CMP302: Algorithms Design and Analysis

Lecture 02: Sorting & Order Statistics

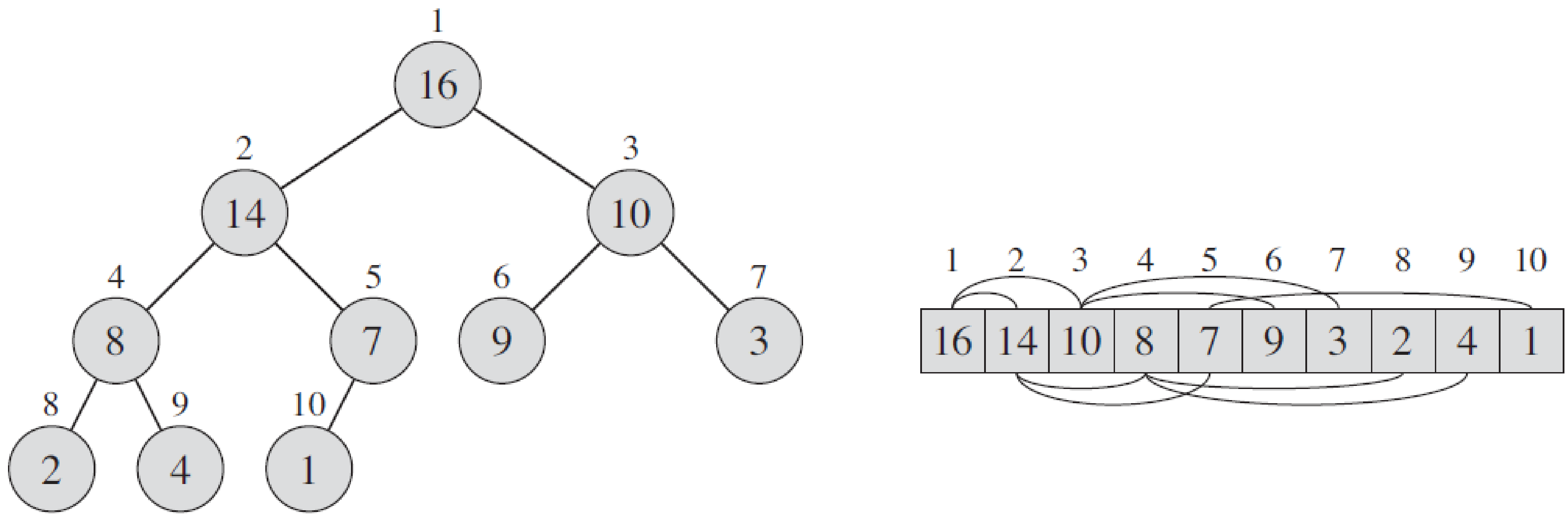
Ahmed Hamdy

Computer Engineering Department

Cairo University

Fall 2017

Heaps



(a) (b)

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

RIGHT(i)

return $2i + 1$

Heaps

Optimizations of heap operations

- $2i$ computed as shift left
- $2i + 1$ computed as shift left and adding 1 / ORing 1
- $\lfloor i/2 \rfloor$ computed as shift right
- Implement heap operations (parent, left, right) as macros or inline functions

Heaps

- Max-heap:

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

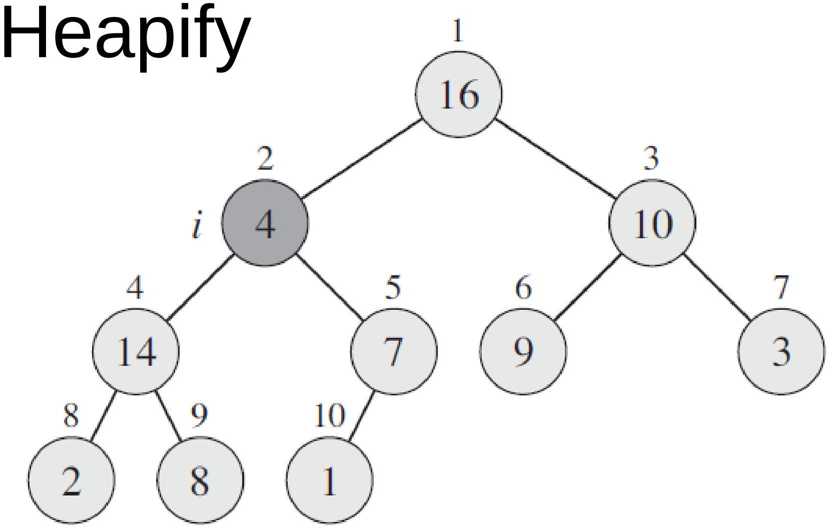
- Min-heap:

$$A[\textsc{Parent}(i)] \leq A[i]$$

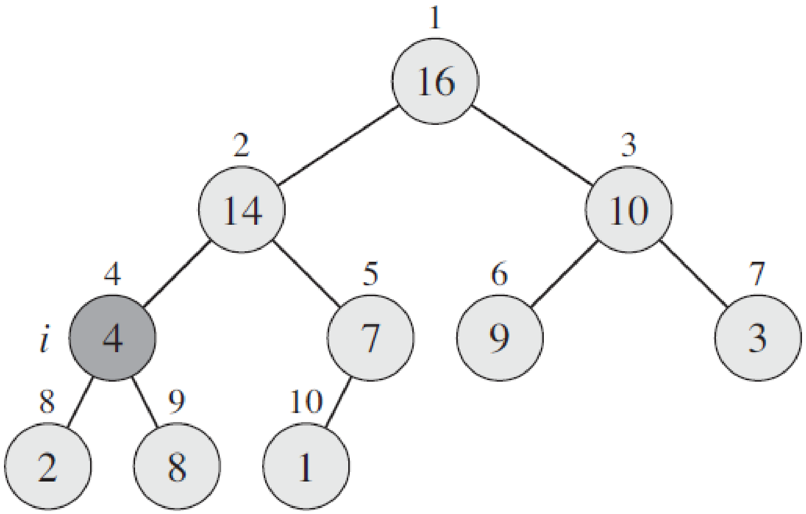
- Height: $\Theta(\log n)$
- Operations: $O(\log n)$

