Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Jan. 18, 2023 (Lecture 3)

Data Structure Review (Continue)

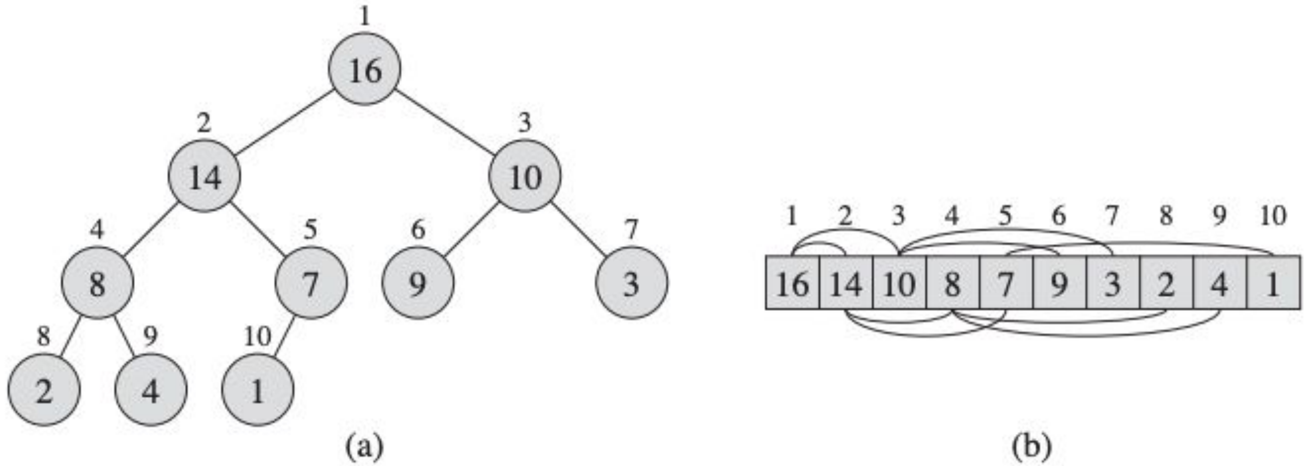
Topics

- Stack
- Queue
- Linked List
- Heap

Heap

- The (binary) heap data structure is an array that we can view as a nearly complete binary tree.
- An array *A* that represents a heap is an object with two attributes:
  - A.length which gives the number of elements in the array, and
  - A.heap-size, which represents how many elements in the heap are stored within array A.
  - Although A[1.. A.length] may contain numbers, only the elements in A[1.. A.heap-size], where 0<A.heap-size <= A.length, are valid elements of the heap. The root of the tree is A[1].

Heap



A max-heap viewed as (a) a binary tree and (b) an array.

The number within the circle at each node in the tree is the value stored at that node.

The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

The tree has height three; the node at index 4 (with value 8) has height one.

