CS 240 – Data Structures and Data Management

Module 2: Priority Queues

T. Biedl   É. Schost   O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2021

References: Sedgewick 9.1-9.4

version 2021-01-03 15:49

Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - *PQ-sort* and *Heapsort*
   - Summary

Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - *PQ-sort* and *Heapsort*
   - Summary

Abstract Data Types

Abstract Data Type (ADT): A description of *information* and a collection of *operations* on that information.

The information is accessed *only* through the operations.

We can have various **realizations** of an ADT, which specify:

- How the information is stored (**data structure**)
- How the operations are performed (**algorithms**)

Stack ADT

Stack: an ADT consisting of a collection of items with operations:

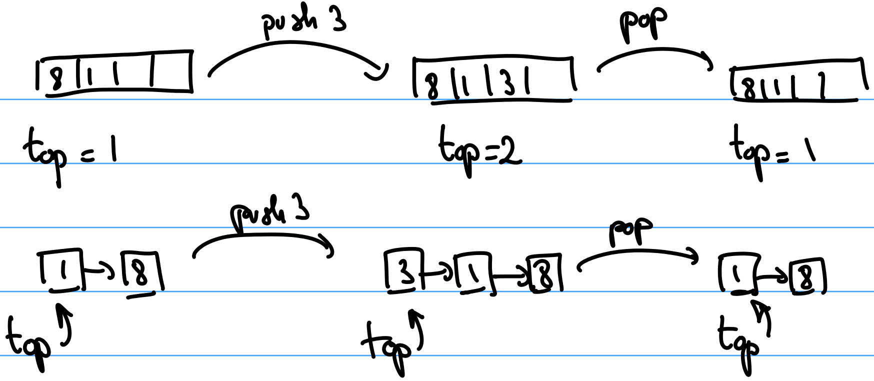
- *push*: inserting an item
- *pop*: removing the most recently inserted item

Items are removed in LIFO (*last-in first-out*) order.
Items enter the stack at the *top* and are removed from the *top*.
We can have extra operations: *size*, *isEmpty*, and *top*

Applications: Addresses of recently visited web sites, procedure calls

Realizations of Stack ADT

- using arrays
- using linked lists



push 3
pop
8 1
8 1 3
8 1
top = 1
top = 2
top = 1
push 3
pop
1 8
3 1 8
1 8
top
top
top

Queue ADT

Queue: an ADT consisting of a collection of items with operations:

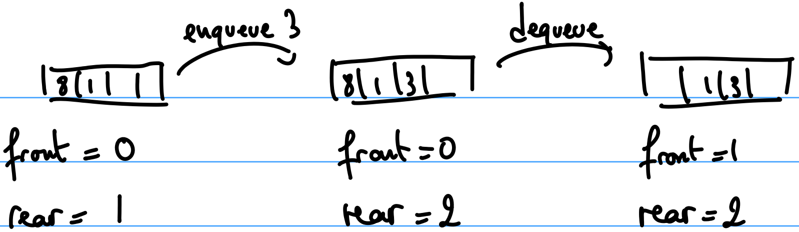
- *enqueue*: inserting an item
- *dequeue*: removing the least recently inserted item

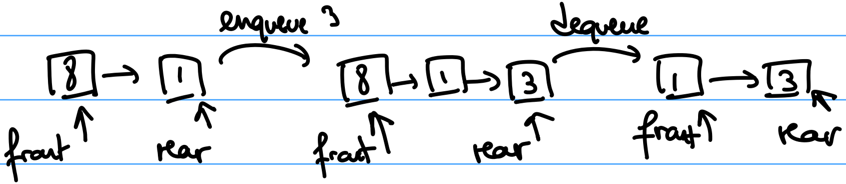
Items are removed in FIFO (*first-in first-out*) order.
Items enter the queue at the *rear* and are removed from the *front*.
We can have extra operations: *size*, *isEmpty*, and *front*

Applications: Waiting lines, printer queues

Realizations of Queue ADT

- using (circular) arrays
- using linked lists


enqueue 3
dequeue
8
1
8
1
3
1
3
front = 0
rear = 1
front = 0
rear = 2
front = 1
rear = 2


enqueue 3
dequeue
8
1
8
1
3
1
3
front
rear
front
rear
front
rear

Outline

Priority Queue ADT

Priority Queue: An ADT consisting of a collection of items (each having a **priority**) with operations

- *insert*: inserting an item tagged with a priority
- *deleteMax*: removing the item of *highest* priority

deleteMax is also called *extractMax* or *getmax*.
The priority is also called *key*.

The above definition is for a **maximum-oriented** priority queue. A **minimum-oriented** priority queue is defined in the natural way, replacing operation *deleteMax* by *deleteMin*,

Applications: typical “todo” list, simulation systems, sorting

Using a Priority Queue to Sort

$A = [3, 9, 1]$

$PQ = \{3, 9, 1\}$

deleteMax ⇝ 9 $A = [3, 9, \underline{9}]$ $PQ = \{3, 1\}$

deleteMax ⇝ 3 $A = [3, \underline{3}, 9]$ $PQ = \{1\}$

deleteMax ⇝ 1 $A = [\underline{1}, 3, 9]$ $PQ = \{\}$

Using a Priority Queue to Sort

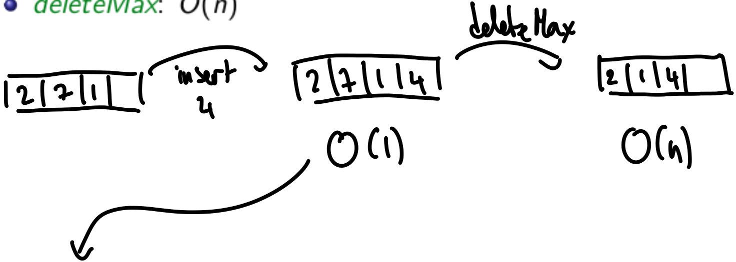
PQ-Sort$(A[0..n-1])$
1. initialize PQ to an empty priority queue
2. **for** $k \leftarrow 0$ **to** $n-1$ **do**
3. $PQ.insert(A[k], A[k])$ (priority and item are equal to $A[k]$)
4. **for** $k \leftarrow n-1$ **down to** 0 **do**
5. $A[k] \leftarrow PQ.deleteMax()$

- run-time $O(\sum_{0 \le i < n} insert(i) + \sum_{0 \le i < n} deleteMax(i))$
- depends on how we implement the priority queue

Realizations of Priority Queues

Realization 1: unsorted arrays

- *insert*: $O(1)$
- *deleteMax*: $O(n)$



O(1) only if the array is not full!

Realizations of Priority Queues

Realization 1: unsorted arrays

- *insert*: $O(1)$
- *deleteMax*: $O(n)$

Note: We assume **dynamic arrays**, i. e., expand by doubling as needed. (Amortized over all insertions this takes $O(1)$ extra time.)

Suppose we start from A of length 1

we do n insert , $n = 2^k$

Total cost of inserts

$$= O(\underbrace{1+1+1+\dots+1}_{n \text{ times}} \quad \underbrace{+1+2+4+8+\dots+2^{k-1}}_{2^k-1 = n-1})$$

$$= O(2n-1) = O(n).$$

Realizations of Priority Queues

Realization 1: unsorted arrays

- *insert*: $O(1)$
- *deleteMax*: $O(n)$

Note: We assume **dynamic arrays**, i. e., expand by doubling as needed. (Amortized over all insertions this takes $O(1)$ extra time.)

Using unsorted linked lists is identical.