Heaps

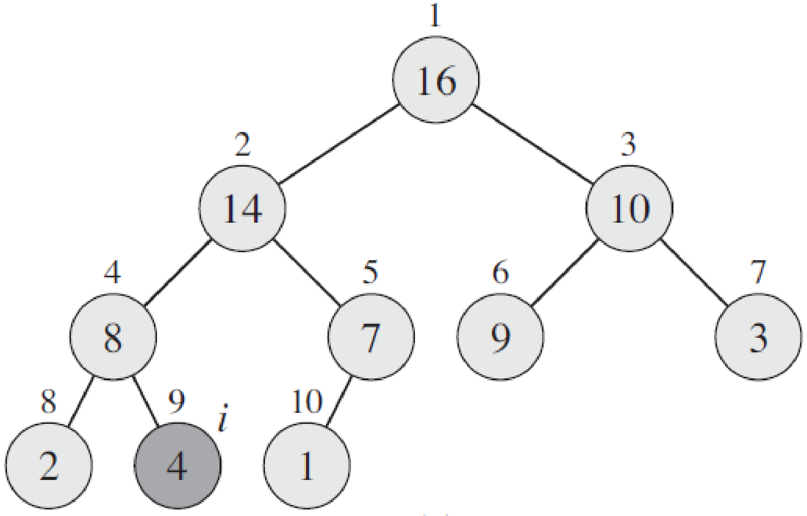
Max-Heapify



(a)



(b)



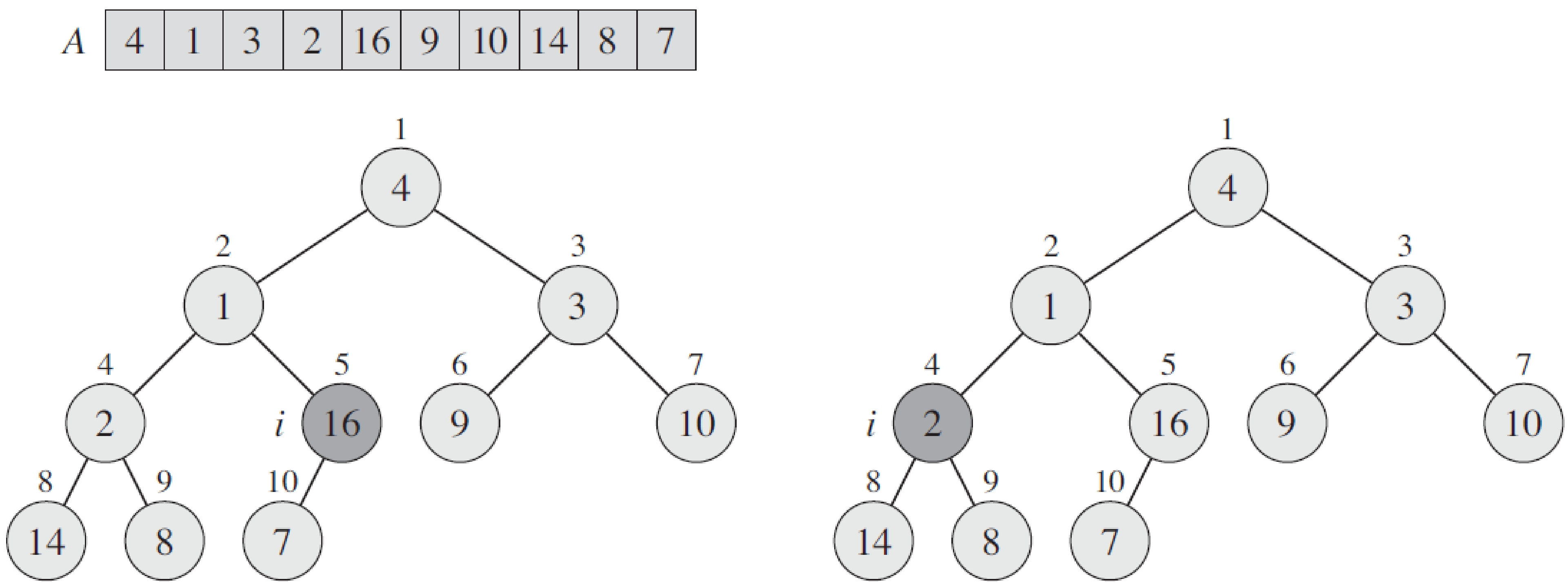
(c)

