Week 9
Sorting

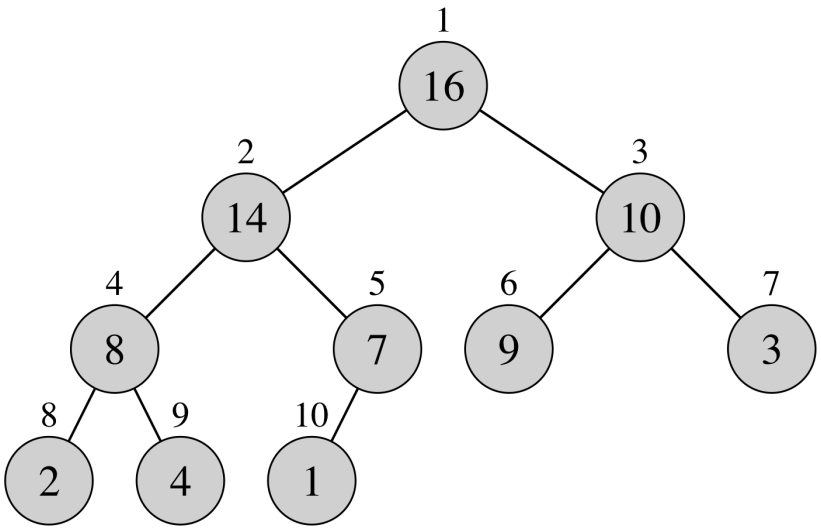
General remarks

- We return to sorting, considering HEAP-SORT and QUICK-SORT.

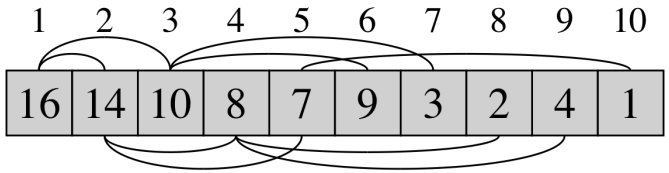
Reading from CLRS for week 7

1. Chapter 6, Sections 6.1 - 6.5.
2. Chapter 7, Sections 7.1, 7.2.

Discover the properties of binary heaps
Running example



(a)



(b)

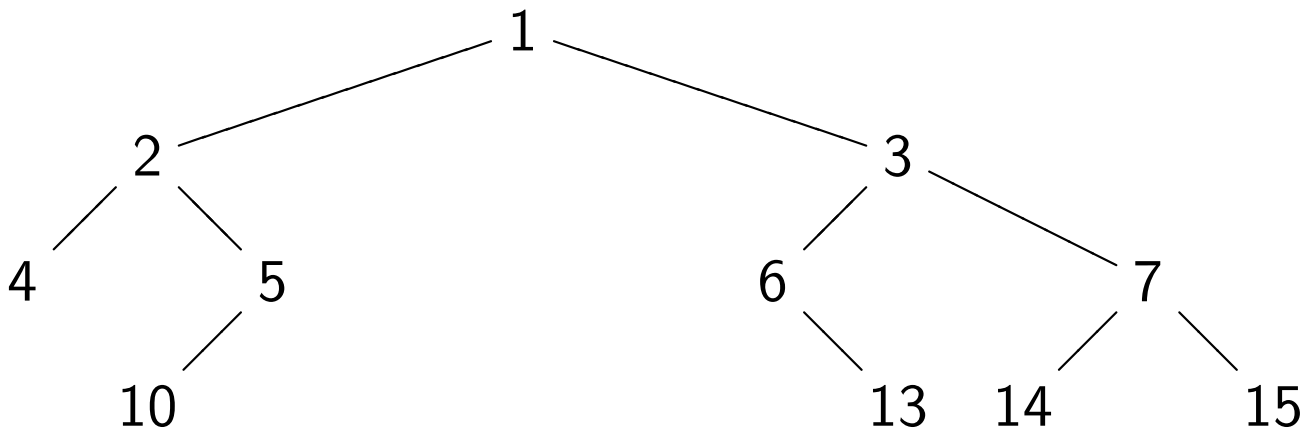
First property: level-completeness

In your first year you have seen **binary trees**:

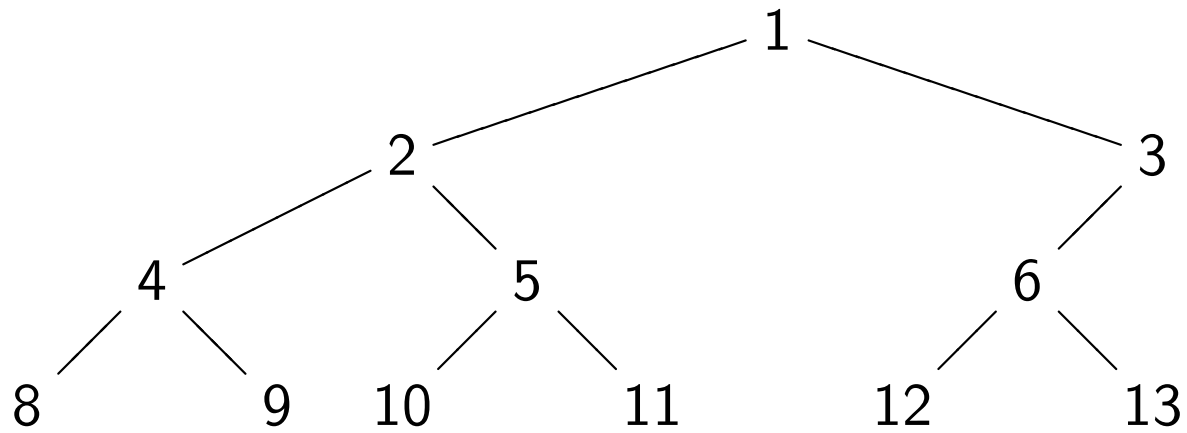
1. In data structures they should be as "balanced" as possible, i.e., their height should be as small as possible for the given number of nodes.
2. Recall that the height $\mathrm{ht}(T)$ of a rooted tree is the maximum distance between the root and any leaf.
3. Perfect are the perfect binary trees (for height h they have $2^{h+1}-1$ nodes).
4. Now close to perfect come the **level-complete binary trees**:
   1. We can partition the nodes of a (binary) tree T into levels, according to their distance from the root.
   2. We have levels $0, 1, \ldots, \mathrm{ht}(T)$.
   3. In general, level k has from 1 to 2^k nodes.
   4. If all levels k except possibly for the last one are full (have precisely 2^k nodes in them), then we call the tree **level-complete**.

