FTP_Alg_Week2

jungkyu.canci@hslu.ch

HS2024

Sorting algorithms

Sorting is considered the most fundamental problem in the study of algorithms:

- Sorting is often a tool for other applications. Often, in order to apply some algorithm, one has to order some objects.
- Many different techniques have been developed for solving different sorting problems.
- We have seen insertion–sort. The running time is $\Theta(n^2)$ in the worst case. It is a fast in-place algorithm for sorting problem with small n.
- We will consider merge sort, heapsort (and quicksort later).
- We will see that merge sort has a better running time than insertion-sort.

How to improve the complexity?

We present a recursive structure based on **divide–and–conquer**.

- **Divide** the problem in sub–problems of *smaller* size.
- **Conquer** the sub-problems, i.e. solve them (usually with a recursive method).
- **Combine** the solutions of the sub–problems into a solution for the original whole problem.

An example of the above procedure is the **merge sort** algorithm, which operates as follows:

- **Divide**: Divide the sequence of n numbers into two sub–sequences of (about) $n/2$ elements each.
- **Conquer**: Sort the each sub–sequence recursively using the merge sort.
- **Combine**: Merge the two above outputs to produce the sorted sequence.

Recursive functions

- As you can see in previous slide, we defined merge sort procedure by using merge sort. This can be viewed as a **recursive procedure** or *recursive function*.
- A typical example of recursive function is related to the Fibonacci numbers: they are a sequence of numbers $\{f_n\}_{n\geq 0}$ defined by

$$\begin{cases} f_0 = 0, f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad \text{for all } n \geq 2 \end{cases}$$

The first elements are the following ones:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \ldots$$

Recursive functions: Examples

https://www.youtube.com/watch?v=BalWjeY2O9g
https://www.youtube.com/watch?v=b005iHf8Z3g

Merge procedure

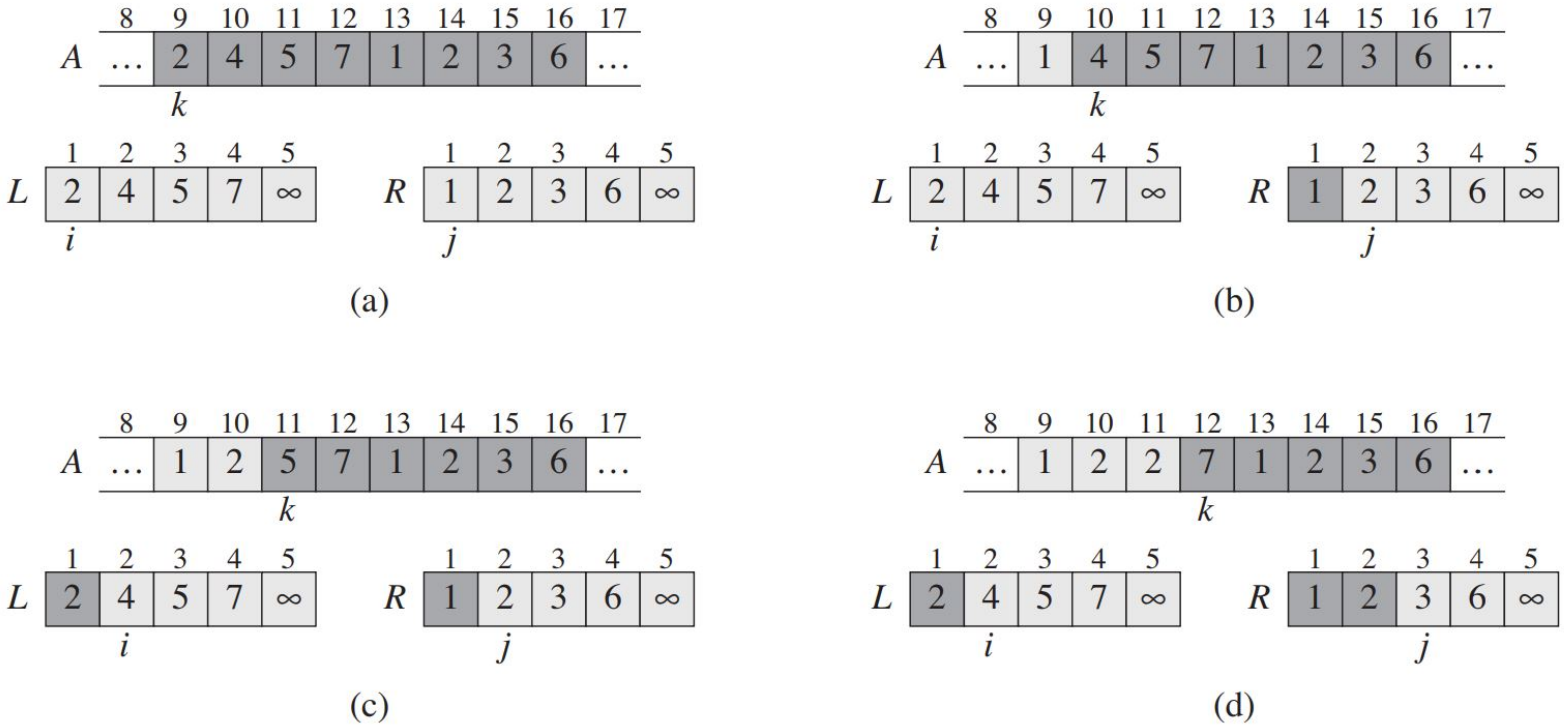
- In Merge–sort we call the auxiliary procedure MERGE(A, p, q, r), where A is an array, $p \leq q < r$ are indexes into the array.
- In the procedure the sub–arrays $A[p, \ldots, q]$ and $A[q+1, \ldots, r]$ are already sorted. The call MERGE(A, p, q, r) merges the above two sub–arrays into a single sorted sub–array.
- In the next slide we show the pseudo–code of merge procedure. We use two sentinels, that are cards denoted by ∞, which determine the bottom of the pile's. This trick simplifies the code.

