Introduction to Data Structures and Algorithms

Chapter: **Sorting**

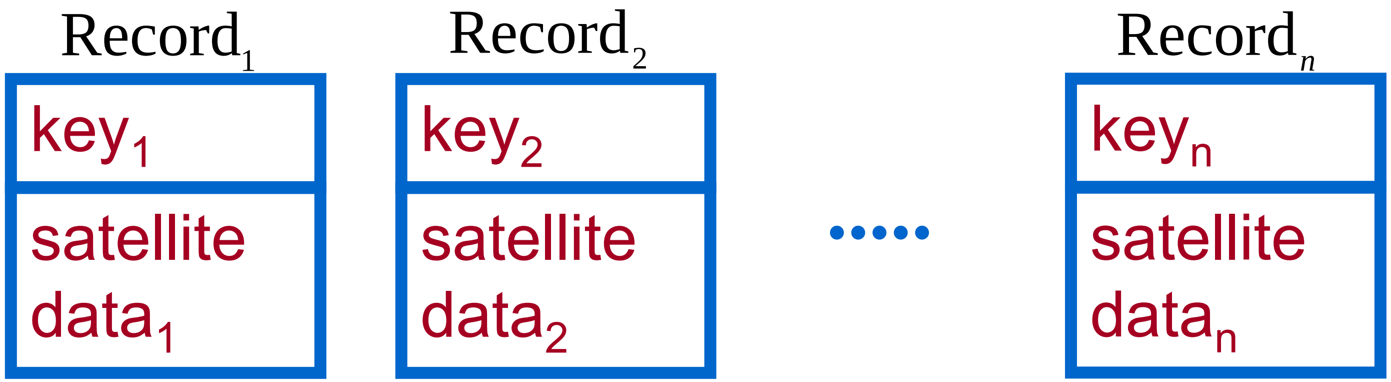
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Sorting

- Given a sequence of **data records**,
  each containing a **key** field and possibly some '**satellite data'**.

- Be K the set of keys.
  Assume there is a (non-strict) ordering relation '≤' (∈ K x K)
  (so the keys may be ordered according to '≤').



Sorting

The "sorting problem"

- If the original order of the data records is such, that the sequence of the corresponding keys is

$$(key_1, key_2, \ldots, key_n)$$

- Then the sorting problem is,
  - to find a permutation (reordering) of the input sequence of the data records such that for the corresponding keys

    $$(key_{1'}, key_{2'}, \ldots, key_{n'})$$

    the following holds:

    $$key_1' \leq key_2' \leq \ldots \leq key_n'$$

Sorting

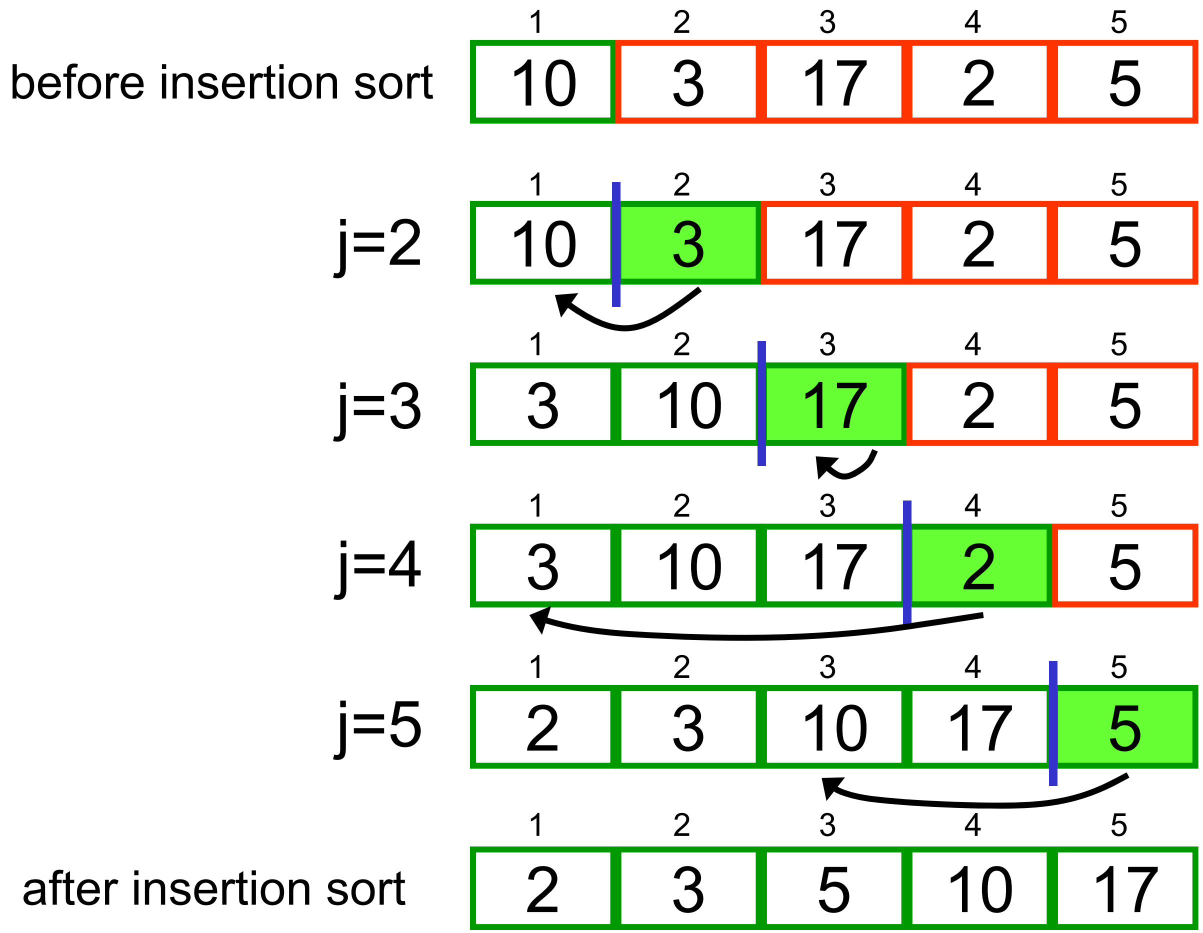
- There are many applications of sorting in practice
- We abstract from the particular application
  and from the satellite data
  and assume that the keys are numbers
  on which the relation "≤" (or analogously "≥") is defined.
- Some people think
  that sorting is the "most fundamental problem"
  in the study of algorithms!
- We start with a simple sorting algorithm: "Sort by **insertion**"
  (the same algorithm we already know a little from the
  introduction):

Insertion sort

- The input is an array A[1..n]
  containing a sequence of n = length[A] keys to be sorted.
- The algorithm in pseudo code:

```
Insertion_sort(A)
   for j:=2 to length[A] do
       key := A[j]
       i := j-1
       while i>0 and A[i]>key do
          A[i+1] := A[i]
          i := i-1
       A[i+1] := key
```

- Example (Insertion Sort)



Sorting

- Observations
  - Sorting is done “in place”,
    i.e. the keys are rearranged within the array A.
  - The basic operation is the comparison of two keys,
    as well as the assignment of values to a variable.
- Such an algorithm is called a “comparison sort”

Correctness of the algorithm (1)

- A typical way of showing that an algorithm is correct, is to find a **loop invariant.**
- A loop invariant is a property with three aspects:
  - **Initialisation:** The property is true at the beginning, i.e. just before the first iteration of the loop
  - **Maintenance:** It is true before an iteration of the loop, and it remains true before the next iteration
  - **Termination**: The loop terminates and so the loop invariant gives us a useful property to show that the algorithm is correct.

Correctness of the algorithm (2)

- We can show that the **insertion sort algorithm** is **correct** by proving the following **loop invariant**:

  **At the beginning of the j-th iteration of the for loop,
  the subarray A[1..j-1] contains
  the first j-1 elements of the input array
  in sorted order.**

- As this loop invariant
  - holds at the **beginning** (j=2),
  - It is **maintained** from one iteration to the next (j → j+1),
- **upon termination**, the algorithm "Insertion_sort" is correct (here: j = n+1)!

Runtime analysis of Insertion Sort

- In the case of sorting the “basic operations of interest” are
  - **assignment** of a value to a variable
  - **comparison** of two elements

- Accordingly we define
  the runtime T(n) for sorting an array of n elements:

  T(n) = number of assignments + number of comparisons
  executed when sorting an array with n elements

(Remember: we measure the runtime of an algorithm as the number of primitive operations or “steps” executed)

Runtime analysis of Insertion Sort (explicitly)

- the for loop is executed (*n -1*) times and it contains 4 assignments
  (*n* = length [A]) $j := 2, \cdots, n \quad key := A[j] \quad i := j - 1 \quad \text{and} \quad A[i+1] := key$

- a while loop in the j-th iteration of the for loop is executed t_j times
- the cost of one iteration of the while loop is 2 comparisons and 2 assignments: $2\,\text{comp}:\ i > 0,\ \ A[i] > key\,; \quad 2\,\text{assigmt}:\ A[i+1] := A[i1] \quad \text{and} \quad i := i - 1$
- this yields the total cost:

$$T(n) = (n-1)\cdot 4 + (t_2 + t_3 + \cdots + t_n)(2+2)$$

- Best case: the array A[ ] is already sorted, the while condition is never true

$$\Rightarrow t_j = 1 \quad \forall j$$

$$\Rightarrow T(n) = (n-1)\cdot 4 + (n-1)\cdot(2+0) = 6n - 6 = \Theta(n)$$

⇨ linear Runtime !

Runtime analysis of Insertion Sort (explicitly)

- Worst case: the array A[ ] is in reverse sorted order, the while condition is true j times

$$\Rightarrow t_j = j \quad \forall j$$

$$\Rightarrow T(n) = (n-1)\cdot 4 + 4\sum_{j=2}^{n} j \;= 4(n-1) + 4(\tfrac{n(n+1)}{2} - 1)$$

$$= 2n^2 + 6n - 8 = \Theta(n^2)$$

⇨ quadratic Runtime !

