CS 240 – Data Structures and Data Management

Module 2: Priority Queues

Olga Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version 2025–01–09 15:48

Outline

2. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps as PQ realization
   - *PQ-sort* and *heap-sort*
   - Towards the Selection Problem

Outline

2. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps as PQ realization
   - *PQ-sort* and *heap-sort*
   - Towards the Selection Problem

Abstract Data Types (review)

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

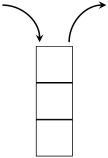
The information is accessed *only* through the operations.

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

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

ADT Stack (review)

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



- *push*: Add an item to the stack.
- *pop*: Remove and return the most recently added item.

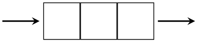
Items are removed in LIFO (*last-in first-out*) order.

We can have extra operations: *size*, *is-empty*, and *top*

ADT Stack can easily be realized using arrays or linked lists such that operations taking constant time (exercise).

ADT Queue (review)

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



- *enqueue* (or *append* or *add-back*): Add an item to the queue.
- *dequeue* (or *remove-front*): Remove and return the least recently inserted item.

Items are removed in FIFO (*first-in first-out*) order.

We can have extra operations: *size*, *is-empty*, and *peek*/*front*

ADT Queue can easily be realized using (circular) arrays or linked lists such that operations taking constant time (exercise).

Outline

2 Priority Queues

- Abstract Data Types
- ADT Priority Queue
- Binary Heaps as PQ realization
- *PQ-sort* and *heap-sort*
- Towards the Selection Problem

ADT Priority Queue

Priority Queue generalizes both ADT Stack and ADT Queue.

It is a collection of items (each having a **priority** or **key**) with operations

- *insert*: inserting an item tagged with a priority
- *delete-max*: removing and returning an item of *highest* priority.

We can have extra operations: *size*, *is-empty*, and *get-max*

This is a **maximum-oriented** priority queue. A **minimum-oriented** priority queue replaces operation *delete-max* by *delete-min*.

Applications:

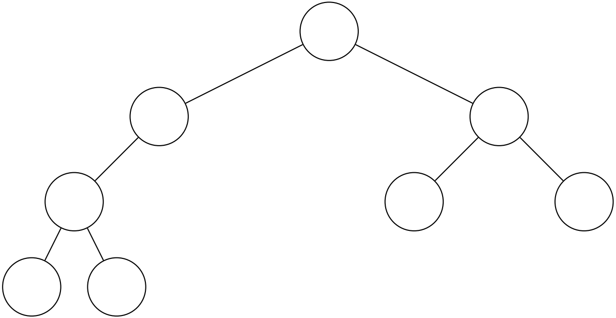
- How would you simulate a stack with a priority queue?
- How would you simulate a queue with a priority queue?
- Other applications: typical todo-list, simulation systems, sorting

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. $\quad PQ.\mathit{insert}$(an item with priority $A[i]$)
4. **for** $i \leftarrow n-1$ **down to** 0 **do**
5. $\quad A[i] \leftarrow$ priority of $PQ.\mathit{delete\text{-}max}()$

- Note: Run-time depends on how we implement the priority queue.
- Sometimes written as: $O(\mathit{initialization} + n \cdot \mathit{insert} + n \cdot \mathit{delete\text{-}max})$

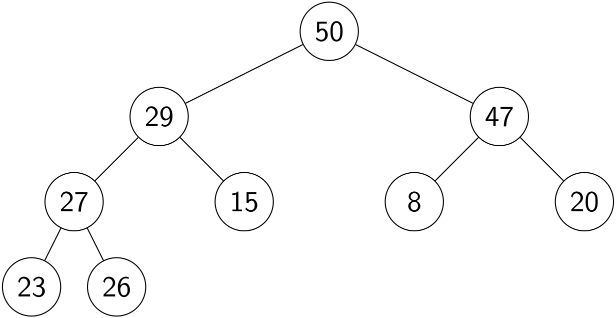
Example Binary Tree and Heap



Binary tree with

1. structural property and

Example Binary Tree and Heap



Binary tree with

1. structural property and
2. heap-order property.

