Unit 2: Sorting and Order Statistics

- Course contents:
  - Heapsort
  - Quicksort
  - Sorting in linear time
  - Order statistics
- Readings:
  - Chapters 6, 7, 8, 9

Comparison-based sorters				
Algorithm	Runtime			In-place?
	Best case	Average case	Worst case	
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Yes
Merge	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	No
Heap	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	Yes
Quicksort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	Yes
Non-comparison-based sorters				
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	No
Radix	$O(d(n+k'))$	$O(d(n+k'))$	$O(d(n+k'))$	No
Bucket	-	$O(n)$	-	No

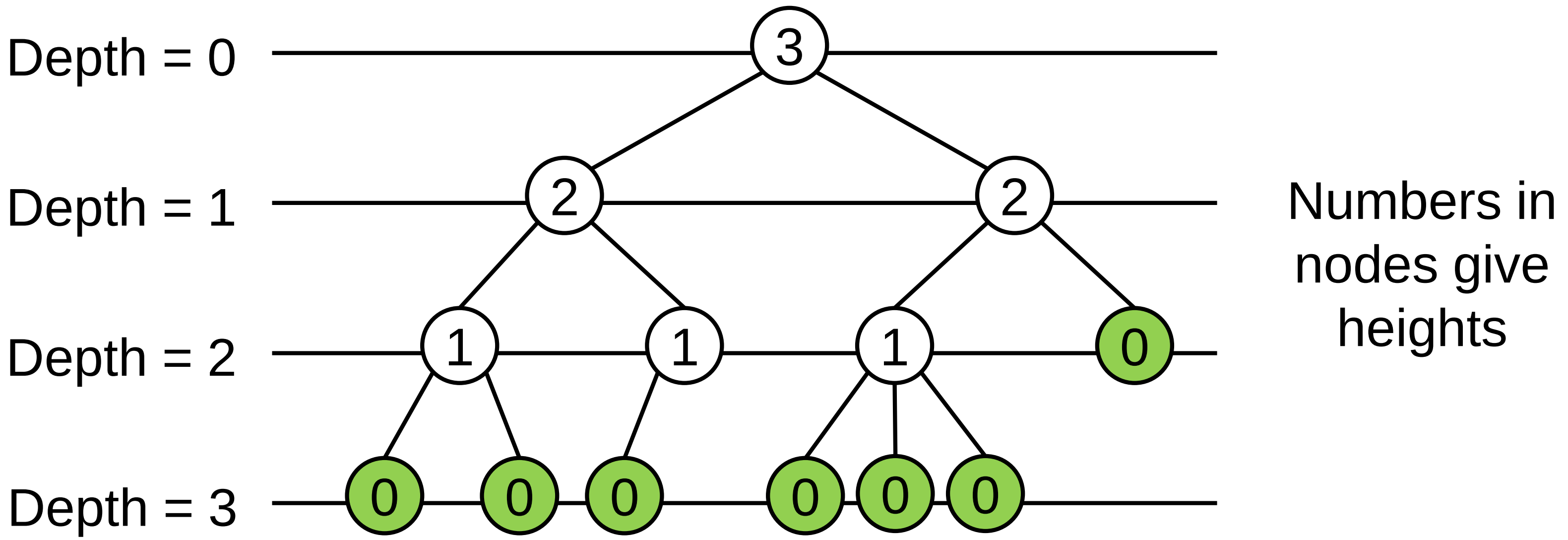
Types of Sorting Algorithms

- A sorter is **in-place** if only a constant # of elements of the input are ever stored outside the array.
- A sorter is **comparison-based** if the only operation on keys is to **compare two keys**.
  - Insertion sort, merge sort, heapsort, quicksort
- The non-comparison-based sorters sort keys by looking at the values of **individual** elements.
  - **Counting sort:** Assumes keys are in [1..k] and uses array indexing to count the # of elements of each value.
  - **Radix sort:** Assumes each **integer** contains d digits, and each digit is in [1..k'].
  - **Bucket sort:** Sort data into buckets and then merge across buckets. Requires information for input **distribution**.

Heap Sort

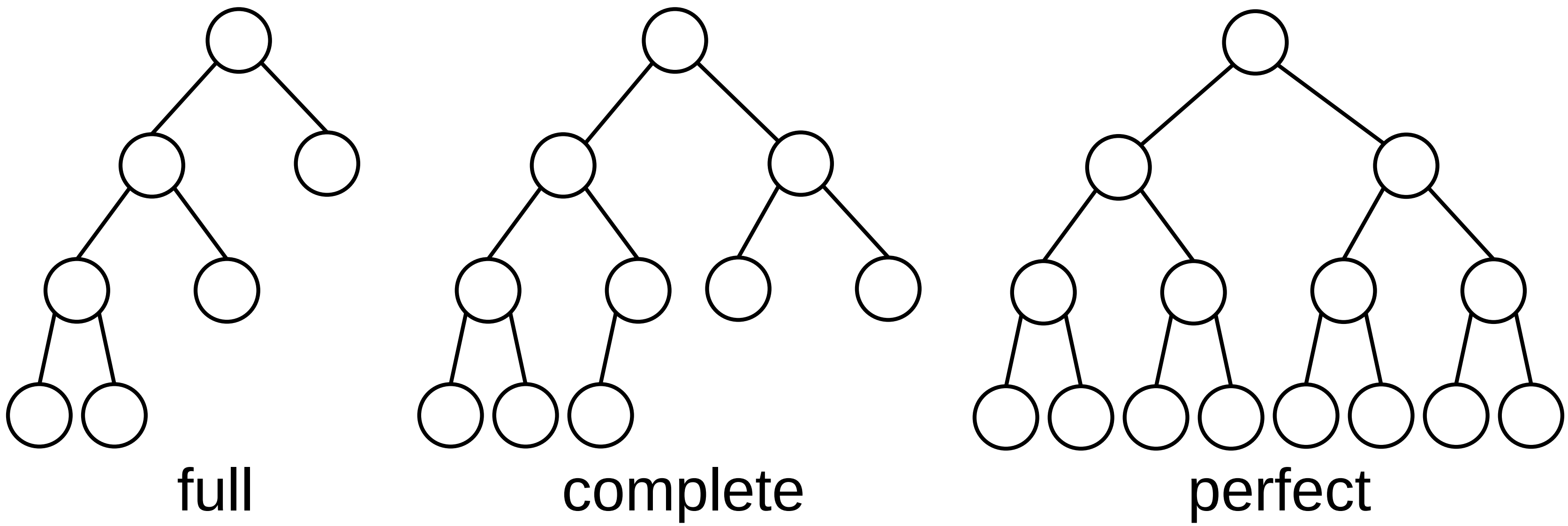
Tree Height and Depth

- **Height** of a node *u*: Length of the longest path from *u* to a leaf.
- **Depth** of a node *u*: Length of the path from the root to *u*
- **Height of a tree**: maximum depth of its nodes.
- A **level** is the set of all nodes at the same depth.



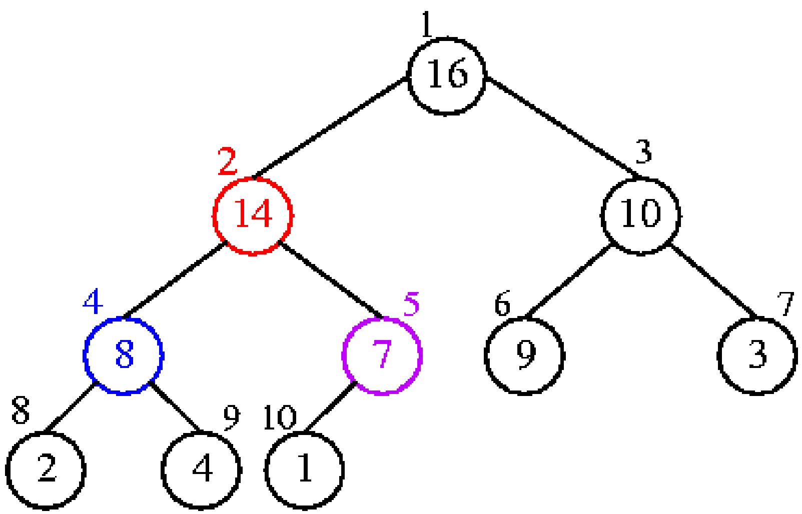
Binary Trees

- A **full** binary tree: every node has either 0 or 2 children.
- A **complete** binary tree: the lowest *d-1* levels of a binary tree of height *d* are filled and level *d* is partially filled from left to right.
- A **perfect** binary tree: all *d* levels of a height-*d* binary tree are filled.



Binary Heap

- Binary heap data structure: represented by an array *A*
  - Complete binary tree.
  - **Max-Heap property:** A node's key ≥ its children's keys.
  - **Min-Heap property:** A node's key ≤ its children's keys.

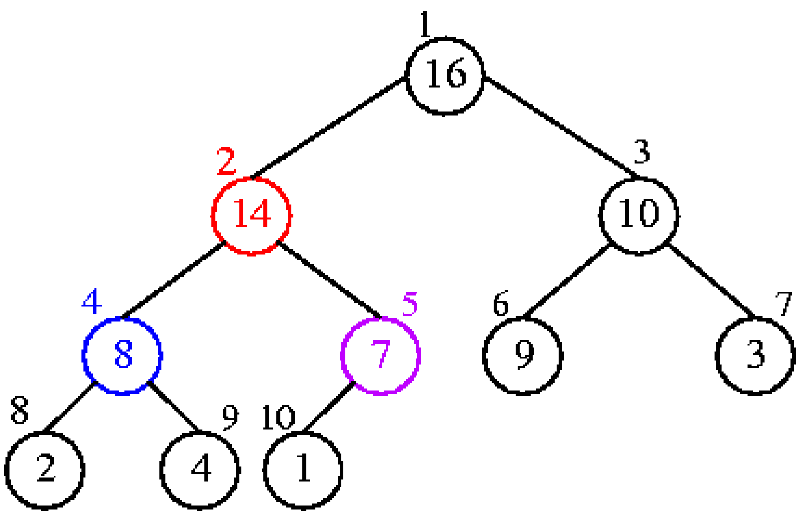


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

array representation

Binary Heap (cont'd)

- Implementation
  - Root: *A*[1].
  - For *A*[*i*], LEFT child is *A*[2*i*], RIGHT child is *A*[2*i*+1], and PARENT is *A*[$\lfloor i/2 \rfloor$].
  - *A.heap-size* (# of elements in the heap stored within *A*) ≤ *A.length* (# of elements in *A*).

