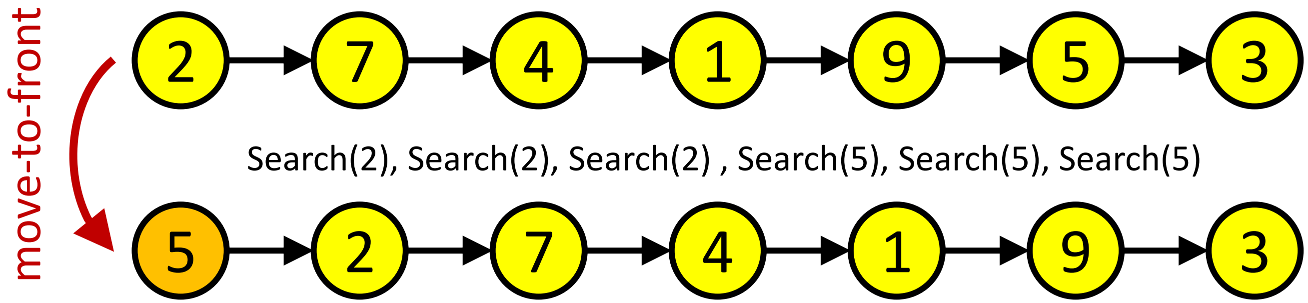
Self-Adjusting Data Structures



Self-Adjusting Data Structures



Lists
[D.D. Sleator, R.E. Tarjan, *Amortized Efficiency of List Update Rules*, Proc. 16th Annual ACM Symposium on Theory of Computing, 488-492, 1984]

Dictionaries
[D.D. Sleator, R.E. Tarjan, *Self-Adjusting Binary Search Trees*, Journal of the ACM, 32(3): 652-686, 1985]
→ splay trees

Priority Queues
[C.A. Crane, *Linear lists and priority queues as balanced binary trees*, PhD thesis, Stanford University, 1972]
[D.E. Knuth. *Searching and Sorting*, volume 3 of The Art of Computer Programming, Addison-Wesley, 1973]
→ leftist heaps

[D.D. Sleator, R.E. Tarjan, *Self-Adjusting Heaps*, SIAM Journal of Computing, 15(1): 52-69, 1986]
→ skew heaps

[C. Okasaki, *Alternatives to Two Classic Data Structures*, Symposium on Computer Science Education, 162-165, 2005]
→ maxiphobic heaps

[A. Gambin, A. Malinowski. *Randomized Meldable Priority Queues,* Proc. 25th Conference on Current Trends in Theory and Practice of Informatics: Theory and Practice of Informatics, 344-349, 1998]
→ randomized version of maxiphobic heaps

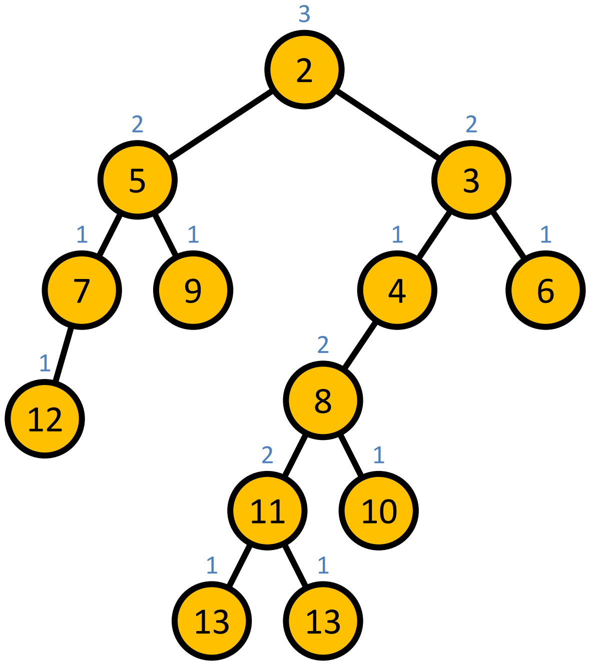
Okasaki: *maxiphobic heaps are an alternative to leftist heaps ... but without the "magic"*

Heaps (via Binary Heap-Ordered Trees)

