Data Structures and Algorithms

Chapter 4

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

Previous Chapter

- Divide and conquer
- Merge sort
- Tiling
- Recurrences
  - repeated substitutions
  - substitution
  - master method
- Example recurrences

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 $\Omega(n \log n)$ 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 $\Theta(n)$

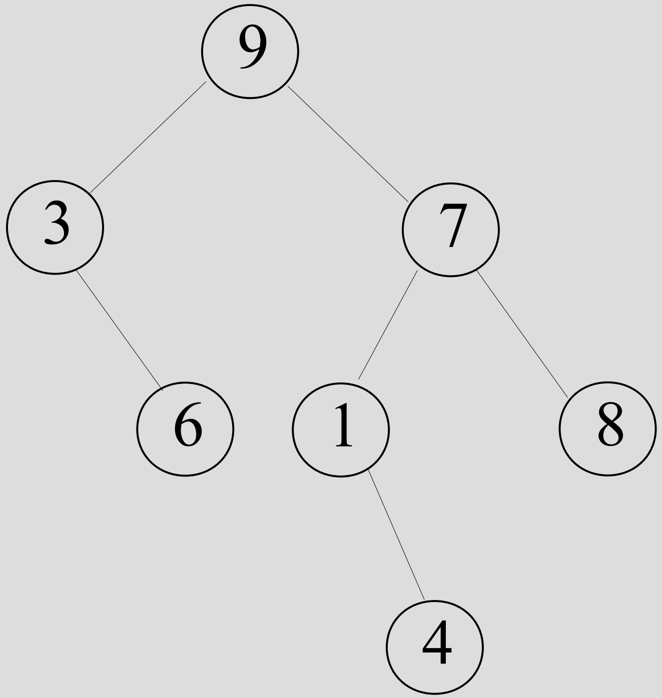
Selection Sort

```
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 $O(n \log n)$

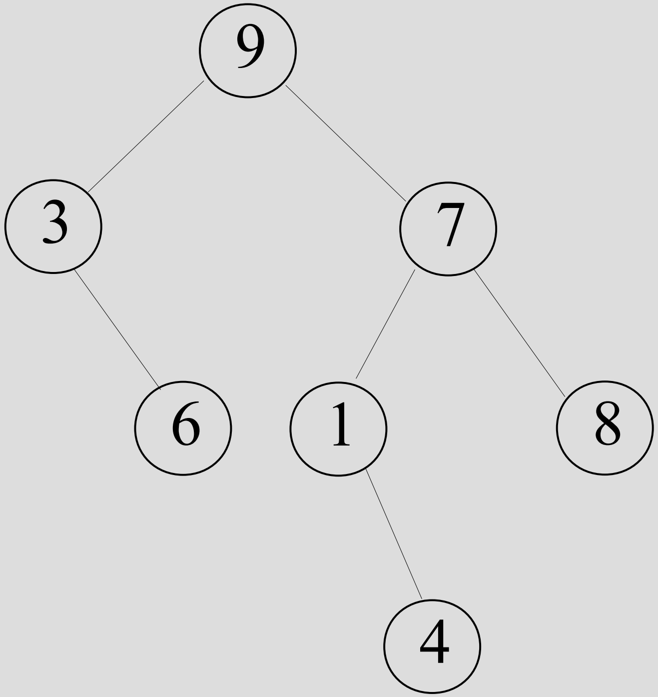
Binary Trees

- 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



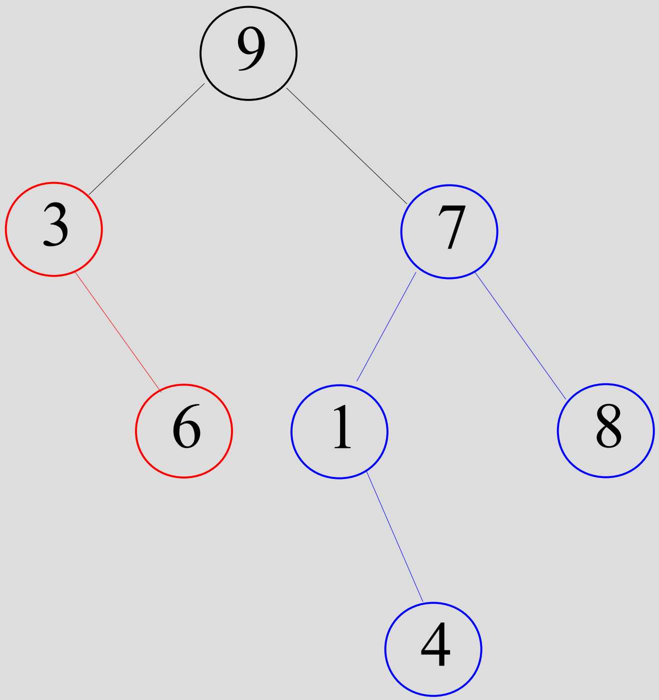
Binary Trees/2

- 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)

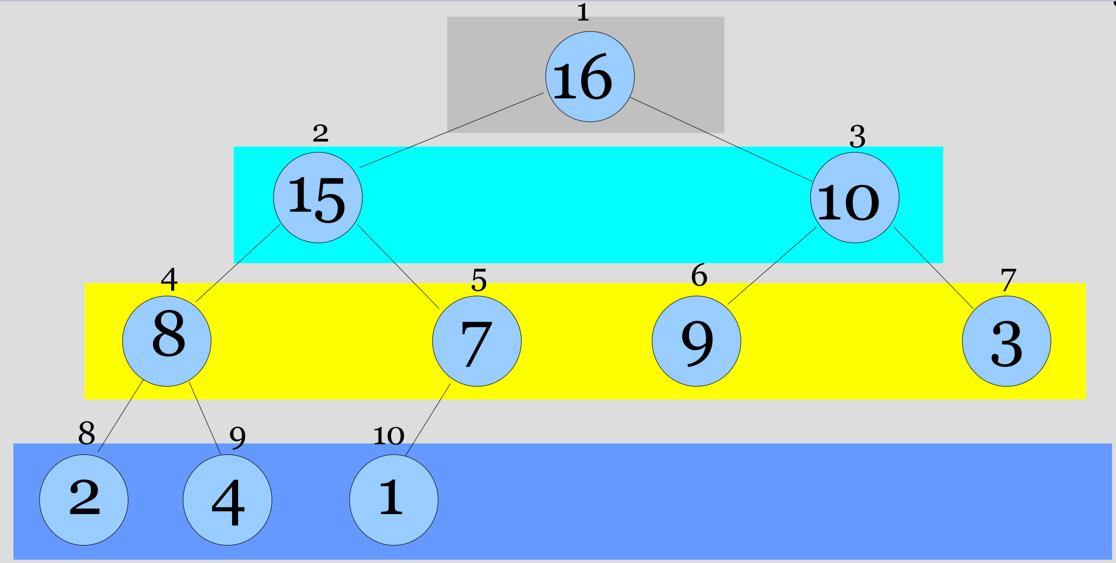
Heaps/2

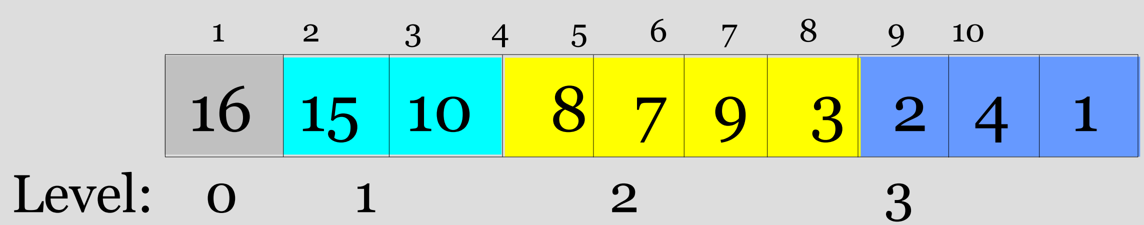
