Data Structures and Algorithms

Chapter 4

Werner Nutt

Acknowledgments

- The course follows the book “Introduction to Algorithms‘”, by **Cormen, Leiserson, Rivest and Stein**, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.
- These slides are based on those developed by Michael Böhlen for this course.

(See http://www.inf.unibz.it/dis/teaching/DSA/)

- The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course

(See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ//)

DSA, Chapter 4: Overview

- About sorting algorithms
- Heapsort
  - complete binary trees
  - heap data structure
- Quicksort
  - a popular algorithm
  - very fast on average

DSA, Chapter 4: Overview

- About sorting algorithms
- Heapsort
  - Complete binary trees
  - Heap data structure
- Quicksort
  - a popular algorithm
  - very fast on average

Why Sorting?

- "When in doubt, sort" –
  one of the principles of algorithm design.

- Sorting is used as a subroutine in many algorithms:
  - Searching in databases:
    we can do binary search on sorted data
  - Element uniqueness, duplicate elimination
  - A large number of computer graphics and computational geometry problems

Why Sorting?/2

- Sorting algorithms represent different algorithm design techniques.
- The lower bound for sorting of $\Omega(n \log n)$ is used to prove lower bounds of other problems.

Sorting Algorithms so Far

- Insertion sort, selection sort, bubble sort
  - Worst-case running time $\Theta(n^2)$
  - In-place
- Merge sort
  - Worst-case running time $\Theta(n \log n)$
  - Requires additional memory $\Theta(n)$

DSA, Chapter 4: Overview

- About sorting algorithms
- Heapsort
  - Complete binary trees
  - Heap data structure
- Quicksort
  - a popular algorithm
  - very fast on average

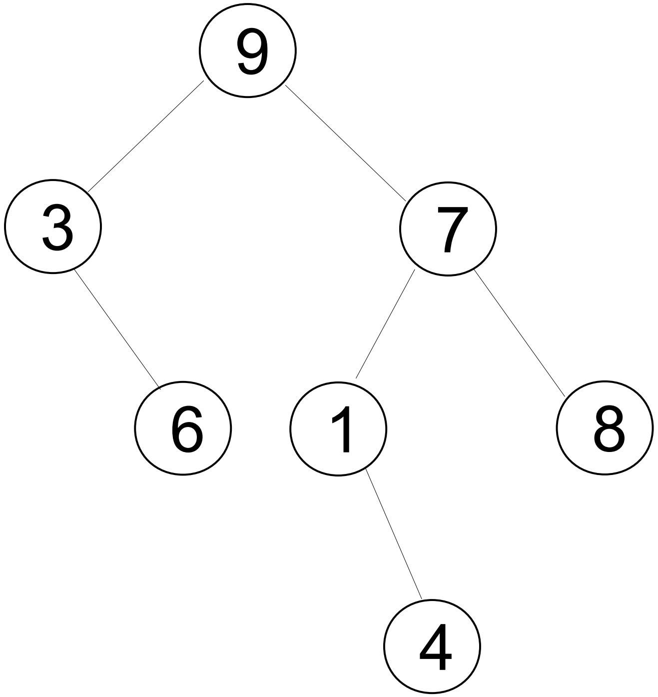
Selection Sort

```
SelectionSort(A[1..n]):
   for i := 1 to n-1
A:      Find the smallest element among A[i..n]
B:      Exchange it with A[i]
```

- A takes $\Theta(n)$ and B takes $\Theta(1)$: $\Theta(n^2)$ in total
- Idea for improvement: smart data structure to
  - do A and B in $\Theta(1)$
  - spend $O(\log n)$ time per iteration to maintain the data structure
  - get a total running time of $O(n \log n)$

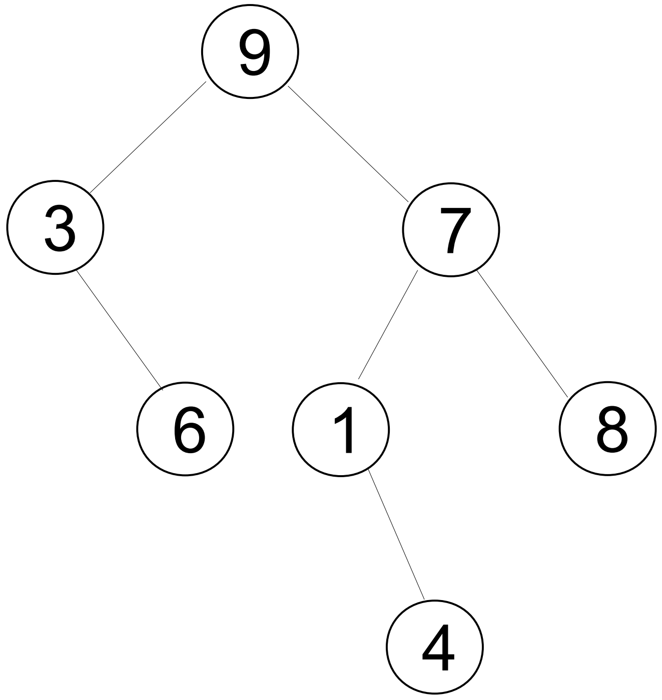
Binary Trees

- Each node may have a left and right **child**
  - The left child of 7 is 1
  - The right child of 7 is 8
  - 3 has no left child
  - 6 has no children
- Each node has at most one **parent**
  - 1 is the parent of 4
- The **root** has no parent
  - 9 is the root
- A **leaf** has no children
  - 6, 4 and 8 are leafs



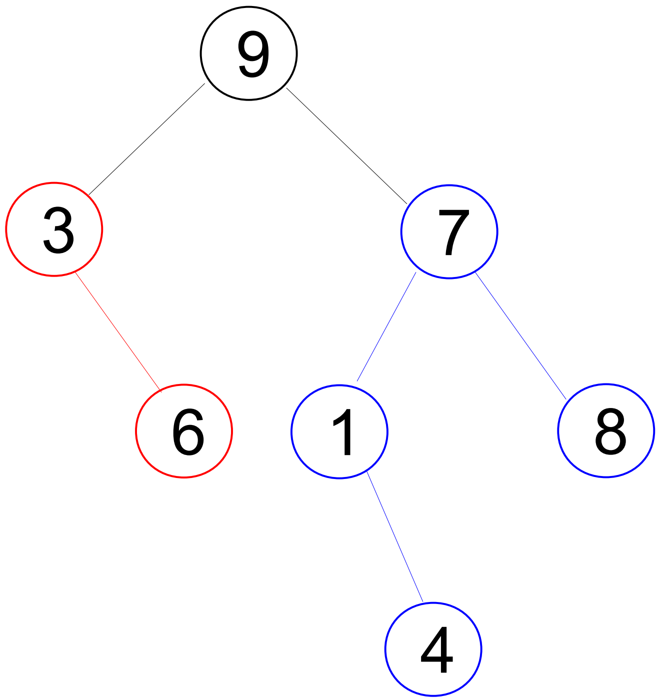
Binary Trees/2

- The **depth** (or **level**) of a node x is the length of the path from the root to x
  - The depth of 1 is 2
  - The depth of 9 is 0
- The **height** of a node x is the length of the longest path from x to a leaf
  - The height of 7 is 2
- The height of a tree is the height of its root
  - The height of the tree is 3



Binary Trees/3

- The right subtree of a node x is the tree rooted at the right child of x
  - The right subtree of 9 is the tree shown in blue
- The left subtree of a node x is the tree rooted at the left child of x
  - The left subtree of 9 is the tree shown in red