Sorting

- Let us summing up and we get the following runtime for the algorithm Insertion_sort

  - **Best case** ("array is already sorted"):

    $T_{best\_case}(n) = 6n - 6 = \Theta(n)$ ⇨ Linear runtime

  - **Worst case** ("array is already sorted in reverse order"):

    $T_{worst\_case}(n) = 2n^2 + 6n - 8 = \Theta(n^2)$ ⇨ Quadratic runtime

  - Or: $T(n) = O(n^2)$ and $T(n) = \Omega(n)$

  - Please remember the more accurate notation:
    - $T_{best\_case}(n) \in \Theta(n)$, $T_{worst\_case}(n) \in \Theta(n^2)$

Introduction to Data Structures and Algorithms

Chapter: **Sorting**

- Merge Sort

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Sorting

- Remember:

  Worst-case runtime of insertion sort is $O(n^2)$.
- Can we do better? And if so, how?

- **Idea: "Divide-and-conquer approach" – three steps:**
  - Divide The original problem is split into smaller sub problems
  - Conquer As long as the solution of the (smaller) sub problems is not trivial - these are solved using the same procedure (recursion)
  - Combine The solutions of the sub problems are suitably combined to a solution of the larger problem

The same idea was employed for the algorithm **pow** (iterative squaring algorithm for computing Fibonacci numbers)!

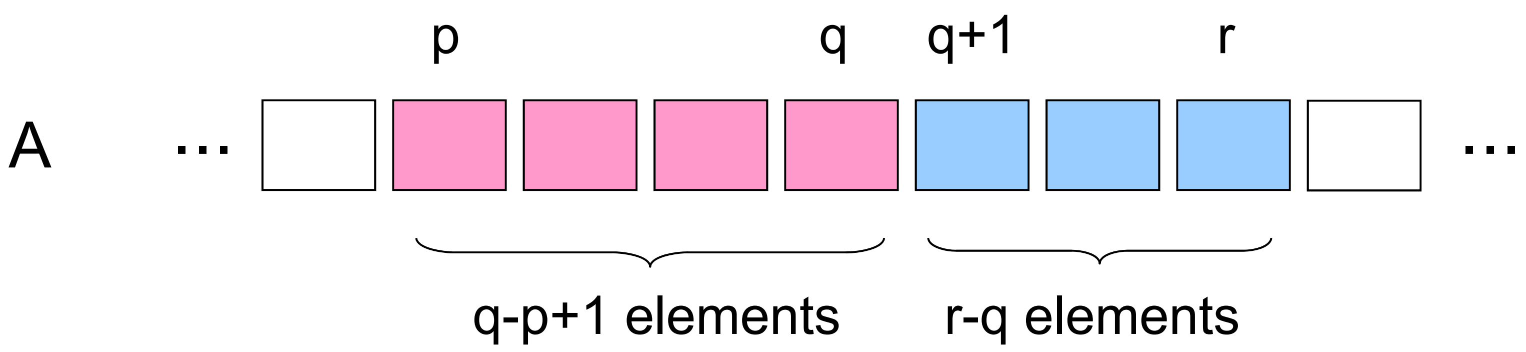
Merge Sort

- **DIVIDE**: Divide the n-element sequence (array) to be sorted into two subsequences of (about) n/2 elements each
- **CONQUER**: Sort the two subsequences recursively using merge sort
- **COMBINE**: Merge the two sorted subsequences to produce one sorted sequence

- The recursion stops when the sequence to be sorted has length 1, since every sequence of length 1 is already sorted
- Now starts the Combine process to build sorted sequences of length 2
- … of length 4 …

Merge Sort

- The most important part of Merge Sort is the merging (COMBINE)
- Procedure Merge (A,p,q,r)
  - A is an array
  - p, q, r are indices such that p ≤ q < r



  - Procedure Merge assumes that A[p .. q] and A[q+1 .. r] are already sorted
  - It merges them to a single sorted subarray to replace the current subarray A[p .. r]

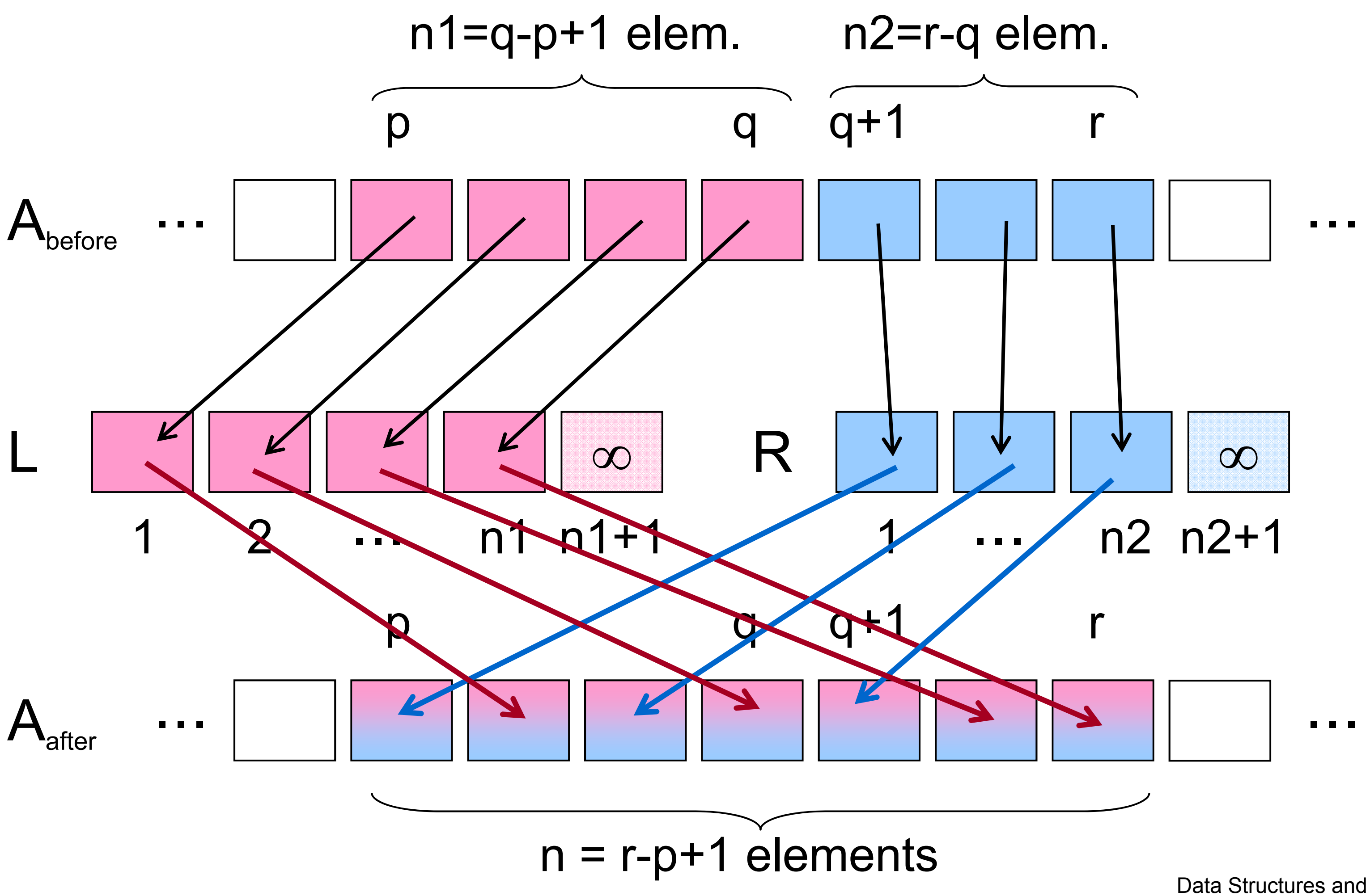
Merge Sort

Pseudo code for Merge(A, p, q, r)

```
n1 := q-p+1 // nr. elem. “left”
n2 := r-q      // nr. elem. “right”

 // create arrays
     L[1..n1+1] and R[1..n2+1]
 for i:=1 to n1 do
    L[i] := A[p+i-1]
for j:=1 to n2 do
    R[j] := A[q+j]
L[n1+1]  := ∞
R[n2+1] :=  ∞
i := 1
j := 1
for k:=p to r do
    if L[i] <= R[j]
       then A[k] := L[i]
              i := i+1
       else A[k] := R[j]
              j := j+1
```

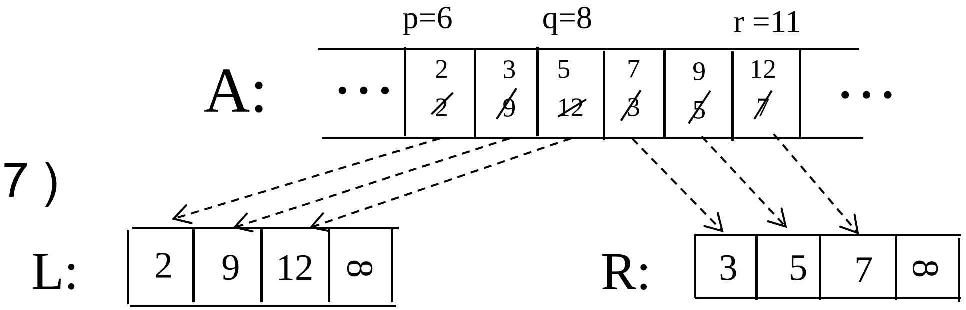
Merge Sort

- Illustration of Merge



Merge Sort

- Example:
  - A[6 .. 11] = (2, 9, 12; 3, 5, 7)
  - call Merge (A, 6, 8, 11)



- **Runtime of Merge** = Θ(n),

  where n = n1+n2 = r-p+1
  - each of lines 1-3 and 8-11 takes constant time
  - the **for** loops in lines 4 - 7 take Θ(n1+n2) = Θ(n) time
  - the **for** loop in lines 12 - 17 have n iterations, each one with constant time ⇨ Θ(n)

- **Data size for Merge** = 2·length(A) = 2n = Θ(n)
  (compare with Insertion Sort that sorts „in place“)

Merge Sort

Complete pseudo code for Merge Sort

(using procedure MERGE as subroutine)

```
Merge_sort(A,p,r)

 if p < r then
   q := floor((p+r)/2)
   Merge_sort(A,p,q)
   Merge_sort(A,q+1,r)
   Merge(A,p,q,r)
```

- Initial call:
  Merge_sort(A, 1, length(A))   (where length(A) = n)

Merge Sort

- Example for Merge_sort

A							
	9	4	2	17	2	3	5

- call Merge_sort (A, 1, 7)

9	4	2	17	2	3	5	before
2	2	3	4	5	9	17	after

Merge Sort

Runtime analysis for Merge_sort

- For simplicity we assume that n = length(A) is 2^k for $k \in \mathbb{N}$.
  (It can be shown that this assumption does not affect the order of growth.)

