Algorithms and Data Structures
Lecture VI

- Sorting algorithms
- Heapsort
  - Heap *data structure* and priority queue *ADT*
- Quick sort
  - a popular algorithm, very fast on average

Why Sorting?

- “When in doubt, sort” – one of the principles of algorithm design. Sorting used as a subroutine in many of the algorithms:
  - Searching in databases: we can do binary search on sorted data
  - A large number of computer graphics and computational geometry problems
  - Closest pair, element uniqueness, frequency distribution

Why Sorting? (2)

- A large number of algorithms developed representing different algorithm design techniques.
- A 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 $\Theta(n \log n)$; but requires additional memory

Selection Sort

```
Selection-Sort(A[1..n]):
   For i = n downto 2
A:     Find the largest element among A[1..i]
B:     Exchange it with A[i]
```

- A takes $\Theta(n)$ and B takes $\Theta(1)$: $\Theta(n^2)$ in total
- Idea for improvement: use a *data structure*, to do both A and B in $O(lg\ n)$ time, balancing the work, achieving a better trade-off, and a total running time $O(n\ log\ n)$

Heap Sort

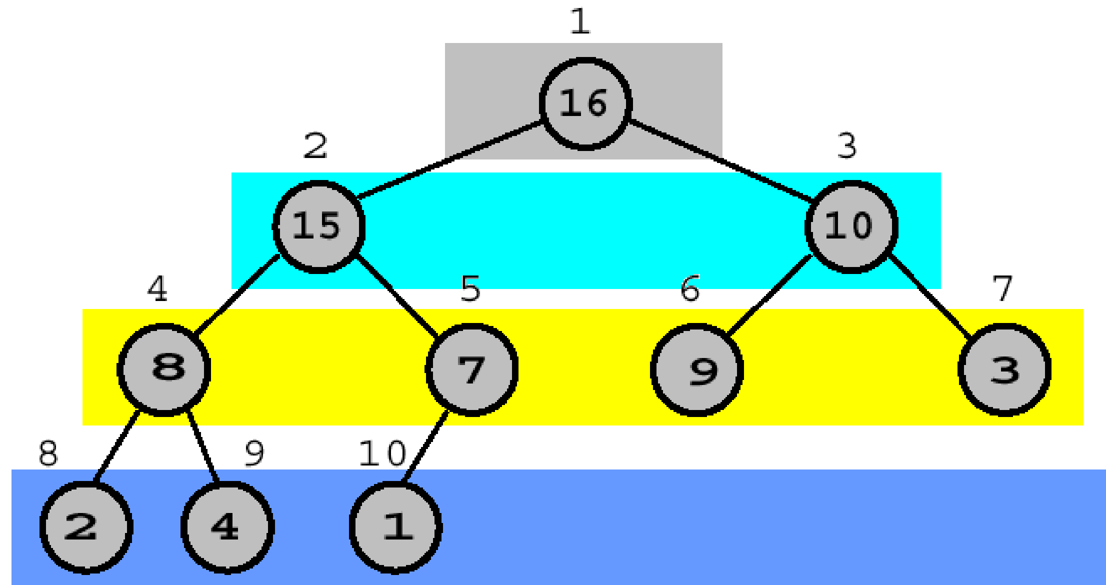
- Binary heap data structure *A*
  - array
  - Can be viewed as a nearly complete binary tree
    - All levels, except the lowest one are completely filled
  - The key in root is greater or equal than all its children, and the left and right subtrees are again binary heaps
- Two attributes
  - length[*A*]
  - heap-size[*A*]

Heap Sort (3)

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

Left (i)
return $2i$

Right (i)
return $2i+1$



Heap propertiy:

$A[\text{Parent}(i)] \geq A[i]$

1	2	3	4	5	6	7	8	9	10
16	15	10	8	7	9	3	2	4	1

Level: 3 2 1 0

Heap Sort (4)

- Notice the implicit tree links; children of node *i* are 2*i* and 2*i*+1
- Why is this useful?
  - In a binary representation, a multiplication/division by two is left/right shift
  - Adding 1 can be done by adding the lowest bit

Heapify

- *i* is index into the array *A*
- Binary trees rooted at Left(*i*) and Right(*i*) are heaps
- But, *A*[*i*] might be smaller than its children, thus violating the heap property
- The method **Heapify** makes *A* a heap once more by moving *A*[*i*] down the heap until the heap property is satisfied again

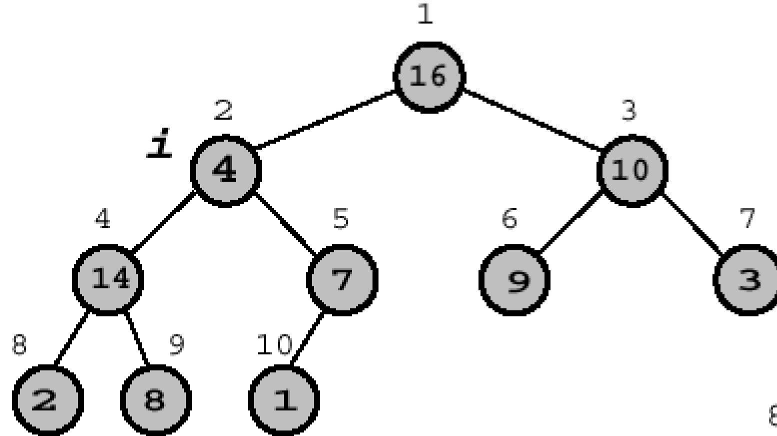
n is total number of elements

HEAPIFY(A, i)