Heap property
  $A[\mathrm{Parent}(i)] \geq A[i]$

Parent(i)
  return $\lfloor i/2 \rfloor$

Left(i)
  return 2i

Right(i)
  return 2i+1





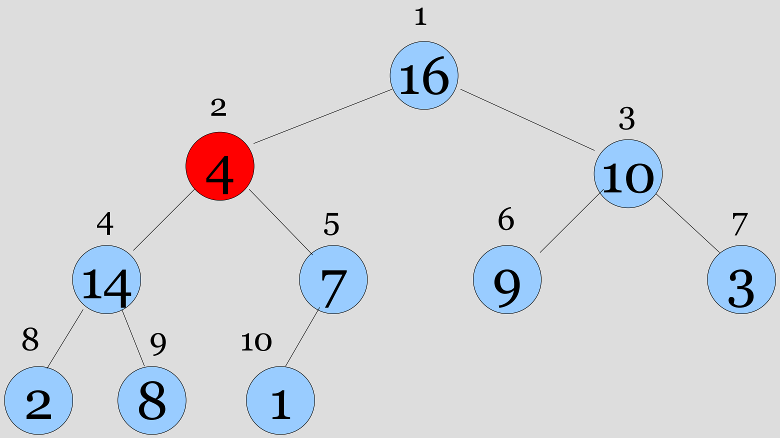
Heaps/3

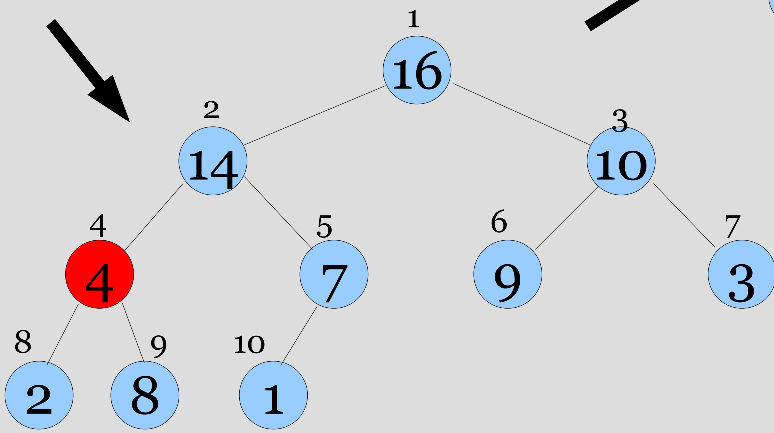
- Notice the implicit tree links in the array: children of node i are $2i$ and $2i+1$
- The heap data structure can be used to implement a fast sorting algorithm.
- The basic elements are
  – **Heapify**: reconstructs a heap after an element was modified
  – **BuildHeap**: constructs a heap from an array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

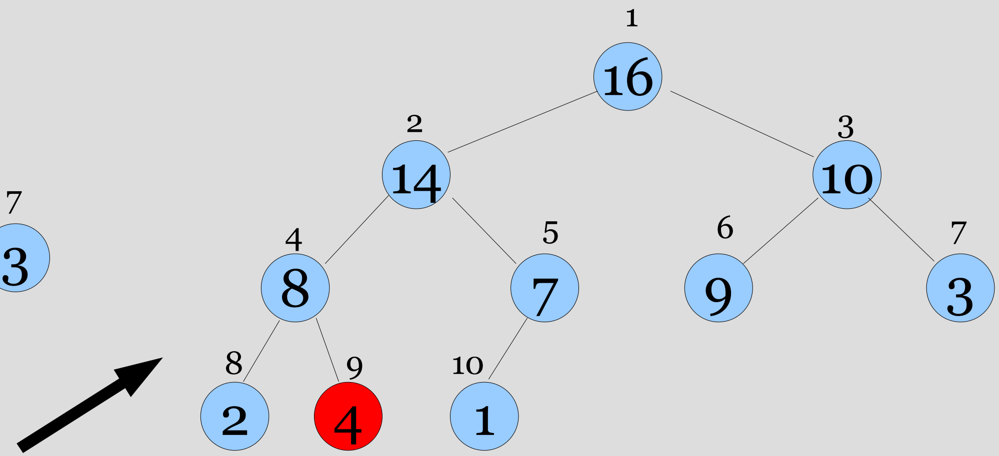
Heapify

- 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 max := l
    else max := i
  if r <= n and A[r] > A[max]
    max := r
  if max != i
    exchange A[i] and A[max]
    Heapify(A, max, n)
```

