CS 240 – Data Structures and Data Management

Module 2: Priority Queues

Leili Rafiee Sevyeri   Éric Schost

Based on lecture notes by many previous cs240 instructors

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

Spring 2023

version 2023−05−29 13:55

Outline

2. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - *PQ-sort* and *Heapsort*
   - Towards the Selection Problem

Outline

2 Priority Queues

- Abstract Data Types
- ADT Priority Queue
- Binary Heaps
- Operations in Binary Heaps
- *PQ-sort* and *Heapsort*
- Towards the Selection Problem

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 (and typically returning) 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

Queue ADT

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

- *enqueue*: inserting an item
- *dequeue*: removing (and typically returning) 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

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 and returning 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

PQ-Sort($A[0..n-1]$)
1. initialize PQ to an empty priority queue
2. **for** $i \leftarrow 0$ **to** $n-1$ **do**
3. $\quad PQ.insert(A[i])$
4. **for** $i \leftarrow n-1$ **down to** 0 **do**
5. $\quad A[i] \leftarrow PQ.deleteMax()$

- Note: Run-time depends on how we implement the priority queue.
- Sometimes written as: $O(initialization + n \cdot insert + n \cdot deleteMax)$

Realizations of Priority Queues

Realization 1: unsorted arrays

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.)

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*.

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*.

Realization 2: sorted arrays

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*.

Realization 2: sorted arrays

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

Using sorted linked lists is identical.
PQ-sort with this realization yields *insertion sort*.

Outline

2 Priority Queues
- Abstract Data Types
- ADT Priority Queue
- Binary Heaps
- Operations in Binary Heaps
- *PQ-sort* and *Heapsort*
- Towards the Selection Problem

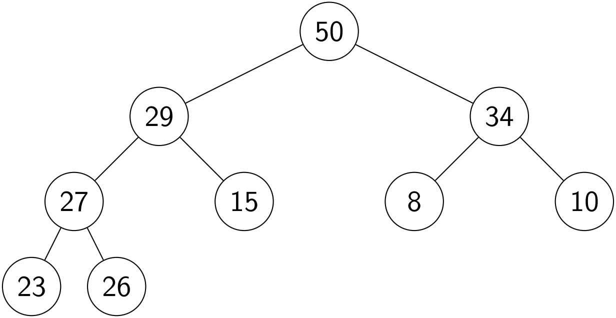
Realization 3: Heaps

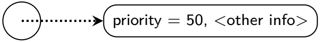
A **(binary) heap** is a certain type of binary tree.

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 $\log(n+1) - 1 \in \Omega(\log n)$.

Example Heap



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

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 $\Theta(\log n)$.

Storing Heaps in Arrays

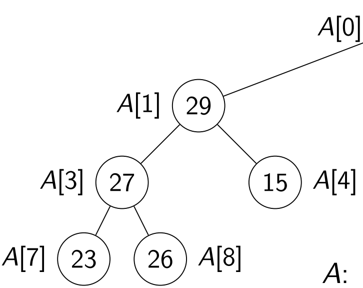
Heaps should *not* be stored as binary trees!

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.

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.



	0	1	2	3	4	5	6	7	8
A:	50	29	34	27	15	8	10	23	26

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 *last* node is $n - 1$ (where n is the size)
- 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 $\lfloor \frac{i-1}{2} \rfloor$
- these nodes exist if the index falls in the range $\{0, \ldots, n-1\}$

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 *last* node is $n-1$ (where n is the size)
- 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 $\lfloor \frac{i-1}{2} \rfloor$
- these nodes exist if the index falls in the range $\{0, \ldots, n-1\}$

We should hide implementation details using helper-functions!

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

Some of these helper-functions need to know n (but we omit this in the code for simplicity).