Examples

The binary tree



is level-complete (level-sizes are $1, 2, 4, 4$), while



is not (level-sizes are $1, 2, 3, 6$).

The height of level-complete binary trees

For a level-complete binary tree T we have

$$\mathrm{ht}(T) = \lfloor \lg(\#\mathrm{nds}(T)) \rfloor ,$$

where $\#\mathrm{nds}(T)$ is the number of nodes of T.

That is, the height of T is the binary logarithm of the number of nodes of T, after removal of the fractional part.

- "Balanced" ("good") trees T should have

$$\mathrm{ht}(T) \approx \lg(\#\mathrm{nds}(T)).$$

- Level-complete binary trees definitely fulfil that.

Operations in binary trees are often
linear in the height of the tree.
Thus the height should be as small as possible.

Second property: completeness

To have simple and efficient access to the nodes of the tree, the nodes of the last layer better are not placed in random order:

- Best is if they fill the positions from the left without gaps.
- A level-complete binary tree with such gap-less last layer is called a **complete tree**.
- So the level-complete binary tree on the examples-slide is not complete.
- While the running-example is complete.

Third property: the heap-property

The running-example is NOT a binary *search* tree:

1. It would be too expensive to have this property (all nodes to the left $\leq$, all nodes to the right $\geq$) *together* with the completeness property.
2. However we have another property related to order (not just related to the structure of the tree): The value of every node is not less than the value of any of its successors (the nodes below it).
3. In other words, all nodes below (to the left *and* to the right) are $\leq$.
4. This property is called the **heap property**.
5. More precisely it is the **max-heap property**.

Definition 1

A **binary heap** is a binary tree which is complete and has the heap property. More precisely we have **binary max-heaps** and **binary min-heaps**.

Fourth property: Efficient index computation

Consider the numbering (not the values) of the nodes of the running-example:

1. This numbering follows the layers, beginning with the first layer and going from left to right.
2. Due to the completeness property (no gaps!) these numbers yield easy relations between a parent and its children.
3. If the node has number p, then the left child has number $2p$, and the right child has number $2p + 1$.
4. And the parent has number $\lfloor p/2 \rfloor$.

Efficient array implementation

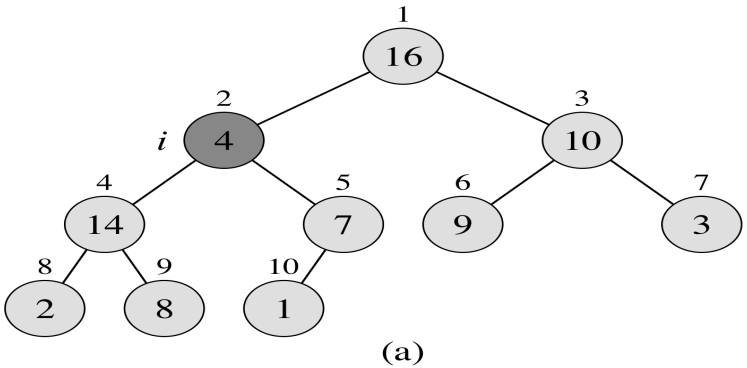
For binary search trees we needed full-fledged trees (as you implemented in your first year):

1. That is, we needed nodes with three pointers: to the parent and to the two children.
2. However now, for complete binary trees we can use a more efficient array implementation, using the numbering for the array-indices.

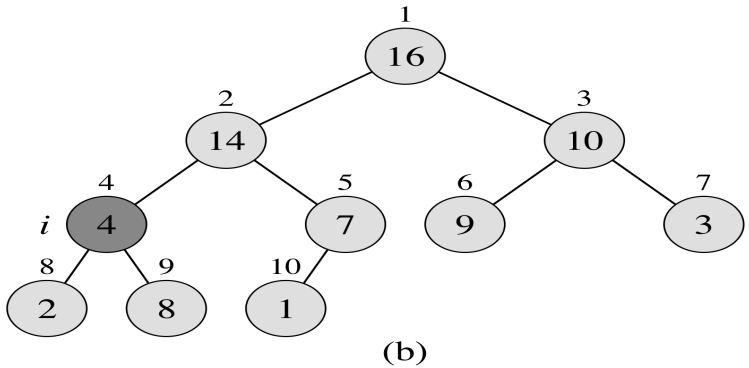
So a binary heap with m nodes is represented by an array with m elements:

- C-based languages use 0-based indices (while the book uses 1-based indices).
- For such an index $0 \leq i < m$ the index of the left child is $2i + 1$, and the index of the right child is $2i + 2$.
- While the index of the parent is $\lfloor (i - 1)/2 \rfloor$.

