COS 423 Lecture 6
Implicit Heaps
Pairing Heaps

Heap(priority queue): contains a set of items x, each with a key $k(x)$ from a totally ordered universe, and associated information. We assume no ties in keys.

Basic Operations:

make-heap: Return a new, empty heap.

insert(x, H): Insert x and its info into heap H.

delete-min(H): Delete the item of min key from H.

Additional Operations:

find-min(H): Return the item of minimum key in H.

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

decrease-key(x, k, H): Replace the key of item x in heap H by k, which is smaller than the current key of x.

delete(x, H): Delete item x from heap H.

Assumption: Heaps are item-disjoint.

A heap is like a dictionary but no access by key; can only retrieve the item of min key: *decrease-key*(x, k, H) and *delete*(x, H) are given a pointer to the location of x in heap H

Applications:

Priority-based scheduling and allocation

Discrete event simulation

Network optimization: Shortest paths, Minimum spanning trees

Lower bound from sorting

Can sort n numbers by doing n *inserts* followed by n *delete-min's*.

Since sorting by binary comparisons takes $\Omega(n\lg n)$ comparisons, the amortized time for either *insert* or *delete-min* must be $\Omega(\lg n)$.

One can modify any heap implementation to reduce the amortized time for *insert* to $O(1)$ $\rightarrow$ *delete-min* takes $\Omega(\lg n)$ amortized time.

Our goal

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

O(1) amortized time for all other operations

Binary search tree implementation

Represent a heap by a binary search tree, with item order symmetric by key.

Need parent pointers for *decrease-key*, *delete*; do a *decrease-key* as a *delete* followed by an *insert*.

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

Alternative: Heap-ordered tree

Heap order: $k(p(x)) \le k(x)$ for all nodes x.
Defined for rooted trees, not just binary trees

Heap order → item in root has min key
→ *find-min* takes O(1) time

What tree structure? How to implement heap operations?

Three heap implementations

Implicit heap: Very simple, fast, small space. O(lg$n$) worst-case time per operation except for *meld*.

Pairing heap: O(lg$n$) amortized time per operation including *meld*, simple, self-adjusting.

Rank-pairing heap: Achieves our goal.

Heap-ordered tree: internal representation

Store items in nodes of a rooted tree, in heap order.

Find-min: return item in root.

Insert: replace any null child by a new leaf containing the new item *x*. To restore heap order, *sift up*: while *x* is not in the root and *x* has key less than that in parent, swap *x* with item in parent.

Delete-min or *delete*: Delete item. To restore heap order, *sift down*: while empty node is not a leaf, fill with item of smallest key in children. Either delete empty leaf, or fill with item from another leaf, sift moved item up, and delete empty leaf. (Allows deletion of an arbitrary leaf, so tree shape can be controlled)

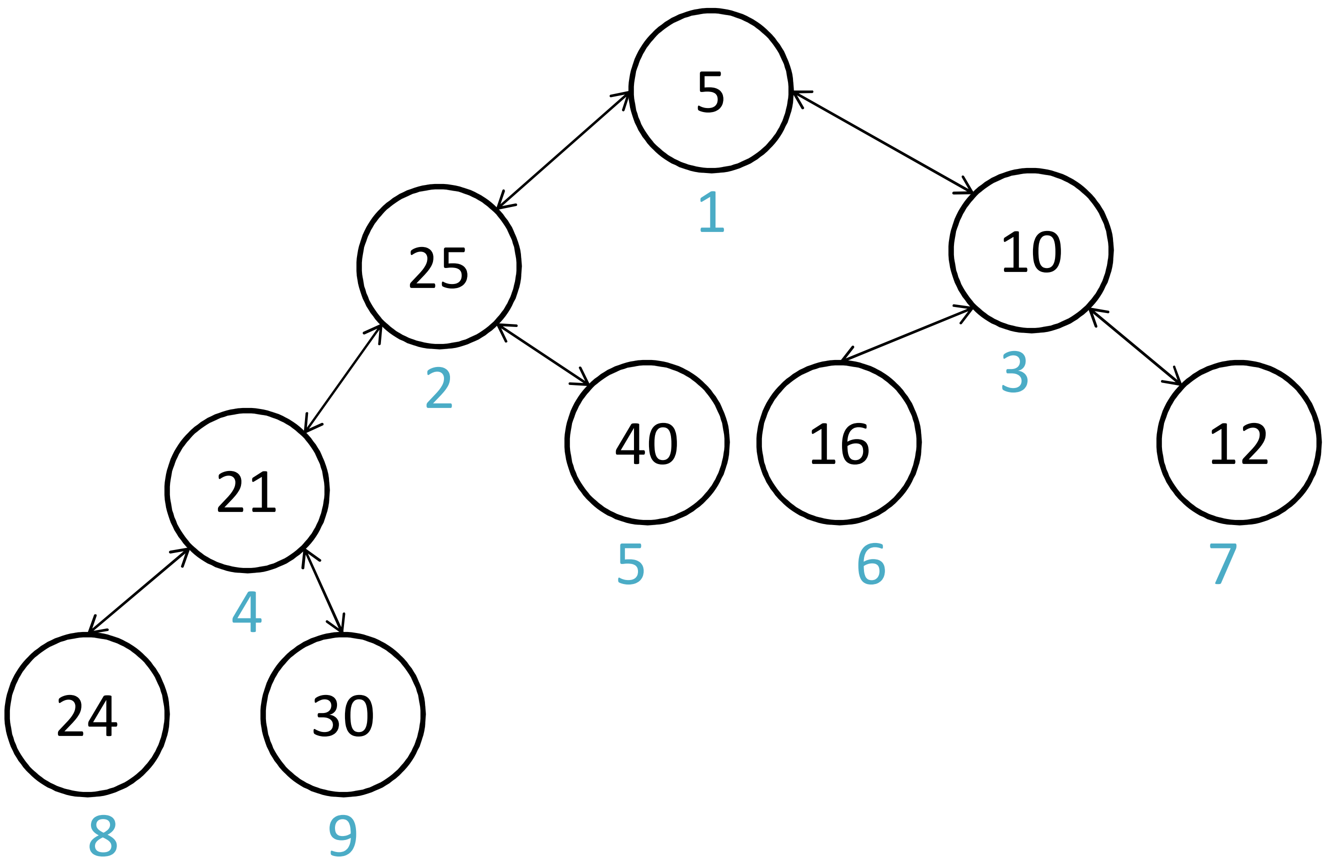
Decrease-key: sift up.

Choice of leaf to add or delete is arbitrary: add level-by-level, delete last-in, first-out.

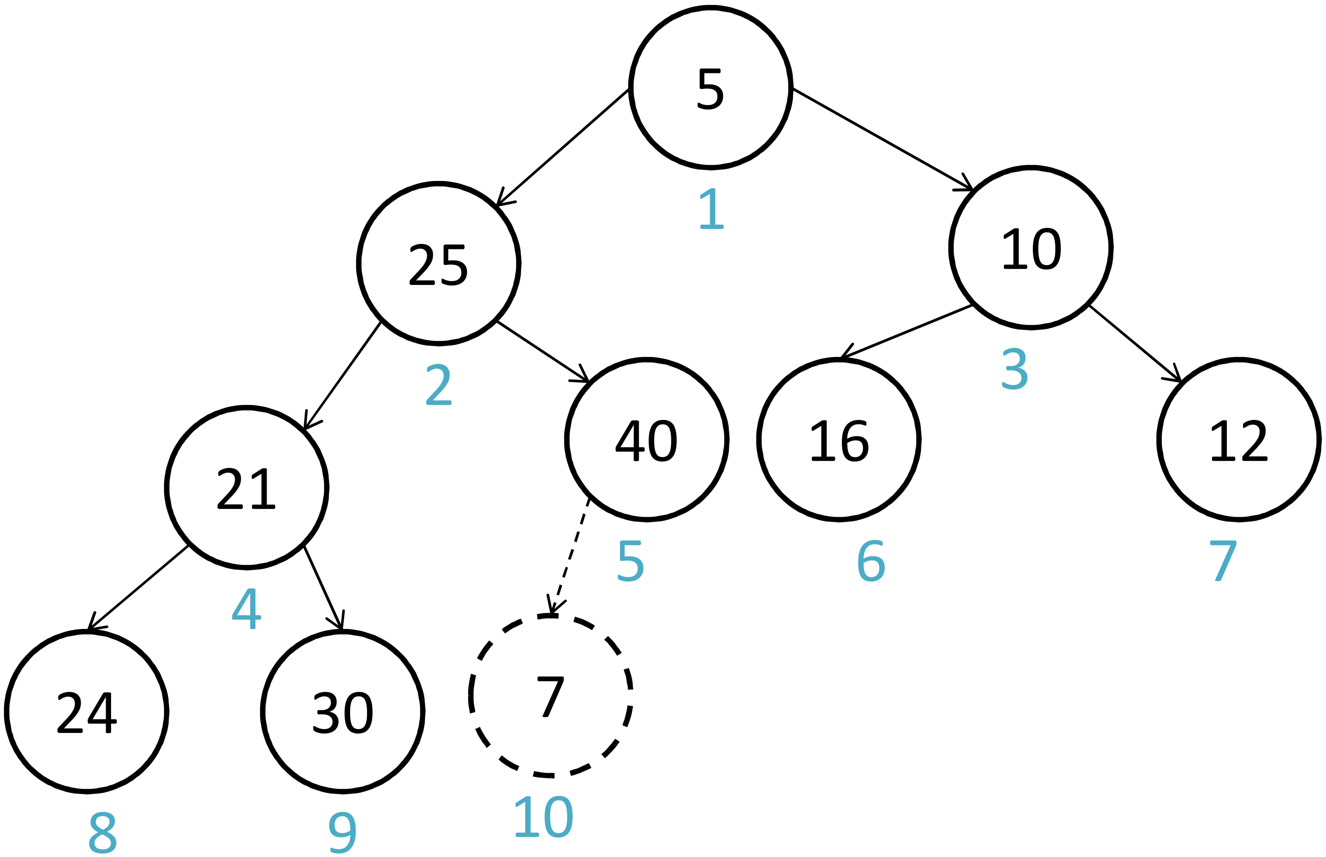
A binary heap

Numbers in nodes are keys.

