# Data Structures and Algorithms 2

**Prof. Ahmed Guessoum**
**The National Higher School of AI**

# Chapter 6

# Priority Queues

## Motivating Example

- 3 jobs have been submitted to a printer, the jobs have sizes 100, 10, 1 page(s). Enqueued in this order.
- Average time taken for FIFO service, (100+110+111)/3 = 107 time units
- Average time for shortest-job-first service, (1+11+111)/3 = 41 time units
- Same things for jobs to execute in a multiuser environment, the Operating System scheduler must decide which of several processes to run?
  - ✓ short jobs finish as fast as possible (should have precedence)
  - ✓ Important jobs should also have precedence

➔ Need to have an appropriate data structure for such tasks: a **Priority Queue**

# Priority Queue

- A data structure that allows at least the following two operations:
  - Insert (enqueue); and
  - deleteMin, which finds, returns, and removes the minimum element in the priority queue (equivalent of dequeue in the Queue ADT).
- Different applications of Priority Queues: greedy algorithms and A* (finding the shortest path between two points), load balancing on servers and interrupt handling in Operating Systems, triage systems in emergency departments (medical systems), etc.

# Simple Implementations of PQs

1. Using a Linked list
    1. Insert at the front in O(1)
    2. Find the minimum element in O(n), thus deletion is O(n)
2. Keeping the list always sorted ➔ insertions become expensive (*O*(*N*)) and deleteMins cheap (*O*(1)).

- First solution is probably better, given that there are never more deleteMins than insertions.

# Simple Implementations of PQs (cont.)

3. Using a Binary Search Tree: both *insert* and *deleteMin* take *O(log N)*
   - Deletion of mins (i.e. from left subtree) would make rightsubtree heavier
   - Since the right subtree is random, in the worst case, where the deleteMins have depleted the left subtree, the right subtree would have at most twice as many elements as it should.

- Using a Search Tree is an overkill as it does many other operations that are not required (key-based search, finding kth smallest element, checking if 2 BSTs are identical, etc.)

# Binary Heap

- ***Binary heap***: a very popular implementation of Priority Queues. (By default, *heap* will refer to *Binary Heap*.)

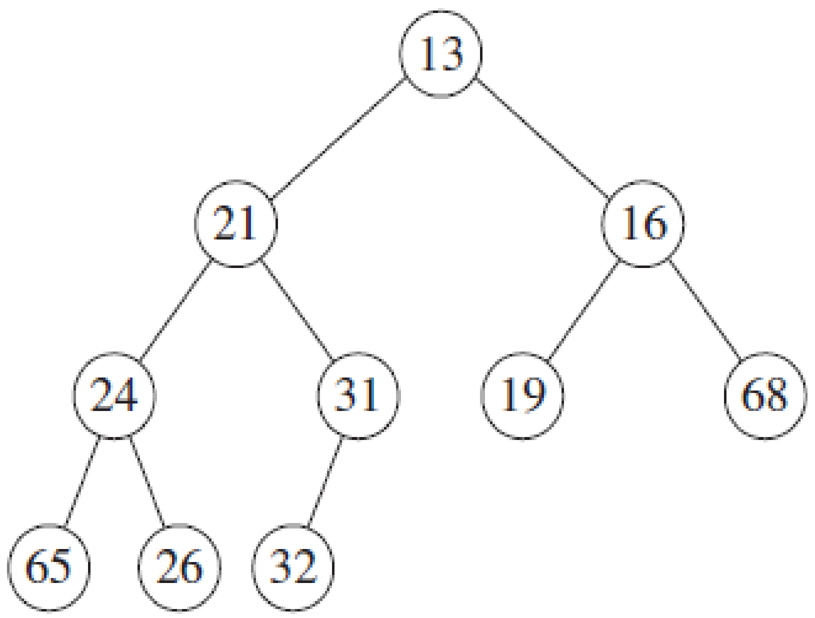
- A Binary heap has a structure property and an order property.

- **Heap structure**: A heap is a binary tree that is completely filled, except possibly for the bottom level, which is filled from left to right (empty nodes are to the right of bottom level)

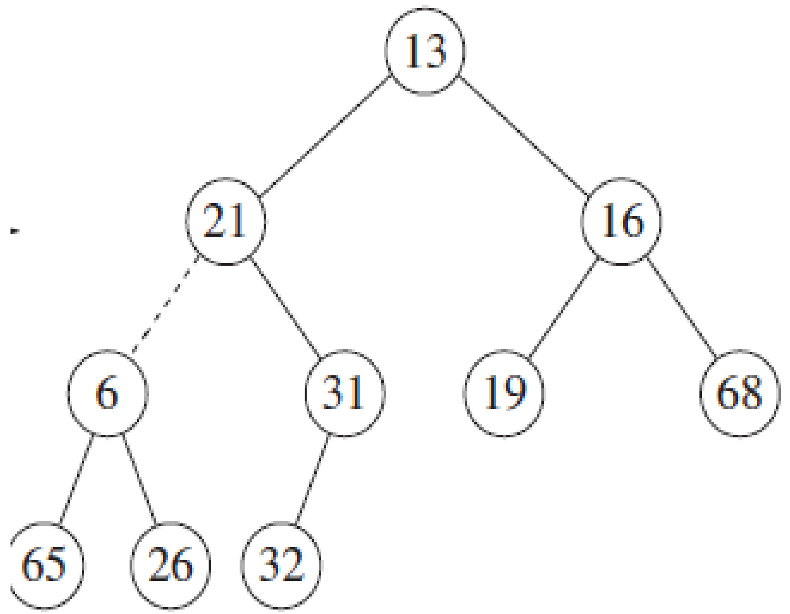
- If such a tree has height h, then it has between $2^h$ and $2^{h+1} – 1$ nodes. ➔ h is O(log n)

- **Heap-order property:** any node should be smaller than all of its descendants

(Smaller than or equal)



Heap

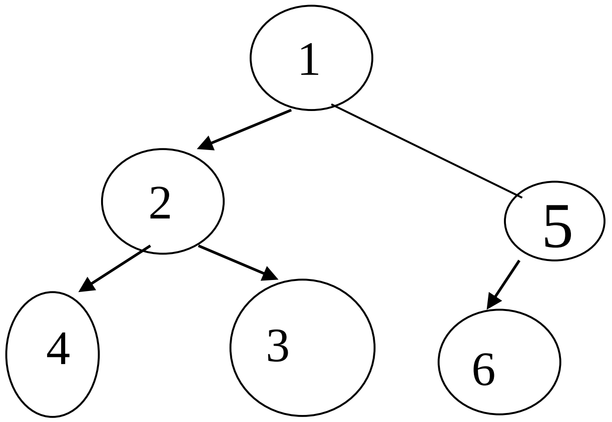


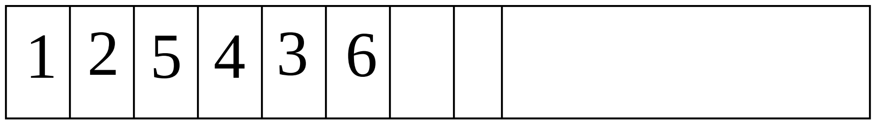
Not Heap

- Because a complete binary tree is so regular, it can be represented in an array and no links are necessary

# Array implementaion

- We start from position 1 in the array.
  - The first element contains the root
  - Left child of element at position j  is at position 2j, right child is at position 2j + 1
  - Parent of element at position j is at ⌊j/2⌋

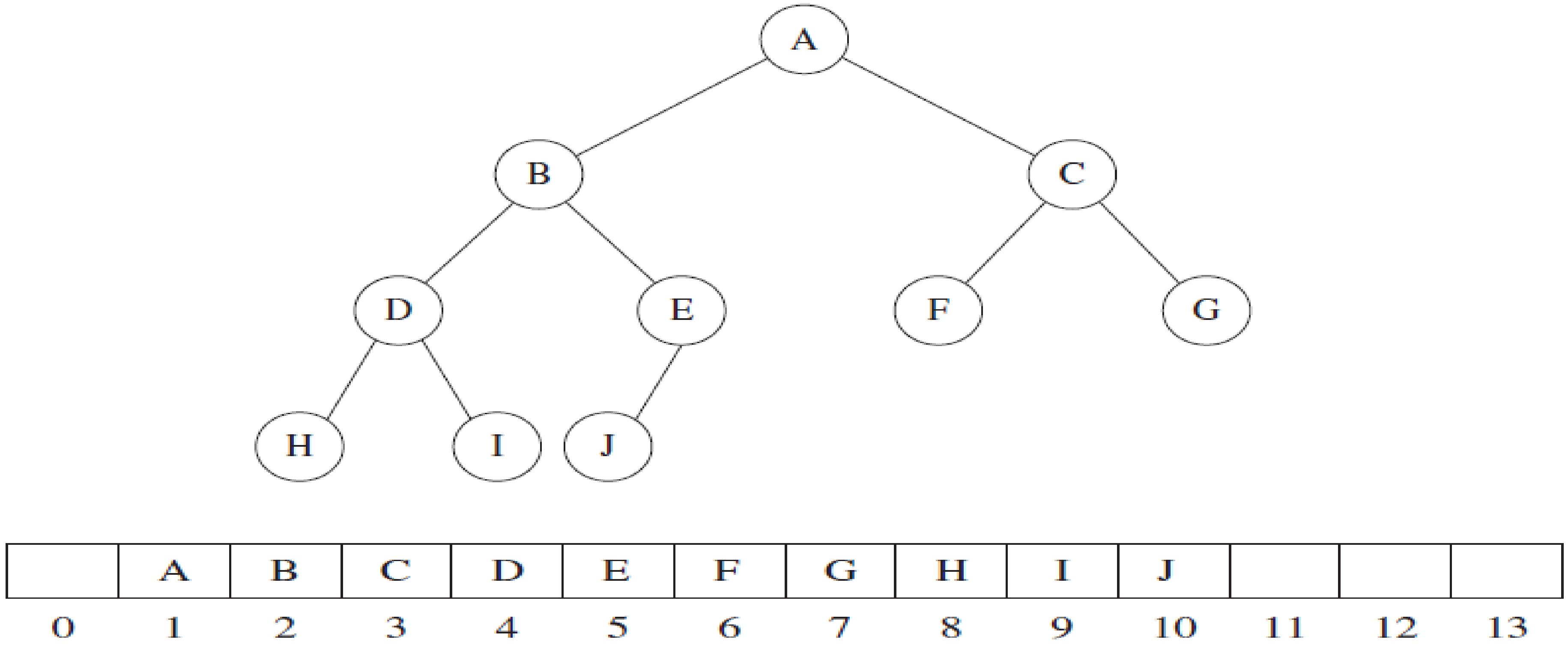




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 4 | 3 | 6 | | | |

- Operations to traverse the tree are extremely simple & very fast
- Only problem with this implementation: an estimate of the maximum heap size is required in advance, but this is not a problem (since we can resize if needed)