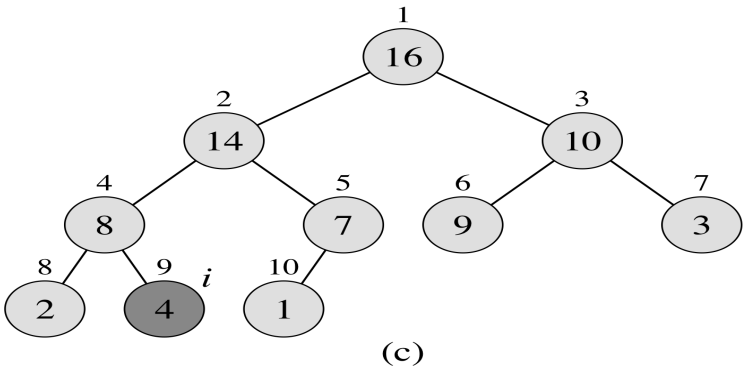
Float down a single disturbance



(a)



(b)



(c)

The idea of heapification

- The input is an array A and index i into A.
- It is assumed that the binary trees rooted at the left and right child of i are binary (max-)heaps, but we do not assume anything on $A[i]$.
- After the "heapification", the values of the binary tree rooted at i have been rearranged, so that it is a binary (max-)heap now.

For that, the algorithm proceeds as follows:

1. First the largest of $A[i], A[l], A[r]$ is determined, where $l = 2i$ and $r = 2i + 1$ (the two children).
2. If $A[i]$ is largest, then we are done.
3. Otherwise $A[i]$ is swapped with the largest element, and we call the procedure recursively on the changed subtree.

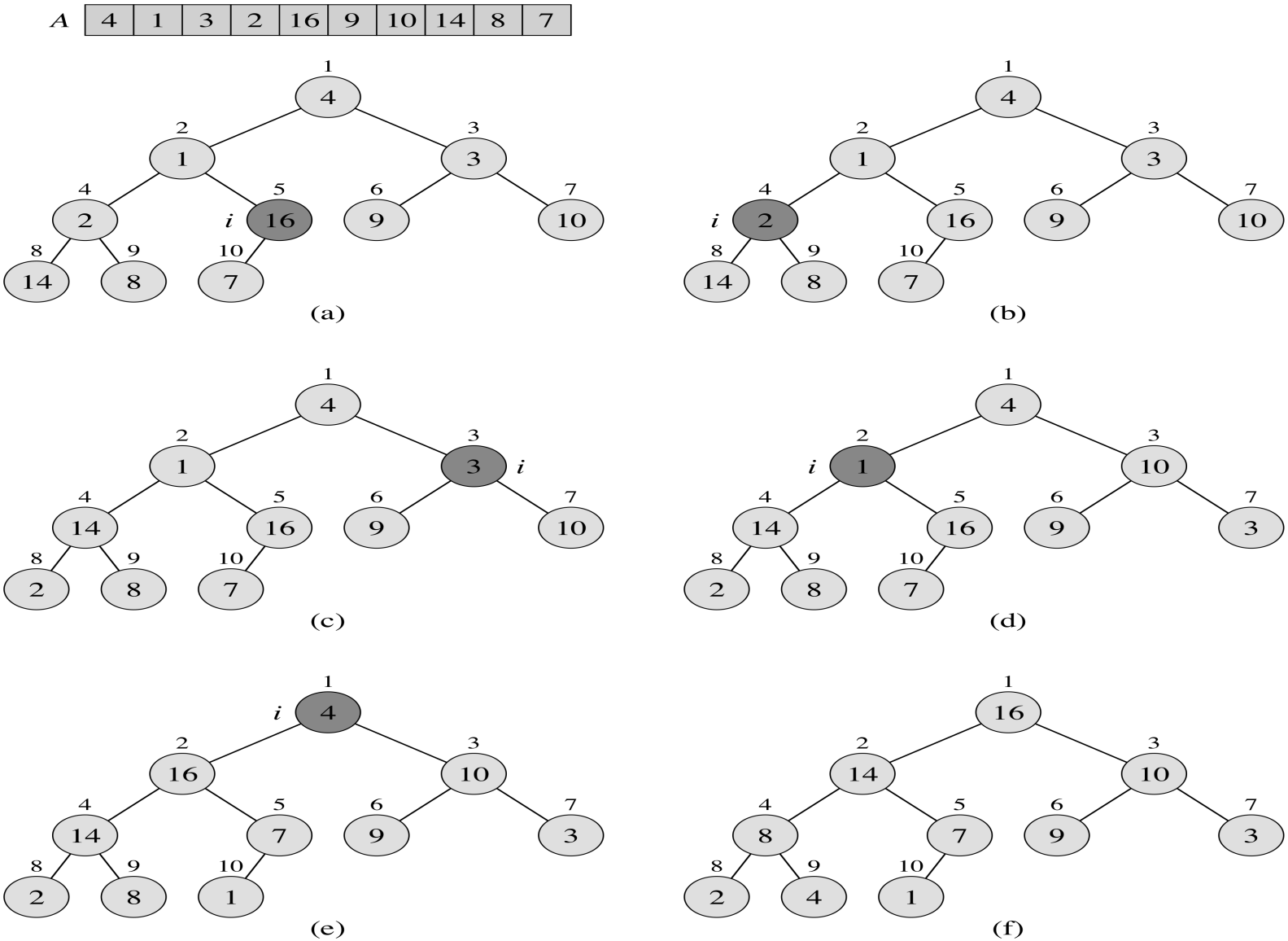
Analysing heapification

Obviously, we go down from the node to a leaf (in the worst case), and thus the running-time of heapification is

linear in the height h of the subtree.