Recall: 15 represents

priority $=$ 15, <other info>

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

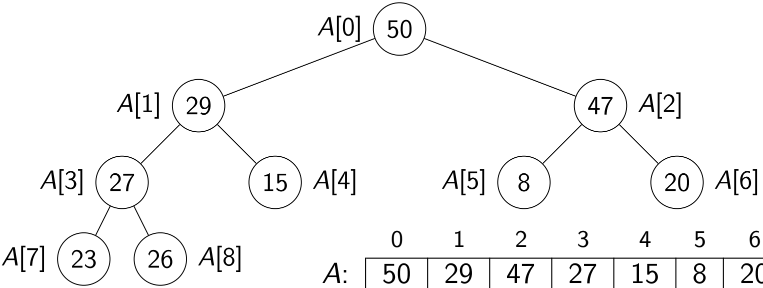
Heaps should *not* be stored as binary trees!

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	47	27	15	8	20	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.

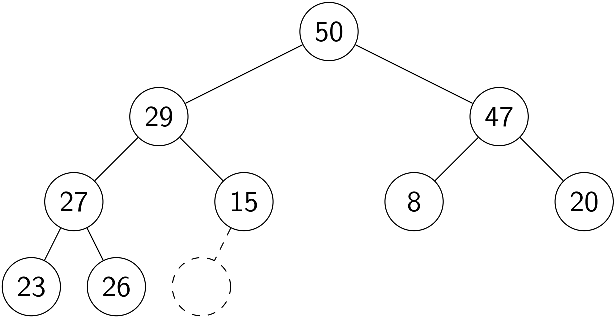
Some of these helper-functions need to know the size n. We assume that the data structure stores this explicitly.

Outline

2. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps as PQ realization
   - *PQ-sort* and *heap-sort*
   - Towards the Selection Problem

insert in Heaps

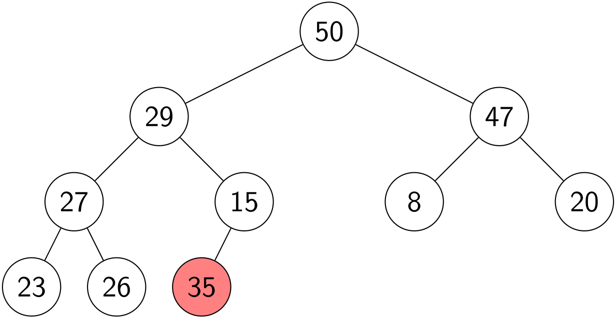
insert(35):



- By structural property: no choice where the new node can go.

insert in Heaps

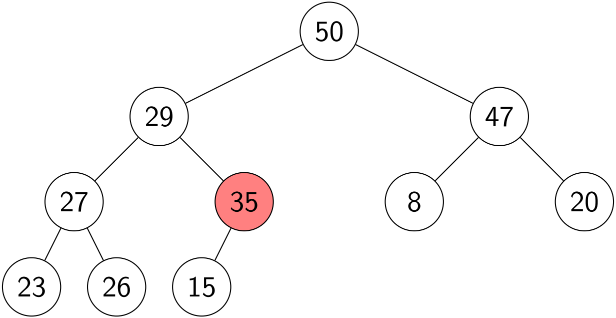
insert(35):



- By structural property: no choice where the new node can go.
- This may or may not lead to heap-order violations.

insert in Heaps

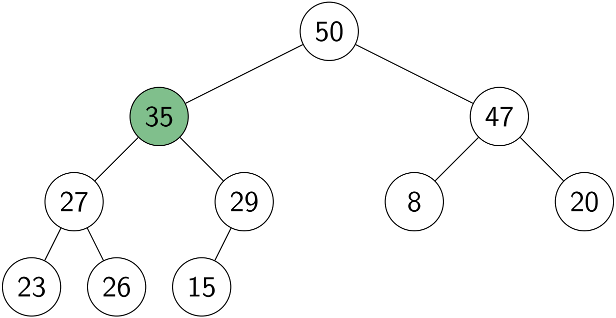
insert(35):



- By structural property: no choice where the new node can go.
- This may or may not lead to heap-order violations.
- Fix violations by "bubbling up" in the tree.

insert in Heaps

insert(35):



- By structural property: no choice where the new node can go.
- This may or may not lead to heap-order violations.
- Fix violations by “bubbling up” in the tree.

insert in Heaps

- Place the new key at the first free leaf
- Use *fix-up* to restore heap-order.

```
insert(x)
1.  ℓ ← last()+1
2.  A[ℓ] ← x            // assume dynamic array used
3.  increase size       // size:  stored by PQ
4.  fix-up(A, ℓ)
```

insert in Heaps

- Place the new key at the first free leaf
- Use *fix-up* to restore heap-order.

insert(x)
1. $\ell \leftarrow last() + 1$
2. $A[\ell] \leftarrow x$ `// assume dynamic array used`
3. increase *size* `//` *size*`: stored by PQ`
4. *fix-up*(A, ℓ)

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. $i \leftarrow parent(i)$

insert in Heaps

- Place the new key at the first free leaf
- Use *fix-up* to restore heap-order.