In fact, array implementation of the heap looks more like the following! (will get clearer later)

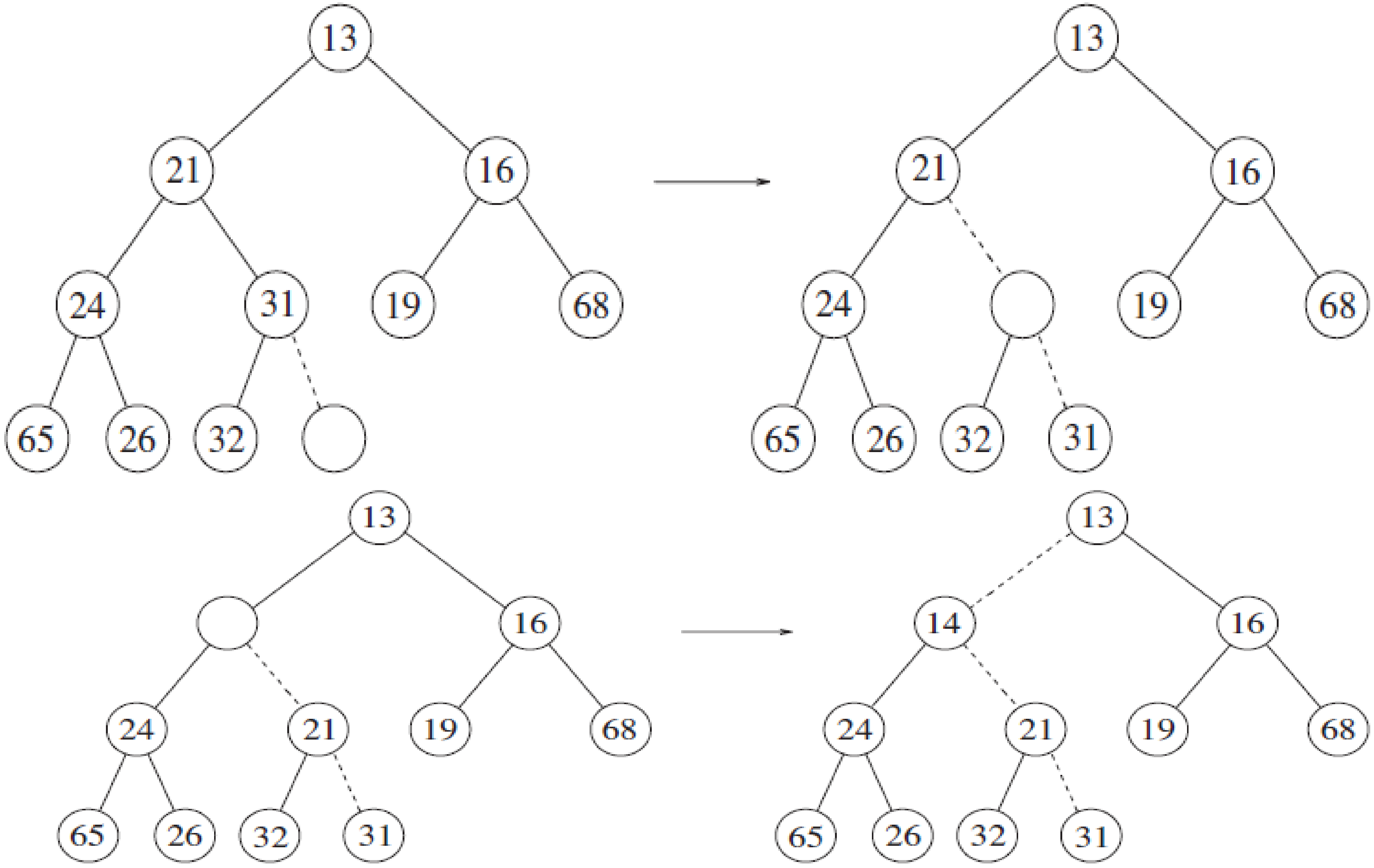


Class interface of Priority Queue

## Inserting an element into a heap

- To insert X
  - Create a *hole* in the next available location,
  - If *X* can be placed in the *hole* without violating heap order, then do so and done.
  - Otherwise, slide the element that is in the *hole*'s parent node into the *hole*, thus bubbling the *hole* up toward the root.
  - Continue this process until *X* can be placed in the *hole*.
- This general strategy is known as a **percolate up.**

## Insert 14 into Binary Heap (below left)



Implementation of Insertion into a Binary Heap

## Performance issues

- We could have used swaps instead of percolation, but each swap takes 3 assignments

➔If an element is percolated up *d* levels, the number of assignments performed by the swaps would be 3*d*.

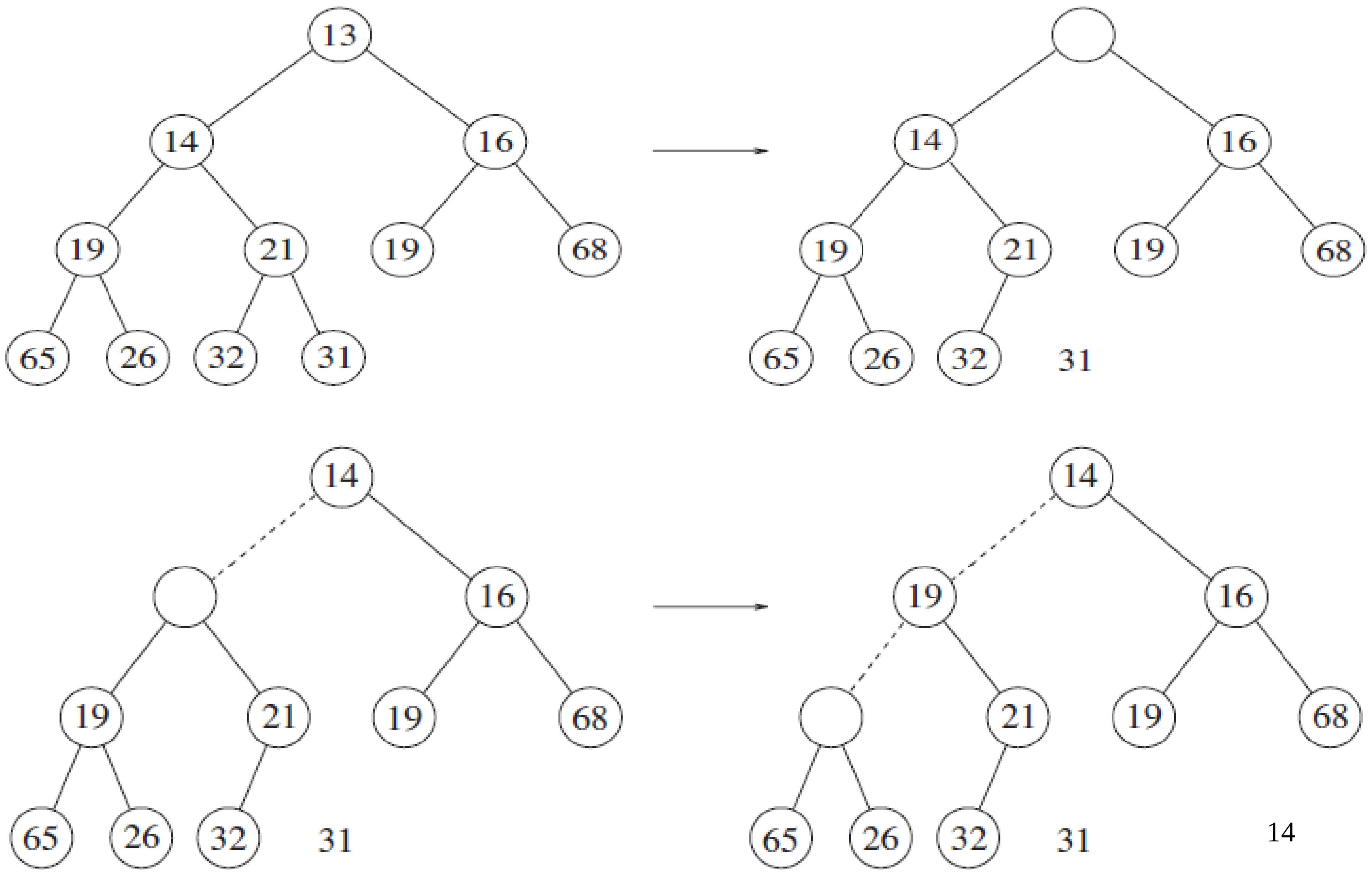
- The percolate up strategy uses *d* + 1 assignments.
- A copy of X (item to be inserted) is placed in position 0 in order to make the loop terminate.
- Worst time to do the insertion is O(logN).
- On average, the percolation terminates early;
  - 2.607 comparisons are required on average to perform an insert.
  - the average insert moves an element up 1.607 levels.

