Chapter 6
Heaps

Introduction

- some systems applications require that items be processed in specialized ways
  - printing
    - may not be best to place on a queue
    - some jobs may be more <u>important</u>
    - small 1-page jobs should be printed before a 100-page job
  - operating system <u>scheduler</u>
    - processes run only for a slice of time
    - queue: FIFO
      - some <u>short</u> jobs may take too long
      - other jobs are more important and should not wait

Heap Model

- specialized queue required
  - heap (priority queue)
  - provides at least
    - insert
    - deleteMin: finds, returns, and removes min (or max)
    - other operations common
  - used in other applications
    - external sorting
    - greedy algorithms
    - discrete event simulation

Heap Implementation

- heaps can be implemented with
  - linked list, with insertions at head
    - insert $O(1)$
    - deleteMin $O(N)$
  - ordered linked list (worse due to number of insertions)
    - insert $O(N)$
    - deleteMin $O(1)$
  - binary search tree
    - insert $O(\log N)$
    - deleteMin $O(\log N)$
    - since only min is deleted, tree will be unbalanced
    - overkill since other included operations not required

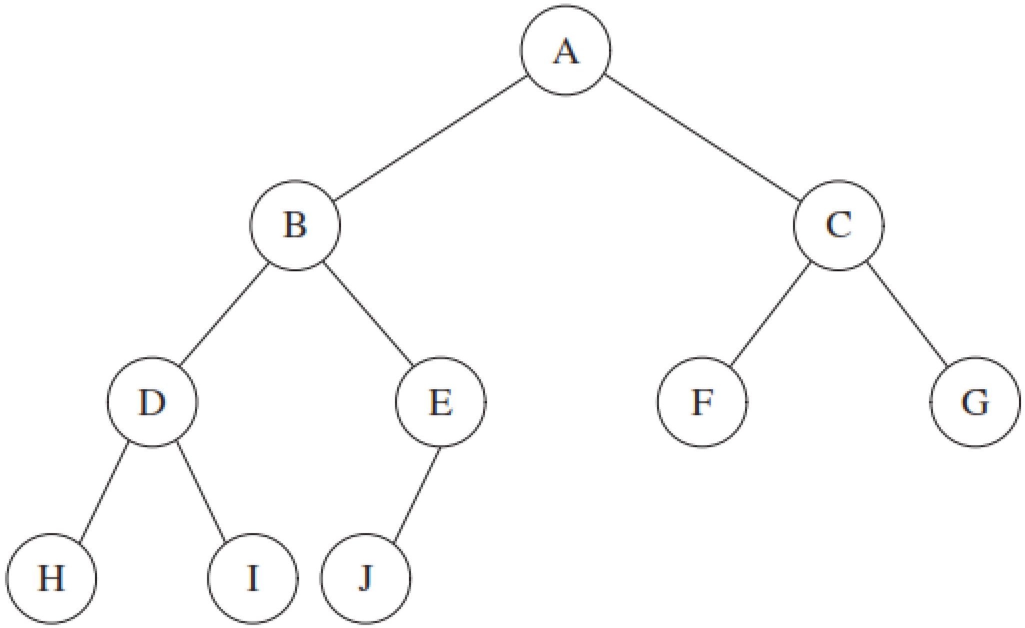
Binary Heap

- binary heap common
  - simply termed *heap*
- two properties
  - <u>structure</u>
  - heap <u>order</u>
- operations can <u>destroy</u> one of the properties
  - operation must continue until heap properties have been restored, which is typically simple

Binary Heap

- structure property
  - completely filled
    - except <u>bottom</u> level, which is filled from left to right
    - complete binary tree has between 2^h and $2^{h+1} - 1$ nodes
    - height: $\lfloor \log N \rfloor$
  - can be represented with an array (no <u>links</u> necessary)
    - array position i
      - left child in $2i$
      - right child in $2i + 1$
      - parent in $\lfloor i/2 \rfloor$
    - operations simple
    - maximum heap size must be <u>known in advance</u>

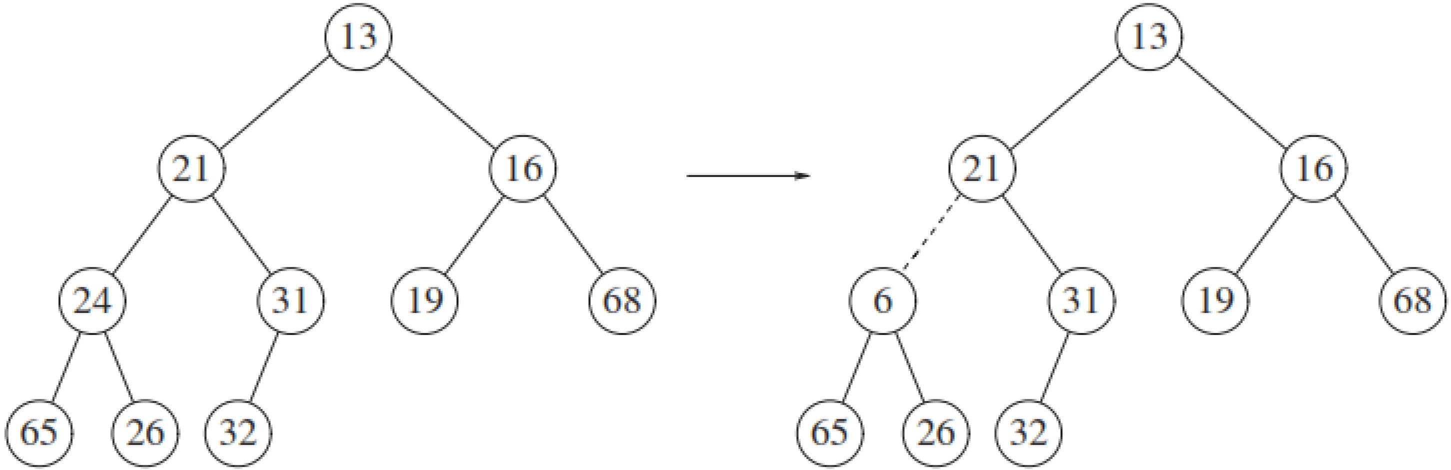
Binary Heap



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Binary Heap

- heap-order property
  - allows operations to be performed quickly
  - want to be able to find minimum quickly
    - smallest element at <u>root</u>
  - any <u>subtree</u> should also be a heap
    - any node should be <u>smaller</u> than all its descendants