Outline

2 Priority Queues
- Abstract Data Types
- ADT Priority Queue
- Binary Heaps
- Operations in Binary Heaps
- *PQ-sort* and *Heapsort*
- Towards the Selection Problem

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

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

Insert in Heaps

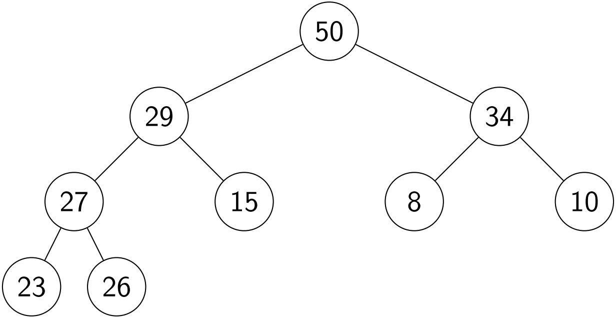
- Place the new key at the first free leaf
- The heap-order property might be violated: perform a *fix-up*:

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

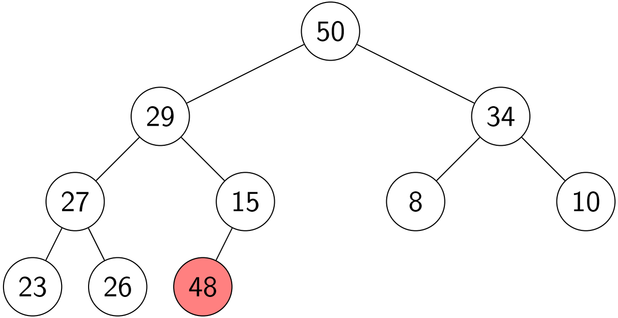
The new item "bubbles up" until it reaches its correct place in the heap.

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

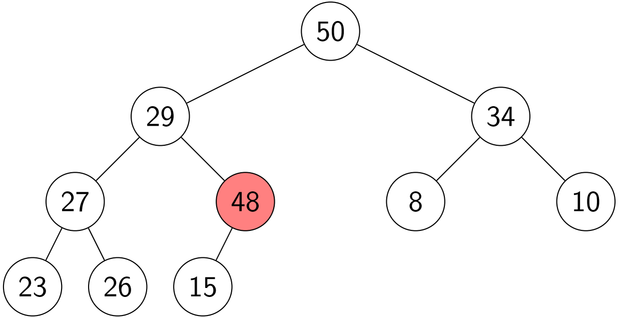
fix-up example



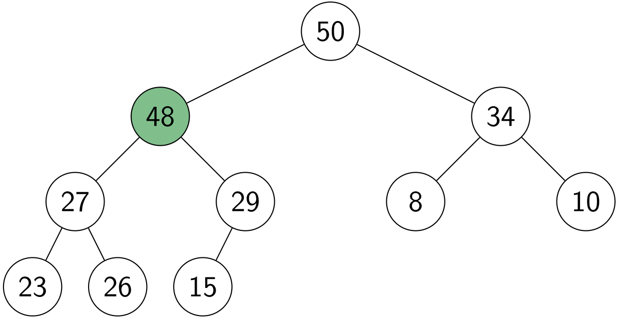
fix-up example



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*:

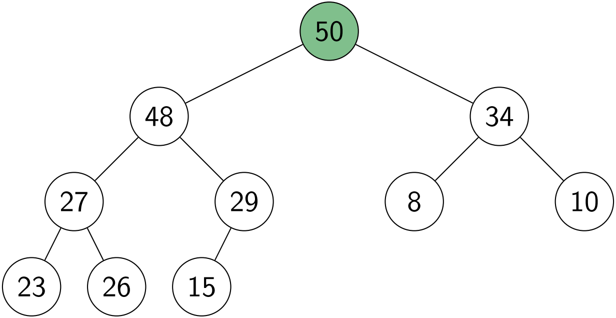
deleteMax in Heaps

- 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*:

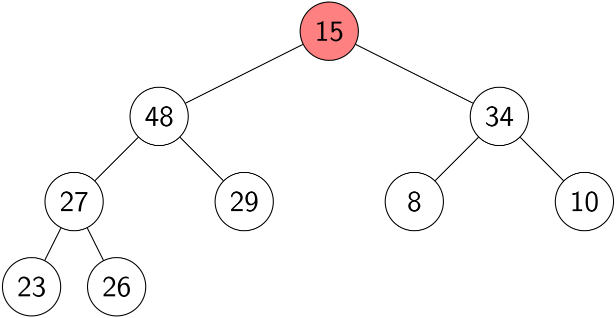
```
fix-down(A, i, n)
A: an array that stores a heap of size n
i: an index corresponding to a node of the heap
    while i is not a leaf do
        j ← left child of i        // Find the child with the larger key
        if (i has right child and A[right child of i].key > A[j].key)
            j ← right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

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

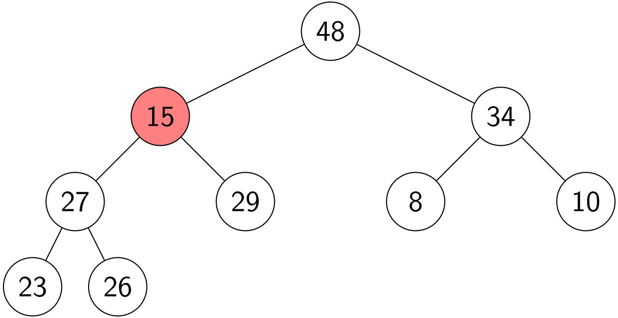
deleteMax example



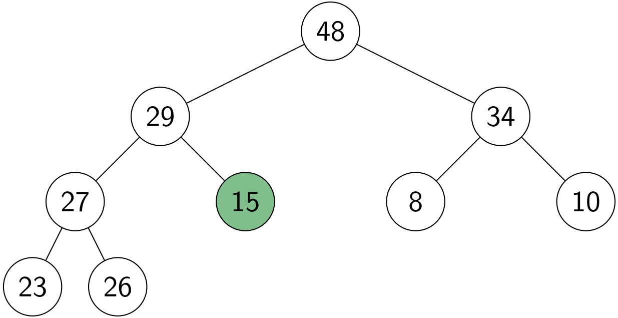
deleteMax example



deleteMax example



deleteMax example



Priority Queue Realization Using Heaps

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

insert(x)

increase *size*
$\ell \leftarrow$ *last*$()$
$A[\ell] \leftarrow x$
fix-up(A, ℓ)

deleteMax$()$

$\ell \leftarrow$ *last*$()$
swap $A[$*root*$()]$ and $A[\ell]$
decrease *size*
fix-down$(A,$ *root*$(),$ *size*$)$
return $A[\ell]$

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

Outline

2. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - *PQ-sort* and *Heapsort*
   - Towards the Selection Problem

Sorting using heaps

- Recall: Any priority queue can be used to *sort* in time

$$O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{deleteMax})$$

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

```
PQsortWithHeaps(A)
    initialize H to an empty heap
    for i ← 0 to n − 1 do
        H.insert(A[i])
    for i ← n − 1 down to 0 do
        A[i] ← H.deleteMax()
```

Sorting using heaps

- Recall: Any priority queue can be used to *sort* in time

$$O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{deleteMax})$$

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

```
PQsortWithHeaps(A)
    initialize H to an empty heap
    for i ← 0 to n − 1 do
        H.insert(A[i])
    for i ← n − 1 down to 0 do
        A[i] ← H.deleteMax()
```

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

⇝ *PQ-Sort* 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.

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 ← 0 to A.size() − 1 do
3.        H.insert(A[i])
```

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 ← 0 to A.size() − 1 do
3.        H.insert(A[i])
```

This corresponds to doing *fix-ups*
Worst-case running time: $\Theta(n \log n)$.

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.

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
    n ← A.size()
    for i ← parent(last()) downto root() do
        fix-down(A, i, n)
```