Merge: Pseudo–code

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

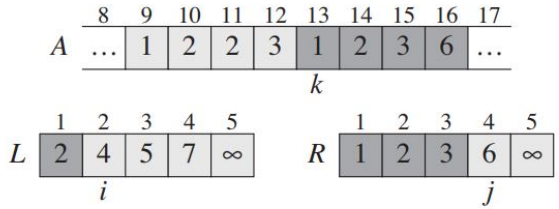
- L1–L2: the length of the sub–arrays are determined.
- L3: new arrays are created of length $n_1 + 1$ and $n_2 + 1$ resp.
- L4–L9: we copy the old array in the next ones L and R where we add the sentinels.
- L10–11: loop initialization
- L12–17, merges the two sub–arrays as illustrated in the next figure.

Merge

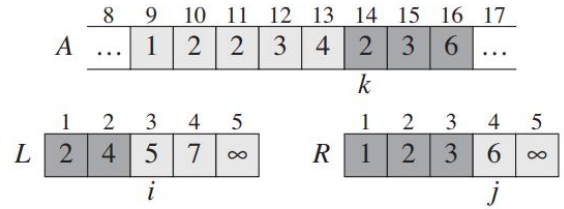


At the start of each loop we have $A[p, \ldots, k-1]$ containing in a sorted order all $k-p$ smallest elements of $L[1, \ldots, n_1+1]$ and $R[1, \ldots, n_1+1]$ and $L[i]$ and $R[j]$ are the smallest element of the two sub–arrays, which have not been copied back into A.

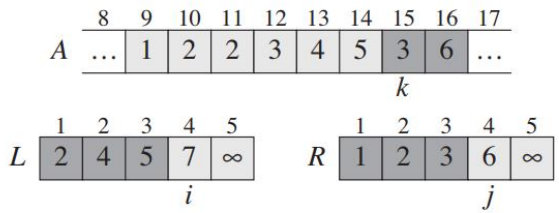
Merge


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

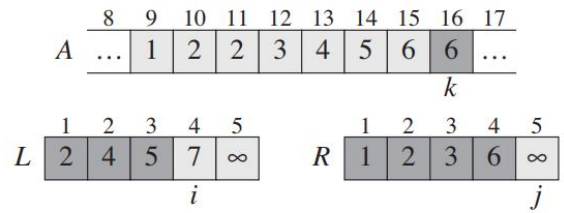
(e)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

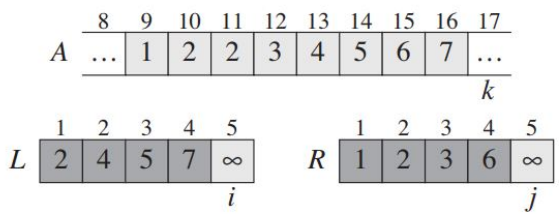
(f)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(g)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(h)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(i)

Merge: running time

Let $n = r - p + 1$.