This is $O(\lg n)$, where n is the number of nodes in the subtree (due to $h = \lfloor \lg n \rfloor$).

Heapify bottom-up



The idea of building a binary heap

One starts with an arbitrary array A of length n, which shall be re-arranged into a binary heap. Our example is

$$A = (4, 1, 3, 2, 16, 9, 10, 14, 8, 7).$$

We repair (heapify) the binary trees bottom-up:

1. The leaves (the final part, from $\lfloor n/2 \rfloor + 1$ to n) are already binary heaps on their own.
2. For the other nodes, from right to left, we just call the heapify-procedure.

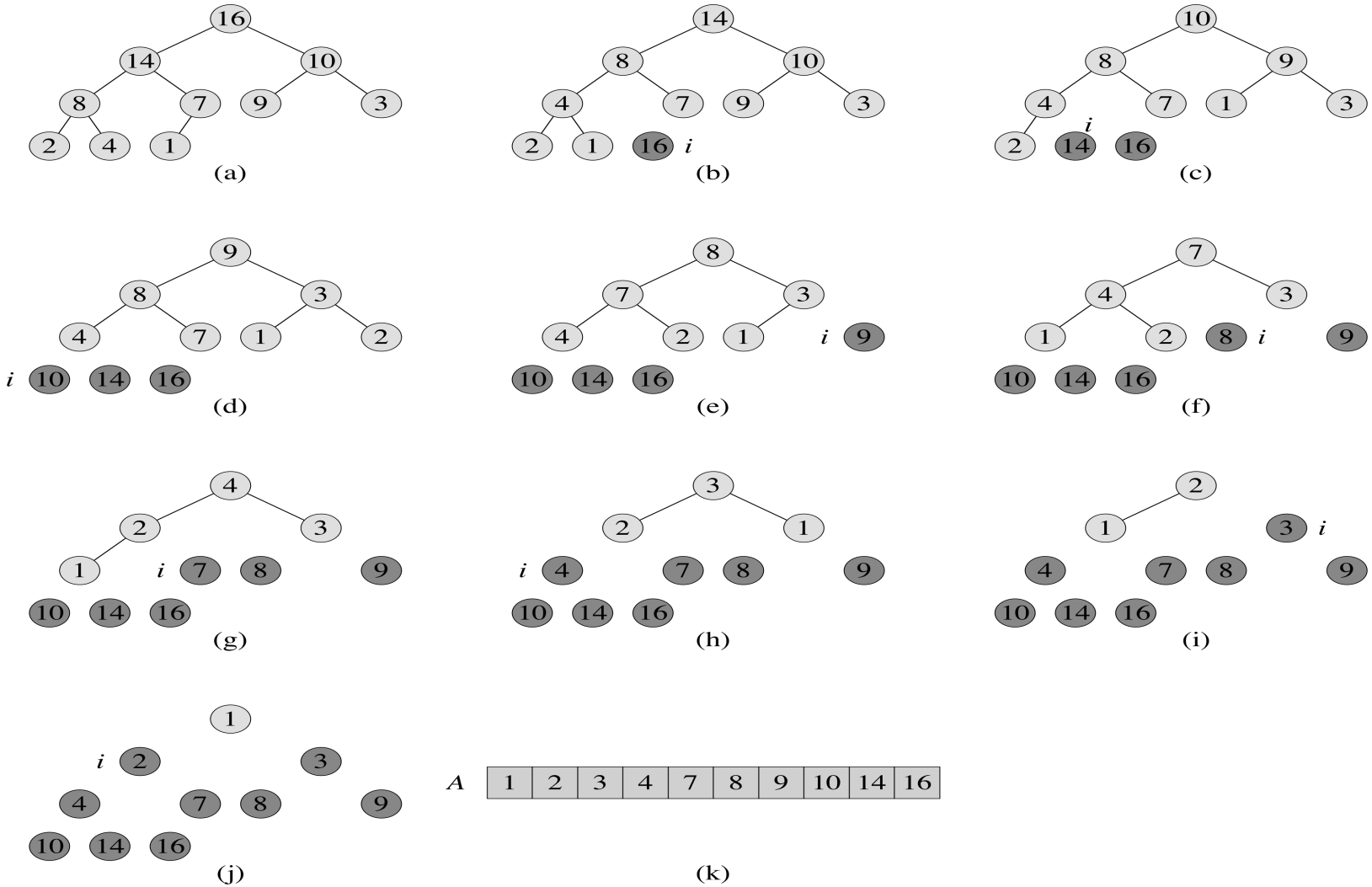
Analysing building a heap

Roughly we have $O(n \cdot \lg n)$ many operations:

1. Here however it pays off to take into account that most of the subtrees are small.
2. Then we get run-time $O(n)$.

So building a heap is linear in the number of elements.