Building Heaps with Fix-down

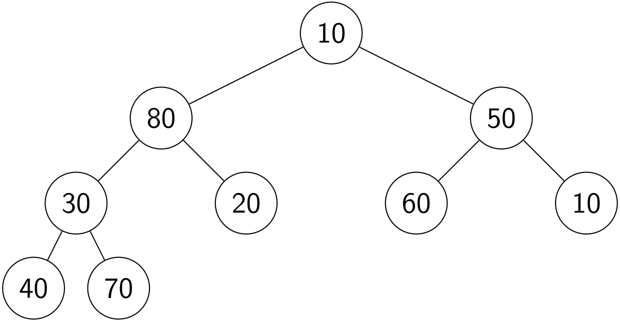
Problem: Given n items all at once (in $A[0 \cdots n-1]$) build a heap containing all of them.

Solution 2: Using *fix-downs* instead:

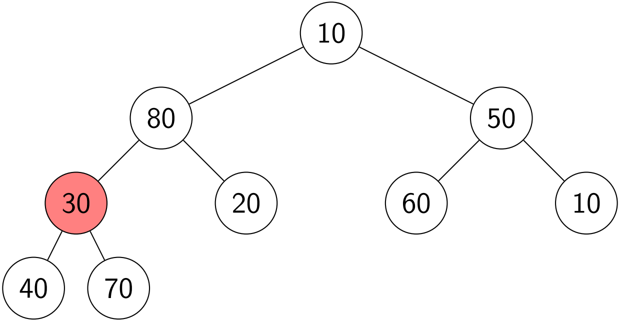
heapify(A)
A: an array
1. $n \leftarrow A.size()$
2. **for** $i \leftarrow$ *parent*(*last*()) **downto** *root*() **do**
3. *fix-down*(A, i, n)