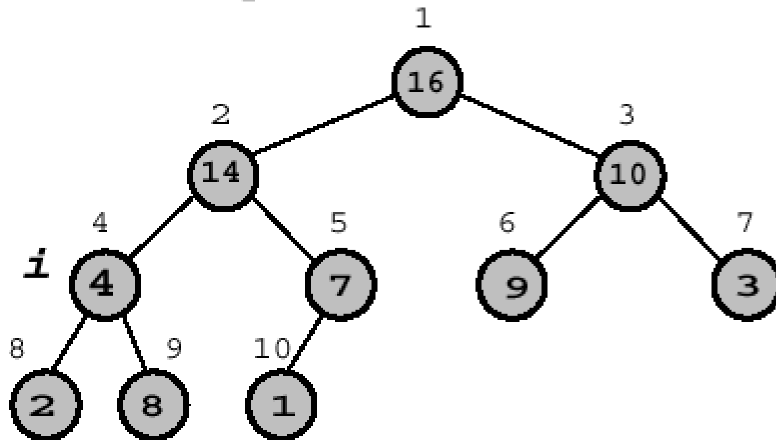
1 ▷ Left & Right subtrees of i are heaps.
2 ▷ Makes subtree rooted at i a heap.
3 $l \leftarrow$ LEFT(i) ▷ $l = 2i$
4 $r \leftarrow$ RIGHT(i) ▷ $r = 2i + 1$
5 **if** $l \leq n$ and $A[l] > A[i]$
6   **then** $largest \leftarrow l$
7   **else** $largest \leftarrow i$
8 **if** $r \leq n$ and $A[r] > A[largest]$
9   **then** $largest \leftarrow r$
10 **if** $largest \neq i$
11   **then** exchange $A[i] \leftrightarrow A[largest]$
12   HEAPIFY$(A, largest)$

Heapify Example

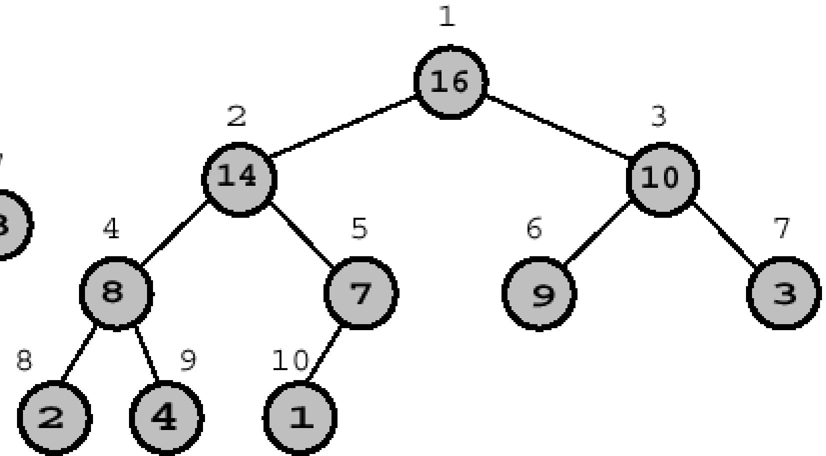
1. Call HEAPIFY(A,2)



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: Running Time

- The running time of Heapify on a subtree of size *n* rooted at node *i* is
  - determining the relationship between elements: $\Theta(1)$
  - plus the time to run Heapify on a subtree rooted at one of the children of *i*, where 2n/3 is the worst case