Heaps

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

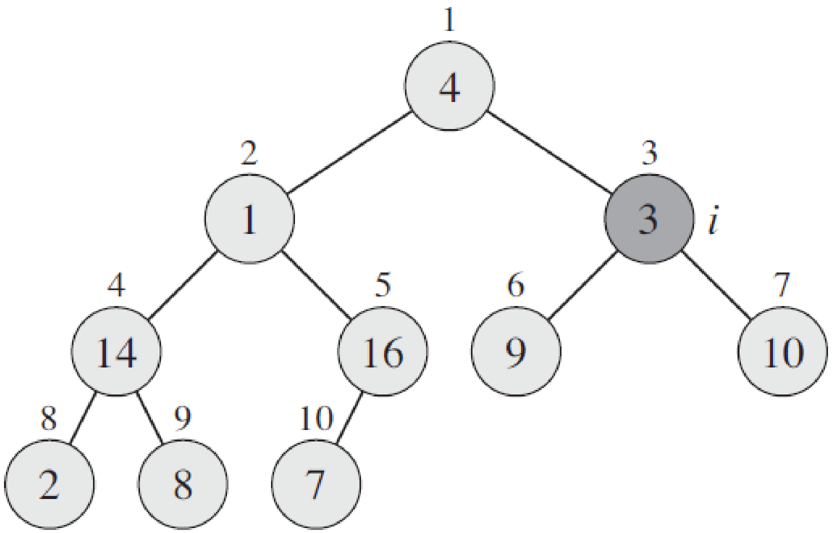
- Max-Heapify complexity: $O(\log n)$

Heaps



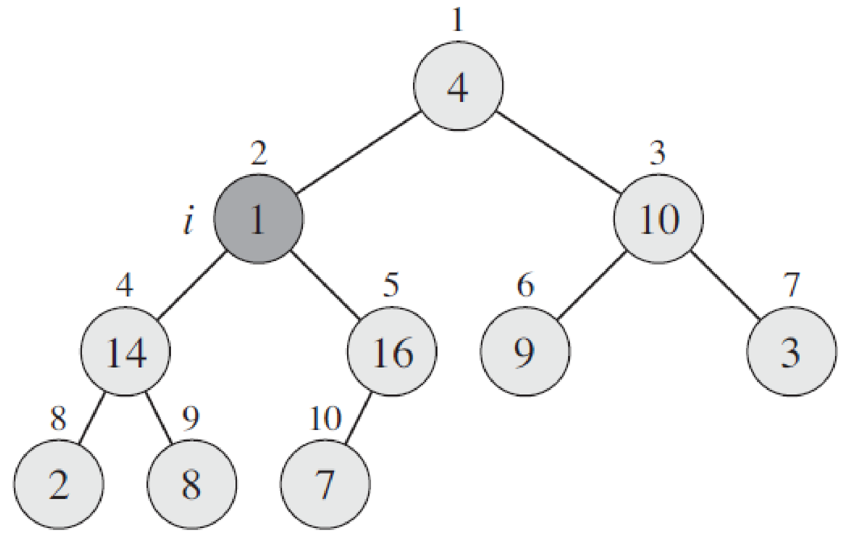
(a) (b)

Heaps



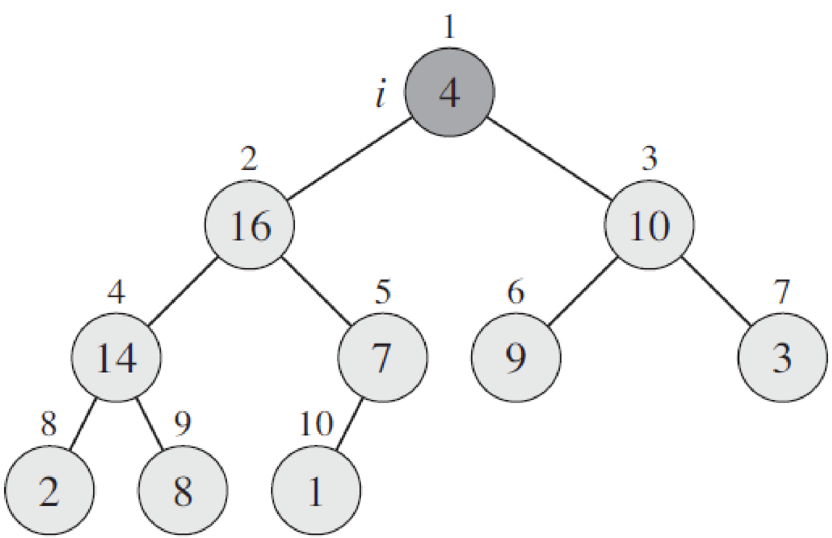
1
4
2
3
1
3
i
4
5
6
7
14
16
9
10
8
9
10
2
8
7

(c)



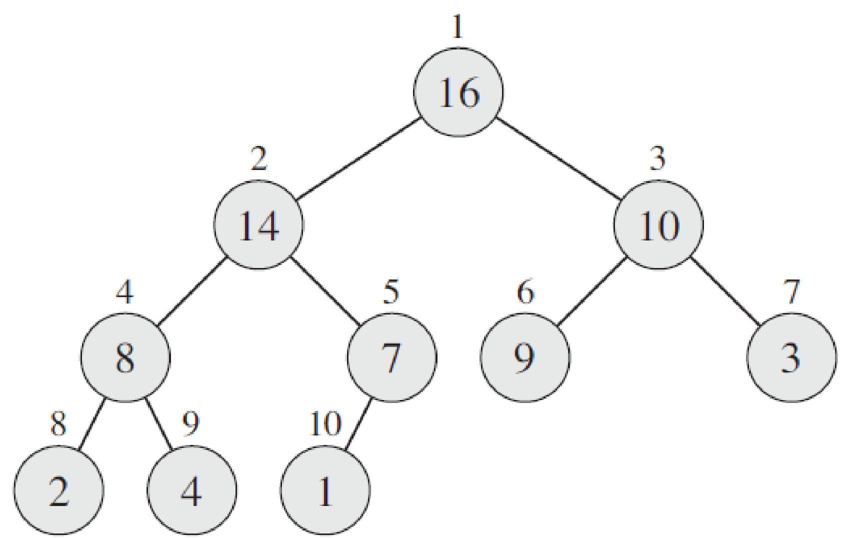
1
4
2
3
i
1
10
4
5
6
7
14
16
9
3
8
9
10
2
8
7

(d)



1
i
4
2
3
16
10
4
5
6
7
14
7
9
3
8
9
10
2
8
1

(e)



1
16
2
3
14
10
4
5
6
7
8
7
9
3
8
9
10
2
4
1

(f)

Heaps

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

- Build-Max-Heap Complexity:
  - $O(n)$ calls to Max-Heapify
  - Thus overall complexity of Build-Max-Heap: $O(n \log n)$
  - Is this tight bound??