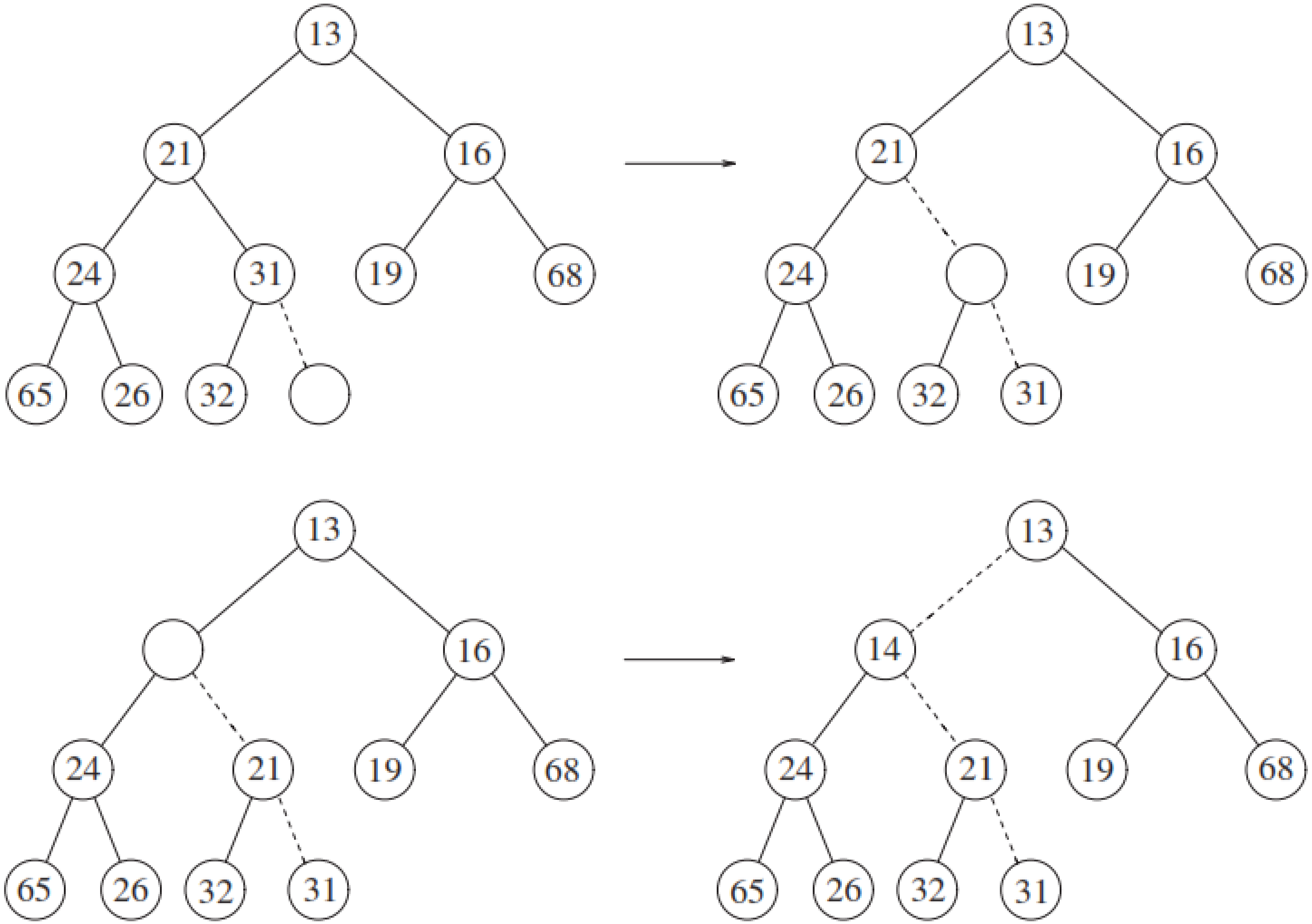


Heap Operations

- two main heap operations
  - insert
  - deleteMin
- insert
  - need to maintain structure property
  - create <u>hole</u> in next available location (at bottom of tree)
  - place new value there if possible
  - otherwise, slide <u>parent</u> into hole and bubble up hole
  - continue until new value can be inserted
  - termed <u>*percolate up*</u> strategy

Binary Heap

– example: insert 14



Heap Operations

- insert (cont.)
  - could have used repeated swaps, but a swap requires <u>three</u> assignments
  - if element percolated up d levels
    - swap method: $3d$ assignments
    - non-swap method: $d + 1$ assignments
  - if new element is smaller than all others in heap, hole will percolate to the <u>root</u>
  - hole will be at index 1 and we will break out of the loop
    - if extra check for 1 in loop, adds unnecessary time
    - could place a copy of new value in position <u>0</u>

Heap Operations

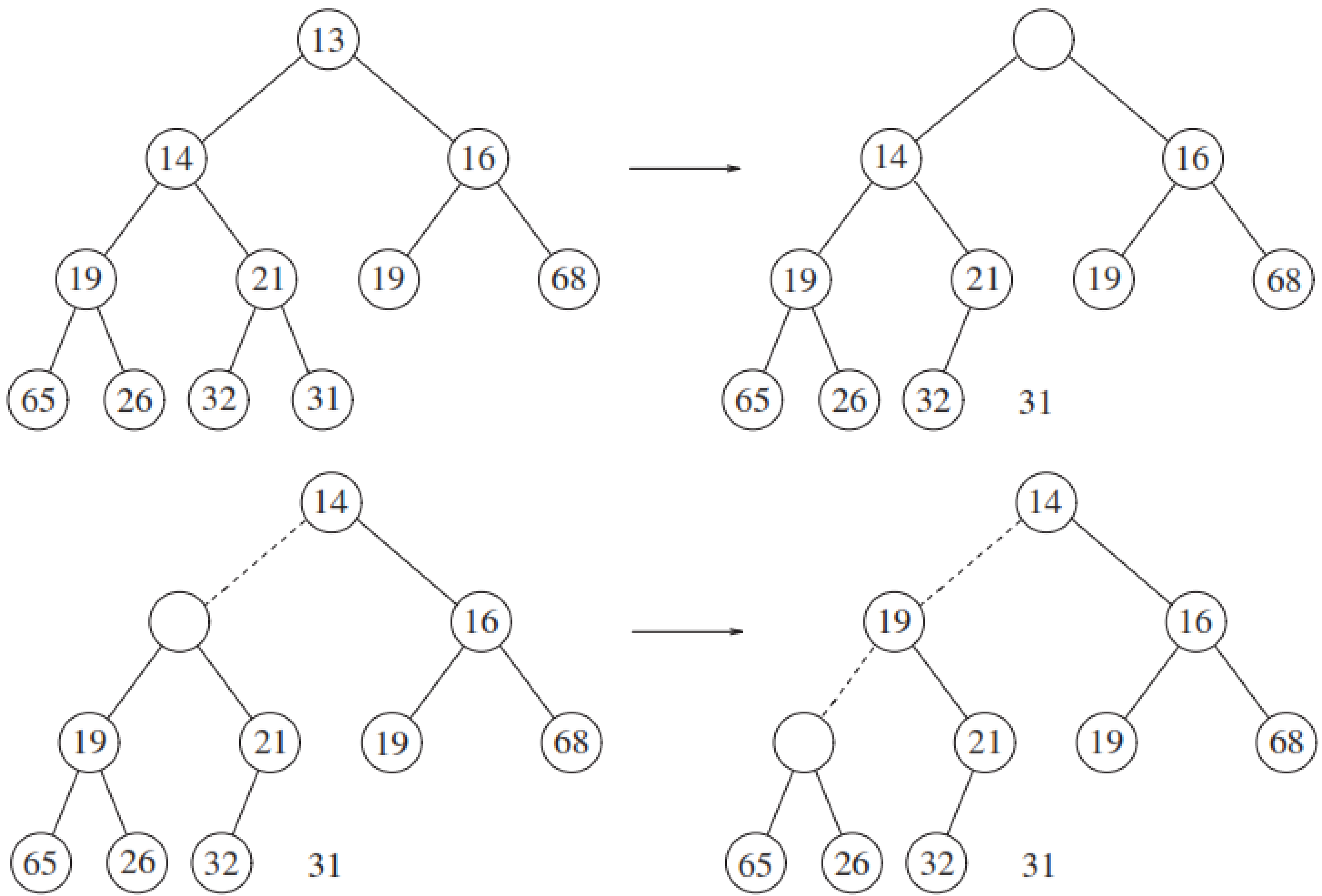
- insert (cont.)
  - could require as much as $O(\log N)$
  - on average, percolation terminates <u>early</u>
    - on average 2.607 comparisons are required

Heap Operations

- deleteMin
  - finding minimum easy
  - <u>removing</u> minimum more difficult
    - hole is created at root
    - <u>last</u> element in complete binary tree must move
    - if last element can be placed in hole, done
    - otherwise, slide hole’s <u>smaller</u> child into hole
      - hole slides down one level
      - repeat until last element can be placed in hole
  - worst case: $O(\log N)$
  - average case: $O(\log N)$

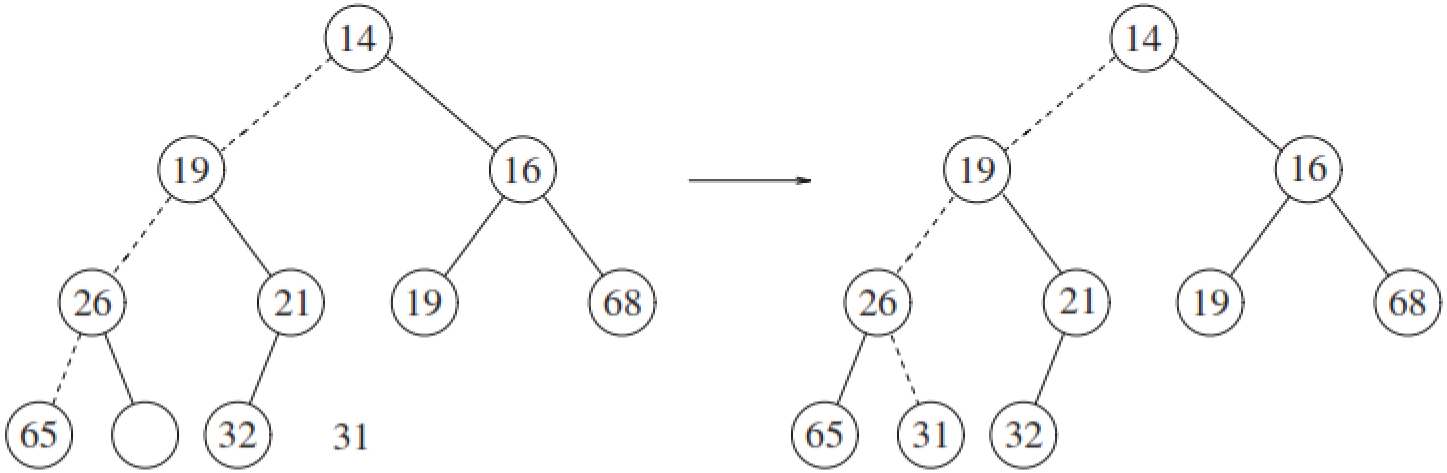
Binary Heap

– example: deleteMin (13)