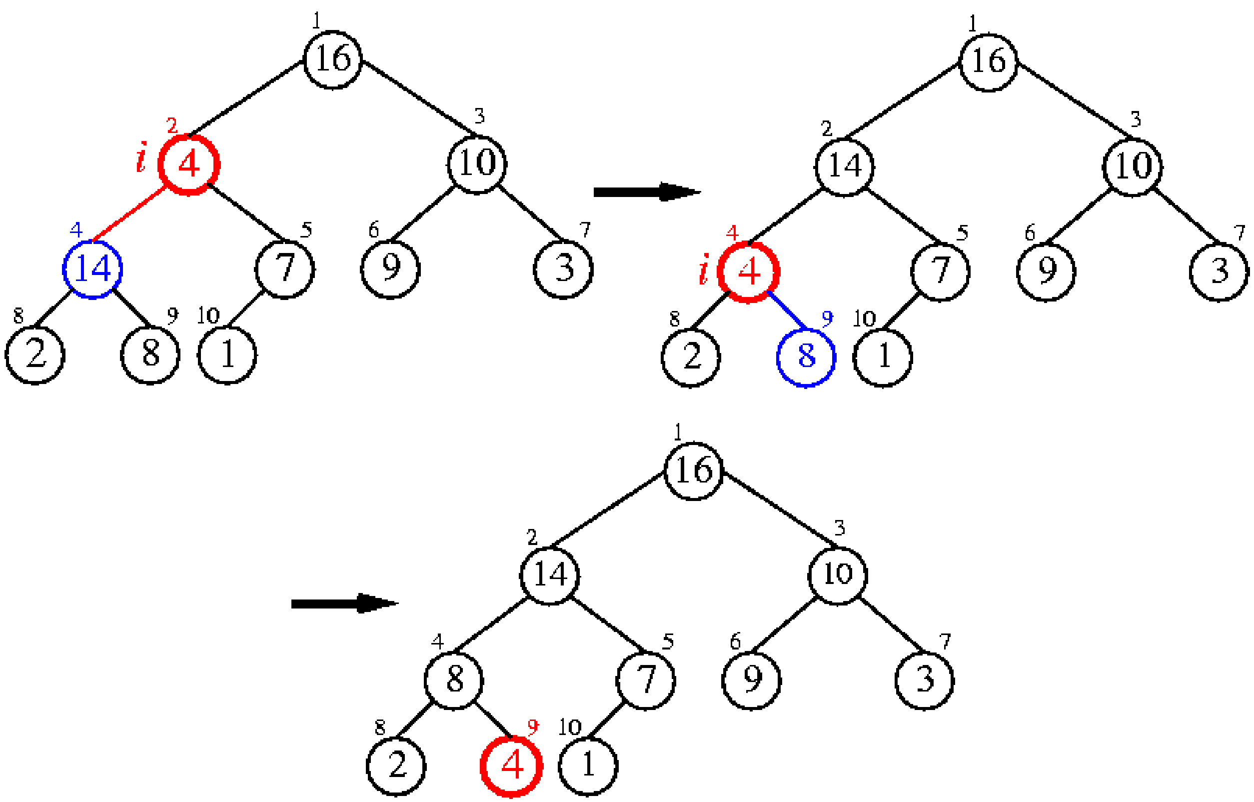


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

array representation

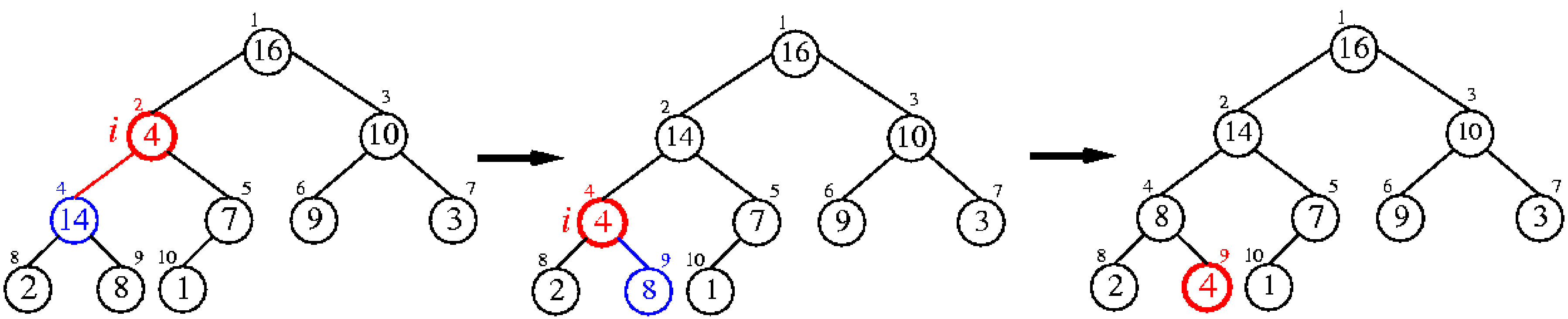
MAX-HEAPIFY: Maintaining the Heap Property

- Assume that subtrees indexed *RIGHT*(*i*) and *LEFT*(*i*) are heaps, but *A*[*i*] may be smaller than its children.
- MAX-HEAPIFY(*A*, *i*) will "float down" the value at *A*[*i*] so that the subtree rooted at *A*[*i*] becomes a heap.



MAX-HEAPIFY: Algorithm

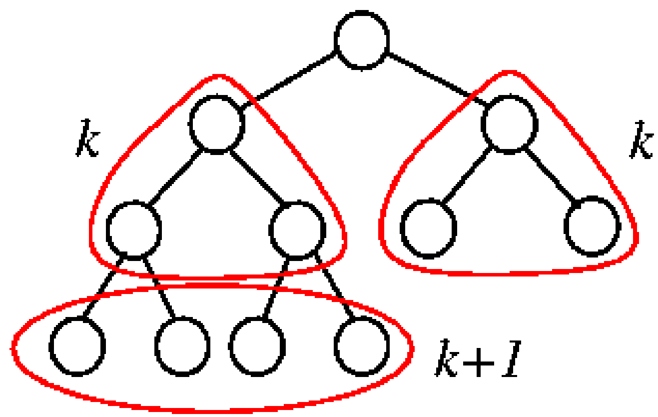
```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```



MAX-HEAPIFY: Complexity