Complete Binary Trees

- A complete binary tree is a binary tree where
  - all leaves have the same depth
  - all internal (non-leaf) nodes have two children
- A nearly complete binary tree is a binary tree where
  - the depth of two leaves differs by at most 1
  - all leaves with the maximal depth are as far left as possible

Heaps

- A binary tree is a binary heap iff
  - it is a nearly complete binary tree
  - each node is greater than or equal to all its children
- The properties of a binary heap allow
  - an efficient storage as an array (because it is a nearly complete binary tree)
  - a fast sorting (because of the organization of the values)

Heaps/2

Heap property

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

Parent(i)

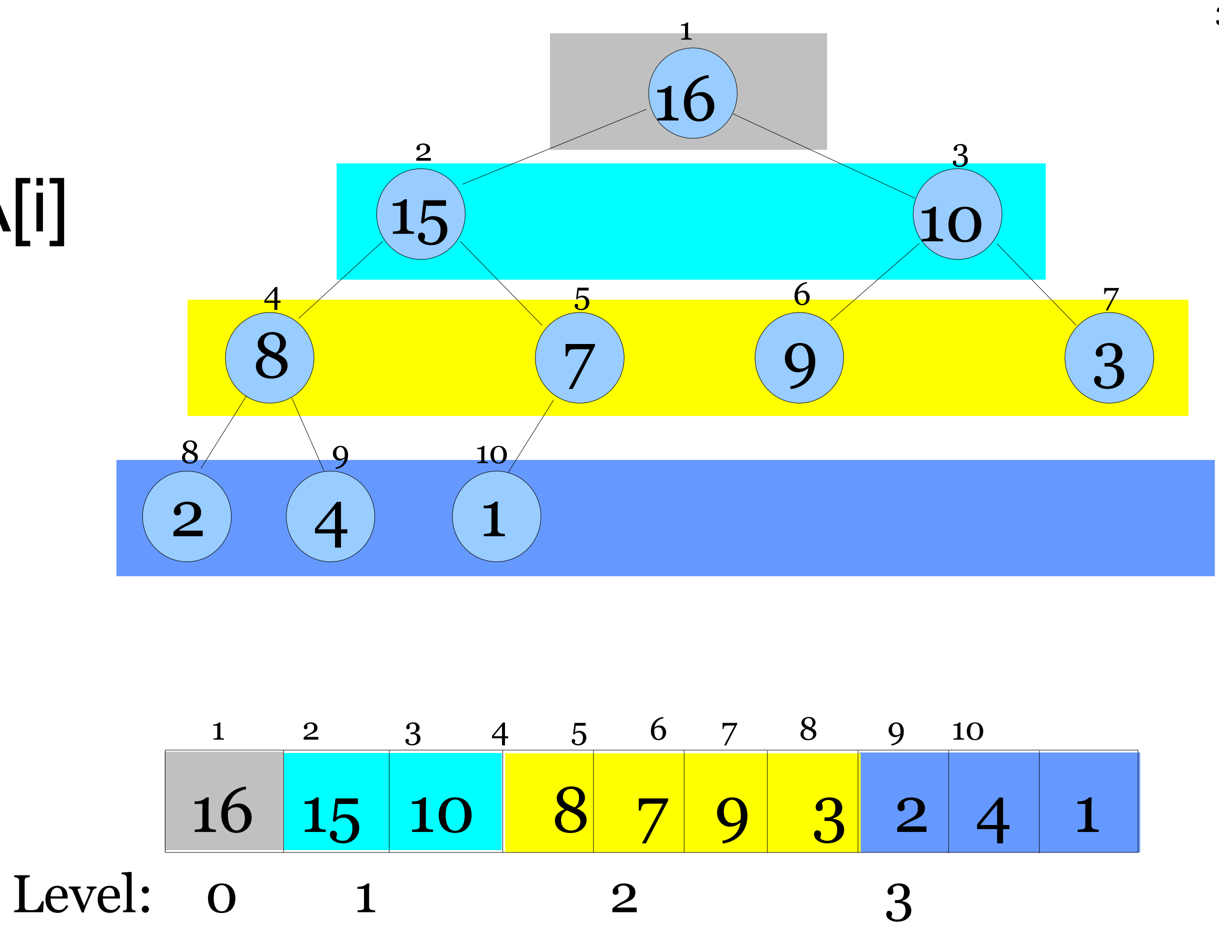
return $\lfloor i/2 \rfloor$

Left(i)

return 2i

Right(i)

return 2i+1



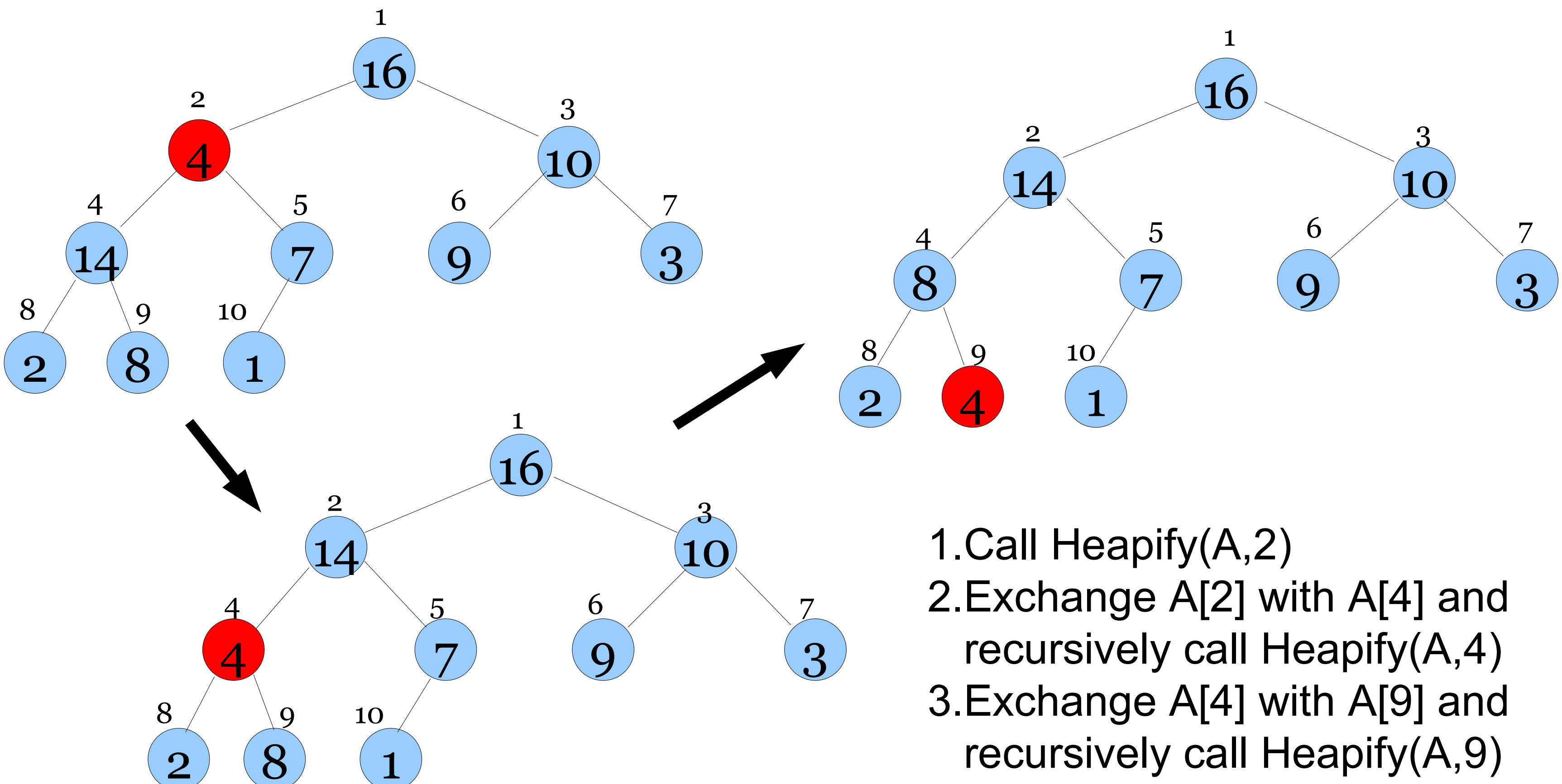
Heaps/3

- Notice the implicit tree links in the array: children of node i are $2i$ and $2i+1$
- The heap data structure can be used to implement a fast sorting algorithm.
- The basic elements are
  - **Heapify**: reconstructs a heap after an element was modified
  - **BuildHeap**: constructs a heap from an array
  - **HeapSort**: the sorting algorithm

