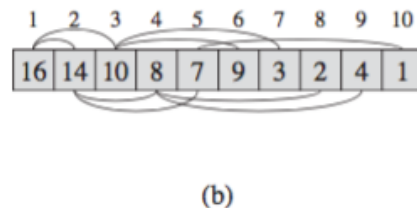
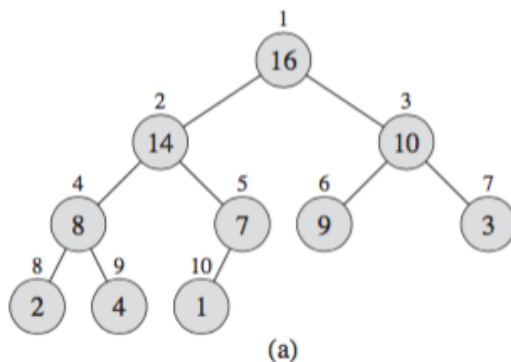


Lecture 5: Sorting Part A

Heapsort

- Running time $O(n \lg n)$, like merge sort
 - Sorts in place (as insertion sort), only constant number of array elements are stored outside the input array at any time
 - Combines better attributes of these other sorting algorithms
 - Uses an algorithm design technique: using a data structure, a heap
 - * Heap also referred as garbage-collected storage
 - * As in java and Lisp
- Binary heap data structure
 - Array object can be viewed as a nearly complete binary tree
 - Each node corresponds to an element of the array
 - Tree completely filled on all levels, except (possibly) the lowest (Lowest filled from the left up to a point)
- Binary heap data structure
 - An array A represents the heap, an object with two attributes
 - * $A.length$, number of elements in the array
 - * $A.heap - size$, how many elements in the heap are stored within array A
 - * $A[1..A.length]$ may contain numbers but only elements in $A[1..A.heap - size]$, where $0 \leq A.heap - size \leq A.length$, are valid elements in the heap
 - Root of tree is $A[1]$
- Given index i of a node, we can compute indices of its parent, left child, and right child
- The values in the nodes satisfy the heap property
 - max-heaps
 - * max-heap property
 - * For every node i other than the root, $A[PARENT(i)] \geq A[i]$
 - * The value of a node is at most the value of its parent
 - * Largest element in a max-heap is stored at the root
 - min-heaps
 - * min-heap property
 - * For every node i other than the root, $A[PARENT(i)] \leq A[i]$
 - * The smallest element in a min-heap is at the root
- max-heaps are used for the heapsort algorithm
- min-heaps implement priority queues
- Viewing heap as a tree
 - height of a node in a heap \rightarrow number of edges on the longest simple downward path from the node to a leaf
 - height of a heap \rightarrow the height of its root
- A heap of n elements based on a complete binary tree (height is $\theta(\log n)$)
- Basic operations on heaps run in time proportional to the height of the tree



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.

Maintaining the Heap Property

- MAX-HEAPIFY, runs in $O(\lg n)$ time, key to maintain the max-heap property
- MAX-HEAPIFY used to maintain the max-heap property
 - Inputs: an array A and an index i
 - MAX-HEAPIFY assumes that binary trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps, but that $A[i]$ might be smaller than its children (violating the max-heap property)
 - MAX-HEAPIFY moves the value $A[i]$ down in the max-heap so that subtree rooted at index i follows the max-heap property
- Running time of MAX-HEAPIFY on subtree of size n at node i is
 - $\Theta(1)$ to fix up relationships among $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$
 - Time to run MAX-HEAPIFY on subtree rooted at one children of node i
 - * Children's subtrees have size at most $2n/3$