Heaps

- Height of n-element heap: $\lfloor \log n \rfloor$
- Number of nodes at height h: $\left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Build-Max-Heap is bounded by

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

- Using

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

for $|x| < 1$.

$$\Rightarrow$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

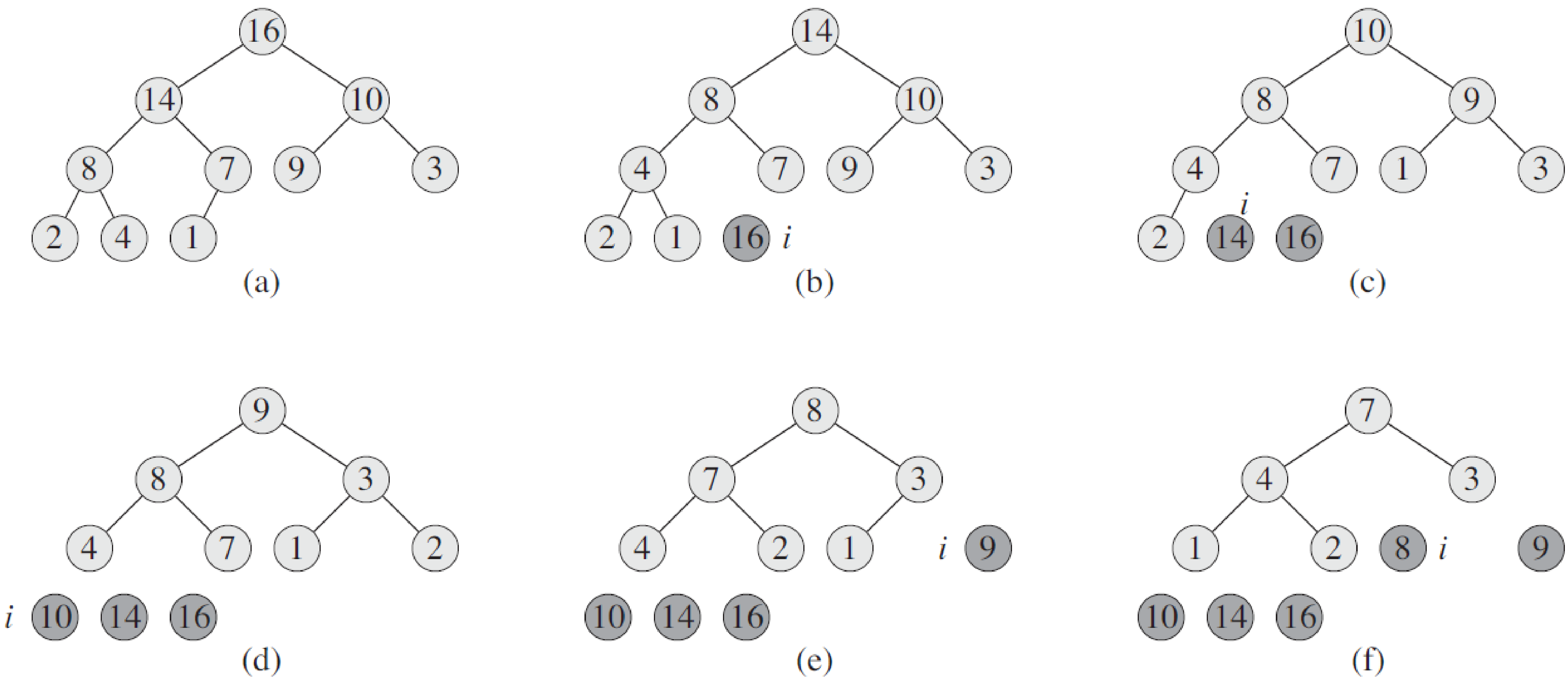
Heaps

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

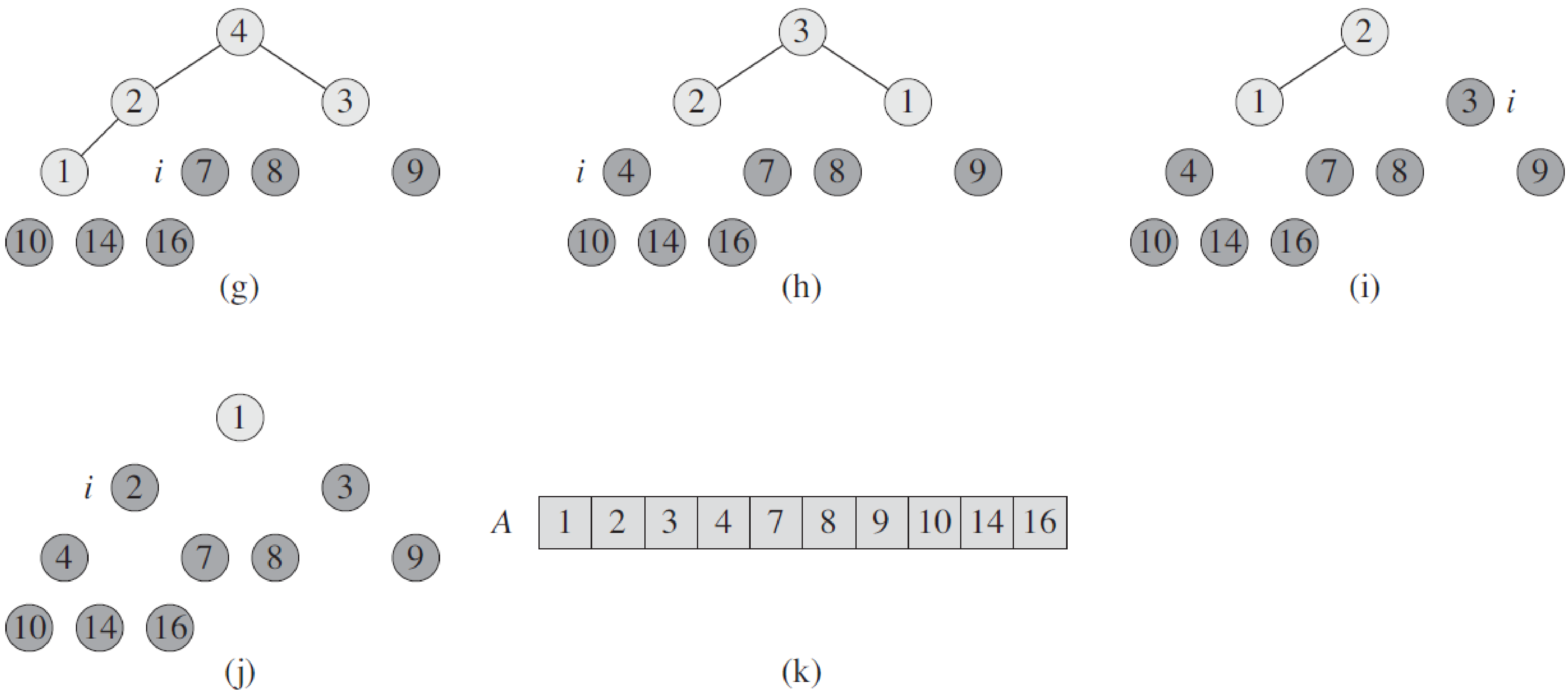
Becomes

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) \, . \end{aligned}$$

Heapsort


16
14
10
8
7
9
3
2
4
1
(a)
14
8
10
4
7
9
3
2
1
16
i
(b)
10
8
9
4
7
1
3
i
2
14
16
(c)
9
8
3
4
7
1
2
i
10
14
16
(d)
8
7
3
4
2
1
i
9
10
14
16
(e)
7
4
3
1
2
8
i
9
10
14
16
(f)

Heapsort



4
2
3
1
i
7
8
9
10
14
16
(g)
3
2
1
i
4
7
8
9
10
14
16
(h)
2
1
3
i
4
7
8
9
10
14
16
(i)
1
i
2
3
4
7
8
9
10
14
16
(j)
A
1 2 3 4 7 8 9 10 14 16
(k)

Heapsort

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

Complexity

– $O(n)$ calls to Max-Heapify

– Thus overall complexity of Heapsort: $O(n \log n)$

Priority queues

- Useful in applications where priority is the criteria for selection
- Example, scheduling jobs
- Max-priority or min-priority queues