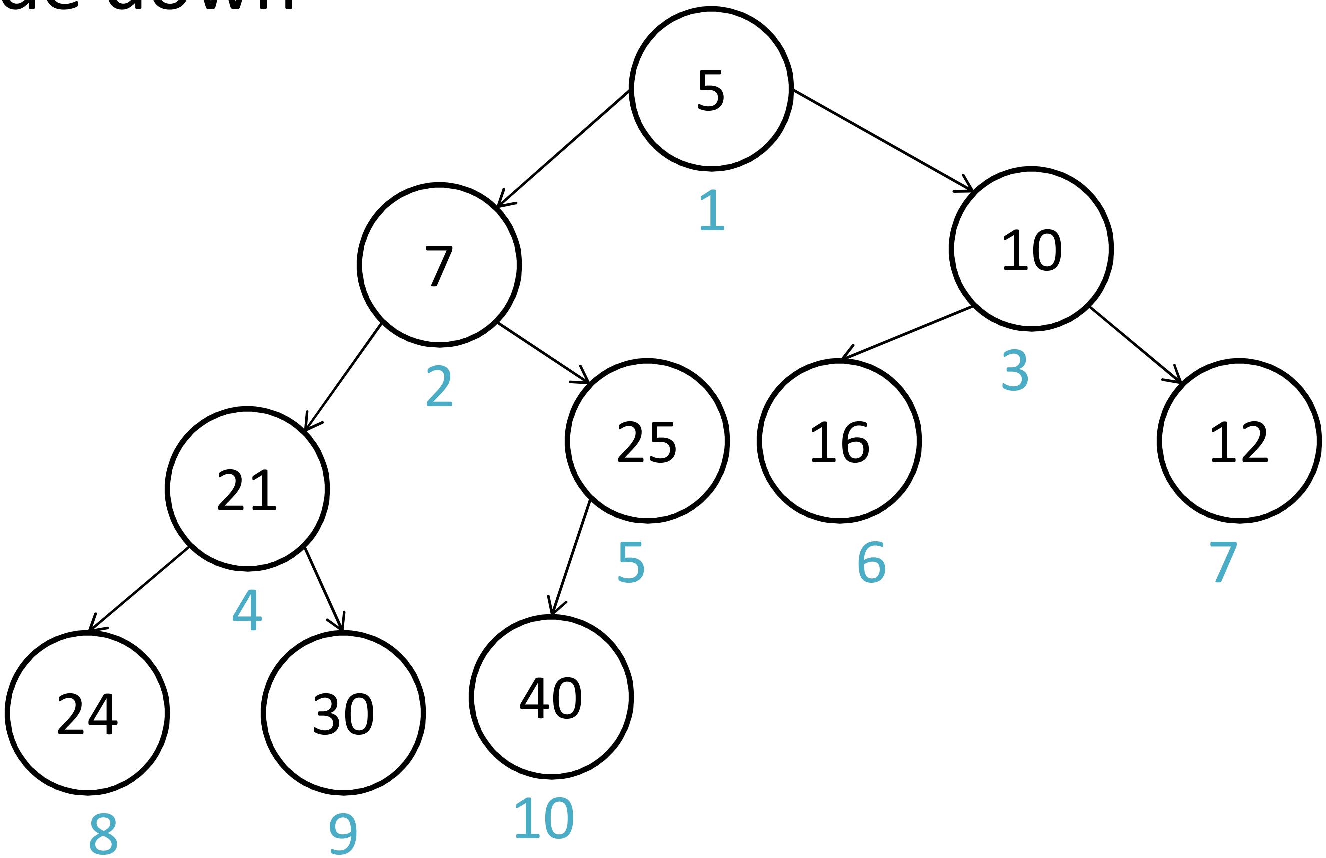
Numbers next to nodes are order of addition.



insert 7



delete-min: remove item in root, sift empty node down

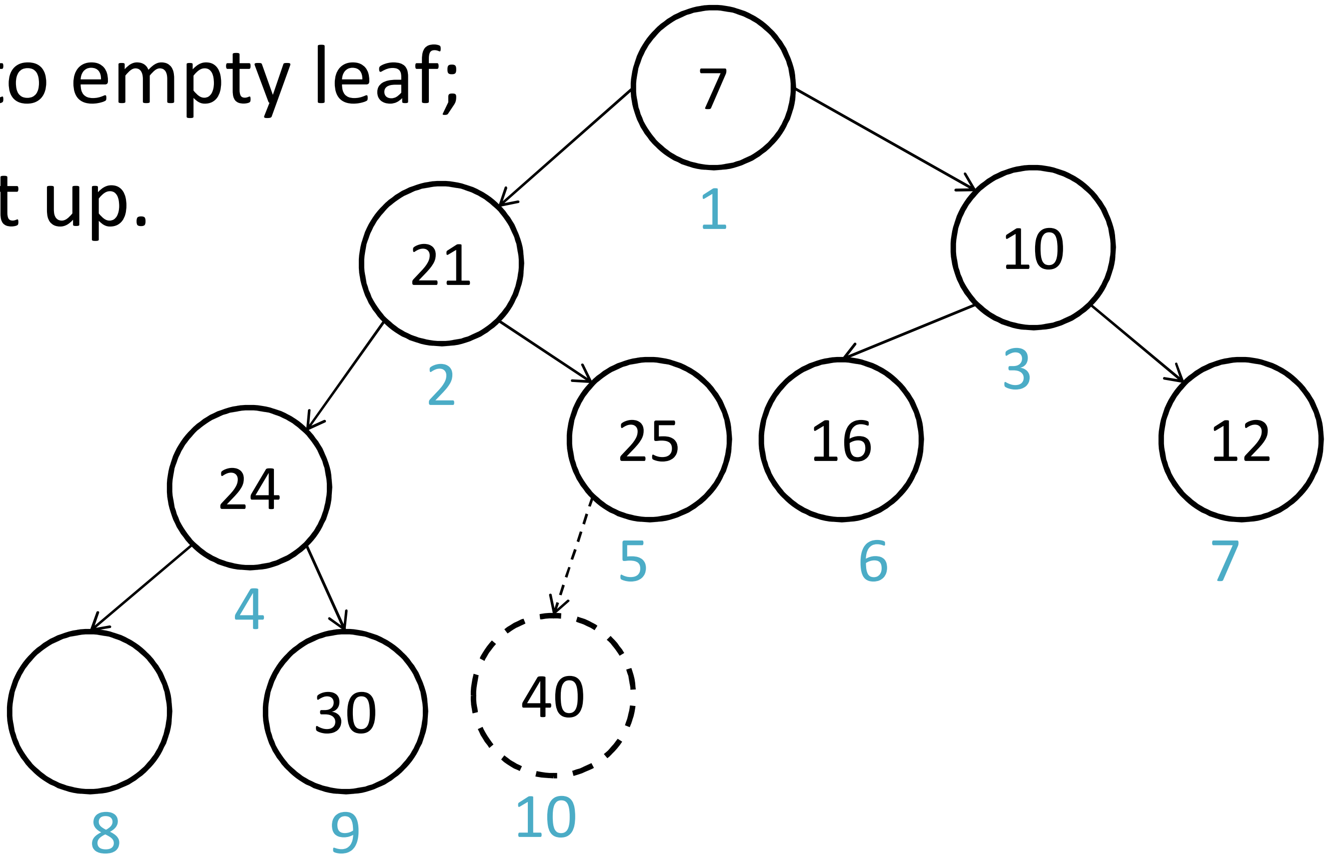


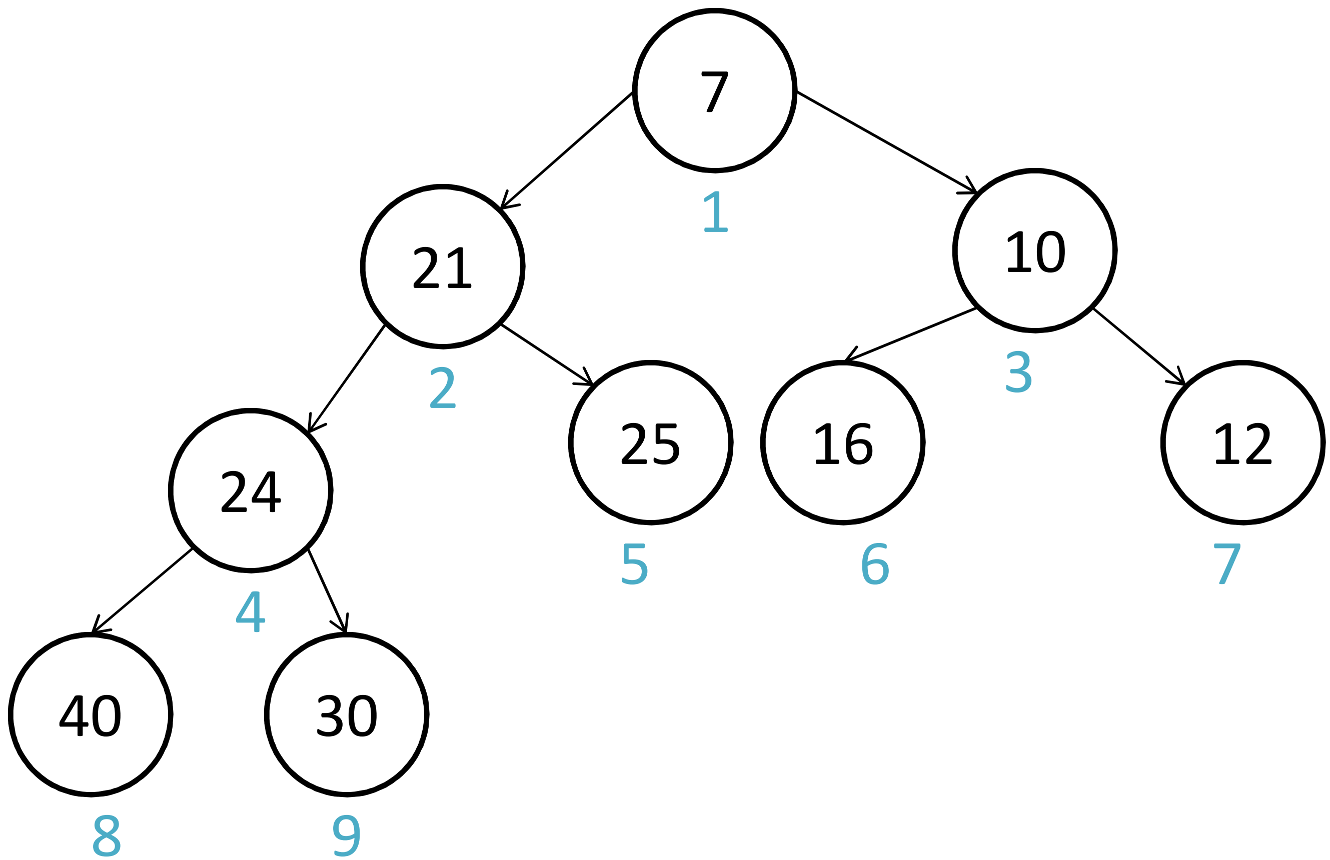
End of sift-down

Swap item in last leaf

into empty leaf;

sift up.