$$T(n) \leq T(2n/3) + \Theta(1) \quad \Rightarrow \quad T(n) = O(\log n)$$

  - Alternatively
    - Running time on a node of height *h*: *O(h)*

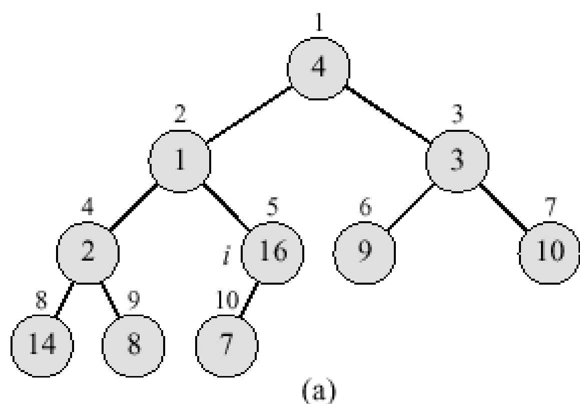
Building a Heap

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

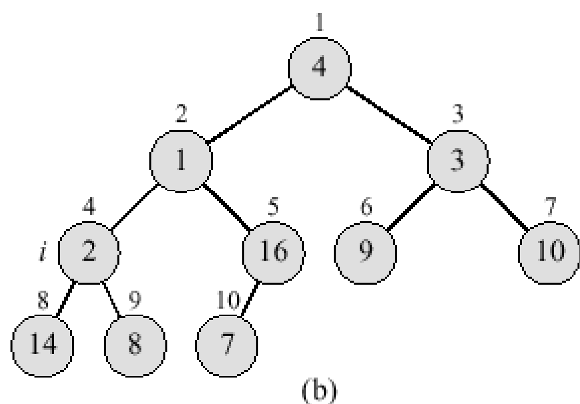
```
BUILD-HEAP(A)
  1 for i ← ⌊n/2⌋ downto 1
  2        do HEAPIFY(A, i)
```

Building a Heap

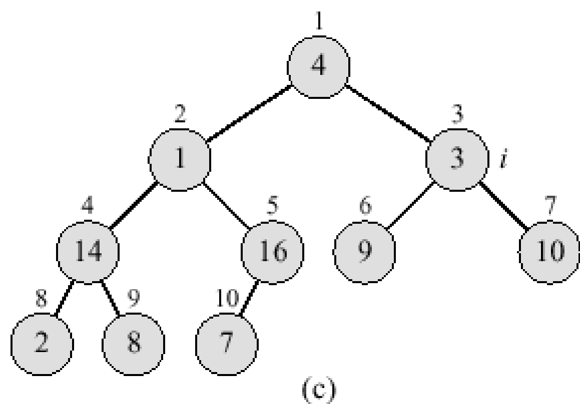
A	4	1	3	2	16	9	10	14	8	7



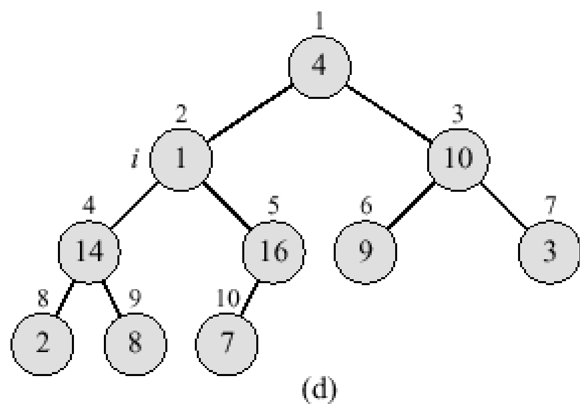
(a)



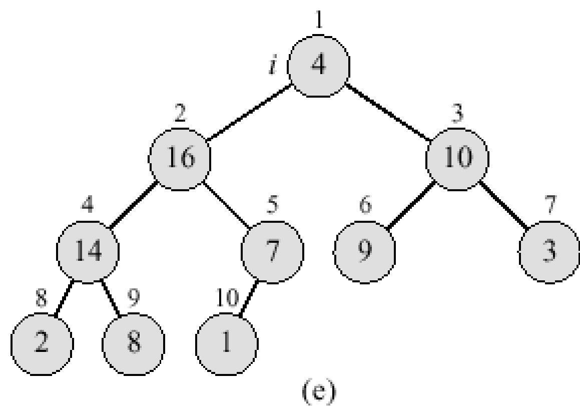
(b)



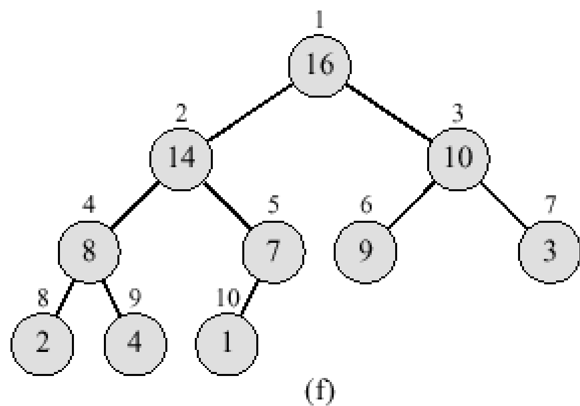
(c)



(d)



(e)



(f)

- Correctness: induction on *i*, all trees rooted at *m* > *i* are heaps
- Running time: *n* calls to Heapify = *n O(lg n)* = *O(n lg n)*
- Good enough for an *O(n lg n)* bound on Heapsort, but sometimes we build heaps for other reasons, would be nice to have a tight bound
  - Intuition: for most of the time Heapify works on smaller than *n* element heaps

Building a Heap: Analysis (2)

- Definitions
  - height of node: longest path from node to leaf
  - height of tree: height of root

```
BUILD-HEAP(A)
 1 for i ← ⌊n/2⌋ downto 1
 2       do HEAPIFY(A, i)
```