Heapify

- Input: index *i* in array *A, number n of elements*
- Binary trees rooted at *Left(i)* and *Right(i)* are heaps
- *A[i]* might be smaller than its children, thus violating the heap property.
- **Heapify** makes *A* a heap by moving *A[i]* down the heap until the heap property is satisfied again.

Heapify Example



1. Call Heapify(A,2)
2. Exchange A[2] with A[4] and recursively call Heapify(A,4)
3. Exchange A[4] with A[9] and recursively call Heapify(A,9)
4. Node 9 has no children, so we are done

Heapify Algorithm

```
Heapify(A,i,n)
   l := 2*i;    // l := Left(i)
   r := 2*i+1; // r := Right(i)
   if l <= n and A[l] > A[i]
      then maxpos := l
      else maxpos := i
   if r <= n and A[r] > A[max]
      maxpos := r
   if max != i
      swap(A,i,maxpos)
      Heapify(A,maxpos,n)
```

Heapify: Running Time

The running time of Heapify on a subtree of size *n* rooted at *i* includes the time to

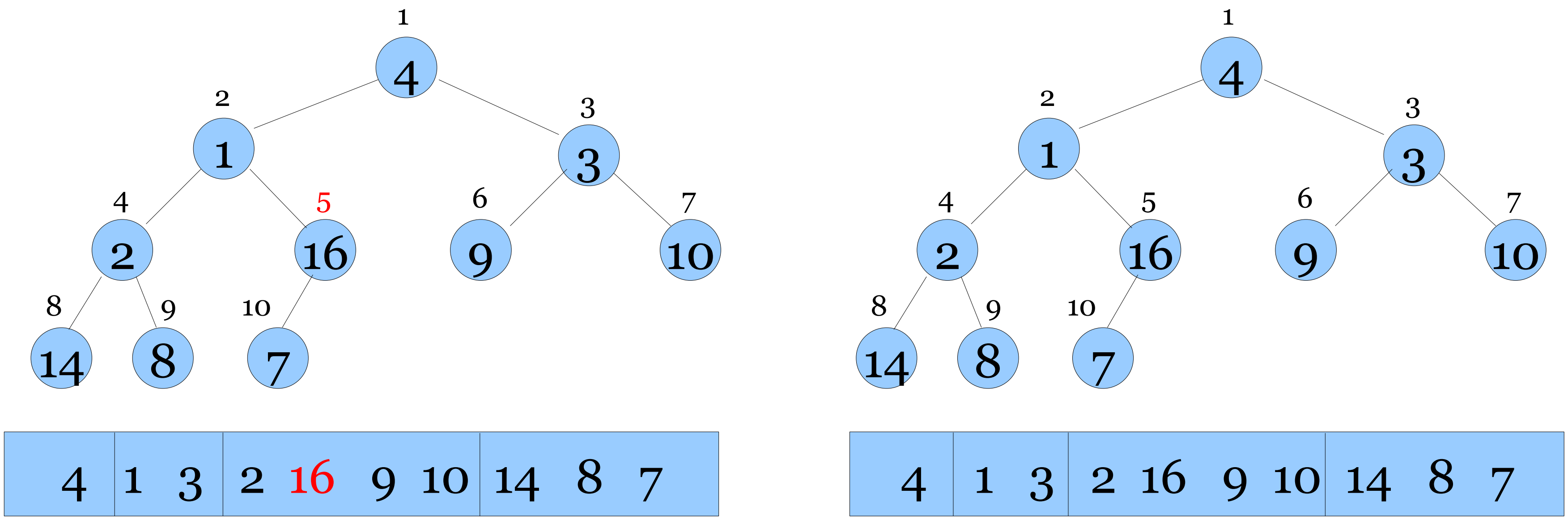
- determine relationship between elements: $\Theta(1)$
- run Heapify on a subtree rooted at one of the children of *i*
  - 2n/3 is the worst-case size of this subtree (half filled bottom level)
  - $T(n) \le T(2n/3) + \Theta(1)$ implies $T(n) = O(\log n)$
- Alternatively
  - Running time on a node of height *h*: $O(h) = O(\log n)$

Building a Heap

- Convert an array $A[1...n]$ into a heap
- Notice that the elements in the subarray $A[(\lfloor n/2 \rfloor + 1)...n]$ are 1-element heaps to begin with

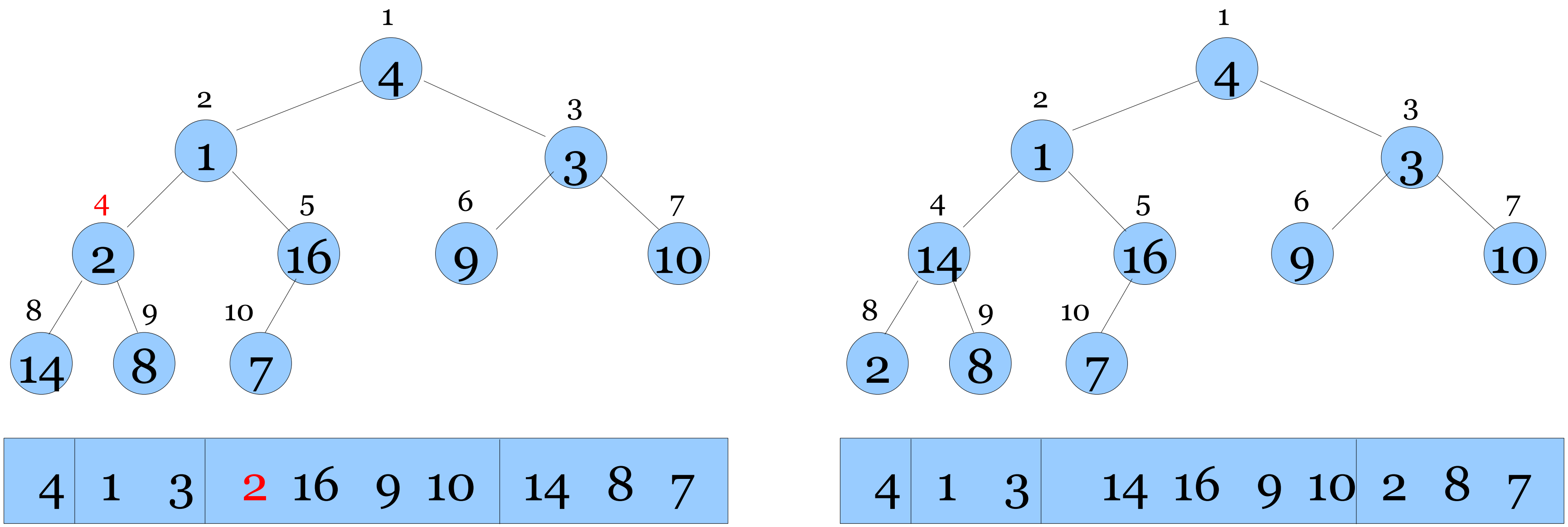
```
BuildHeap(A)
  for i := ⌊n/2⌋ to 1 do
    Heapify(A, i, n)
```