Now, the following recurrence equation is developed:

- Define **T(n) = worst case running time of Merge_sort**, where $n = 2^k$
- Merge_sort on just one element takes constant time c
- DIVIDE: This step computes the middle of the sub array, which takes constant time ⇨ $Divide(n) = \Theta(1) = c$
- CONQUER: Recursively solve two sub problems each of size n/2, which contributes 2T(n/2) to the running time ⇨ $Conquer(n) = 2T(n/2)$

Merge Sort

Runtime analysis for Merge_sort

- COMBINE: Merge
  ⇨ Combine(n) = Θ(n) = c·n

- We simplify the calculation by defining S(n) = Divide(n) + Combine(n)
  ⇨ S(n) = Θ(1) + Θ(n) = c·n

- Adding S(n) to the running time of CONQUER gives:

$$T(n) = \begin{cases} c & \text{if } \ n = 1 \\ 2T(n/2) + c \cdot n & \text{if } \ n > 1 \end{cases}$$

(where constant c represents the time required to solve problems of size 1)

Merge Sort

Runtime analysis for Merge_sort

- For solution of this recurrence equation we use a **<u>recursion tree</u>** and add up the cost

- So, lastly we get

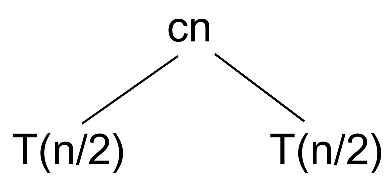
  ⇨ **$T(n) = cn \log_2 n + cn = \Theta(n \log_2 n)$**

  ⇨ The worst case runtime of Merge_sort is $\Theta(n \log_2 n)$

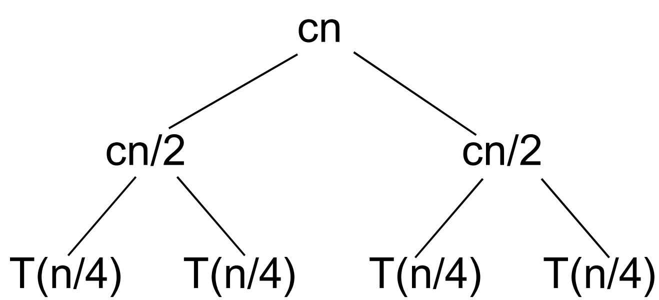
Merge Sort

Recurrence tree:

T(n) = cn + 2T(n/2)



T(n) = cn + 2T(n/2) = cn + 2[cn/2 + 2T(n/4)] = cn + (cn/2 + cn/2) + 4 T(n/4)



Merge Sort

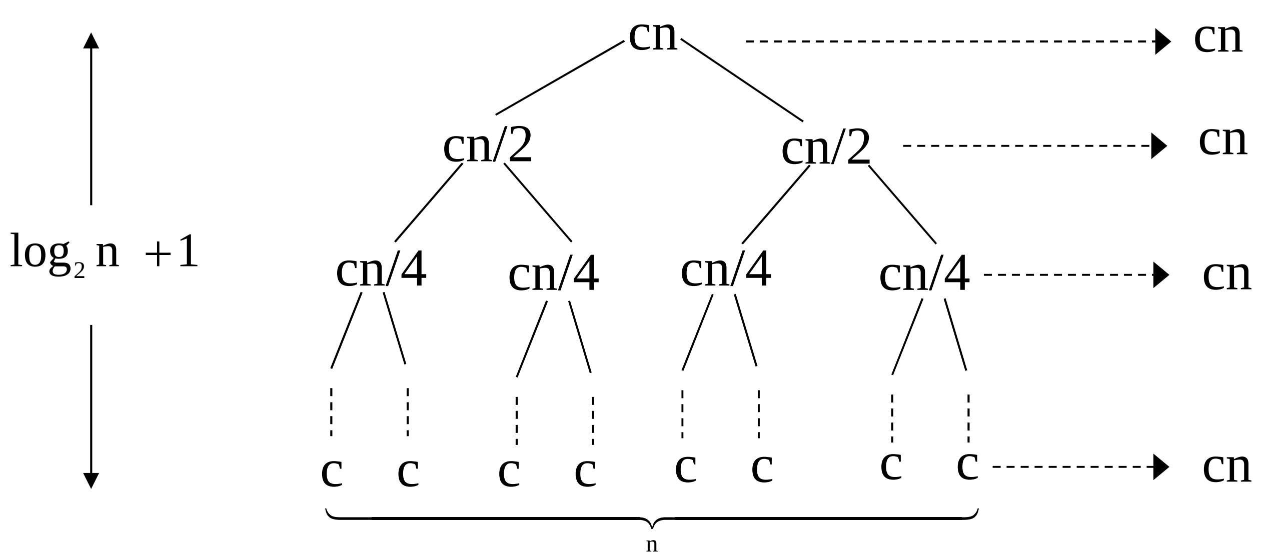
Recurrence tree:

T(n) = cn + 2T(n/2) = cn + 2[cn/2 + 2T(n/4)] = cn + (cn/2 + cn/2) + 4 T(n/4) =

$$= c \cdot n + (c \cdot n/2 + c \cdot n/2) + \overbrace{c \cdot n/4 + \cdots + c \cdot n/4}^{4-times} + \cdots + \overbrace{c \cdot n/n + \cdots + c \cdot n/n}^{n-times} =$$

$$= \overbrace{cn + cn \cdots + cn}^{(\log_2 n+1)-times} = (\log_2 n + 1) \cdot cn$$

cost:



$$\Rightarrow \text{Total cost: } (\log_2 n + 1) \cdot cn = cn \cdot \log_2 n + cn = \Theta(n \cdot \lg n)$$

Introduction to
Data Structures and Algorithms

Chapter: **Sorting**

- Heap Sort

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Heap Sort

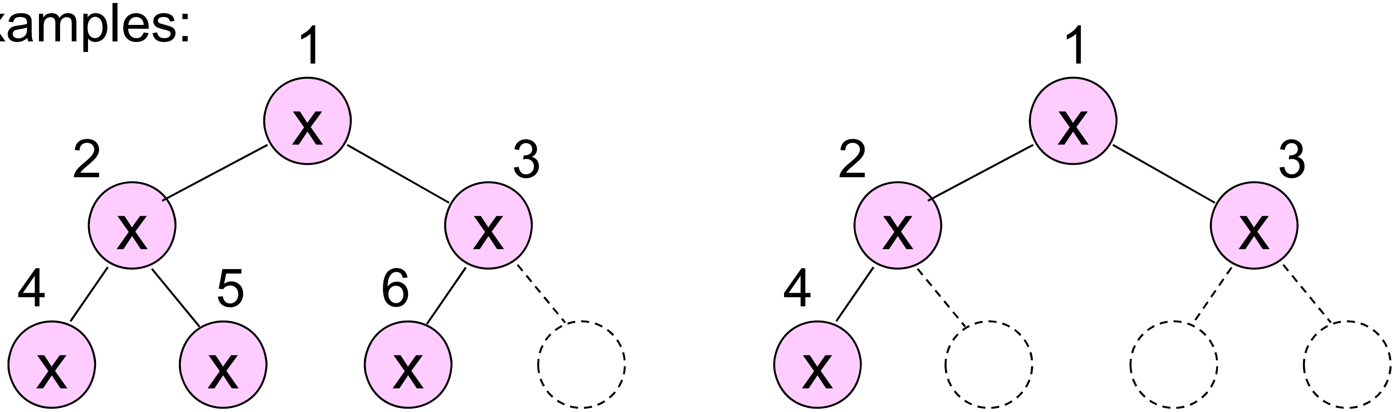
- Heap sort
  combines the advantages
  of insertion sort and merge sort

  - Its runtime is $O(n \log n)$
    (in contrast to insertion sort's $O(n^2)$)
  - It sorts in place
    (in contrast to merge sort
    where twice as much memory is needed)
  - It introduces another algorithm design technique: the use of **data structure** (here a "heap")