* Worst case when bottom level of tree is exactly half full

- Running time of MAX-HEAPIFY with recurrence

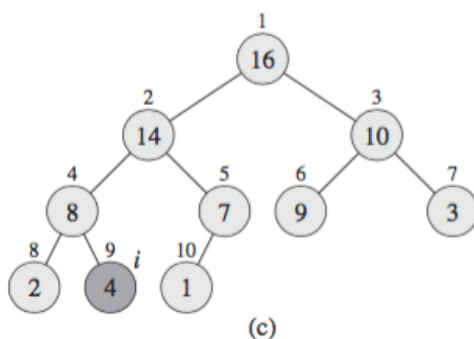
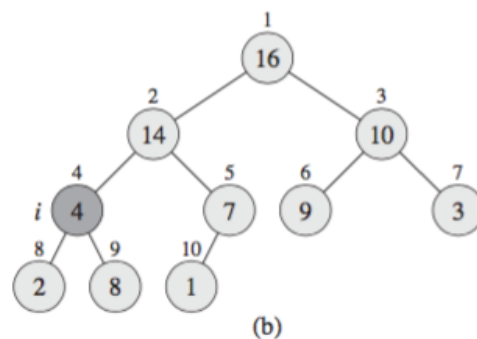
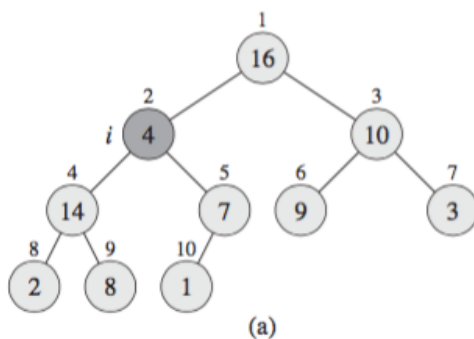
$$* T(n) \leq T(2n/3) + \Theta(1)$$

item Solution to recurrence by case 2 of master theorem

$$* T(n) = O(\lg n)$$

MAX-HEAPIFY(A, i)

1. $l = \text{left}(i)$
2. $r = \text{right}(i)$
3. if $l \leq A.\text{heap-size}$ and $A[l] > A[i]$
4. $\text{largest} = l$
5. else $\text{largest} = i$
6. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$
7. $\text{largest} = r$
8. if $\text{largest} \neq i$
9. exchange $A[i]$ with $A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}$)