Building a Heap/2



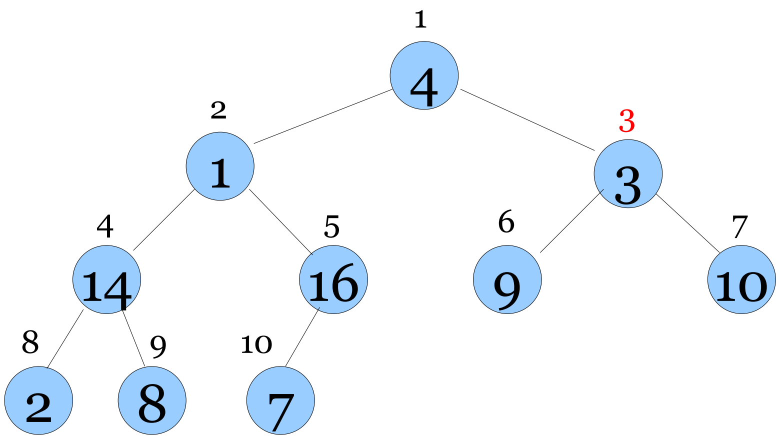
- `Heapify(A, 7, 10)`
- `Heapify(A, 6, 10)`
- `Heapify(A, 5, 10)`

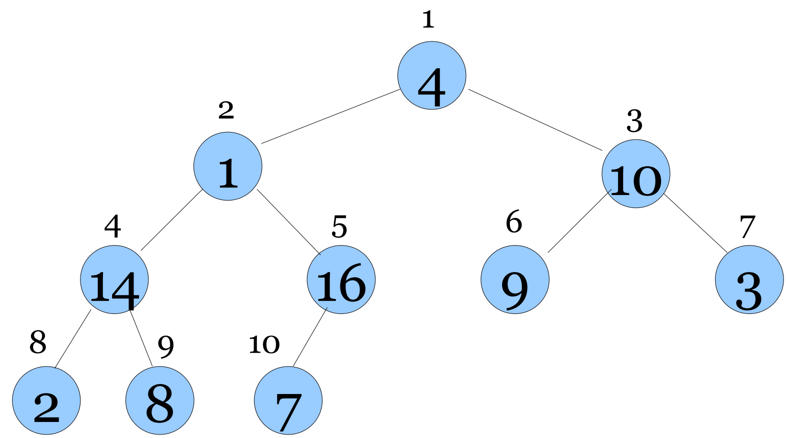
Building a Heap/3



- `Heapify(A, 4, 10)`

Building a Heap/4



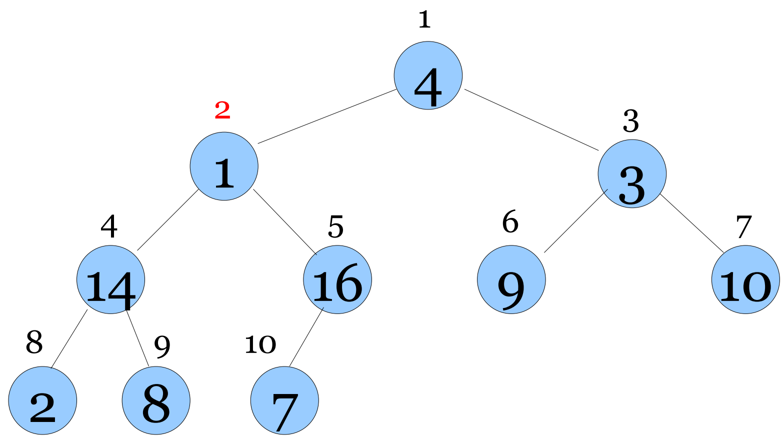


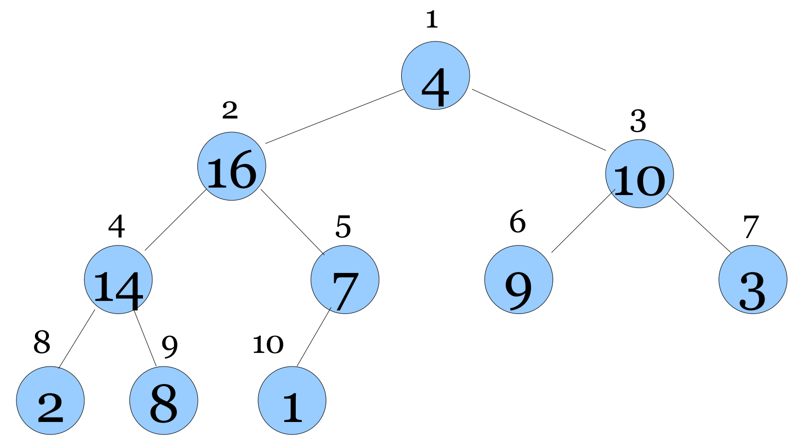
4	1	3	14	16	9	10	2	8	7

4	1	10	14	16	9	3	2	8	7

- `Heapify(A, 3, 10)`

Building a Heap/5



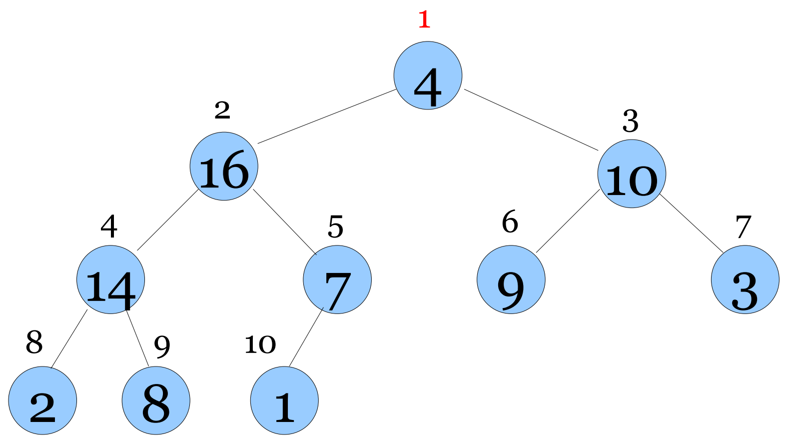


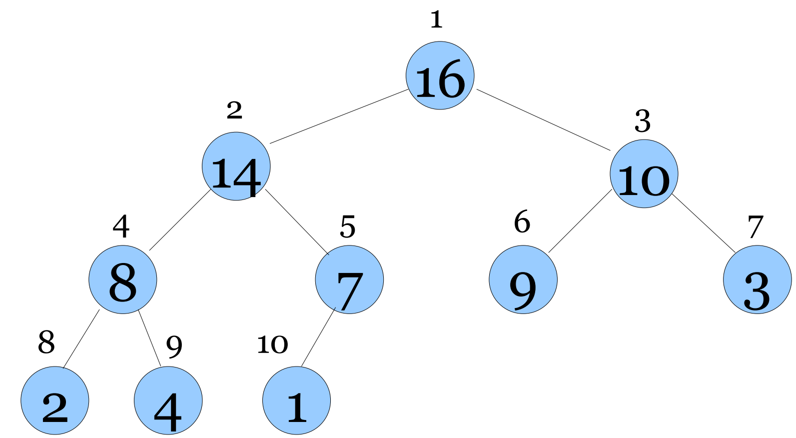
4	1	10	14	16	9	3	2	8	7

4	16	10	14	7	9	3	2	8	1

- `Heapify(A, 2, 10)`

Building a Heap/6





4	16	10	14	7	9	3	2	8	1

16	14	10	8	7	9	3	2	4	1

- `Heapify(A, 1, 10)`

Building a Heap: Analysis

- Correctness: induction on *i*,
  all trees rooted at $m > i$ are heaps
- Running time: *n* calls to Heapify
  $= n\ O(\log n) = O(n \log n)$
- Non-tight bound, but good enough for an overall $O(n \log n)$ bound for Heapsort.
- Intuition for a tight bound:
  - most of the time Heapify works on less than *n* element heaps