For analyzing the running time of merge, observe that:

- Lines 1–3 and 8–1 take constant time.
- The two **for** loops in 4-7 both require $O(n)$ operations, that should be added up.
- Finally there is n iteration of the loop at lines 12–17, each of which takes constant time.

Thus the running time is $\Theta(n)$.

Merge-sort

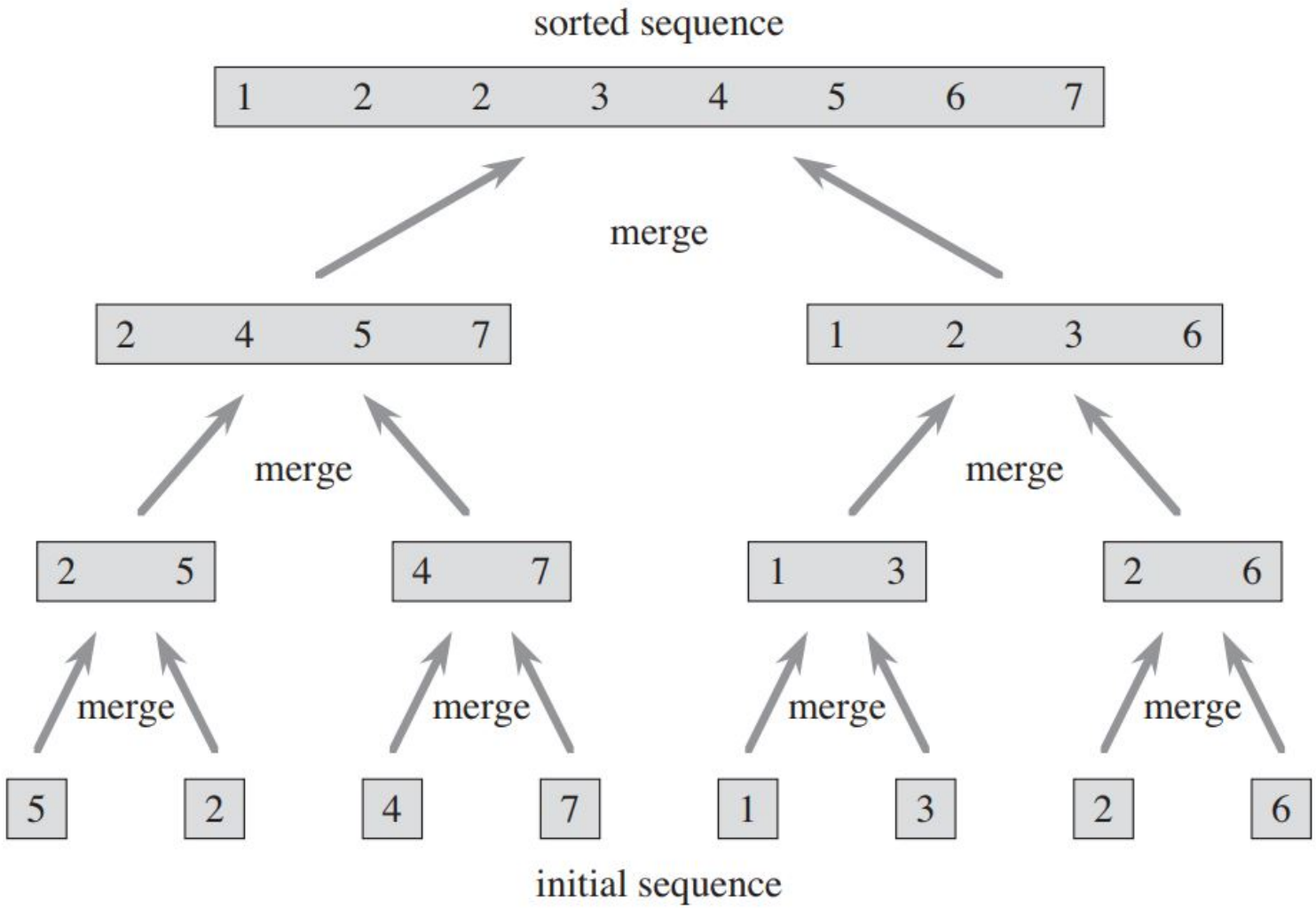
- As we have seen in a previous slide, MERGE is a sub–routine in the merge sort algorithm.
- Let $p < r$ be non negative integers. MERGE–SORT(A,p,r) sorts the elements of the array $A[p, \ldots, r]$.
- If this array has more that one element, the first step of the procedure is to divide into two sub–arrays $A[p, \ldots, q]$ and $A[q+1, \ldots, r]$, where $q = \lfloor n/2 \rfloor$.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