Heap Sort

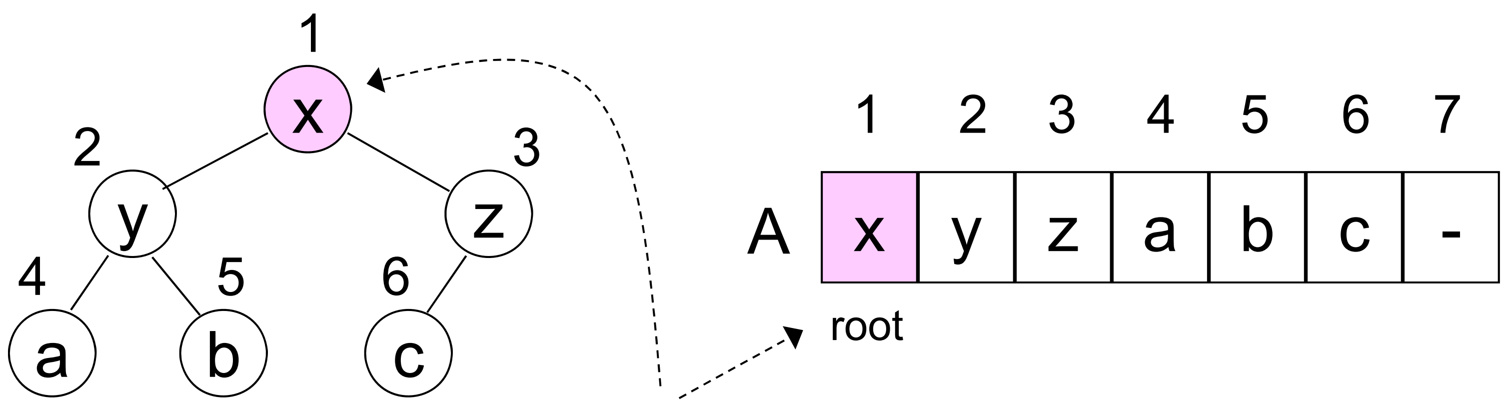
- A **Binary Heap** is a data structure
  that can be viewed as a “nearly complete binary tree”
- The Binary Heap is filled completely,
  with the exception
  that at the lowest level
  possibly one or more right most elements
  may be missing

Examples:



Heap Sort

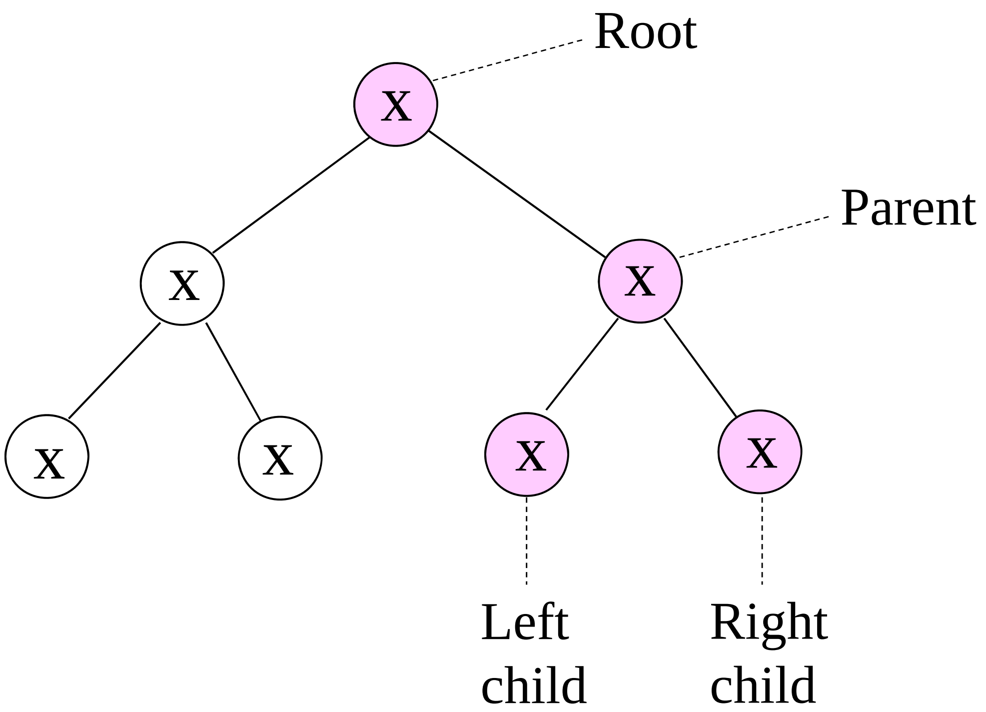
- Each **Binary Heap** can be represented by an array in the following way



{The root of the tree is A[1]}

- The mapping of the elements of the tree to the array elements is one-to-one
  - starting with the root of the tree
  - and then sequentially down the layers of the tree always from left to right

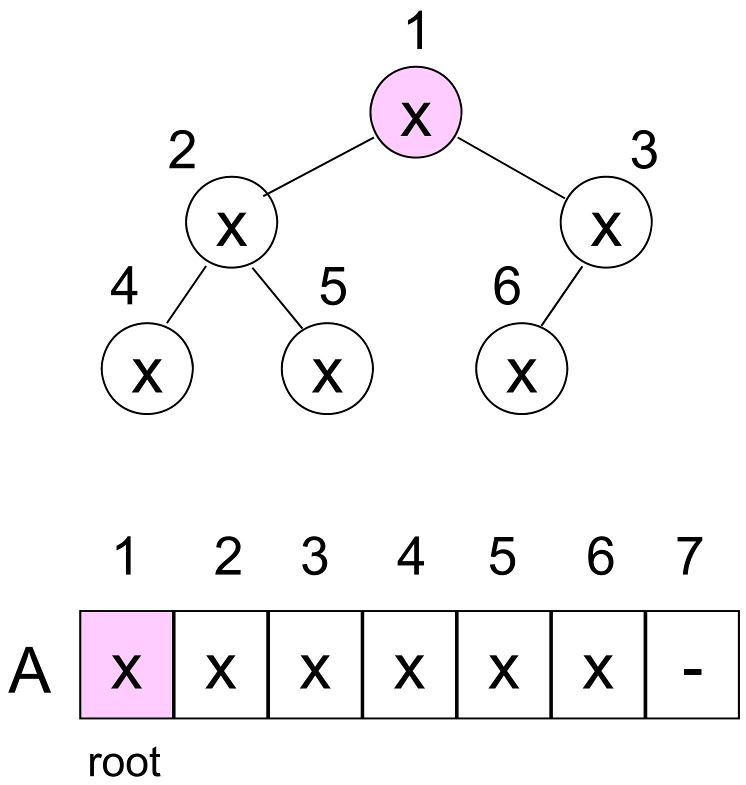
Parent, left or right child



Heap Sort

Functions for computing the index of parent, left or right child

- Root()
  return 1
- for i > 1:
  Parent(i)
  return floor(i/2)
- Left(i)
  return 2*i
- Right(i)
  return 2*i + 1



Heap Sort

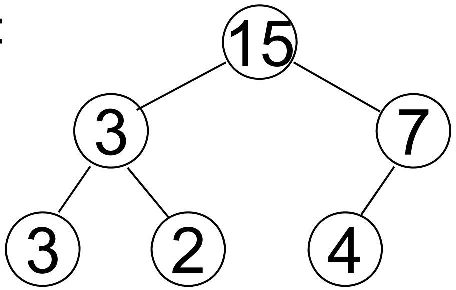
Max-heaps

- A heap A is a Max-heap, if for all nodes i (i ≠ root)

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

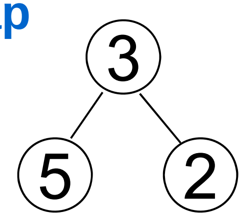
- That means
  - The largest element is stored at the root
  - For all subtrees S:
    all elements in S are not larger than the element at the root of S

Example 1:
max-heap



Example 2:
not a max-heap

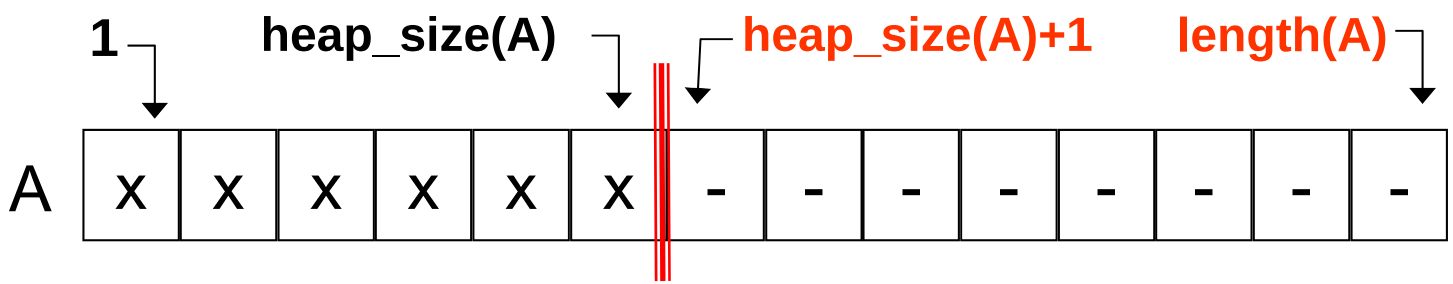
(as 5 ≥ 3)



Heap Sort

Max-heaps

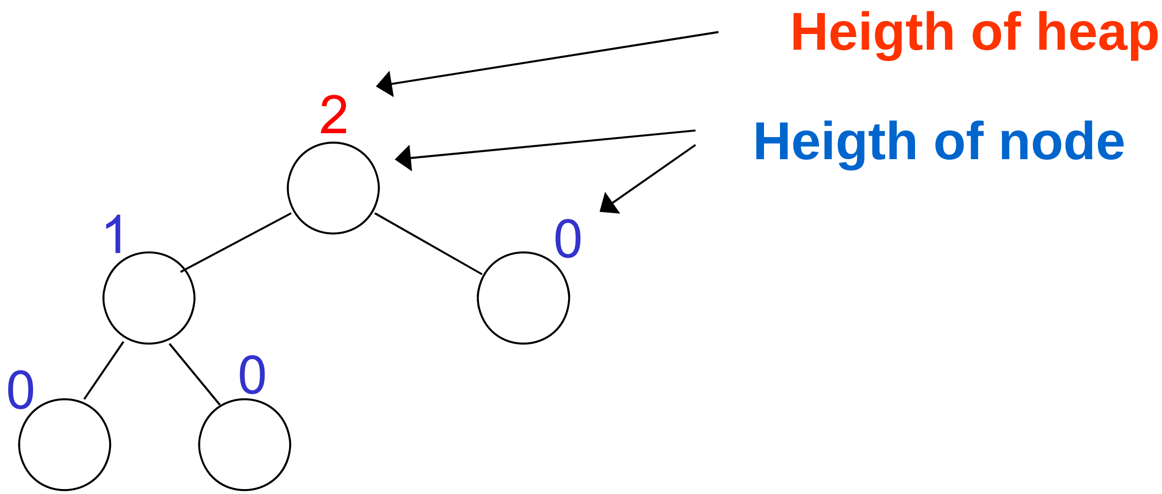
- An array A that represents a heap has further on two attributes:
  - length(A) = number of elements in the array
  - heap_size(A) = number of elements in the heap stored in array A



- So elements A[heap_size(A)+1], …, A[length(A)] are not elements of the heap

Height of heaps

- The **height** of a node in a heap
  is the number of edges on the longest simple downward path
  from the node to a leaf.
- The height of a heap
  is the height of its root.