  - time to Heapify = O(height of subtree rooted at *i*)
  - assume $n = 2^k - 1$ (a complete binary tree $k = \lfloor lg\ n \rfloor$)

$$
\begin{aligned}
T(n) &= O\left(\frac{n+1}{2} + \frac{n+1}{4} \cdot 2 + \frac{n+1}{8} \cdot 3 + \ldots + 1 \cdot k \right) \\
&= O\left((n+1) \cdot \sum_{i=1}^{\lfloor \lg n \rfloor} \frac{i}{2^i} \right) \text{ since } \sum_{i=1}^{\lfloor \lg n \rfloor} \frac{i}{2^i} = \frac{1/2}{(1-1/2)^2} = 2 \\
&= O(n)
\end{aligned}
$$

Building a Heap: Analysis (3)

- How? By using the following "trick"

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ if } |x| < 1 \quad \text{//differentiate}$$

$$\sum_{i=1}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2} \quad \text{//multiply by } x$$

$$\sum_{i=1}^{\infty} i \cdot x^{i} = \frac{x}{(1-x)^2} \quad \text{//plug in } x = \frac{1}{2}$$

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1/2}{1/4} = 2$$

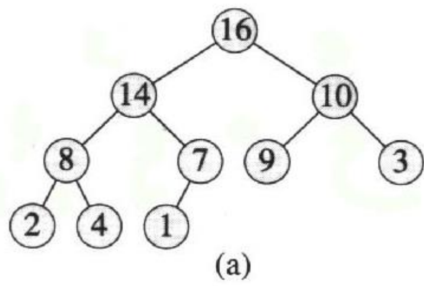
- Therefore Build-Heap time is O(*n*)

Heap Sort

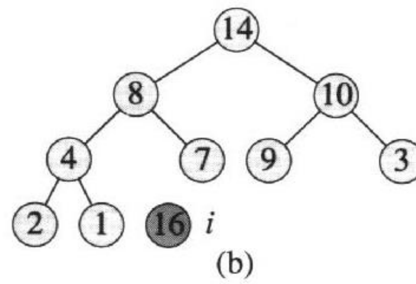
HEAPSORT(A)	**Analysis**
1 BUILD-HEAP(A)	O(n)
2 **for** $i \leftarrow n$ **downto** 2	n times
3 **do** exchange $A[1] \leftrightarrow A[i]$	O(1)
4 $n \leftarrow n - 1$	O(1)
5 HEAPIFY($A, 1$)	O($\lg n$)

- The total running time of heap sort is *O(n lg n)* + Build-Heap(*A*) time, which is *O(n)*

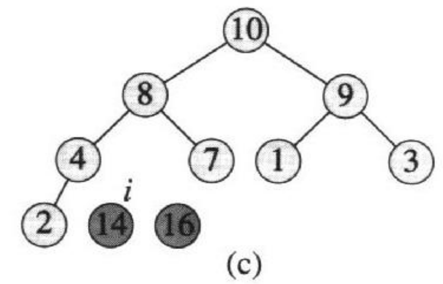
Heap Sort



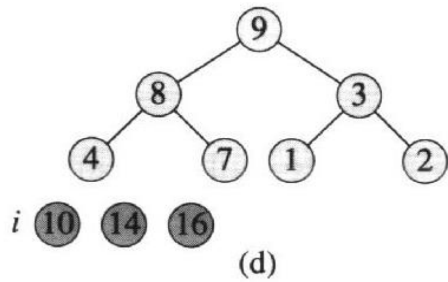
(a)



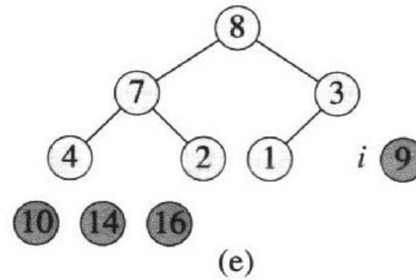
(b)



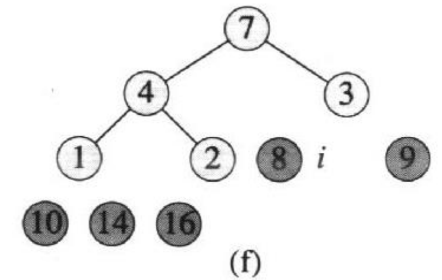
(c)



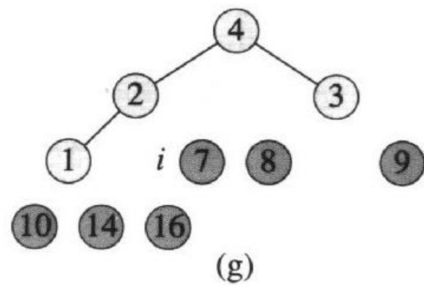
(d)



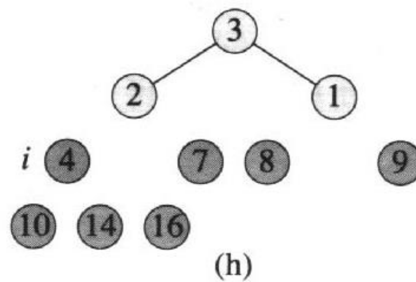
(e)