```
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)
```

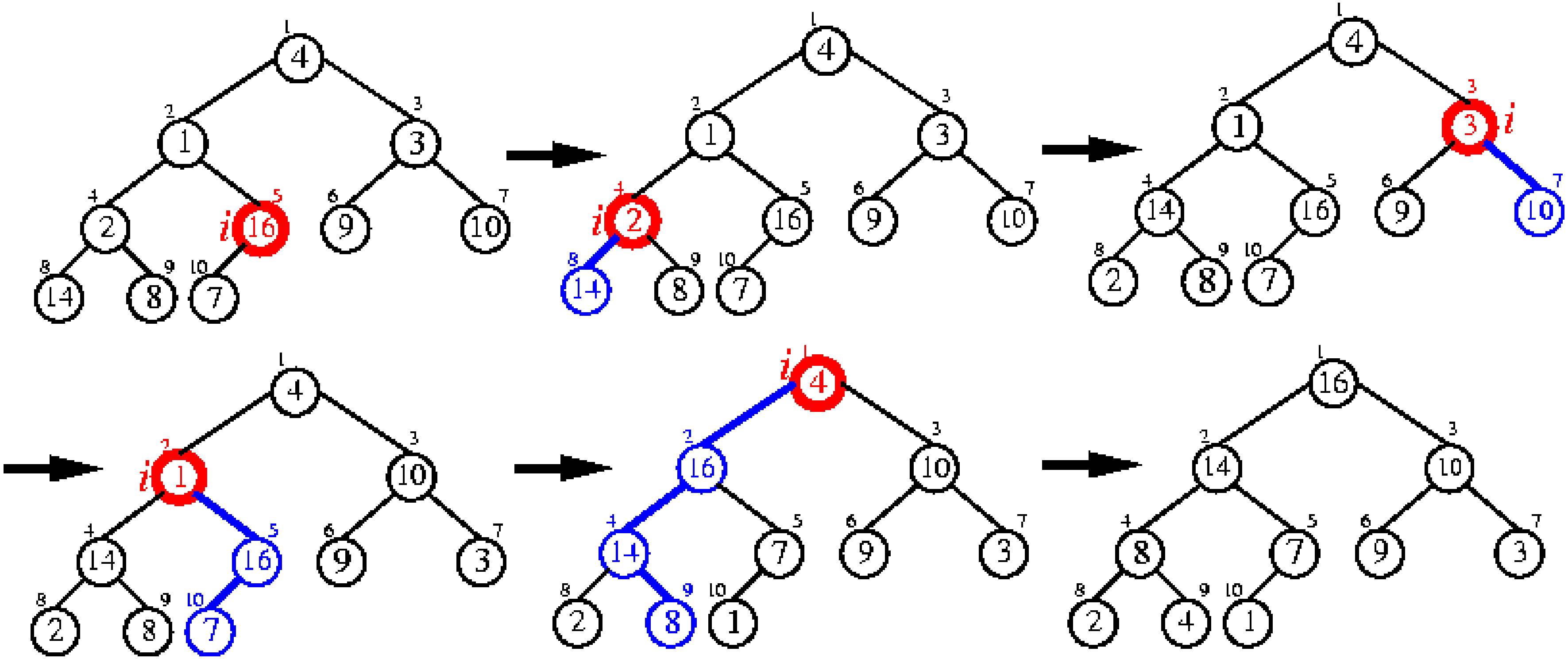
- Worst case: last row of binary tree is half empty $\Rightarrow$ children's subtrees have size $\leq 2n/3$.
- Recurrence: $T(n) \leq T(2n/3) + \theta(1)$ $\Rightarrow T(n) = O(\lg n)$



BUILD-MAX-HEAP: Building a Max-Heap

- Intuition: Use MAX-HEAPIFY in a bottom-up manner to convert *A* into a heap.
  – Leaves are already heaps, start at parents of leaves, and work upward till the root.

A									
4	1	3	2	16	9	10	14	8	7



BUILD-MAX-HEAP: Algorithm

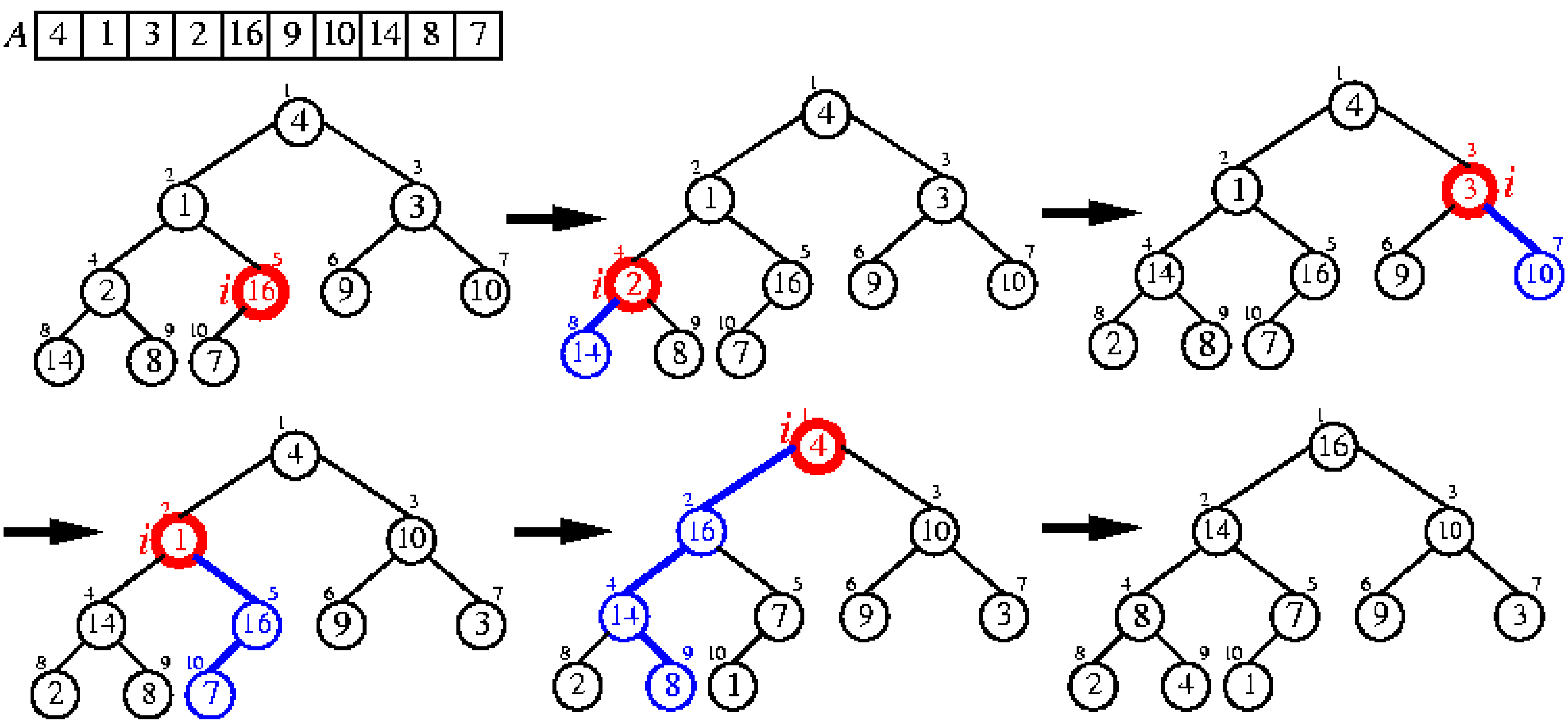
BUILD-MAX-HEAP(A)

$A.heap\text{-}size = A.length$

for $i = \lfloor A.length/2 \rfloor$ **downto** 1

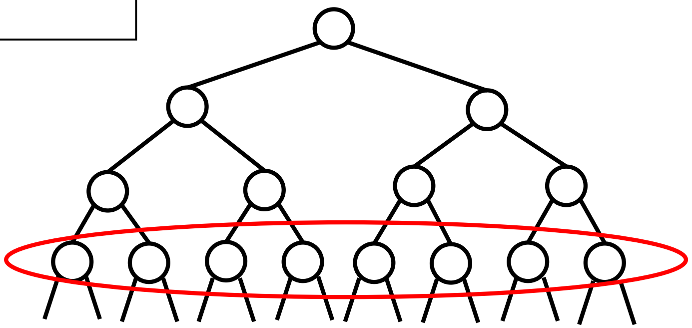
    MAX-HEAPIFY(A,i)

The number of non-leaf nodes in a complete binary tree of n nodes is $\lfloor n/2 \rfloor$.



BUILD-MAX-HEAP: Complexity

BUILD-MAX-HEAP(*A*)
1. *A.heap-size* = *A.length*
2. **for** *i* = $\lfloor A.length/2 \rfloor$ **downto** 1
3. MAX-HEAPIFY(*A*,*i*)



#nodes of height 0: $\lceil 15/2^{0+1} \rceil = 8$

- Naive analysis: $O(n \lg n)$ time in total.
  - About $n/2$ calls to HEAPIFY.
  - Each takes $O(\lg n)$ time.
- Careful analysis: $O(n)$ time in total.
  - Each MAX-HEAPIFY takes $O(h)$ time (h: tree height).
  - At most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h in an n-element array.
  - $T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} (\#nodes\ of\ height\ h) O(h) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) =$

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

$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} < 2$, since $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

Let $x = \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\mathbf{0}}{2^0} + \frac{\mathbf{1}}{2^1} + \frac{\mathbf{2}}{2^2} + \frac{\mathbf{3}}{2^3} + \cdots$ (1)

$$\frac{1}{2}x = \frac{0}{2^1} + \frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \cdots \quad (2)$$

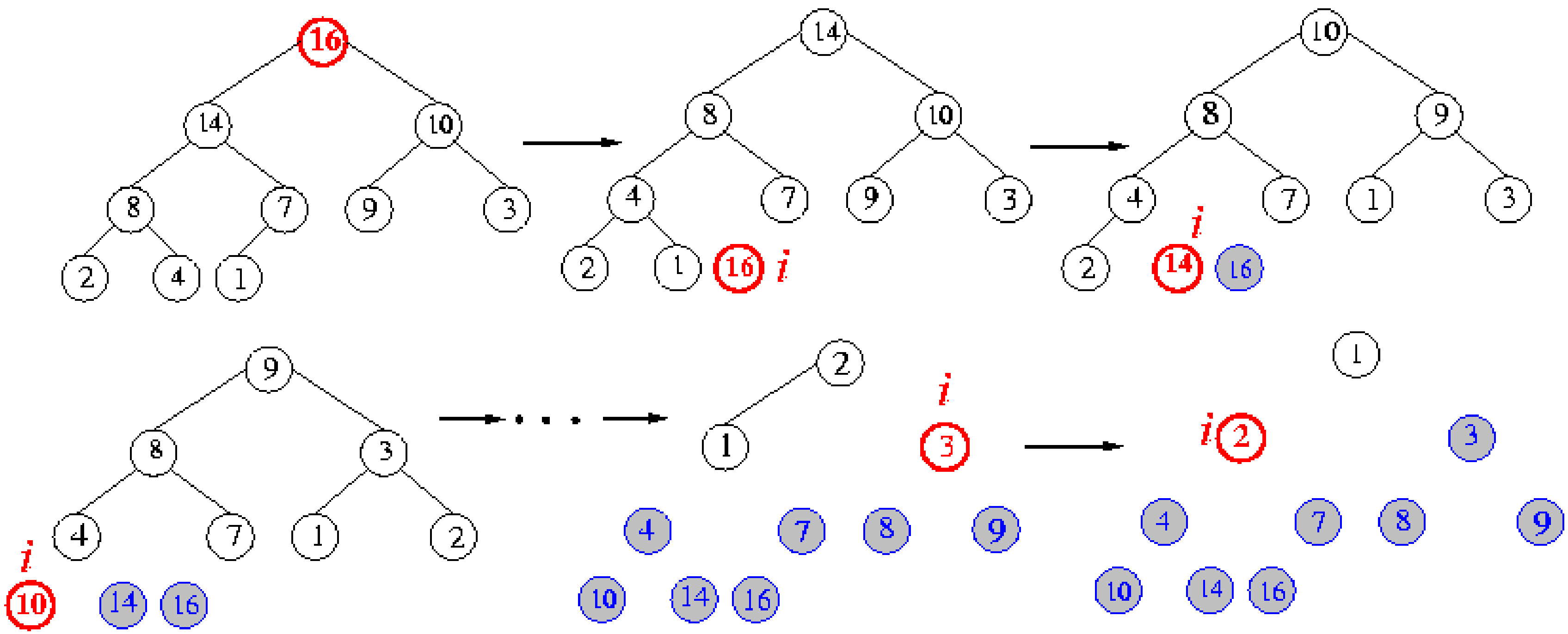
(1) – (2): $\frac{1}{2}x = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \cdots$

$$\frac{1}{2}x = \frac{a_0}{1-r} = \frac{\frac{1}{2}}{1-\frac{1}{2}} = 1$$

Thus, $x = \sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

HEAPSORT: Algorithm

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