Heapify: Running Time

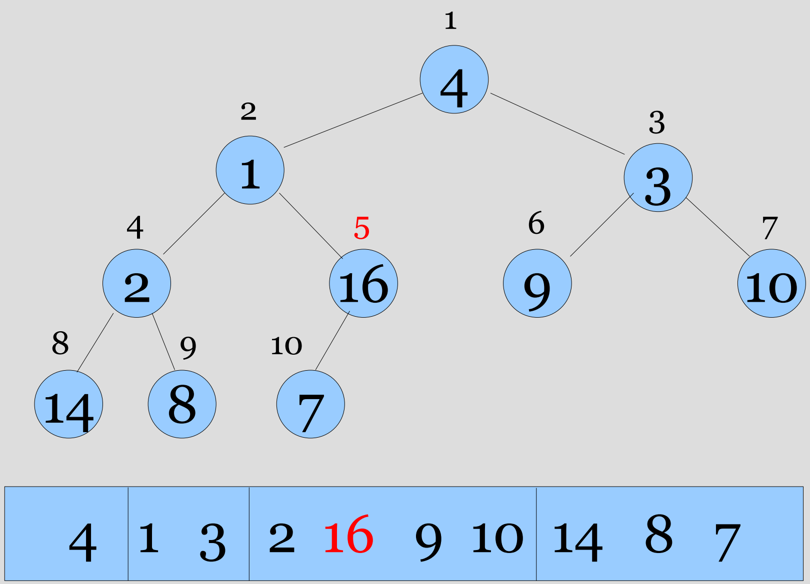
- The running time of Heapify on a subtree of size *n* rooted at *i* includes the time to
  - determine relationship between elements: $\Theta(1)$
  - run Heapify on a subtree rooted at one of the children of *i*
    - 2n/3 is the worst-case size of this subtree (half filled bottom level)
    - $T(n)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

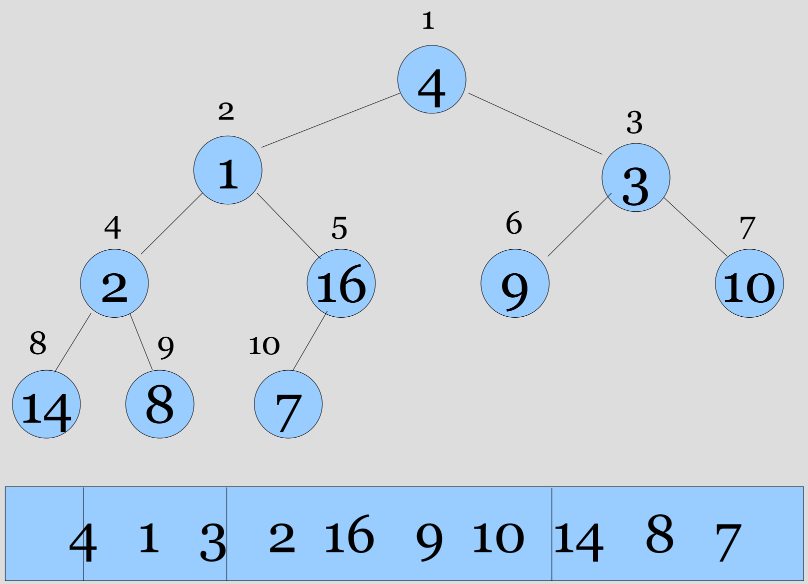
Building a Heap

- 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 := $\lfloor n/2 \rfloor$ **to** 1 **do**
  Heapify(A, i, n)