Heapify and remove from last to first



The idea of HEAP-SORT

Now the task is to sort an array A of length n:

1. First make a binary max-heap out of A (in linear time).
2. Repeat the following until $n = 1$:
   1. The maximum element is now $A[1]$ — swap that with the last element $A[n]$, and remove that last element, i.e., set $n := n - 1$.
   2. Now perform heapification for the root, i.e., $i = 1$. We have a binary (max-)heap again (of length one less).

The run-time is $O(n \cdot \lg n)$.

Remark on SELECTION-SORT

We see that HEAP-SORT is an itelligent form of SELECTION-SORT (where we move from right to left, not from left to right):

1. We extract the largest element, the second largest etc., as does SELECTION-SORT.
2. But we support the search by the binary-heap data structure.
3. Thus each search takes only time $O(\lg n)$, where n is the number of remaining elements, instead of $\Theta(n)$ as for SELECTION-SORT.

All basic operations are (nearly) there

Recall that a (basic) (max-)priority queue has the operations:

- MAXIMUM
- DELETE-MAX
- INSERTION.

We use an array A containing a binary (max-)heap (the task is just to maintain the max-heap-property!):

1. The maximum is $A[1]$.
2. For deleting the maximum element, we put the last element $A[n]$ into $A[1]$, decrease the length by one (i.e., $n := n - 1$), and heapify the root (i.e., $i = 1$).
3. And we add a new element by adding it to the end of the current array, and heapifying all its predecessors up on the way to the root.

Examples

Using our running-example, a few slides ago for HEAP-SORT:

1. Considering it from (a) to (j), we can see what happens when we perform a sequence of $\mathrm{DELETE\text{-}MAX}$ operations, until the heap only contains one element (we ignore here the shaded elements — they are visible only for the HEAP-SORT).
2. And considering the sequence in reverse order, we can see what happens when we call $\mathrm{INSERTION}$ on the respective first shaded elements (these are special insertions, always inserting a new max-element).

Analysis

- MAXIMUM is a constant-time operation.
- DELETE-MAX is one application of heapification, and so need time $O(\lg n)$ (where n is the current number of elements in the heap).
- INSERTION seems to invoke up to the current height many applications of heapification, and thus would look like $O((\lg n)^2)$, but it's easy to see that it is $O(\lg n)$ as well (see the tutorial).

The idea of QUICK-SORT

Remember MERGE-SORT:

- A divide-and-conquer algorithm for sorting an array in time $O(n \cdot \lg n)$.
- The array is split in half, the two parts are sorted recursively (via MERGE-SORT), and then the two sorted half-arrays are merged to the sorted (full-)array.

Now we split along an element x of the array:

- We partition into elements $\leq x$ (first array) and $> x$ (second array).
- Then we sort the two sub-arrays recursively.
- Done!

Remark on ranges

In the book arrays are 1-based:

1. So the indices for an array A of length n are $1, \ldots, n$.
2. Accordingly, a sub-array is given by indices $p \leq r$, meaning the range $p, \ldots, r$.

For Java-code (C, C++, C#, etc.) we use 0-based arrays:

1. So the indices are $0, \ldots, n-1$.
2. Accordingly, a sub-array is given by indices $p < r$, meaning the range $p, \ldots, r-1$.

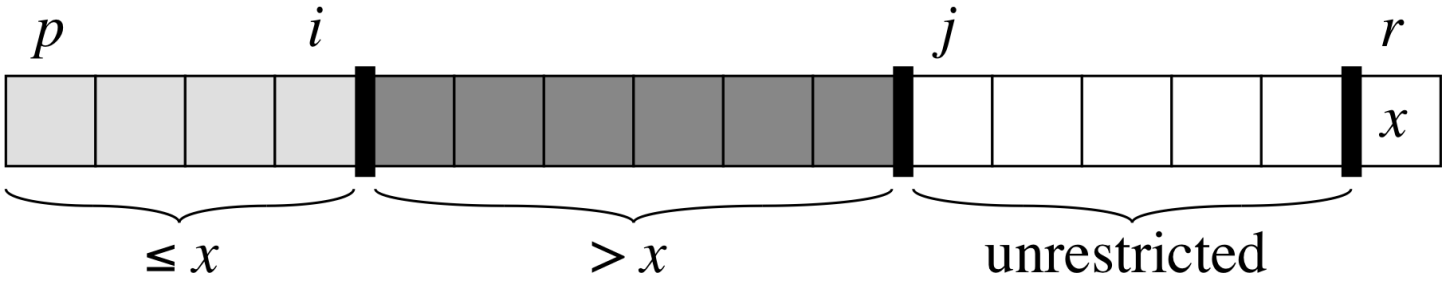
Range-bounds for a sub-array are here now always left-closed and right-open!

So the whole array is given by the range-parameters $0, n$.