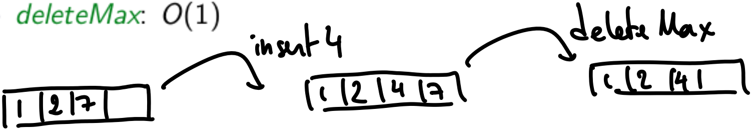
PQ-sort with this realization yields *selection sort*, so runtime is

$$O(\sum_{i<n} i) = O(n^2)$$

Realizations of Priority Queues

Realization 2: sorted arrays

- *insert*: $O(n)$
- *deleteMax*: $O(1)$



Realizations of Priority Queues

Realization 2: sorted arrays

- *insert*: $O(n)$
- *deleteMax*: $O(1)$

Using sorted linked lists is identical.

PQ-sort with this realization yields *insertion sort*, runtime is

$$O(\sum_{i<n} i) = O(n^2)$$

Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - *PQ-sort* and *Heapsort*
   - Summary

Realization 3: Heaps

A **(binary) heap** is a certain type of binary tree.

You should know:

- A **binary tree** is either
  - empty, or
  - consists of three parts:
    a node and two binary trees (left subtree and right subtree).
- Terminology: root, leaf, parent, child, level, sibling, ancestor, descendant, etc.
- Any binary tree with n nodes has height at least $\log(n+1) - 1 \in \Omega(\log n)$.

The height of a non-empty tree is the length of the longest path root → node.
The height of the empty tree is −1.

Heaps – Definition

A **heap** is a binary tree with the following two properties:

1. **Structural Property:** All the levels of a heap are completely filled, except (possibly) for the last level. The filled items in the last level are *left-justified*.
2. **Heap-order Property:** For any node i, the key of the parent of i is larger than or equal to key of i.

Heaps – Definition

A **heap** is a binary tree with the following two properties:

1. **Structural Property:** All the levels of a heap are completely filled, except (possibly) for the last level. The filled items in the last level are *left-justified*.
2. **Heap-order Property:** For any node i, the key of the parent of i is larger than or equal to key of i.

The full name for this is *max-oriented binary heap*.

Heaps – Definition

A **heap** is a binary tree with the following two properties:

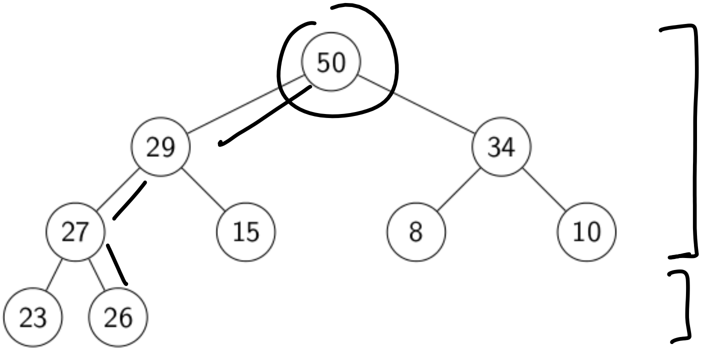
1. **Structural Property:** All the levels of a heap are completely filled, except (possibly) for the last level. The filled items in the last level are *left-justified*.

2. **Heap-order Property:** For any node i, the key of the parent of i is larger than or equal to key of i.

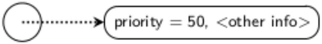
The full name for this is *max-oriented binary heap*.

Lemma: The height of a heap with n nodes is $\Theta(\log n)$.

Example Heap

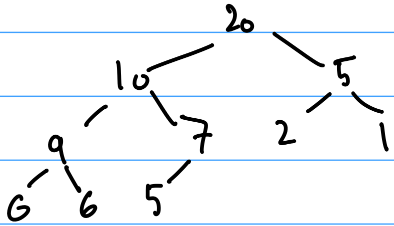


(In our examples we only show the priorities, and we show them directly in the node. A more accurate picture would be



)

For a heap of height $h=3$



1 node at level 0

2 nodes —— 1

4 nodes —— 2

between 1 and 8 nodes at level 3

Total: between 8 and 15 nodes

For a heap of height h, we have

- at least $1+2+4+\dots+2^{h-1}+1 = 2^h-1+1 = 2^h$ nodes
- at most $1+2+4+\dots+2^h = 2^{h+1}-1$ nodes

Call n the number of nodes. We got:

$$2^h \leq n \leq 2^{h+1} - 1 \leq 2^{h+1}$$ ← true for **any** binary tree.

$\log_2$ ↓

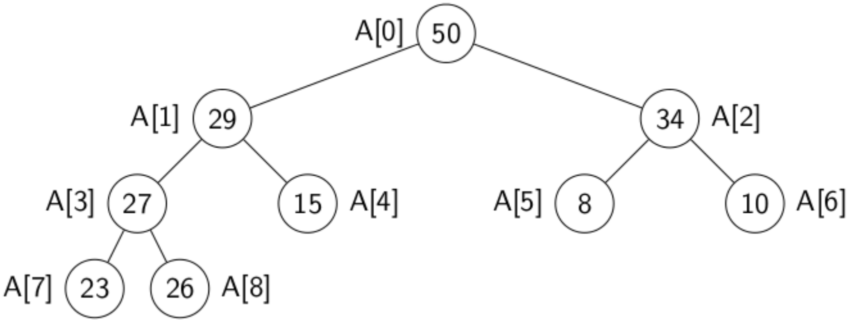
$$h \leq \log_2(n) \leq h+1$$

$$\underline{\log_2(n) - 1 \leq h} \leq \log_2(n) \rightarrow h \in \Theta(\log_2(n))$$

Storing Heaps in Arrays

Heaps should *not* be stored as binary trees!

Let H be a heap of n items and let A be an array of size n. Store root in $A[0]$ and continue with elements *level-by-level* from top to bottom, in each level left-to-right.



Heaps in Arrays – Navigation

It is easy to navigate the heap using this array representation:

- the *root* node is at index 0
  (We use "node" and "index" interchangeably in this implementation.)
- the *left child* of node i (if it exists) is node $2i + 1$
- the *right child* of node i (if it exists) is node $2i + 2$
- the *parent* of node i (if it exists) is node $\lfloor \frac{i-1}{2} \rfloor$
- the *last* node is $n - 1$

Heaps in Arrays – Navigation

It is easy to navigate the heap using this array representation:

- the *root* node is at index 0
  (We use "node" and "index" interchangeably in this implementation.)
- the *left child* of node i (if it exists) is node $2i + 1$
- the *right child* of node i (if it exists) is node $2i + 2$
- the *parent* of node i (if it exists) is node $\lfloor \frac{i-1}{2} \rfloor$
- the *last* node is $n - 1$

We should hide implementation details using helper-functions!

- functions *root()*, *parent(i)*, *last(n)*, etc.

Outline

Insert in Heaps

- Place the new key at the first free leaf
- The heap-order property might be violated: perform a *fix-up*:

Insert in Heaps

- Place the new key at the first free leaf
- The heap-order property might be violated: perform a *fix-up*:

```
fix-up(A, k)
k: an index corresponding to a node of the heap
1.    while parent(k) exists and A[parent(k)] < A[k] do
2.        swap A[k] and A[parent(k)]
3.        k ← parent(k)
```

The new item "bubbles up" until it reaches its correct place in the heap.