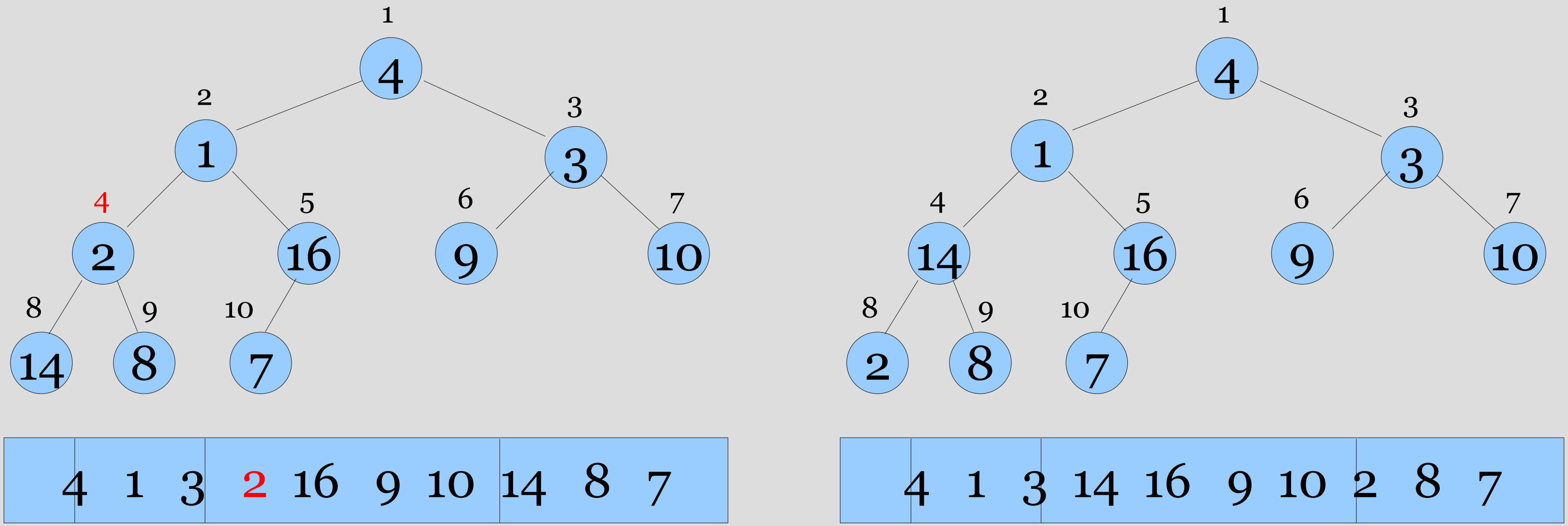
Building a Heap/2





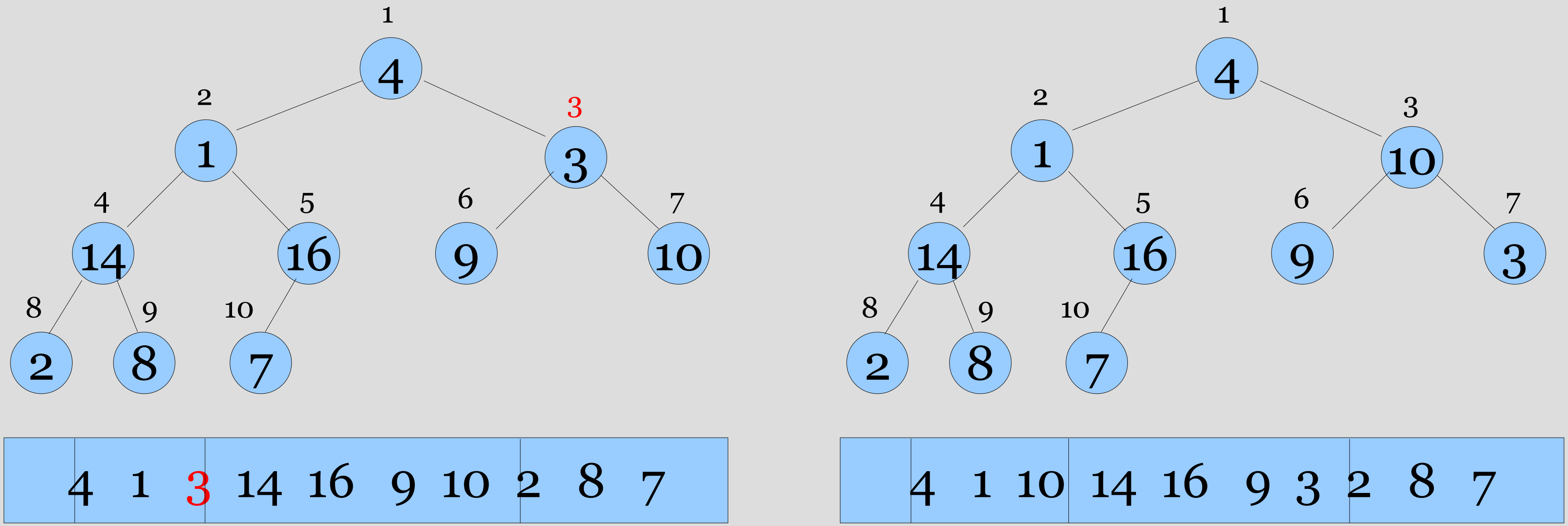
- `Heapify(A, 7, 10)`
- `Heapify(A, 6, 10)`
- `Heapify(A, 5, 10)`