Heap Sort

Maintaining the Max-heap Property

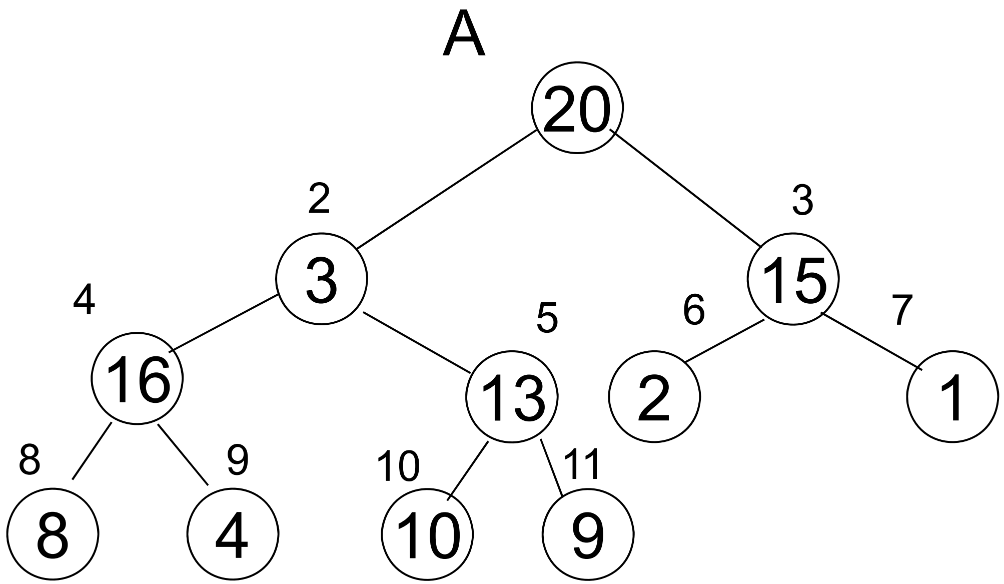
- The following procedure *Max_heapify* is important for manipulating a max-heap
- It is assumed that the binary trees rooted at *Left(i)* and *Right(i)* are max-heaps, when *Max_heapify(A,i)* is called
- If *A[i]* is smaller than one of its children, then the procedure lets it “float down”
- Upon termination, the subtree rooted at *i* is a max-heap

Maintaining the Max-heap Property

```
Max_heapify(A,i)

  l := Left(i)
  r := Right(i)

  if l<=heap_size[A] and A[l]>A[i]
     then largest := l
     else largest := i

  if r<=heap_size[A] and A[r]>A[largest]
     then largest := r

  if largest != i
     then exchange A[i] <-> A[largest]
          Max_heapify(A,largest)

      Stop!
```

- The running time of *Max_heapify* on a node of height *h* is O(h) = O(log n)

Maintaining the Max-heap Property - Example



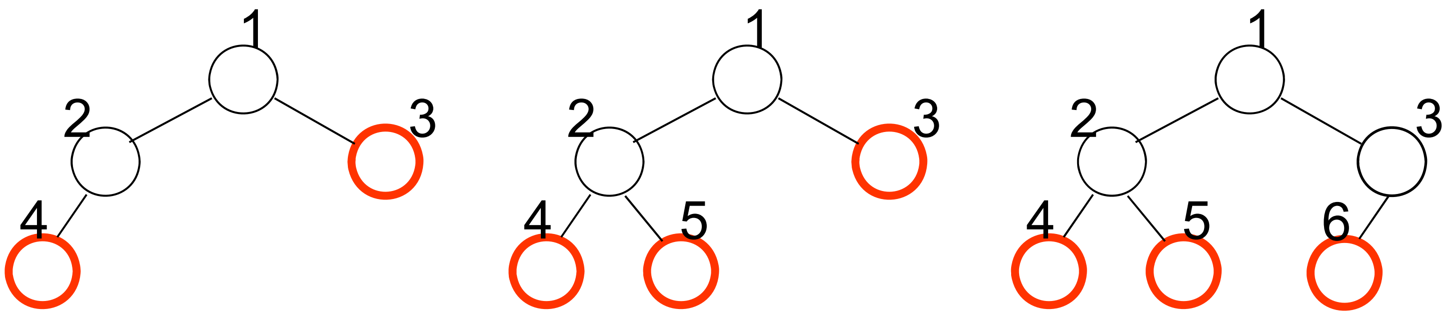
- Is this a max-heap?

  If not ⇨ call max_heapify(A, k) for suitable k

(see below: Build_max_heap)

Building a Max-heap

- Note that in the array representation for storing an *n*-element heap, the leaves are indexed by
  - floor(n/2) +1, floor(n/2) +2, …, n-1, n
- Example



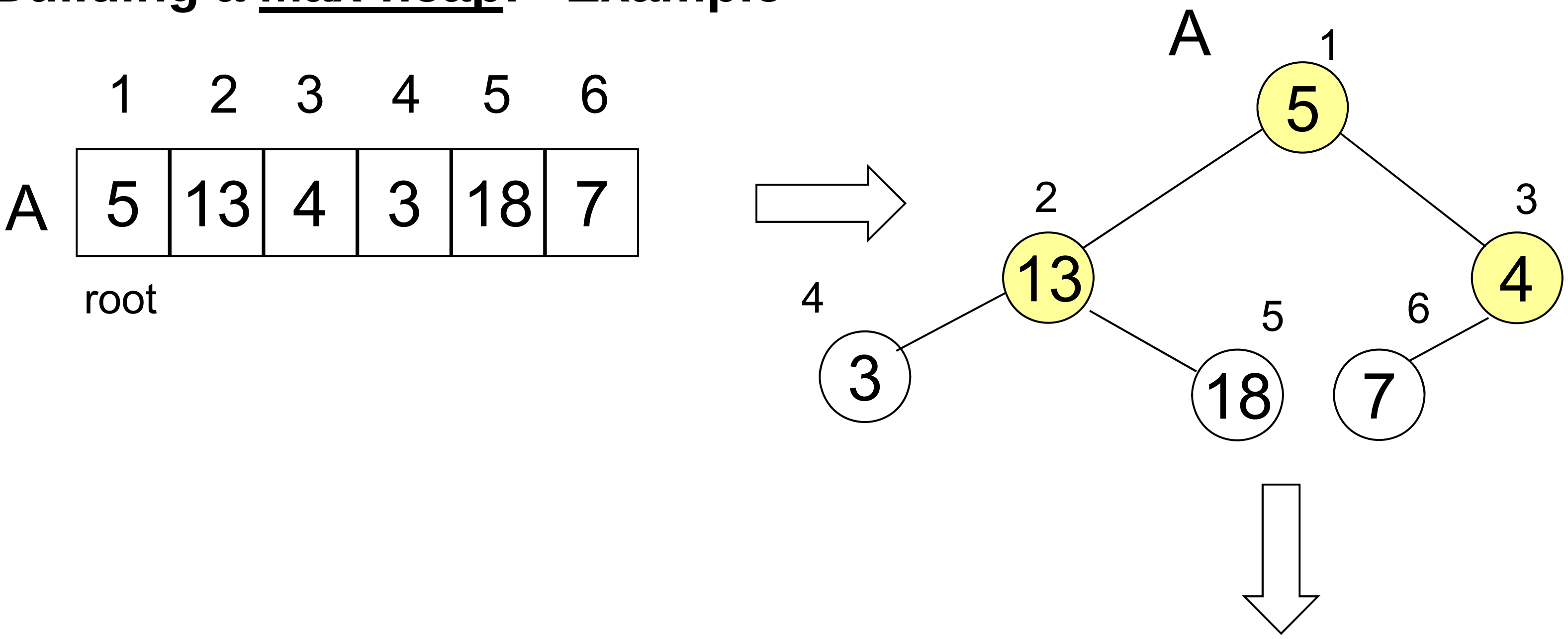
- In general, leaves are represented by those indices *i* where *Left(i)* = 2*i is outside the array boundary, i.e. where 2*i > n, or i > n/2.