Priority queues

```
HEAP-MAXIMUM(A)
return A[1]
```

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

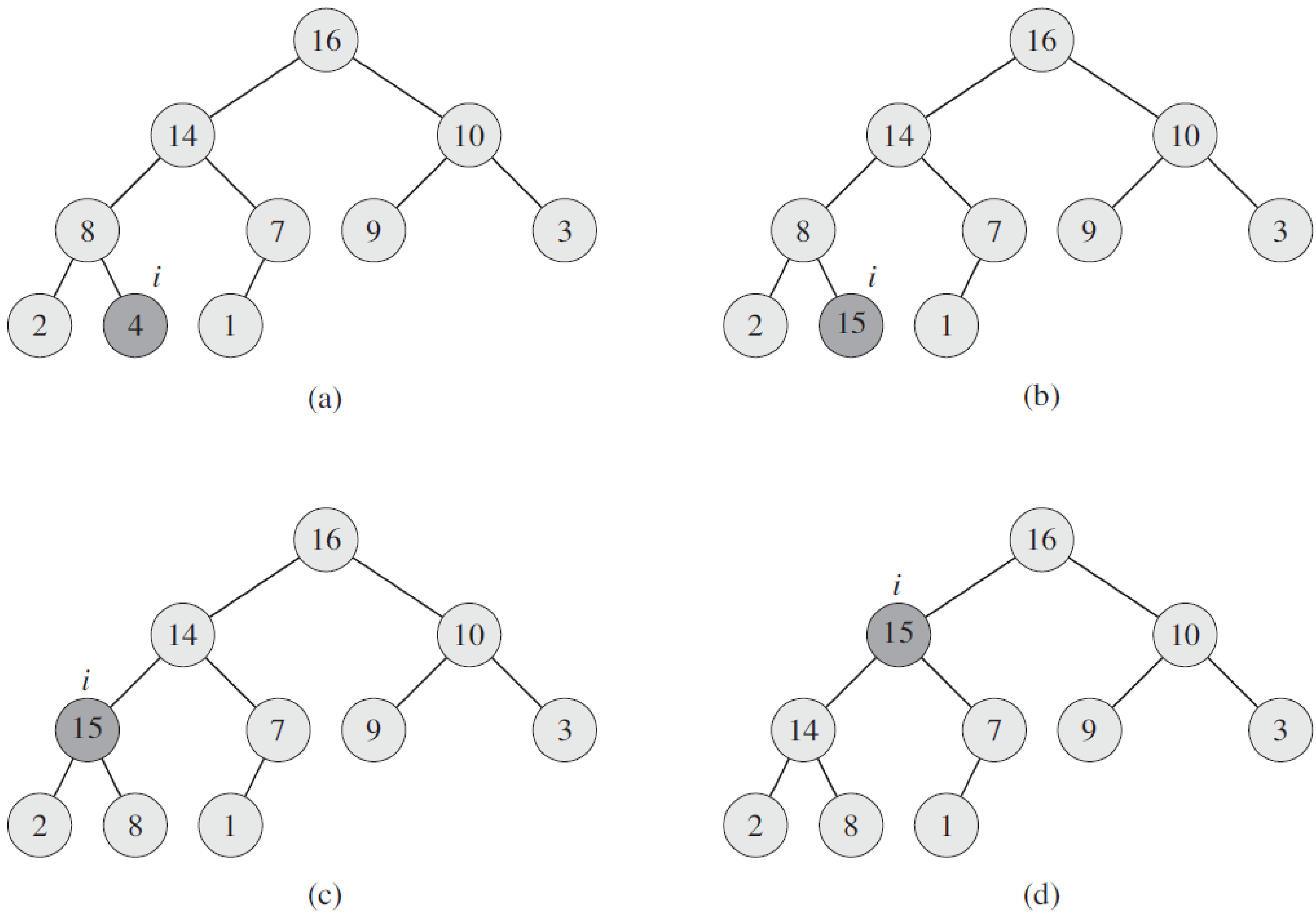
Priority queues

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

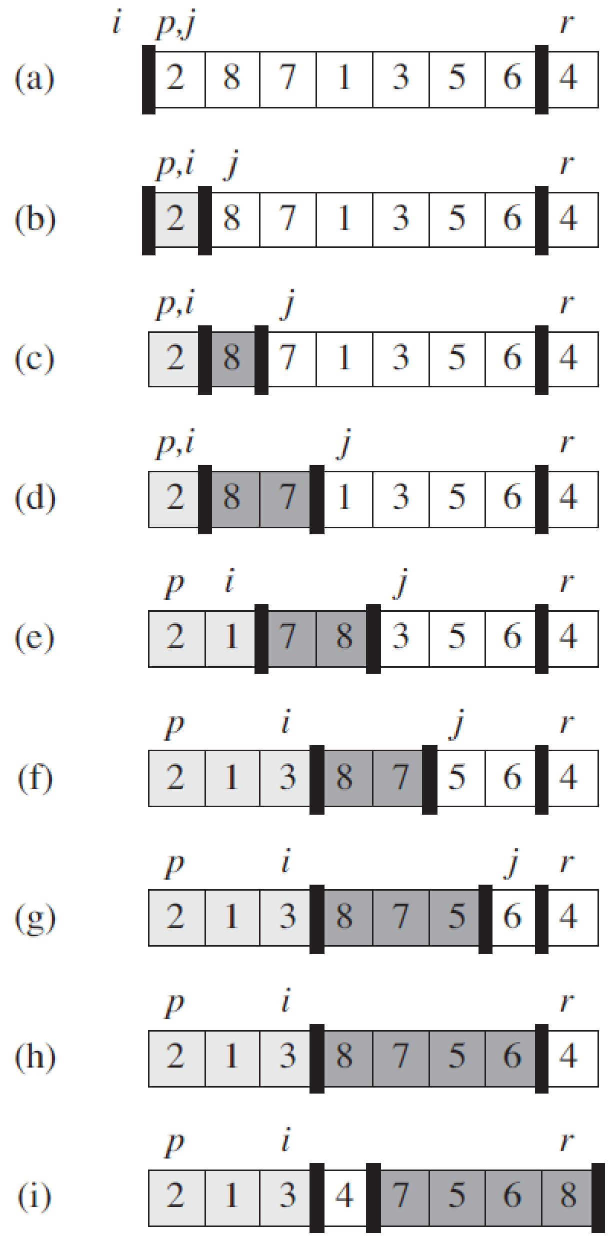
```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Priority queue

- Heap-Increase-Key



Quicksort



(a)
i p,j r
2 8 7 1 3 5 6 4
(b)
p,i j r
2 8 7 1 3 5 6 4
(c)
p,i j r
2 8 7 1 3 5 6 4
(d)
p,i j r
2 8 7 1 3 5 6 4
(e)
p i j r
2 1 7 8 3 5 6 4
(f)
p i j r
2 1 3 8 7 5 6 4
(g)
p i j r
2 1 3 8 7 5 6 4
(h)
p i r
2 1 3 8 7 5 6 4
(i)
p i r
2 1 3 4 7 5 6 8

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quicksort

Performance

- Worst case partitioning:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \ . \end{aligned}$$

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

- Best case partitioning:

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

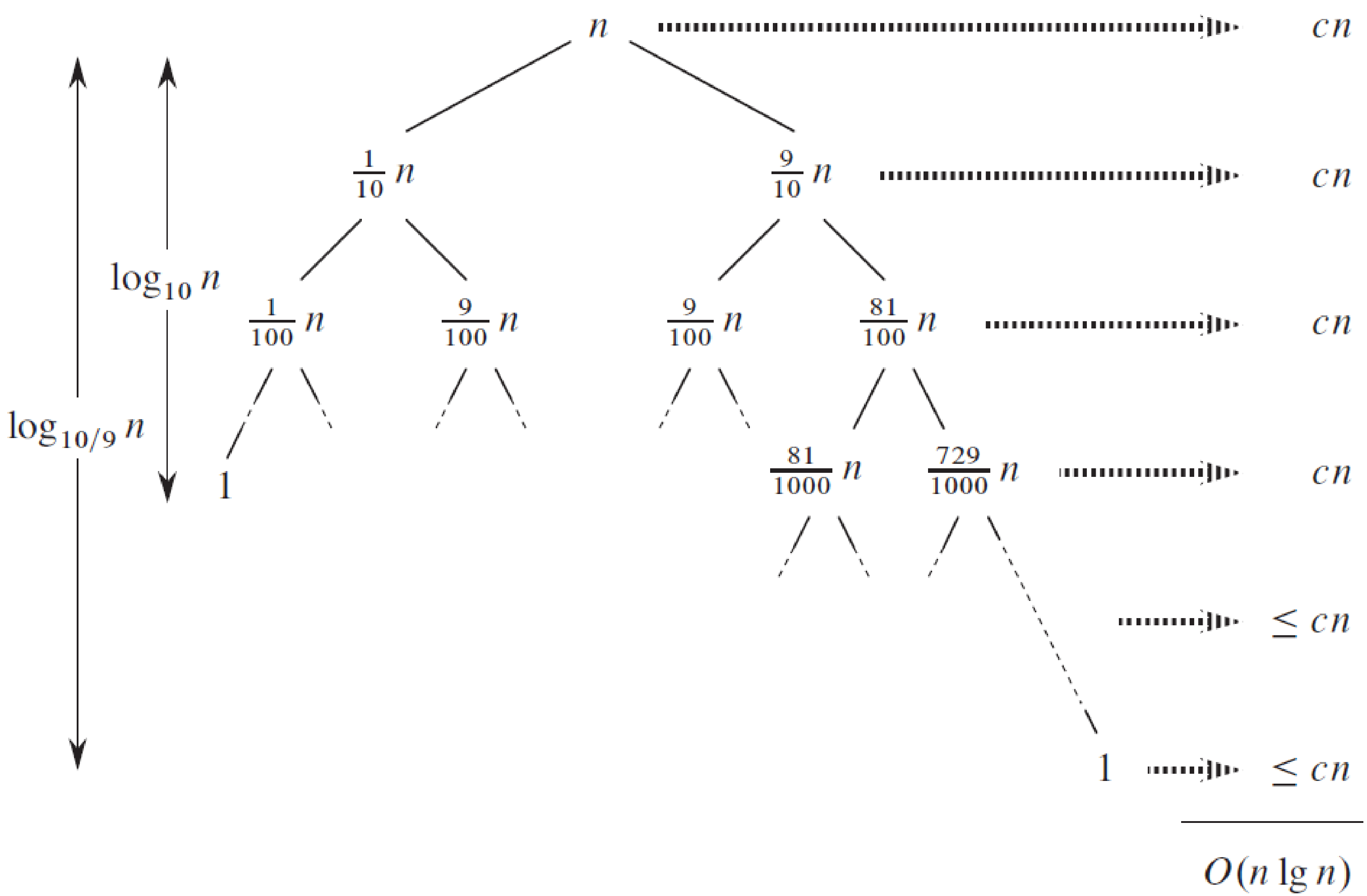
$$T(n) = \Theta(n \lg n)$$

Quicksort

Performance

- Balanced partitioning:

$$T(n) = T(9n/10) + T(n/10) + cn$$



Randomized quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