To sort the entire sequence $A =< A[1], A[2], \ldots, A[n] >$, where $n = A.length$, we make the call MERGE–SORT$(A, 1, A.length)$.

Merge–Sort: Example

Next figure shows an example of iterations of MERGE–SORT.



Merge-sort: Analyzing divide-and-conquer algorithms

1. MERGE–SORT is a divide–and–conquer algorithm.
2. Let us denote by $T(n)$ be the running time of an algorithm for a size n elements.
3. If the problem size is small enough, say $n \leq c$ for a constant c, the solution take $\Theta(1)$ time. (For merge sort $c = 1$)
4. Suppose that the algorithm divide the problem in a sub–problems, each with size $1/b$ of the original one (for merge sort $a = b = 2$).
5. Let $D(n)$ be the time for dividing in sub–problems and $C(n)$ be the time for combining together the solutions of the sub–problems.
6. We get the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Master Theorem

Let $a \geq 1$, $b > 1$ be constants, $f(n)$ be a function and $T(n)$ defined by the following recurrence

$$T(n) = aT(n/b) + f(n).$$

where n/b is interpreted as either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then

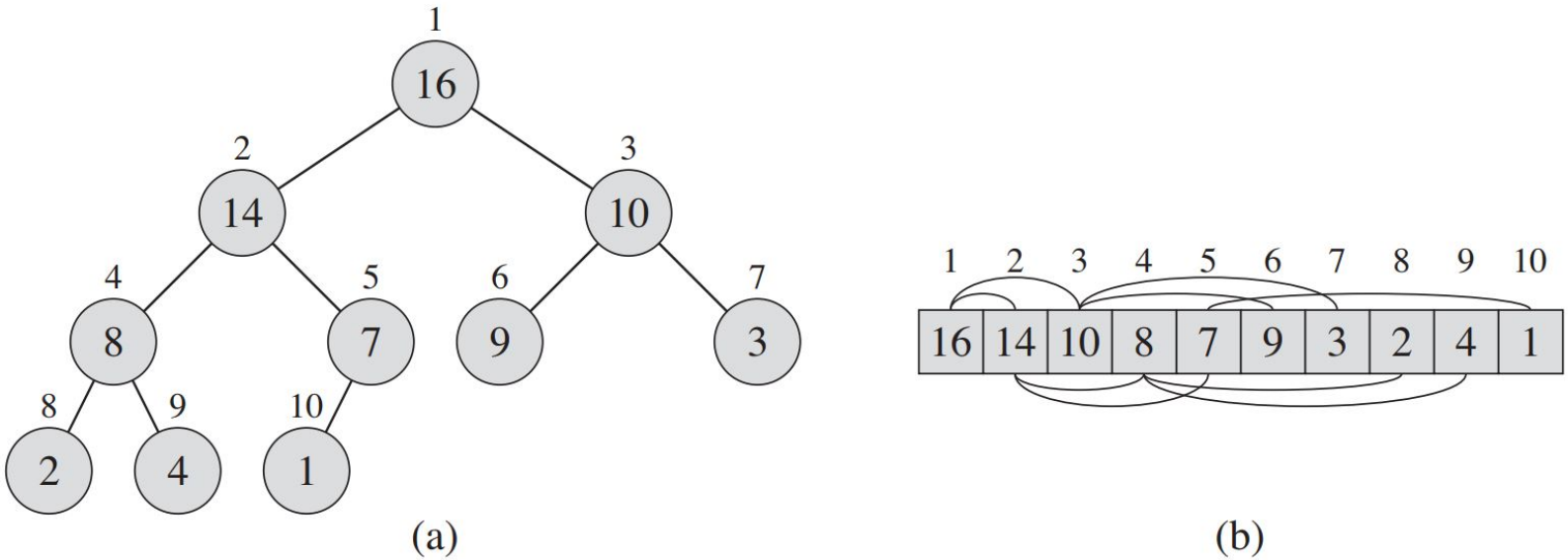
1. $T(n) = \Theta\left(n^{\log_b a}\right)$ *if* $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ *for some* $\epsilon > 0$.
2. $T(n) = \Theta\left(n^{\log_b a} \ln n\right)$ *if* $f(n) = \Theta\left(n^{\log_b a}\right)$.
3. $T(n) = \Theta\left(f(n)\right)$ *if* $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ *for some* $\epsilon > 0$, *and if* $af(n/b) \leq c \cdot f(n)$ *for all* $n \geq n_0$, *where* c *and* n_0 *are positive constant with* $c < 1$.