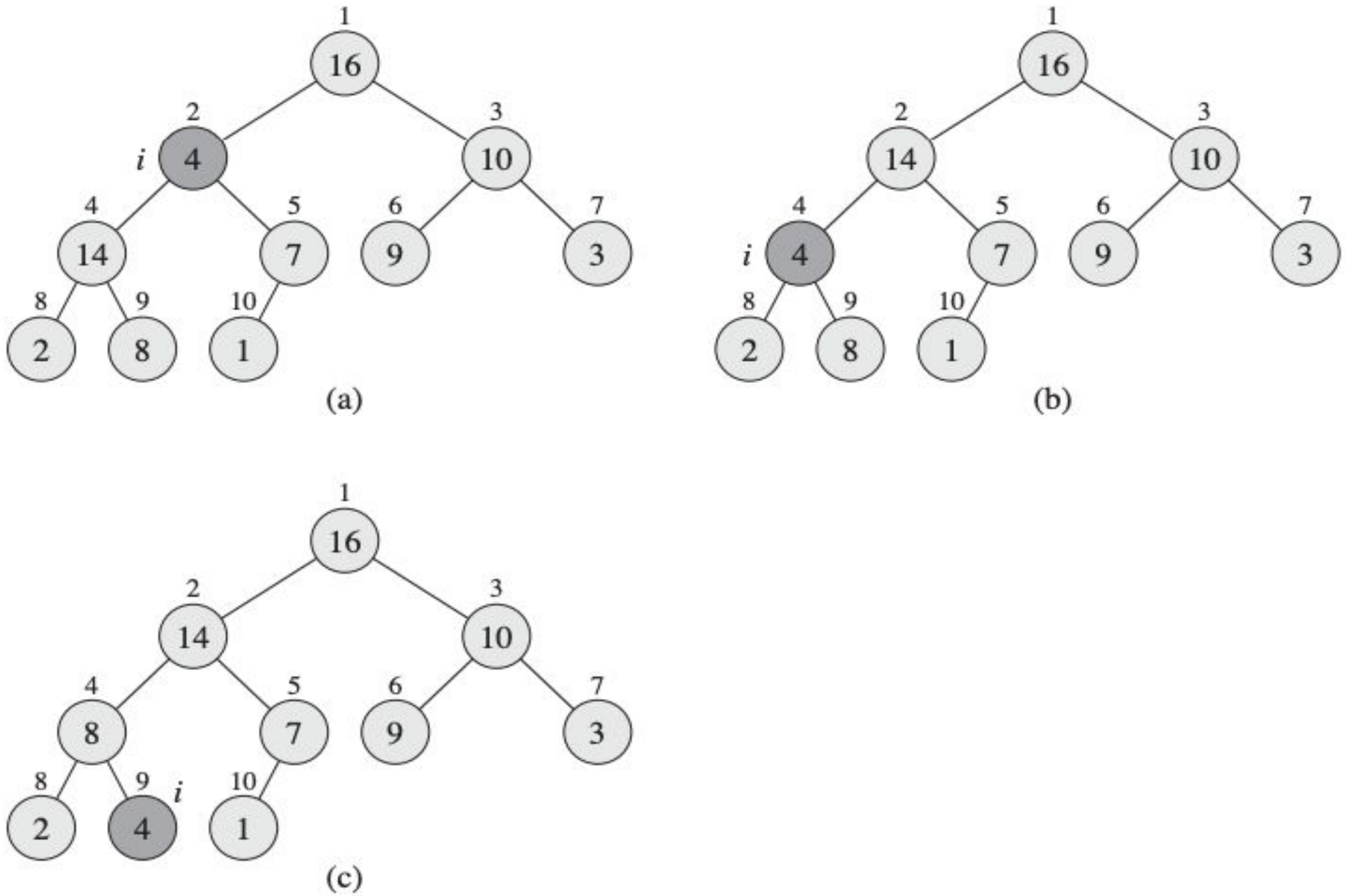
Notes for Heaps

- On most computers, it is very efficient to compute $2i$ or $\lfloor i/2 \rfloor$, just shift the binary representation of i left, or right, by one bit position.
- There are two kinds of binary heaps: max-heaps and min-heaps. In a max-heap, $A[Parent(i)] \geq A[i]$, while in a min-heap, $A[Parent(i)] \leq A[i]$ for any node A[i].
- Next we consider main operations (use max-heap as example).

MAX-HEAPIFY(A, i)

$l = $ LEFT(i)
$r = $ RIGHT(i)
if $l \leq A.\mathit{heap\text{-}size}$ and $A[l] > A[i]$
    $\mathit{largest} = l$
else $\mathit{largest} = i$
if $r \leq A.\mathit{heap\text{-}size}$ and $A[r] > A[\mathit{largest}]$
    $\mathit{largest} = r$
if $\mathit{largest} \neq i$
    exchange $A[i]$ with $A[\mathit{largest}]$
    MAX-HEAPIFY$(A, \mathit{largest})$