Insert in Heaps

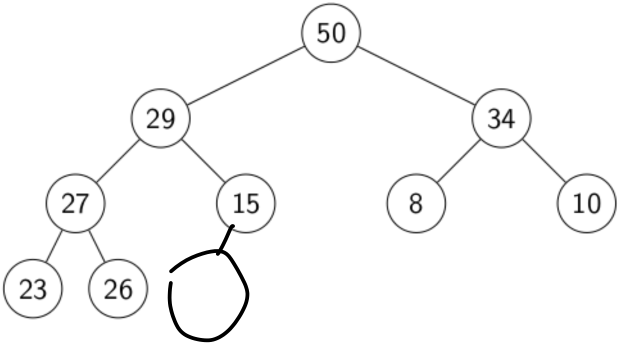
- Place the new key at the first free leaf
- The heap-order property might be violated: perform a *fix-up*:

> *fix-up*(A, k)
> k: *an index corresponding to a node of the heap*
> 1. **while** $parent(k)$ exists **and** $A[parent(k)] < A[k]$ **do**
> 2. swap $A[k]$ and $A[parent(k)]$
> 3. $k \leftarrow parent(k)$

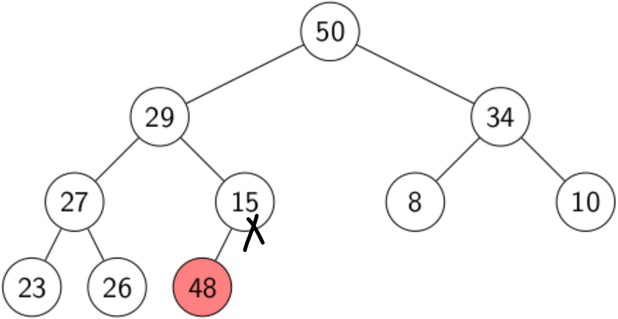
The new item "bubbles up" until it reaches its correct place in the heap.

Time: $O(\text{height of heap}) = O(\log n)$.

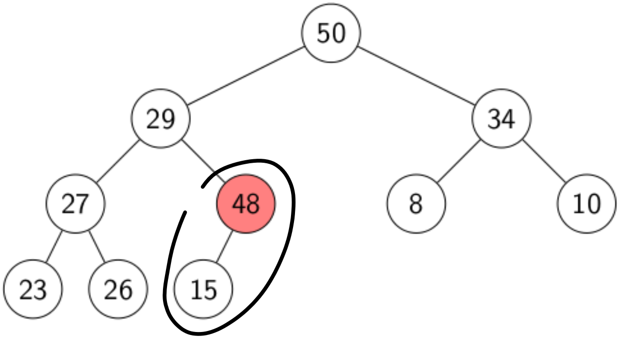
fix-up example



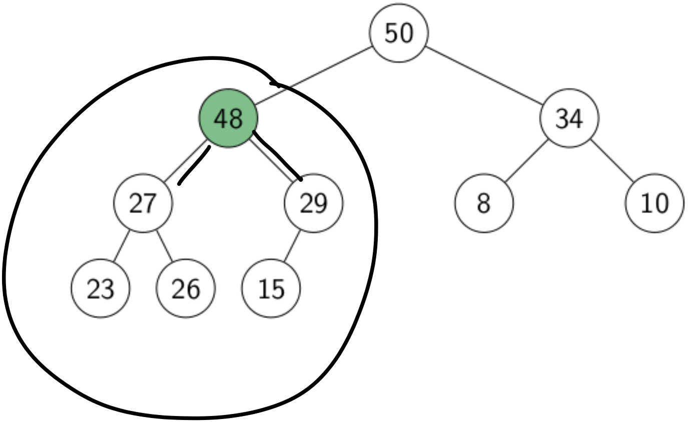
fix-up example



fix-up example



fix-up example



deleteMax in Heaps

- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

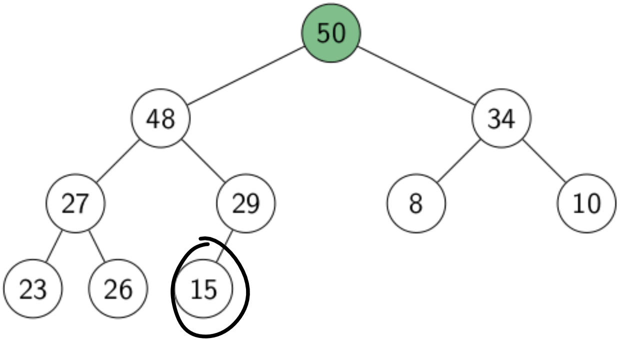
deleteMax in Heaps

- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

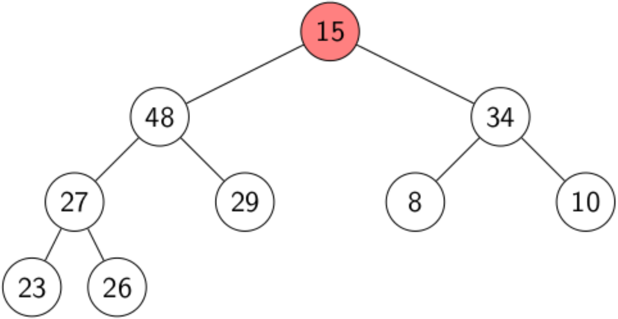
```
fix-down(A, n, k)
A: an array that stores a heap of size n
k: an index corresponding to a node of the heap
    while k is not a leaf do
        // Find the child with the larger key
        j ← left child of k
        if (j is not last(n) and A[j + 1] > A[j])
            j ← j + 1
        if A[k] ≥ A[j] break
        swap A[j] and A[k]
        k ← j
```

Time: $O(\text{height of heap}) = O(\log n)$.

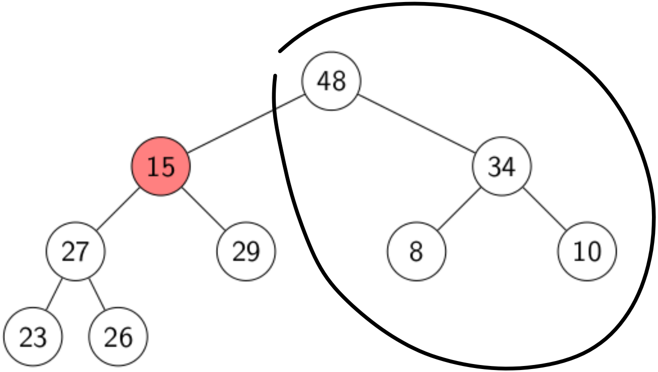
deleteMax example



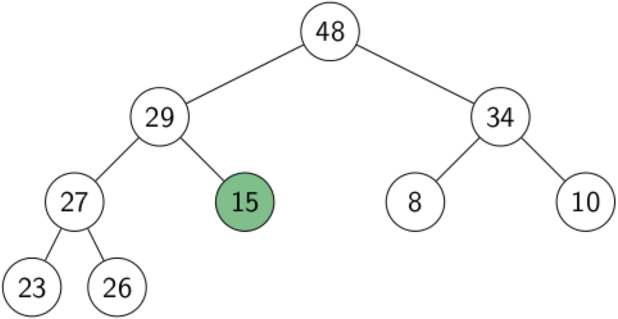
deleteMax example



deleteMax example



deleteMax example



Priority Queue Realization Using Heaps

- Store items in array A and globally keep track of *size*

insert(x)
1. increase *size*
2. $\ell \leftarrow$ *last*(*size*)
3. $A[\ell] \leftarrow x$
4. *fix-up*(A, ℓ)

deleteMax()
1. $\ell \leftarrow$ *last*(*size*)
2. *swap* A[*root*()] and $A[\ell]$
3. decrease *size*
4. *fix-down*(A, *size*, *root*())
5. **return** $A[\ell]$

insert and *deleteMax*: $O(\log n)$

Outline

Sorting using heaps

- Using the binary-heaps implementation of PQs, we obtain:

```
PQsortWithHeaps(A)
    initialize H to an empty heap
    for k ← 0 to n − 1 do
        H.insert(A[k])  (we just insert keys, no items)
    for k ← n − 1 down to 0 do
        A[k] ← H.deleteMax()
```

- Recall: runtime is

$$O\left(\sum_{0 \le i < n} insert(i) + \sum_{0 \le i < n} deleteMax(i)\right)$$

- both operations run in $O(\log n)$ time for heaps

⇝ *PQ-Ssrt* using heaps takes $O(n \log n)$ time.