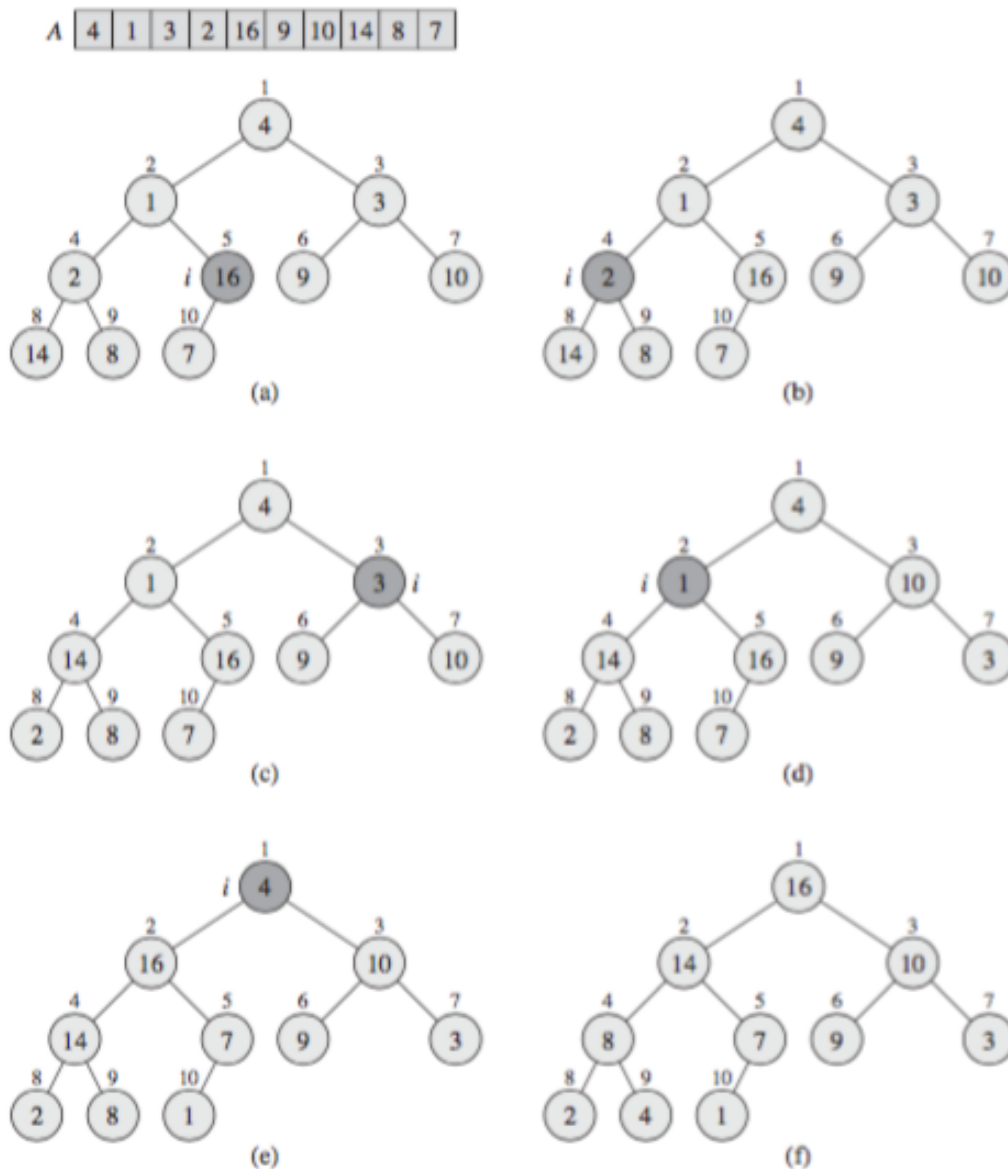
The action of MAX-HEAPIFY($A, 2$), where $A.\text{heap-size} = 10$. (a) The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in (c), node 4 is fixed up, and the recursive call MAX-HEAPIFY($A, 9$) yields no further change to the data structure.

Building a Heap

- BUILD-MAX-HEAP, linear time, produces a max-heap from an unordered input array
- Use MAX-HEAPIFY bottom-up
 - Convert array $A[1..n]$, $n = A.length$, into a max-heap
 - Elements in the subarray $A[(\lfloor n/2 \rfloor + 1)..n]$ are the leaves of the tree
 - Each of these is a 1-element heap to begin with
- BUILD-MAX-HEAP(A) goes through remaining nodes of the tree
 - Runs MAX-HEAPIFY on each of them

BUILD-MAX-HEAPIFY(A)

1. $A.heap-size = A.length$
2. for $i = \lfloor A.length/2 \rfloor$ down to 1
3. MAX-HEAPIFY(A, i)



The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. (a) A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). (b) The data structure that results. The loop index i for the next iteration refers to node 4. (c)–(e) Subsequent iterations of the for loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. (f) The max-heap after BUILD-MAX-HEAP finishes.

Loop Invariant for BUILD-MAX-HEAP

- Start of each iteration of for loop, lines 2-3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap
- Loop invariant shows
 - This invariant is true prior to first loop iteration
 - Each iteration of loop maintains the invariant
 - The invariant provides useful property to show correctness when the loop terminates
- **Initialization:** Prior to first iteration of loop
 - $i = \lfloor n/2 \rfloor$.
 - Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and also the root of a trivial max-heap
- **Maintenance:** See that each iteration maintains the loop invariant
 - Children of node i numbered higher than i , they are both roots of max-heaps
 - This is required by MAX-HEAPIFY(A, i) to make node i a max-heap root
 - MAX-HEAPIFY(A, i) preserves property that nodes $i + 1, i + 2, \dots, n$ are all roots of a max-heap
 - Decreasing i in for loop update, reestablishes loop invariant for next iteration
- **Termination:** at termination $i = 0$
 - By loop invariant, each node $1, 2, \dots, n$ is the root of a max-heap
 - In particular, node 1, the root of the tree