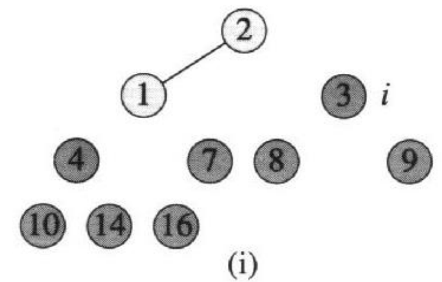
(f)



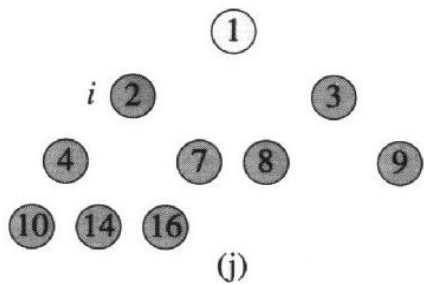
(g)



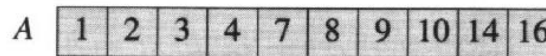
(h)



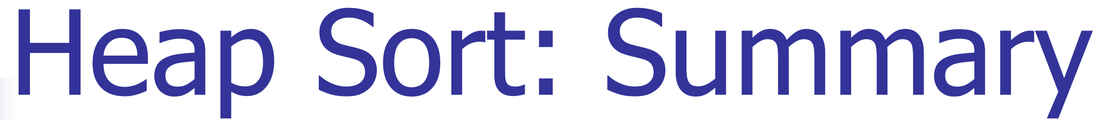
(i)



(j)



(k)



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

Priority Queues

- A priority queue is an *ADT(abstract data type)* for maintaining a set S of elements, each with an associated value called key
- A PQ supports the following operations
  - Insert(S,x) insert element x in set S ($S \leftarrow S \cup \{x\}$)
  - Maximum(S) returns the element of S with the largest key
  - Extract-Max(S) returns and removes the element of S with the largest key

Priority Queues (2)

- Applications:
  - job scheduling shared computing resources (Unix)
  - Event simulation
  - As a building block for other algorithms
- A Heap can be used to implement a PQ

Priority Queues (3)

- Removal of max takes constant time on top of Heapify $\Theta(\lg n)$

HEAP-EXTRACT-MAX(A)

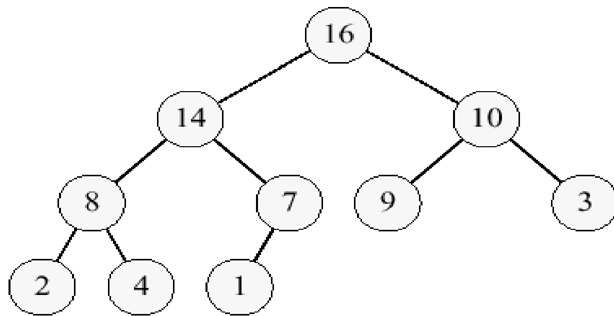
1. $\triangleright$ Removes and returns largest element of A
2. $max \leftarrow A[1]$
3. $A[1] \leftarrow A[n]$
4. $n \leftarrow n - 1$
5. HEAPIFY$(A, 1)$ $\quad\triangleright$ Remakes heap
6. **return** max

Priority Queues (4)

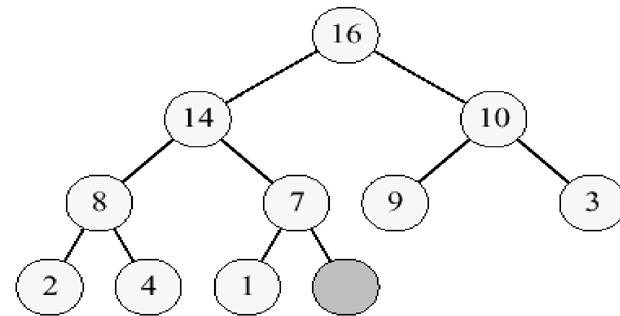
- Insertion of a new element
  - enlarge the PQ and propagate the new element from last place "up" the PQ
  - tree is of height *lg n*, running time: $\Theta(\lg n)$

```
HEAP-INSERT(A, key)
  1 heap-size[A] ← heap-size[A] + 1
  2 i ← heap-size[A]
  3 while i > 1 and A[PARENT(i)] < key
  4        do A[i] ← A[PARENT(i)]
  5           i ← PARENT(i)
  6 A[i] ← key
```

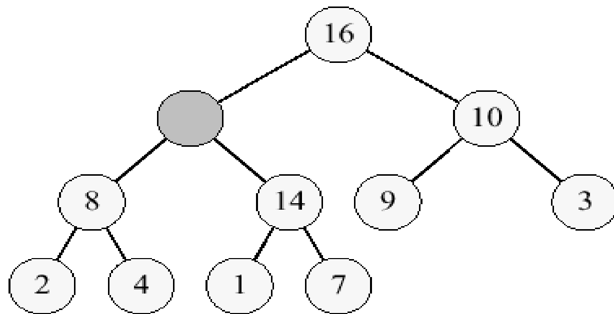
Priority Queues (5)