MakeHeap, FindMin, Insert, **Meld**, DeleteMin

Insert = Meld

DeleteMin = Cut root + Meld



Leftist Heaps

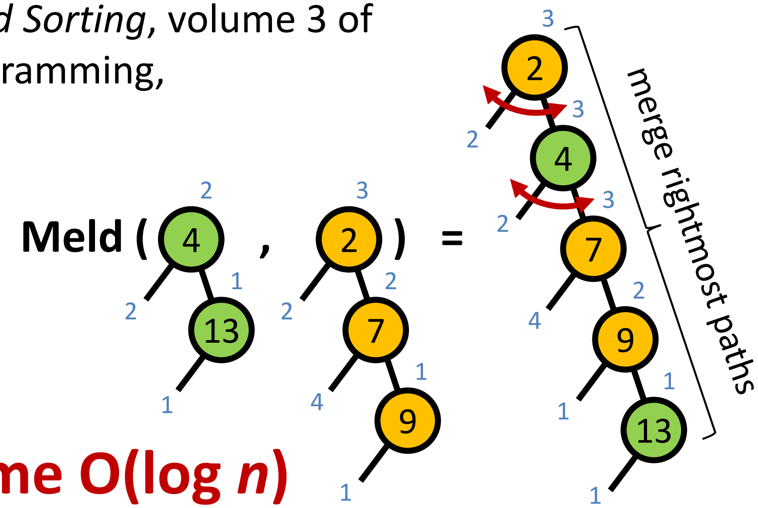
[C.A. Crane, *Linear lists and priority queues as balanced binary trees*, PhD thesis, Stanford University, 1972]

[D.E. Knuth. *Searching and Sorting*, volume 3 of The Art of Computer Programming, Addison-Wesley, 1973]

Each node **distance to empty leaf**

Inv. Distance right child $\leq$ left child

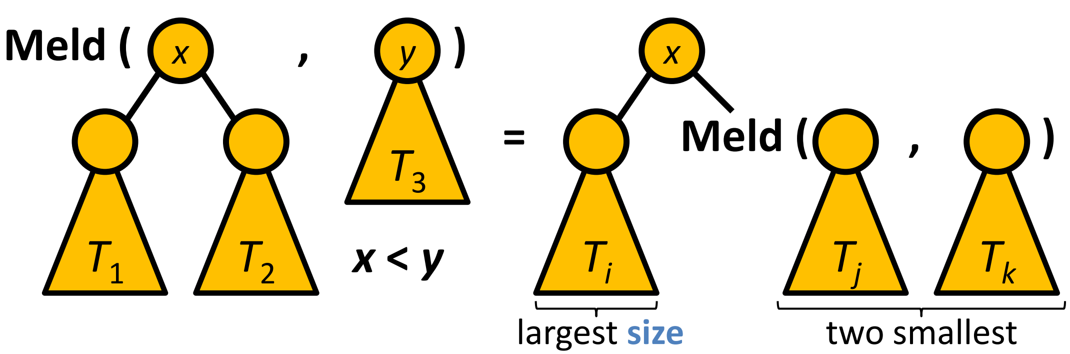
$\Rightarrow$ rightmost path $\leq \lceil \log n+1 \rceil$ nodes



Time O(log *n*)

Maxiphobic Heaps

[C. Okasaki, *Alternatives to Two Classic Data Structures*, Symposium on Computer Science Education, 162-165, 2005]

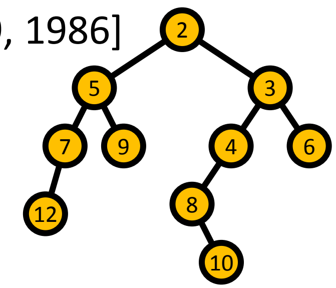


Max size $n \rightarrow \frac{2}{3}n$

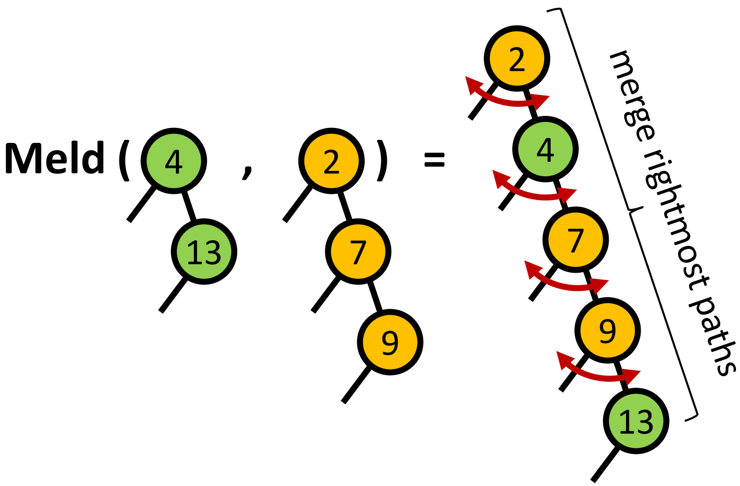
Time $O(\log_{3/2} n)$

Skew Heaps

[D.D. Sleator, R.E. Tarjan, *Self-Adjusting Heaps*, SIAM Journal of Computing, 15(1): 52-69, 1986]