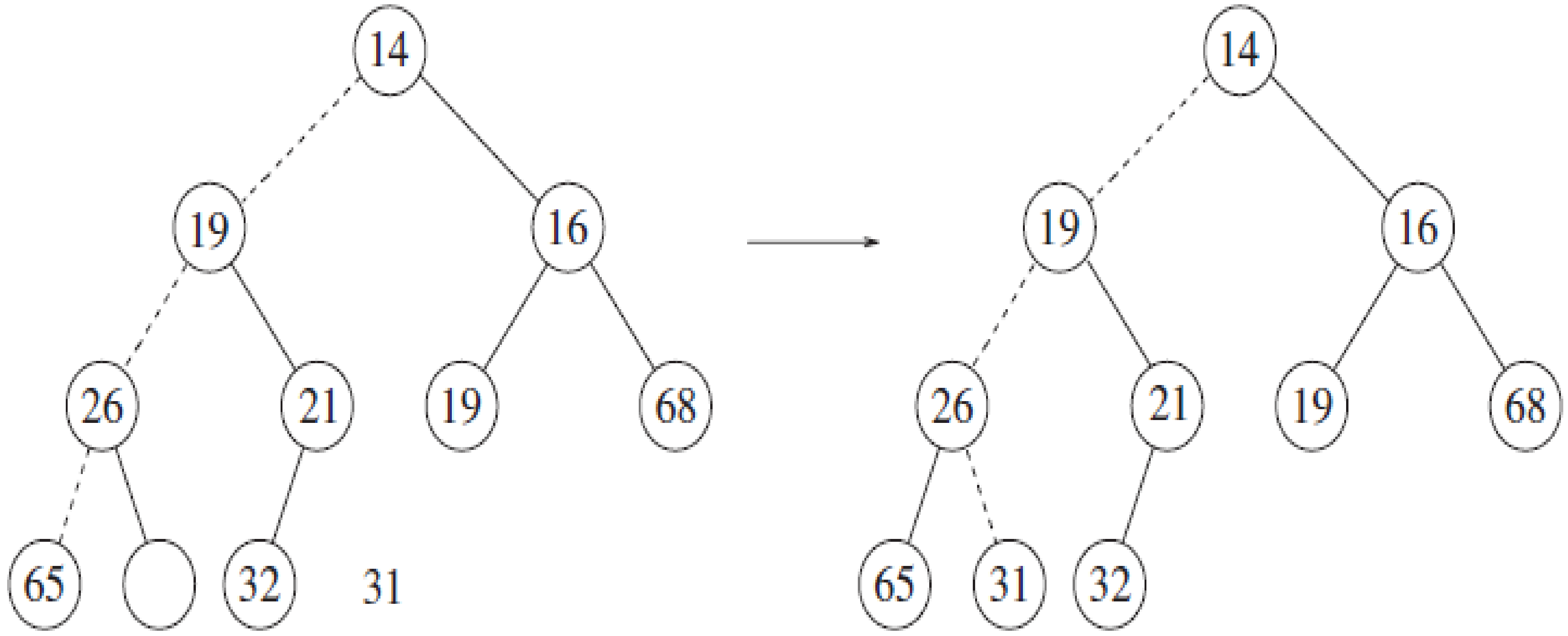
# DeleteMin

Similar idea to Insert, but now **percolate down**:

- When the minimum is removed, a *hole* is created at the root.
- The heap has one element less ➔ last element *X* in the heap must move somewhere in the heap.
- If *X* can be placed in the *hole*, then done.
- Else, slide the smaller of the *hole*'s children into the *hole*, thus pushing the *hole* down one level.
- Repeat this step until *X* can be placed in the *hole*.
- Thus, the goal is to place *X* in its correct spot along a path from the root containing *minimum* children.
- Same technique used as in insertion to avoid swaps

## Delete the minimum (13) from the left tree below





- The **worst-case** running time for this operation is ***O(log N)***.
- **On average**, the element that is placed at the root is percolated almost to the bottom of the heap (which is the level it came from), so the average running time is ***O(logN)***.

- **Code for DeleteMin-PercolateDown**

# Other Heap Operations

- A heap has very little ordering information ➔ no way to find any particular element without a linear scan through the entire heap
- If it is important to know where elements are, some other data structure, such as a hash table, must be used in addition to the heap.
- If we assume that the position of every element is known by some other method, then several other operations become cheap.
- The first three operations increaseKey, decreaseKey, and remove; all run in logarithmic worst-case time.

# DecreaseKey

- The ***decreaseKey(p, Δ)*** operation lowers the value of the item at position ***p*** by a positive amount Δ.
- Since this might violate the heap order, it must be fixed by a *percolate up*.
    - Decrease the value of the element,
    - Interchange it with its parent if it is less than its parent,
    - Continue doing this till a root is reached

- Example of application: could be useful to system administrators who can make their programs run with highest priority.

# Example of decreaseKey

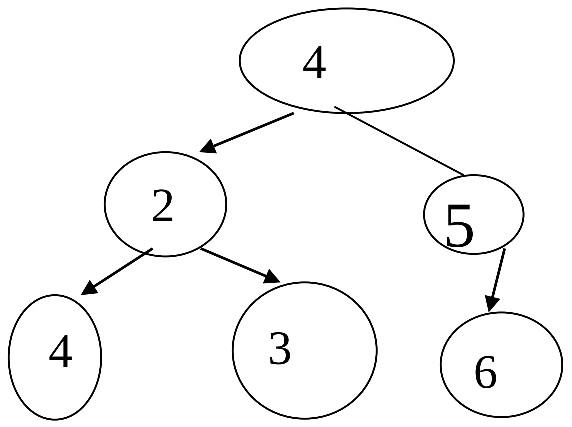
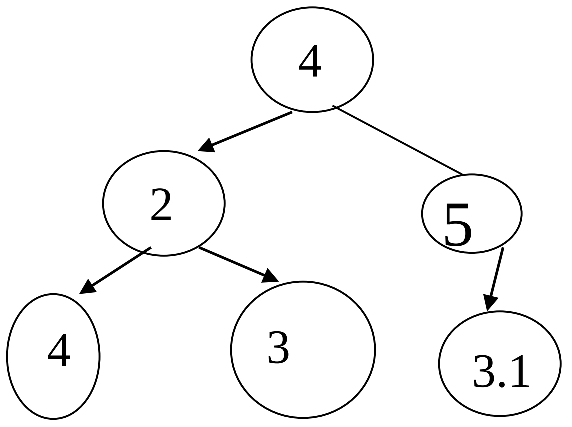


Fig 1



Decrease 6 by 2.9 (i.e. to 3.1)

Fig 2

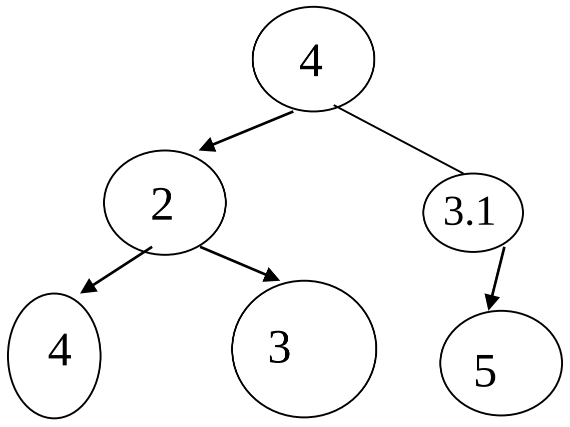


Fig 3

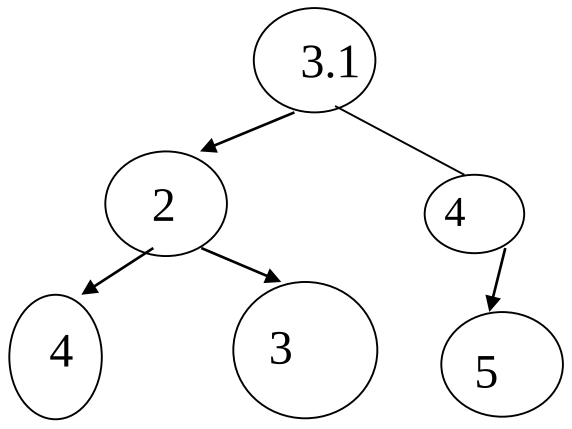


Fig 4

# IncreaseKey

- The ***increaseKey(p, $\Delta$)*** operation increases the value of the item at position *p* by a positive amount.
- This is done with a *percolate down.*
  - Increase the value of the element,
  - Interchange it with the lesser of its children if it is greater than any of its children,
  - Continue doing this till a leaf is reached

- Example application: Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

# Example of increaseKey

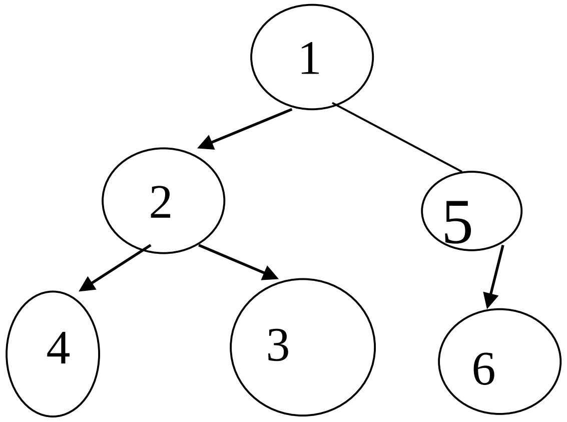
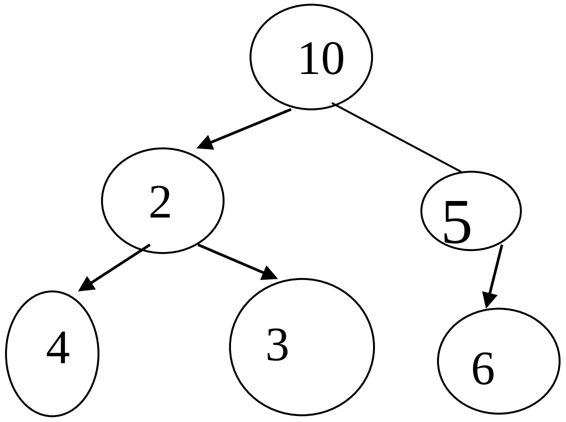