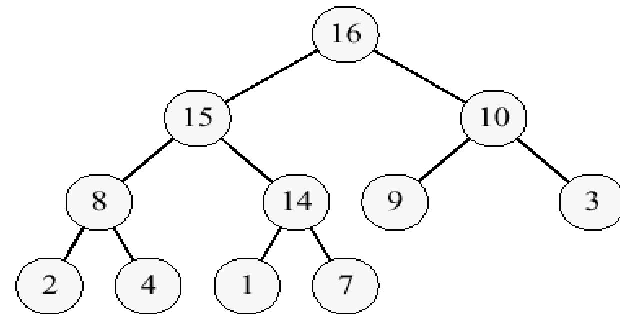
(a)



(b)



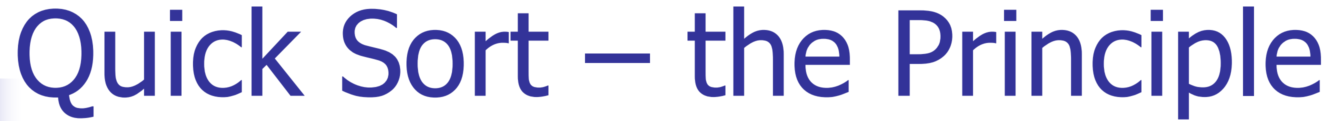
(c)



(d)

Quick Sort

- Characteristics
  - sorts in "place," i.e., does not require an additional array
  - like insertion sort, unlike merge sort
  - very practical, average sort performance $O(n \log n)$, 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

- Linear time partitioning procedure

```
Partition(A,p,r)
01  x←A[r]
02  i←p-1
03  j←r+1
04  while TRUE
05     repeat j←j-1
06        until A[j] ≤x
07     repeat i←i+1
08        until A[i] ≥x
09     if i<j
10        then exchange A[i]↔A[j]
11        else return j
```

≤ X=10 ≤

17	12	6	19	23	8	5	10
10	12	6	19	23	8	5	17
10	5	6	19	23	8	12	17
10	5	6	8	23	19	12	17

Quick Sort Algorithm

- Initial call **Quicksort(A, 1, length[A])**

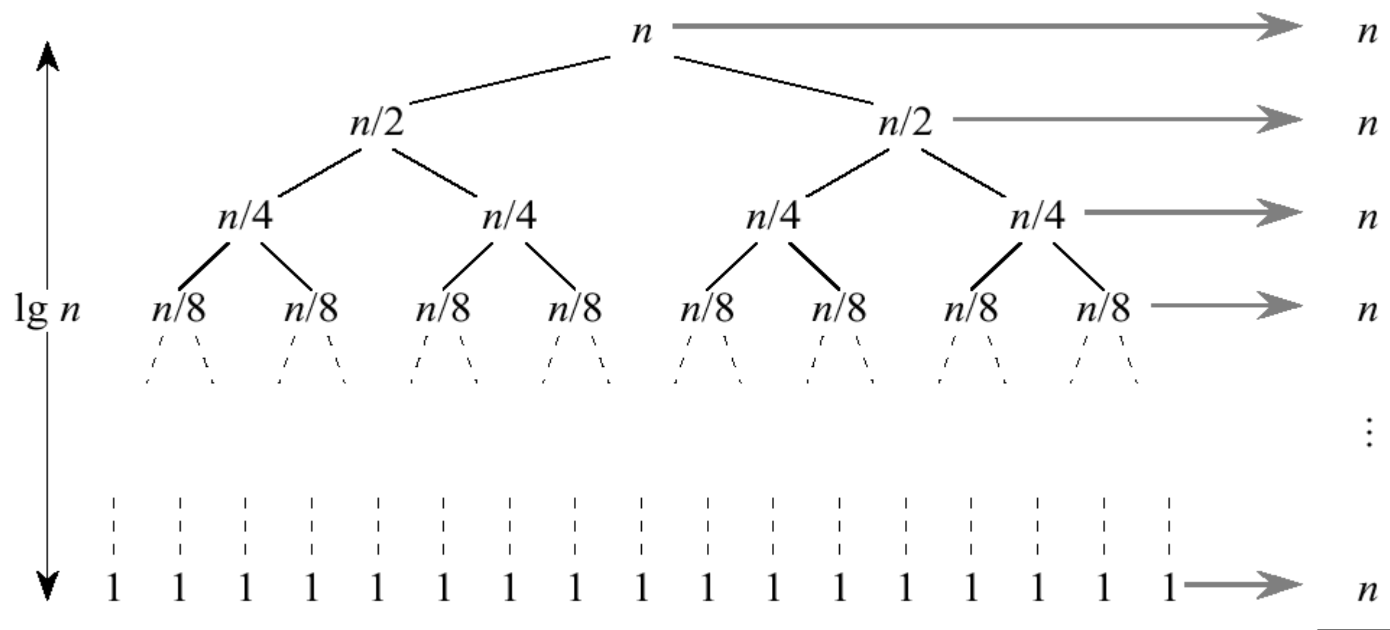
```
Quicksort(A,p,r)
01 if p<r
02    then q←Partition(A,p,r)
03          Quicksort(A,p,q)
04          Quicksort(A,q+1,r)
```