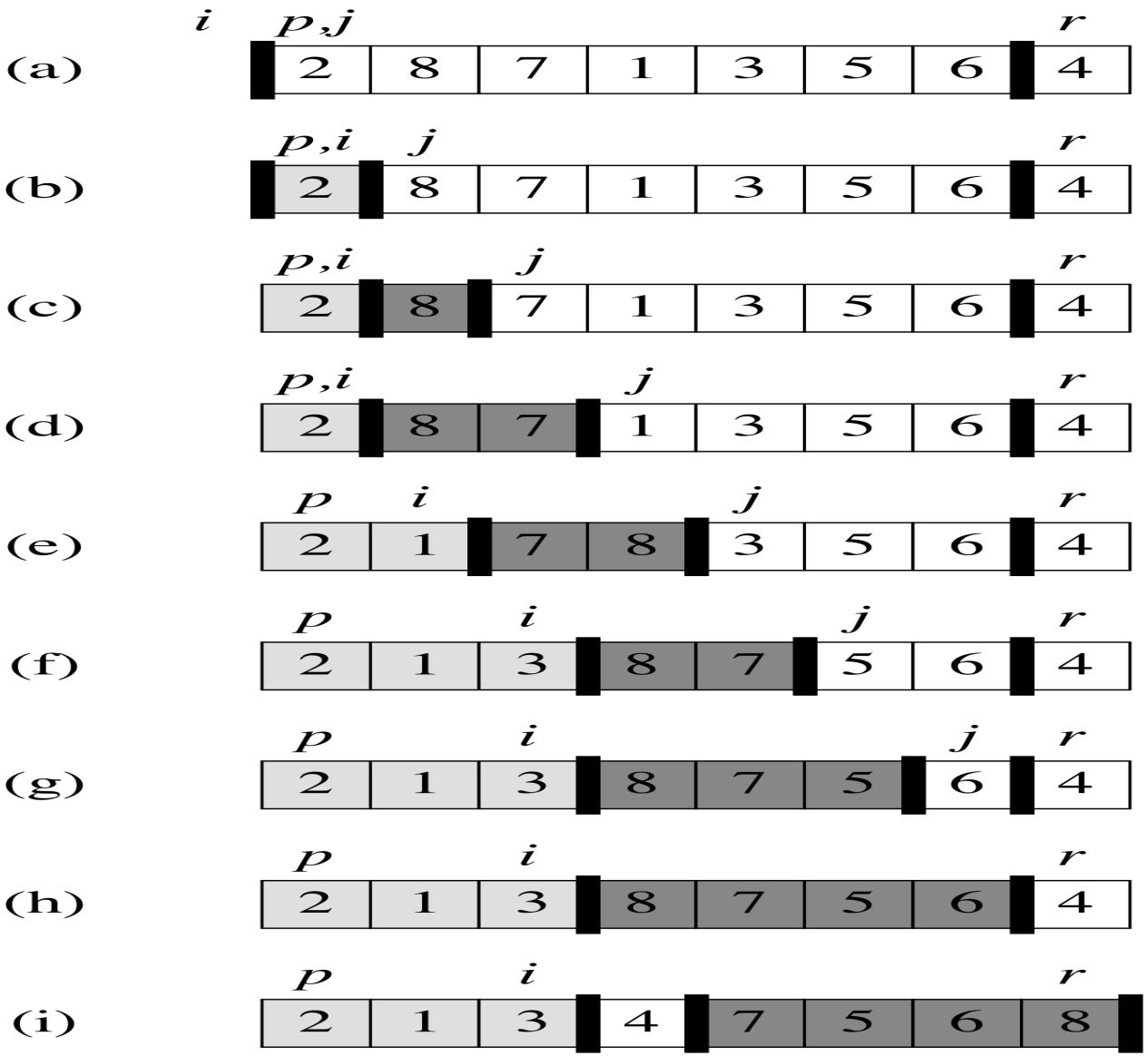
The main procedure

```
public static void sort(final int[] A, final
    int p, final int r) {
  assert(A != null);
  assert(p >= 0);
  assert(p <= r);
  assert(r <= A.length);

  if (r-p <= 1) return;

  place_partition_element_last(A,p,r);
  final int q = partition(A,p,r);
  assert(p <= q);
  assert(q < r);

  sort(A,p,q);
  sort(A,q+1,r);
}
```

The idea of partitioning in-place



An example


(a)
i
p,j
r
2 8 7 1 3 5 6 4
(b)
p,i
j
r
2 8 7 1 3 5 6 4
(c)
p,i
j
r
2 8 7 1 3 5 6 4
(d)
p,i
j
r
2 8 7 1 3 5 6 4
(e)
p
i
j
r
2 1 7 8 3 5 6 4
(f)
p
i
j
r
2 1 3 8 7 5 6 4
(g)
p
i
j
r
2 1 3 8 7 5 6 4
(h)
p
i
r
2 1 3 8 7 5 6 4
(i)
p
i
r
2 1 3 4 7 5 6 8

The code

Instead of i we use $q = i + 1$:

```
private static int partition(final int[] A,
    final int p, final int r) {
  assert(p+1 < r);
  final int x = A[r-1];

  int q = p;
  for (int j = p; j < r-1; ++j) {
    final int v = A[j];
    if (v <= x) {A[j] = A[q]; A[q++] = v;}
  }

  A[r-1] = A[q]; A[q] = x;
  return q;
}
```