Building a Max-heap

- Build a max_heap out of an **arbitrary heap A**:
  - Use of *Max_heapify* in a bottom-up manner to convert some array A[1 …n] into a max-heap
  - Go through all nodes that are not leaf nodes - largest to smallest index (the leaf nodes are max_heaps of course!)
  - and run Max_heapify on each of these nodes:

```
Build_max_heap(A)

  heap_size[A] := length[A]
  for i:=floor(length[A]/2) downto 1 do
      Max_heapify(A,i)
```

Building a <u>Max-heap</u>: Example

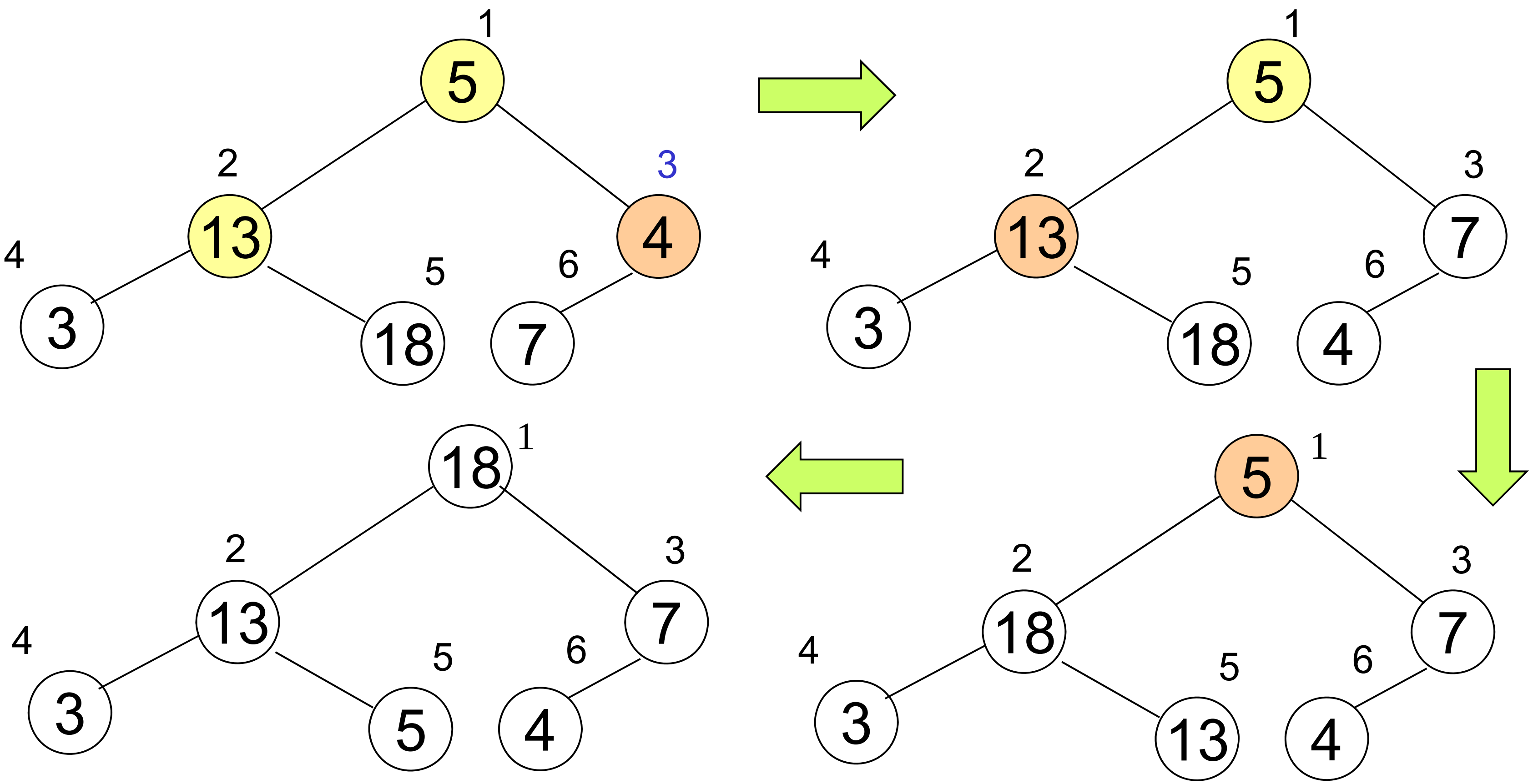


Build_max_heap (A)

- It can be shown that the running time of *Build_max_heap* is *O(n)*.
- For details see [Cormen et al.]

Heap Sort

Building a Max-heap: length [A] = 6 ⇨ floor(length[A]/2) = 3

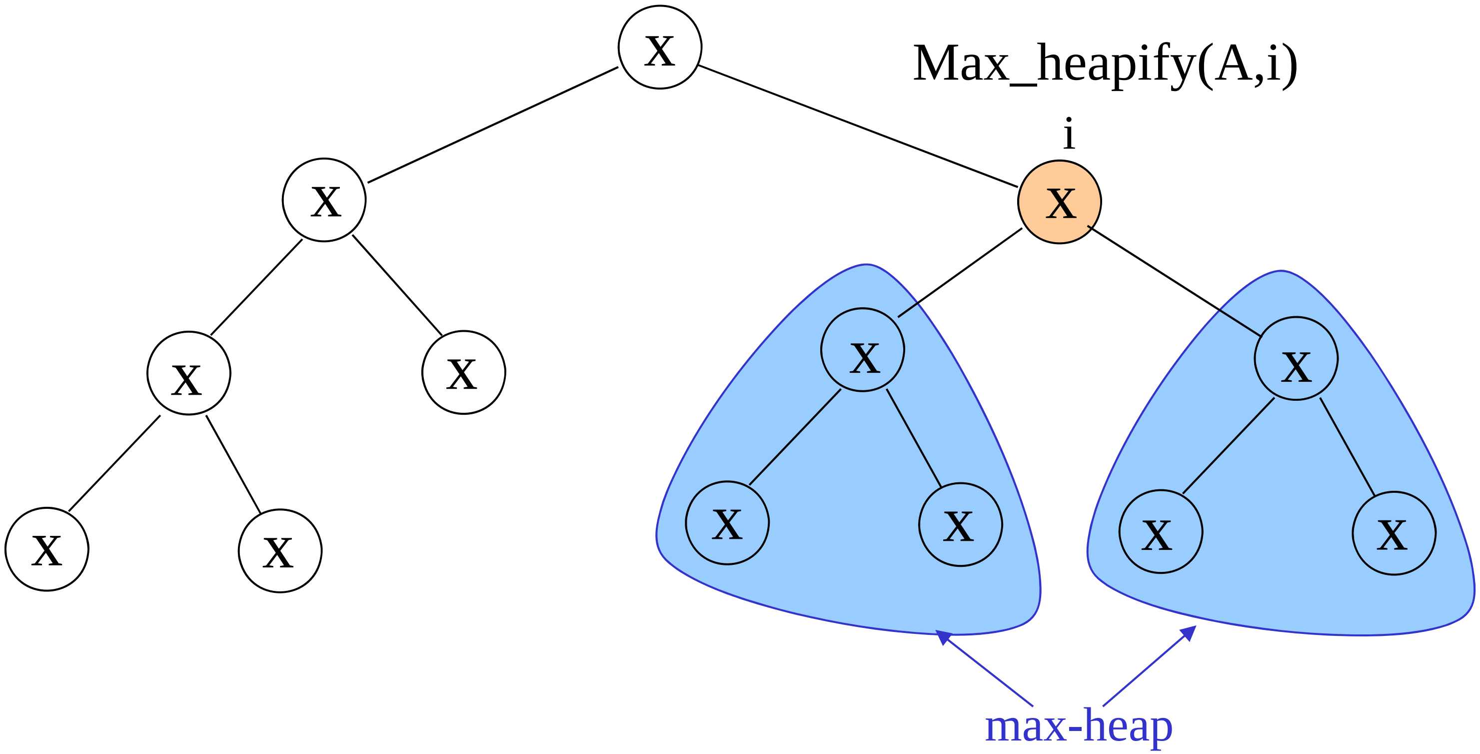


A=[5,13,4,3,18,7] → **Build_max_heap(A)** → A=[18,13,7,3,5,4]

Heap Sort

- <u>Observe:</u>

  Whenever *Max_heapify* is called on node i, the two subtrees of that node i are both **max-heaps !**



Heap Sort

The Heap Sort algorithm

Aim: a completely sorted new array

```
Heapsort(A)

  Build_max_heap(A)

  for i:= length[A] downto 2 do
      exchange(A[1],A[i])
      heap_size[A] := heap_size[A]-1
      Max_heapify(A,1)
```

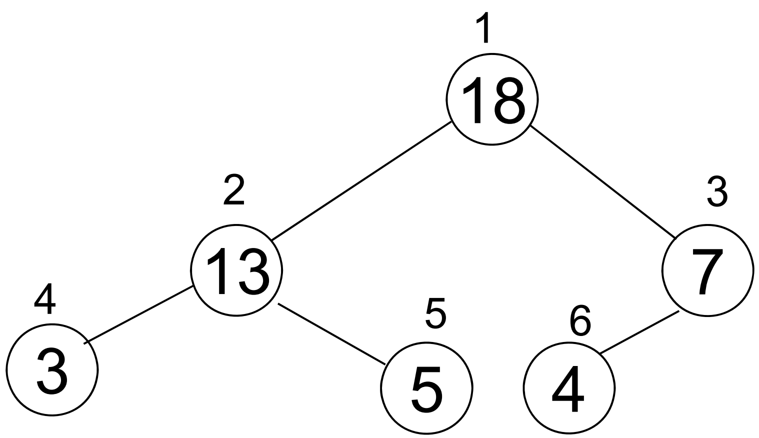
- First A is turned into a max-heap [Run time O(n)]
- In the for loop,
  - the largest element of the heap (A[1]) is swapped with the last element of the heap and the heap-size is decreased.
  - Then A[1] … A[heap_size[A]] is turned into a max-heap by Max_heapify(A,1) [Run time (n-1) O(log n)]
- Thus the overall run time is O(n log n).