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 $T(n) = 2T(n/2) + \Theta(n)$

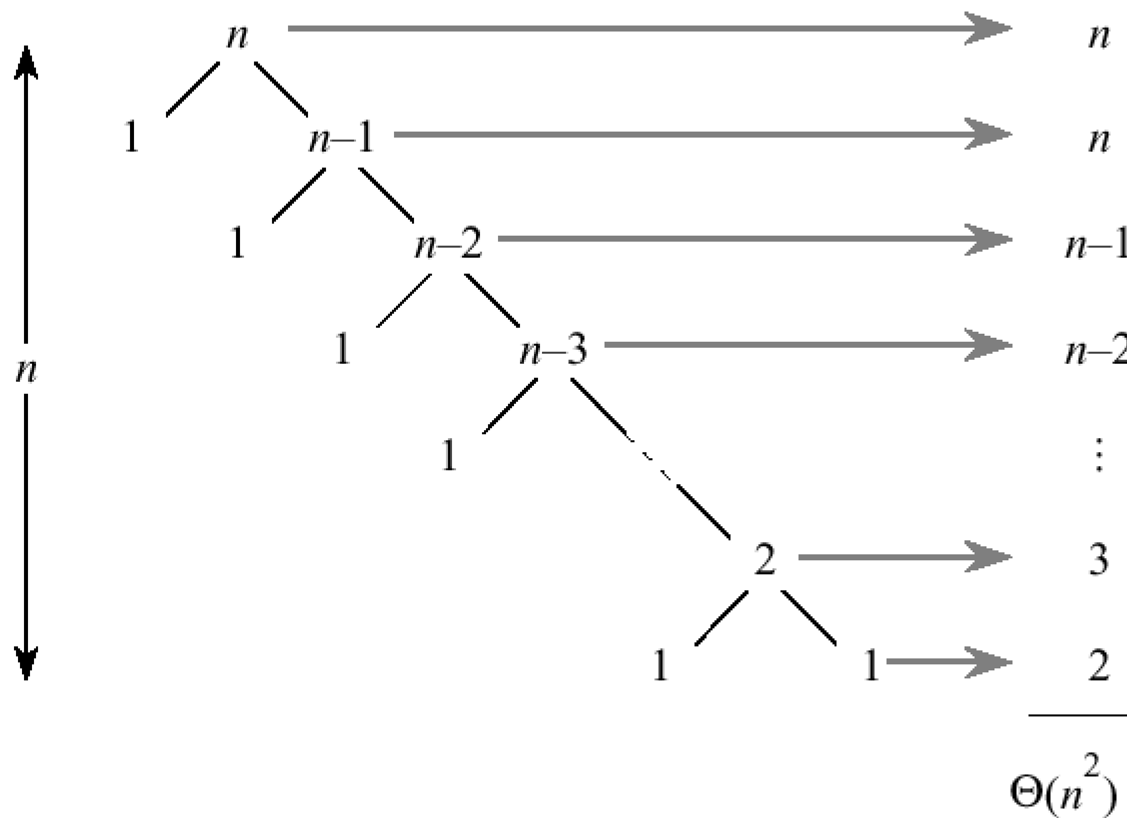


Worst Case

- What is the worst case?
- One side of the parition has only one element

$$
\begin{aligned}
T(n) = T(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta(\sum_{k=1}^{n} k) \\
&= \Theta(n^2)
\end{aligned}
$$

Worst Case (2)