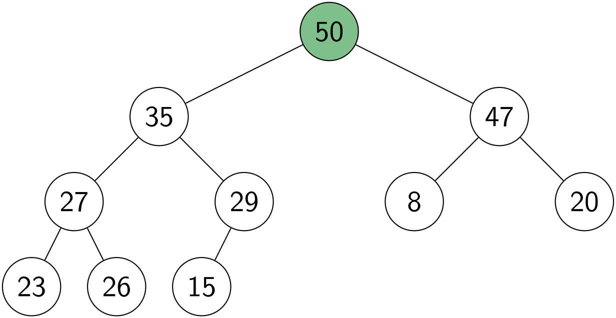
insert(x)
1. $\ell \leftarrow$ *last*$()+1$
2. $A[\ell] \leftarrow x$ `// assume dynamic array used`
3. increase *size* `//` *size*`: stored by PQ`
4. *fix-up*(A, ℓ)

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. $i \leftarrow parent(i)$

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

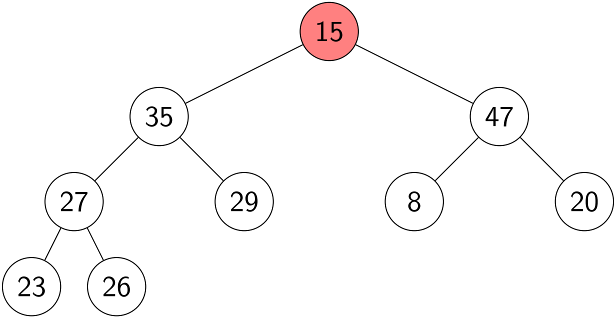
(Correctness may seem obvious, but is actually non-trivial.)

delete-max in Heaps



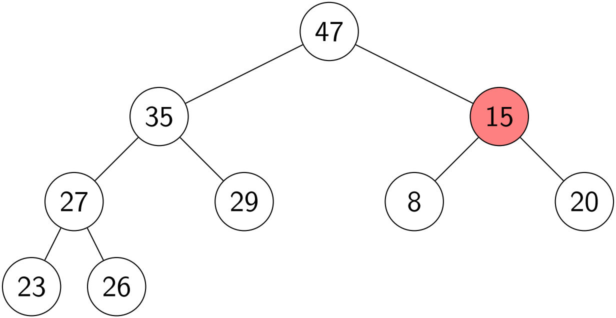
- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

delete-max in Heaps



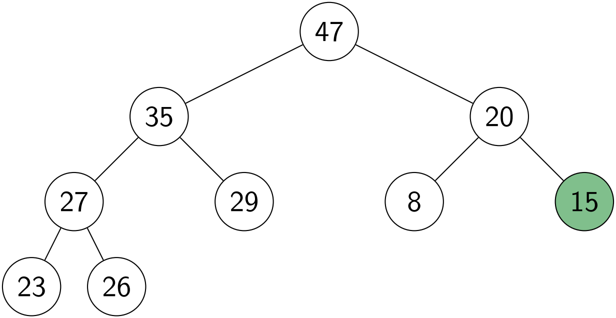
- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

delete-max 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*:

delete-max 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*:

delete-max in Heaps

delete-max in Heaps

fix-down(A, i)
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**
  $j \leftarrow$ left child of i `// find child with larger key`
  if (i has right child and A[right child of i].*key* $> A[j]$.*key*)
    $j \leftarrow$ right child of i
  if $A[i]$.*key* $\geq A[j]$.*key* **break**
  swap $A[j]$ and $A[i]$
  $i \leftarrow j$

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

Priority Queue Realization Using Heaps

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

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

Priority Queue Realization Using Heaps

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

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

Binary heap are a realization of priority queues where the operations *insert* and *delete-max* take $\Theta(\log n)$ **time**.