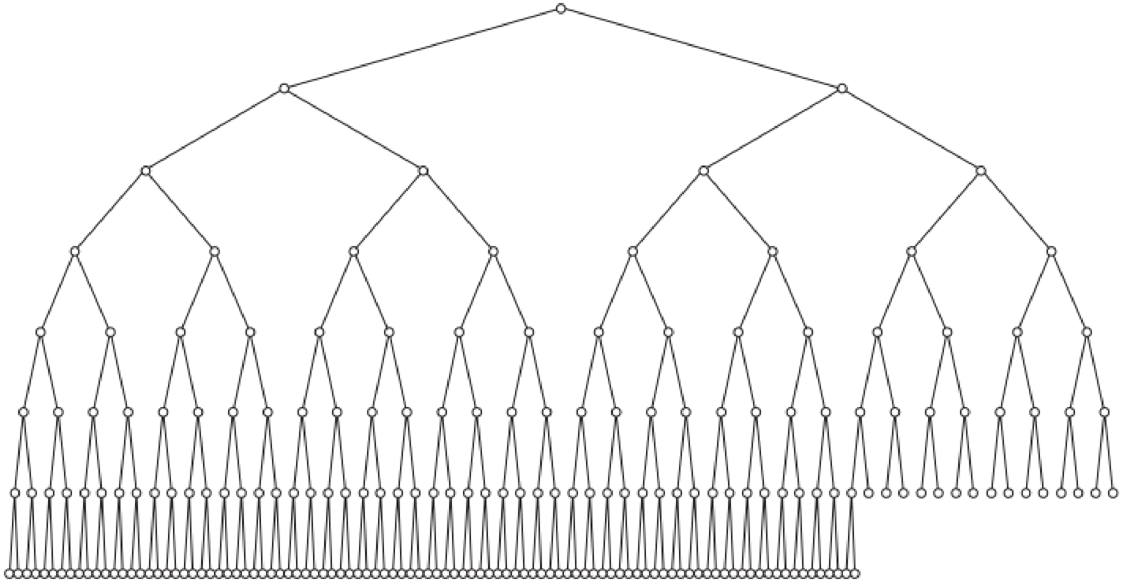
Binary Heap

- example: deleteMin (13) (cont.)



Heap Operations

- other heap operations
  - finding minimum <u>fast</u>
  - finding maximum not possible without <u>linear</u> scan through entire heap
    - maximum in one of the leaves
  - could use separate data structure, such as <u>hash table</u>



Heap Operations

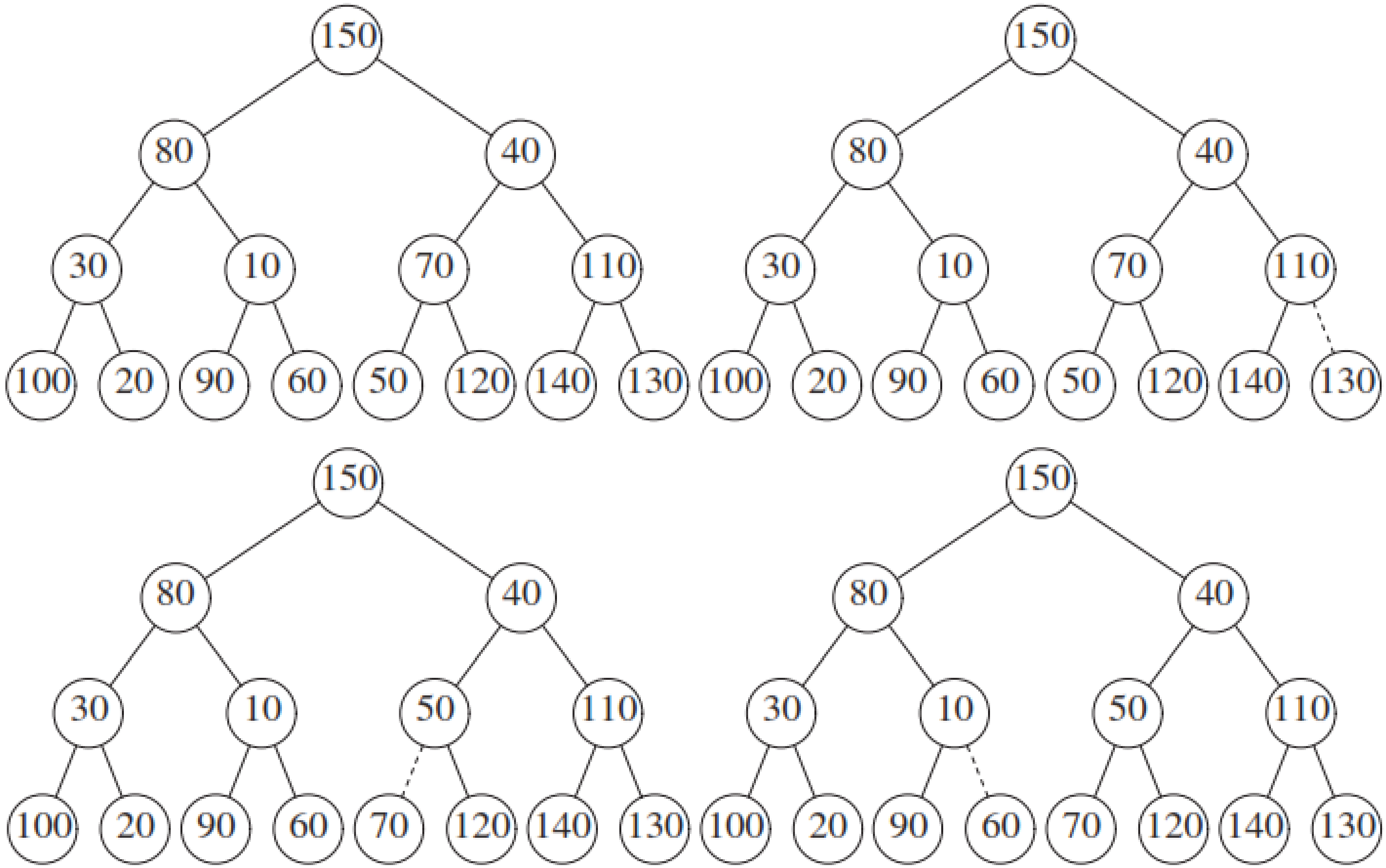
- other heap operations (cont.)
  - decreaseKey
    - lowers value at position p by given amount
      - percolate up
    - example: change process <u>priority</u> for more run time
  - increaseKey
    - increases value at position p by given amount
      - percolate down
    - example: drop process priority if taking <u>too much time</u>
  - remove
    - decreaseKey to root, then deleteMin
    - example: process is <u>terminated early</u> by user

Heap Operations

- other heap operations (cont.)
  - buildHeap
    - place N items into the tree in any <u>order</u>
      - maintains <u>structure</u> property
      - use percolateDown(i) from node i