Building a Heap/3



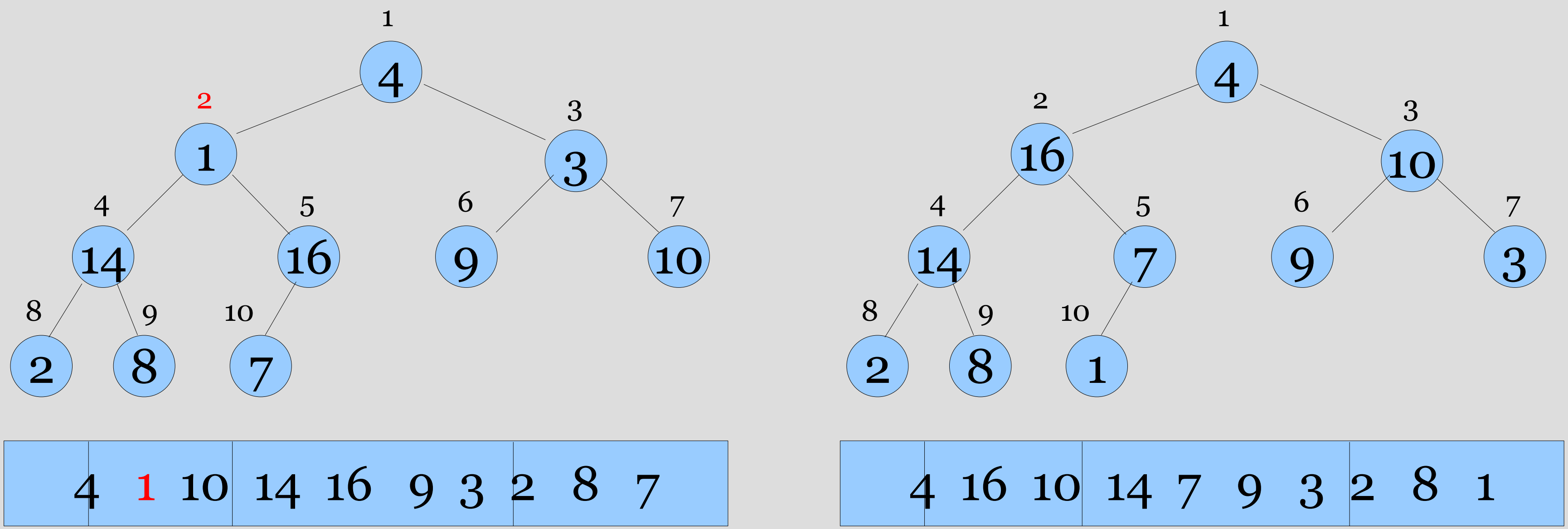
- `Heapify(A, 4, 10)`

Building a Heap/4



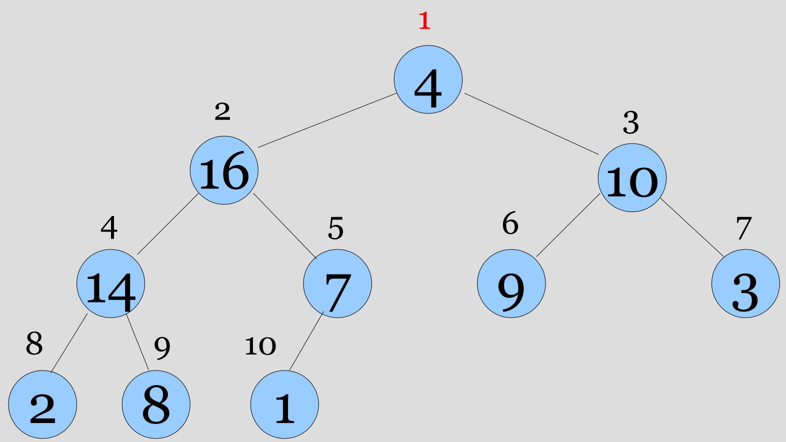
- `Heapify(A, 3, 10)`

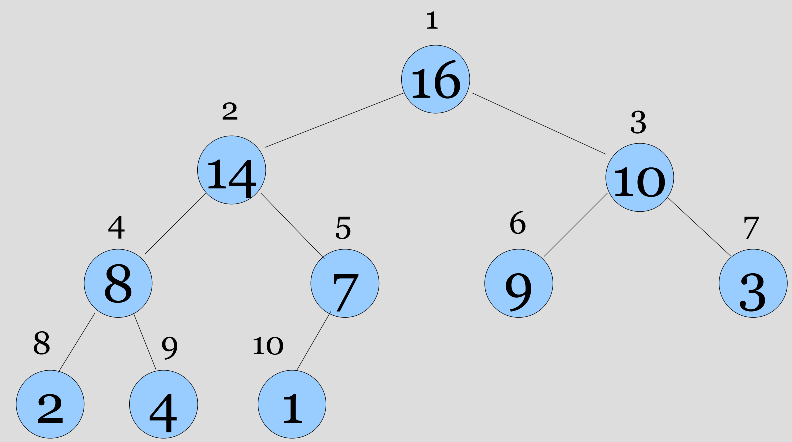
Building a Heap/5



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

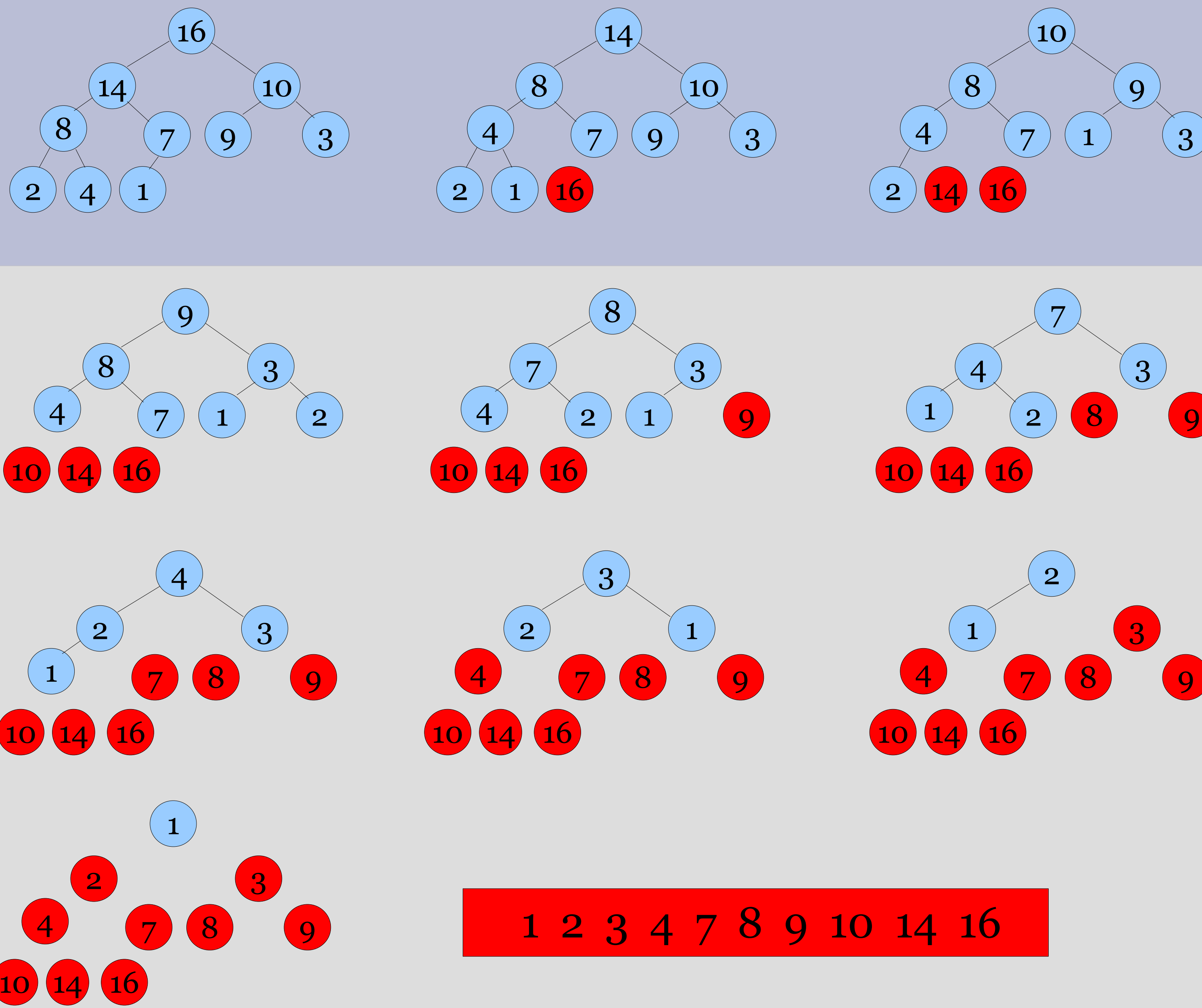
- $$T(n) = O(n \sum_{h=0}^{\log n} \frac{h}{2^h}) = O(n \frac{1/2}{(1-1/2)^2}) = O(n)$$

HeapSort

- The total running time of heap sort is $O(n) + n * O(\log n) = O(n \log n)$

```
HeapSort(A)
  BuildHeap(A)                          O(n)
  for i := n to 2 do                    n times
    exchange A[1] and A[i]                O(1)
    n := n-1                              O(1)
    Heapify(A, 1, n)                      O(log n)
```

Heap Sort



Heap Sort: Summary

- Heap sort 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 used for other things than sorting.