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

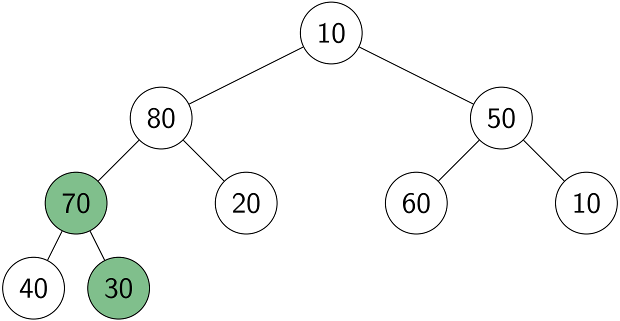
heapify example



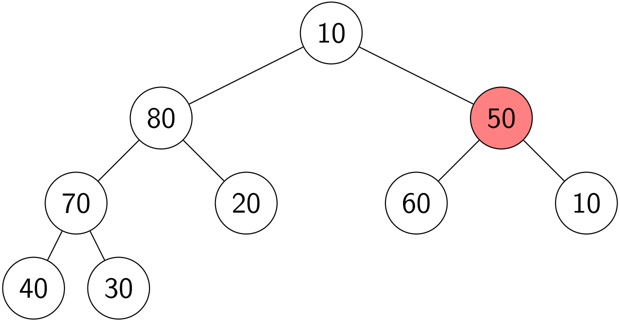
heapify example



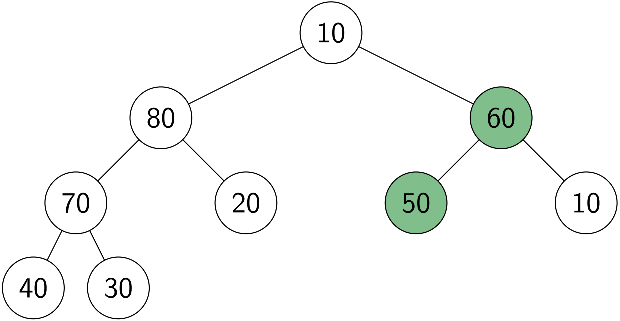
heapify example



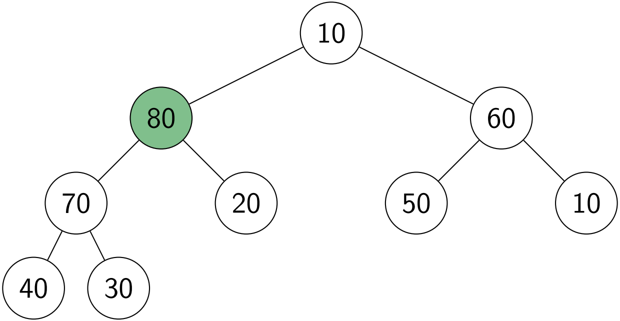
heapify example



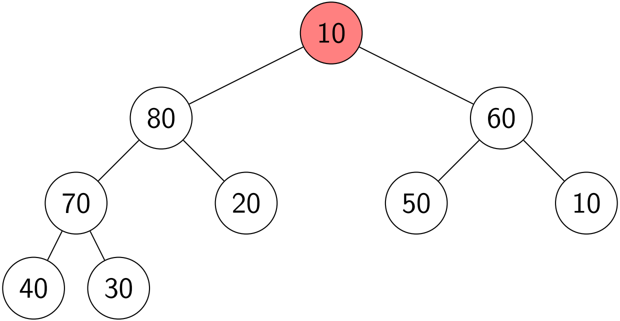
heapify example



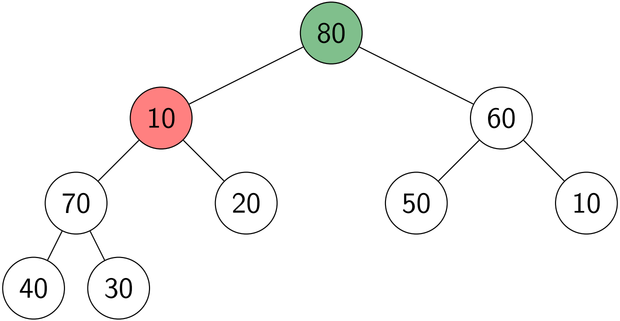
heapify example



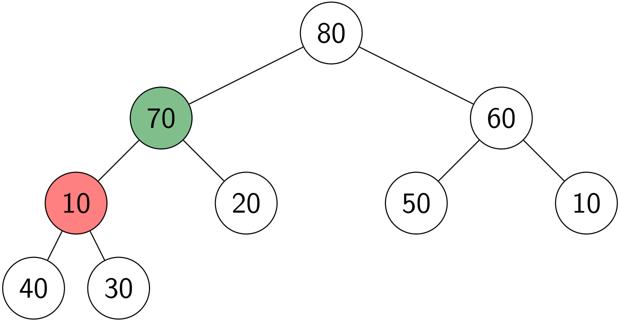
heapify example



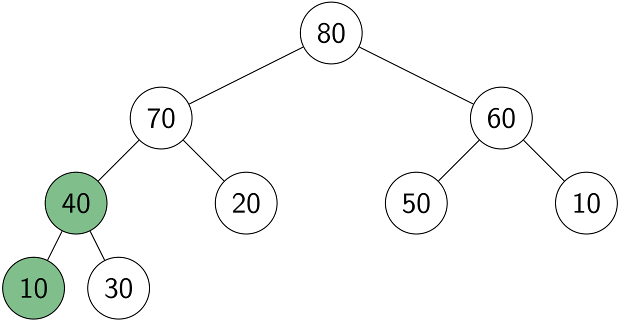
heapify example



heapify example



heapify example



Efficient sorting with heaps

- Idea: *PQ-sort* with heaps.
- $O(1)$ auxiliary space: Use same input-array A for storing heap.