Heap Sort

Heap sort: Example

	1	2	3	4	5	6
A	5	13	4	3	18	7
	root					

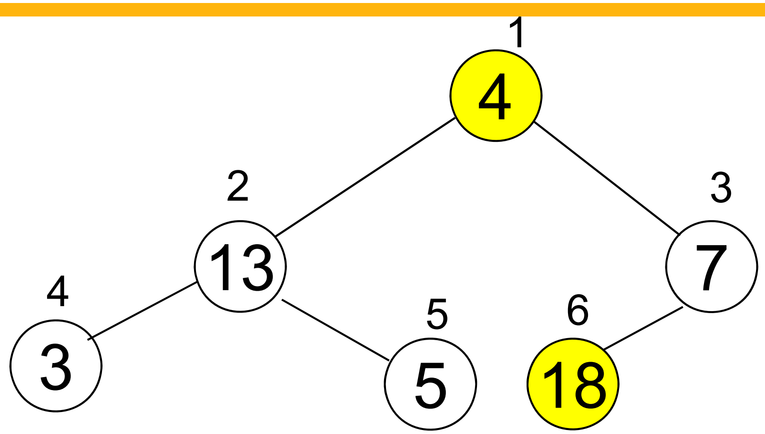
⇨ **Build_max_heap(A)**



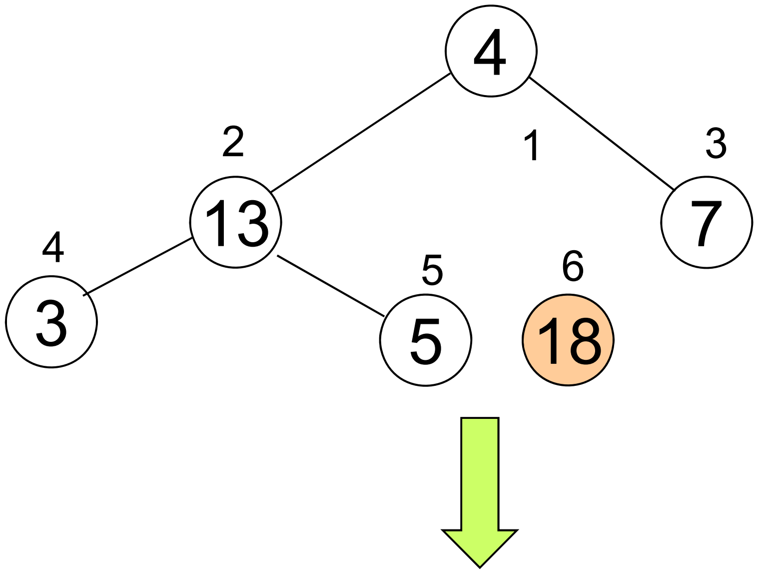
	1	2	3	4	5	6
A	18	13	7	3	5	4
	root					

Exchange (A[1],A[6])