- Can improve this with two simple tricks → **Heapsort**
  1. Heaps can be built faster if we know all input in advance.
  2. Can use the same array for input and heap. ⇝ *$O(1)$ auxiliary space!*

Building Heaps with Fix-up

Problem: Given n items all at once (in $A[0 \cdots n-1]$) build a heap containing all of them.

Solution 1: Start with an empty heap and insert items one at a time:

simpleHeapBuilding(A)
A: an array
1. initialize H as an empty heap
2. **for** $i \leftarrow 0$ **to** *size*$(A) - 1$ **do**
3. H.*insert*$(A[i])$

This corresponds to doing *fix-ups*
Worst-case running time: $\Theta(n \log n)$ (we proved $O(\ )$, $\Omega(\ )$ is an exercise)

Building Heaps with Fix-down

Problem: Given n items all at once (in $A[0 \cdots n-1]$) build a heap containing all of them.

Solution 2: Using *fix-downs* instead:

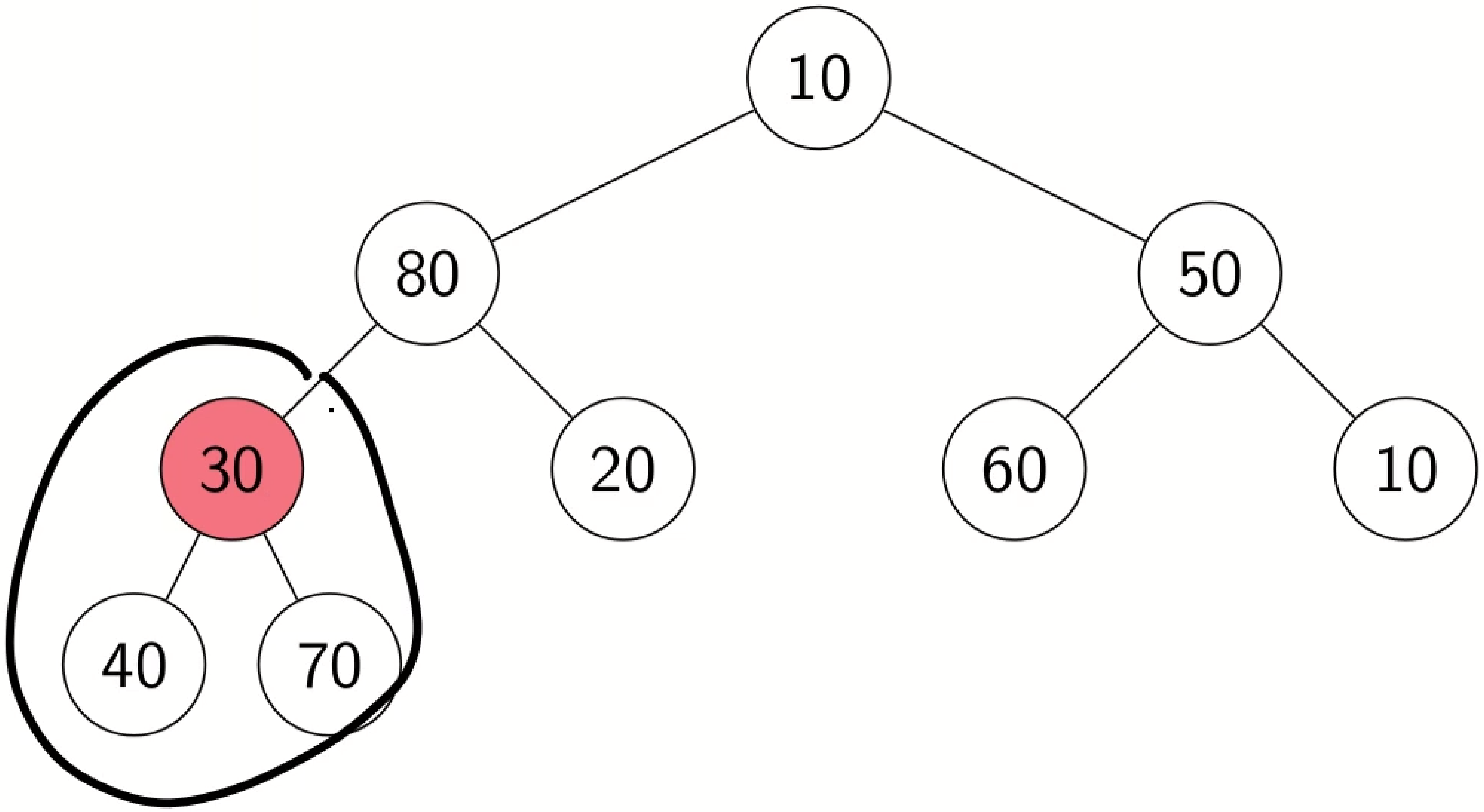
heapify(A)
A: an array

```
1.    n ← A.size()
2.    for i ← parent(last(n)) downto 0 do
3.        fix-down(A, n, i)
```

A careful analysis yields a worst-case complexity of $\Theta(n)$.
A heap can be built in linear time.

heapify example

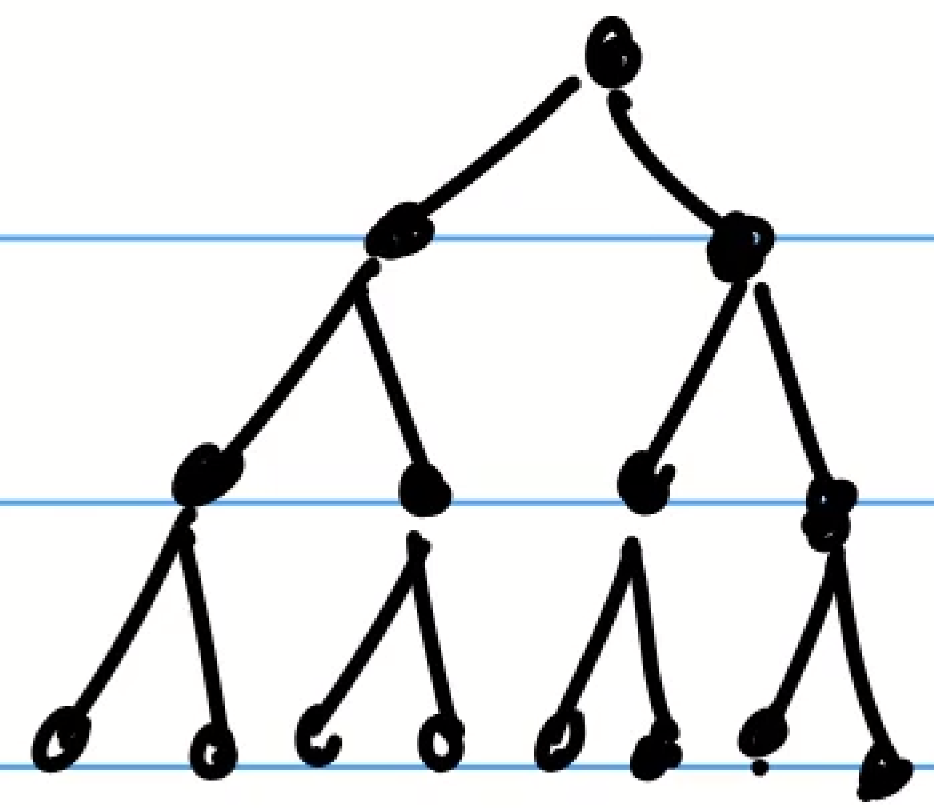
$A = [10, 80, 50, 30, 20, 60, 10, 40, 70]$



$T(n)$ = worst-case runtime of heapify for A of length n.