Fig 1



Increase Root by 9

Fig 2

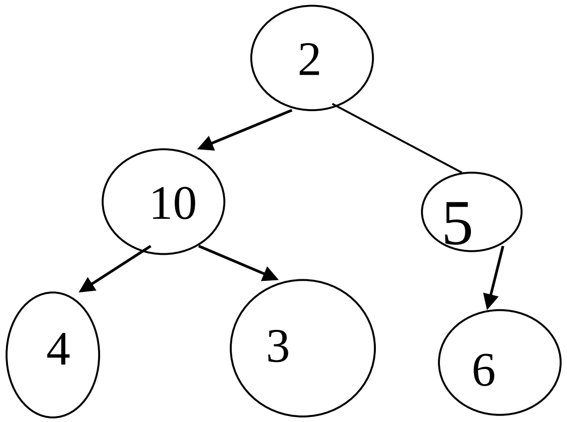


Fig 3

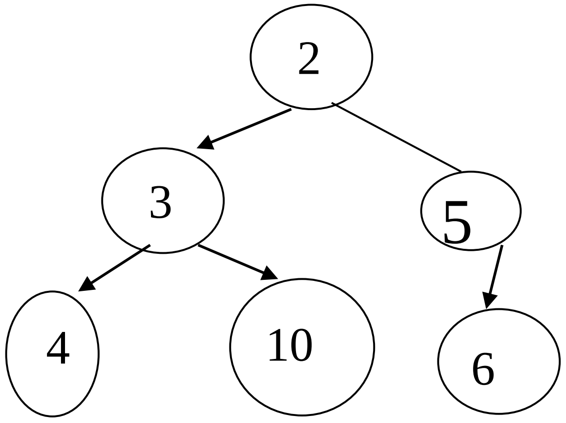


Fig 4

# Remove & buildHeap

***remove(p):*** removes the node at position ***p*** from the heap. This is done by first performing ***decreaseKey(p,∞)*** and then performing ***deleteMin()***.

***buildHeap***:

- A binary heap is sometimes constructed from an initial collection of items.
- The constructor takes as input $N$ items and places them into a heap. This can be done with $N$ successive inserts.
- Each insert will take $O(1)$ average and $O(\log N)$ worst-case time ➔ the total running time of buildHeap would be $O(N)$ average but $O(N \log N)$ worst-case.
- With care a linear time bound can be guaranteed

# buildHeap algorithm

- General algorithm:
  - place the *N* items into the tree in any order, maintaining the structure property.
  - if ***percolateDown(i)*** percolates down from node i, the **buildHeap routine** can be used by the constructor to create a heap-ordered tree.