Building a Heap: Analysis/2

- Tight bound:
  - An n element heap has height log n.
  - The heap has $n/2^{h+1}$ nodes of height h.
  - Cost for one call of Heapify is O(h).

$$T(n) = \sum_{h=0}^{\log n} \frac{n}{2^{h+1}} O(h) = O(n \sum_{h=0}^{\log n} \frac{h}{2^h})$$

- Math:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \qquad \sum_{k=0}^{\infty} \frac{k}{x^k} = \sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$$

$$T(n) = O(n \sum_{h=0}^{\log n} \frac{h}{2^h}) = O(n \frac{1/2}{(1-1/2)^2}) = O(n)$$

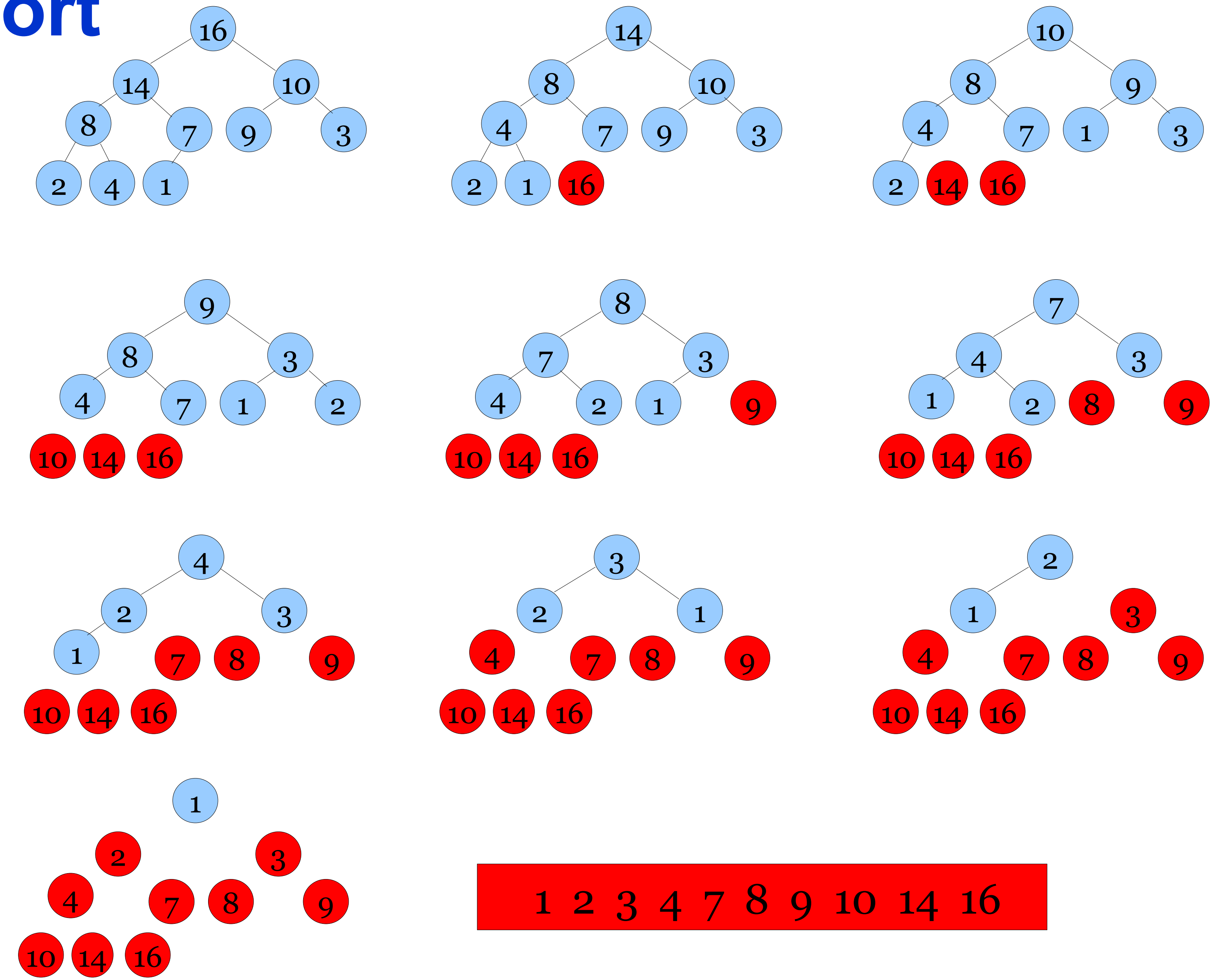
HeapSort

```
Heapsort(A)
  BuildHeap(A)                          O(n)
  for i := n to 2 do                    n times
   swap(A,1,i)                          O(1)
   n := n - 1                           O(1)
   Heapify(A, 1, n)                     O(log n)
```

The total running time of Heapsort is

*O(n) + n * O(log n) = O(n log n)*

Heapsort



Heapsort: Summary

- Heapsort uses a heap data structure to improve selection sort and make the running time asymptotically optimal
- Running time is $O(n \log n)$ – like Merge Sort, but unlike selection, insertion, or bubble sorts
- Sorts in place – like insertion, selection or bubble sorts, but unlike merge sort
- The heap data structure is also used for other things than sorting

DSA, Chapter 4: Overview

- About sorting algorithms
- Heapsort
  - Complete binary trees
  - Heap data structure
- Quicksort
  - a popular algorithm
  - very fast on average

Quicksort

Characteristics

- Sorts in place
  (like insertion sort, but unlike merge sort)
  i.e., does not require an additional array
- Very practical, average sort performance $O(n \log n)$
  (with small constant factors), but worst case $O(n^2)$

Quicksort: The Principle

When applying the Divide&Conquer principle to sorting, we obtain the following schema for an algorithm:

- Divide array segment A[l..r] into two subsegments, say A[l..m] and A[m+1,r]
- Conquer: sort each subsegment by a recursive call
- Combine the sorted subsegments into a sorted version of the original segment A[l..r]

Merge Sort takes an extreme approach in that

- no work is spent on the division
- a lot of work is spent on the combination

What does an algorithm look like where no work is spent on the combination?

Quicksort: The Principle/2

If no work is spent on the combination of the sorted segments, then, after the recursive call,

- all elements in the left subsegment A[l..m] must be ≤ all elements in the right subsegment A[m+1..r]

However, the recursive call can only have sorted the segments!

We conclude that the division must have partitioned A[l..r] into

- a subsegment with small elements A[l..m]
- a subsegment with big elements A[m+1..r]

How can we write code for such a partition algorithm?