Quick Sort

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

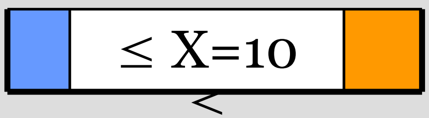
Quick Sort – the Principle

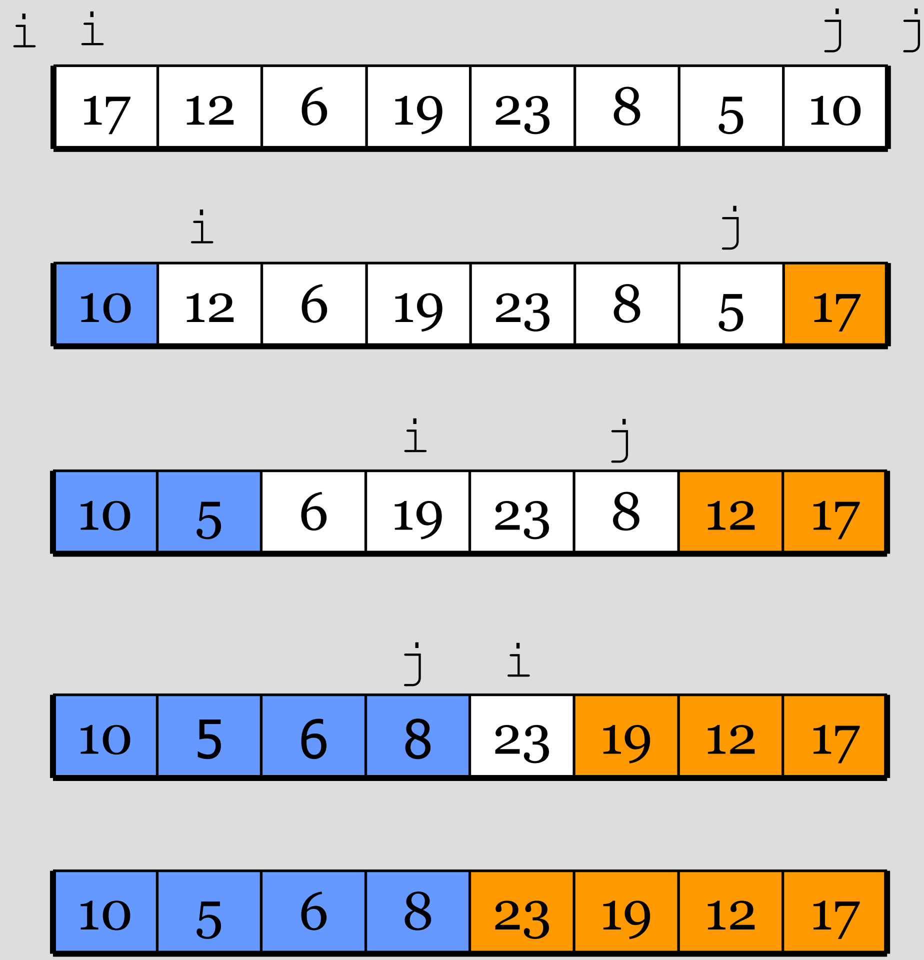
- To understand quick sort, let’s look at a high-level description of the algorithm.
- A divide-and-conquer algorithm
  - **Divide**: partition array into 2 subarrays such that elements in the lower part ≤ elements in the higher part.
  - **Conquer**: recursively sort the 2 subarrays
  - **Combine**: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

Partitioning

6

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





Quick Sort Algorithm

3 2 9

- Initial call **Quicksort(A, 1, n)**

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

Alternate Formulation of Quicksort (Lomuto)

First difference: we do not touch the middle element in the recursion

Quicksort(A, l, r)

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

Alternate Formulation of Quicksort /2

Second difference: Partioning proceeds from left to right

Partition(A, l, r)

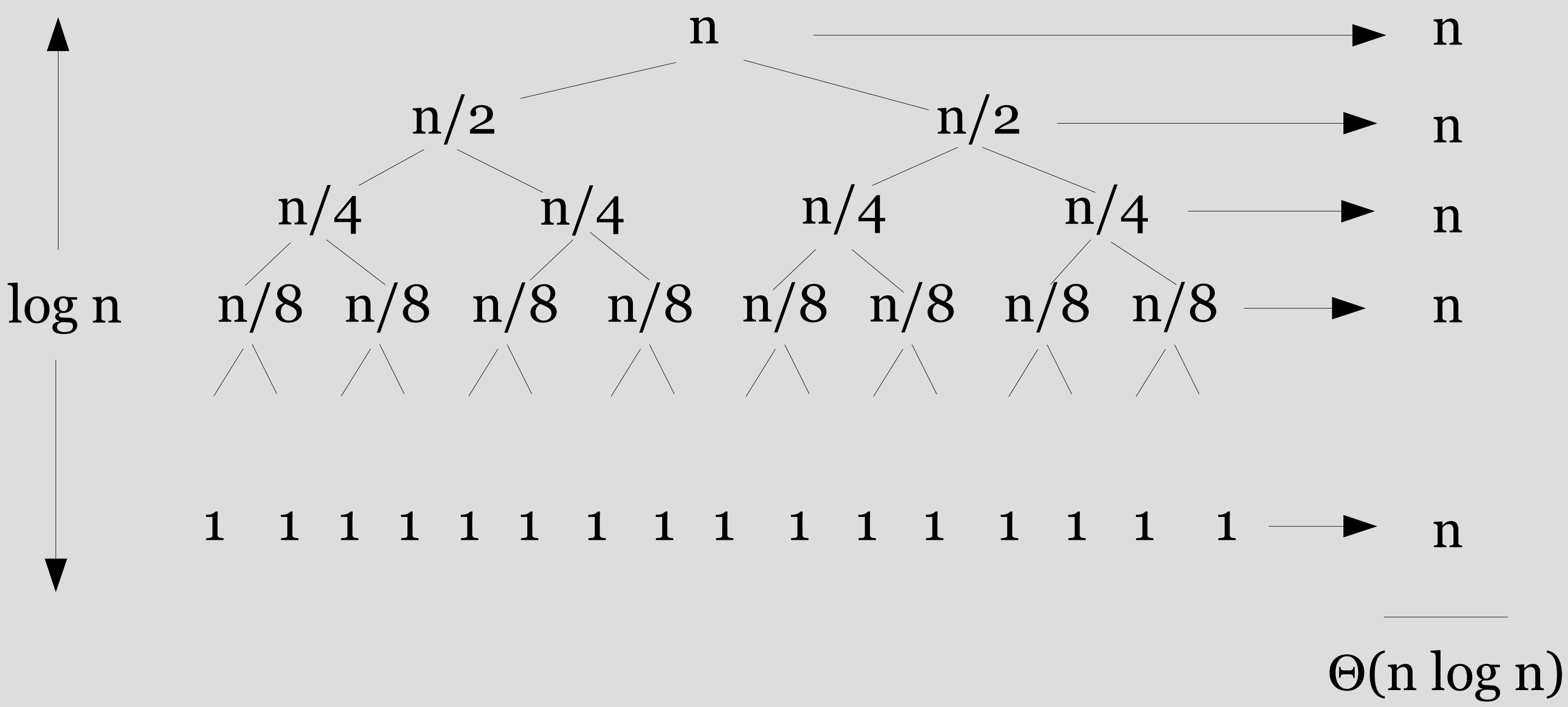
```
x := A[r]; ll := l-1;
for fu:=l to r-1 do
    if A[fu] <= x
    then Swap(A, ll+1, fu);
         ll++;
m := ll + 1;
Swap(A, m, r);     % put x into the middle
return m
```