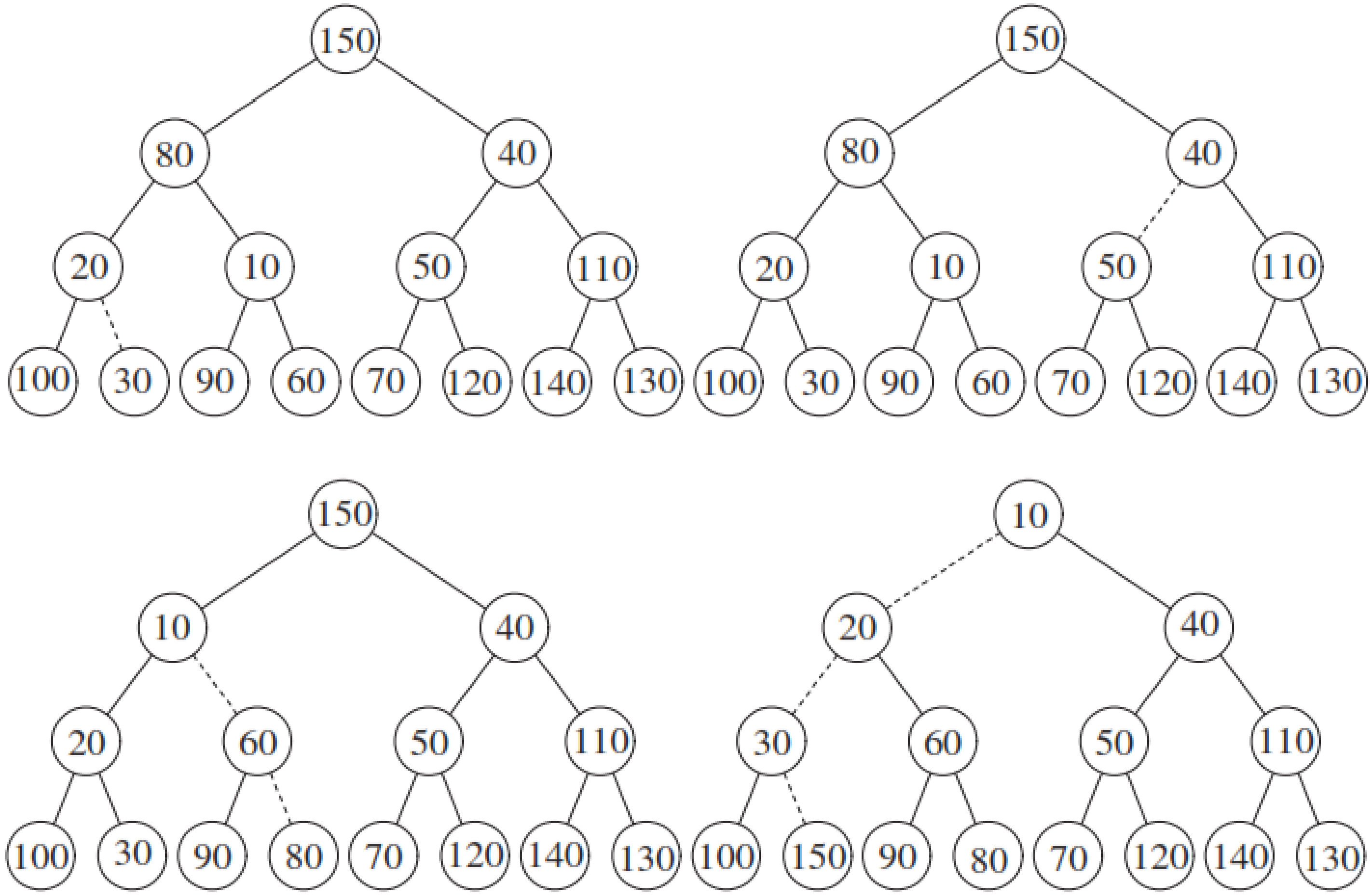
Binary Heap

- example: start with random placement
  - start with percolateDown(7)
  - each dashed line represents 2 comparisons



Binary Heap

– example: (cont.)



Heap Applications

- selection problem
  - from a list of N elements, find kth largest
  - original algorithm
    - sort list and index kth element
    - with simple sort, $O(N^2)$
  - alternative algorithm
    - read k elements into array and sort them
    - smallest is in kth position
    - other elements processed one by one, placing them into the array
    - running time $O(N \cdot k)$
      - if $k = \lceil N/2 \rceil$, $O(N^2)$

Heap Applications

- selection problem (cont.)
  - if heap is used
    - build heap of N elements
    - perform k deleteMin operations
    - last element extracted is kth smallest element
    - if $k = \lceil N/2 \rceil$, $\Theta(N \log N)$
  - another algorithm using heap
    - as in previous algorithm, but put k elements in heap
    - other elements processed one by one, placing them into the array
    - find smallest in array
    - running time $\Theta(N \log N)$

Heap Applications

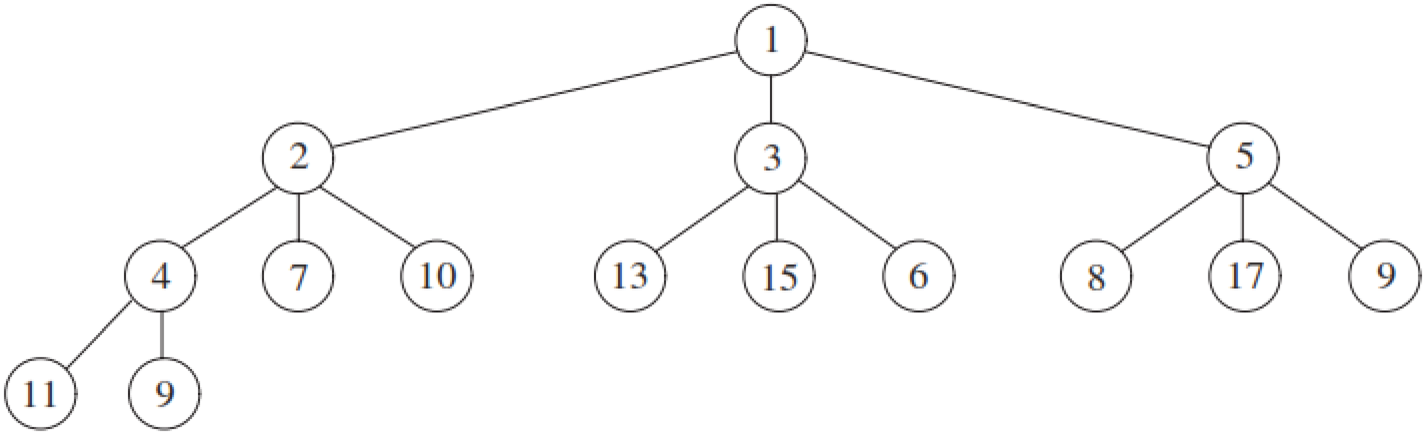
- event simulation
  - bank where customers arrive and wait until one of k tellers is available
    - customer <u>arrival</u> and <u>service time</u> based on probability distribution function
    - compute statistics on the length of time a customer must wait, or the length of the line
    - need to consider event that will occur in the least amount of time
      - heap can be used to order events

d-Heaps

- d-heap
  - same as binary heap, but all nodes have d children
  - tends to be more shallow than binary heap
  - running time reduced to $O(\log_d N)$
  - deleteMin more expensive since more comparisons required
  - useful when heap is too large to fit entirely into main memory
  - 4-heaps may outperform 2-heaps (binary heaps)

d-Heaps

- example: d-heap with $d = 3$



Leftist Heaps

- one weakness of heaps so far is <u>combining</u> two heaps is difficult
- three data structures that can help
  - <u>leftist</u> heaps
  - <u>skew</u> heaps
  - binomial queues

Leftist Heaps

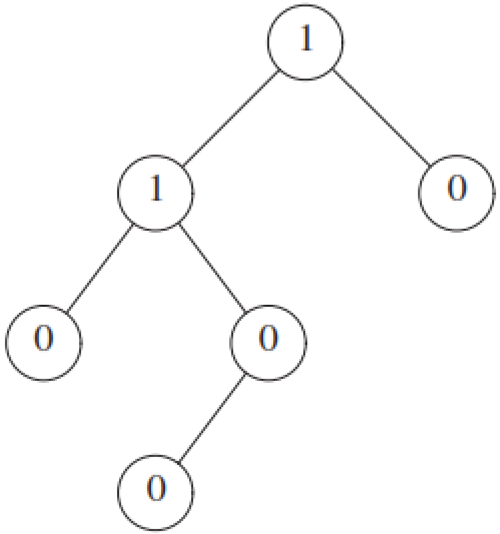
- leftist heaps
  - can be difficult to design a data structure for merging that uses an array, but runs efficiently
    - linked data structure therefore required
  - leftist heap
    - structure and ordering properties of binary heaps
    - difference is that heap is not perfectly balanced
      - very unbalanced is desired

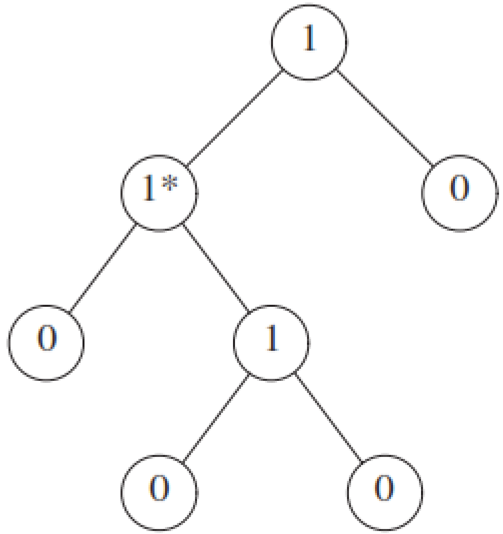
Leftist Heaps

- leftist heaps (cont.)
  - null path length (npl): length of shortest path from current node to a node without <u>two</u> children
    - npl of a node with zero or one child is 0
    - npl of a null pointer is -1
  - npl of each node is 1 more than the minimum of the null path lengths of its <u>children</u>
  - leftist heap property
    - npl of the left child is at least as large as that of the right child
    - biases tree to <u>deeper</u> left subtree