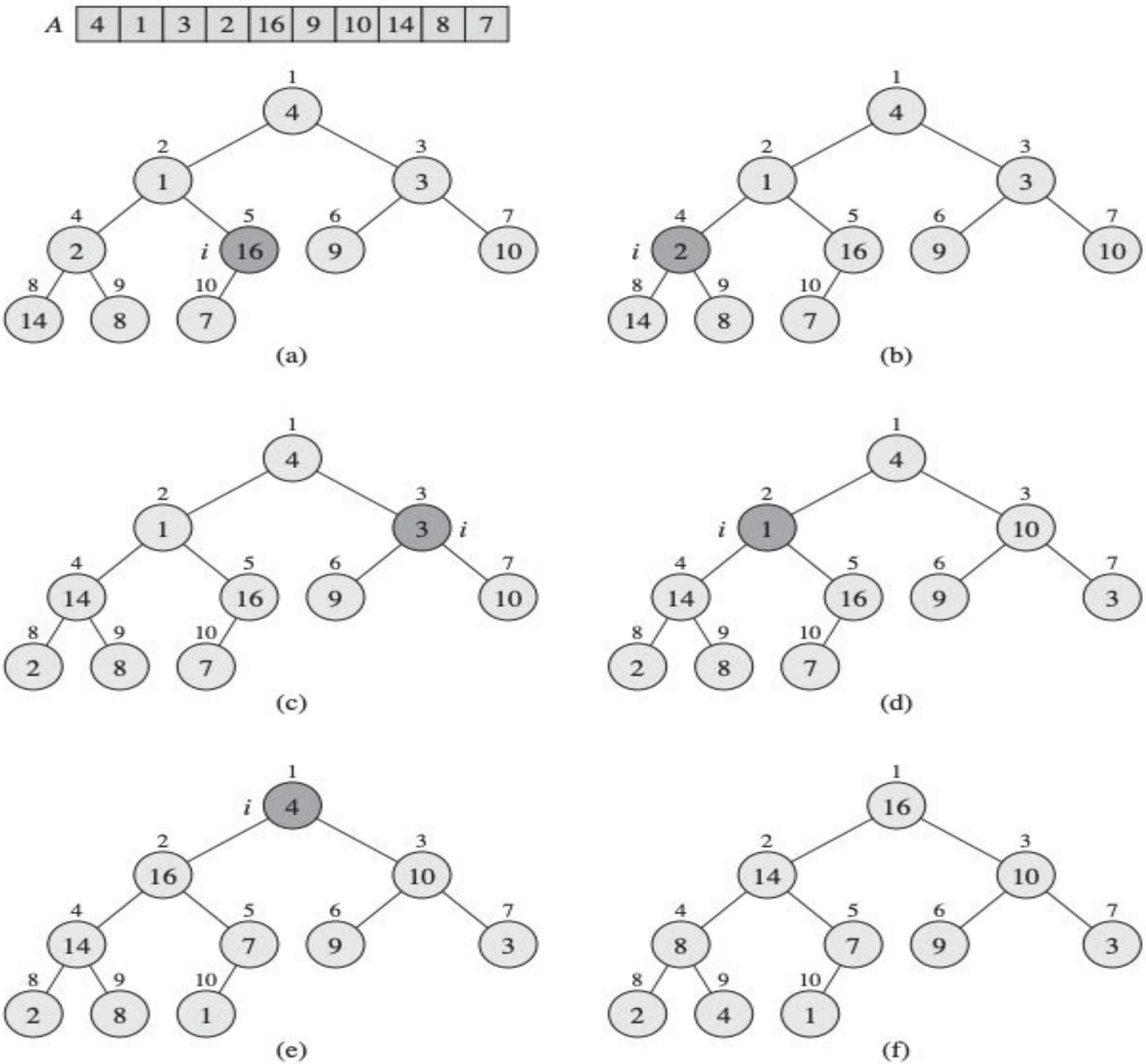
The running time for Max-Heapify is *O(log n),* where *n* is the heap size starting from index *i .*


(a)
(b)
(c)

Build Max Heaps

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A into a max-heap.A more careful discussion can show that the asymptotically running time is O(n). We omitted the proof here.