Outline

2. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps as PQ realization
   - *PQ-sort* and *heap-sort*
   - Towards the Selection Problem

Sorting using heaps

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

$$O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{delete-max})$$

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

```
PQ-sort-with-heaps(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.delete-max()
```

Sorting using heaps

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

$$O(\mathit{initialization} + n \cdot \mathit{insert} + n \cdot \mathit{delete\text{-}max})$$

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

PQ-sort-with-heaps(A)
1. initialize H to an empty heap
2. **for** $i \leftarrow 0$ **to** $n-1$ **do**
3. $\quad H.\mathit{insert}(A[i])$
4. **for** $i \leftarrow n-1$ **down to** 0 **do**
5. $\quad A[i] \leftarrow H.\mathit{delete\text{-}max}()$

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

$\rightsquigarrow$ *PQ-sort* using heaps takes $O(n \log n)$ time (and this is tight).

Sorting using heaps

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

$$O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{delete-max})$$

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

```
PQ-sort-with-heaps(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.delete-max()
```

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

$\rightsquigarrow$ *PQ-sort* using heaps takes $O(n \log n)$ time (and this is tight).

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

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:

```
simple-heap-building(A)
A: an array
    initialize H as an empty heap
    for i ← 0 to A.size() − 1 do
        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:

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

This corresponds to doing *fix-ups*