Upper Bound for BUILD-MAX-HEAP

- Each call to MAX-HEAPIFY costs $O(\lg n)$ time
- BUILD-MAX-HEAP makes $O(n)$ such calls
- Running time is $O(n \lg n)$
- This upper bound is correct but not asymptotically tight

• A Tighter Bound for BUILD-MAX-HEAP

- Time for MAX-HEAPIFY to run at a node varies with height of the node in the tree
- Heights of most nodes are small
- Tighter analysis uses properties that an n -element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h
- Time required by MAX-HEAPIFY when called on node of height h is $O(h)$
- Total cost of BUILD-MAX-HEAP:

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

- Evaluating summation by substituting $x = \frac{1}{2}$ in formula

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

- We bound running time of BUILD-MAX-HEAP as

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

The Heapsort Algorithm

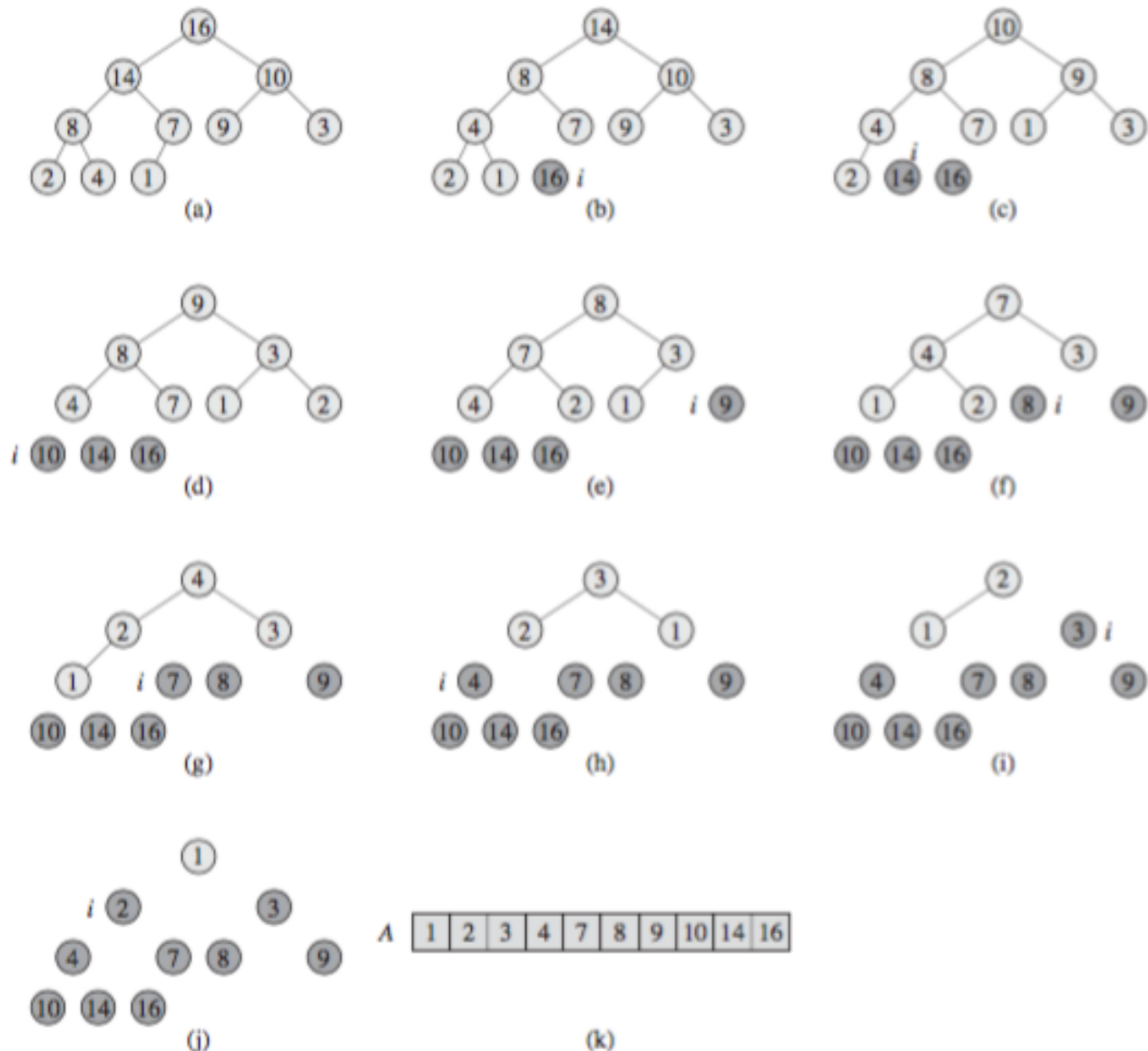
- HEAPSORT, runs in $O(n \lg n)$ time, sorts an array because each of $n-1$ calls to MAX-HEAPIFY take $O(\lg n)$
- Starts by using BUILD-MAX-HEAP on input array $A[1..n]$, where $n = A.length$
 - Maximum element of array stored at root $A[1]$, we can put it in its final position, exchanging it by $A[n]$
 - The new root might violate the max-heap property, we restore it by calling MAX-