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


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

More Sorting Algorithms

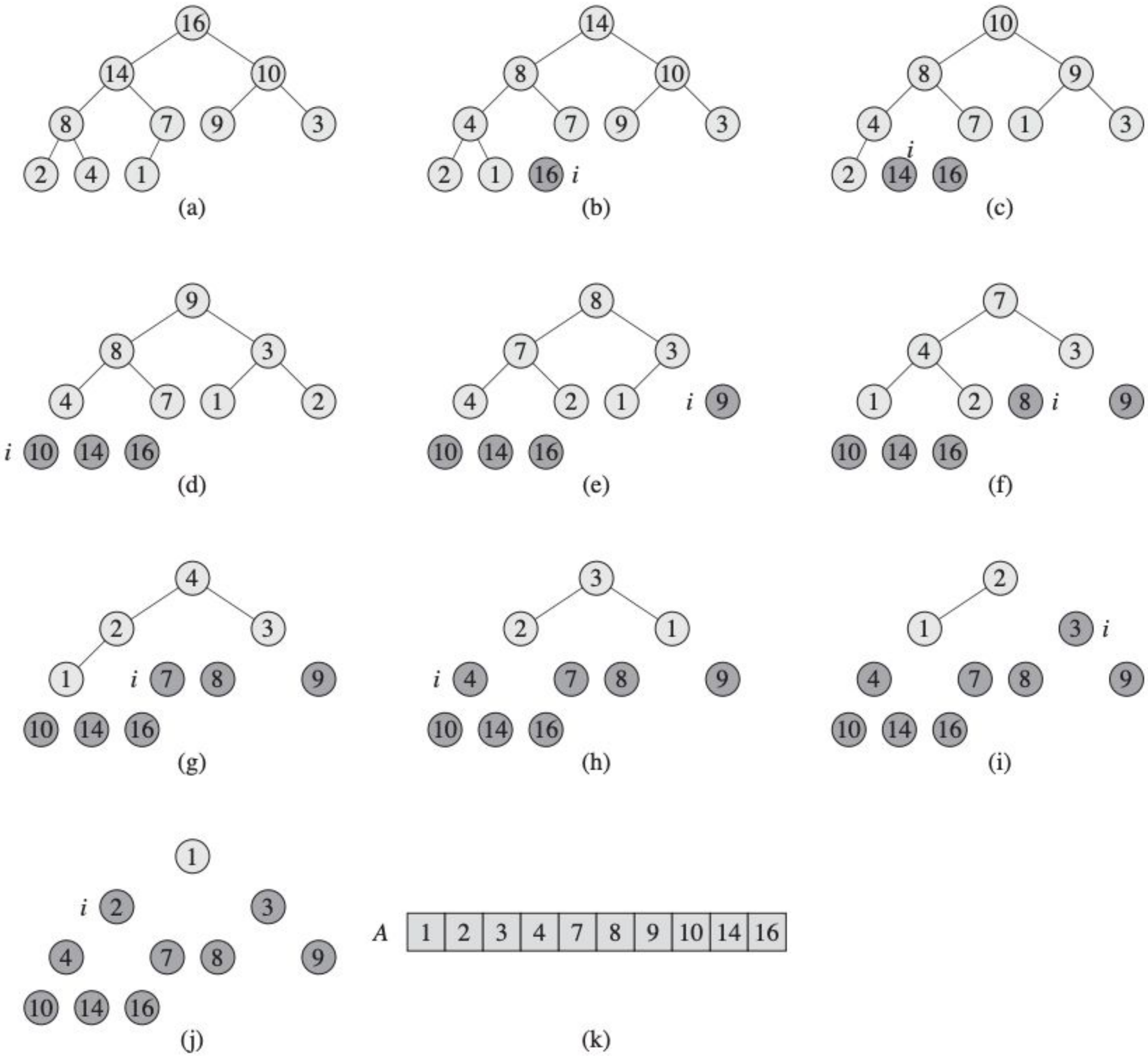
- Insertion Sort
- Merge Sort
- Selection Sort (homework)
- Heap Sort
- Quick Sort
- etc.

Heap Sort

Heap Sort

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A with length n. The Heapsort procedure takes time O(n log n).

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


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)

Application – priority queues

One important application for the heap is priority queues.

There are two types of priority queues: max-priority queues and min-priority queues. We use a max-priority queue to explain the main ideas. The min-priority queue is similar.

A max-priority queue maintains a set S and supports the following operations.

- **Insert(S, x)** inserts the element x into the set S, which is equivalent to the set operation $S = S \cup \{x\}$.
- **Maximum(S)** returns the element S with the largest key.
- **Extract-Max(S)** removes and returns the element of S with the largest key.
- **Increase-Key(S, x, k)** increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