Leftist Heaps

- example: leftist heap





Leftist Heaps

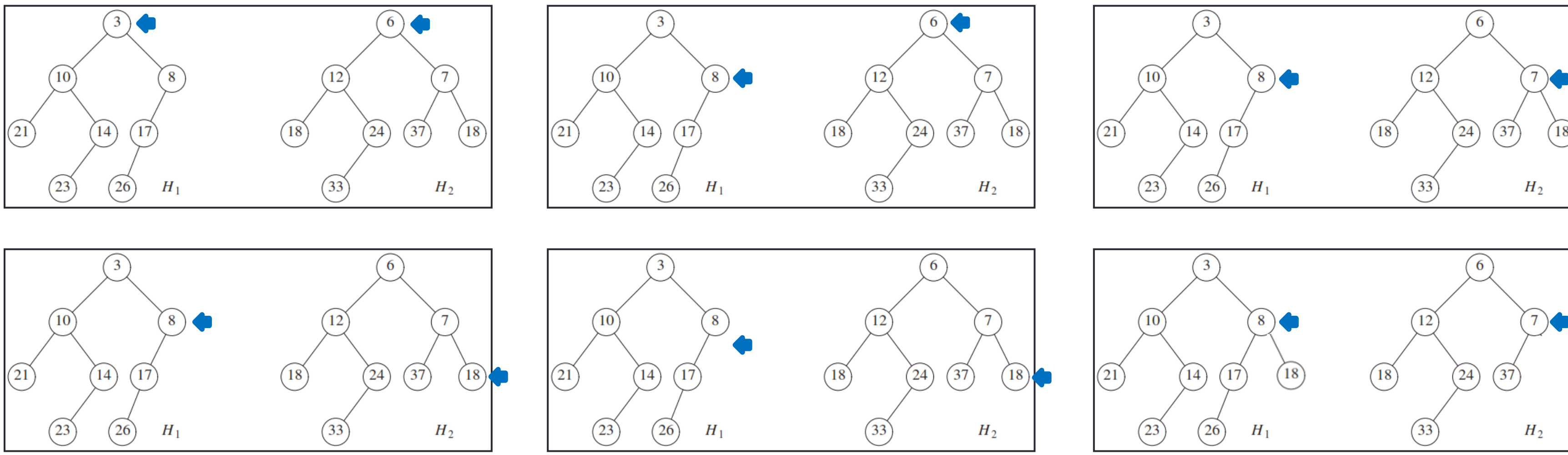
- leftist heaps operations
  - all work should be done on the right path, which is guaranteed to be short
  - <u>inserts</u> and <u>merges</u> may destroy the leftist heap property
    - not difficult to fix

Leftist Heaps

- leftist heaps operations (cont.)
  - merging
    - insertion special case of merging a 1-node heap with a larger heap
    - if either of the two heaps is <u>empty</u>, return the other heap
    - otherwise, compare <u>roots</u>
    - <u>recursively</u> merge heap with the larger root with the right subheap of the heap with the smaller root

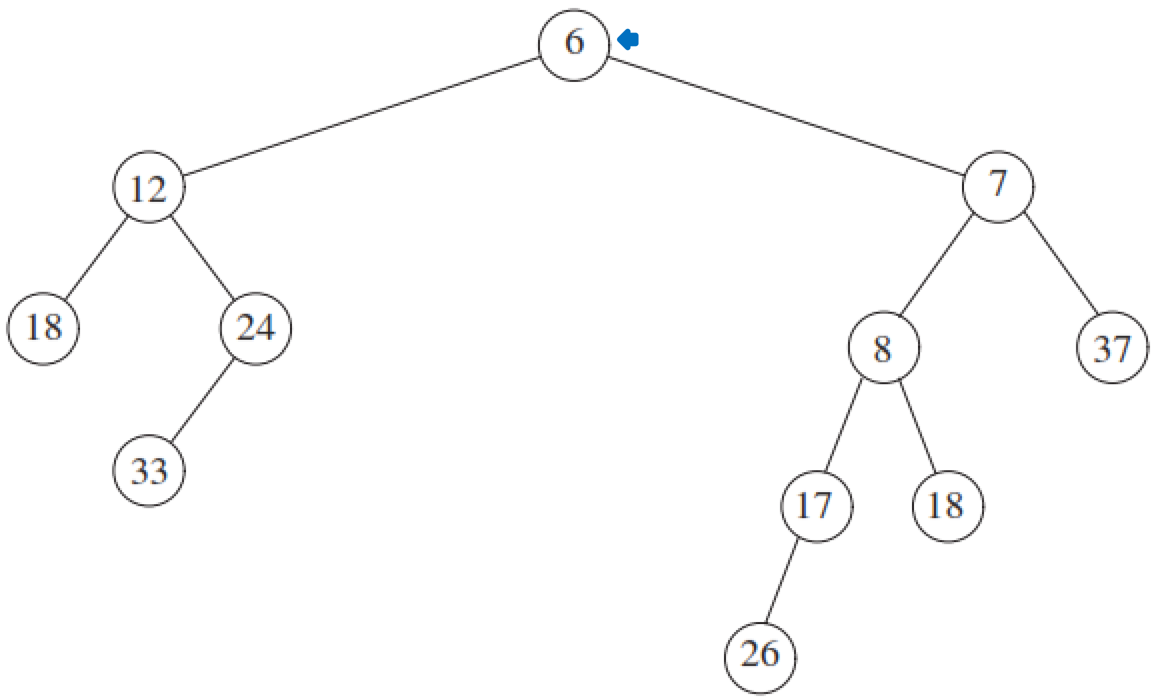
Leftist Heaps

- leftist heaps operations (cont.)
  - merging example
    - start comparing at roots; take right branch of smaller; merge when hitting dead end; recurse back up tree



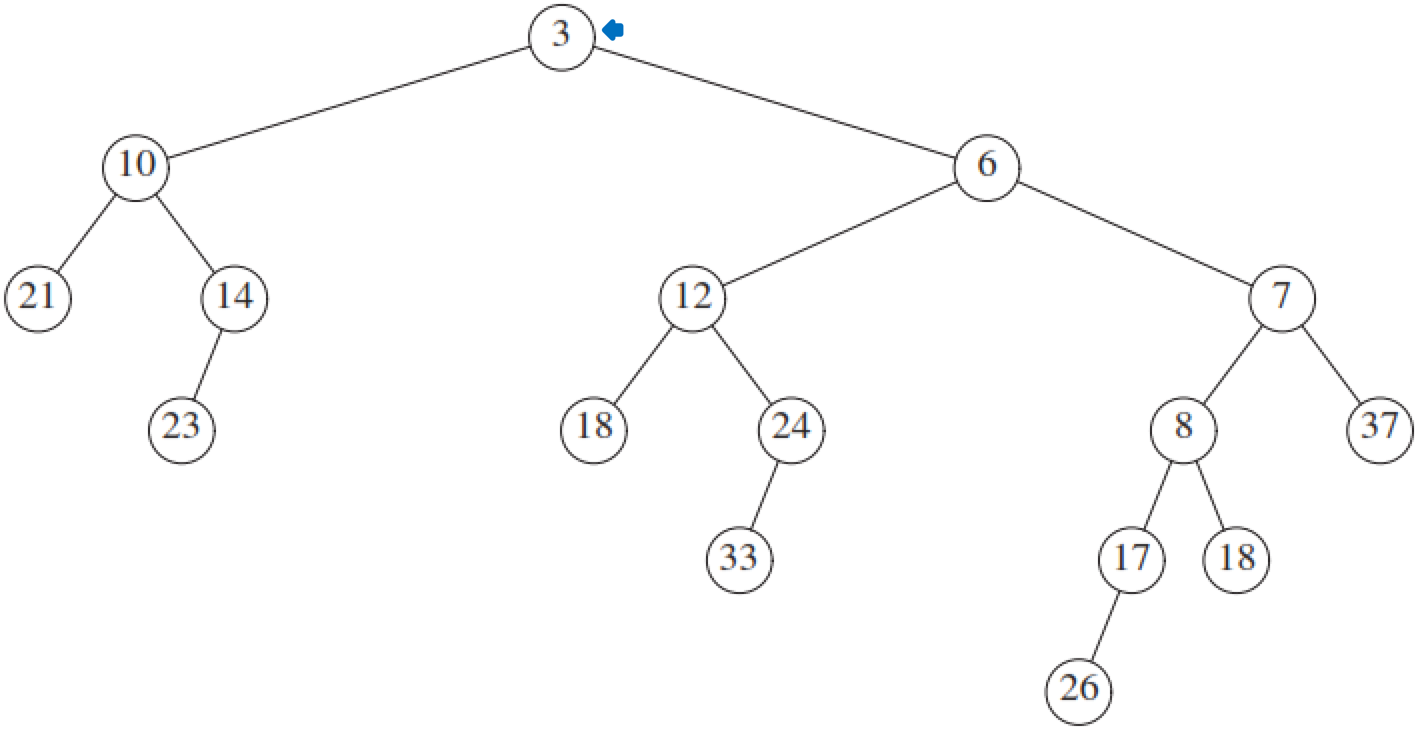
Leftist Heaps

- leftist heaps operations (cont.)
  - merging example
    - must swap children anytime subtree non-leftist