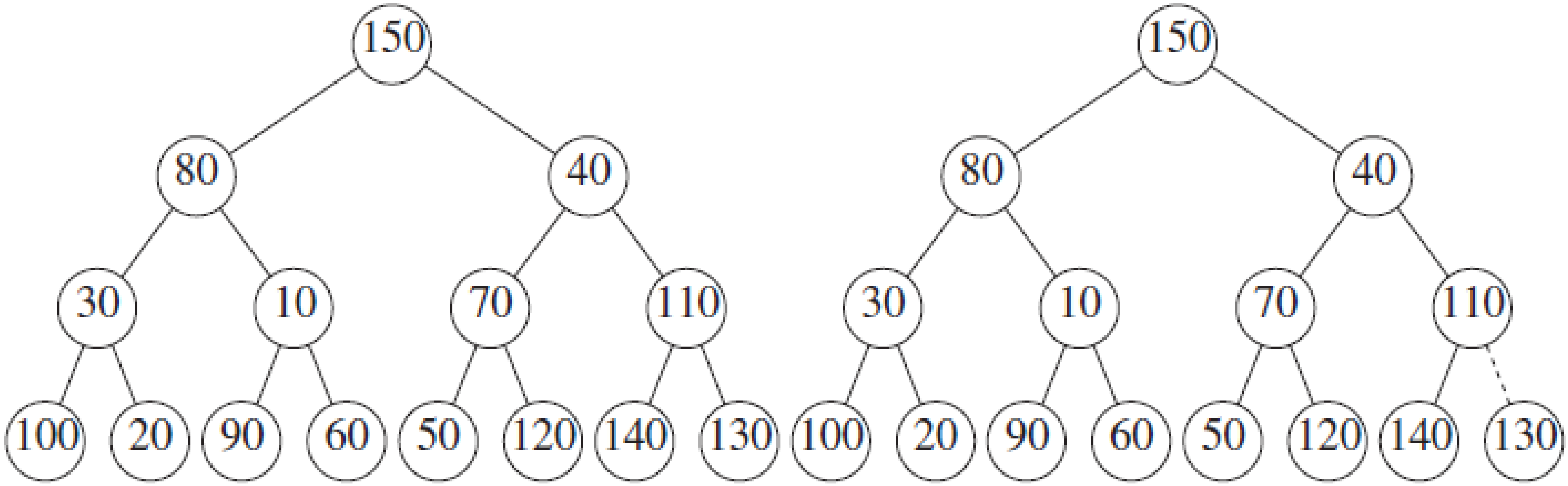


**Figure 6.15** Left: initial heap; right: after `percolateDown(7)`

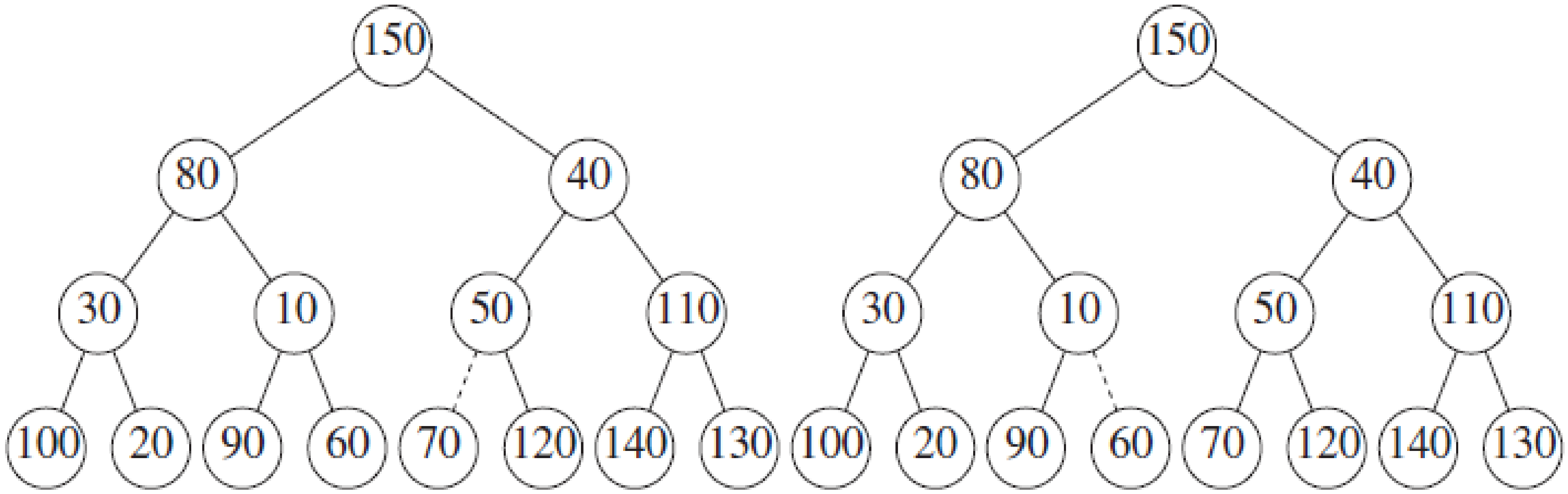


**Figure 6.16** Left: after `percolateDown(6)`; right: after `percolateDown(5)`

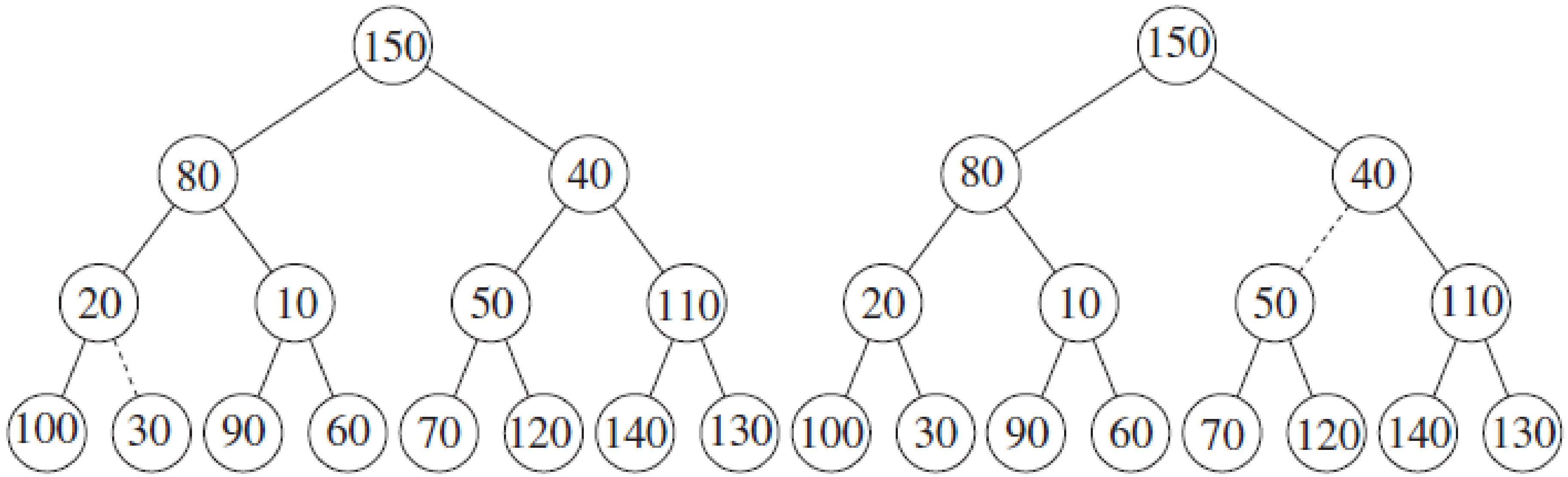


**Figure 6.17** Left: after `percolateDown(4)`; right: after `percolateDown(3)`

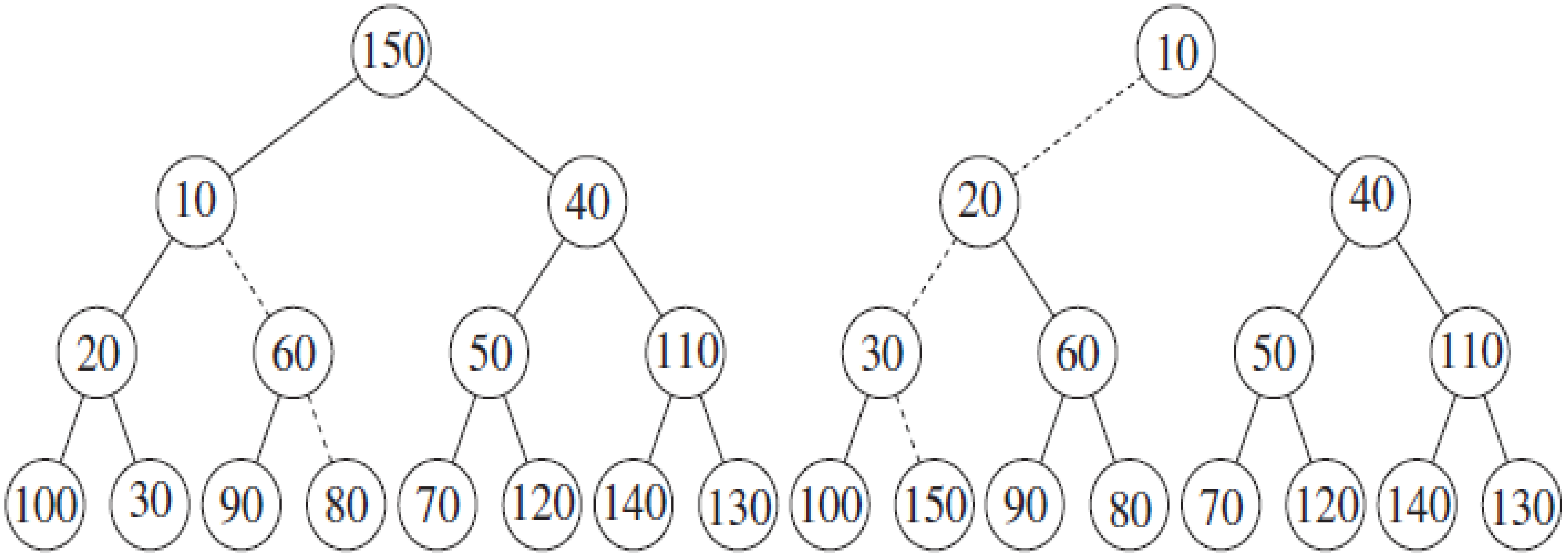


**Figure 6.18** Left: after `percolateDown(2)`; right: after `percolateDown(1)`

## buildHeap routine

# Applications of PQs

The Selection Problem: Given a list of *N* elements, which can be totally unordered, and an integer *k*, find the *k*th largest element.

2 algorithms from Chapter 1:

- Algorithm 1A: read the elements into an array and sort them, returning the appropriate element. Assuming a simple sorting algorithm, the running time is $O(N^2)$.
- Algorithm 1B: read ***k*** elements into an array and sort them. The smallest of these is in the kth position. Process the remaining elements one by one. As an element arrives, it is compared with the kth element in the array. If it is larger, then the kth element is removed, and the new element is placed in the correct place among the remaining k−1 elements. When the algorithm ends, the element in the kth position is the answer. The running time is O(N·k). Prove it!

## The Selection Problem (cont.)

- If $k$ = ceiling($N$/2), then both algorithms 1A and 1B are $O(\boldsymbol{N}^2)$.
- The next two algorithms run in $O(N \log N)$ in the extreme case of $k$ = ceiling($N$/2), which is a great improvement.

## Algorithm 6A for the Selection Pb

- Suppose we are interested in finding the *k*th *smallest* element.
- The algorithm is simple:
  - Read the *N* elements into an array.
  - Apply the *buildHeap* algorithm to this array.
  - Perform *k deleteMin* operations.
  - The last element extracted from the heap is the answer.
- Clearly, by changing the heap-order property, the original problem of finding the *k*th *largest* element can be solved

# Complexity of Algorithm 6A

- The worst-case timing is $O(N)$ to construct the heap, if buildHeap is used
- Worst-case timing for each *deleteMin* is $O(\log N)$.
- Since there are $k$ *deleteMin* operations, the total running time is $O(N + k \log N)$.
- If $k = O(N / \log N)$, then the running time is dominated by the buildHeap operation and is $O(N)$.
- For larger values of $k$, the running time is $O(k \log N)$. If $k = \text{ceiling}(N / 2)$, then the running time is $\theta(N \log N)$.

# Algorithm 6B for the Selection Pb

Use the same idea from algorithm 1B but with a heap.

- At any point in time, maintain a set S of the k largest elements.
- After the first *k* elements are read, when a new element is read it is compared with the *k*th largest element, denoted by $S_k$. (Notice that $S_k$ is the smallest element in *S*.)
- If the new element is larger, then it replaces $S_k$ in *S*. *S* will then have a new smallest element, which may or may not be the newly added element.
- At the end of the input, find the smallest element in *S* and return it as the answer

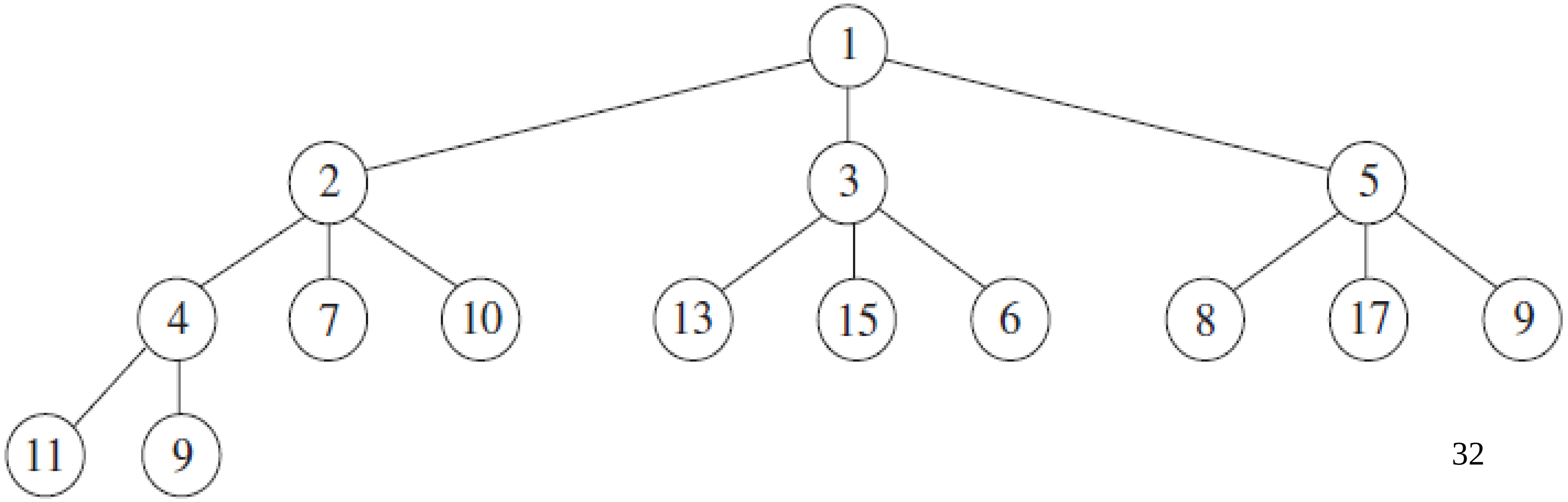
## Algorithm 6B for the Selection Pb

- 6B $\neq$ 1B in that a heap is used to implement *S*.
- The first *k* elements are placed into the heap in total time $O(k)$ with a call to buildHeap.
- The time to process each of the remaining elements is
  - $O(1)$, to test if the element goes into S, plus
  - O(log k), to delete $S_k$ and insert the new element if this is necessary.
- Thus, the total time is O(k + (N − k) log k) = O(N log k).
- This algorithm also gives a bound of $\theta(N \log N)$ for finding the median.

# d-Heaps

- Binary heaps are so simple that they are almost always used when priority queues are needed.
- A ***d*-heap** is a simple generalization of a binary heap where all nodes have *d* children (thus, a binary heap is a 2-heap).

Here is a d-heap where d=3



- a *d*-heap is much shallower than a binary heap, improving the running time of inserts to $O(\log_d N)$.
- However, for large *d*, the *deleteMin* operation is more expensive, because even though the tree is shallower, the minimum of *d* children must be found, which takes $d-1$ comparisons using a standard algorithm ➔ time for this operation $O(d \log_d N)$.
- If *d* is a constant, both running times are, of course, $O(\log_d N)$
- Arrays can still be used for implementation. BUT unless *d* is a power of 2, the running time will seriously increase because we can no longer implement division by a bit shift.