A									
1	2	3	4	7	8	9	10	14	16

HEAPSORT: Complexity

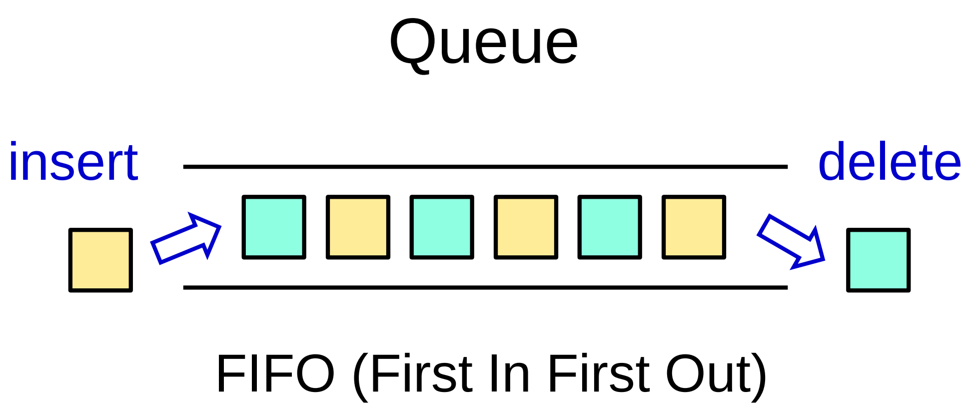
HEAPSORT(*A*)		
1.	BUILD-MAX-HEAP(*A*)	O(n)
2.	**for** *i* = *A.length* **downto** 2	
3.	exchange *A*[1] with *A*[*i*]	O(1)
4.	*A*.*heap-size* = *A*.*heap-size* – 1	O(1)
5.	MAX-HEAPIFY(*A*,1)	O(lg*n*)

- Time complexity: $O(n \lg n)$.
- Space complexity: $O(n)$ for array, **in-place.** **(Stable??)**

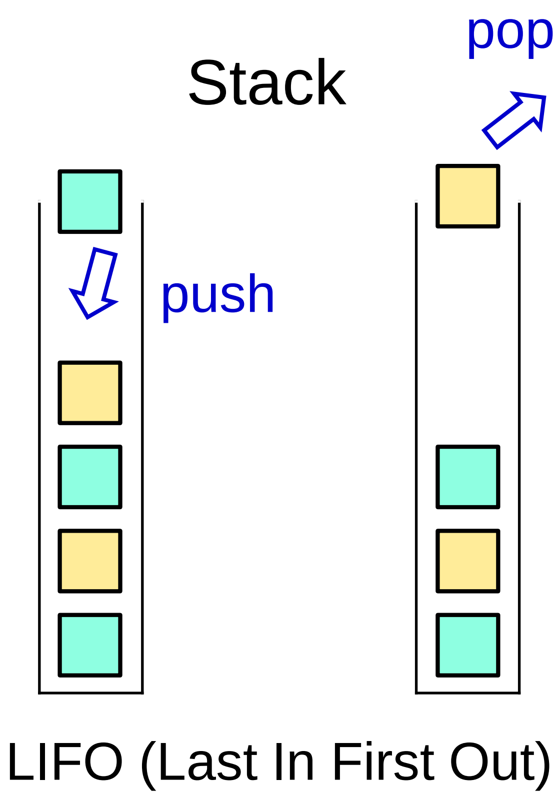
Priority Queues

- A **priority queue** is a data structure on sets of keys; a max-priority queue supports the following operations:
  - INSERT(S, x): insert x into set S.
  - MAXIMUM(S): return the largest key in S.
  - EXTRACT-MAX(S): return and **remove** the largest key in S.
  - INCREASE-KEY(S, x, k): increase the value of element x's key to the new value k.
- These operations can be easily supported using a heap.
  - INSERT: Insert the node at the end and fix heap in $O(\lg n)$ time.
  - MAXIMUM: read the first element in $O(1)$ time.
  - INCREASE-KEY: traverse a path from the target node toward the root to find a proper place for the new key in $O(\lg n)$ time.
  - EXTRACT-MAX: delete the 1st element, replace it with the last, decrement the element counter, then heapify in $O(\lg n)$ time.
- Compare with an array?

Queue & Stack

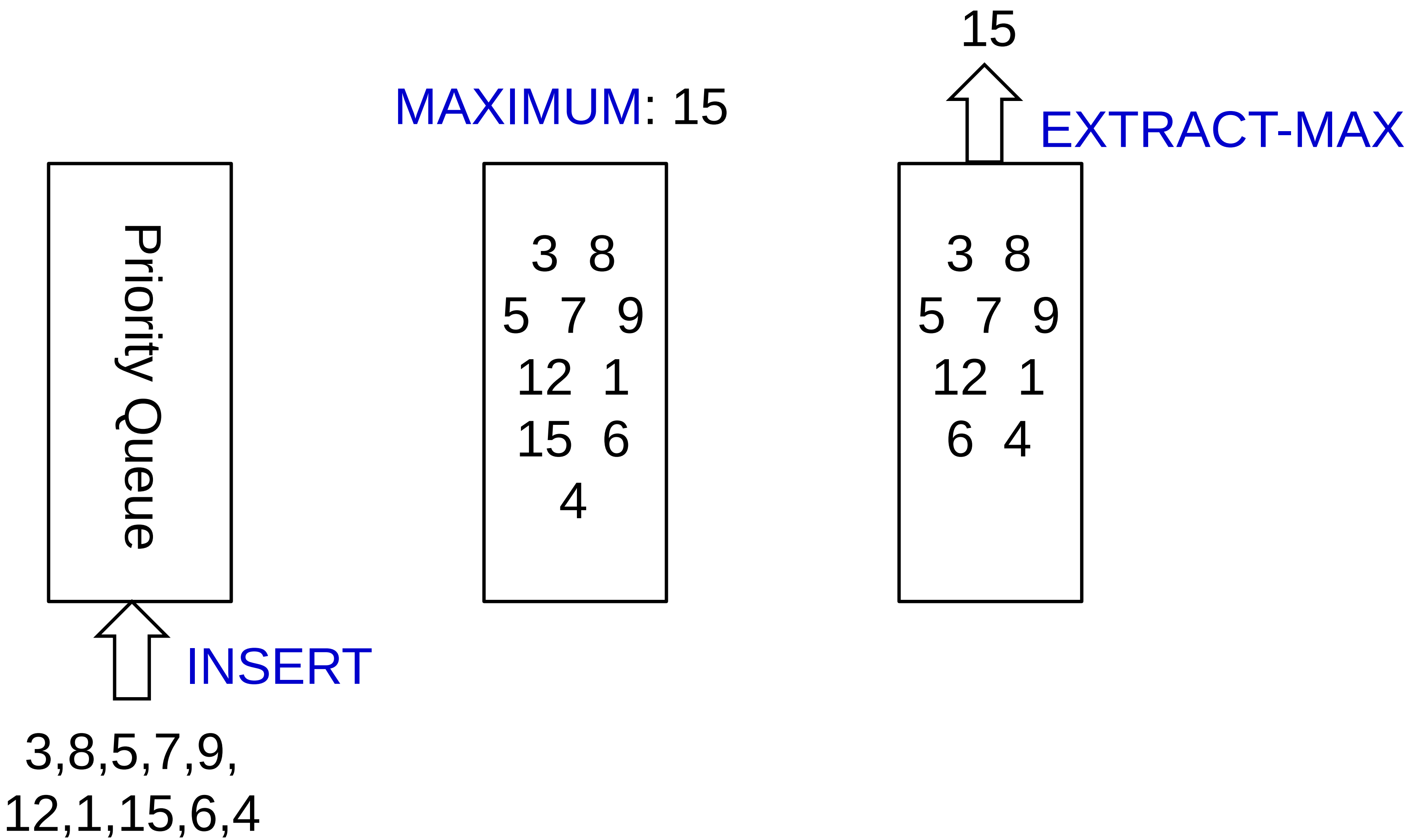


Queue
insert
delete
FIFO (First In First Out)



Stack
pop
push
LIFO (Last In First Out)

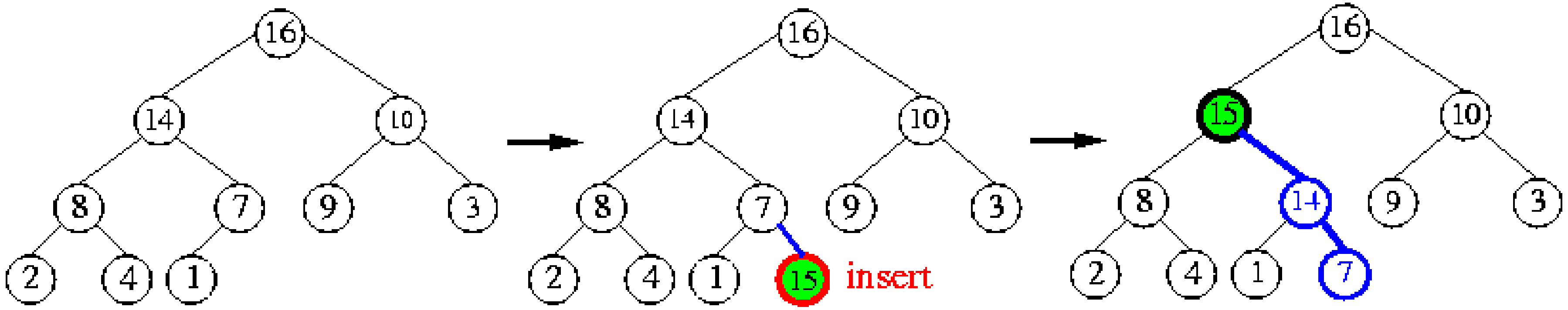
Priority Queue



Heap: EXTRACT-MAX and INSERT

HEAP-EXTRACT-MAX(*A*)
1. **if** *A*.*heap-size* **<** 1
2.   **error** "heap underflow"
3. *max* = *A*[1]
4. *A*[1] = *A*[*A*.*heap-size*]
5. *A*.*heap-size* = *A*.*heap-size* -1
6. MAX-HEAPIFY(*A*,1)
7. **return** *max*

MAX-HEAP-INSERT(*A*,*key*)
1. *A*.*heap-size* = *A*.*heap-size* + 1
2. *i* = *A*.*heap-size*
3. **while** *i* **>** 1 and *A*[*PARENT*(*i*)] **<** *key*
4.   *A*[*i*] = *A*[*PARENT*(*i*)]
5.   *i* = *PARENT*(*i*)
6. *A*[*i*] = *key*



Quicksort

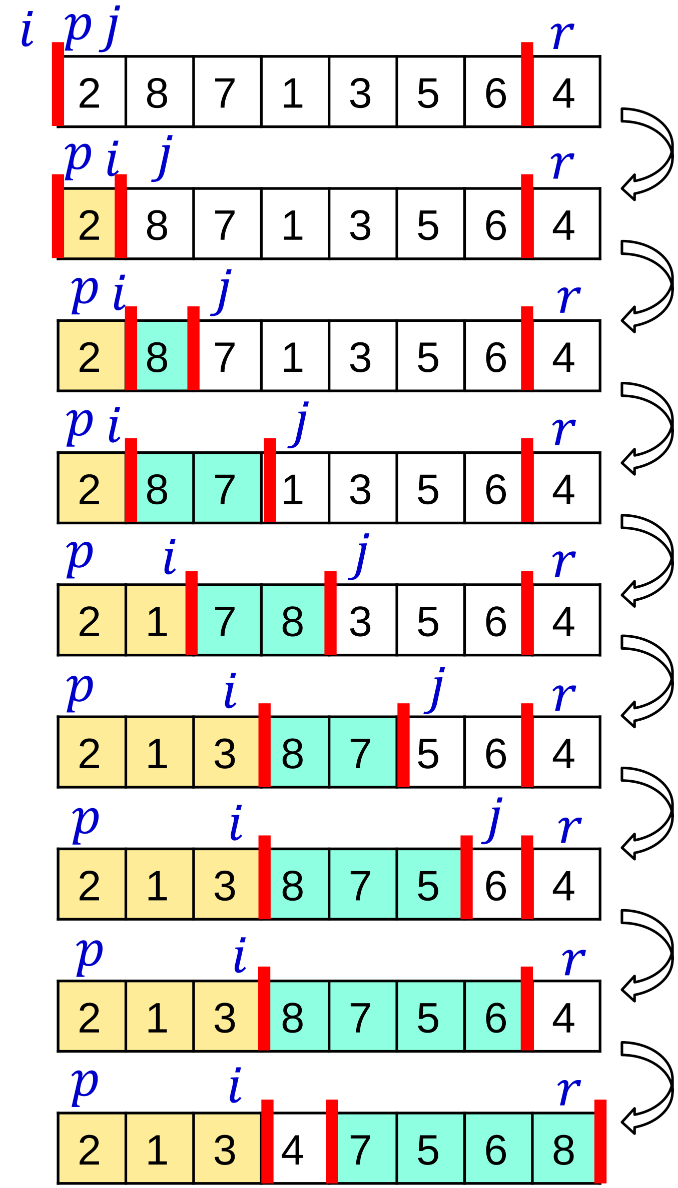
Quicksort

- A divide-and-conquer algorithm
  - **Divide:** Partition $A[p..r]$ into $A[p..q-1]$ and $A[q+1..r]$; each key in $A[p..q-1] \leq$ each key in $A[q+1..r]$.
  - **Conquer:** Recursively sort two subarrays.
  - **Combine:** Do nothing; quicksort is an in-place algorithm.