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

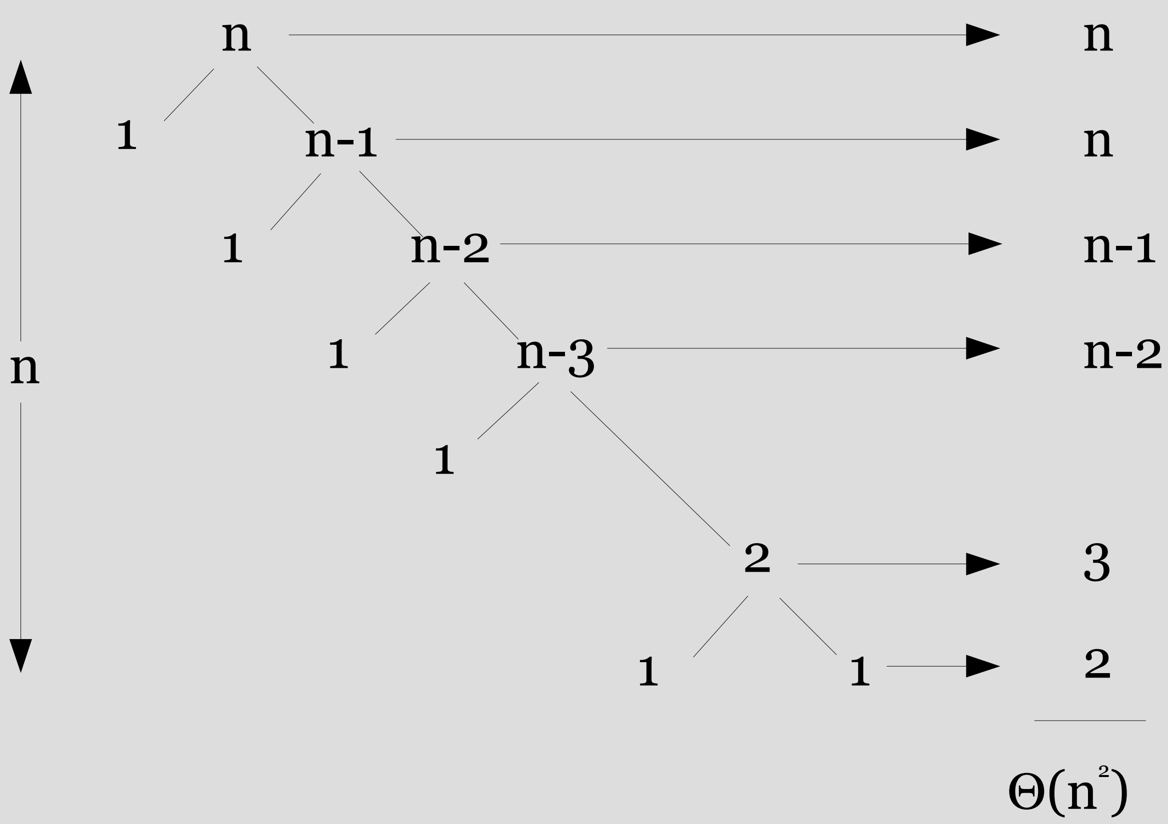
$$= T(n-1) + 0 + \Theta(n)$$

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

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

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

Worst Case/2

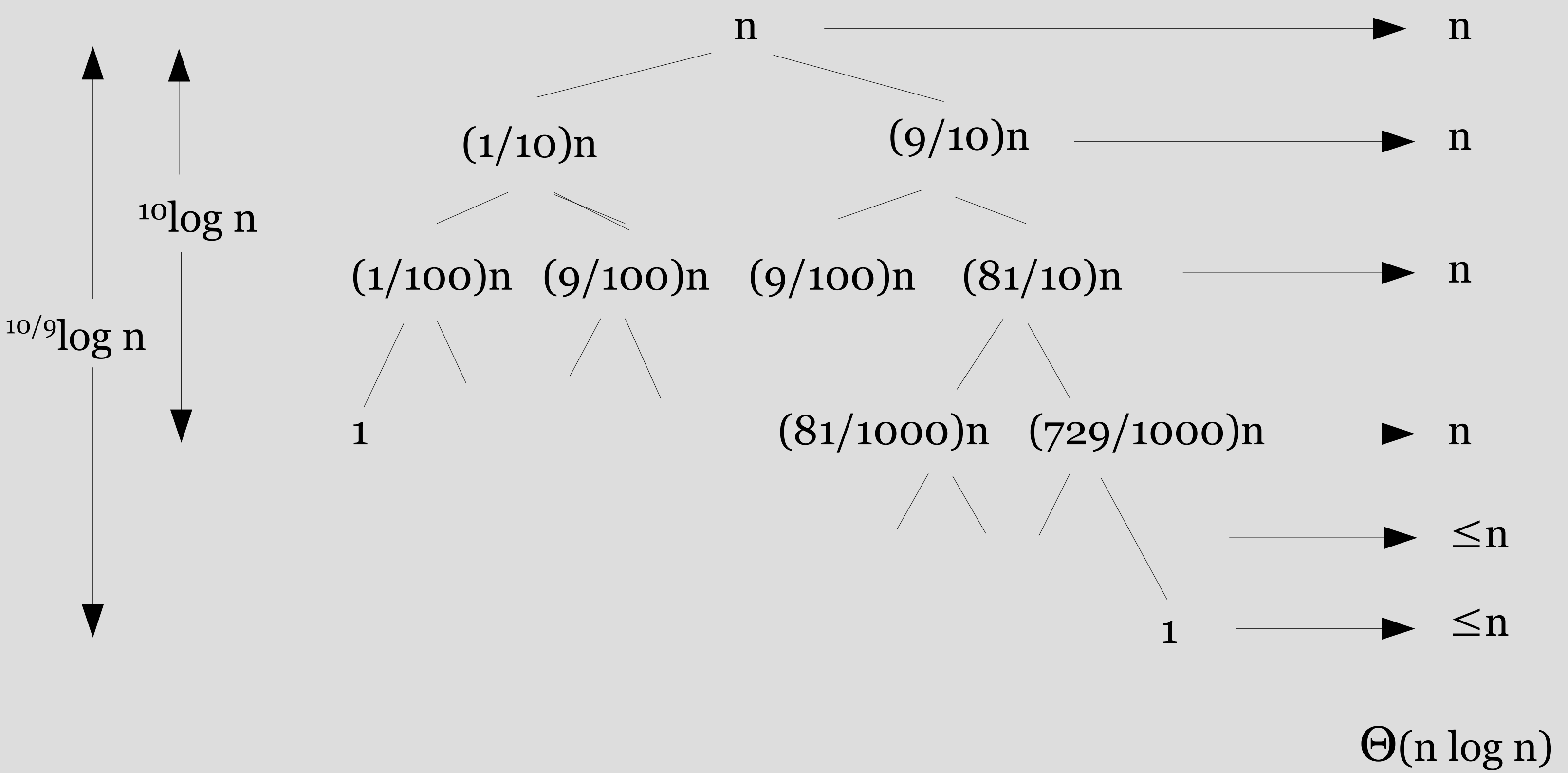


Worst Case/3

- When does the worst case appear?
  - input is sorted
  - input reverse sorted
- Same recurrence for the worst case of insertion sort (reverse order, all elements have to be moved).
- Sorted input yields the best case for insertion sort.

Analysis of Quicksort

- Suppose the split is 1/10 : 9/10



An Average Case Scenario

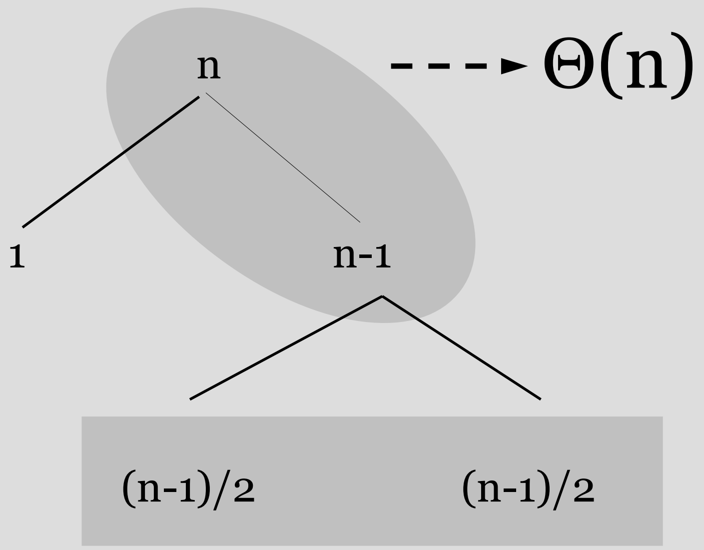
- Suppose, we alternate lucky and unlucky cases to get an average behavior

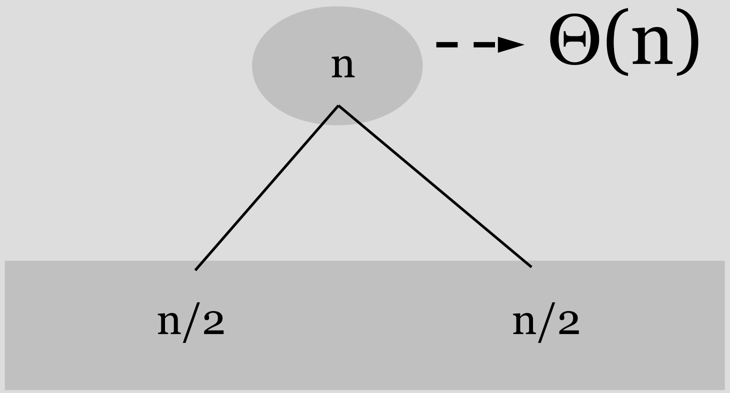
$L(n) = 2U(n/2) + \Theta(n)$ lucky

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

we consequently get

$$L(n) = 2(L(n/2 - 1) + \Theta(n)) + \Theta(n)$$
$$= 2L(n/2 - 1) + \Theta(n)$$
$$= \Theta(n \log n)$$





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

1 4 7 5

- 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

```
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)
       RandomizedQuicksort(A,m+1,r)
```

Stability

- Quicksort and Heap Sort are not stable
  - swaps during partitioning destroy previous order
  - research has been done on making Quicksort stable, but did not lead to practical outcomes
- When stability is needed:
  - remember the original position and use it in sorting
  - use a different algorithm (e.g., Merge Sort)

Summary

- Nearly complete binary trees
- Heap data structure
- Heapsort
  - based on heaps
  - worst case is n log n
- Quicksort:
  - partition based sort algorithm
  - popular algorithm
  - very fast on average
  - worst case performance is quadratic

Summary/2

- Comparison of sorting methods.
- Absolute values are not important; relate values to each other.
- Relate values to the complexity (n log n, n^2).
- Running time in seconds, n=2048.

	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