# D-heaps pros and cons

- *d*-heaps are interesting in theory, because there are many algorithms where the number of insertions is much greater than the number of *deleteMins* (and thus a theoretical speedup is possible).
- When a priority queue is too large to fit entirely in main memory, a d-heap can be advantageous in much the same way as B-trees.
- There is evidence suggesting that 4-heaps may outperform binary heaps in practice.
- Heaps cannot perform finds.
- Combining two heaps into one (*merge*) is a hard operation; different ways exist to implement heaps with a running time of *O(log N)*. **Next topic!**

## Leftist Heaps

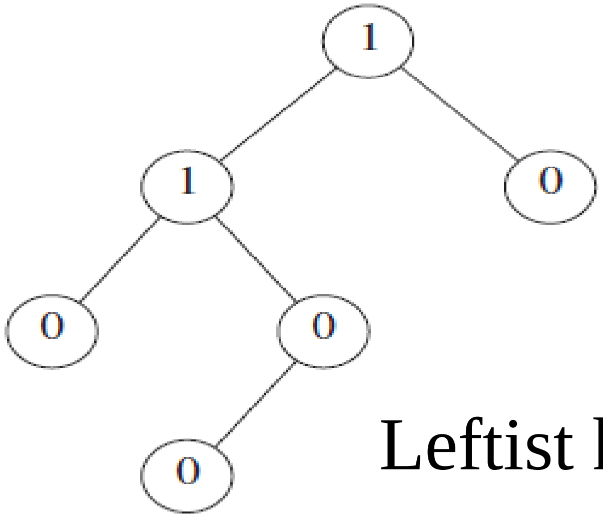
- A *leftist heap* is a binary tree with the same heap-order property.
- Only difference between leftist heaps and binary heaps: leftist heaps are not perfectly balanced; actually they attempt to be very unbalanced
- Definition: The **null path length**, *npl(X)*, of any node *X*, is the **length of the shortest path from *X* to a node without two children**. So the *npl* of a node with zero or one child is 0, while *npl(nullptr)* = −1.

  *npl(x) = min(npl(Lchild(x)), npl(Rchild(x))) +1.*

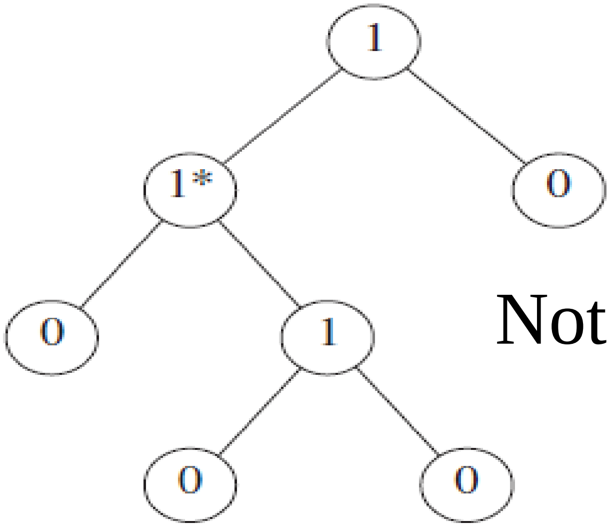
  (Also applies to nodes with 0 or 1 child since

  *npl(nullptr)= −1*)

**Leftist heap property:** for every node *X* in the heap, the *null path length* of the left child of X is at least as large as that of its right child.



Leftist heap



Not a leftist heap

- Leftist heap property goes out of its way to ensure the tree is unbalanced; it biases the tree to get deep toward the left.
- A tree consisting of a long path of left nodes is possible (and actually preferable to facilitate merging)—hence the name *leftist heap*.

- The *npl* of a node can be seen as the distance from that node to the nearest empty position in the subtree rooted at that node.
- Leftist heaps tend to have deep left paths ➔ the right path ought to be short.
- The right path (path from root to the rightmost leaf) in a leftist heap is as short as any path in the heap.
- **Theorem 6.2:** A leftist tree with *r* nodes on the right path must have at least $2^r - 1$ nodes.

## Why the leftist property?

Because it guarantees that:

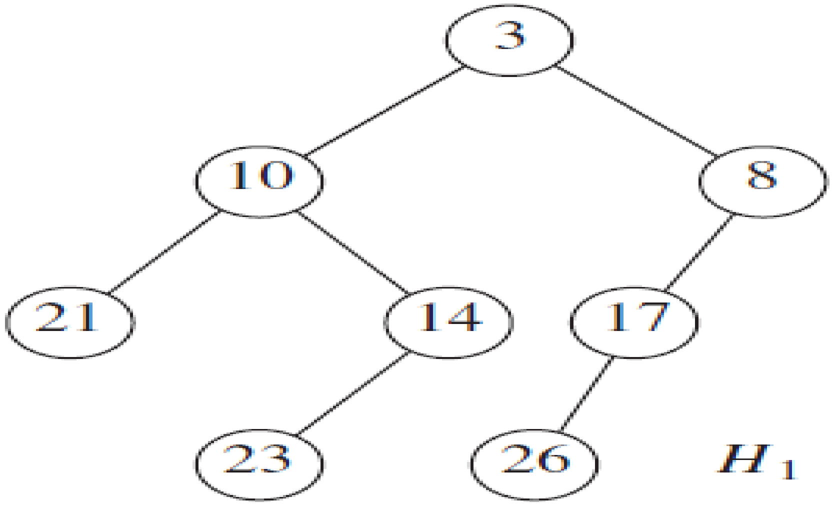
- The right path is really short compared to the number of nodes in the tree.
- A Leftist tree of N nodes has a right path of at most $\log_2 (N+1)$ nodes.
- So perform all the work (of merging) on the right path.
- Note: Performing inserts and merges on the right path could destroy the leftist heap property. But it turns out to be extremely easy to restore the property

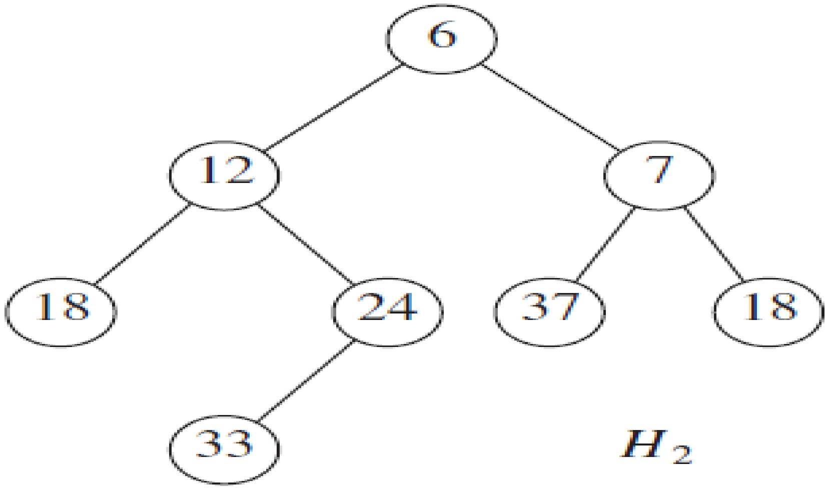
# Leftist Heap Operations

- The fundamental operation on LH is *merge*. (Insertion can be seen as a special case: a merge of a one-node heap with a larger heap.)
- A node entry: (Data, leftPtr, rightPtr, npl)

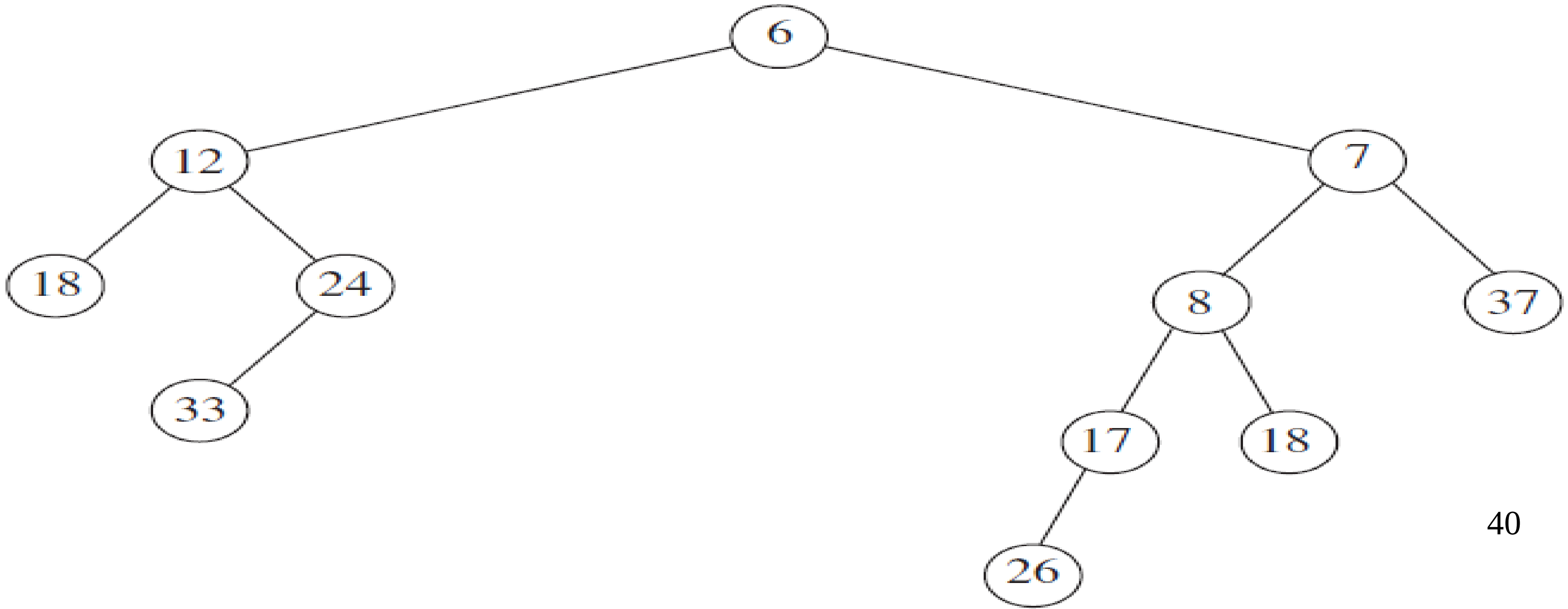
Merge(H1, H2) of two LHs:

- If either one is empty, return the other LH
- Otherwise, compare their roots
  - First, **recursively** merge **the heap that has the larger root** with the **right subheap of the heap with the smaller root**.