```
QUICKSORT(A, p, r)
 // Call QUICKSORT(A, 1, A.length) to sort an entire array
1.   if p < r
2.       q = PARTITION(A, p, r)
3.       QUICKSORT(A, p, q-1)
4.       QUICKSORT(A, q+1, r)
```

Quicksort: Partition

```
PARTITION(A, p, r)
x = A[r] /* break up A wrt x */
i = p - 1
for j = p to r -1
    if A[j] ≤ x then
        i = i +1
        exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i+1
```

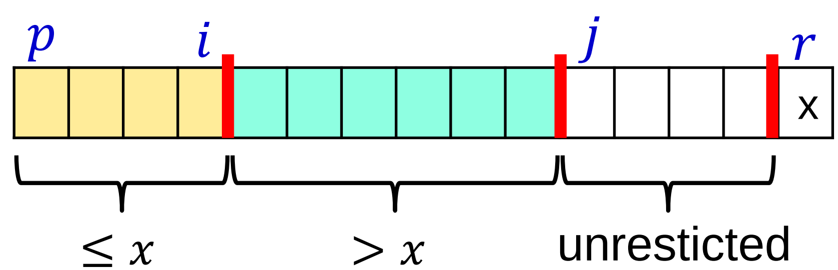


- Partition A into two subarrays $A[j] \leq x$ and $A[i] \geq x$.
- PARTITION runs in $\theta(n)$ time, where $n = r - p + 1$.
- Ways to pick x: always pick $A[r]$, pick a key at random, pick the median of several keys, etc.
- There are several partitioning variants

Loop Invariant of Partition

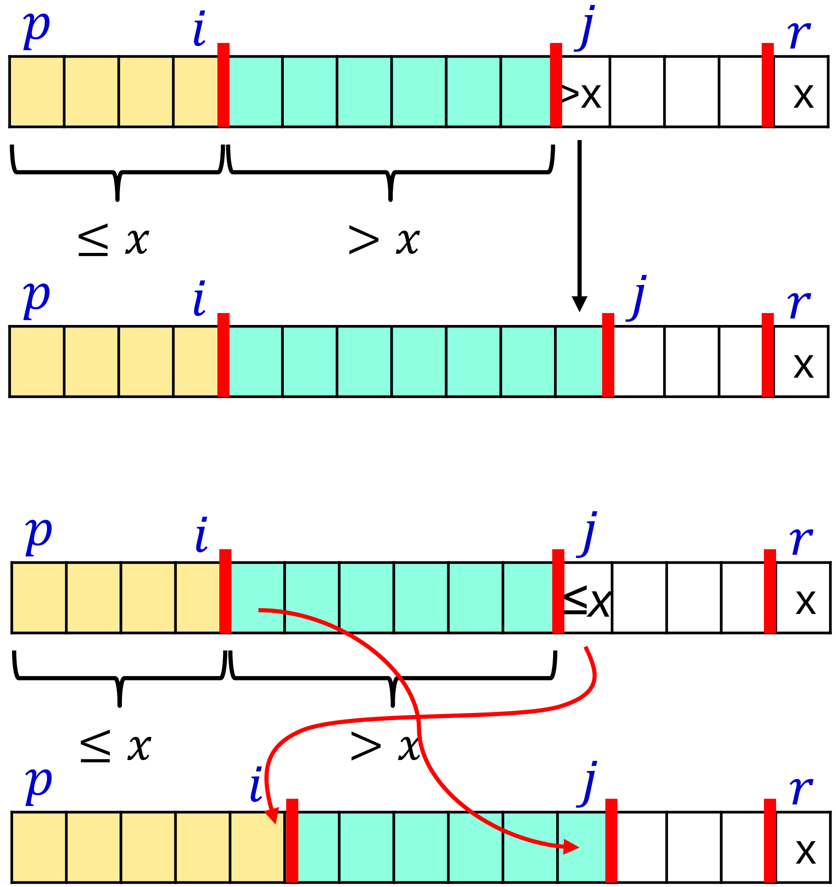
PARTITION(A, p, r)
1. $x = A[r]$ /* break up A wrt x */
2. $i = p - 1$
3. **for** $j = p$ **to** $r - 1$
4.   **if** $A[j] \leq x$ **then**
5. $i = i + 1$
6.   exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. **return** $i+1$

- At the beginning of each iteration of the loop of lines 3—6, for any array index k,
  - 1. if $p \leq k \leq i$, then $A[k] \leq x$.
  - 2. if $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
  - 3. if $k = r$, then $A[k] = x$.



Loop Invariant of Partition (cont'd)

- At the beginning of each iteration of the loop of lines 3—6, for any array index k,
  - 1. if $p \le k \le i$, then $A[k] \le x$.
  - 2. if $i + 1 \le k \le j - 1$, then $A[k] > x$.
  - 3. if $k = r$, then $A[k] = x$.



Quicksort Runtime Analysis: Best Case

- A divide-and-conquer algorithm

$$T(n)=T(q - p) + T(r - q) + \theta(n)$$

  – Depends on the position of q in $A[p..r]$, but ???

- Best-, worst-, average-case analyses?
- **Best case:** Perfectly balanced splits---each partition gives an $n/2$: $n/2$ split.

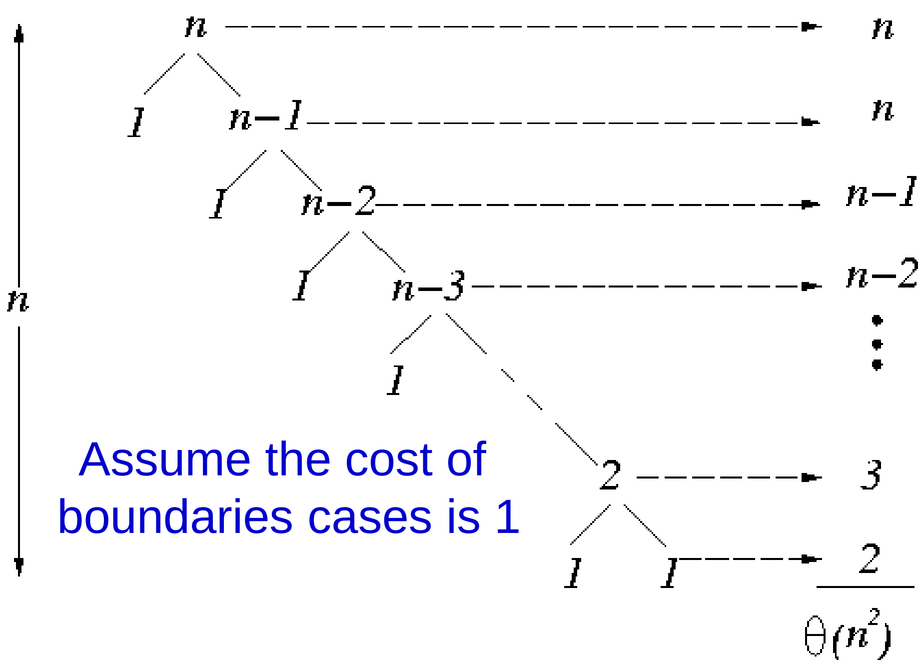
$$T(n) = T(n/2)+T(n/2) + \theta(n)$$
$$= 2T(n/2) + \theta(n)$$

- Time complexity: $\theta(n \lg n)$
  – Master method? Iteration? Substitution?

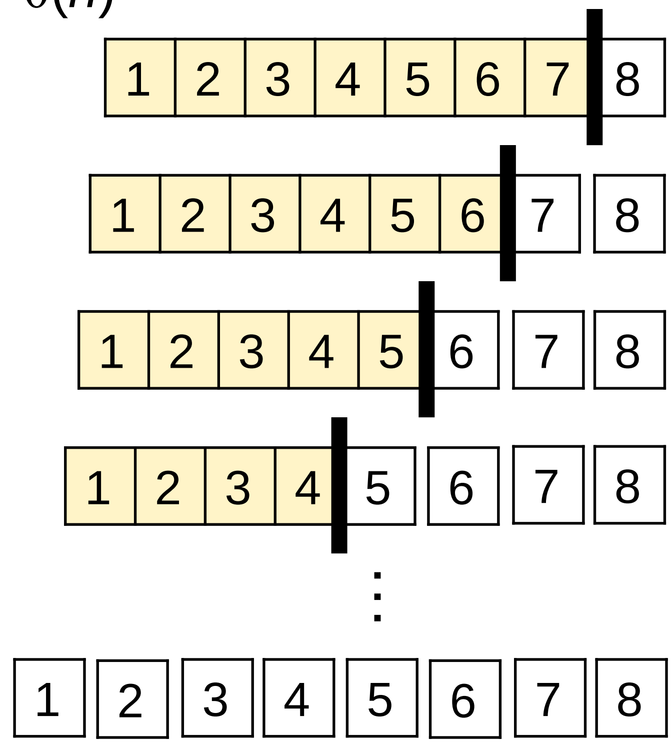
Quicksort Runtime Analysis: Worst Case

- **Worst case:** Each partition gives a 1 : n - 1 split.

$$
\begin{aligned}
T(n) &= T(1) + T(n\text{-}1) + \theta(n) \\
&= T(1) + (T(1) + T(n\text{-}2) + \theta(n\text{-}1)) + \theta(n) \\
&= \ldots \\
&= nT(1) + \Theta\left(\textstyle\sum_{k=1}^{n} k\right) \\
&= \theta(n^2)
\end{aligned}
$$



Assume the cost of boundaries cases is 1



The worst case occurs when an array is sorted!!

More on Worst-Case Analysis

- The **real** upper-bound:

$$T(n) = \max_{1 \le q \le n} \big(T(q-1) + T(n-q) + \Theta(n)\big)$$

- Guess $T(n) \le cn^2$ and verify it inductively:

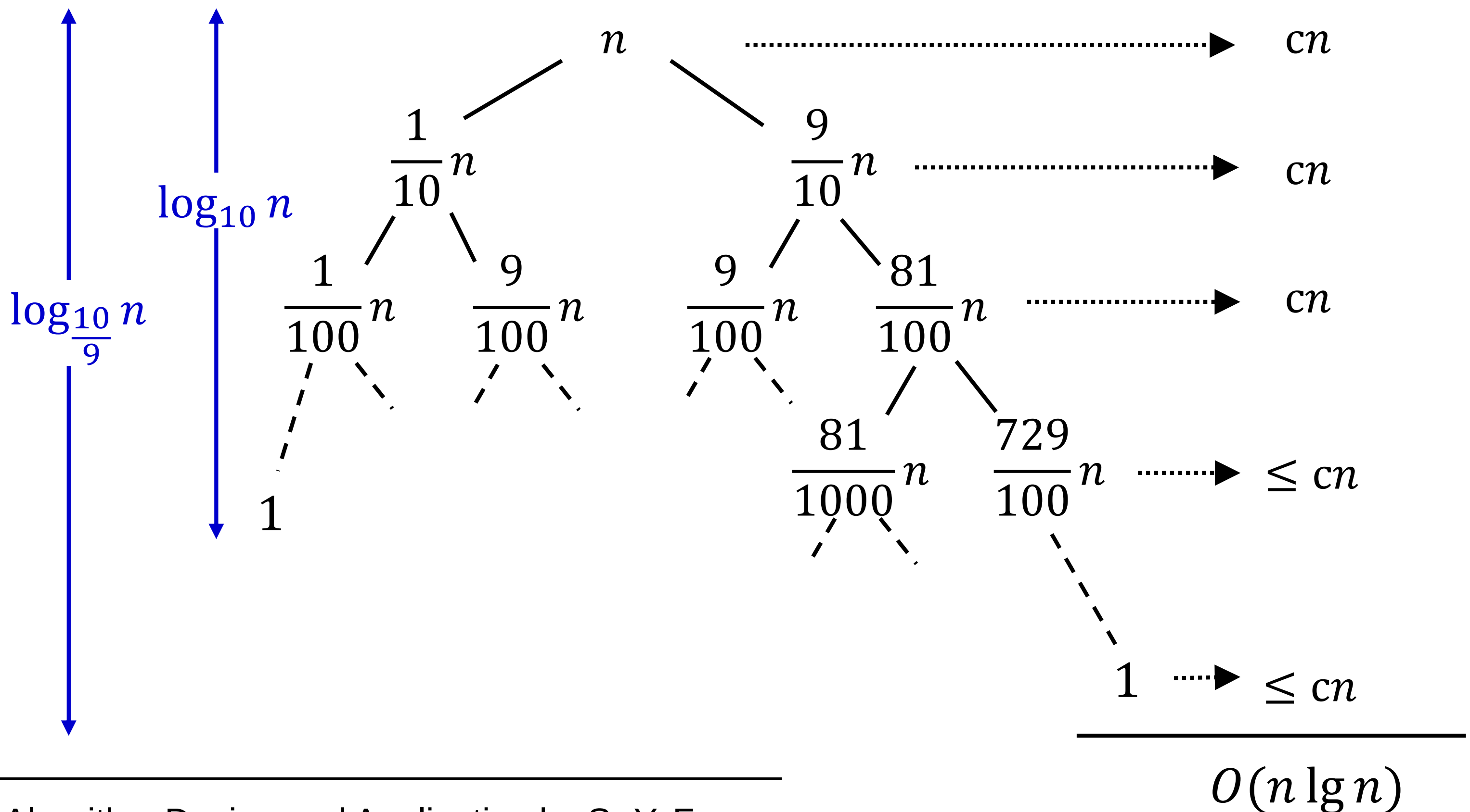
$$\begin{aligned} T(n) &\le \max_{1 \le q \le n} \big(c(q-1)^2 + c(n-q)^2 + \Theta(n)\big) \\ &= c \max_{1 \le q \le n} \big((q-1)^2 + (n-q)^2 + \Theta(n)\big) \end{aligned}$$

- $(q-1)^2 + (n-q)^2$ is maximum at its endpoints:

$$\begin{aligned} T(n) &\le c\big((n-1)^2 + \Theta(n)\big) \\ &= cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 \end{aligned}$$

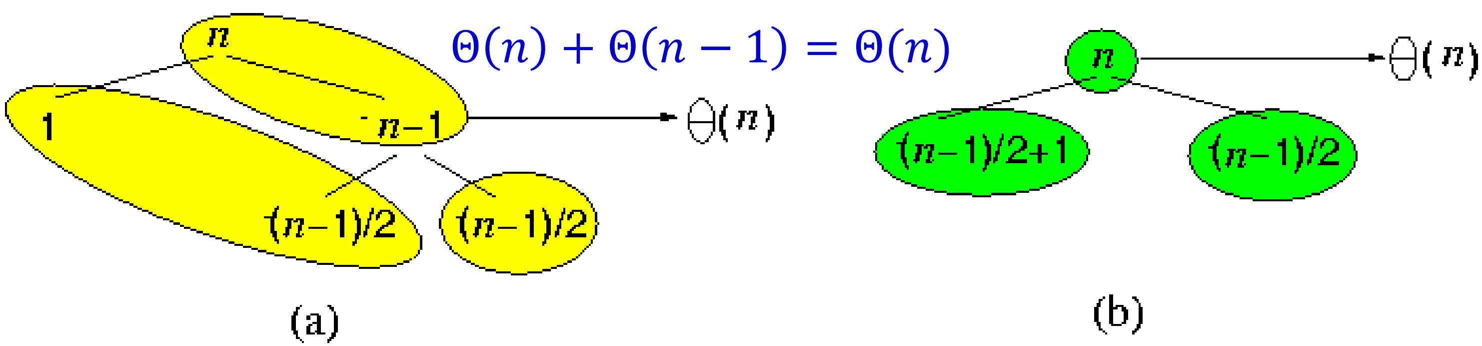
Quicksort Runtime Analysis: Balanced Partitioning

- Suppose the partitioning algorithm always produces a 9-to-1 proportional split
  - $T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{n}{10}\right) + \Theta(n) = O(n \lg n)$



Quicksort: Average-Case Analysis

- **Intuition:** Some splits will be close to balanced and others imbalanced; good and bad splits will be **randomly** distributed in the recursion tree.
- **Observation:** Asymptotically bad run time occurs only when we have many bad splits **in a row**.
  - A bad split followed by a good split results in a good partitioning after one extra step!
  - Thus, we will still get $O(n \lg n)$ run time.



Randomized Quicksort

- How to modify quicksort to get good average-case behavior on **all** inputs?
- **Randomization!**
  - Randomly permute inputs, or
  - Choose the partitioning element *x* randomly at each iteration.

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