In merge sort $a = b = 2$ and $f(n) = D(n) + C(n) = \Theta(n)$. Thus the running time is $\Theta(n \log n)$.

Heapsort

- We have seen insertion sort and merge sort. We present heapsort that mixes the advantages of the previous two algorithms.
- Insertion sort operates **in place** with running time $O(n^2)$.
- Merge sort has running time $O(n \log n)$, but **it does not sort in place**.
- Heapsort has running time $O(n \log n)$ and sorts in place.
- The (**binary**) **heap** data structure is stored in an array, but it is used like a (binary) tree.
- An array representing a heap has two attributes: $A.length$, which is the number of the element in the array, and $A.heap.size$, which is the number of elements that are stored in the heap.

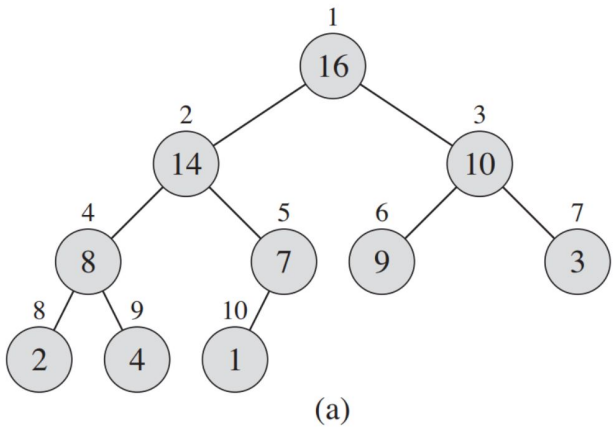
Heapsort



The tree is complete at each level, except possibly the lowest, which is filled from the left.

- In (a) we have a tree representing a max–heap (we give the definition in the next slide); the nodes contain the stored objects and above each node there is the corresponding index.
- In (b) we have the corresponding array.

Heaps construction



(a)

The root is $A[1]$. A parent node can have two children, a left and a right one. PARENT(i) returns $\lfloor i/2 \rfloor$. If a node i has two children, LEFT(i) returns $2i$, RIGHT(i) returns $2i + 1$

There are two types of heaps: max-heaps and min-heaps.

- For max–heap, the following property is true:

$$A[Parent(i)] \geq A[i]$$

- For min–heap, the following property is true:

$$A[Parent(i)] \leq A[i]$$

For the heapsort algorithm we use a max–heap.

Heap functions

- MAX-HEAPIFY procedure maintains the max–heap property. It has running time $O(\log n)$
- BUILD-MAX-HEAP procedure creates a max–heap from an non-ordered input array. It runs in $O(n)$ time.
- HEAPSORT procedure sorts an array in place with a running time $O(n \log n)$.