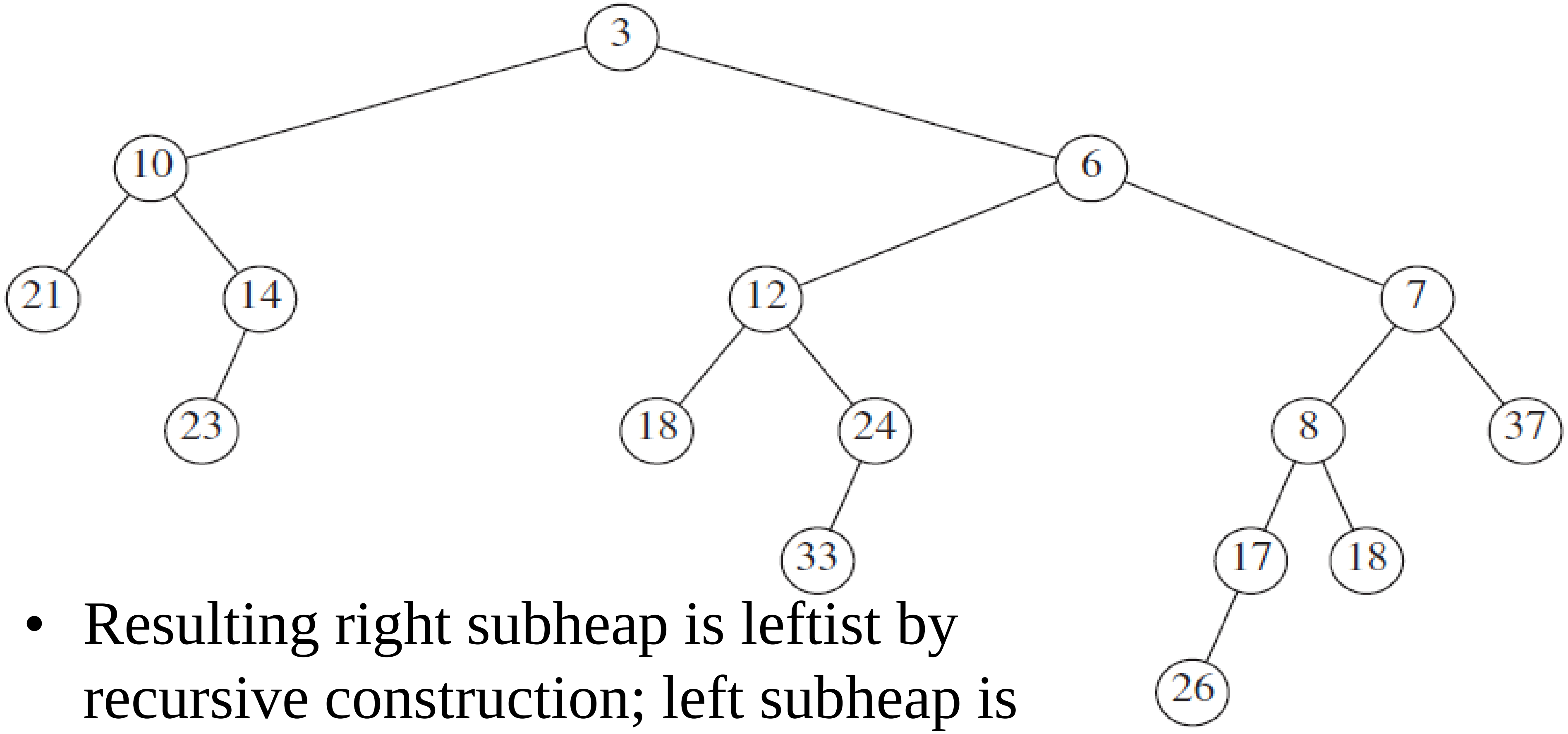




**Recursively** merge H2 with the subheap rooted at 8.
**Swap subheaps if needed** to restore Leftist heap property.



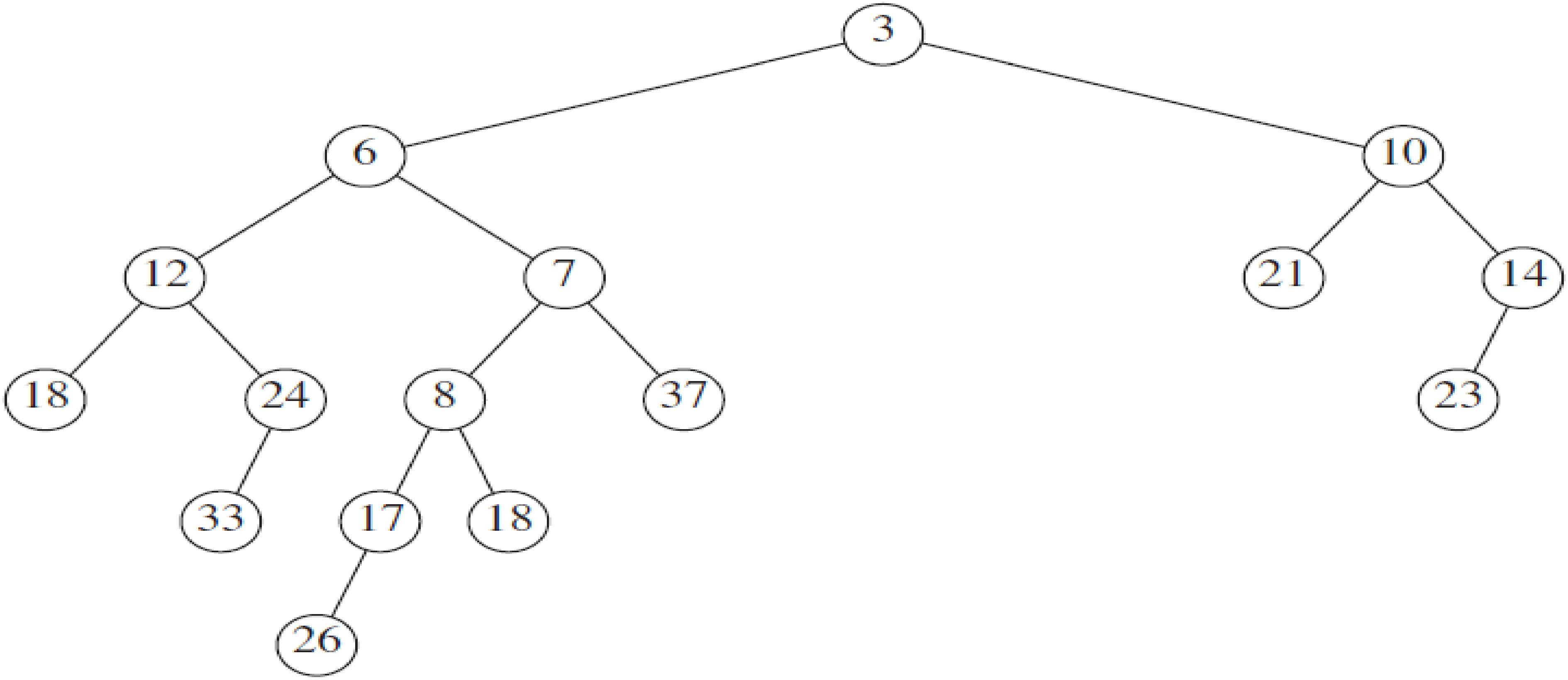
## Attach the leftist heap of previous figure as $H_1$'s right child



- Resulting right subheap is leftist by recursive construction; left subheap is not changed so it is leftist.
- Leftist property is violated at the root.
- Restore it by swapping the Left and Right subheaps.

Update the *npl* of the root (having the new *npl* of right child)

## Result of swapping children of $H_1$'s root



Leftist heap type declarations

routines for merging leftist heaps

## Insert / DeleteMin

- *Insert item* in a heap $h_1$:
  - make *item* a one-node heap $h_2$ and
  - perform a merge between $h_2$ and $h_1$.
- *DeleteMin*:
  - Destroy the root, creating two heaps,
  - Merge these two heaps
- Complexities:
  - Merge two heaps $O(\log N)$
  - Insert an element into the heap $O(\log N)$.
  - DeleteMin from heap $O(\log N)$.