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

The procedure HEAP-MAXIMUM has running time in Θ(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
```

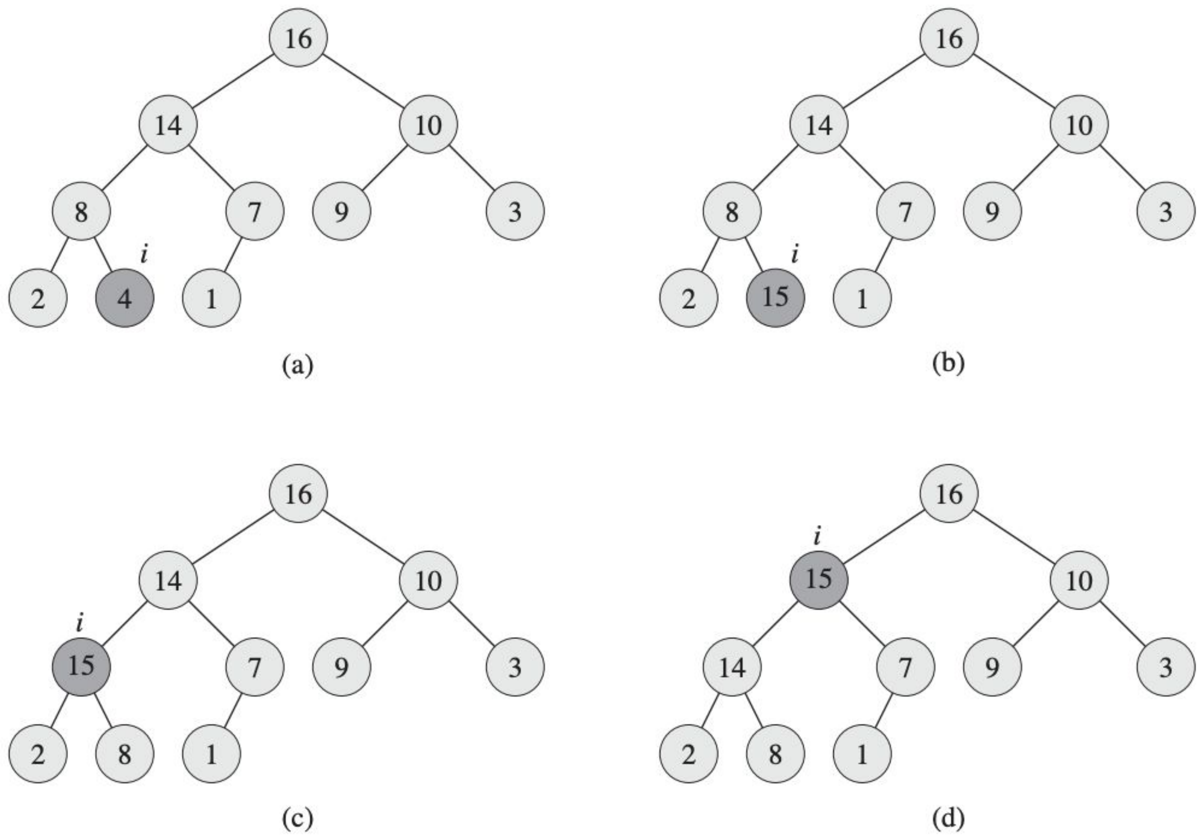
The procedure HEAP-EXTRACT-MAX has running time O(log n).

```
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)
```

The procedure HEAP-INCREASE-KEY has running time in O(log n).

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

The procedure MAX-HEAP-INSERT has running time O(log n).



Exercise: Illustrate the operation of MAX-HEAP-INSERT(A, 10) on the heap A = <15; 13; 9; 5; 12; 8; 7; 4; 0; 6; 2; 1>.

Quick Sort

Quick Sort

Quicksort is a divide-and-conquer algorithm.

Divide: Partition (rearrange) the array A[p..r] into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that each element of A[p.. q-1] is less than or equal to q, which is, in turn, less than or equal to each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort.

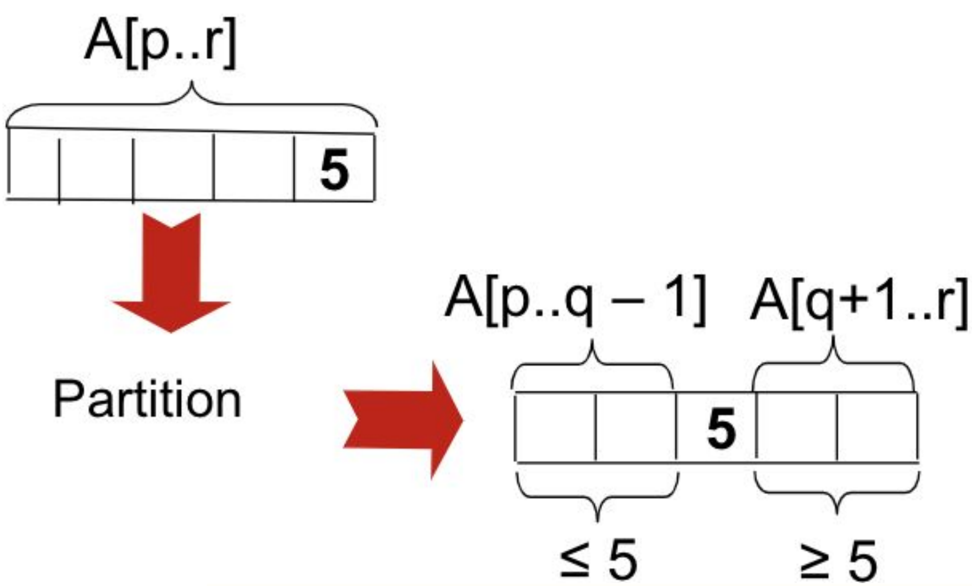
Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p..r] is now sorted.

How do the divide and combine steps of quicksort compare with those of merge sort?

Pseudocode of Quick Sort

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

To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).