7
1
21
2
10
3
24
4
25
5
16
6
12
7
40
8
30
9

Implicit binary heap

Binary tree, nodes numbered in addition order

root = 1

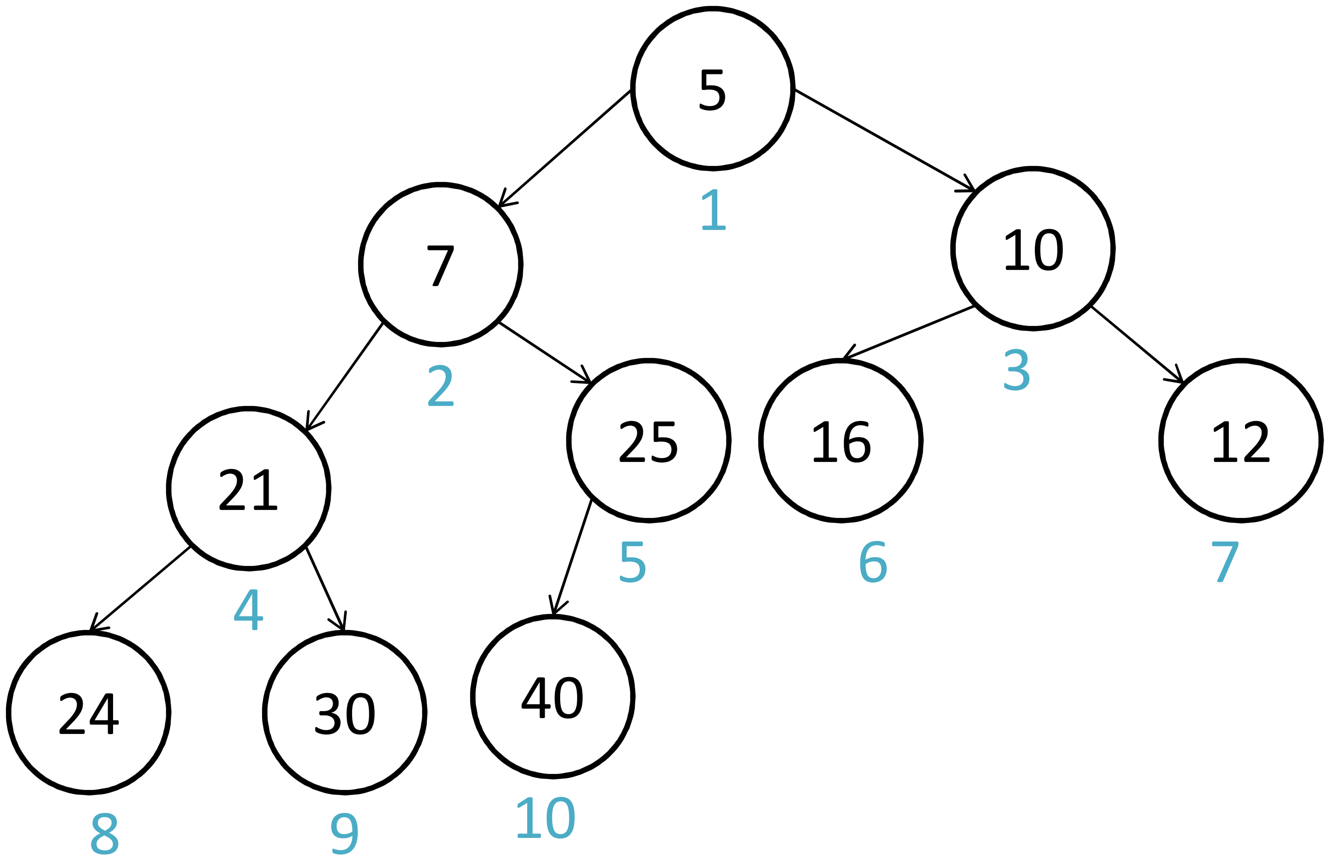
children of v = $2v$, $2v + 1$

$p(v) = \lfloor v/2 \rfloor$

→ no pointers needed! Can store in array

insert: add node $n + 1$ *delete*: delete node n

depth = $\lfloor \lg n \rfloor$



1	2	3	4	5	6	7	8	9	10
5	7	10	21	25	16	21	24	30	40

Each operation except *meld* takes O(lg*n*) time:

insert takes ≤lg*n* comparisons (likely O(1))

delete takes ≤2lg*n* comparisons (likely lg*n* + O(1))

Can reduce comparisons (but not data movement) to lglg*n* worst-case for *insert*, lg*n* + lglg*n* for *delete*

Instead of binary, can make tree d-ary. Some evidence suggests 4-ary is best in practice.

Heap-ordered tree: external representation

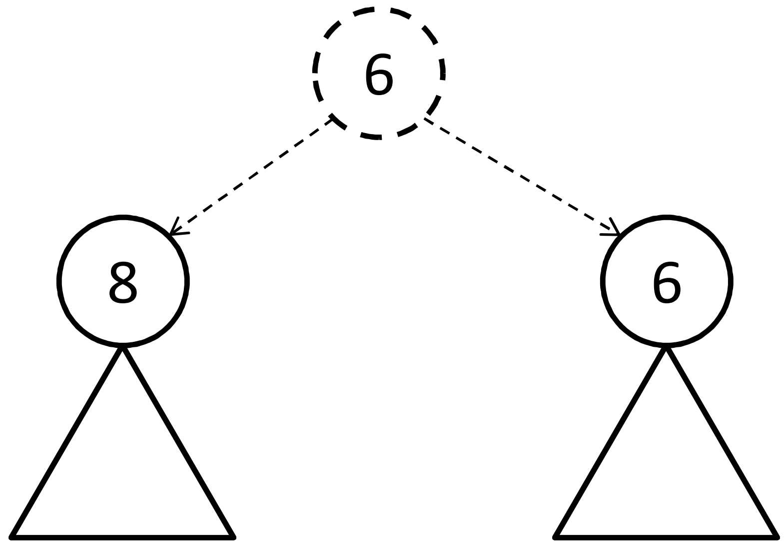
Store items in external nodes of a binary tree, in any order.

To fill internal nodes, run a tournament: bottom-up, fill each internal node with item of smaller key in children.

Find-min: return root.

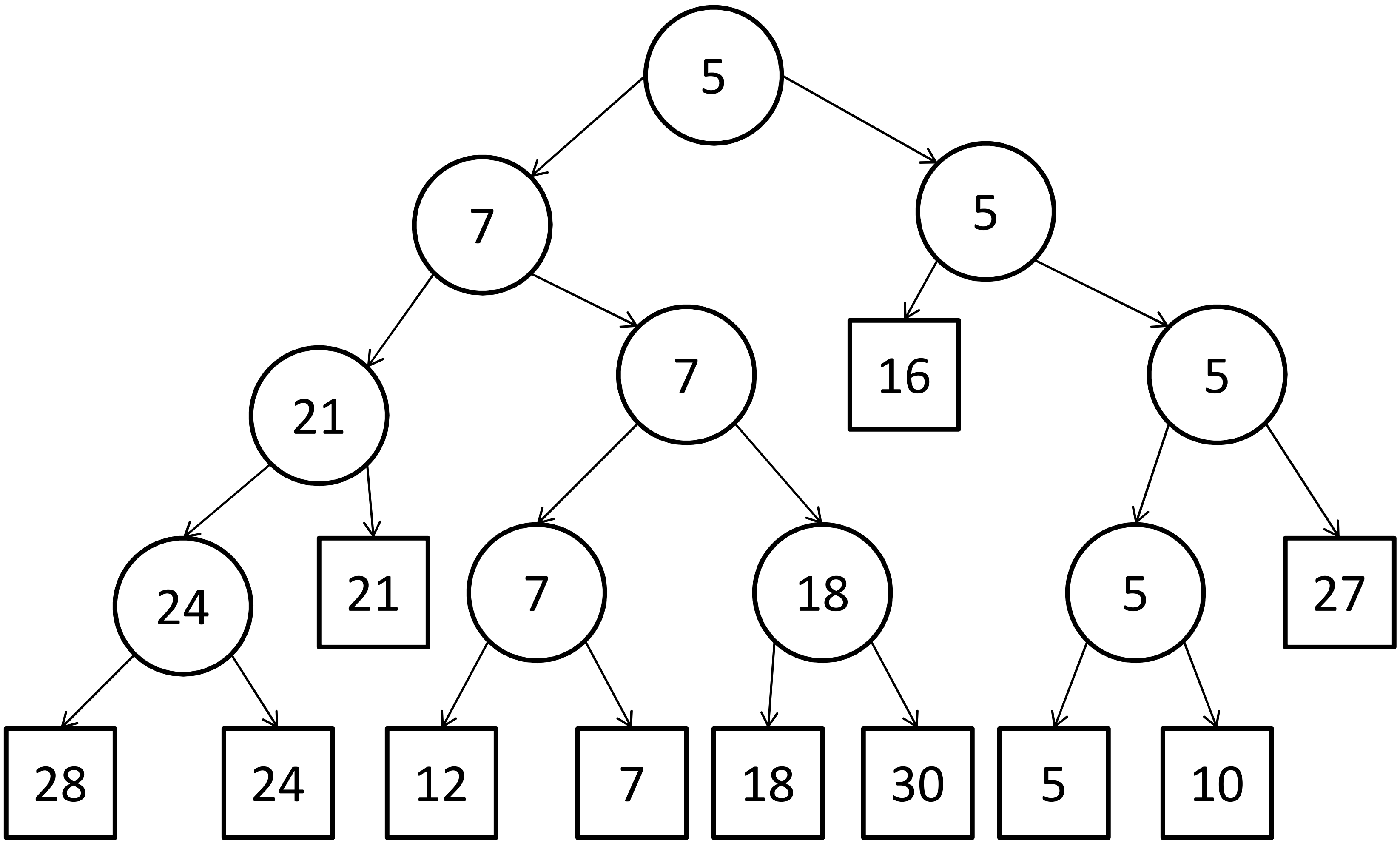
Primitive operation *link*: combine two trees by creating a new root with old roots as children, filling with item of smaller key in old roots.

A link takes one comparison and O(1) time. We will build all operations out of links and cuts.



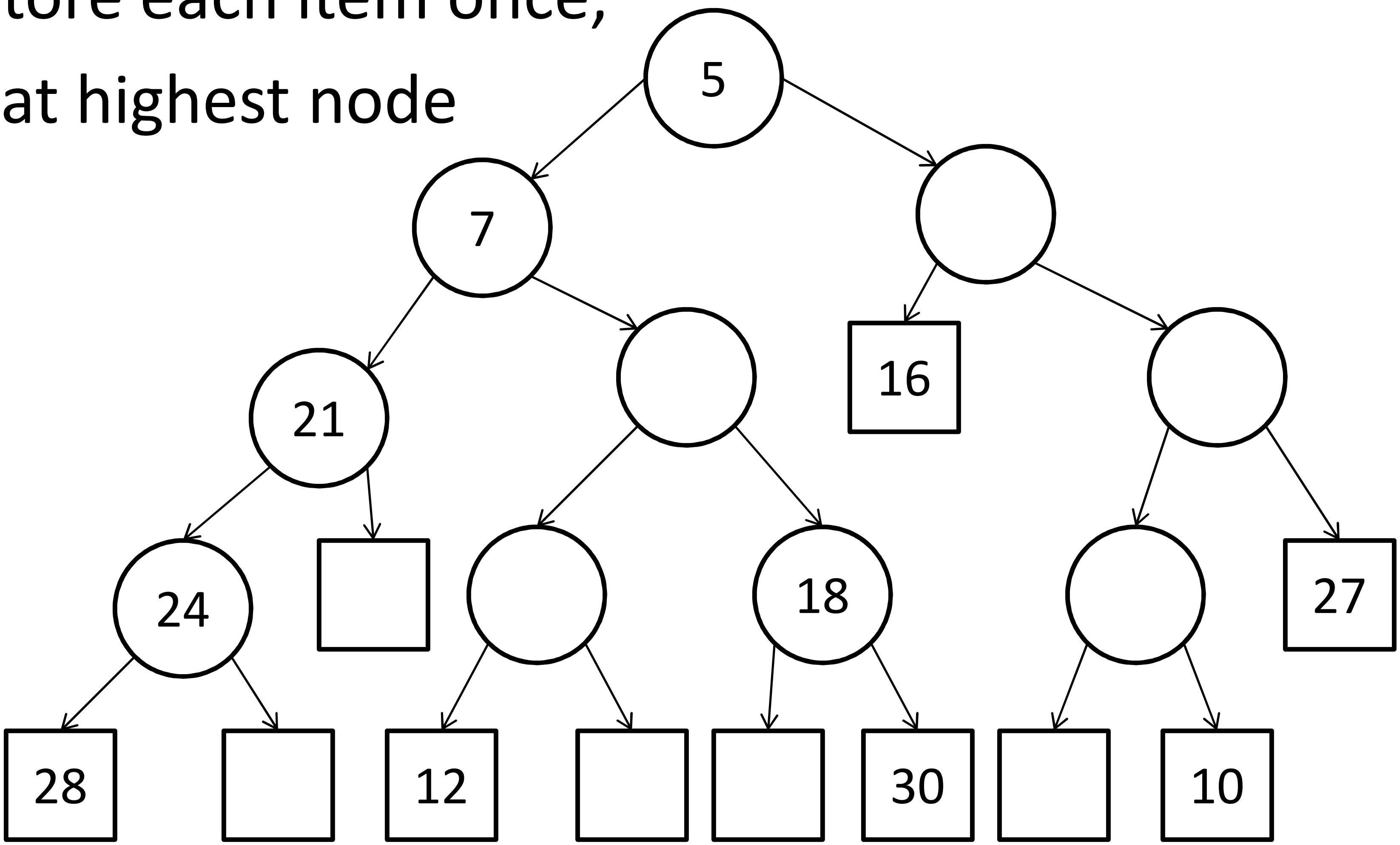
First: alternative ways to represent tournaments