<u>Claim</u> $T(n) \in \Theta(n)$.

Proof for $n = 2^{h+1} - 1$



$h = 3$

$n = 2^4 - 1 = 15.$

$T(n) = \Theta(\text{worst case number of key swaps})$

$= \Theta(0 \cdot 2^{h-0} + 1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + 3 \cdot 2^{h-3} + \dots + h \cdot 2^{h-h})$

$= \Theta\left(2^h \left(\frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \dots + \frac{h}{2^h}\right)\right)$ ← ≤ 2

$= \Theta(2^h) = \Theta(n).$

HeapSort

- Idea: *PQ-sort* with heaps.
- But: Use same input-array A for storing heap.

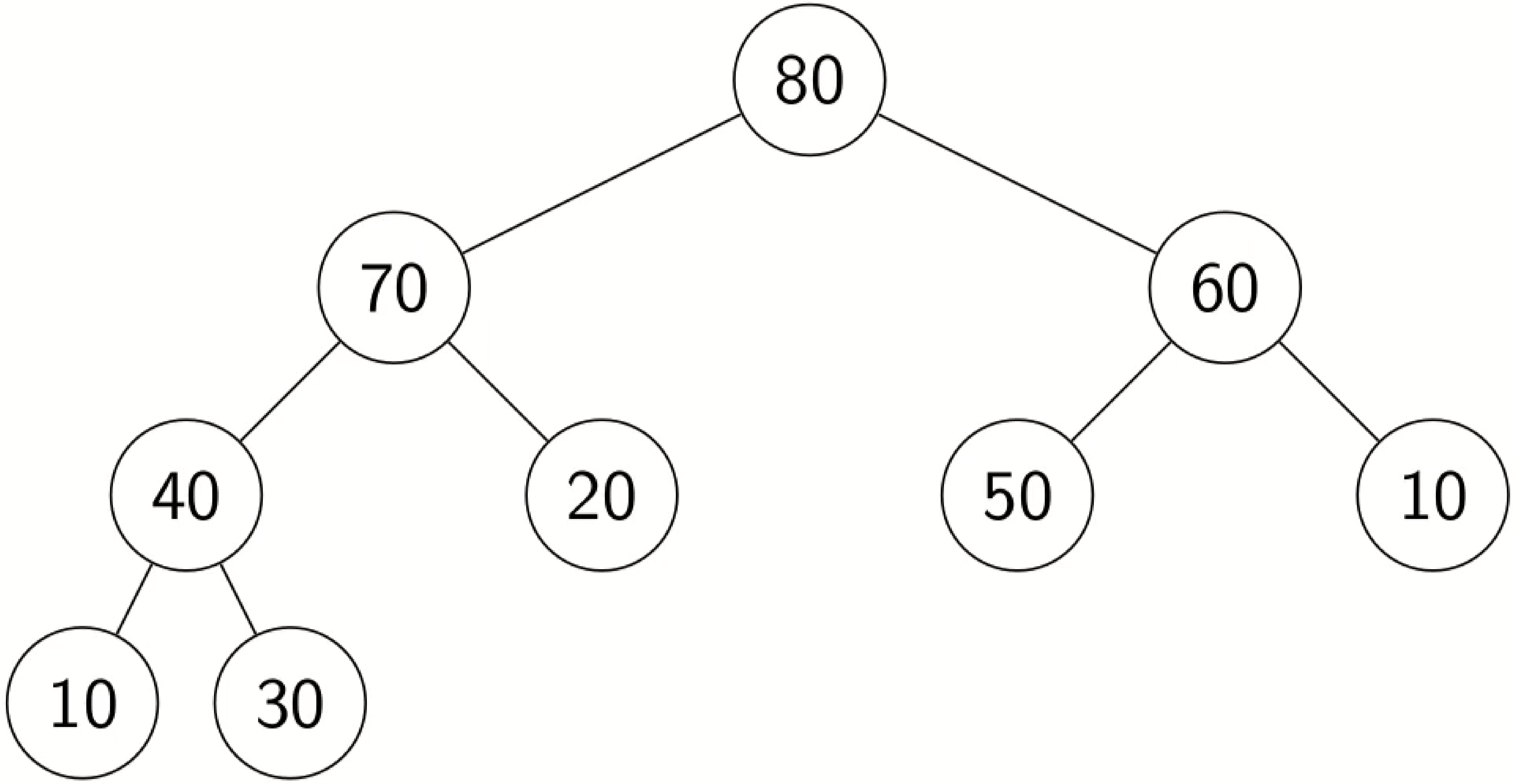
```
HeapSort(A, n)
    // heapify
    n ← A.size()
    for i ← parent(last(n)) downto 0 do
        fix-down(A, n, i)

    // repeatedly find maximum
    while n > 1
        // delete the maximum
        swap items at A[root()] and A[last(n)]
        decrease n
        fix-down(A, n, root())
```

The for-loop takes $\Theta(n)$ time and the while-loop takes $O(n \log n)$ time.