Worst-case running time: $O(n \log n)$ (and this is tight).

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

$n \leftarrow A.size()$
for $i \leftarrow$ *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:

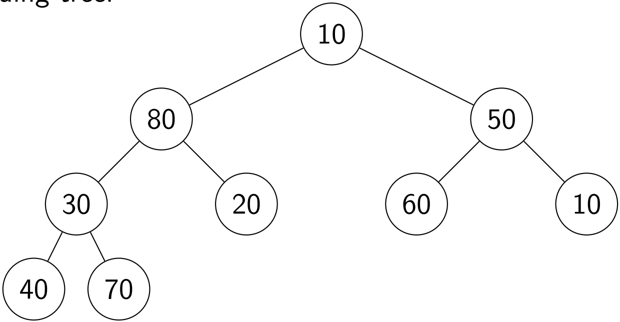
```
heapify(A)
A: an array
1.   n ← A.size()
2.   for i ← parent(last()) downto root() do
3.        fix-down(A, i, n)
```

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

heapify example

	0	1	2	3	4	5	6	7	8
A :	10	80	50	30	20	60	10	40	70

Corresponding tree:

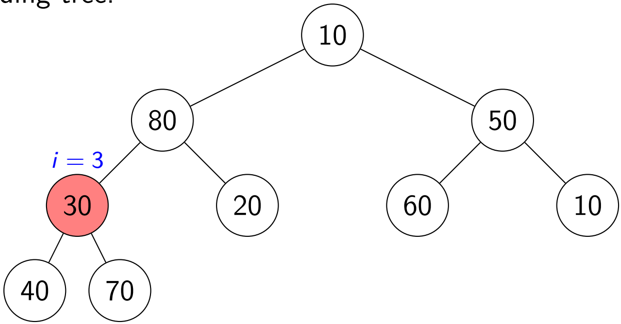


heapify example

A :

0	1	2	3	4	5	6	7	8
10	80	50	30	20	60	10	40	70

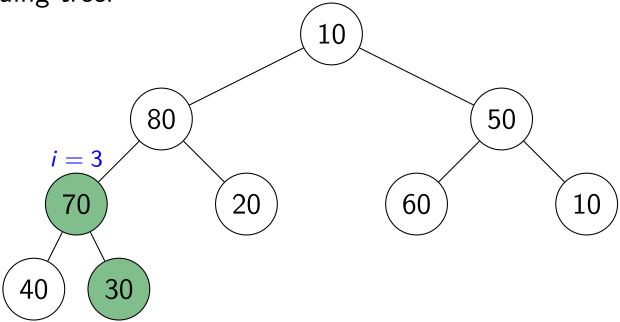
Corresponding tree:



heapify example

	0	1	2	3	4	5	6	7	8
A :	10	80	50	70	20	60	10	40	30

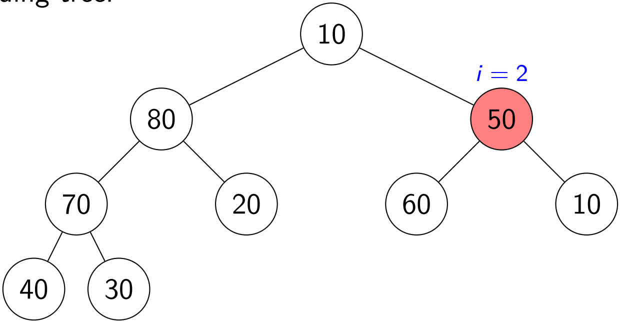
Corresponding tree:



heapify example

	0	1	2	3	4	5	6	7	8
A :	10	80	50	70	20	60	10	40	30

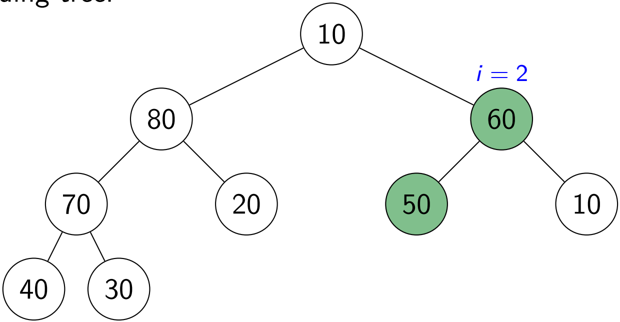
Corresponding tree:



heapify example

	0	1	2	3	4	5	6	7	8
A :	10	80	60	70	20	50	10	40	30

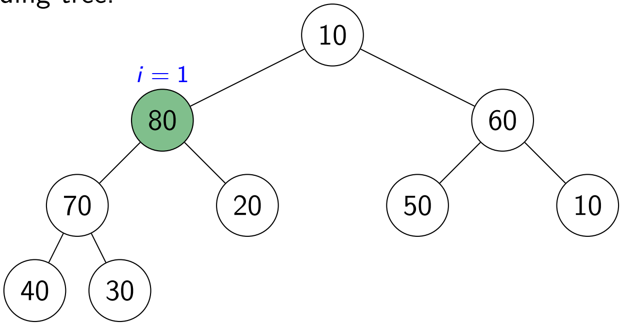
Corresponding tree:



heapify example

A:	0	1	2	3	4	6	7	8
	10	80	60	70	20	10	40	30

Corresponding tree:

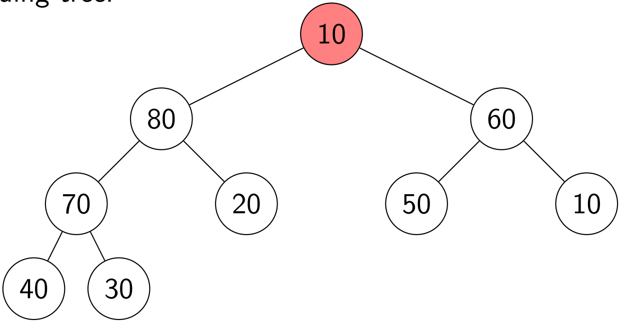


heapify example

A :

0	1	2	3	4	6	7	8
10	80	60	70	20	10	40	30

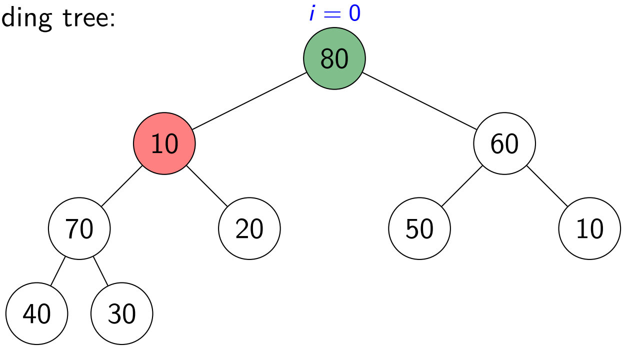
Corresponding tree:



heapify example

	0	1	2	3	4	6	7	8
A :	80	10	60	70	20	10	40	30

Corresponding tree:

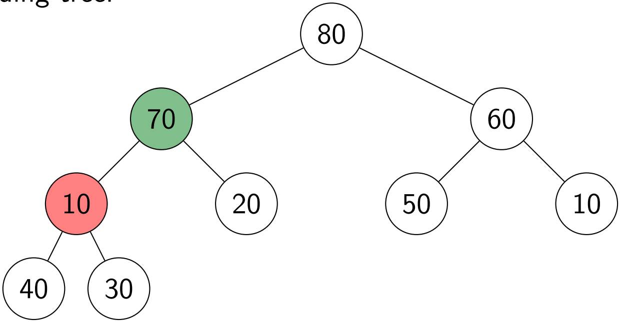


heapify example

	0	1	2	3	4	6	7	8
A :	80	70	60	10	20	10	40	30

Corresponding tree:

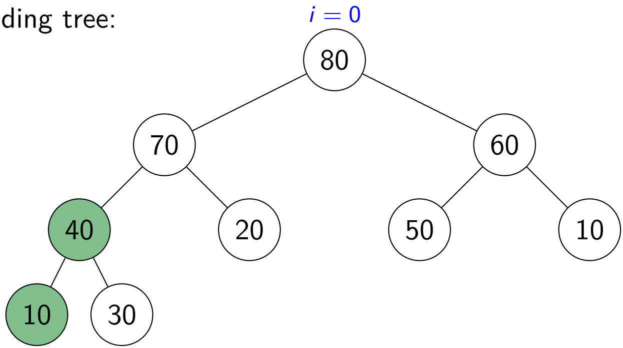
$i = 0$



heapify example

	0	2	3	4	6	7	8
A :	80	60	40	20	10	10	30

Corresponding tree:



Efficient sorting with heaps

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

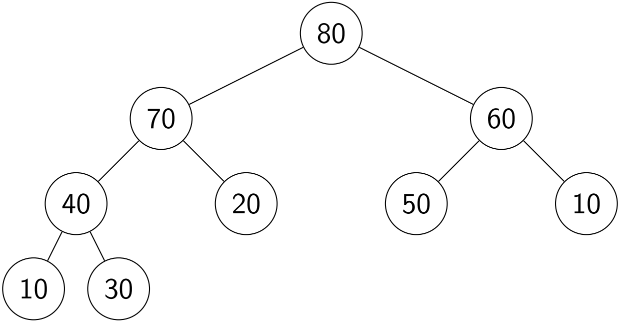
```
heap-sort(A)
  // heapify
  n ← A.size()
  for i ← parent(last()) downto 0 do
        fix-down(A, i, n)