```
RANDOMIZED-QUICKSORT(A, p, r)
1.   if p < r
2.       q = RANDOMIZED-PARTITION(A, p, r)
3.       RANDOMIZED-QUICKSORT(A, p, q-1)
4.       RANDOMIZED-QUICKSORT(A, q+1, r)
```

Average-Case Analysis

- Input array $A[1 \ldots n]$, partition at an index q with probability $1/n \Longrightarrow$ The expected value of $T(n)$:

$$E[T(n)] = \sum_{q=1}^{n} \frac{1}{n}\big(T(q-1) + T(n-q)\big) + \Theta(n)$$

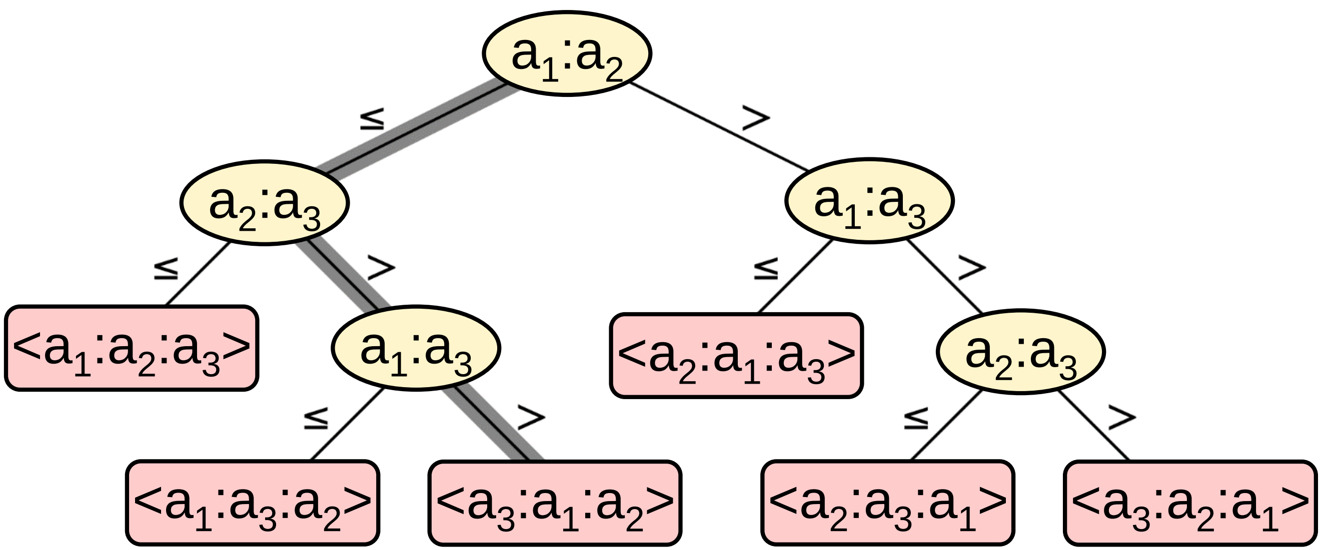
$$= \frac{1}{n}\sum_{q=1}^{n}\big(T(q-1) + T(n-q)\big) + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

- Guess $T(n) \le an \lg n + n$ and verify it inductively $\Rightarrow$ **the average-case is bounded by $O(n \lg n)$**
- Practically, quicksort is often 2-3 times faster than merge sort or heap sort.

Decision-Tree Model for Comparison-Based Sorter

- Comparison-based sorters use only comparisons between elements to gain order information
- The execution of a sorting algorithm $\Leftrightarrow$ tracing a simple path from the root to a leaf of the decision tree
  - $\leq \Rightarrow$ go to the left branch; **>** $\Rightarrow$ go to the right branch.
- Tree leaves represent all permutations of input (n! leaves)



$\Omega(n \lg n)$ Lower Bound for Comparison-Based Sorters

- There must be $n!$ leaves in the decision tree.
- Worst-case # of comparisons = #edges of the longest path in the tree (tree **height).**
- **Theorem:** Any decision tree that sorts n elements has height $\Omega(n \lg n)$.
  - Let h be the height of the tree T.
  - T has $\geq n!$ leaves.
  - T is binary, so has $\leq 2^h$ leaves.

$$2^h \geq n!$$
$$h \geq \lg n!$$
$$= \Omega(n \lg n)$$ /* Stirling's approximation $n! > \left(\frac{n}{e}\right)^n$ */

- Thus, any comparison-based sorter takes $\Omega(n \lg n)$ time in **the worst case**.
- Merge sort and heap sort are asymptotically optimal **comparison** sorters.

Comparisons of Comparison-based Sorters

Comparison-based sorters				
Algorithm	Runtime			In-place?
	Best case	Average case	Worst case	
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Yes
Merge	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	No
Heap	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	Yes
Quicksort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	Yes

Sorting in Linear Time

Comparison vs. Non-comparison

- A sorter is **comparison-based** if the only operation on keys is to compare two keys.
  - Insertion sort, merge sort, heapsort, quicksort
- The **non-comparison-based** sorters sort keys by looking at the values of individual elements.
  - **Counting sort:** Assume keys are in [1..*k*] and uses array indexing to count the # of elements of each value.
  - **Radix sort:** Assume each **integer** contains *d* digits, and each digit is in [1..*k*'].
  - **Bucket sort:** Sort data into buckets and then merge across buckets. Require information for input **distribution**.

Counting Sort: A Non-comparison-Based Sorter

- **Requirement:** Input integers are in known range [1..*k*].
- **Idea:** For each *x*, find # of elements ≤ *x* (say *m,* excluding *x*) and put *x* in the (*m* +1)*st* slot.
- Runs in $O(n+k)$ time, but needs extra $O(n+k)$ space.
- Example: ***A*: input; *B*: output; *C*: working array.**

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							**3**	

	0	1	2	3	4	5
C	2	2	4	**6**	7	8

(c)

	1	2	3	4	5	6	7	8
B		**0**					3	

	0	1	2	3	4	5
C	**1**	2	4	6	7	8

(d)

	1	2	3	4	5	6	7	8
B		0				**3**	3	

	0	1	2	3	4	5
C	1	2	4	**5**	7	8

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

Counting Sort