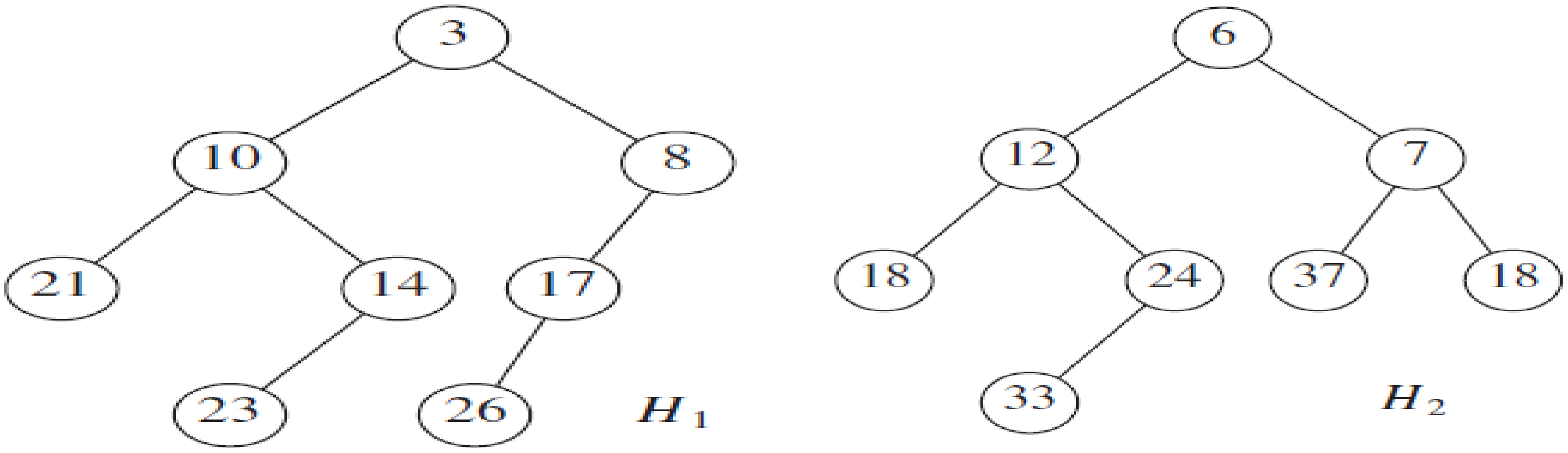
Implementation of *Insert* and *DeleteMin*

## Skew Heaps

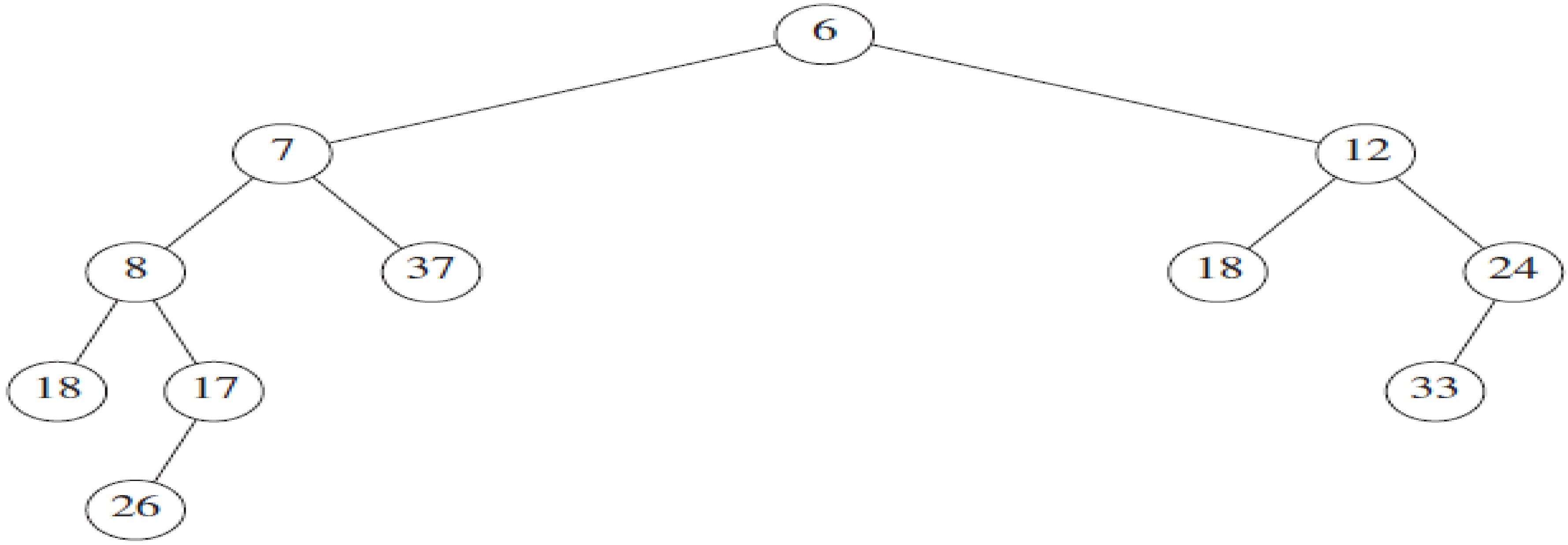
- A **Skew Heap** is a self-adjusting version of a leftist heap that is very simple to implement
- Skew Heaps are binary trees with heap order, but there is no structural constraint on these trees.
- No information is maintained about the null path length of any node
- The right path of a skew heap can be arbitrarily long at any time ➔ the worst-case running time of all operations is $O(N)$
- For any M consecutive operations, the total worst-case running time is O(M logN)
  ➔ skew heaps have O(log N) amortized cost per oeration

- The fundamental operation on skew heaps is merging
- We perform the exact same operations as before, with one exception:

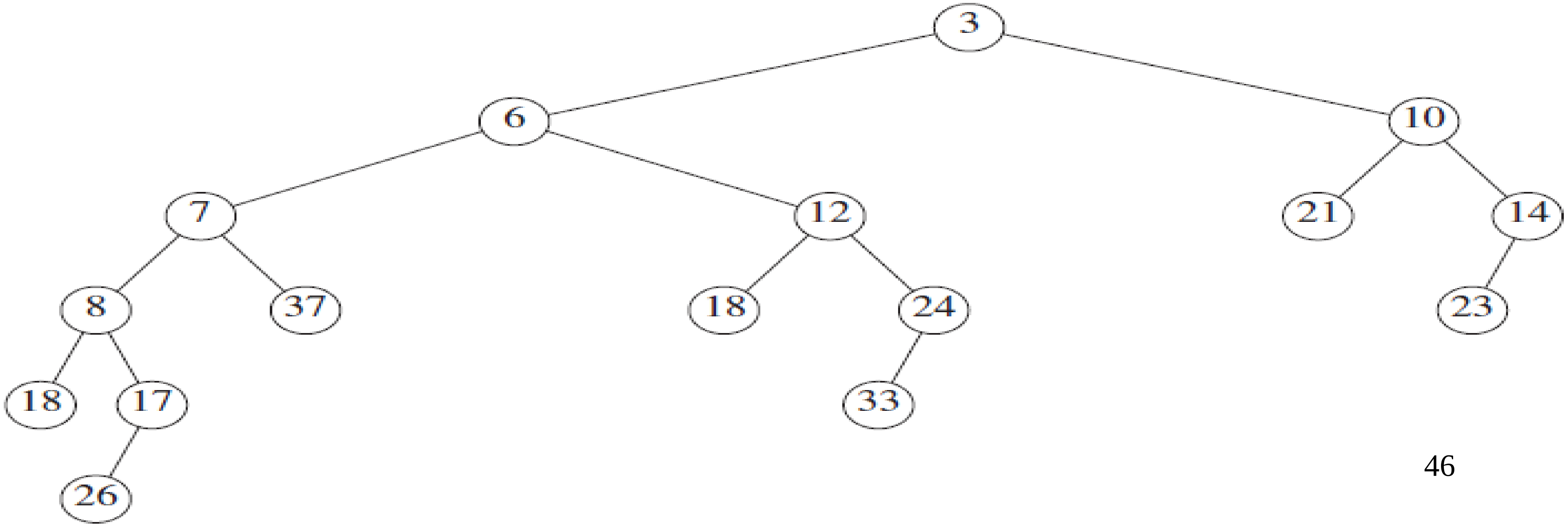
the swap is unconditional; we ***always*** do it, with the one exception that the largest of all the nodes on the right paths does not have its children swapped



**Recursively** merge H2 with the subheap rooted at 8.



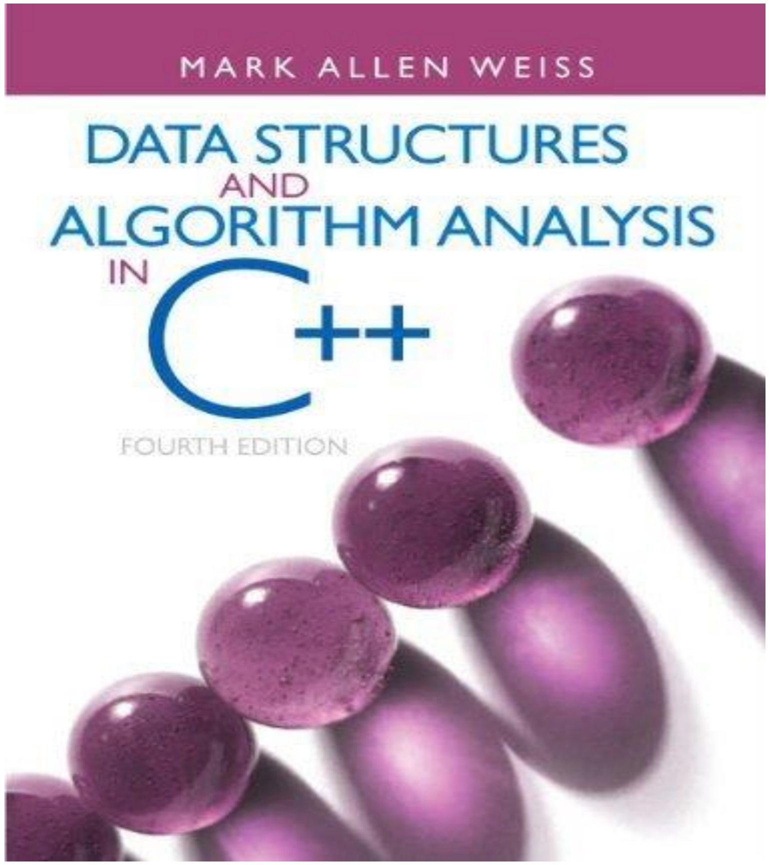
Result of merging the two heaps



## Final notes

- Because a right path could be long, a recursive implementation could fail because of lack of stack space, even though performance would otherwise be acceptable
- Advantage of Skew heaps: no extra space is required to maintain path lengths and no tests are required to determine when to swap children.

## Slides based on the textbook



Mark Allen Weiss, (2014 ) Data Structures and Algorithm Analysis in C++, 4th edition, Pearson.

Acknowledgement: This **course PowerPoints** make substantial (non-exclusive) use of the PPT chapters prepared by Prof. Saswati Sarkar from the University of Pennsylvania, USA, themselves developed on the basis of the course textbook. Other references, if any, will be mentioned wherever applicable.