Selecting the pivot

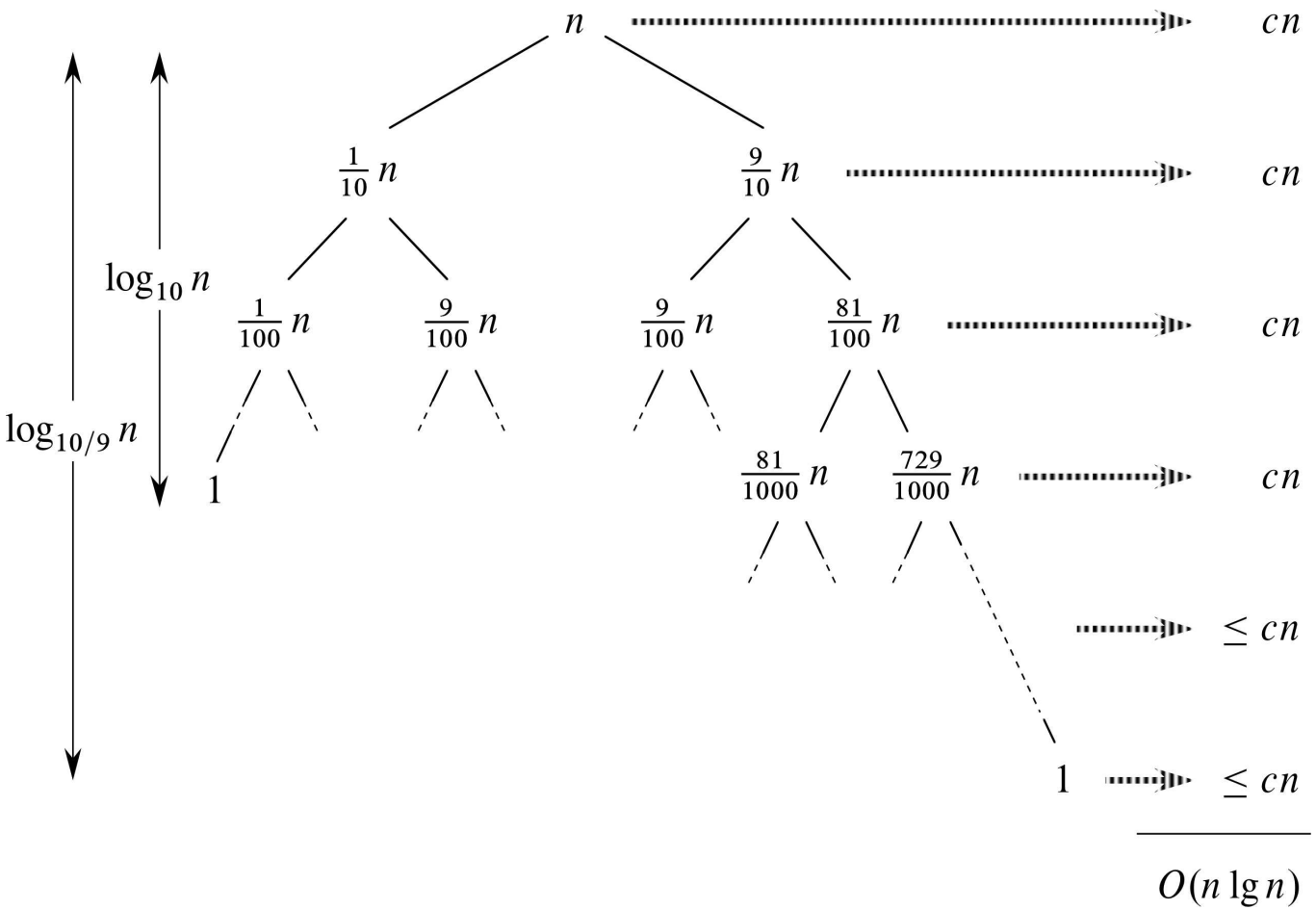
The partitioning-procedure expects the partitioning-element to be the last array-element. So for selecting the pivot, we can just choose the last element:

```
private static void
      place_partition_element_last(final int[]
       A, final int p, final int r) {}
```

However this makes it vulnerable to "malicious" choices, so we better **randomise**:

```
private static void
      place_partition_element_last(final int[]
       A, final int p, final int r) {
  final int i = p+(int) Math.random()*(r-p);
  {final int t=A[i]; A[i]=A[r-1]; A[r-1]=t;}
}
```

A not unreasonable tree



Average-case

If we actually achieve that both sub-arrays are at least a constant fraction α of the whole array (in the previous picture, that's $\alpha = 0.1$), then we get

$$T(n) = T(\alpha \cdot n) + T((1 - \alpha) \cdot n) + \Theta(n).$$

That's basically the second case of the Master Theorem (the picture says it's similar to $\alpha = \frac{1}{2}$), and so we *would* get

$$T(n) = \Theta(n \cdot \log n).$$

And we actually get that:

- for the non-randomised version (choosing always the last element as pivot), when averaging over all possible input sequences (without repetitions);
- for the randomised version (choosing a random pivot), when averaging over all (internal!) random choices; here we do not have to assume something on the inputs, except that all values are different.

Worst-case

However, as the tutorial shows:

The worst-case run-time of QUICK-SORT is $\Theta(n^2)$ (for both versions)!

- This can be repaired, making also the worst-case run-time $\Theta(n \cdot \log n)$.
- For example by using median-computation in linear time for the choice of the pivot.
- In practice these days often (for example in gcc) Introsort is used, which starts with QUICK-SORT, and switches to HEAP-SORT if recursion depth becomes too big, while when the array is small enough, INSERTION-SORT is used.
- When pointers are to be handled, then Timsort is preferable, which is derived from MERGE-SORT and INSERTION-SORT.

HEAP-SORT on sorted sequence

What does HEAP-SORT on an already sorted sequence? And what's the complexity? Consider the input sequence

$$1, 2, \ldots, 10.$$

HEAP-SORT on sorted sequence (cont.)