Full representation



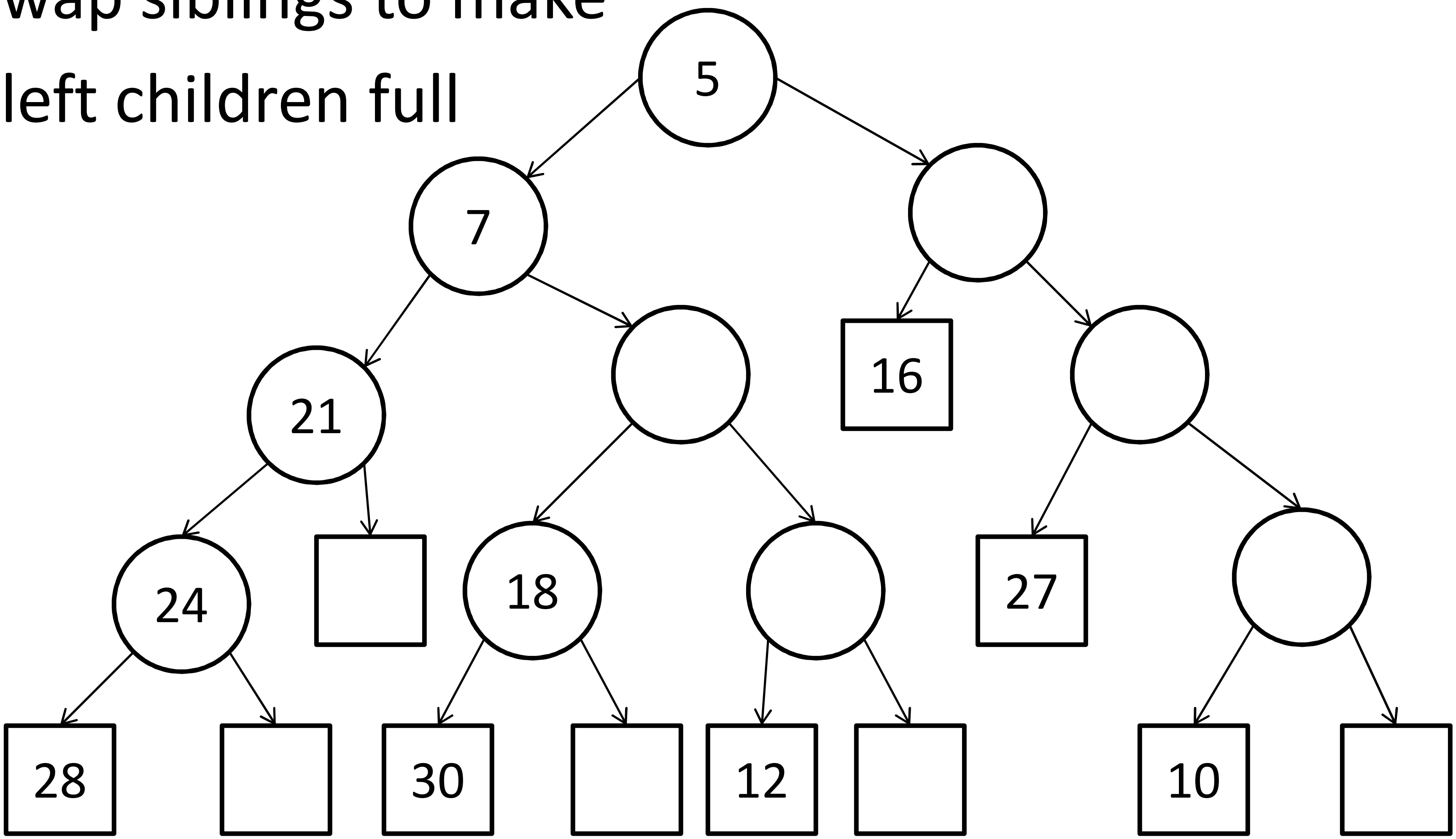
Half-full representation

Store each item once,
at highest node



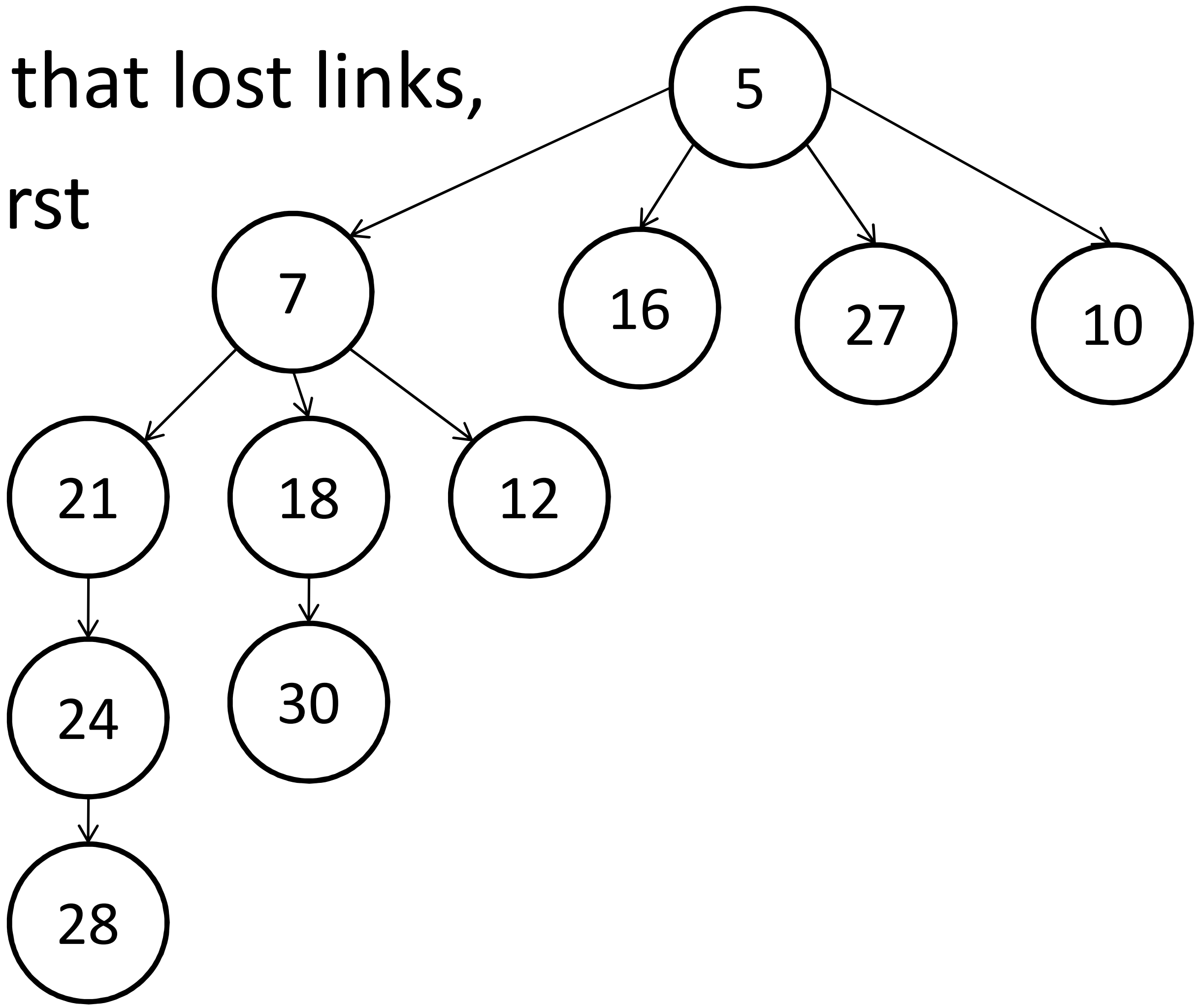
Left-full representation

Swap siblings to make left children full



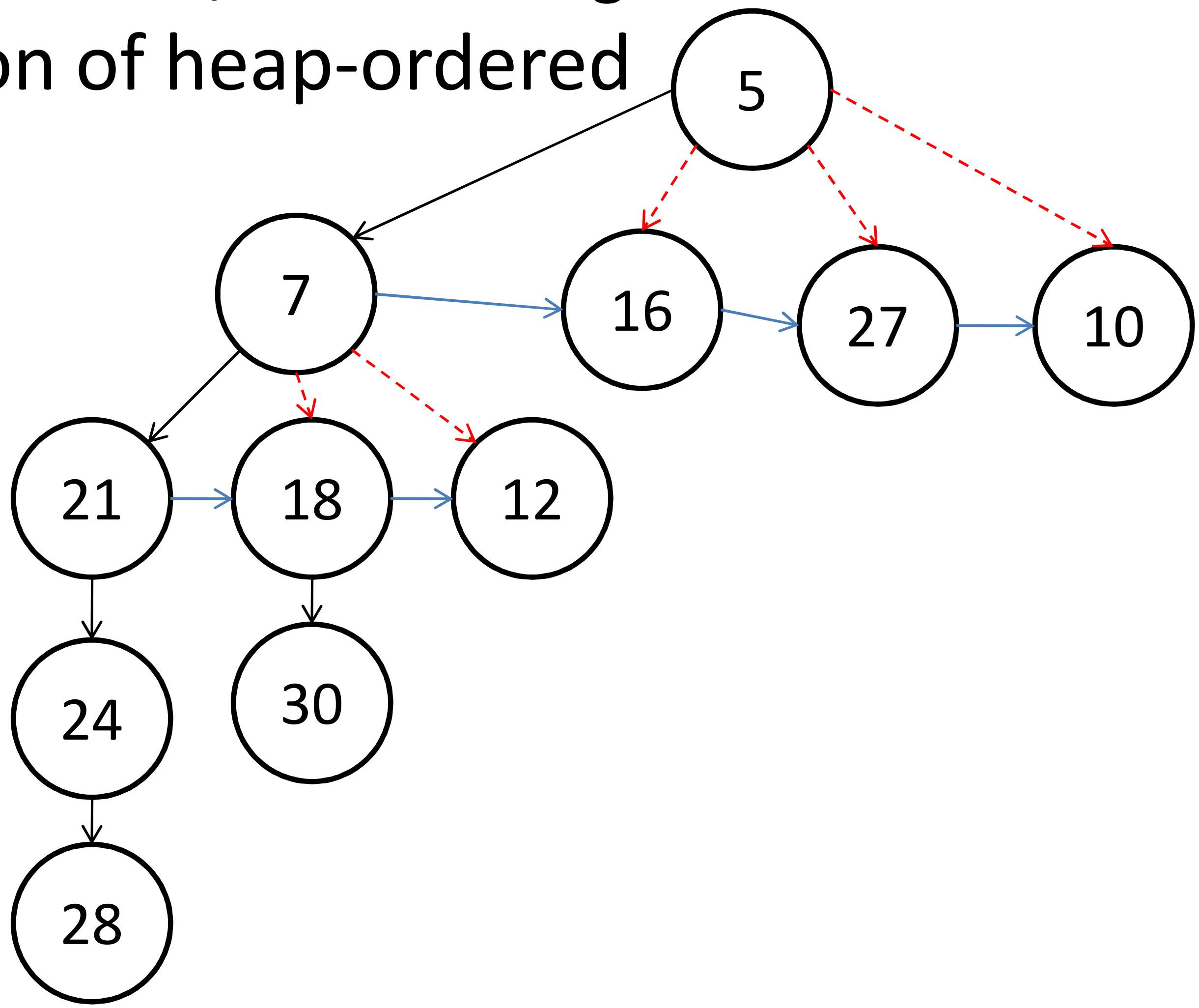
Heap-ordered representation

Heap-ordered tree, children contain items that lost links, most recent first



Half-ordered representation

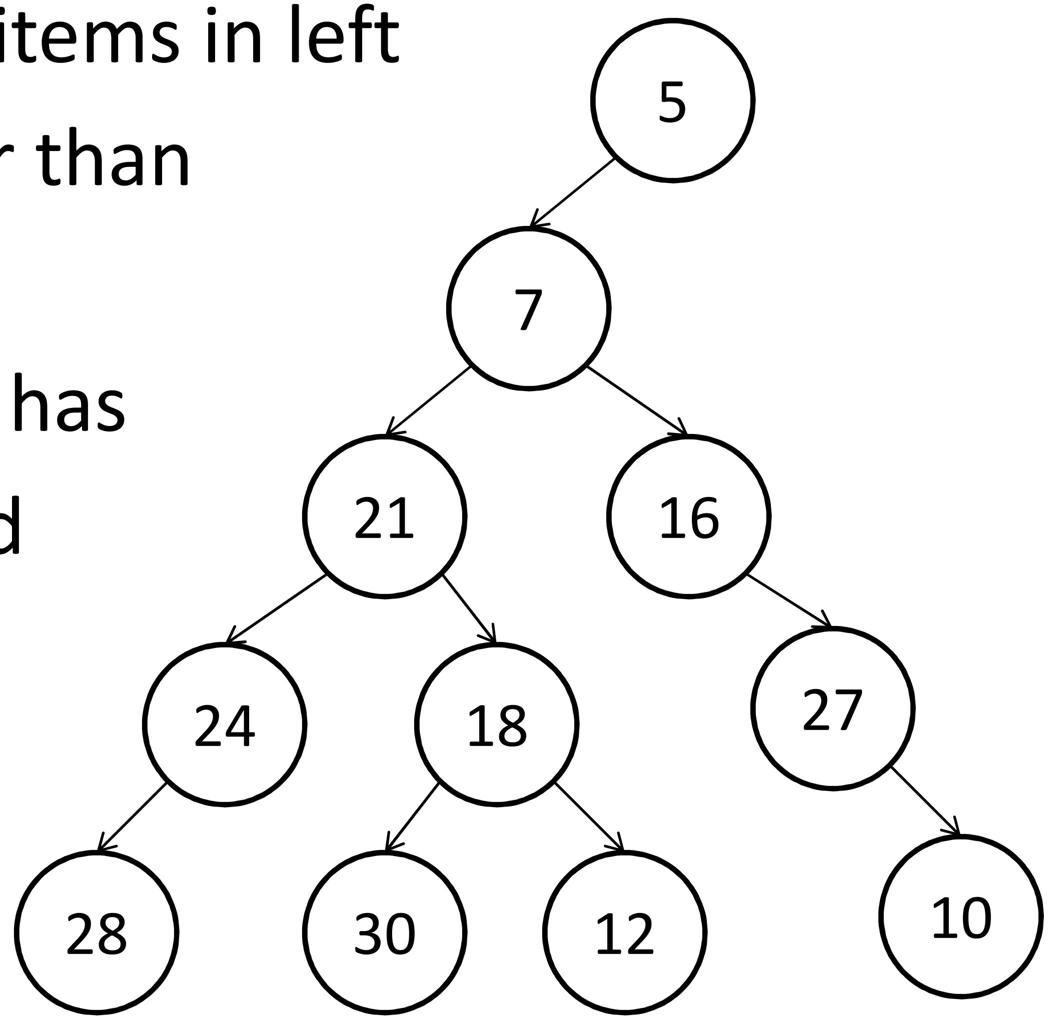
Binary tree: *first child, next sibling* representation of heap-ordered tree



Half-ordered representation

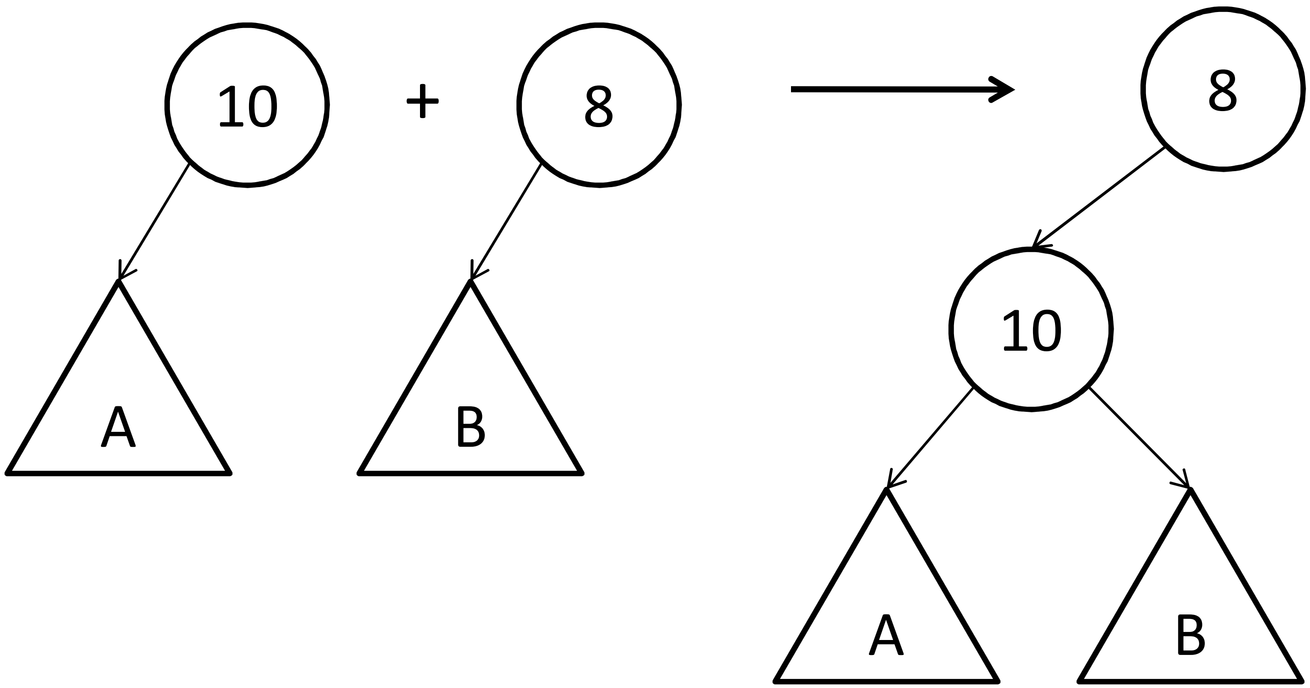