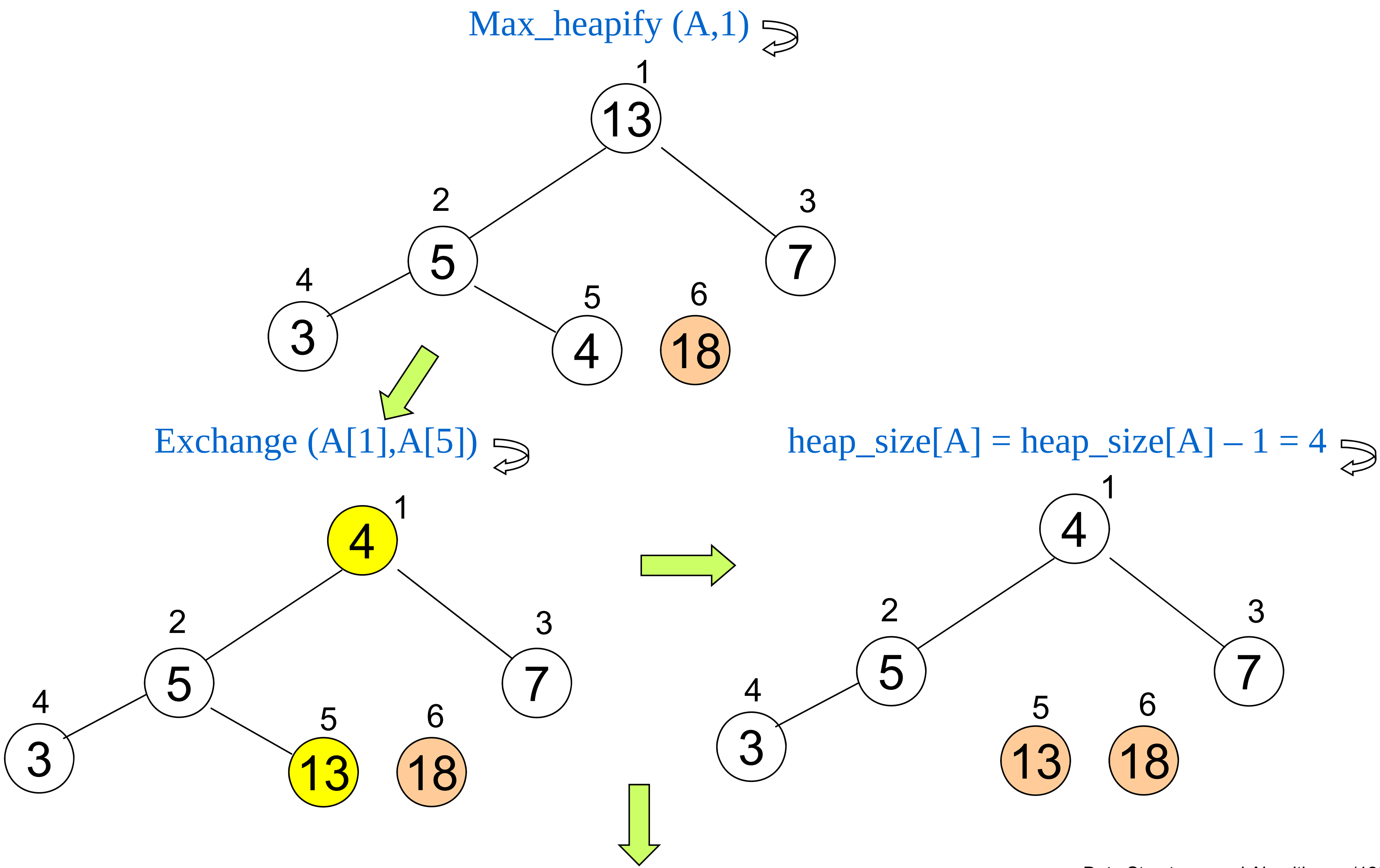
Heap Sort



heap_size[A] = heap_size[A] – 1 = 6 - 1 = 5

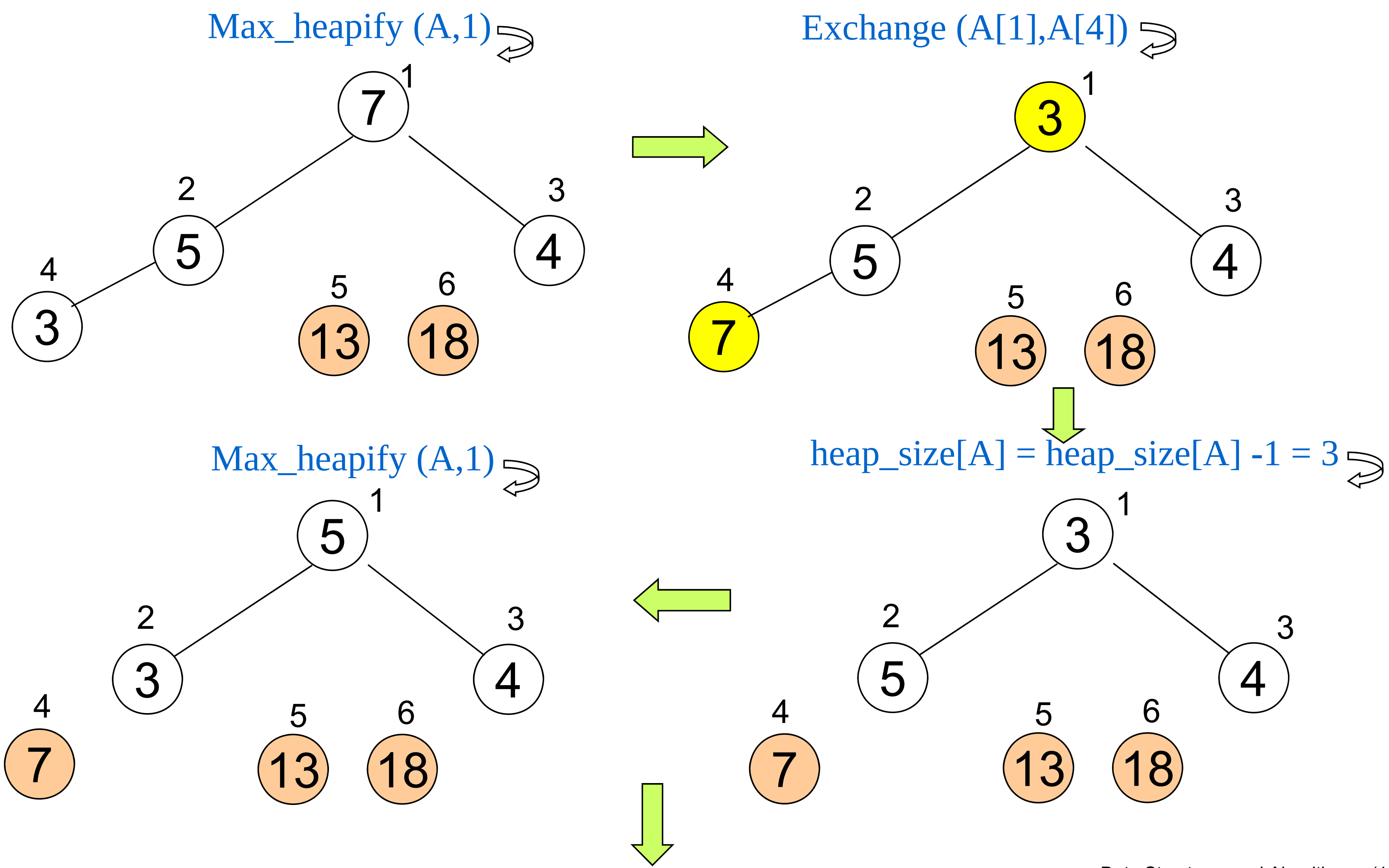


Heap Sort



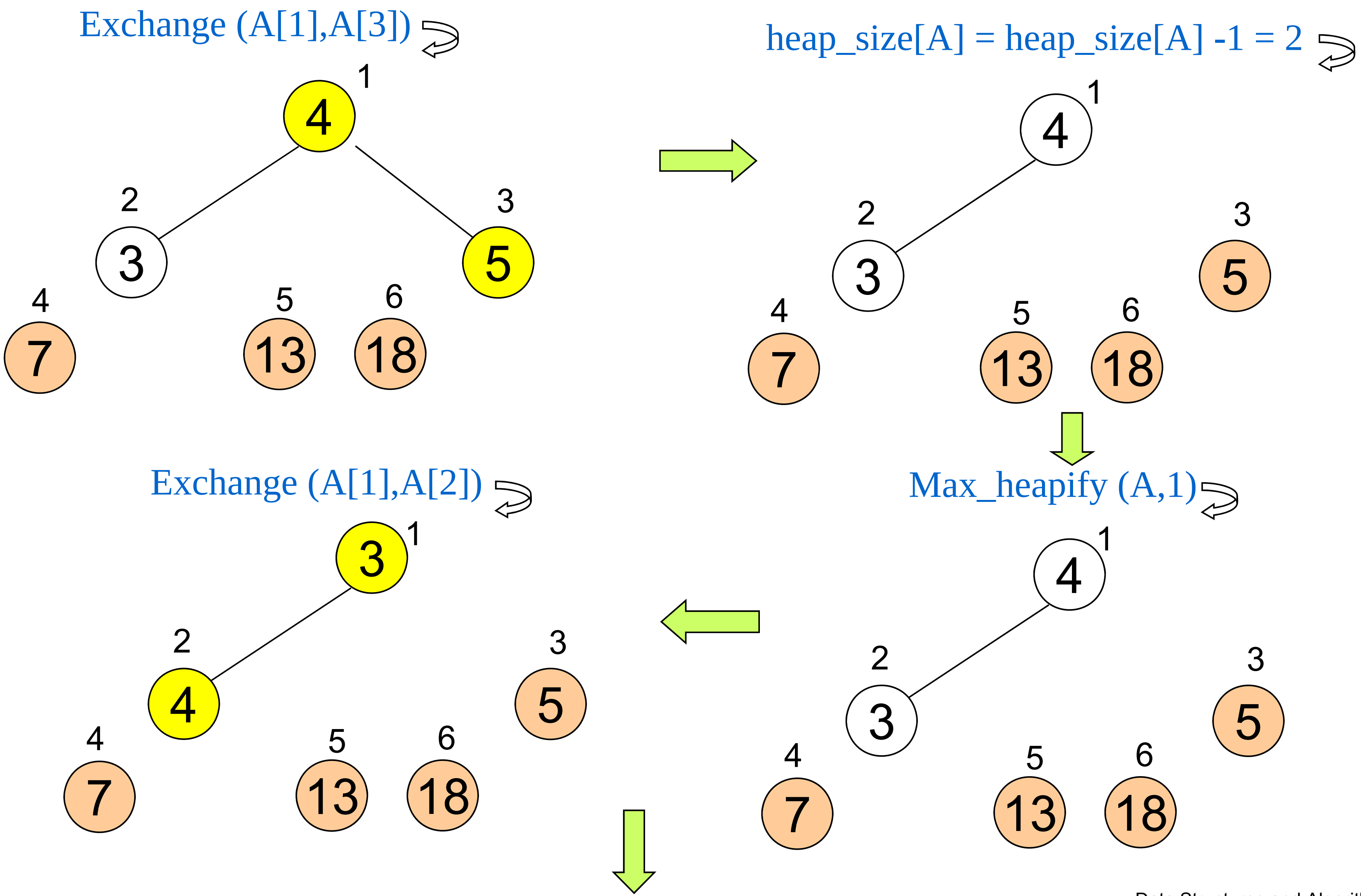
Max_heapify (A,1)
1
13
2
5
3
7
4
3
5
4
6
18
Exchange (A[1],A[5])
1
4
2
5
3
7
4
3
5
13
6
18
heap_size[A] = heap_size[A] – 1 = 4
1
4
2
5
3
7
4
3
5
13
6
18

Heap Sort



Max_heapify (A,1)
Exchange (A[1],A[4])
heap_size[A] = heap_size[A] -1 = 3
Max_heapify (A,1)
7
5
4
3
13
18
3
5
4
7
13
18
3
5
4
7
13
18
5
3
4
7
13
18

Heap Sort



Exchange (A[1],A[3])
heap_size[A] = heap_size[A] -1 = 2
Max_heapify (A,1)
Exchange (A[1],A[2])

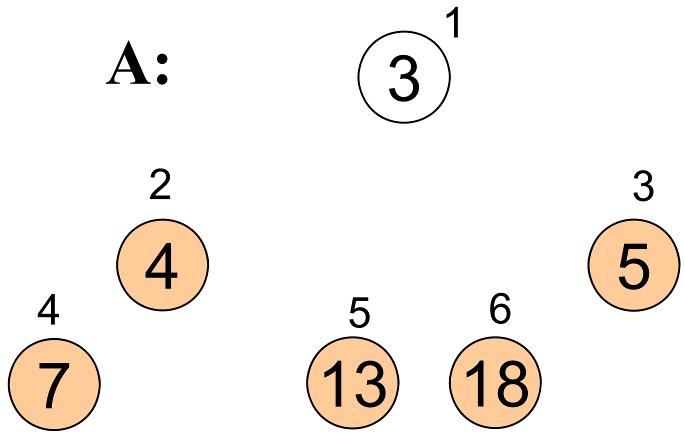
Heap Sort

heap_size[A] = heap_size[A] -1 = 1

⇨ i < 2 ➡ stop

The resulting **completely sorted** array:

A:



= A

1	2	3	4	5	6
3	4	5	7	13	18

Quick Sort

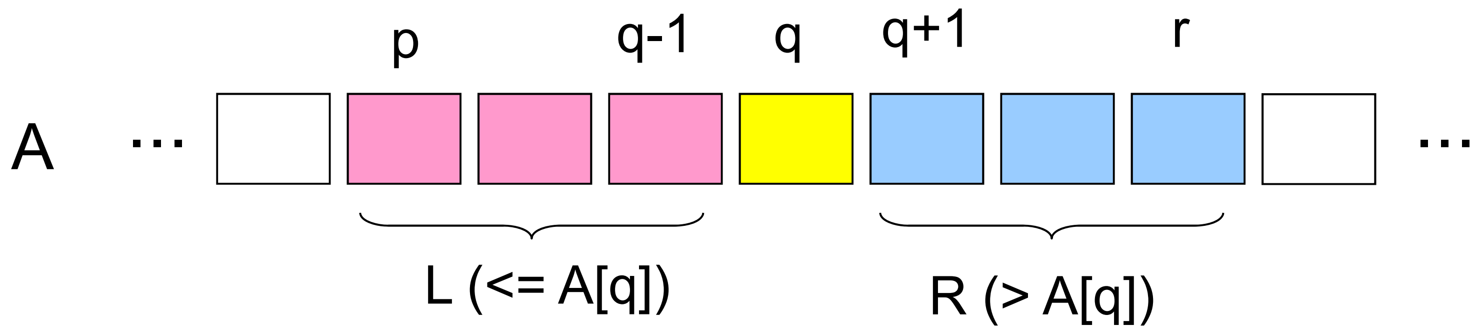
- Another well known sorting algorithm is Quick Sort.

- The characteristics of Quick Sort are similar to Heap Sort
  - Its average runtime is $\Theta(n \cdot \lg n)$ (obtained by randomization)
  - It sorts in place

- On the other hand
  - Quick Sort's worst case runtime is $\Theta(n^2)$,
    so for arbitrary input the run time is $O(n^2)$
  - But it can be shown (see Corman)
    that the average case performance of Quick Sort
    is much closer to the best case than to the worst case
    $\Rightarrow$ so Quick Sort is usually a good choice !

Quick Sort

Quick Sort is a *Divide and Conquer* Algorithm



- DIVIDE: Partition the current subarray A[p .. r] into two subarrays A[p ... q-1] and A[q+1 ...r] plus a pivot element A[q]
  - such that all elements of A[p ... q-1] are less than or equal to A[q]
  - and all elements of A[q+1 ...r] are greater than A[q]
  - the index q – the pivot index – is computed during the partitioning
- CONQUER: Sort the two subarrays A[p ... q-1] and A[q+1 … r] by two recursive calls to Quick Sort
- COMBINE: Nothing is to do for combining the two subarrays

Quick Sort

Pseudo code for Quick Sort

```
Quicksort (A,p,r)
  if p < r then
     q := Partition (A,p,r)
     Quicksort (A,p,q-1)
     Quicksort (A,q+1,r)
```

- Initial call for sorting an array A:
  Quicksort (A,1,length(A))

Partitioning the Array

Partition (A,p,r)

```
x := A[r]      // pivot element
i  := p -1

for  j := p  to  r-1 do
   if  A[j] <=  x then
       i := i+1
       exchange (A[i], A[j])
exchange (A[i+1], A[r])
return (i+1)
```

Quick Sort

Pseudo code for Randomized Quick Sort

```
Randomized-Quicksort (A,p,r)
  if p < r    then
      q := Randomized-Partition (A,p,r)
      Randomized-Quicksort (A,p,q-1)
      Randomized-Quicksort (A,q+1,r)

Randomized-Partition (A,p,r)
      i := Random (p,r)
      exchange A[r] ⟷ A[i]
      return Partition (A,p,r)
```

- **Now:** Average runtime $T(n) = \Theta(n \cdot \lg n)$