Now the sorting works as follows:

1. $10, 9, 7, 8, 5, 6, 3, 1, 4, 2$
2. $9, 8, 7, 4, 5, 6, 3, 1, 2; 10$
3. $8, 5, 7, 4, 2, 6, 3, 1; 9, 10$
4. $7, 5, 6, 4, 2, 1, 3; 8, 9, 10$
5. $6, 5, 3, 4, 2, 1; 7, 8, 9, 10$
6. $5, 4, 3, 1, 2; 6, 7, 8, 9, 10$
7. $4, 2, 3, 1; 5, 6, 7, 8, 9, 10$
8. $3, 2, 1; 4, 5, 6, 7, 8, 9, 10$
9. $2, 1; 3, 4, 5, 6, 7, 8, 9, 10$
10. $1; 2, 3, 4, 5, 6, 7, 8, 9, 10$

So HEAP-SORT doesn't take any advantage from some already existing sorting.

Simplifying insertion

When discussing insertion into a (max-)priority-queue, implemented via a binary (max-)heap, we just used a general addition of one element into an existing heap:

- The insertion-procedure used heapification up on the path to the root.
- Now actually we have always special cases of heapification — namely which?

Simplifying insertion

When discussing insertion into a (max-)priority-queue, implemented via a binary (max-)heap, we just used a general addition of one element into an existing heap:

- The insertion-procedure used heapification up on the path to the root.
- Now actually we have always special cases of heapification — namely which?

Answer: The new element only *goes up*, swapping with its parent, which then has already its place, until the new element found its place.

Change to the partitioning procedure

What happens if we change the line

```
if (v <= x) {A[j] = A[q]; A[q++] = v;}
```

of function partition to

```
if (v < x) {A[j] = A[q]; A[q++] = v;}
```

- Can we do it?
- Would it have advantages?

Change to the partitioning procedure

What happens if we change the line

```
if (v <= x) {A[j] = A[q]; A[q++] = v;}
```

of function partition to

```
if (v < x) {A[j] = A[q]; A[q++] = v;}
```

- Can we do it?
- Would it have advantages?

Answer: Yes, we can do it: it would save work, due to values equal x not being moved to the left part.

QUICK-SORT on constant sequences

What is QUICK-SORT doing on a constant sequence, in its three incarnations:

- pivot is last element
- pivot is random element
- pivot is median element?

QUICK-SORT on constant sequences

What is QUICK-SORT doing on a constant sequence, in its three incarnations:

- pivot is last element
- pivot is random element
- pivot is median element?

One of the two sub-arrays will have size 1, and QUICK-SORT degenerates to an $O(n^2)$ algorithm (which does nothing).

What can we do about it?

QUICK-SORT on constant sequences

What is QUICK-SORT doing on a constant sequence, in its three incarnations:

- pivot is last element
- pivot is random element
- pivot is median element?

One of the two sub-arrays will have size 1, and QUICK-SORT degenerates to an $O(n^2)$ algorithm (which does nothing).

What can we do about it?

We can refine the partition-procedure by

- not just splitting into two parts,
- but into three parts: all elements $< x$, all elements $= x$, and all elements $> x$.

Then we use only the left and the right sub-array (leaving out the whole middle) for recursion. We get $O(n \log n)$ for constant sequences.

Worst-case for QUICK-SORT

Consider sequences without repetitions, and assume the pivot is always the last element:

- What is a worst-case input?
- And what is QUICK-SORT doing on it?

Worst-case for QUICK-SORT

Consider sequences without repetitions, and assume the pivot is always the last element:

- What is a worst-case input?
- And what is QUICK-SORT doing on it?

Every already sorted sequence is a worst-case example!
QUICK-SORT behaves as with constant sequences.

Note that this is avoided with randomised pivot-choice (and, of course, with median pivot-choice).

Worst-case $O(n \log n)$ for QUICK-SORT

How can we achieve $O(n \log n)$ in the **worst-case** for QUICK-SORT?

Worst-case $O(n \log n)$ for QUICK-SORT

How can we achieve $O(n \log n)$ in the **worst-case** for QUICK-SORT?

- Just choosing, within our current framework, the median-*element* is not enough, but we need the change the framework, allowing to compute the median-*index* (or we do the triple-partition as discussed).
- Best is to remove the function `place_partition_element_last`, and leave the partitioning fully to function `partition`.

Then the main procedure becomes (without the asserts):

```
public static void sort(final int[] A, final
    int p, final int r) {
  if (r-p <= 1) return;
  final int q = partition(A,p,r);
  sort(A,p,q); sort(A,q+1,r);
}
```

Divide-and-Conquer

- Matrix Multiplication is a great example, which you should have seen.
- Otherwise, if you do not have special interests, I think you can move on to other paradigms.
- Sure, algorithmic analysis (and the underlying mathematics) is an infinite game, but you might learn it as you go.
- The Wikipedia page on the Master Theorem is quite nice.

Graphs

- Graph theory has many many aspects.
- Much theory, much experimentation.
- "Network" are graphs, but with a specialised focus on growth of the graph.

Data structures

- The book has plenty on the classical data structures.
- Hashing is a very active topic, with special links to cryptography.

Complexity theory

- This is about about the resource needs of algorithmic problem IN GENERAL.
- Most central (and most famous) the "P versus NP" question: is there a polytime algorithm for finding an assignment for a propositional formula?

SAT

- SAT is the field of deciding whether propositional formulas are satisfiable (have an assignment making them true), or not (are unsatisfiable).
- See The Science of Brute Force.