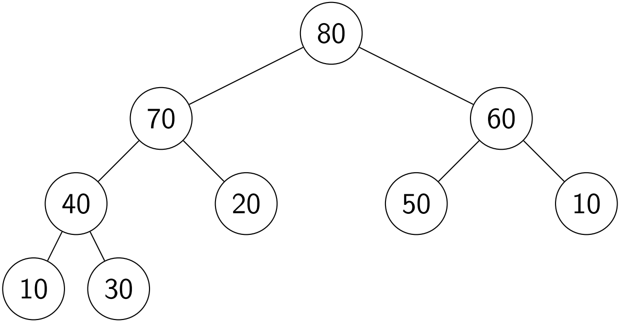
```
HeapSort(A, n)
    // heapify
    n ← A.size()
    for i ← parent(last()) downto 0 do
        fix-down(A, i, n)

    // repeatedly find maximum
    while n > 1
        // 'delete' maximum by moving to end and decreasing n
        swap items at A[root()] and A[last()]
        decrease n
        fix-down(A, root(), n)
```

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

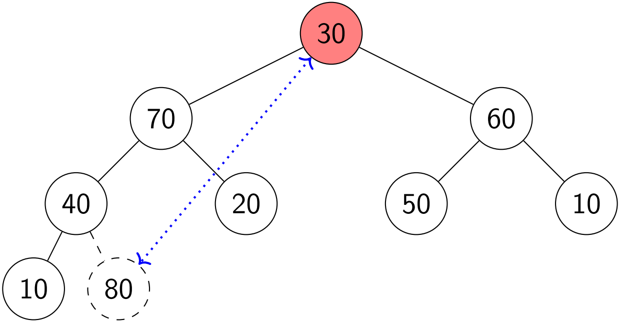
Heapsort example

Continue with the example from heapify:



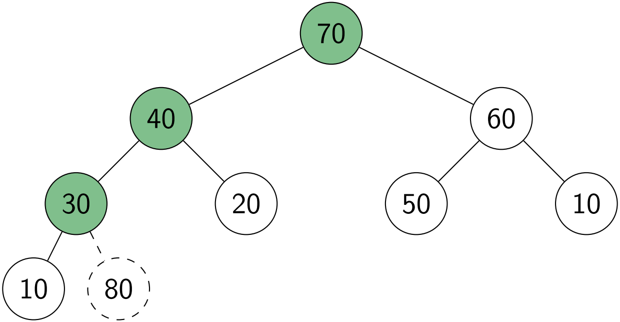
Heapsort example

Continue with the example from heapify:



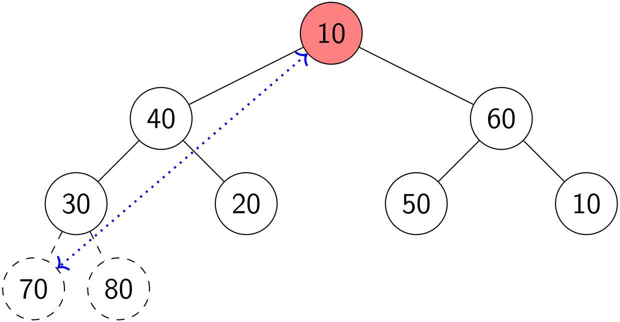
Heapsort example

Continue with the example from heapify:



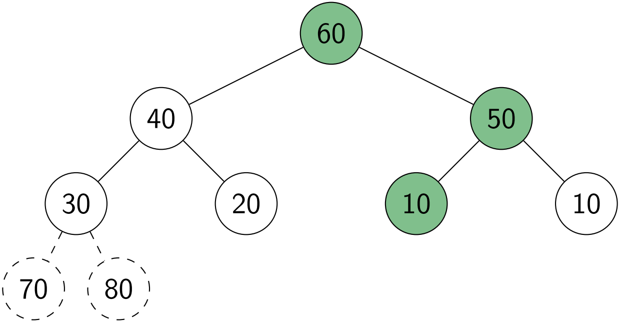
Heapsort example

Continue with the example from heapify:



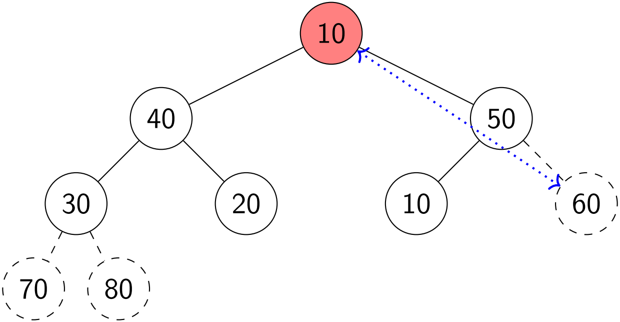
Heapsort example

Continue with the example from heapify:



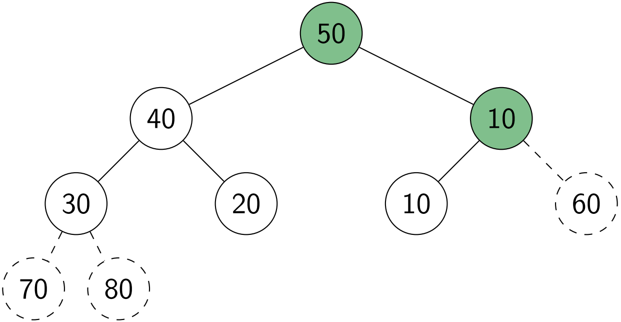
Heapsort example

Continue with the example from heapify:



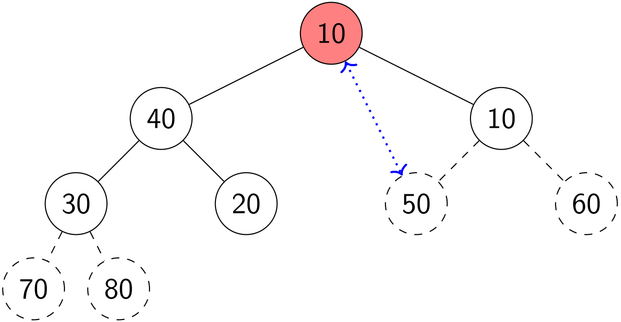
Heapsort example

Continue with the example from heapify:



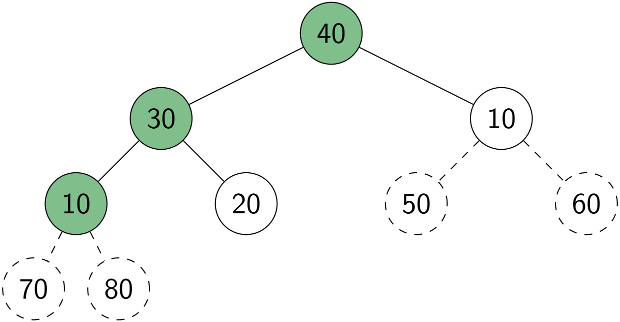
Heapsort example

Continue with the example from heapify:



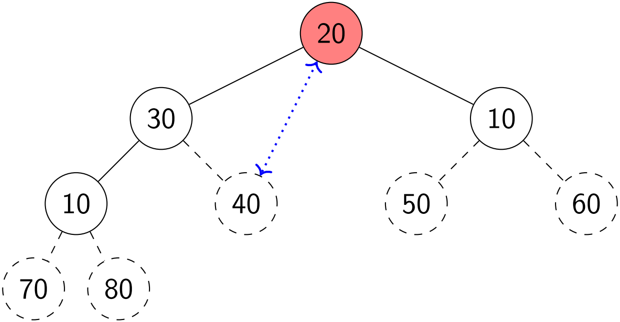
Heapsort example

Continue with the example from heapify:



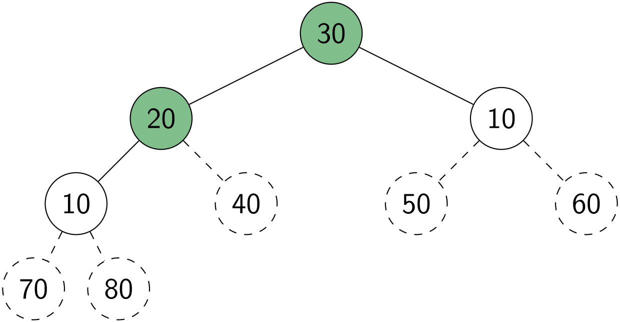
Heapsort example

Continue with the example from heapify:



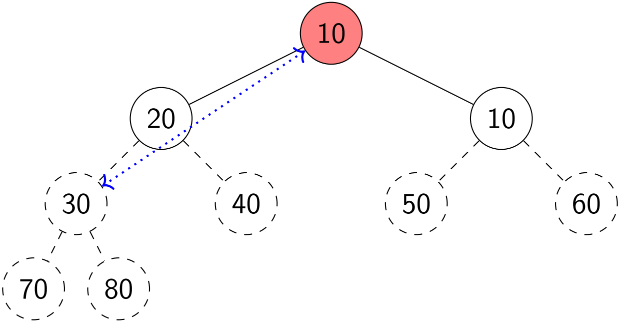
Heapsort example

Continue with the example from heapify:



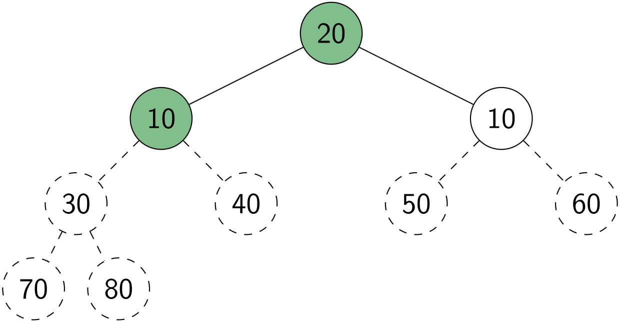
Heapsort example

Continue with the example from heapify:



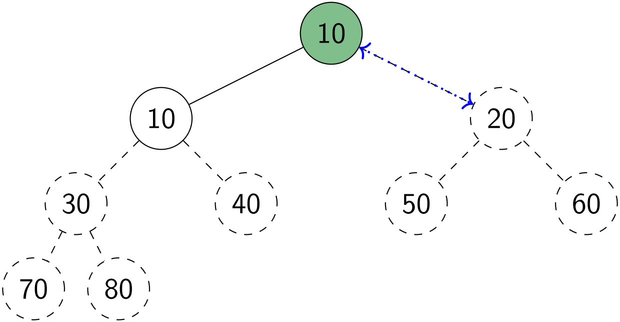
Heapsort example

Continue with the example from heapify:



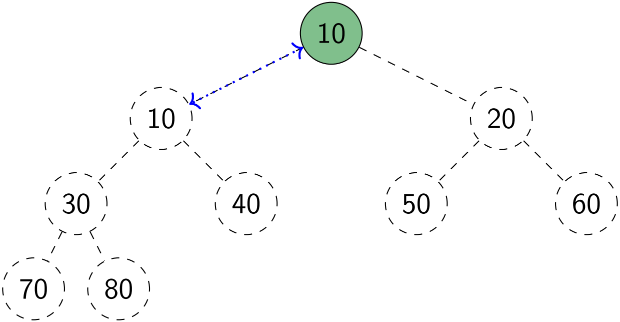
Heapsort example

Continue with the example from heapify:



Heapsort example

Continue with the example from heapify:



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

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.

Outline

Finding the smallest items

Problem: Find the *kth smallest item* in an array A of n distinct numbers ($k = 0, \ldots, n-1$)

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

Solution 2: Make $k+1$ passes through the array, deleting the minimum number each time.
Complexity: $\Theta(kn)$.

Solution 3: Scan the array and maintain the $k+1$ 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) $k+1$ times.
Complexity: $\Theta(n + k \log n)$.