```
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
```

An Example

initially:

p r
2 5 8 3 9 4 1 7 10 **6**
i j

note: pivot (x) = 6

next iteration:

2 5 8 3 9 4 1 7 10 **6**
i j

next iteration:

2 5 8 3 9 4 1 7 10 **6**
i j

next iteration:

2 5 8 3 9 4 1 7 10 **6**
i j

next iteration:

2 5 3 8 9 4 1 7 10 **6**
i j

```
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
```

An Example (Continue)

next iteration: 2 5 3 8 9 4 1 7 10 **6**

i j

next iteration: 2 5 3 8 9 4 1 7 10 **6**

i j

next iteration: 2 5 3 4 9 8 1 7 10 **6**

i j

next iteration: 2 5 3 4 1 8 9 7 10 **6**

i j

next iteration: 2 5 3 4 1 8 9 7 10 **6**

i j

next iteration: 2 5 3 4 1 8 9 7 10 **6**

i j

after final swap: 2 5 3 4 1 **6** 9 7 10 8

i j

```
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
```

Computational complexity is $\Theta(n)$, where $n = r - p + 1$

Alternative Way of Partition

1. Choose an array value (say, the first) to use as the pivot;
2. Starting from the left end, find the first element that is greater than or equal to the pivot;
3. Searching backward from the right end, find the first element that is less than the pivot;
4. Interchange (swap) these two elements;
5. Repeat, searching from where we left off, until done;

Alternative Way of Partition