```
COUNTING-SORT(A, B, k)
for  i = 1 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 = 2 to k
    C[i] = C[i] + C[i-1]
// C[i] now contains the number of elements ≤ i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

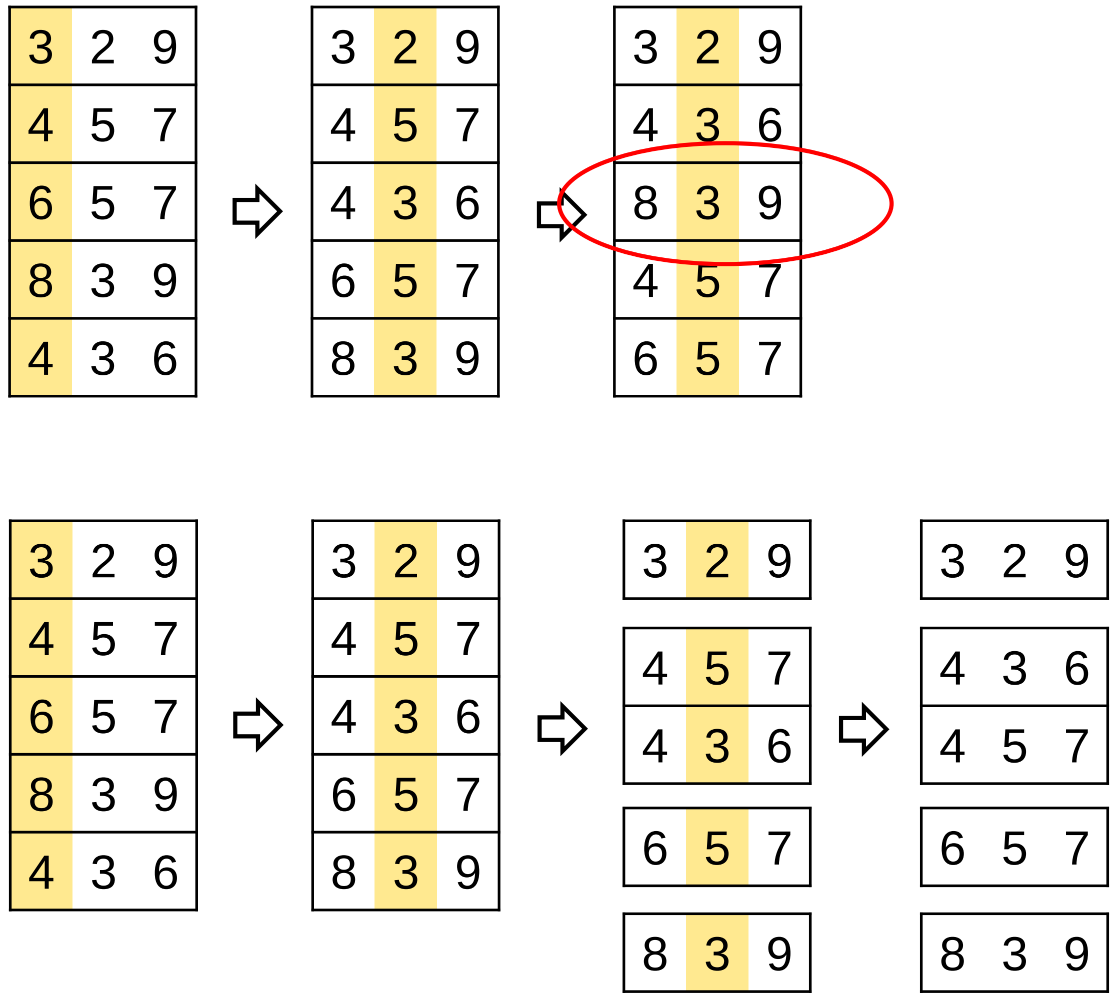
- Linear time if $k = O(n)$.
- **Stable** sorters: counting sort **(why?)**, insertion sort, merge sort.
- **Unstable** sorters: heap sort, quicksort.

Radix Sort

- **Requirement:** input an array of integers, each with *d* digits
- **Idea:** intuitively, one should first sort the numbers on their most significant digit, followed by the 2nd most significant digit, and so on
  - Problem: a lot of intermediate sets of numbers must be kept
- **Method:** counter-intuitively, it sorts the numbers on their least significant digit first, the 2nd least significant digit second, and so on

928		520		101		101
101		101		401		228
401		401		308		308
228	→	928	→	520	→	329
329		228		928		401
308		308		228		520
520		329		329		928

Sort from the Most Significant Digit



Radix Sort (cont'd)

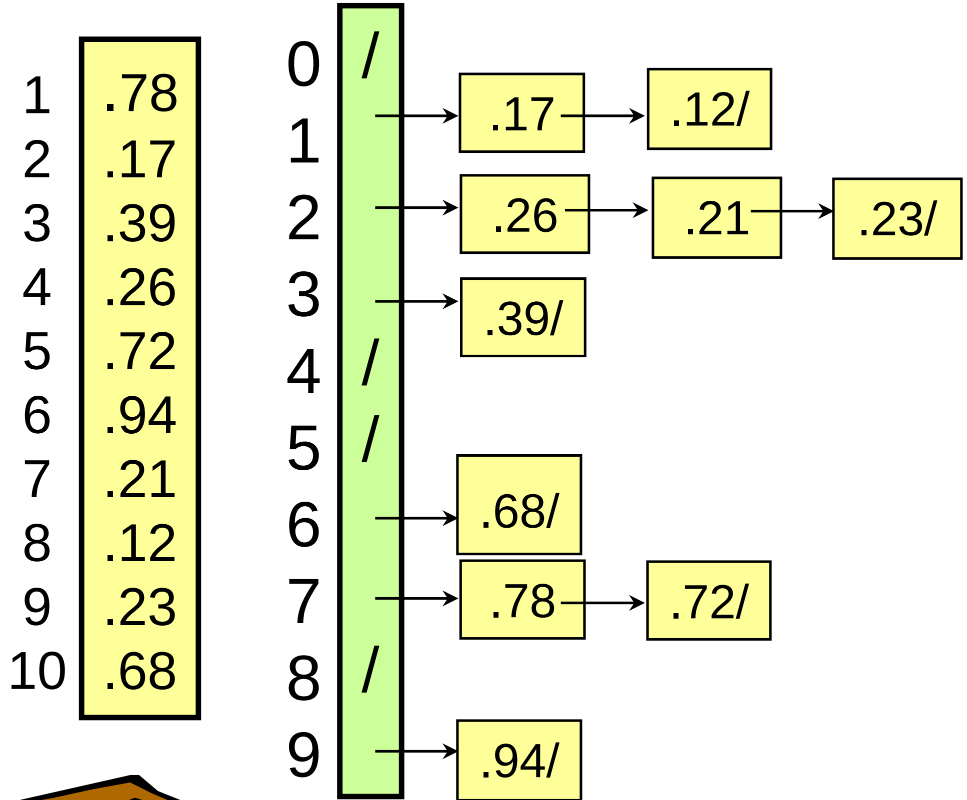
RADIX-SORT(*A*, *d*)
1. **for** *i* = 1 **to** *d*
2. Use a **stable** sorter to sort array *A* on digit *i*

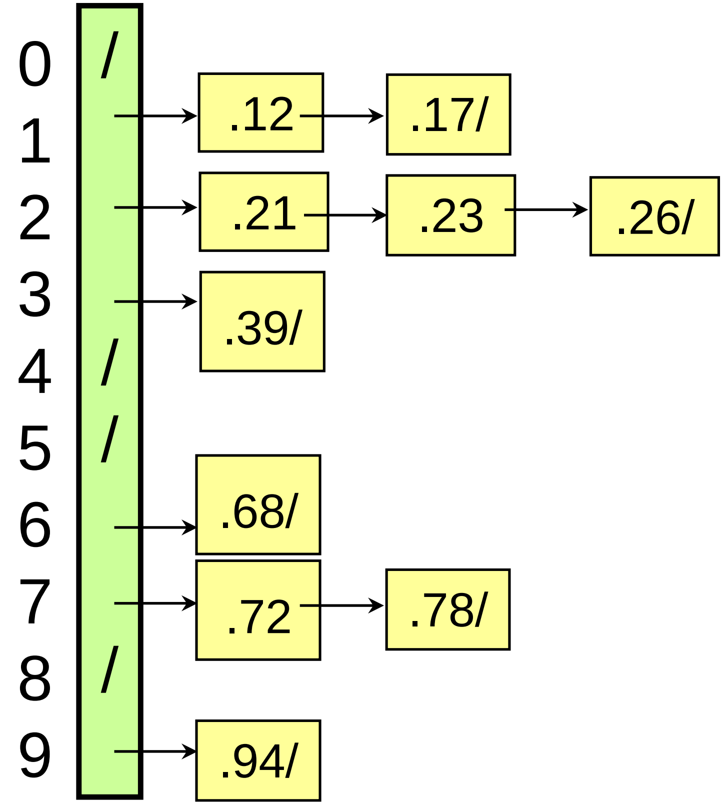
Why?

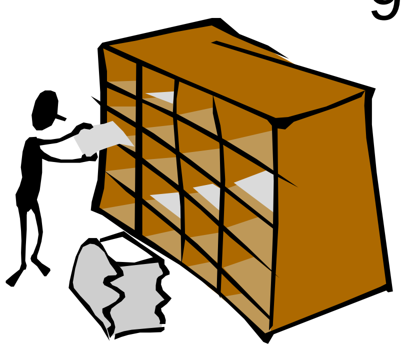
928	520	101	101
101	101	401	228
401	401	308	308
228	928	520	329
329	228	928	401
308	308	228	520
520	329	329	928

- Time complexity: $\Theta(d(n+k))$ for n d-digit numbers in which each digit has k possible values.
  - Which sorter?