- Expected running time: $O(n \log n)$ when element values are distinct

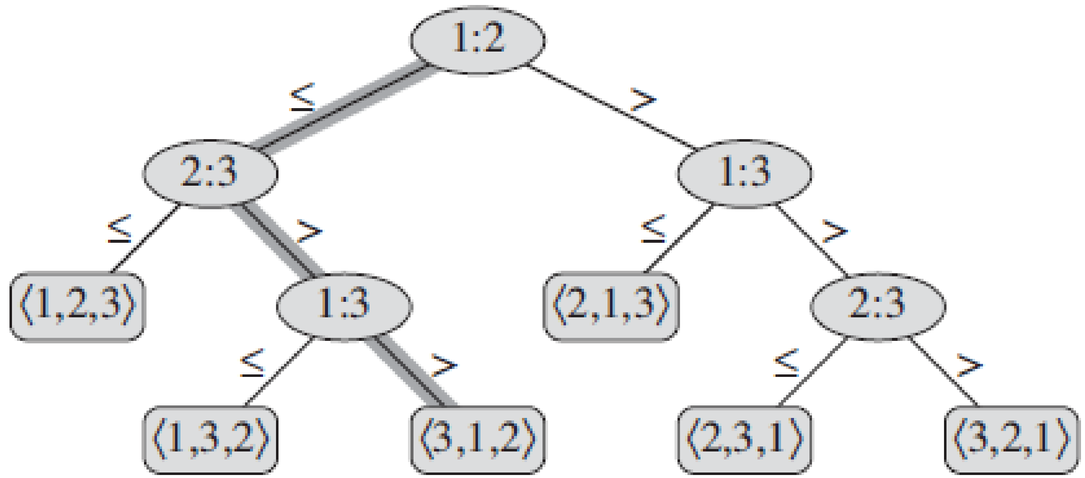
Comparison sort

- Insertion-sort, mergesort, heapsort and quicksort all use comparisons to gain order

- Can we obtain a lower bound for any comparison sort?

Comparison sort

- Decision tree



- Number of leaves in decision tree: $\geq n!$
- Number of leaves in binary tree: 2^h

$$n! \leq l \leq 2^h$$

$$\begin{aligned} h &\geq \lg(n!) \\ &= \Omega(n \lg n) \end{aligned}$$

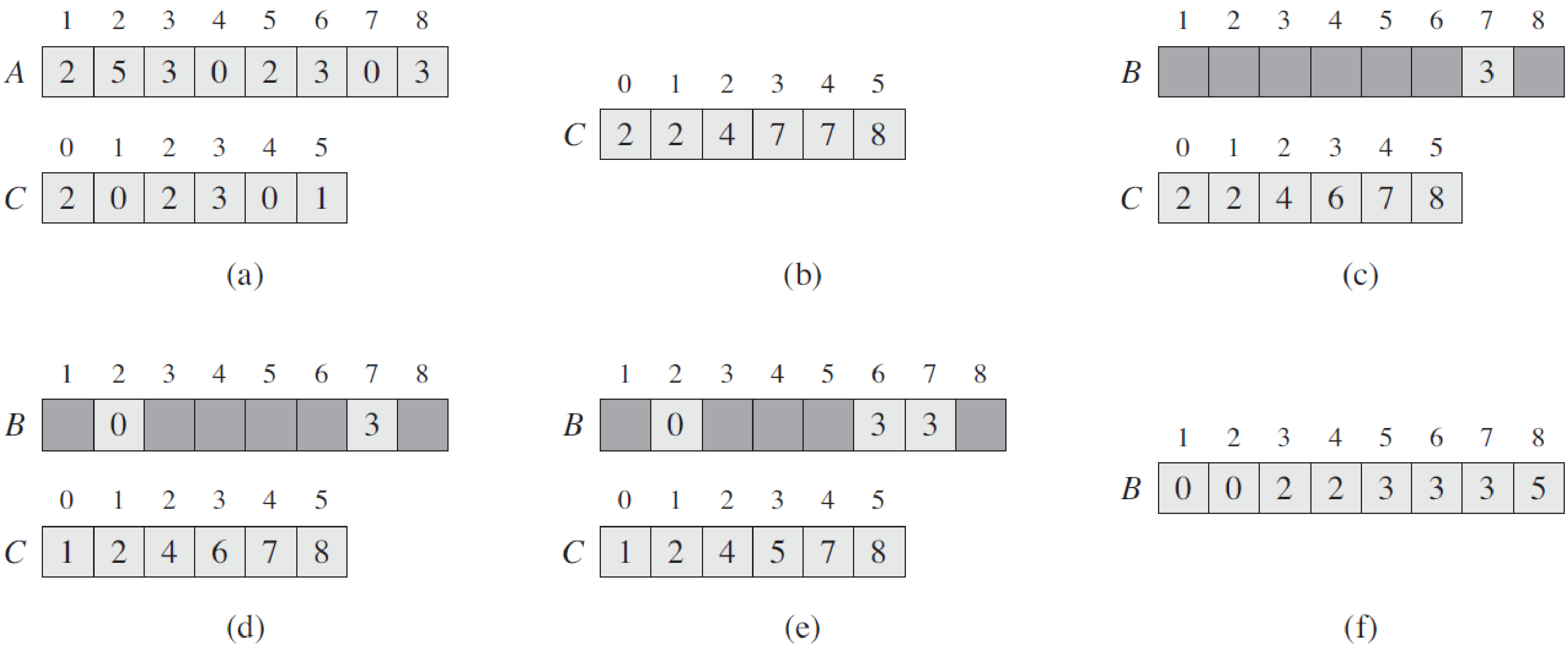
Comparison sort

- Mergesort and heapsort are asymptotically optimal comparison sorts

- Both have upper bound of $O(n \log n)$

- Quicksort isn't, why?

Counting sort



Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

When $k = O(n)$, the sort runs in $\Theta(n)$ time.

Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

- Complexity: $\Theta(n + k)$
- Typically used when $k = O(n)$, complexity $\Theta(n)$
- Stable

Sorting problem

- Sort a set of files according to their timestamps in ascending order??

- Assume just year, month, day.

Radix sort