Half order: all items in left subtree larger than item in node

Half tree: root has null right child



Linking half trees

One comparison, O(1) time



Half-tree representation:

Left and right child pointers

Parent pointers if *decrease-key*, *delete* allowed

Heap operations:

find-min: return item in root

make-heap: return a new, empty half tree

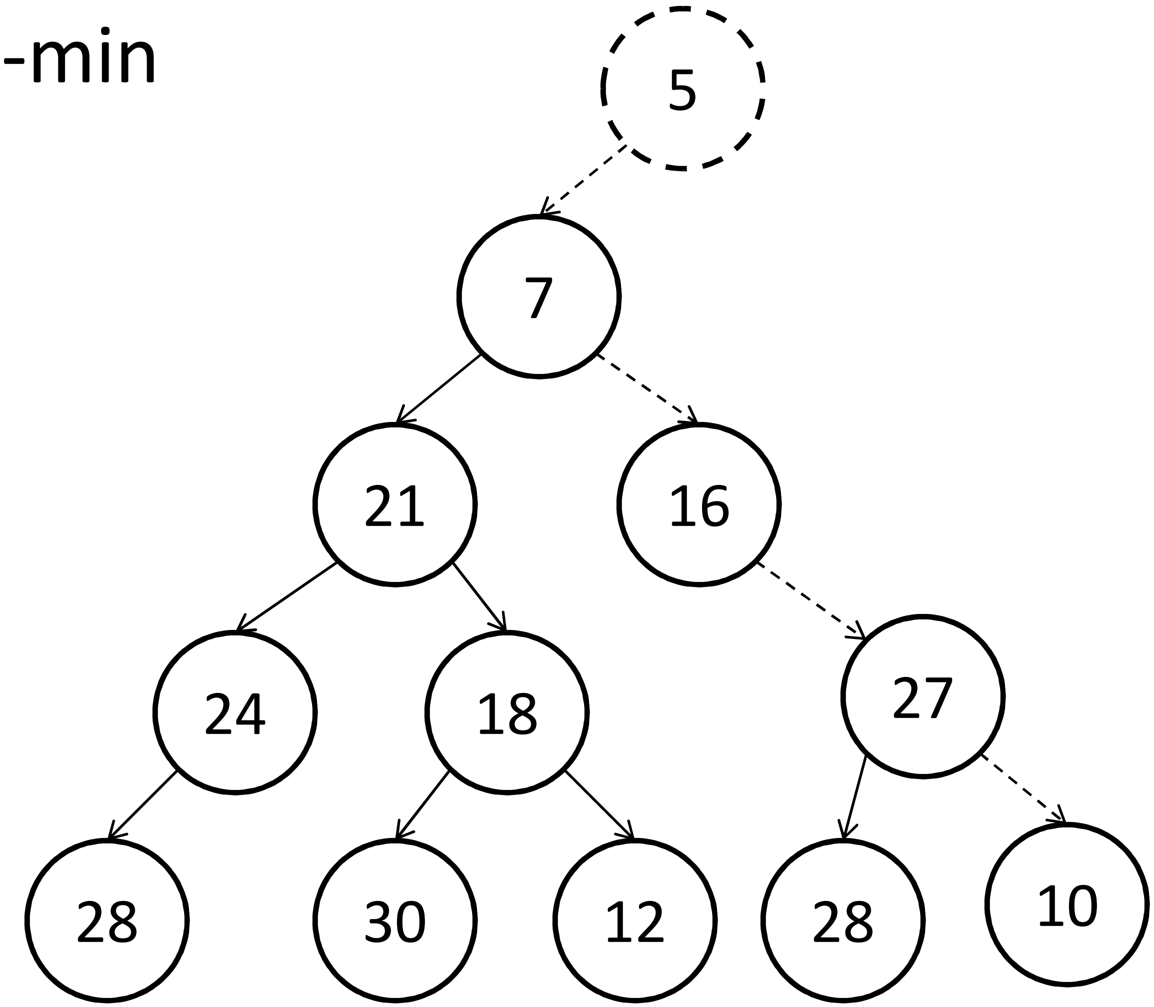
insert: create a new, one-node half tree, link with existing half tree

meld: link two half trees

delete-min: Delete root. Cut edges along right path down from new root. Roots of the resulting half trees are the losers to the old root. Must link these half trees.

How?