MAX–HEAPIFY

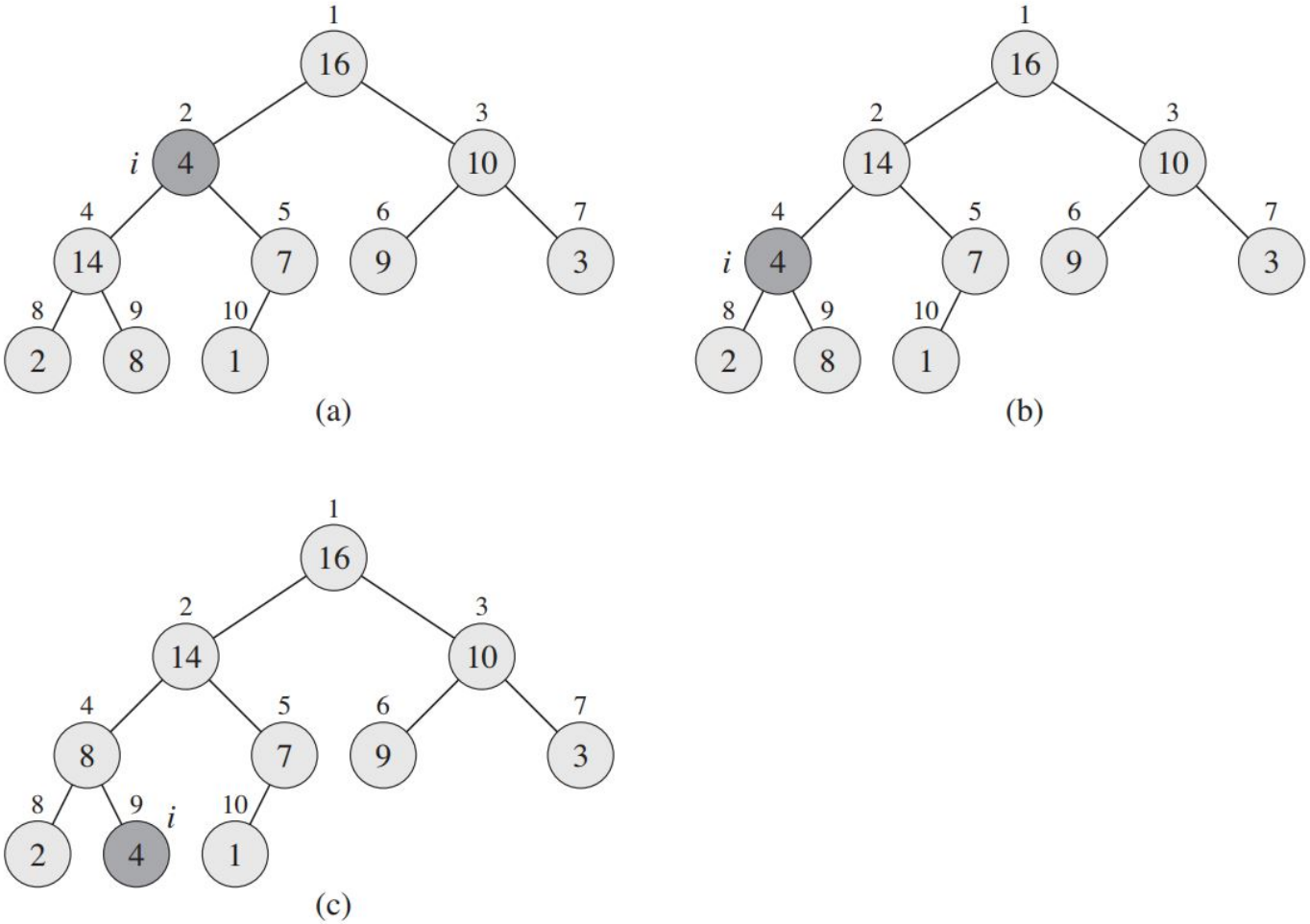
```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

The Input of MAX–HEAPIFY is an array A and an index i of the array, where we assume that the binary trees with roots LEFT(i) and RIGHT(i) are max–heaps and $A[i]$ may be smaller of its children.

At each step the largest of the elements $A[i], A[\mathrm{LEFT}(i)]$ and $A[\mathrm{RIGHT}(i)]$ is determined and its index becomes the *largest*. In case the largest is one of the child, parent and largest child are permuted. Then we iterate the process to the sub–tree with the new root.

MAX–HEAPIFY: Example

The figure below shows max-heaps, obtained with the MAX–HEAPIFY procedure.



MAX–HEAPIFY: Running time

- Let n be the size of the sub–tree with root i to which we call MAX–HEAPIFY.
- The first step is to fix up the relationship among the elements on i and its children. This takes $\Theta(1)$ time.
- Then, if needed (in the worst case), we iterate the MAX–HEAPIFY to the tree with root the largest child of i.
- A children subtree has size at most $2n/3$. Thus