Bucket Sort







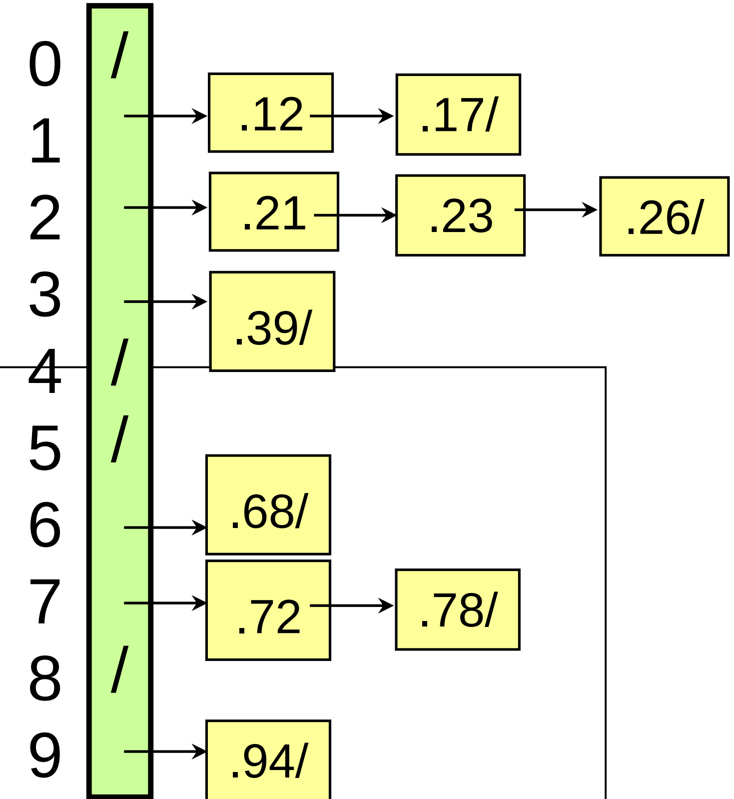
Step 1: distribute

Step 2: sort

Step3: combine

Bucket Sort (cont'd)

- **Requirement:** input is generated by a random process that distributes elements uniformly and independently over the interval [0,1)



BUCKET-SORT(A)
1. Let $B[0 \dots n-1]$ be a new array
2. $n = A.length$
3. **for** $i = 0$ **to** $n - 1$
4. Make $B[i]$ an empty list
5. **for** $i = 1$ **to** n
6. Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
7. **for** $i = 0$ **to** $n - 1$
8. Sort list $B[i]$ with insertion sort
9. Concatenate the lists $B[0], B[1], \dots, B[n-1]$ together in order

Why?

Sorting Algorithm Comparisons

	Runtime			
	Best case	Average case	Worst case	
Non-comparison-based sorters				
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	No
Radix	$O(d(n+k'))$	$O(d(n+k'))$	$O(d(n+k'))$	No
Bucket	-	$O(n)$	-	No

- **Counting sort:** Linear time if $k = O(n)$; pseudo-linear time, otherwise.
- **Radix sort:** Linear time if d is a constant and $k' = O(n)$; pseudo-polynomial time, otherwise.
- **Bucket sort:** Expected linear time if the sum of the squares of the bucket sizes is linear in the # of elements (uniform distribution)

Medians and Order Statistics

Order Statistics

- **Def:** Let *A* be an ordered set containing *n* elements. The ***i*-th order statistic** is the *i*-th smallest element.
  - Minimum: 1st order statistic
  - Maximum: *n*-th order statistic
  - Median: the $\left\lfloor \frac{n+1}{2} \right\rfloor$-th and the $\left\lceil \frac{n+1}{2} \right\rceil$-th order statistics

    Lower median: $\left\lfloor \frac{n+1}{2} \right\rfloor$; Higher median: $\left\lceil \frac{n+1}{2} \right\rceil$
- **The selection problem:** find the *i*-th order statistic for a given *i*
  - **Input:** a set *A* of *n* (distinct) numbers and a number *i*, $1 \le i \le n$.
  - **Output:** The element $x \in A$ that is larger than exactly (i -1) elements of *A*.
- Naive selection: sort *A* and return *A*[*i*].
  - Time complexity: $O(n\lg n)$.
  - Can we do better??

Finding Minimum (Maximum)

```
Minimum(A)
1.  min = A[1]
2.  for i = 2 to A.length
3.      if  min > A[i]
4.          min = A[i]
5.  return min
```

- **Exactly** n-1 comparisons.
  - Best possible?
  - Yes!! Think about a tournament.

Simultaneous Minimum and Maximum

- Naive simultaneous minimum and maximum: 2*n*-2 comparisons.
  - Best possible?
- The minimum and the maximum can be simultaneously found using at most $3\lfloor n/2 \rfloor$ comparisons!
  - A pair of elements is compared first, and then the smaller is compared with the current minimum and the larger with the current maximum => **3 comparisons for every 2 elements.**
  - If n is odd, take the first element to initial both the min and the max
    - #comparisons = $3\lfloor n/2 \rfloor$
  - If n is even, perform 1 comparison on the first 2 elements and initial the min and the max
    - #comparisons = $1 + 3\frac{(n-2)}{2} = \frac{3n}{2} - 2$

Selection in Expected Linear Time

```
Randomized-Select(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = Randomized-Partition(A,p,r)
4.  k = q – p + 1
5.  if i == k
6.      return A[q]
7.  if i < k
8.      return Randomized-Select(A, p, q-1, i)
9.  else return Randomized-Select(A, q+1, r, i-k)
```

- Randomized-Partition first swaps $A[r]$ with a random element of A and then proceeds as in regular PARTITION.
- Randomized-Select is like Randomized-Quicksort, except that we only need to make one recursive call.
- Time complexity
  - Worst case: 1:n-1 partitions $\Rightarrow T(n) = T(n-1) + \theta(n) = \theta(n^2)$.
  - Best case: $T(n) = \theta(n)$.
  - Average case? Like quicksort, asymptotically close to best case.

Selection in Linear Expected Time: Average Case

- For each k such that $1 \le k \le n$, the subarray $A[p \dots q]$ has k elements with probability $1/n$
- The expected value of $T(n)$:

$$E[T(n)] = \sum_{k=1}^{n} \frac{1}{n} \cdot T(\max(k-1, n-k)) + O(n)$$

$$\boxed{\max(k-1, n-k) = \begin{cases} k-1, if\ k > \lceil n/2 \rceil \\ n-k, if\ k \le \lceil n/2 \rceil \end{cases}}$$

$$E[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + O(n)$$

Selection in Linear Expected Time: Average Case

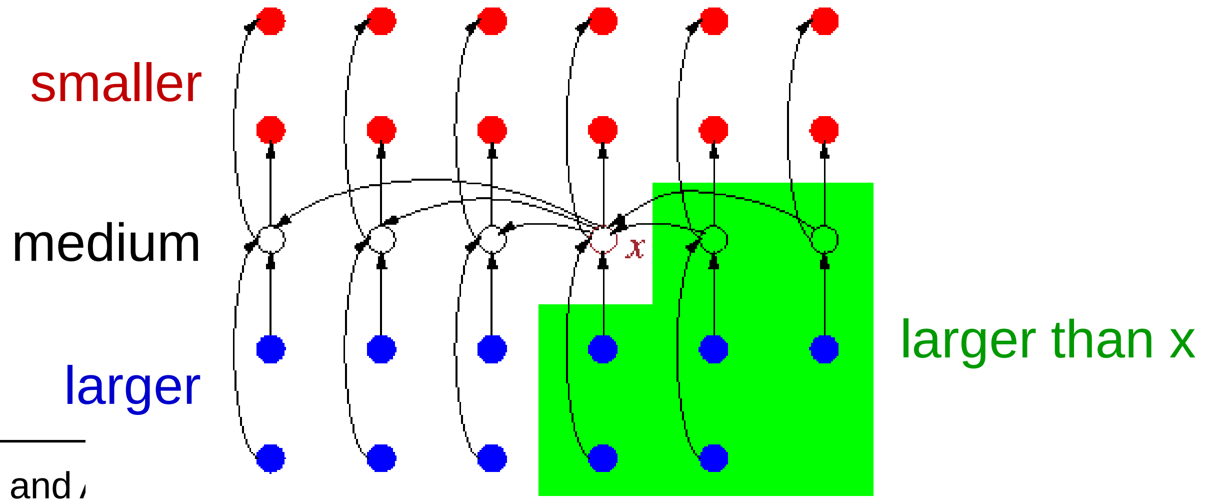
- Assume $T(n) \le cn$

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + O(n) = \frac{2c}{n}\left(\sum_{k}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + O(n) \\
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} + \frac{(\lfloor n/2 \rfloor - 1)(\lfloor n/2 \rfloor)}{2}\right) + O(n) \\
&\le \frac{2c}{n}\left(\frac{(n-1)n}{2} + \frac{(n/2-2)(n/2-1)}{2}\right) + O(n) \\
&= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + O(n) \\
&\le \frac{3cn}{4} + \frac{c}{2} + O(n) \\
&= cn - c\left(\frac{n}{4} - \frac{1}{2}\right) + O(n) \le cn
\end{aligned}
$$

- Thus, on average, Randomized-Select runs in linear time.

Selection in Worst-Case Linear Time

- **Key:** Guarantee a good split when array is partitioned.
- Select(*A*, *p*, *r*, *i*)
  1. Divide input array *A* into $\lfloor n/5 \rfloor$ groups of size 5 (possibly with a leftover group of size < 5).
  2. Find the median of each of the $\lceil n/5 \rceil$ groups.
  3. Call Select recursively to find the median *x* of the $\lceil n/5 \rceil$ medians.
  4. Partition array around *x*, splitting it into two arrays of *A*[*p*, *q-1*] (with *k-1* elements) and *A*[*q*+1, *r*] (with *n-k* elements).
  5. **if** ($i = k$) **then** return x
     elseif ($i < k$) **then** Select(*A*, *p*, *q-1*, *i*) **else** Select(*A*, *q* + 1, *r*, *i* - *k*).



Runtime Analysis

- **Main idea:** Select guarantees that x causes a good partition.
  - At least $3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6$ elements > x (or < x)
  - Worst-case split has $\frac{7n}{10} + 6$ elements in the bigger subproblem.
- Run time: $T(n) = T(\lceil n/5 \rceil) + T(7n/10+6) + O(n)$.
  1. $O(n)$: break into groups.
  2. $O(n)$: finding medians (constant time for 5 elements).
  3. $T(\lceil n/5 \rceil)$: recursive call to find median of the medians.
  4. $O(n)$: partition.
  5. $T(7n/10+6)$: searching in the bigger partition.
- Apply the substitution method to prove that $T(n)=O(n)$.