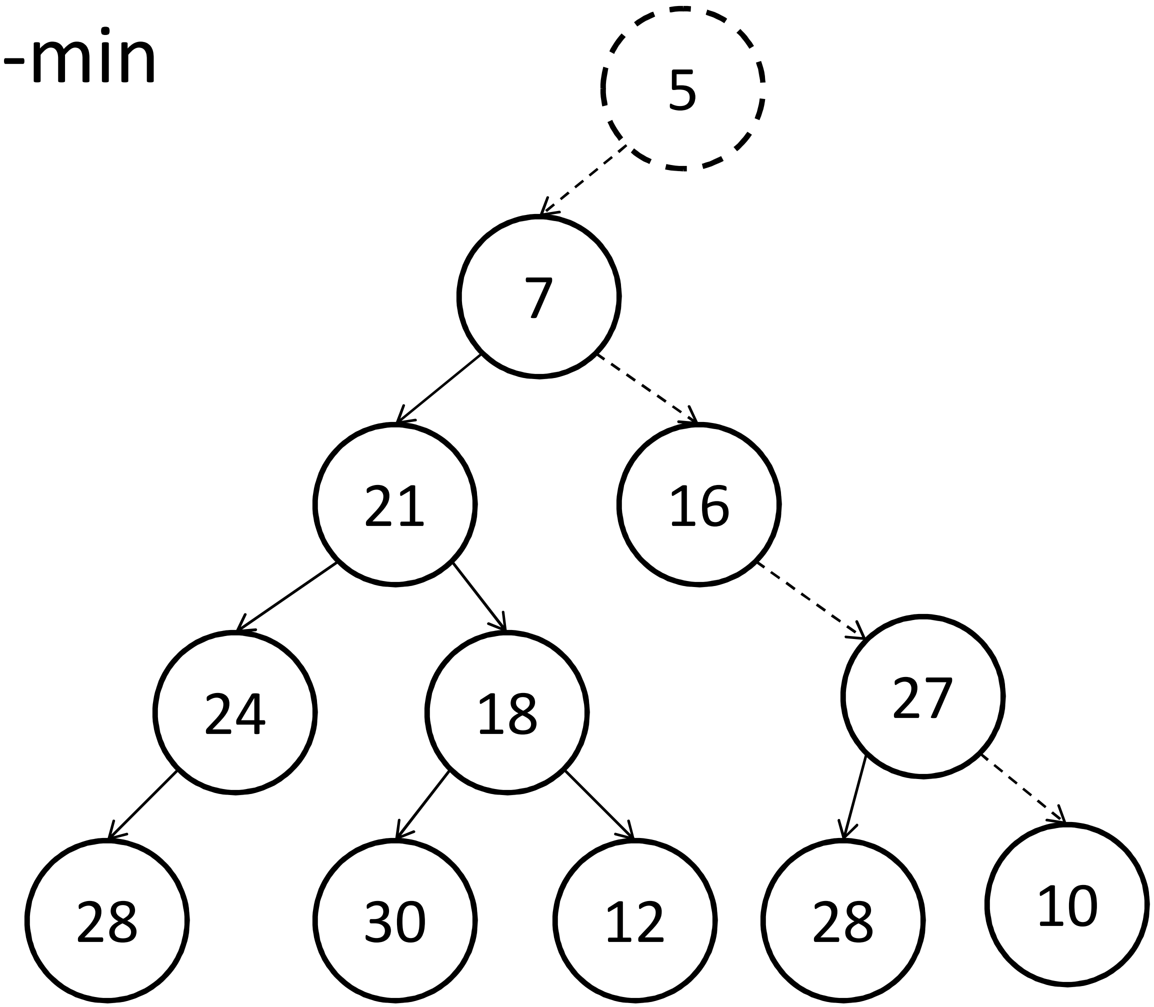
Delete-min



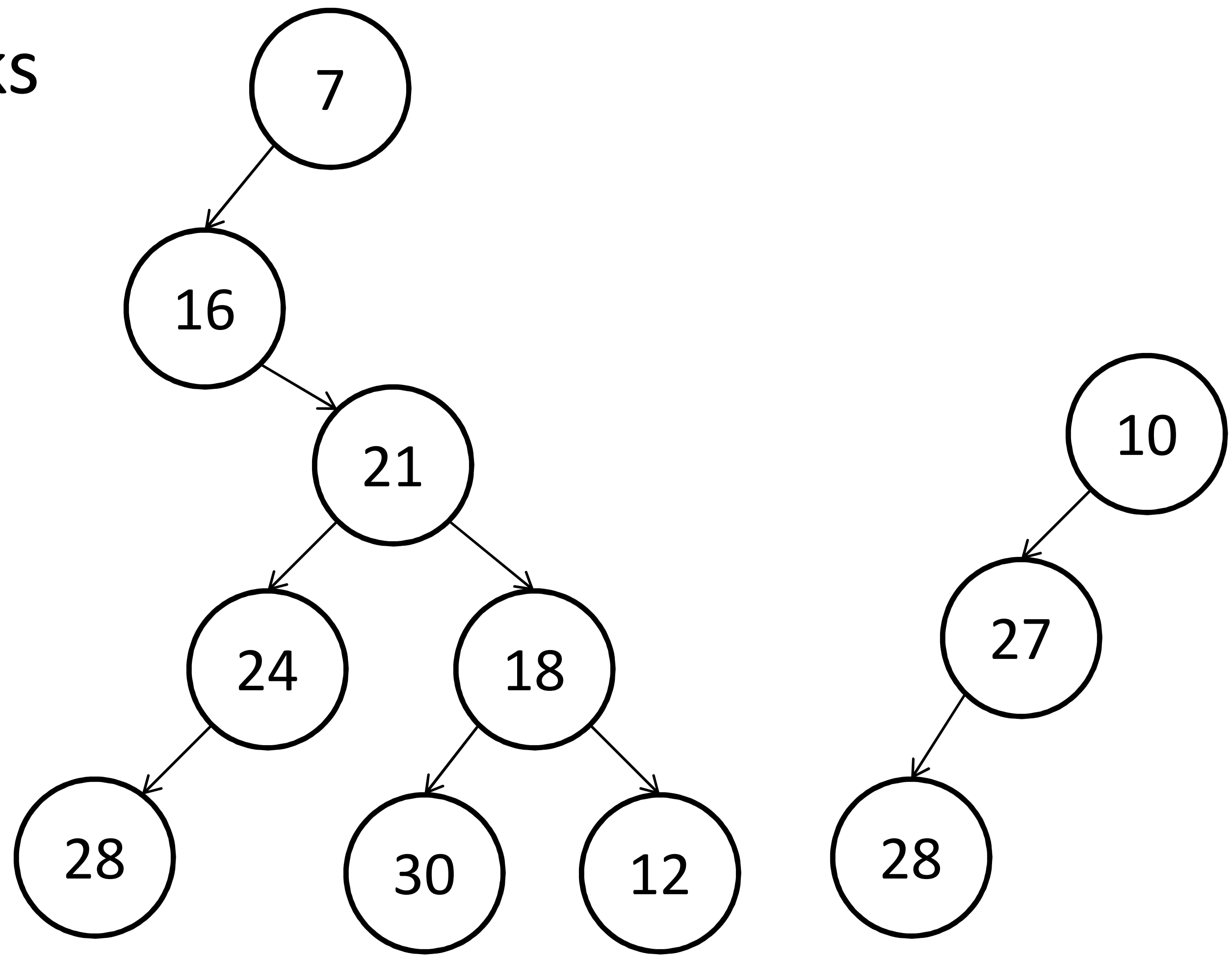
Link half trees in pairs, top-down. Then take bottom half tree and link with each new half tree, bottom-up

Pairing heap

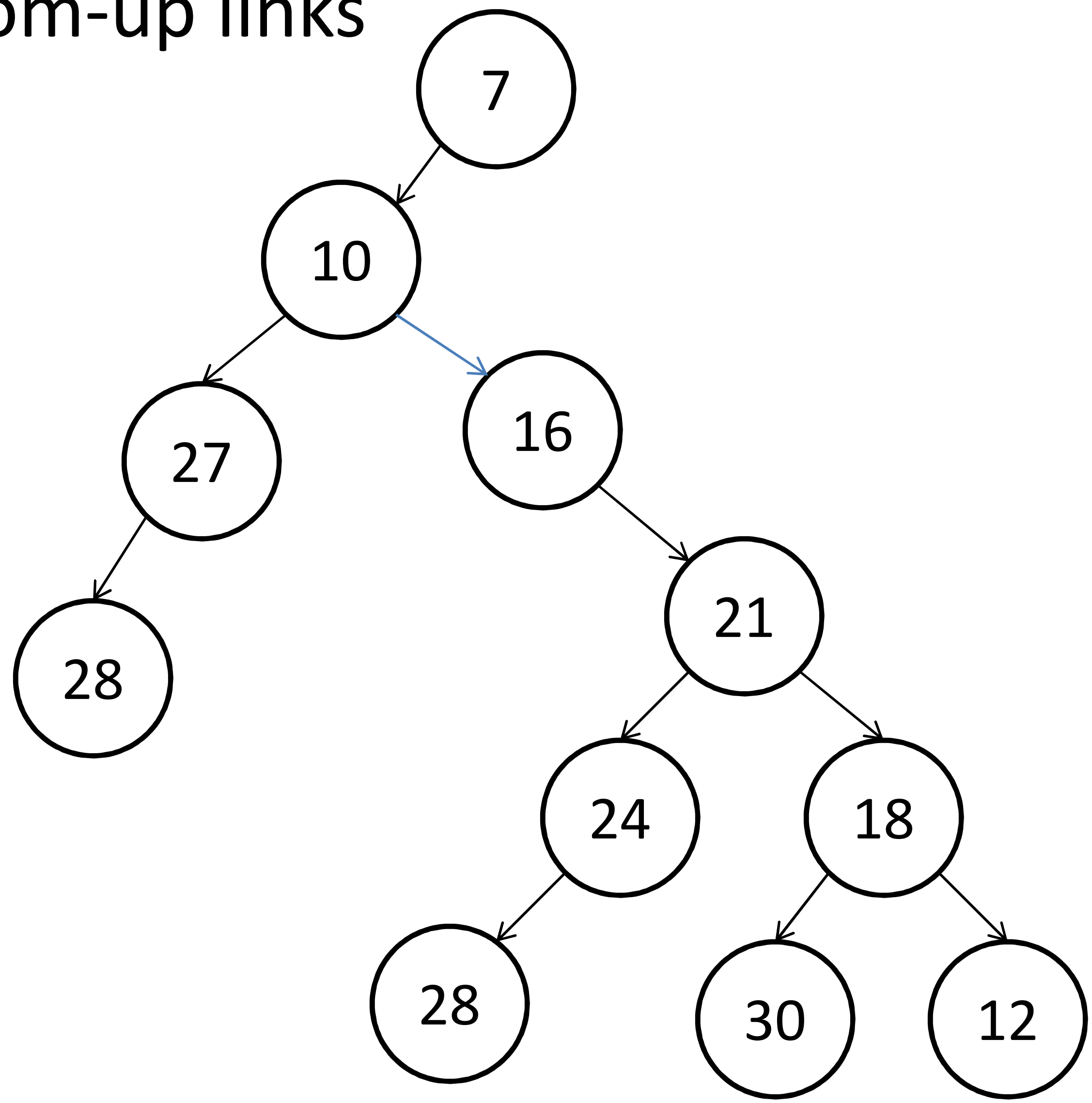
Delete-min



After top-down pairing links



After bottom-up links



Remaining heap operations:

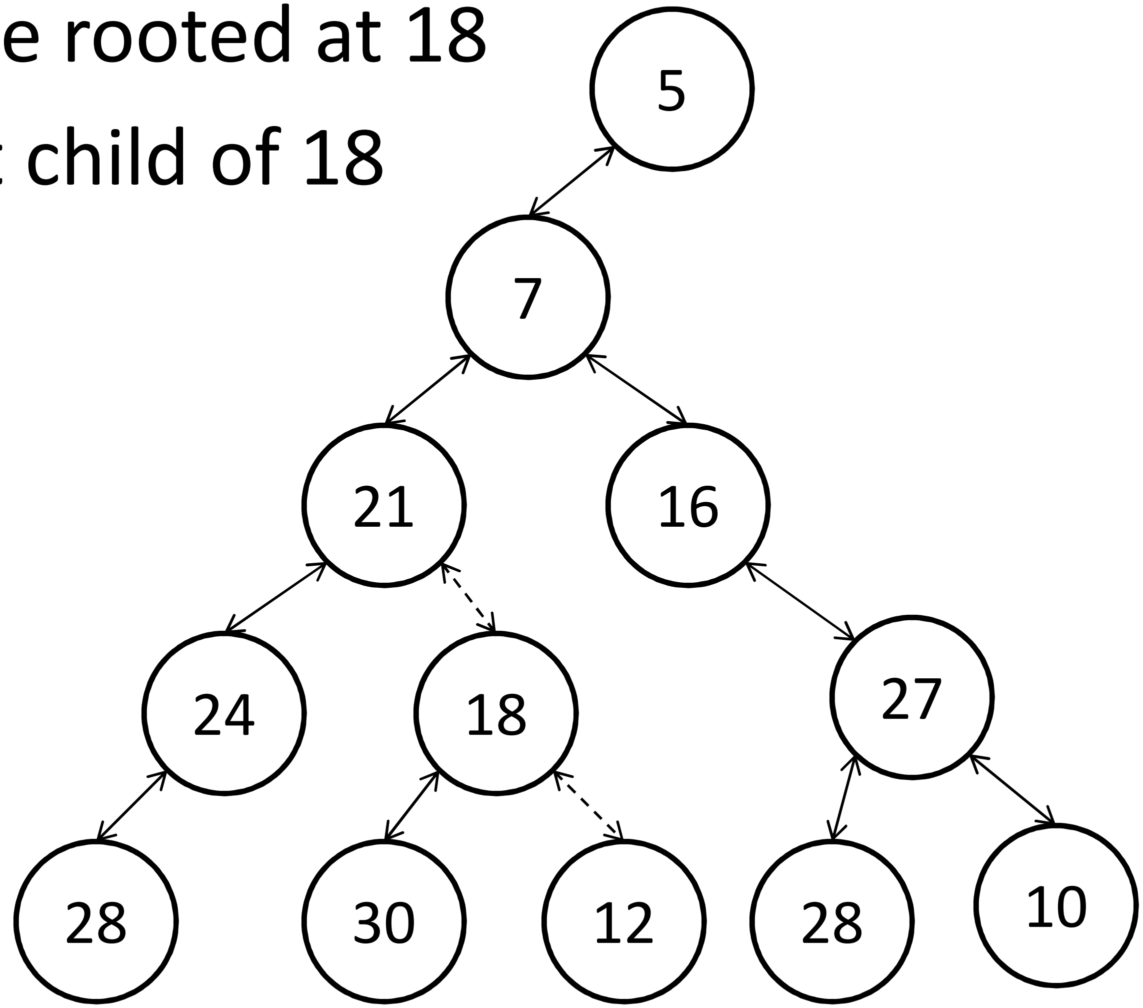
decrease key of x in heap H: Remove x and its left subtree (becomes a new half tree). Replace x by its right child. Decrease $k(x)$. Link the old half tree with the new half tree rooted at x.

delete x in heap H: *Decrease key* of x to $-\infty$; *delete-min*.

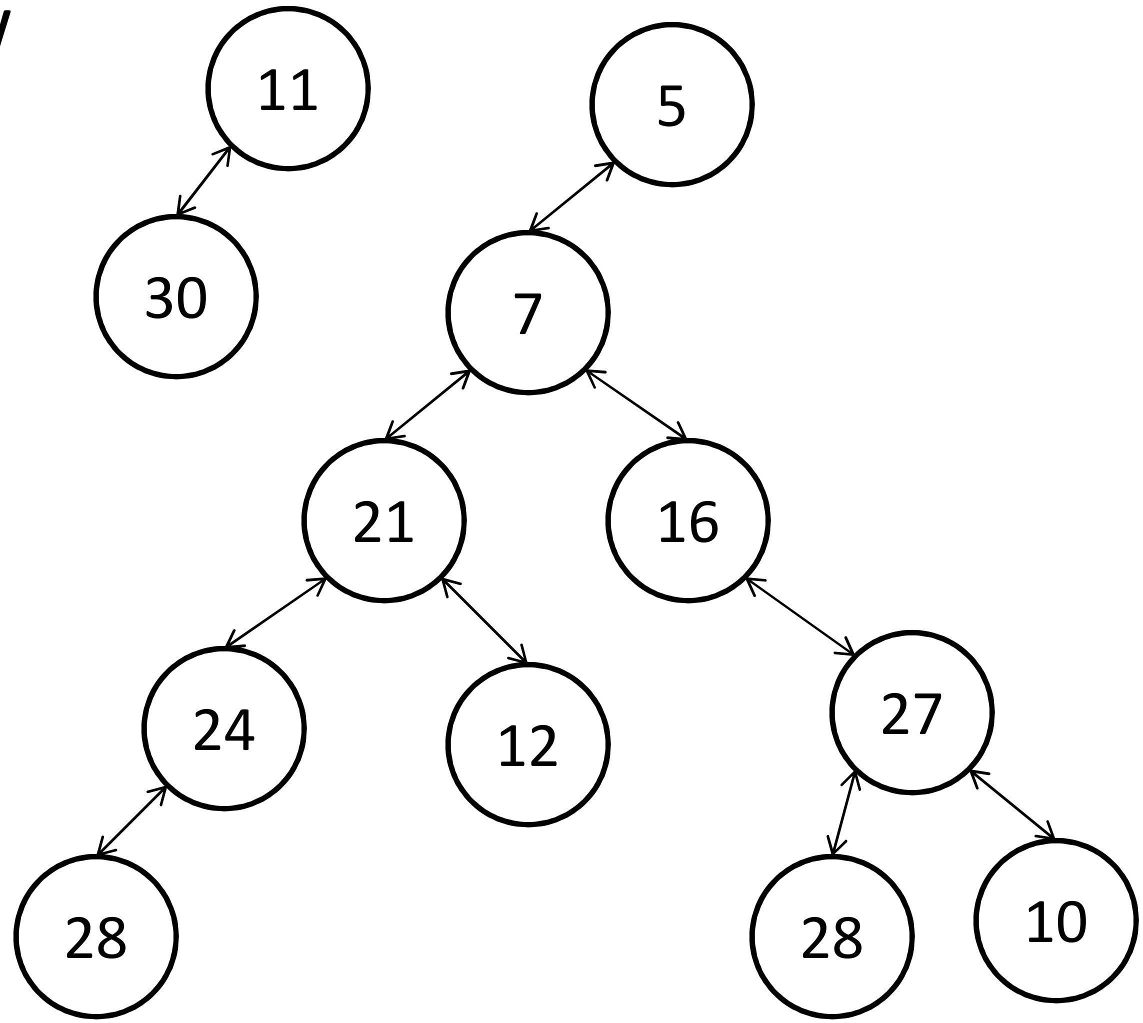
decrease key 18 to 11

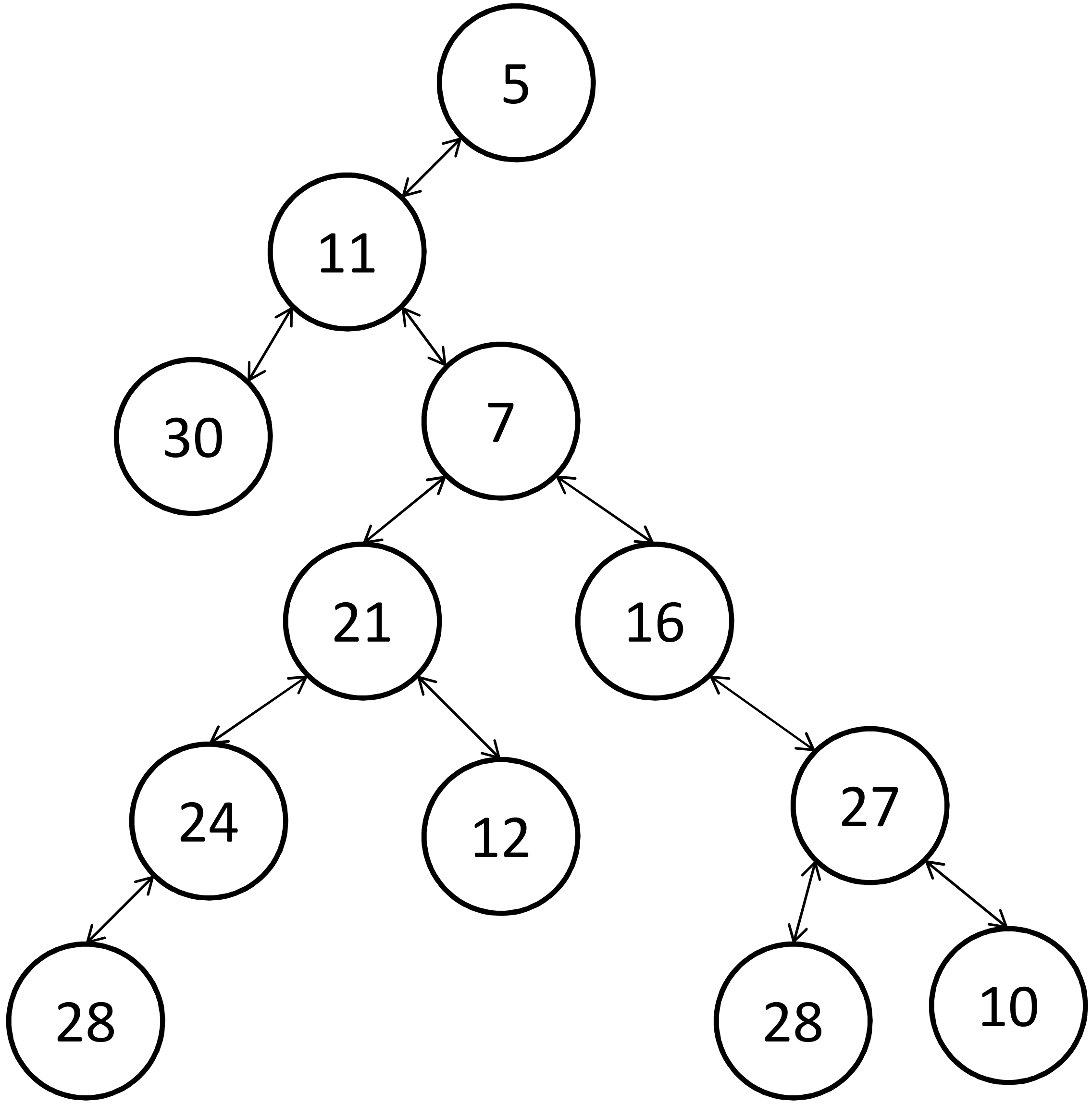
Remove half tree rooted at 18

Replace by right child of 18



Link old and new
half trees




5
11
30
7
21
16
24
12
27
28
28
10

Analysis of pairing heaps

Need to count links done during *delete-min*.