```
Partition(A, left, right)
    pivot = a[left], l = left + 1, r = right;
    while l < r, do
        while (l < right) and (A[l] < pivot)
            l = l + 1
        while (r > left) and (A[r] >= pivot)
            r = r - 1
        if (l < r) swap A[l] and A[r]
    Swap A[left] and A[r]
    return r
```

Complexity is $\Theta(n)$, where $n = right - left + 1$

Alternative Way of Partition

v a[]

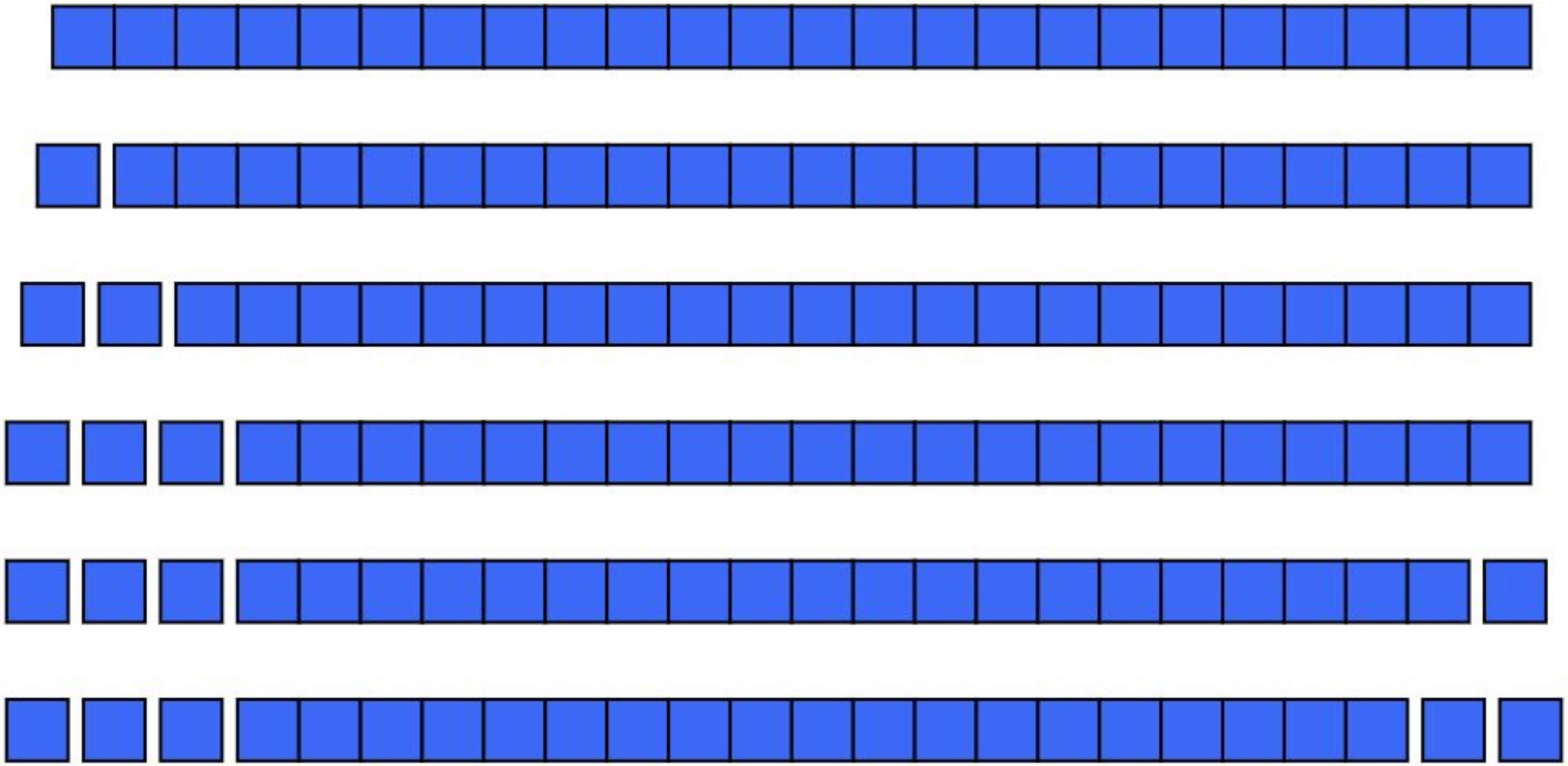
	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Performance of Quick Sort

Worst Case:

- In the worst case, partitioning always divides the size `n` array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length `n-1` part, containing everything else
- We don’t recur on the zero-length part
- Recurring on the length `n-1` part requires (in the worst case) recurring to depth `n-1`

Worst Case



Worst Case

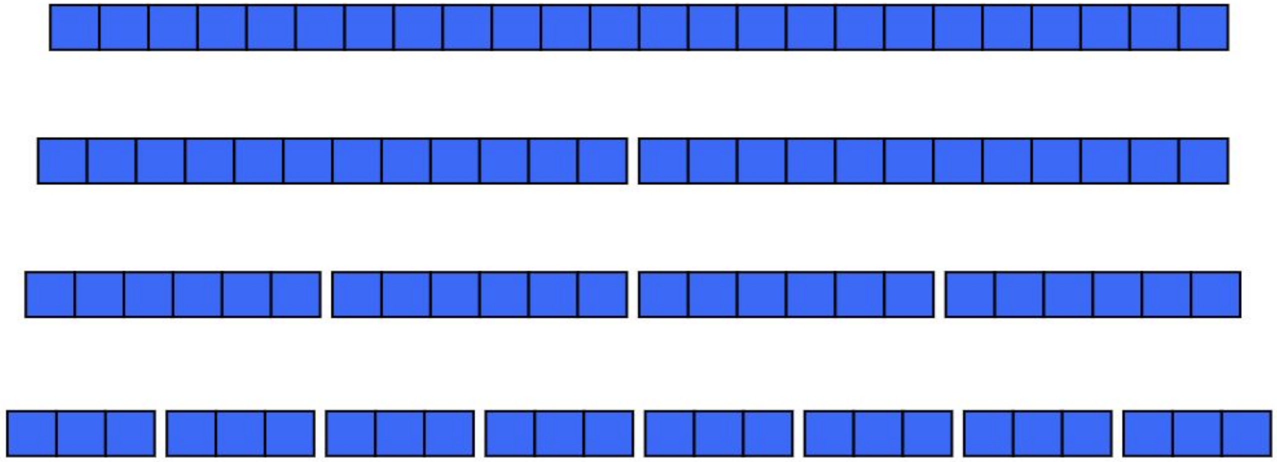
$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

- By using substitution method, we can prove that