$$T(n) \leq T(2n/3) + \Theta(1)$$

- Master Theorem tells us that MAX–HEAPIFY has running time $O(\ln n)$.

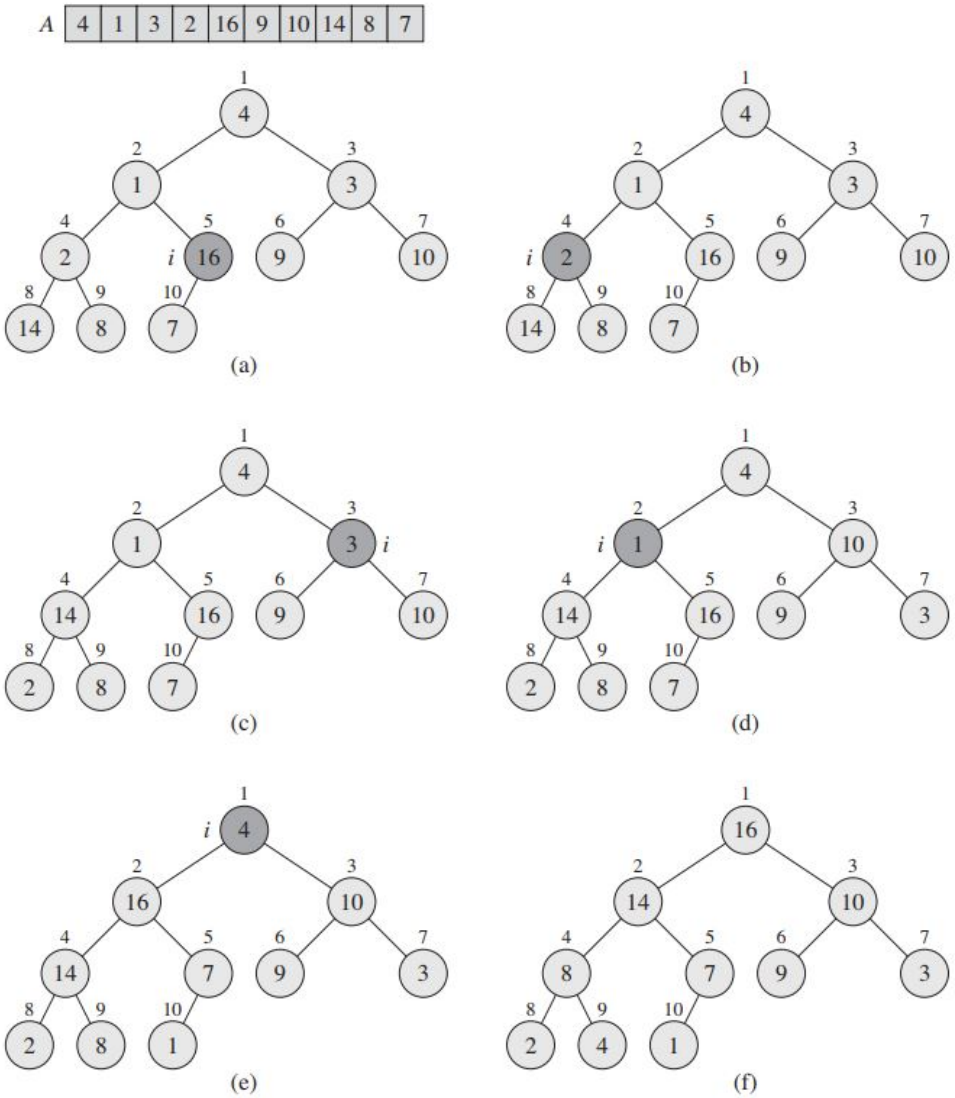
BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

It converts an array $A[1, \ldots, n]$ into a max–heap. The nodes in $A[\lfloor n/2 \rfloor + 1, \ldots, n]$ are leaves (i.e. do not have any children).

We call MAX–HEAPIFY for all nodes except the ones in the last level of the tree (the one with only leaves). One can see that the loop invariant holds at each step. The running time is $O(n)$ (the proof can be read on the book).

BUILD-MAX-HEAP: Example



Heapsort

The goal is to sort (in increasing order) an array $A[1, \ldots, n]$. The algorithm starts with the call of BUILD–MAX–HEAP. Therefore $A[1]$ is a max of the elements in A.

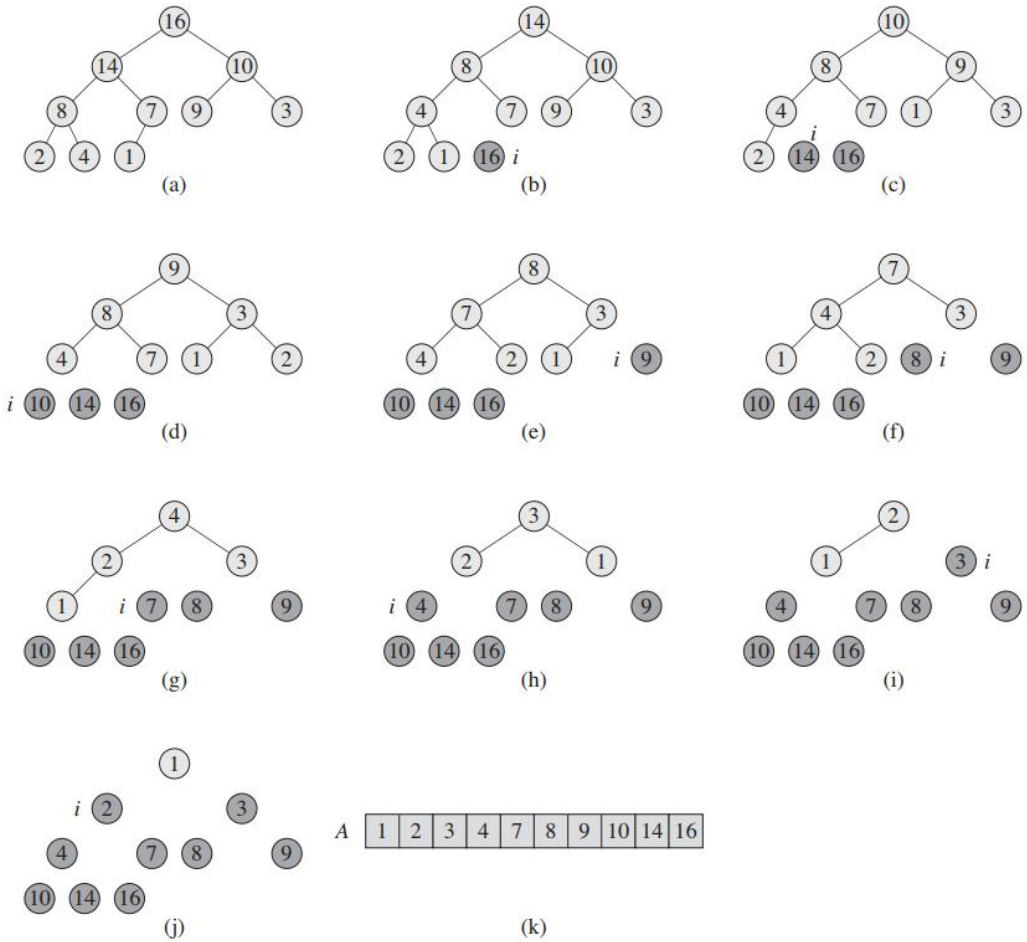
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Lines 2 and 3 (in the first loop) exchange $A[n]$ with $A[1]$. Line 4 excludes $A[n]$ form the heap. Line 5 create a new Max-Heap without $A[n]$

The loops continue with final point $n - 1$ and so on. The running time is $O(n \log n)$, because BUILD–MAX–HEAP takes $O(n)$ and each of the $n - 1$ calls to MAX–HEAPIFY takes $O(\log n)$.

Heapsort: Example

The example start after the call BUIL–MAX–HEAP:



Heapsort: conclusion

- It sorts in-place, i.e. does not require a separate array.
- Excellent complexity $O(n \log n)$, compared to the $O(n^2)$ of the insertion sort and others.
- However, a good implemention of quicksort is usually faster, even though they have same average complexity $O(n \log n)$.
- We will see it in the next chapter on randomized algorithms.

Sorting algorithms complexity

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

n denotes the number of object to sort. For counting sort, the items are integers in $\{0, 1 \ldots, k\}$. In Radix the number have d–digits

References: The material contained in these slides is taken form the book "Introduction to Algorithms: third edition": Section 2.3.1, Section 2.3.2, Section 4.5, Section 6.1, Section 6.2, Section 6.3, Section 6.4.