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

- Use counting sort to sort each digit
- Complexity: $\Theta(d(n+k))$

Radix sort

- What is the optimal d to use given n b-bit numbers?

- Ask it in a different way, what is the optimal r-bits out of the b-bits to use for every digit ($d = \lceil b/r \rceil$)?

$$T(n, b) = \Theta(d(n + k)) = \Theta\left(\frac{b}{r}(n + 2^r)\right)$$

Radix sort

$$T(n,b) = \Theta(d(n+k)) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

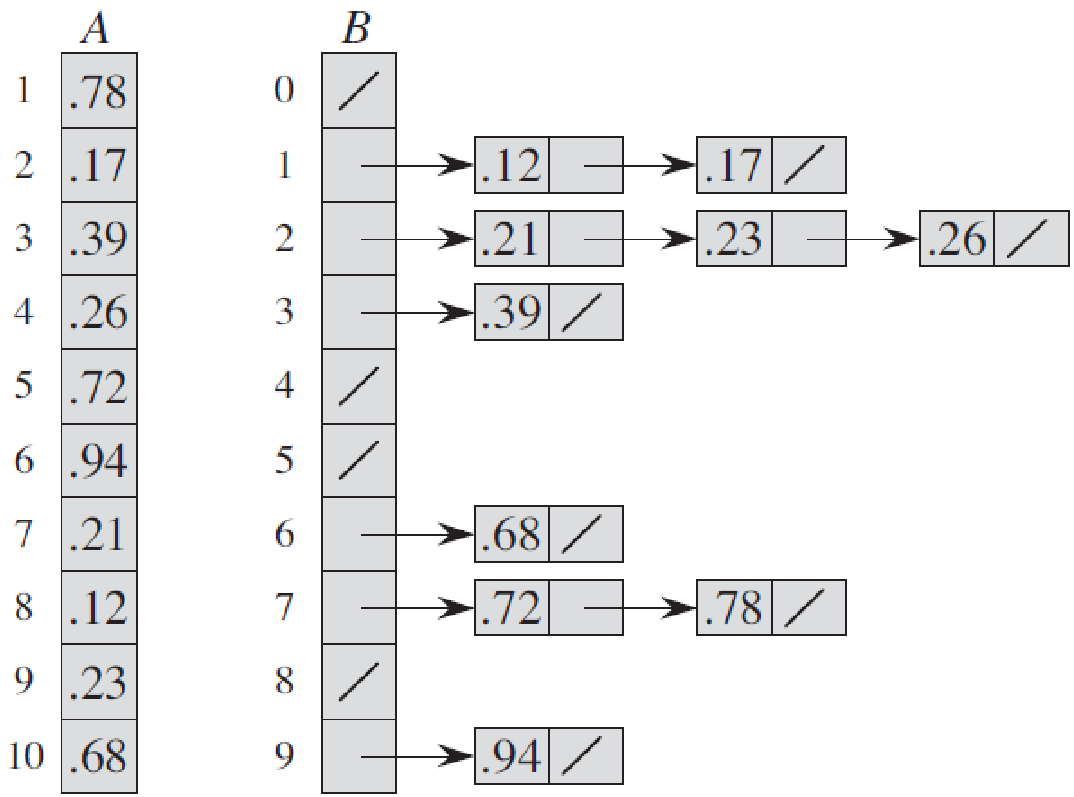
- As r decreases ($2^r \ll n$), $\frac{b}{r}$ increases and $(n+2^r)$ stays the same as $\Theta(n)$
- As r increases ($2^r \gg n$), $\frac{b}{r}$ decreases but $(n+2^r)$ increases exponentially.
- Logically, $(\mathrm{n}+2^r) = \Theta(n)$, then $r = \lfloor \log n \rfloor$

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right) = \Theta\left(\frac{bn}{\log n})\right)$$

Radix sort

- If $b = O(\log n)$, radix sort becomes $\Theta(n)$
- Counting sort is NOT in-place
- Quicksort is better
  - Better cache utilization
  - In-place

Bucket sort



(a) (b)

Bucket sort

BUCKET-SORT(A)

```
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Bucket sort

- When in put is uniformly distributed, the average case is $O(n)$
- It is fast, as counting-sort because it makes an assumption about the input
- RULE OF THUMB:
  – Extra assumption/extra information → room for optimization/customization

Order statistics

- The i^{th} *order statistic* of a set of n elements is the i^{th} *smallest element*

Order statistics

- Minimum:

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

- Requires $O(n)$, optimal as $n - 1$ comparisons are needed

Selection algorithm

- Find i^{th} *smallest element*:

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q - p + 1
if i == k            // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q - 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

- $O(n)$, prove it?

Selection algorithm

- On average, the left or right subarray that will be selected for next iteration will be of size $n/2$
- The recurrence becomes

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n)$$

- Using the master method with $a = 1$, $b = 2$, $f(n) = \Theta(n)$
- Thus, $O(n)$ on average