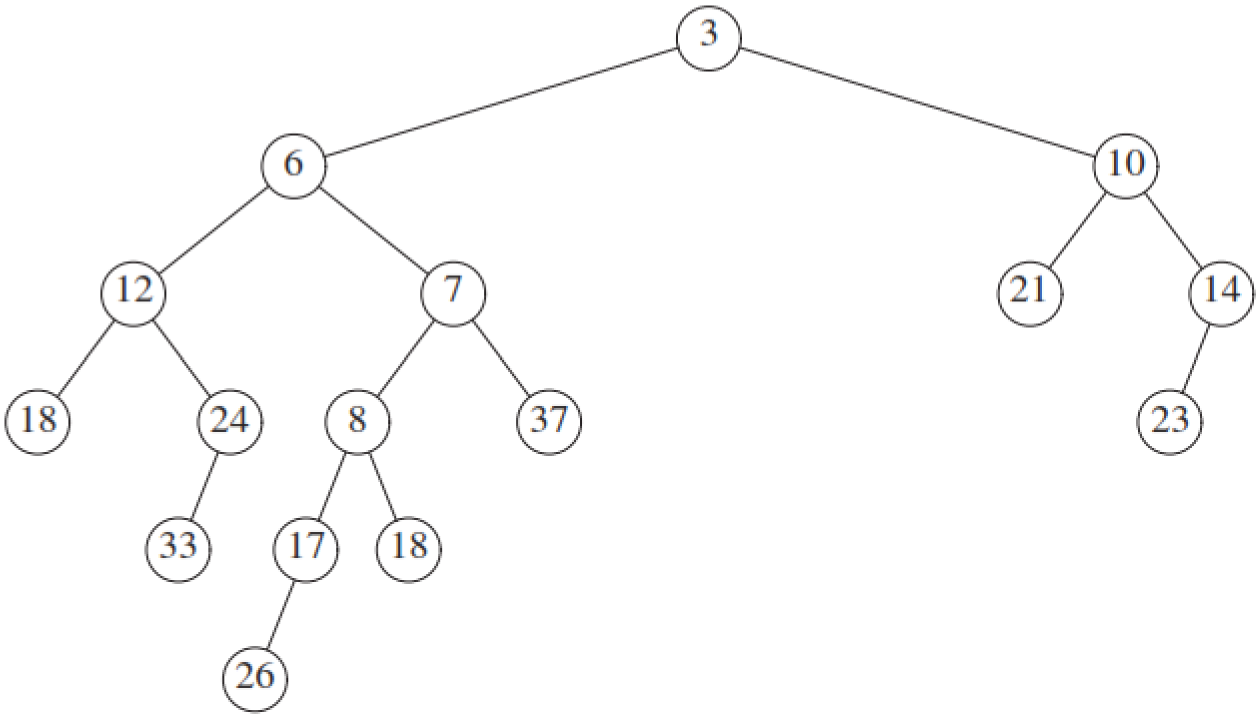
Leftist Heaps

- leftist heaps operations (cont.)
  - merging example
    - result is not leftist: left npl = 1, right npl = 2



Leftist Heaps

- leftist heaps operations (cont.)
  - merging example
    - fix by swapping children



Skew Heaps

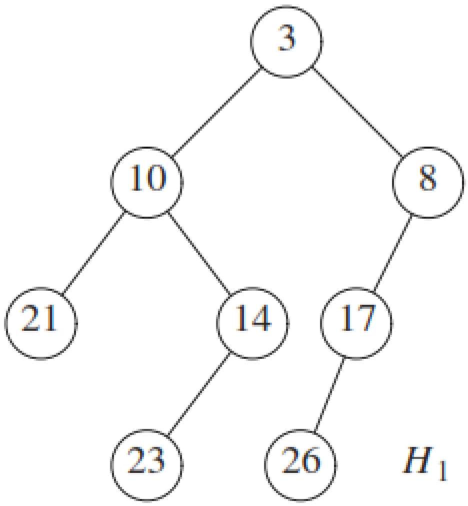
- self-adjusting version of leftist heap
- relationship of skew heap to leftist heap is analogous to splay trees and AVL trees
- skew heaps
  - binary trees with heap order
  - but no structural constraint
    - no information kept about null path length
    - right path can be arbitrarily long
  - worst case of all operations: $O(N)$
  - for M operations, total worst case: $O(M \log N)$, or $O(\log N)$ amortized

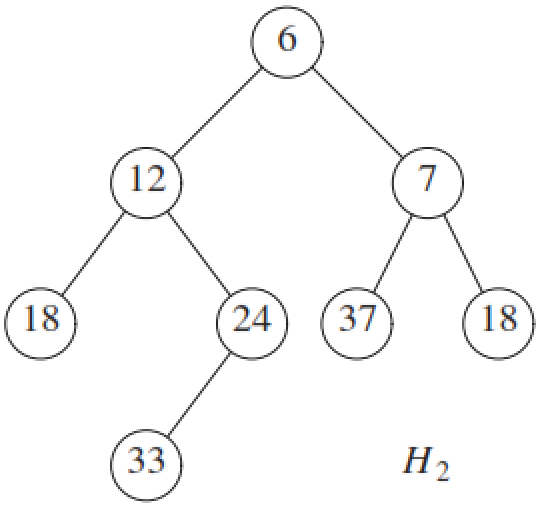
Skew Heaps

- skew heaps (cont.)
  - fundamental operation is merging
  - after merging, for leftists heaps, check both children for structure and swap children if needed
  - in skew heaps, <u>always</u> swap children
    - except <u>largest</u> nodes on right paths do not swap children
    - no extra <u>space</u> required to maintain path lengths
    - no tests required to determine when to swap children

Skew Heaps

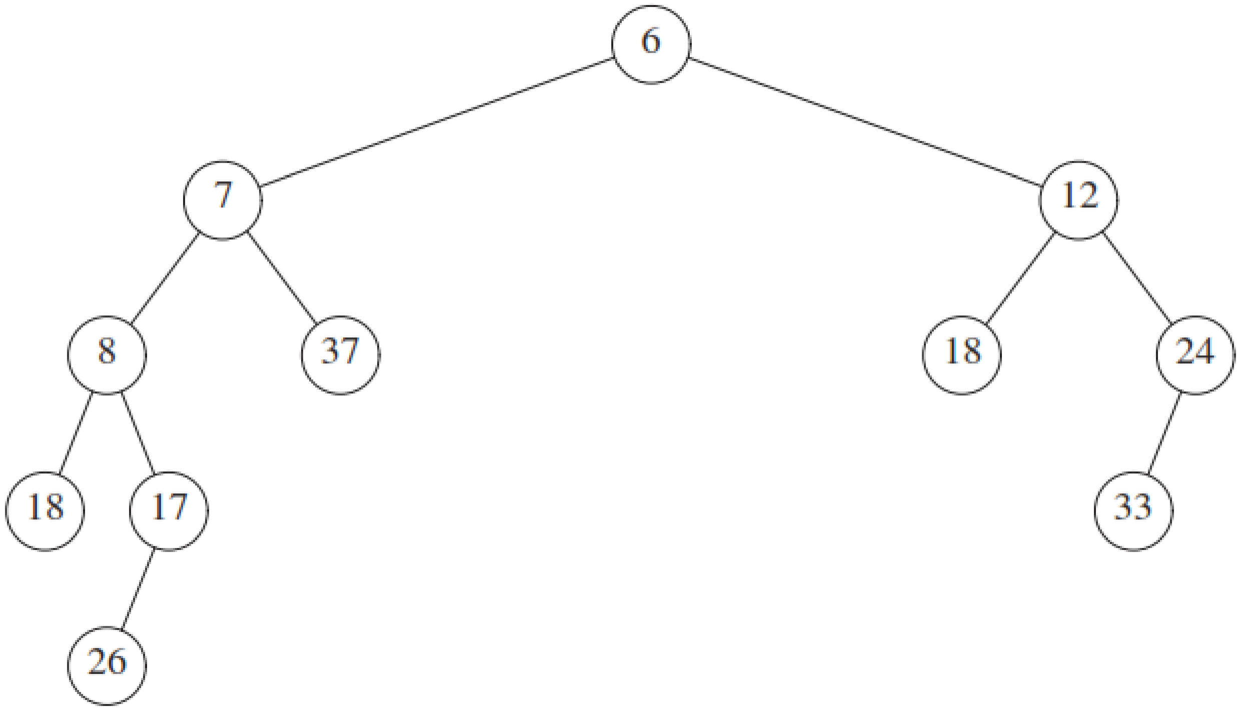
- skew heaps example
  - merge two skew heaps
    - tree with larger root will merge onto tree with smaller root