Quicksort – the Principle

- To understand quick sort, let's look at a high-level description of the algorithm.
- A divide-and-conquer algorithm
  - **Divide**: partition array into 2 subarrays such that elements in the lower part ≤ elements in the higher part.
  - **Conquer**: recursively sort the 2 subarrays
  - **Combine**: trivial since sorting is done in place

Quick Sort Algorithm: Overview

```
INPUT: A[1..n] - an array of integers
       l,r - integers satisfying 1≤l≤r≤n
OUTPUT: permutation of the segment A[l..r] s.t.
        A[l]≤A[l+1]≤...≤A[r]

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

Partition divides the segment A[l..r] into

- a segment of "little elements" A[l..m-1]
- a segment of "big elements" A[m+1..r],

with A[m] in the middle between the two

Partition

```
INPUT: A[1..n] – an array of integers
       l,r – integers satisfying 1 ≤ l < r ≤ n
OUTPUT: m – an integer with l ≤ m ≤ r
        a permutation of A[l..r] such that
        A[i]< A[m]  for all i with l ≤ i < m
        A[m]≤ A[i]  for all i with m < i ≤ r

int Partition(A,l,r)
    p  := A[m]; // pivot, used for the split
    ll := l-1;  // last of the little ones
    for bu := l to r-1 do
           // bu is the beginning of the unknown area
        if A[bu] < p
           then swap(A,ll+1,bu); ll++;
           // all elements < p are little ones
    swap(A,ll+1,m)
           // move the pivot into the middle position
```

Partition: Loop Invariant

This version of partition has the following loop invariant

- $A[i] < p$, for all i with $l \leq i \leq ll$
  (all little ones are $< p$)
- $A[i] \geq p$ for all I with $ll < i < bu$
  (all big ones are $\geq p$).

Clearly,

- this holds at the beginning of the execution
- this is maintained during the loop
- the loop terminates.

At the end of the loop, A[l..ll] comprises the little ones, and A[ll+1..r-1] comprises the big ones.
Since p = A[r] is a big one, the postcondition holds after the swap of A[ll+1] and A[p].

Partitioning from the Endpoints

There is another approach to partitioning, due to Tony Hoare, the inventor of Quicksort.
As before, we choose p:=A[r] as the pivot.
Then repeatedly,

- walk from right to left until you find an element $\leq$ p
- walk from left to right until you find an element $\geq$ p
- swap those elements.

Note that in this approach, we have no control where p ends up.
Therefore, Partition returns an index m such that

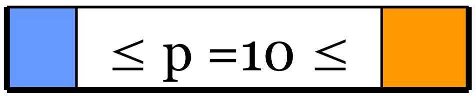
A[i] <= A[j], for all i, j with $l \leq i \leq m$ and $m+1 \leq j \leq r$

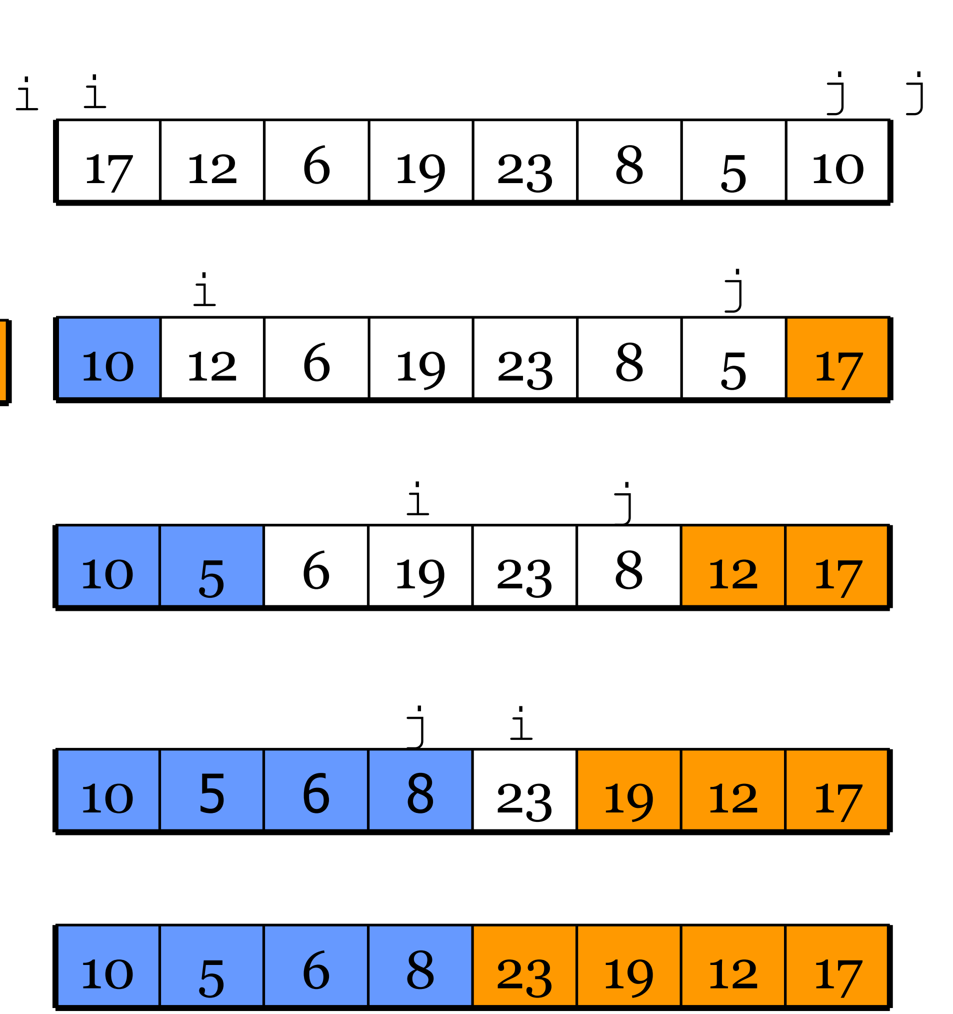
Consequently, Quicksort(A,l,r) launches two recursive calls

Quicksort(A,l,m) and Quicksort(A,m+1,r)

Partitioning from the Endpoints/2

```
int Partition(A,l,r)
p := A[r]
i := l-1
j := r+1
while TRUE
  repeat j := j-1
    until A[j] ≤ p
  repeat i := i+1
    until A[i] ≥ p
  if i<j
    then swap(A,i,j)
    else return i
```





Quicksort with Partitioning from the Endpoints

```
INPUT: A[1..n] – an array of integers
       l,r – integers satisfying 1≤l≤r≤n
OUTPUT: permutation of the segment A[l..r] s.t.
        A[l]≤A[l+1]≤...≤A[r]