- Heap ordered binary tree with ***no*** balance information
- MakeHeap, FindMin, Insert, **Meld**, DeleteMin
  (Insert = Meld, DeleteMin = Cut root + Meld)
- **Meld** = merge rightmost paths + swap ***all*** siblings on merge path



v **heavy** if $|T_v| > |T_{p(v)}|/2$, otherwise **light**

$\Rightarrow$ any path $\leq \log n$ **light** nodes

Potential Φ = # heavy right children in tree

O(log *n*) amortized Meld

Heavy right **child** on merge path before meld $\rightarrow$ replaced by **light child**

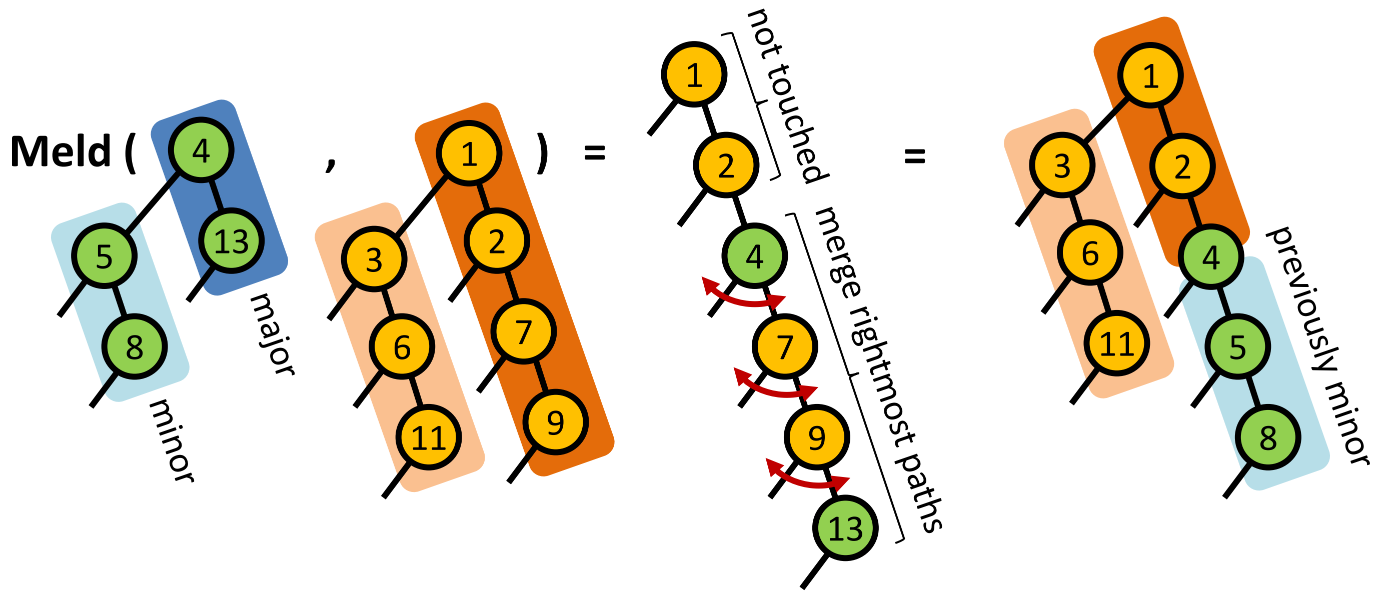
$\Rightarrow$ 1 potential released for heavy child

$\Rightarrow$ amortized cost 2· # **light** children on rightmost paths before meld

Skew Heaps – O(1) time Meld

[D.D. Sleator, R.E. Tarjan, *Self-Adjusting Heaps*, SIAM Journal of Computing, 15(1): 52-69, 1986]

- **Meld** = Bottom-up merge of rightmost paths + swap ***all*** siblings on merge path



Φ = # heavy right children in tree + 2 · **# light children on minor & major path**

O(1) amortized Meld

Heavy right child on merge path before meld → replaced by light child ⇒ 1 potential released

Light nodes disappear from major paths (but might → heavy) ⇒ ≥ 1 potential released

(4) and (5) become a heavy or light right children on major path ⇒ potential increase by ≤ 4