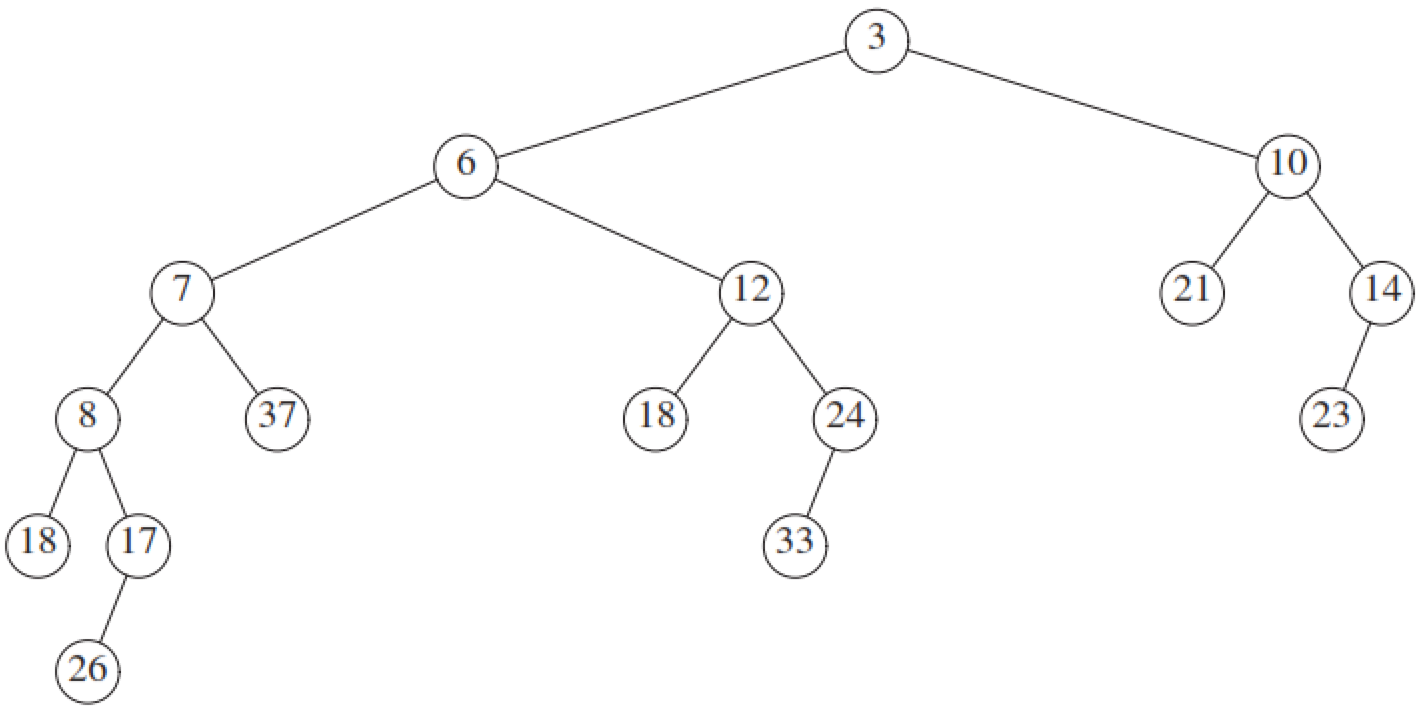
Skew Heaps

- skew heaps example (cont.)
  - recursively merge H_2 with the subheap of H_1 rooted at 8



  - heap happens to be <u>leftist</u>

- skew heaps example (cont.)
  - make this heap the new left child of H_1 and the old left child of H_1 becomes the new right child

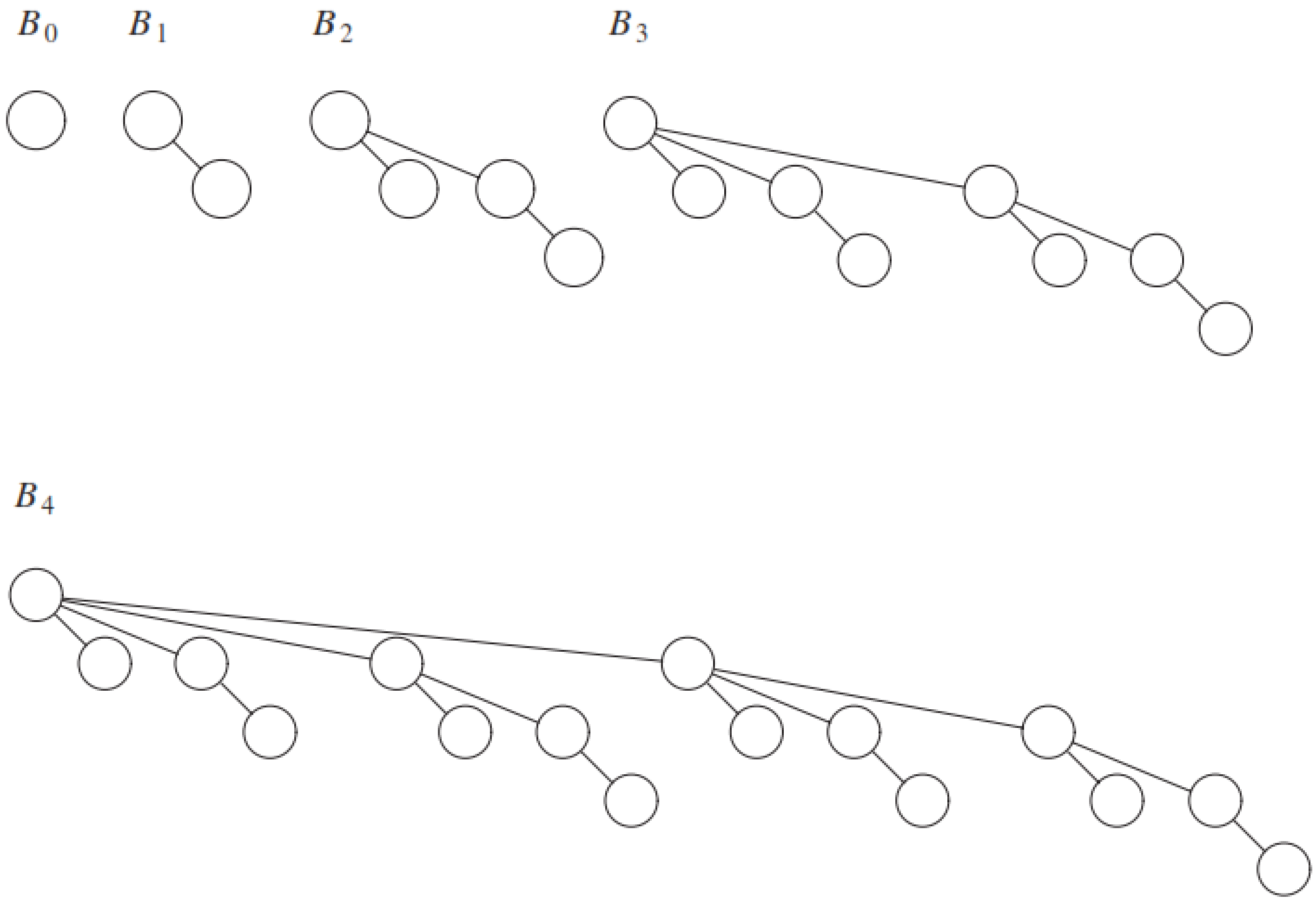


Binomial Queues

- binomial queues
  - keep a collection of heap-ordered trees, known as a forest
  - at most one binomial tree of every height
    - heap order imposed on each binomial tree
    - can represent any priority queue
    - example: a priority queue of size 13 can be represented by the forest B_3, B_2, B_0 or 1101
  - worst case of all operations: $O(\log N)$