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

Best Case of Quick Sort

- Cut the array size in half each time
- So the depth of the recursion in lg n
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $T(n) = 2T(n/2) + \Theta(n)$, that is $T(n) = \Theta(n \lg n)$.



Master Theorem Method

The master theorem

The master method depends on the following theorem.

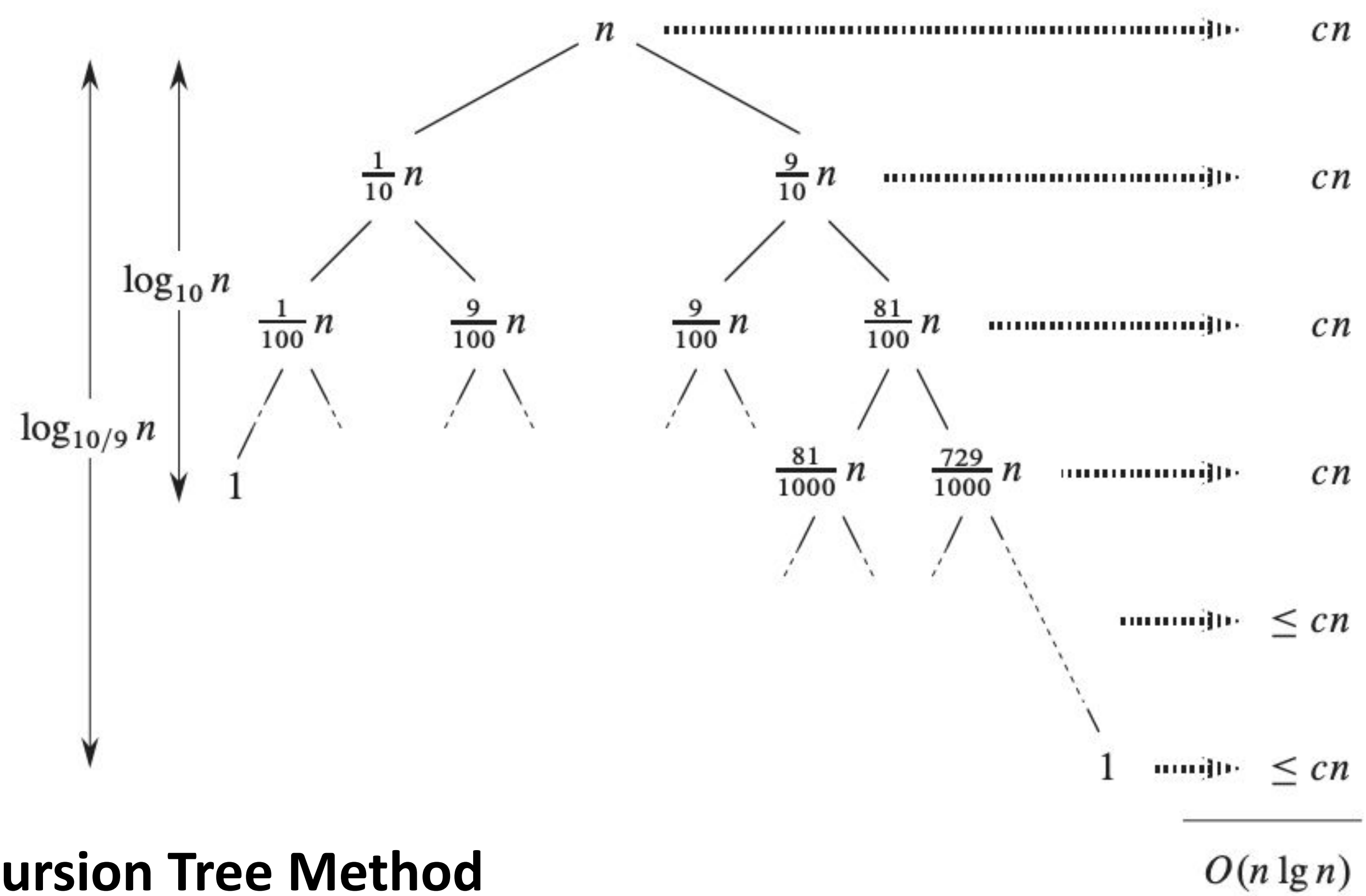
Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Average Case of Quick Sort



Recursion Tree Method

Randomized Version of Quicksort

RANDOMIZED-PARTITION(A, p, r)

$i = \text{RANDOM}(p, r)$
exchange $A[r]$ with $A[i]$
return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)

if $p < r$
    $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
    RANDOMIZED-QUICKSORT$(A, p, q - 1)$
    RANDOMIZED-QUICKSORT$(A, q + 1, r)$

Key point: randomize to obtain good expected performance over all inputs.

Summary of Sorting Algorithms

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

After Class

Read: Part II Chapter 6 and 7