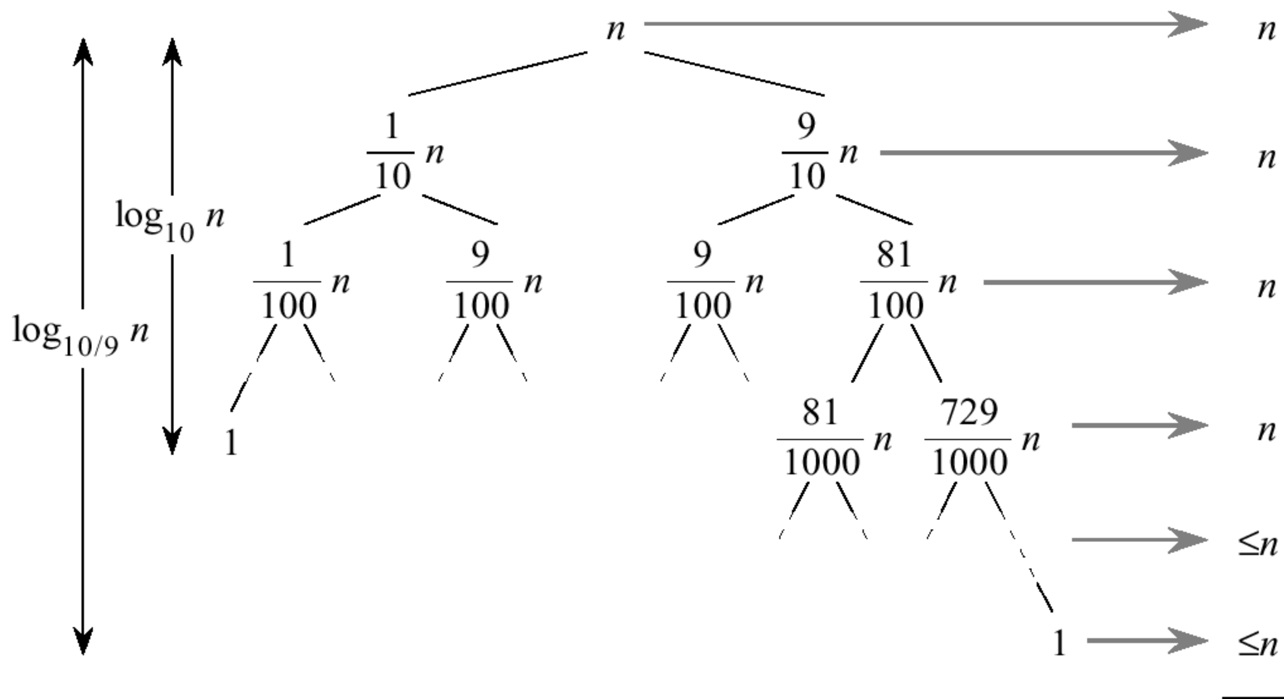
Worst Case (3)

- When does the worst case appear?
  - input is sorted
  - input reverse sorted
- Same recurrence for the worst case of insertion sort
- However, sorted input yields the best case for insertion sort!

Analysis of Quicksort

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

$$T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n)!$$



An Average Case Scenario

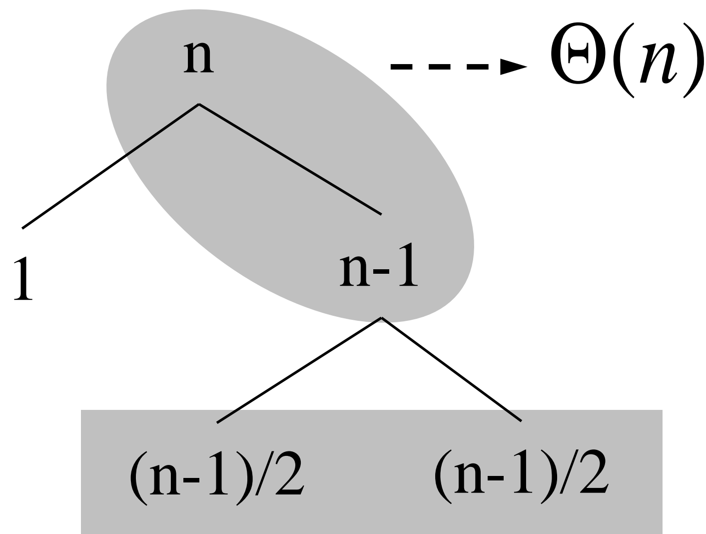
- Suppose, we alternate lucky and unlucky cases to get an average behavior

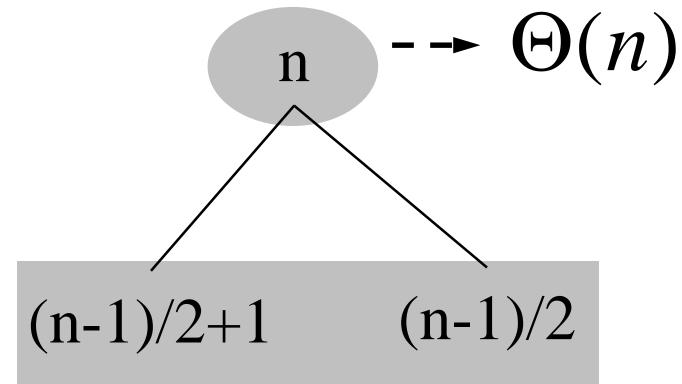
$L(n) = 2U(n/2) + \Theta(n)$ lucky

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

we consequently get

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





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)

```
Randomized-Partition(A,p,r)
01 i←Random(p,r)
02 exchange A[r]↔A[i]
03 return Partition(A,p,r)
```

```
Randomized-Quicksort(A,p,r)
01 if p<r then
02    q←Randomized-Partition(A,p,r)
03    Randomized-Quicksort(A,p,q)
04    Randomized-Quicksort(A,q+1,r)
```

Next lecture

- ADTs and Data Structures
  - Elementary data structures
  - Trees