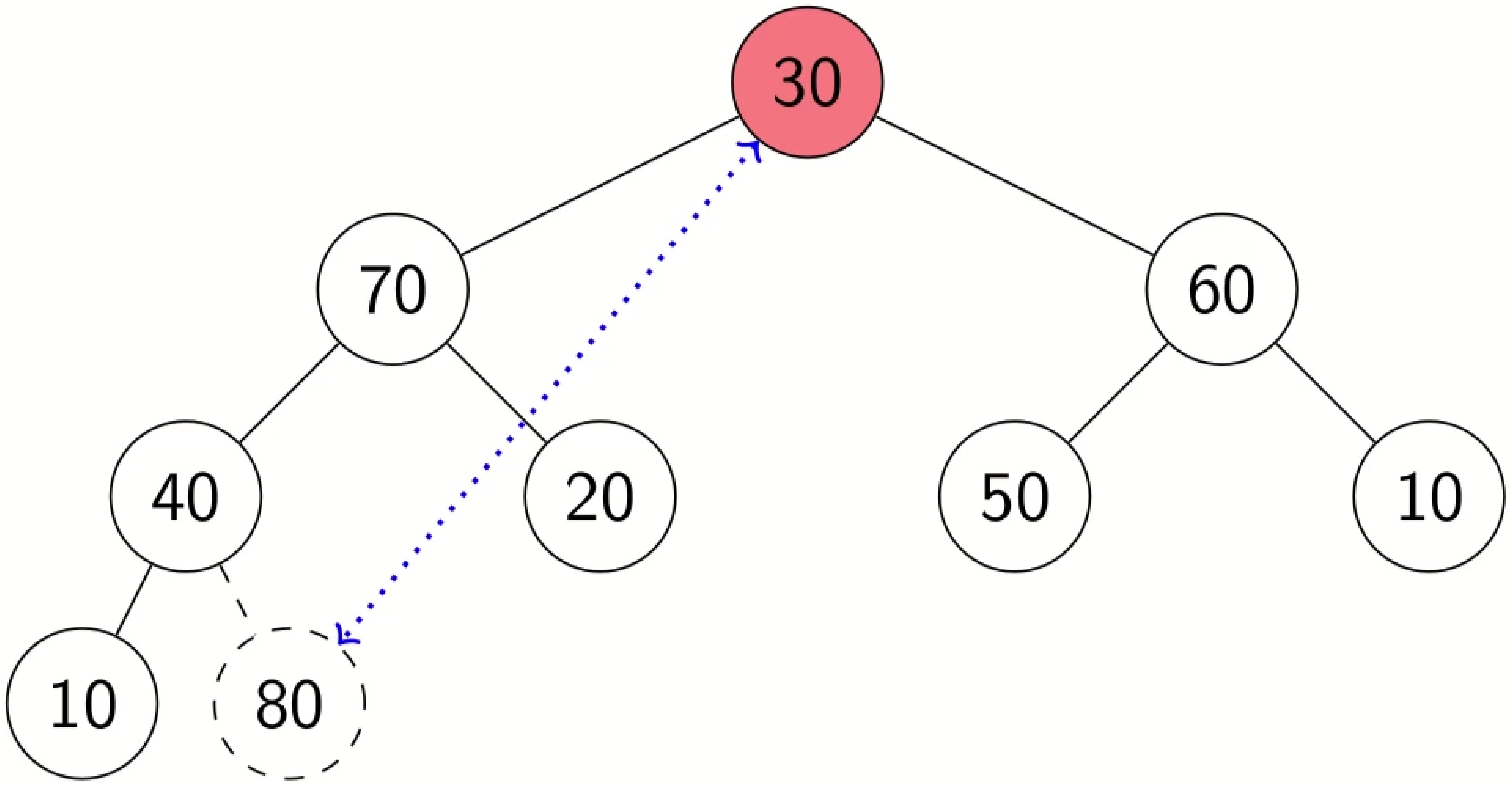
Heapsort example

Continue with the example from heapify:



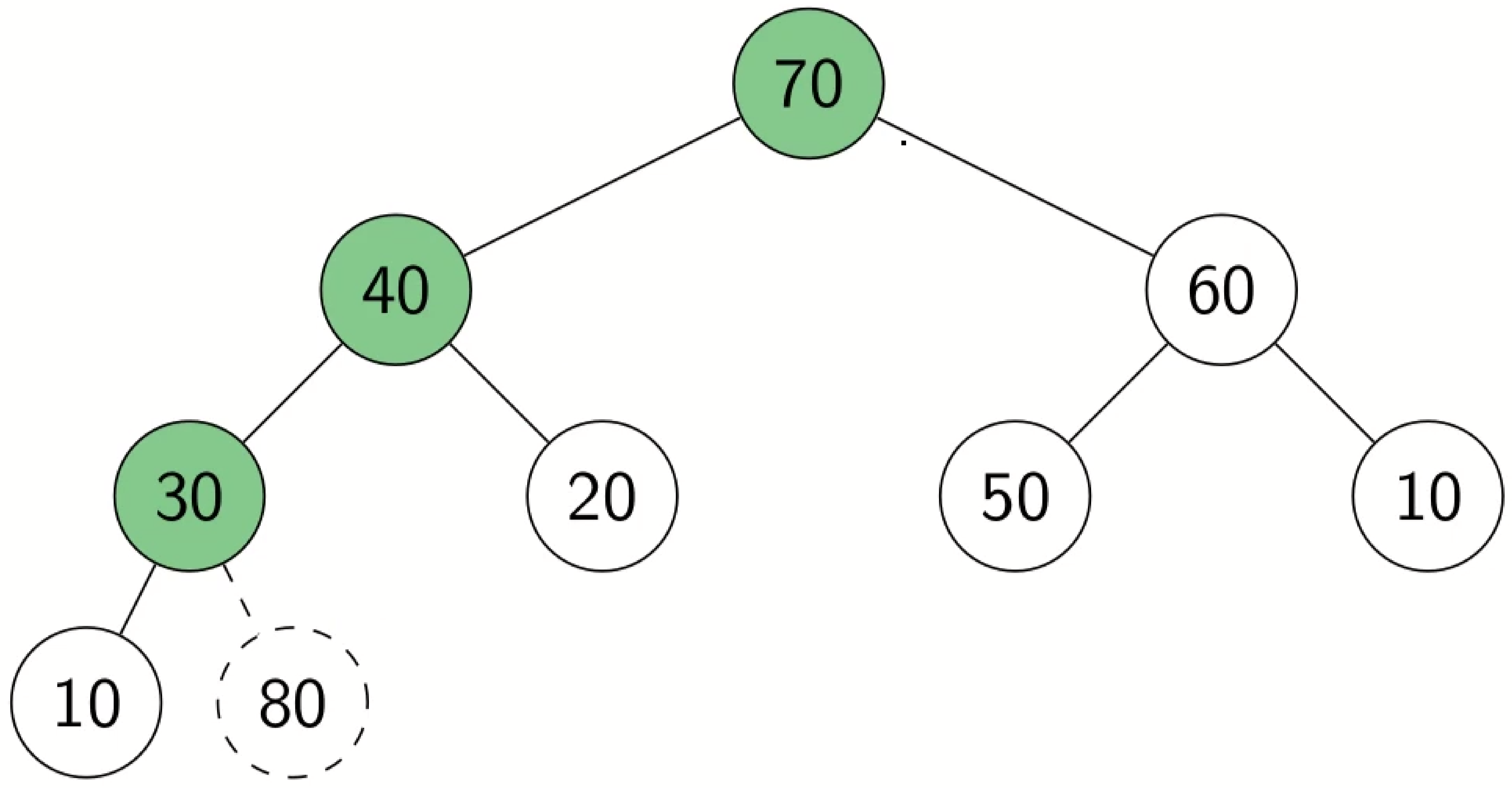
Heapsort example

Continue with the example from heapify:



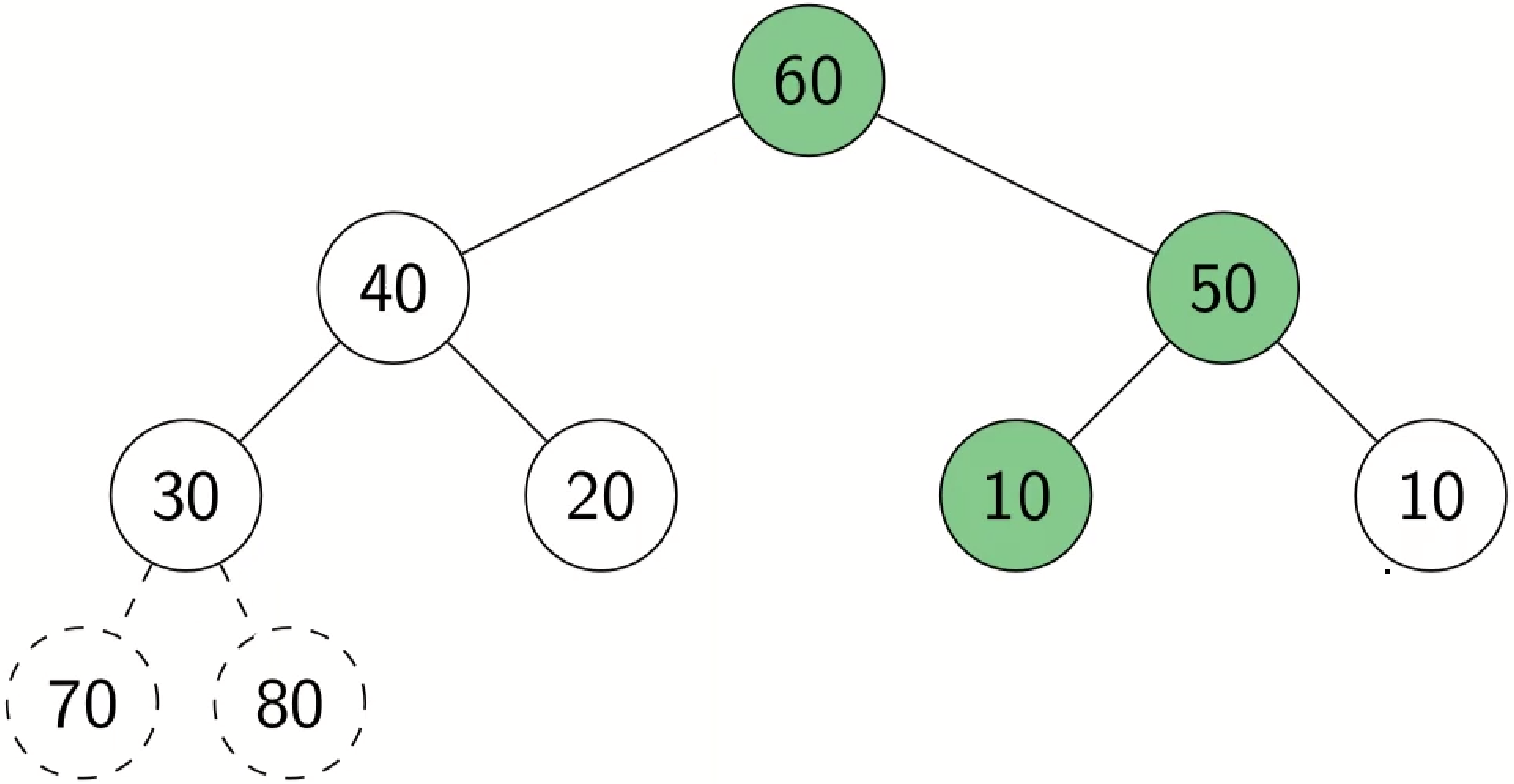
Heapsort example

Continue with the example from heapify:



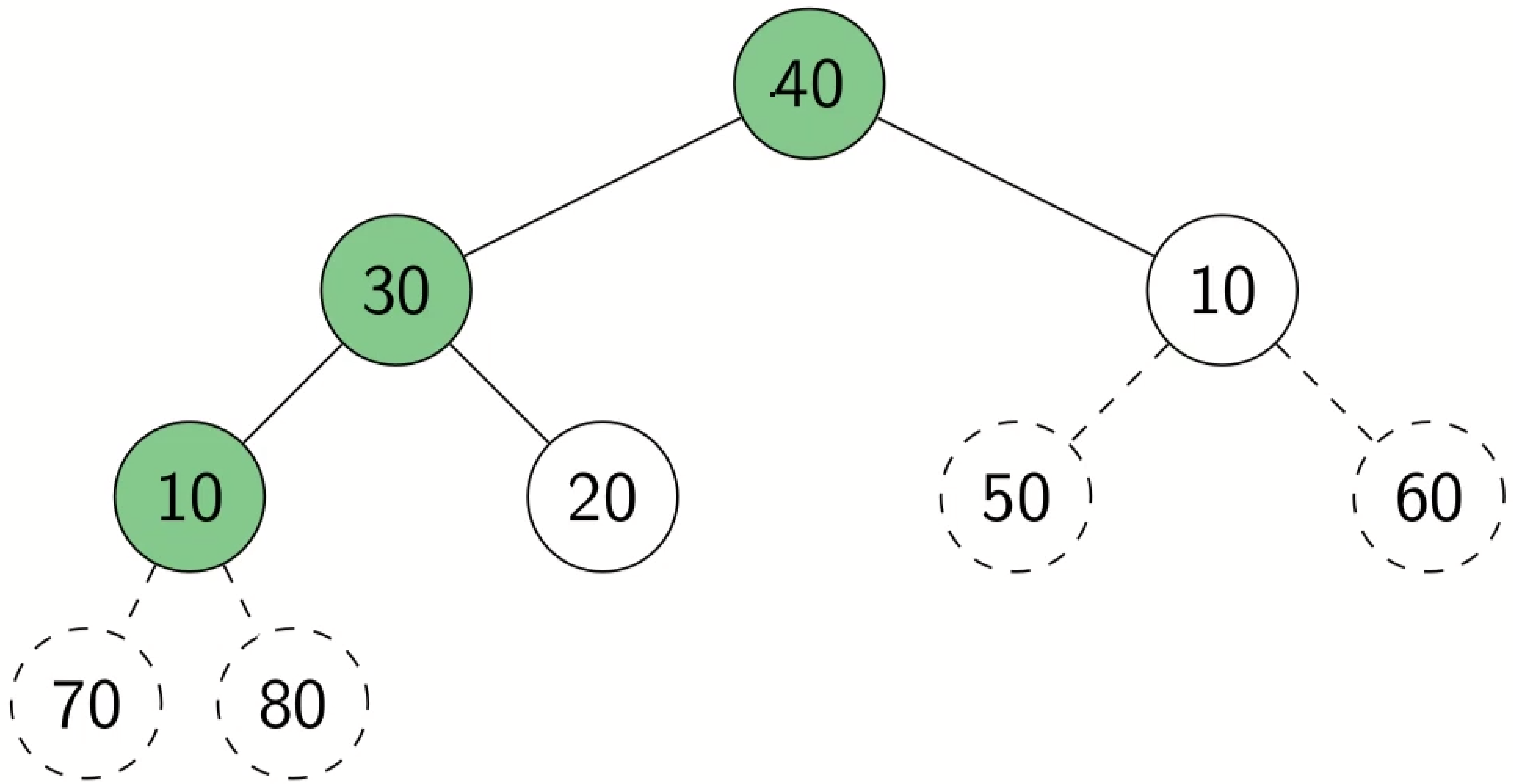
Heapsort example

Continue with the example from heapify:



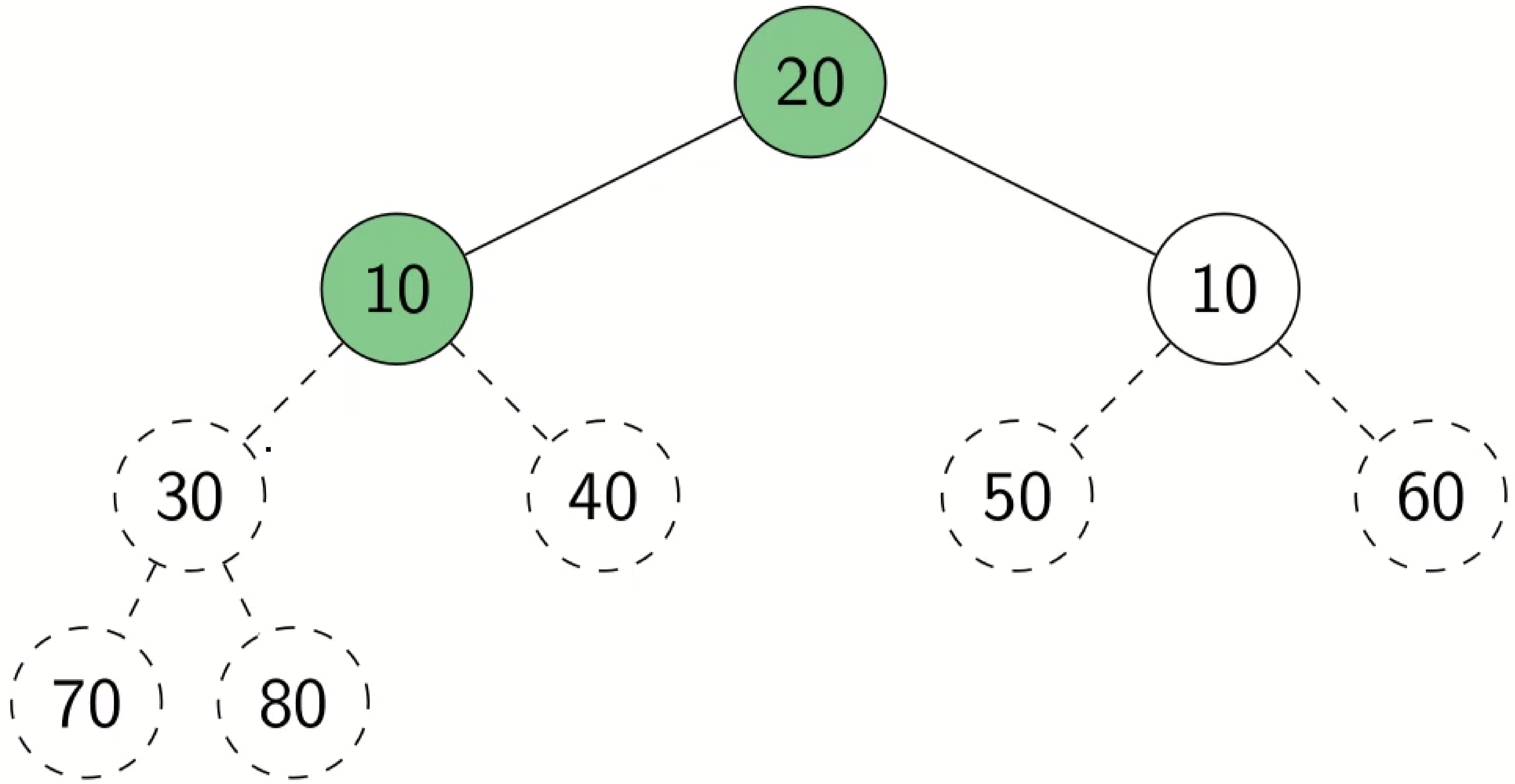
Heapsort example

Continue with the example from heapify:



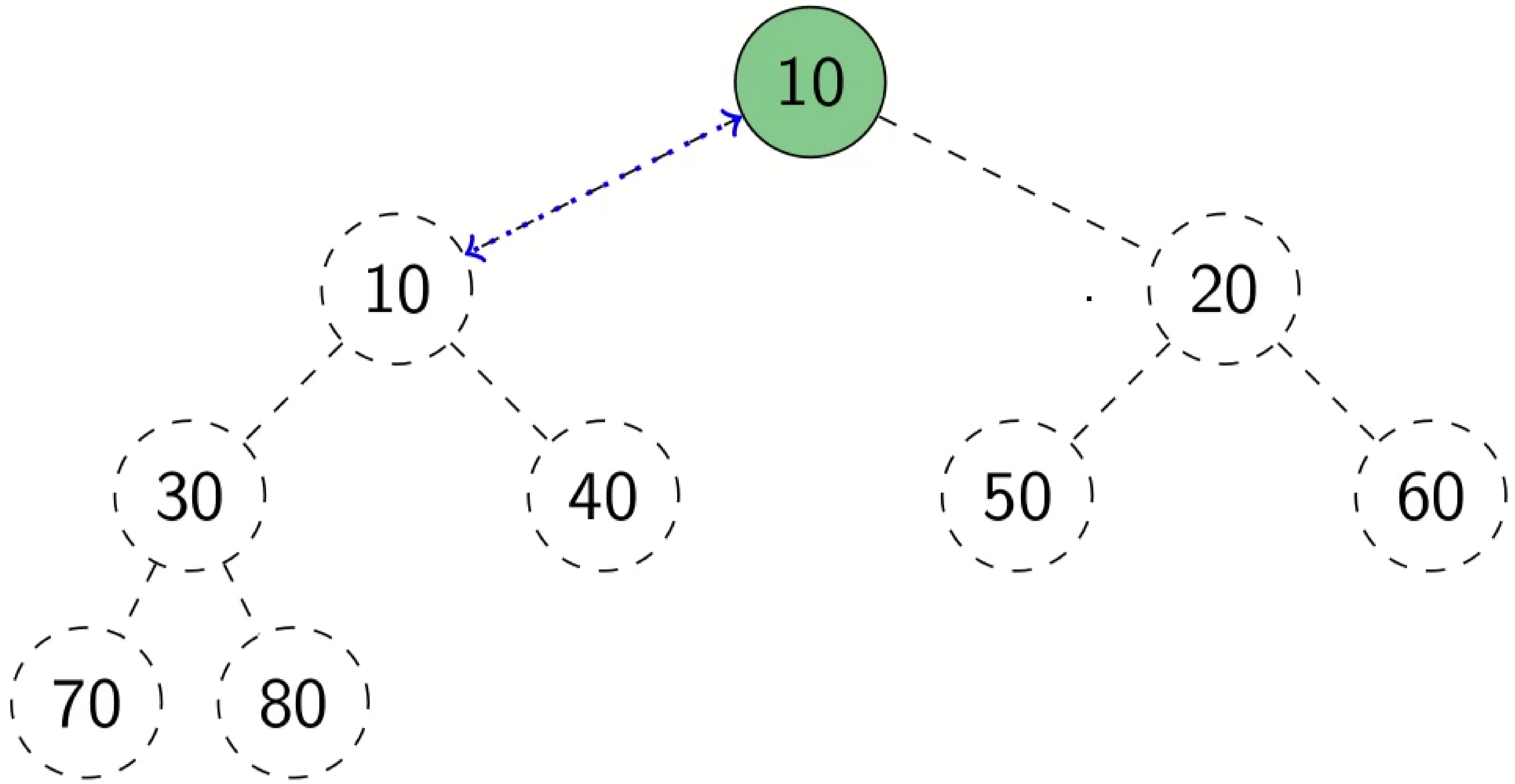
Heapsort example

Continue with the example from heapify:



Heapsort example

Continue with the example from heapify:



The array (i.e., the heap in level-by-level order) is now in sorted order.

Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - *PQ-sort* and *Heapsort*
   - Summary

Heap summary

- **Binary heap**: A binary tree that satisfies structural property and heap-order property.
- Heaps are one possible realization of ADT PriorityQueue:
  - *insert* takes time $O(\log n)$
  - *deleteMax* takes time $O(\log n)$
  - Also supports *findMax* in time $O(1)$
- A binary heap can be built in linear time.
- *PQ-sort* with binary heaps leads to a sorting algorithm with $O(n \log n)$ worst-case run-time ($\rightsquigarrow$ *HeapSort*)
- We have seen here the *max-oriented version* of heaps (the maximum priority is at the root).
- There exists a symmetric *min-oriented version* that supports *insert* and *deleteMin* with the same run-times.

Finding the smallest item

Problem: Find the *kth smallest item* in an array A of n distinct numbers.

Solution 1: Make k passes through the array, deleting the minimum number each time.
Complexity: $\Theta(kn)$.

Solution 2: Sort A, then return $A[k-1]$.
Complexity: $\Theta(n \log n)$.

Solution 3: Scan the array and maintain the k smallest numbers seen so far in a max-heap
Complexity: $\Theta(n \log k)$.

Solution 4: Create a min-heap with *heapify*(A). Call *deleteMin*(A) k times.
Complexity: $\Theta(n + k \log n)$.

$A = [10, 25, 1, 7, 3]$ $\quad k = 2$

10 25 10 7 3

10 1 1 1

$A = [10, 25, 1, 7, 3]$ $\qquad k = 2$

10 25 10 7 (3)

10 1 1 1

Claim: $k \log(n) \le n \log(k)$

Proof: the function $x \mapsto \frac{x}{\log(x)}$ is increasing

$k \le n$ so $\frac{k}{\log(k)} \le \frac{n}{\log(n)}$