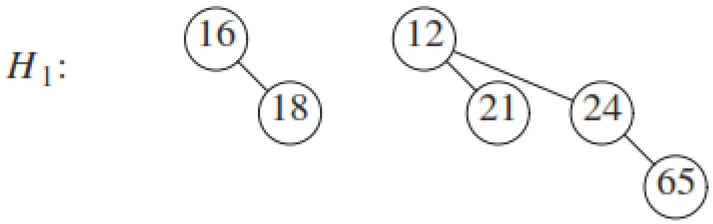
Binomial Queues

– binomial queues example



Binomial Queues

- binomial queue of size 6 example

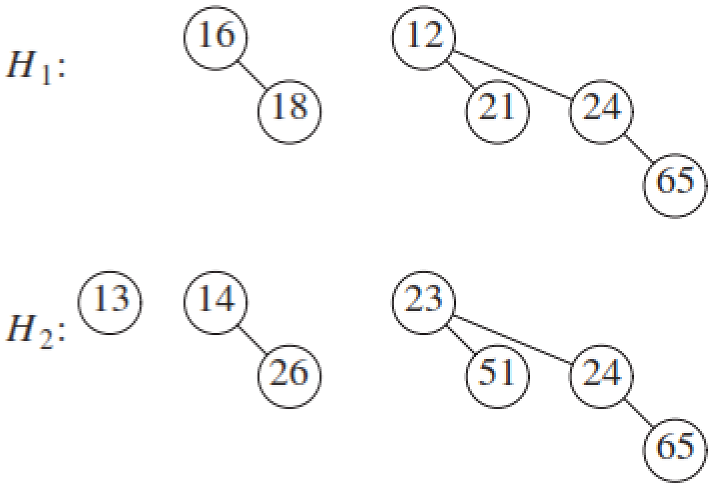


Binomial Queues

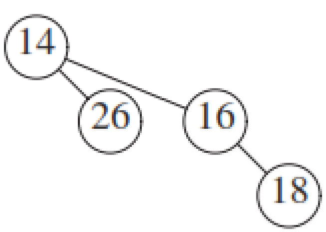
- binomial queue operations
  - minimum element found by scanning <u>roots</u> of all trees
    - found in $O(\log N)$
    - can keep ongoing information to reduce to $O(1)$
  - merging two queues
    - merge takes $O(\log N)$

Binomial Queues

- merge example

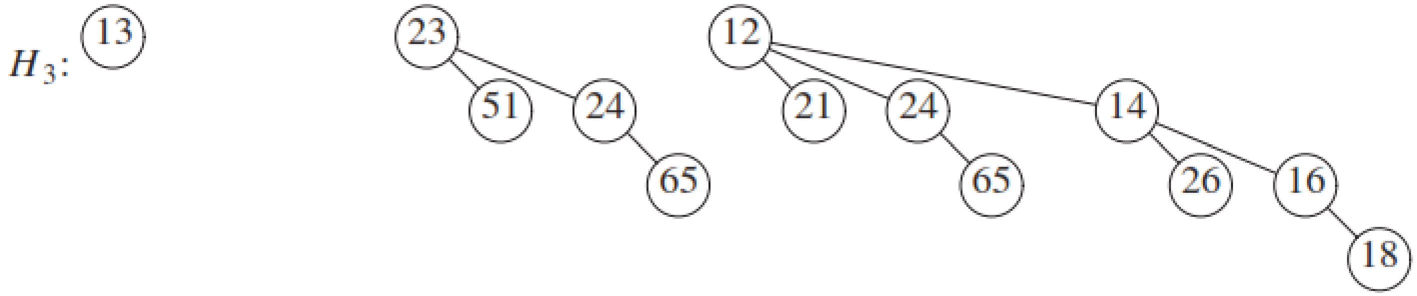


- merge of two B_1 trees



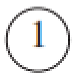
Binomial Queues

- merge example (cont.)
  - now we have three binomial trees of height 3; keep one and merge the other two (with two smallest roots)

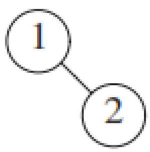


Binomial Queues

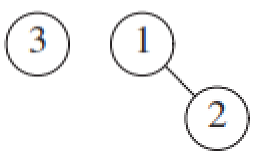
- merge values 1–7 example
  - insert 1



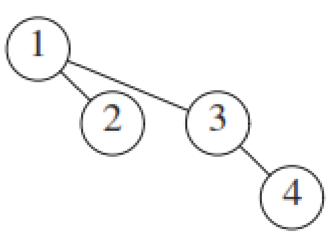
  - insert 2



  - insert 3

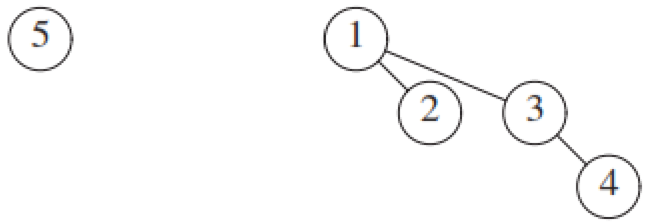


  - insert 4

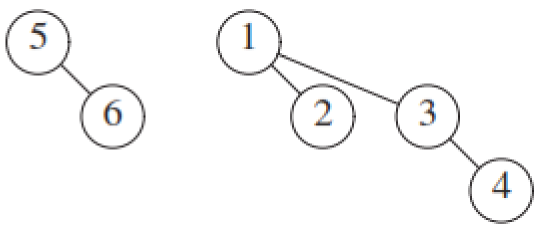


Binomial Queues

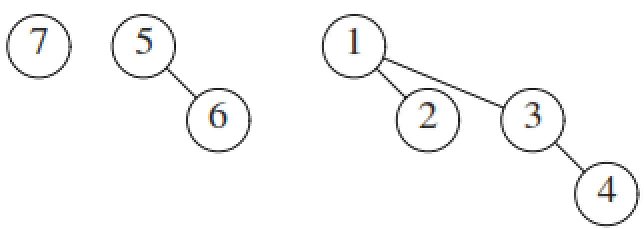
- merge values 1–7 example (cont.)
  - insert 5



  - insert 6



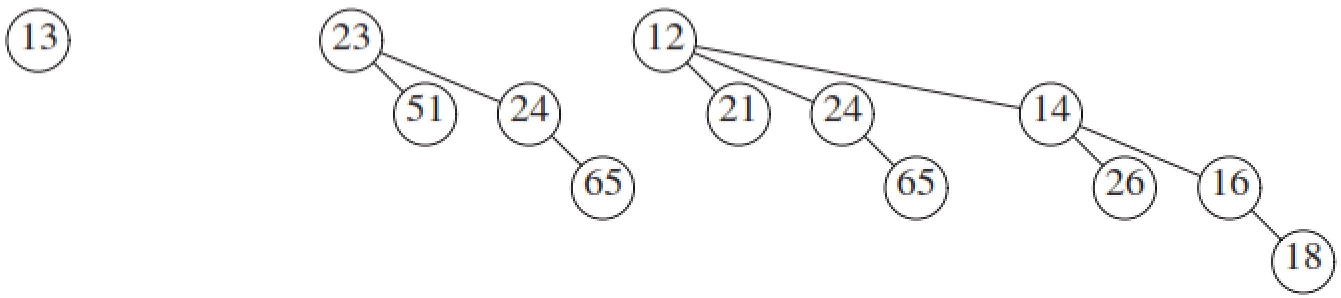
  - insert 7



Binomial Queues

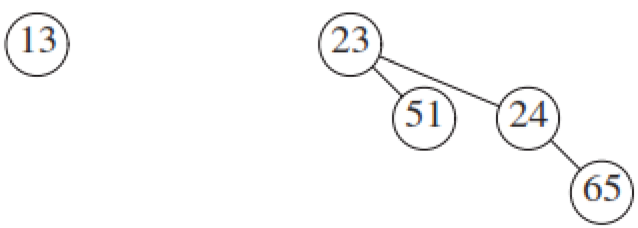
- deleteMin example

H_3:



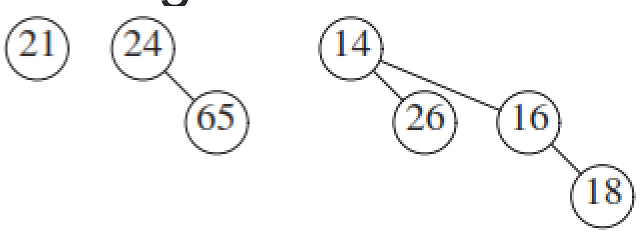
  - separate tree with minimum root from rest of tree

H':



  - remaining trees after removing min

H'':



Binomial Queues

- deleteMin example (cont.)
  - after merge