Quicksort(A,l,r)
     if l < r then
        m := Partition(A,l,r)
        Quicksort(A,l,m)
        Quicksort(A,m+1,r)
```

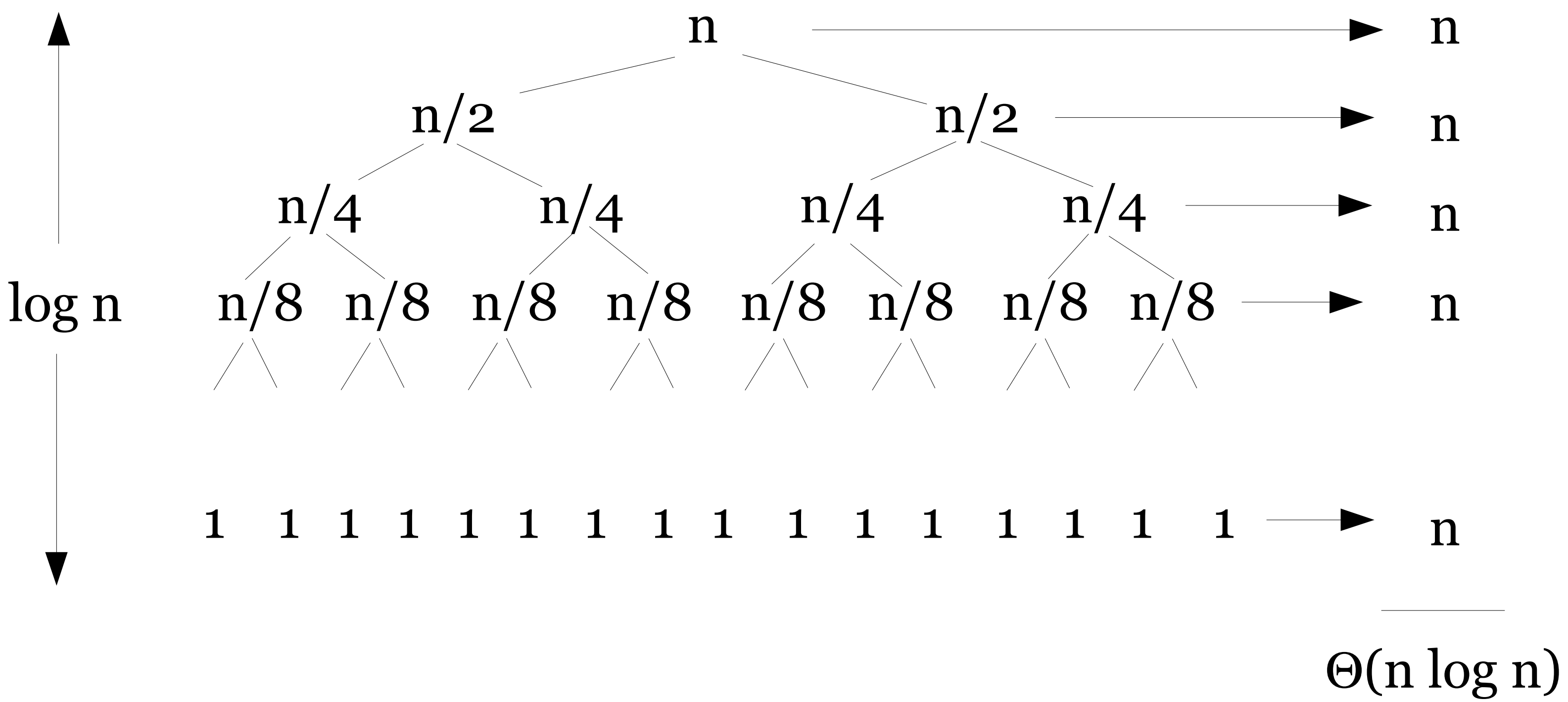
- Note the different parameters of the first recursive call!

Analysis of Quicksort

- Assume that all input elements are distinct
- The running time depends on the distribution of splits

Best Case

- If we are lucky, Partition splits the array evenly:
  $T(n) = 2\ T(n/2) + \Theta(n)$



Worst Case

- What is the worst case?
- One side of the partition has one element|
- T(n) = T(n-1) + T(1) + Θ(n)

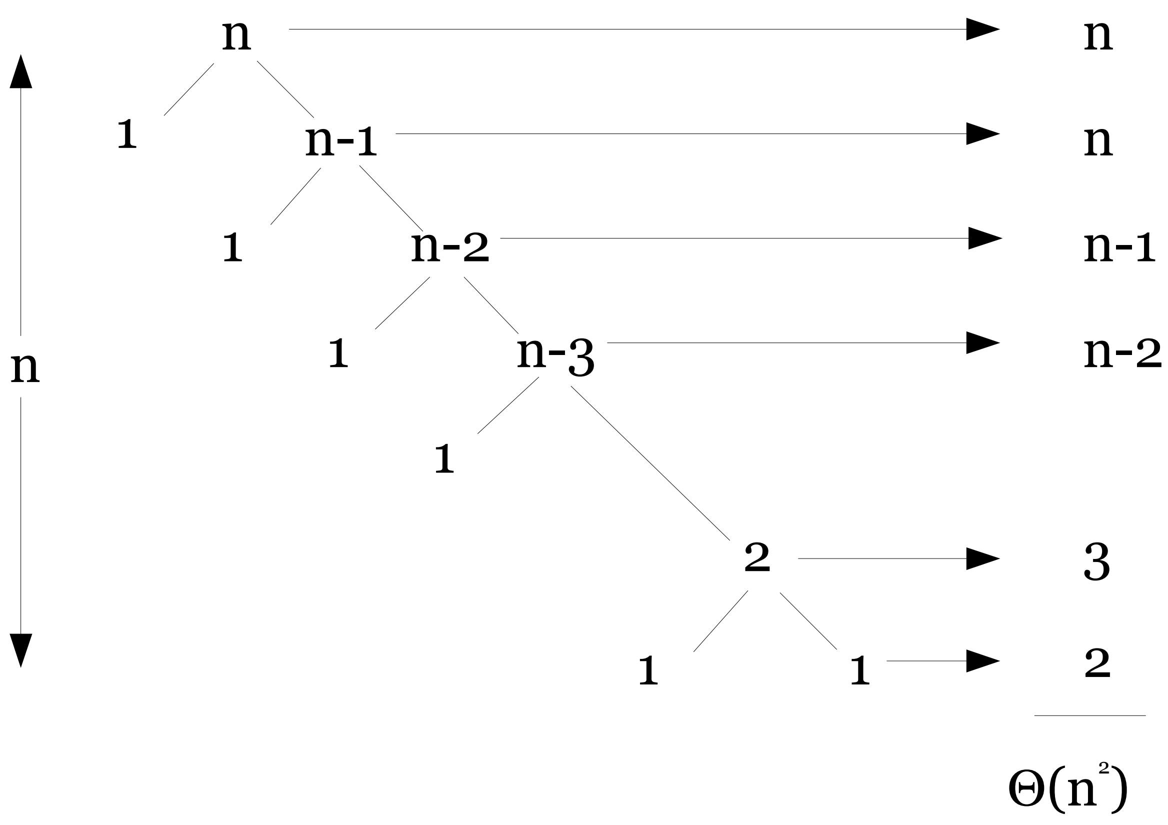
$$= T(n-1) + 0 + \Theta(n)$$

$$= \sum_{k=1}^{n} \Theta(k)$$

$$= \Theta(\sum_{k=1}^{n} k)$$

$$= \Theta(n^2)$$

Worst Case/2



Worst Case/3

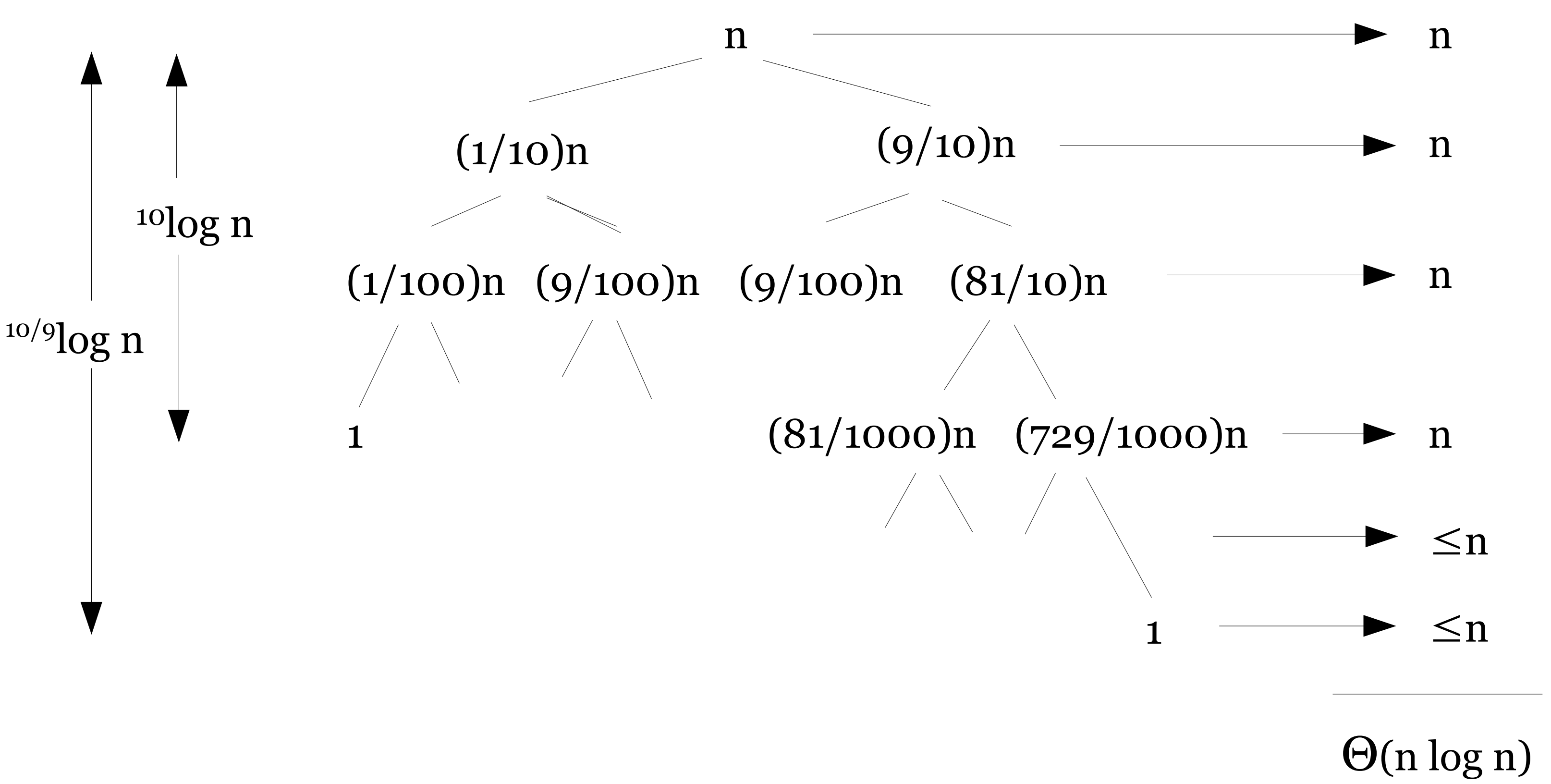
- When does the worst case appear?
  => one of the partition segments is empty
  – input is sorted
  – input is reversely sorted

- Similar to the worst case of Insertion Sort
  (reverse order, all elements have to be moved)

- But sorted input yields the best case for insertion sort

Analysis of Quicksort

Suppose the split is 1/10 : 9/10



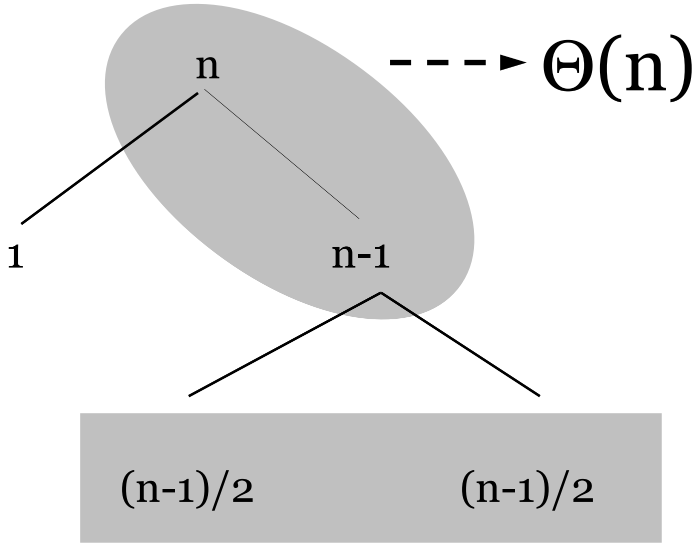
An Average Case Scenario

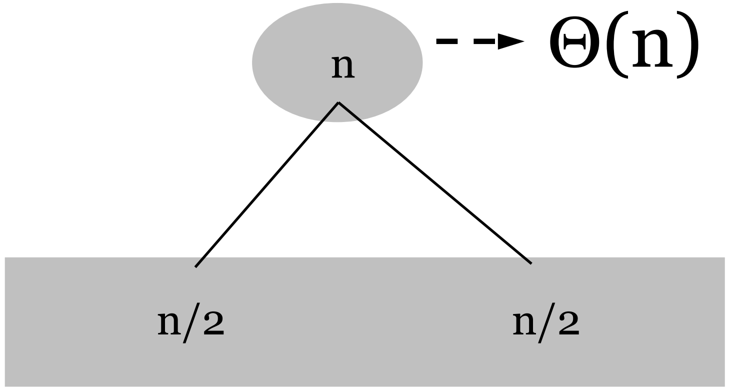
Suppose, we alternate lucky and unlucky cases to get an average behavior

$L(n) = 2U(n/2) + \Theta(n)$ lucky
$U(n) = L(n-1) + \Theta(n)$ unlucky

we consequently get

$$
\begin{aligned}
L(n) &= 2(L(n/2 - 1) + \Theta(n)) + \Theta(n) \\
&= 2L(n/2 - 1) + \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}
$$





An Average Case Scenario/2

- How can we make sure that we are usually lucky?
  - Partition around the “middle” (n/2th) element?
  - Partition around a random element (works well in practice)
- Randomized algorithm
  - running time is independent of the input ordering
  - no specific input triggers worst-case behavior
  - the worst-case is only determined by the output of the random-number generator

Randomized Quicksort

- Assume all elements are distinct
- Partition around a random element
- Consequently, all splits

  1:n-1,
  2:n-2,
  ...,
  n-1:1

  are equally likely with probability 1/n.

- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.

Randomized Quicksort/2