  // 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 $\Theta(n \log n)$ time.

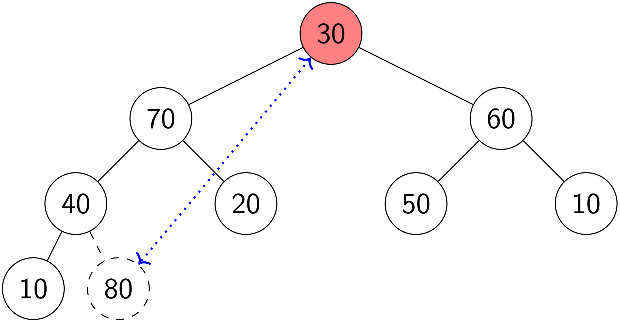
heap-sort example

Continue with the example from heapify:



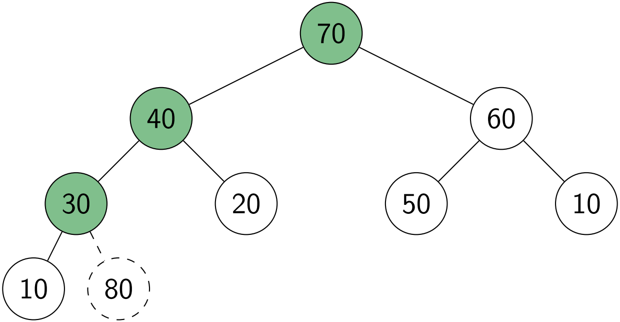
heap-sort example

Continue with the example from heapify:



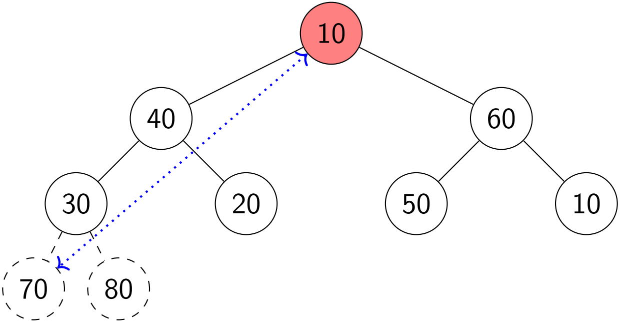
heap-sort example

Continue with the example from heapify:



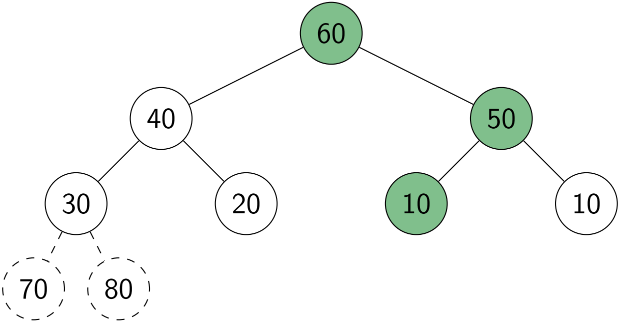
heap-sort example

Continue with the example from heapify:



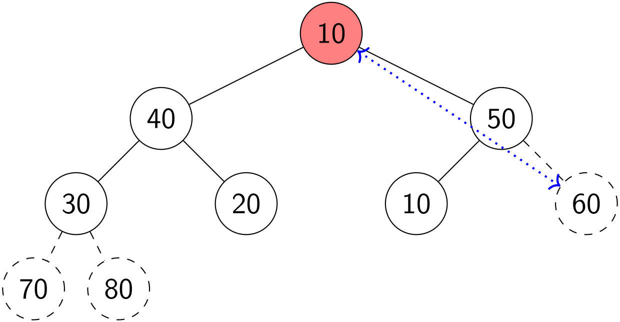
heap-sort example

Continue with the example from heapify:



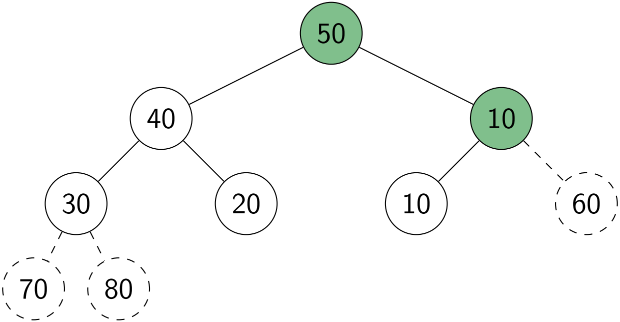
heap-sort example

Continue with the example from heapify:



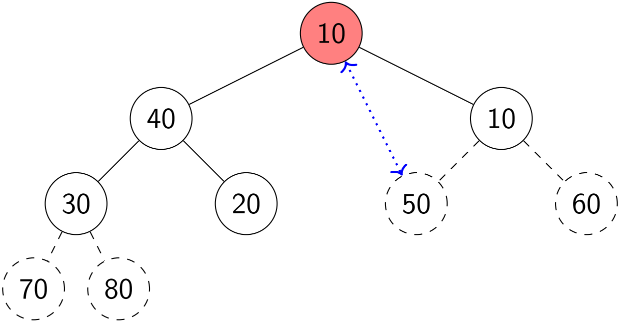
heap-sort example

Continue with the example from heapify:



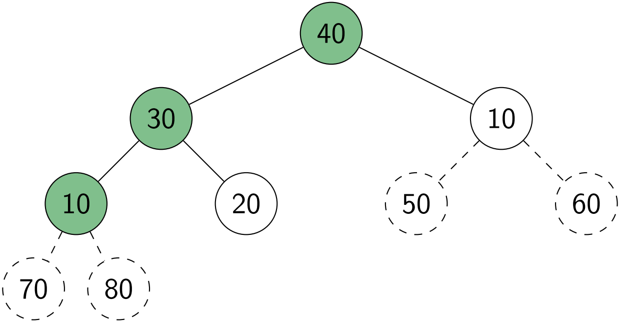
heap-sort example

Continue with the example from heapify:



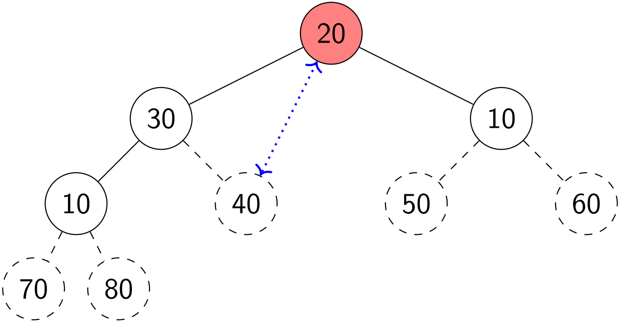
heap-sort example

Continue with the example from heapify:



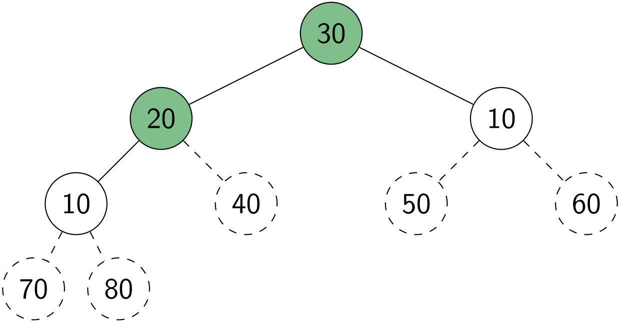
heap-sort example

Continue with the example from heapify:



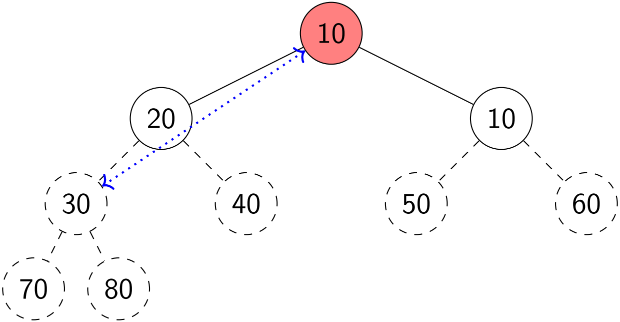
heap-sort example

Continue with the example from heapify:



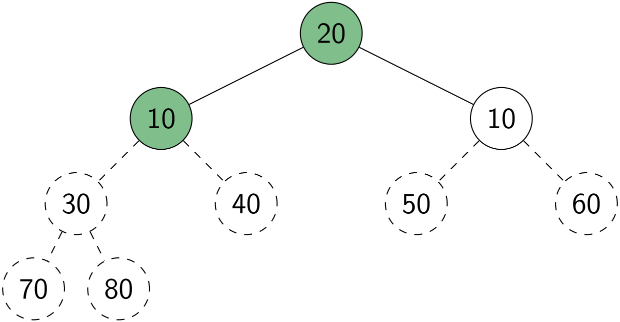
heap-sort example

Continue with the example from heapify:



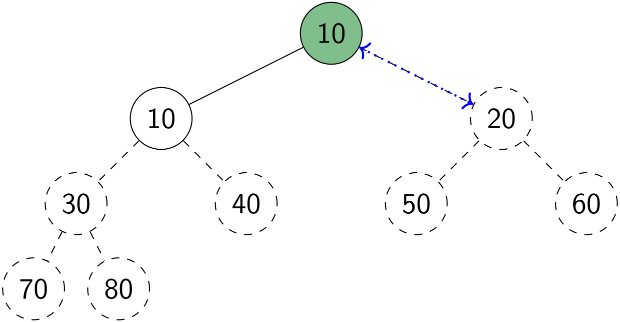
heap-sort example

Continue with the example from heapify:



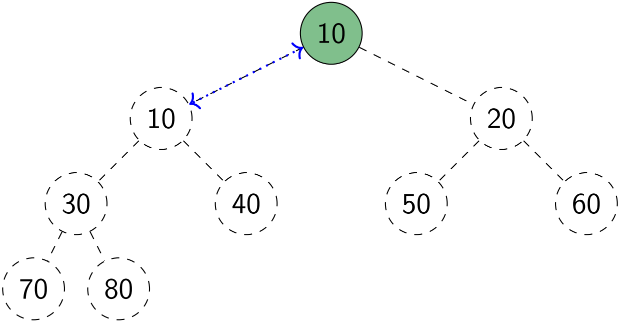
heap-sort example

Continue with the example from heapify:



heap-sort 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 $O(\log n)$
  - *delete-max* 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 ($\rightsquigarrow$ *heap-sort*)
- 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 *delete-min* with the same run-times.

Outline

2. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps as PQ realization
   - *PQ-sort* and *heap-sort*
   - Towards the Selection Problem

Finding the smallest items

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

Finding the smallest items

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

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

Finding the smallest items

Problem: Find the *kth smallest item* in an array A of n numbers.
(Formally: kth smallest = the item that would be at $A[k]$ if sorted.)

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

Finding the smallest items

Problem: Find the *kth smallest item* in an array A of n numbers.
(Formally: kth smallest = the item that would be at $A[k]$ if sorted.)

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

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

Finding the smallest items

Problem: Find the *kth smallest item* in an array A of n numbers.
(Formally: kth smallest = the item that would be at $A[k]$ if sorted.)

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

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

Solution 3: Create a min-heap with *heapify*(A). Call *delete-min*(A) $k+1$ times.
Complexity: $\Theta(n + k \log n)$.

Finding the smallest items

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

(Formally: kth smallest = the item that would be at $A[k]$ if sorted.)

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

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

Solution 3: Create a min-heap with *heapify*(A). Call *delete-min*(A) $k+1$ times.
Complexity: $\Theta(n + k \log n)$.

We can achieve $\Theta(n \log n)$ worst-case time easily, but can we do better?