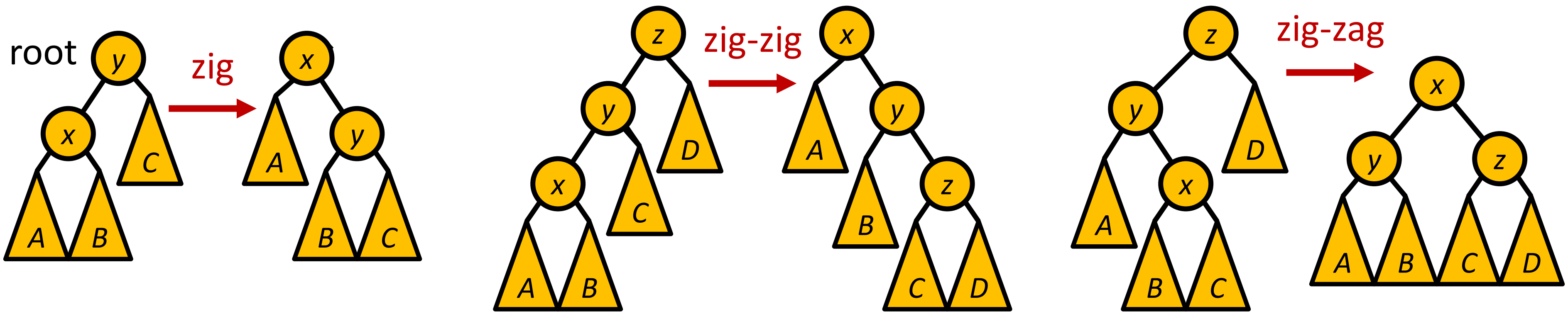
O(log *n*) amortized DeleteMin

Cutting root ⇒ 2 new minor paths, i.e. ≤ **2·log *n* new light** children on minor & major paths

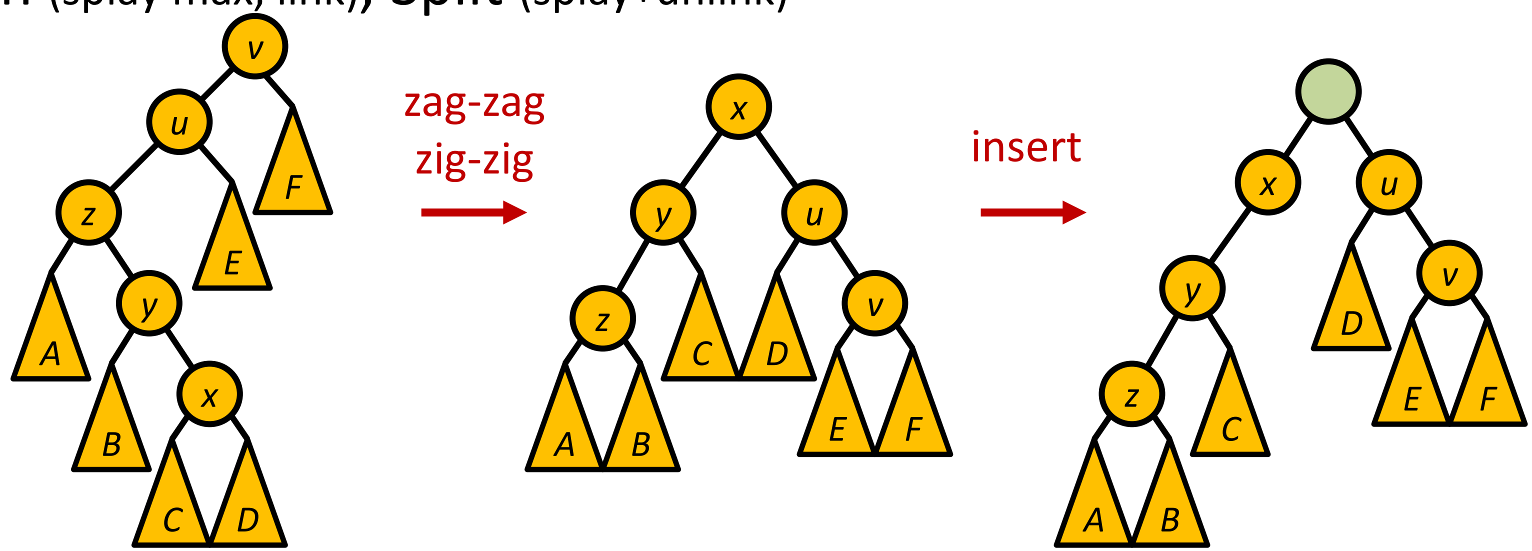
Splay Trees

[D.D. Sleator, R.E. Tarjan, *Self-Adjusting Binary Search Trees*, Journal of the ACM, 32(3): 652-686, 1985]

- Binary search tree with ***no*** balance information
- **splay(*x*)** = rotate x to root (zig/zag, zig-zig/zag-zag, zig-zag/zag-zig)



- Search (splay), Insert (splay predecessor+new root), Delete (splay+cut root+join), Join (splay max, link), Split (splay+unlink)



Splay Trees

[D.D. Sleator, R.E. Tarjan, *Self-Adjusting Binary Search Trees*, Journal of the ACM, 32(3): 652-686, 1985]

- The access bounds of splay trees are amortized

  (1) O(log n)
  (2) Static optimal
  (3) Static finger optimal
  (4) Working set optimal (proof requires dynamic change of weight)

- **Static optimality: $\Phi = \sum_v \log |T_v|$**