```
int RandomizedPartition(A,l,r)
   i := Random(l,r)
   exchange A[r] and A[i]
   return Partition(A,l,r)

RandomizedQuicksort(A,l,r)
   if l < r then
      m := RandomizedPartition(A,l,r)
      RandomizedQuicksort(A,l,m-1)
      RandomizedQuicksort(A,m+1,r)
```

Summary

- Heapsort
  - same idea as Max sort, but heap data structure helps to find the maximum quickly
  - a heap is a nearly complete binary tree, which here is implemented in an array
  - worst case is n log n
- Quicksort
  - partition-based: extreme case of D&C, no work is spent on combining results
  - popular, behind Unix ”sort” command
  - very fast on average
  - worst case performance is quadratic

Comparison of Sorting Algorithms

- Running time in seconds, n=2048
- Absolute values are not important; compare values with each other
- Relate values to asymptotic running time (n log n, n^2)

	ordered	random	inverse
Insertion	0.22	50.74	103.8
Selection	58.18	58.34	73.46
Bubble	80.18	128.84	178.66
Heap	2.32	2.22	2.12
Quick	0.72	1.22	0.76

Next Chapter

- Dynamic data structures
  - Pointers
  - Lists, trees
- Abstract data types (ADTs)
  - Definition of ADTs
  - Common ADTs