Delete-min is just like splaying, except for

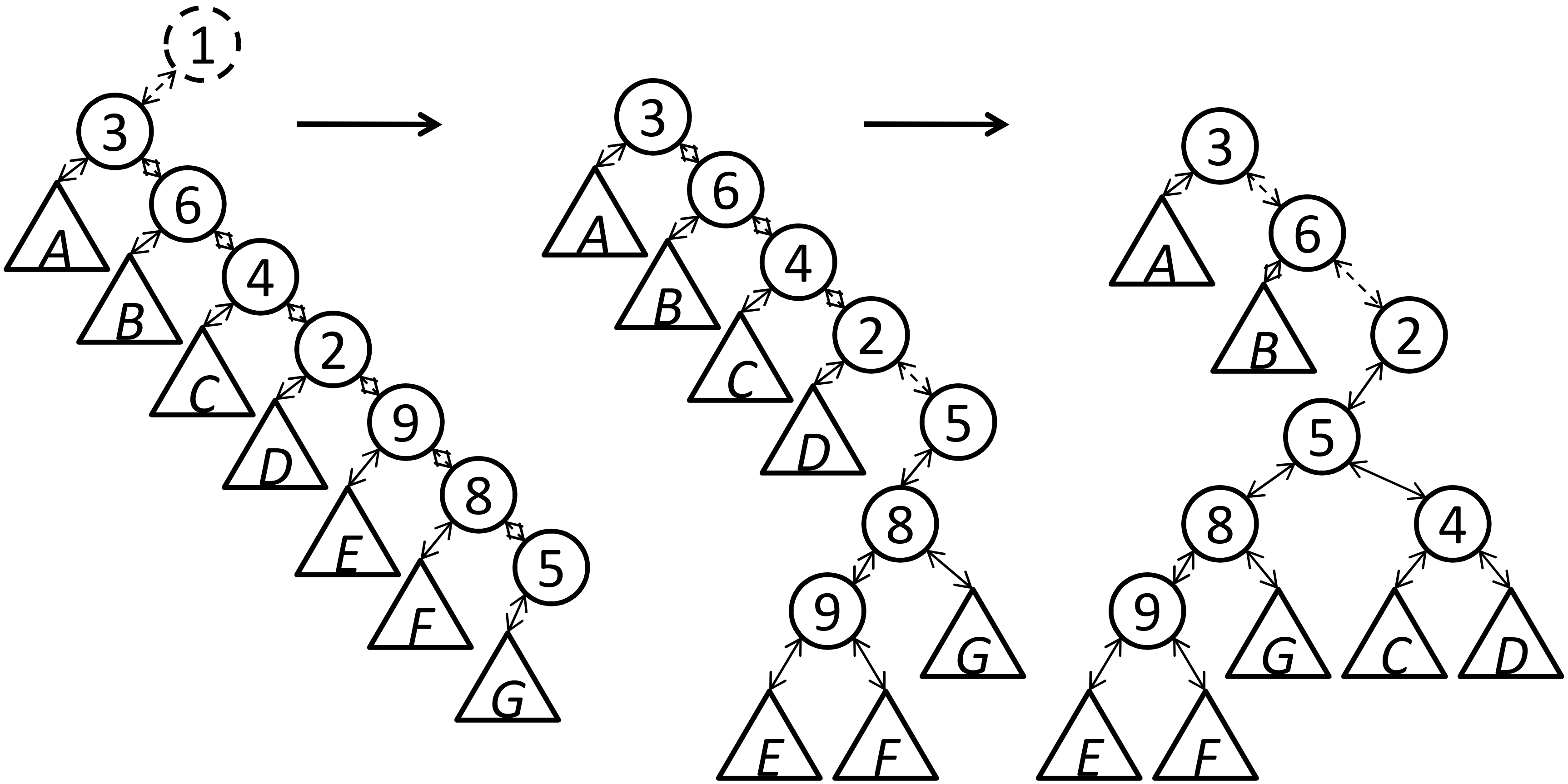
(i) swapping of some left and right subtrees and some nodes;

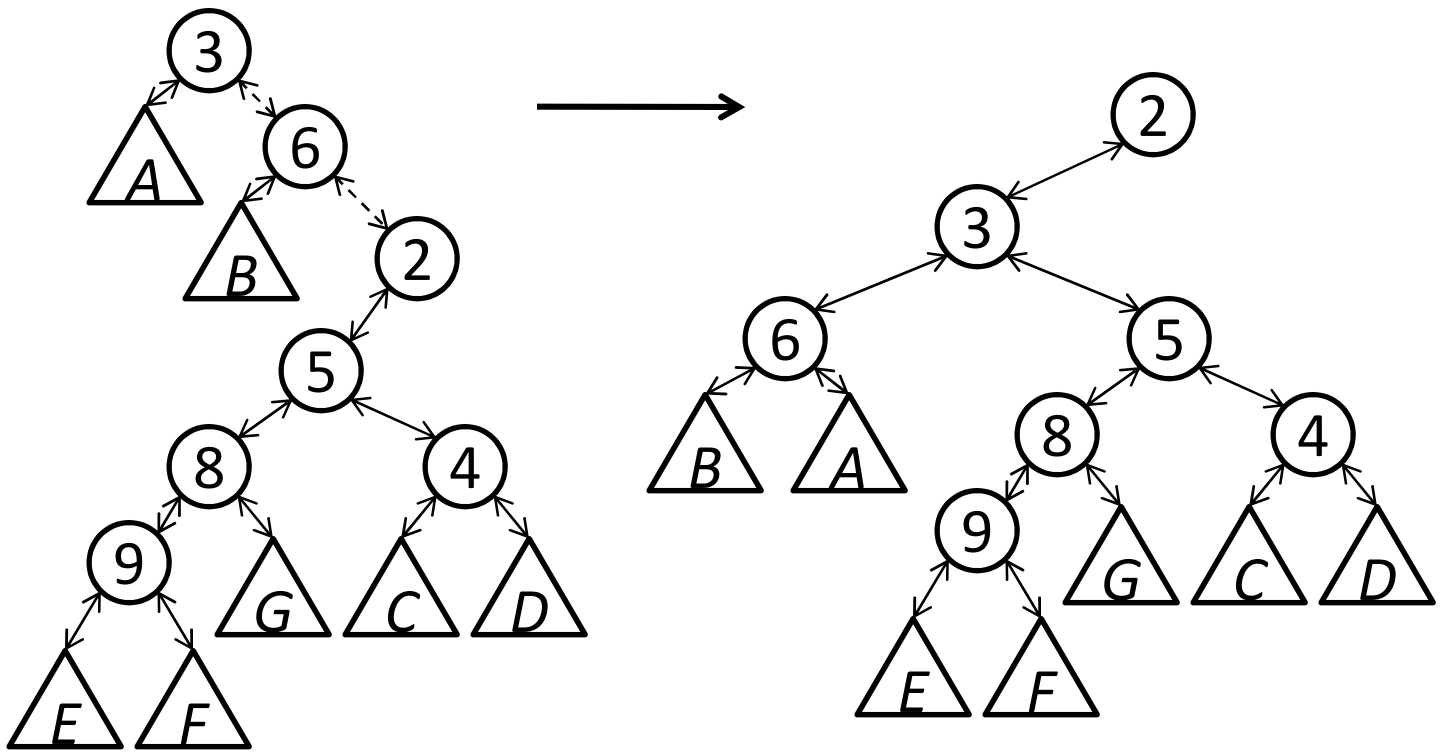
(ii) zig step, if one occurs, is at the bottom, not the top;

(iii) no zig-zag steps.

→ Use the Φ used to analyze splay trees!

Bottom-up view of delete-min




3
A
6
B
2
5
8
4
9
G
C
D
E
F
2
3
6
5
B
A
8
4
9
G
C
D
E
F

$\Phi(x) = \lg(s(x))$ $\quad 0 \le \Phi(x) \le \lg n$

$\Phi(T) = \Sigma\Phi(x)$ $\quad 0 \le \Phi(T)$

Make-heap, find-min take O(1) actual time,

$\Delta\Phi = 0 \rightarrow$ O(1) amortized time

Insert, *meld*, *decrease-key* take O(1) actual time,

$\Delta\Phi \le 2\lg n$ (at most two nodes increase in Φ)

$\rightarrow$ O($\lg n$) amortized time

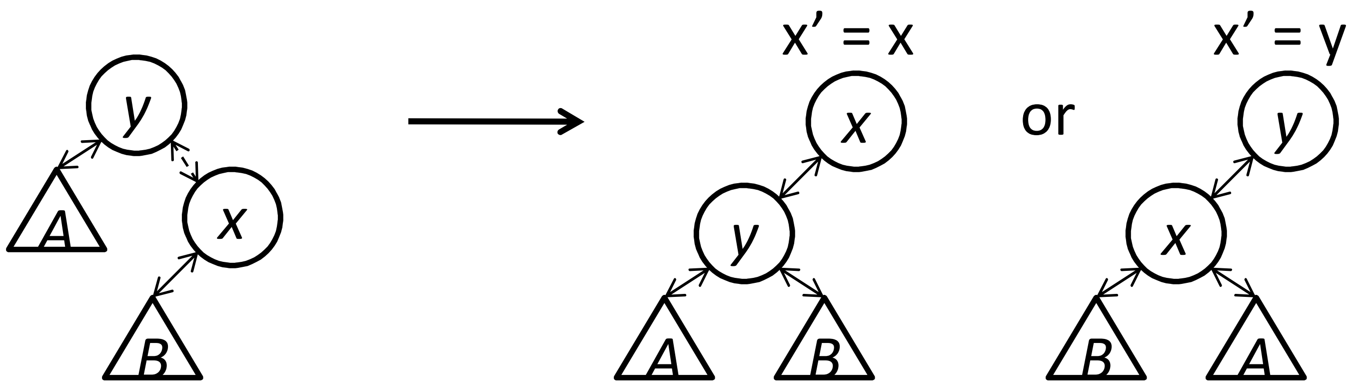
Delete-min: time = 1 + #links. Take bottom-up view. Let x = root of bottom half tree, x' = root after link step (one or two links).

zig: 1 link, occurs at most once

$$\Delta\Phi(T) = \Phi'(x) + \Phi'(y) - \Phi(x) - \Phi(y)$$

$$\leq \Phi'(x') - \Phi(x) \leq 3(\Phi'(x') - \Phi(x))$$

$\rightarrow$ amortized time $\leq 3(\Phi'(x') - \Phi(x)) + 1$



zig-zig: 2 links

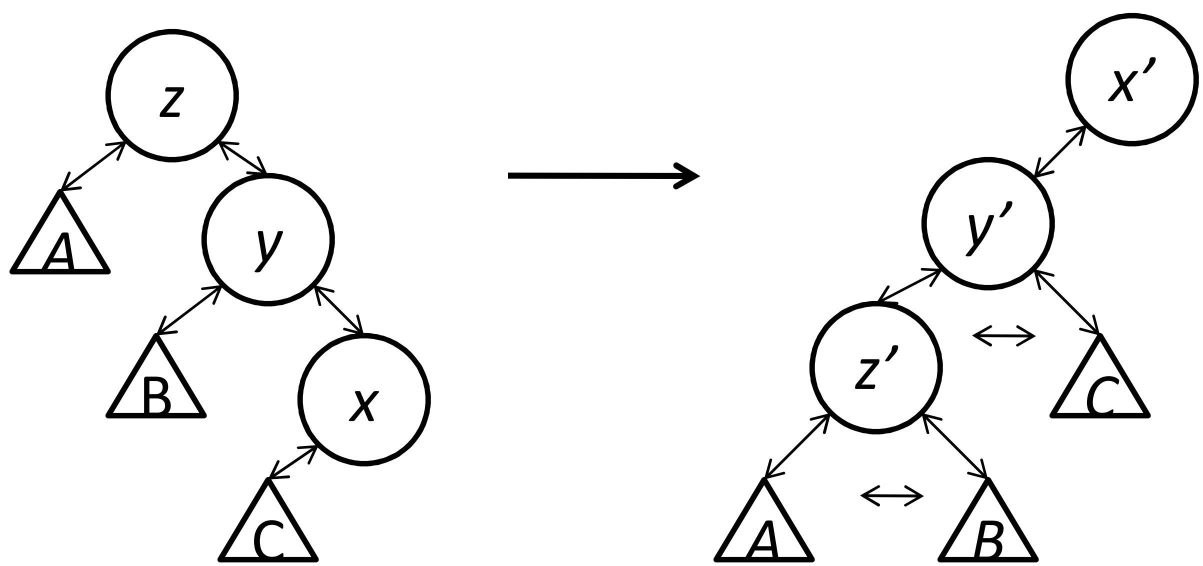
$$\Delta\Phi(T) = \Phi'(y') + \Phi'(z') - \Phi(x) - \Phi(y)$$

$$= \Phi'(y') + \Phi'(z') + \Phi(x) - 2\Phi(x) - \Phi(y)$$

$$\leq \Phi'(x') + 2\Phi'(x') - 2 - 3\Phi(x) \text{ by } (*)$$

$$= 3\Phi'(x') - 3\Phi(x) - 2$$

$\rightarrow$ amortized time $\leq 3(\Phi'(x') - \Phi(x))$



Sum over all link steps. The sum telescopes, since x' in one step is x in the next step, giving an amortized time for the *delete-min* of at most $3(\Phi_F(x_F) - \Phi_0(x_0)) + 2$, where x_0 is the initial x and x_F is the final x, the root of the half tree after all links.

→ amortized time of *delete-min*, *delete* is O(lg*n*)

O(lg*n*) is not a tight bound for decrease-key; tight bound is unknown (Ω(lglg*n*))