HEAPIFY($A, 1$), that leaves a max-heap $A[1..n-1]$

- The process is repeated for max-heap size $n-1$ down to a heap of size 2

HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. for $i = A.length$ down to 2
3. exchange $A[1]$ with $A[i]$
4. $A.heap-size = A.heap-size - 1$
5. MAX-HEAPIFY($A, 1$)



The operation of HEAPSORT. (a) The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. (b)–(j) The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. (k) The resulting sorted array A .

HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. for $i = A.length$ down to 2
3. exchange $A[1]$ with $A[i]$
4. $A.heap-size = A.heap-size - 1$
5. MAX-HEAPIFY($A, 1$)

Priority Queues

- Heapsort is a good sorting algorithm but quicksort usually beats it in practice
- The heap data structure has many uses
- A popular application of heaps is an efficient priority queue
- A priority queue can be max-priority or min-priority
- We focus on max-priority queues, based on max-heaps
- A **priority queue** is a data structure that maintains a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations.
- **MAXIMUM**(S): returns the element S with the largest key and runs in $\Theta(1)$ time

HEAP-MAXIMUM(A)

1. return $A[1]$

- **INSERT**(S, x): inserts element x into set S , equivalent to $S = S \cup \{x\}$ and runs in $O(\lg n)$ time.

MAX-HEAP-INSERT(A, key)

1. $A.heap-size = A.heap-size + 1$
2. $A[A.heap-size] = -\infty$
3. HEAP-INCREASE-KEY($A, A.heap-size, key$)

- **EXTRACT-MAX**(S): removes and returns the element of S with the largest key and runs in $O(\lg n)$ time since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY.

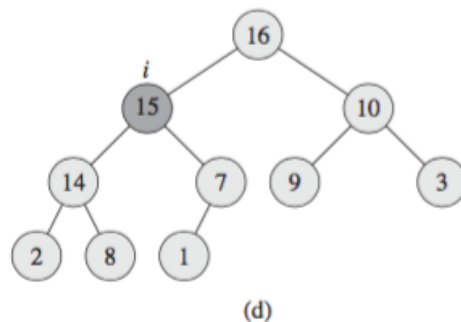
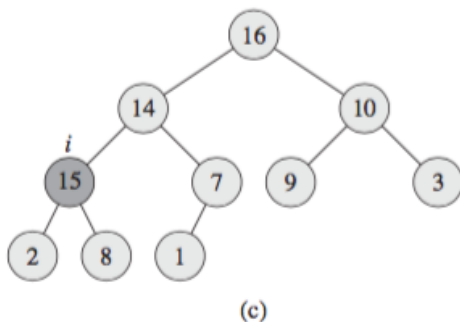
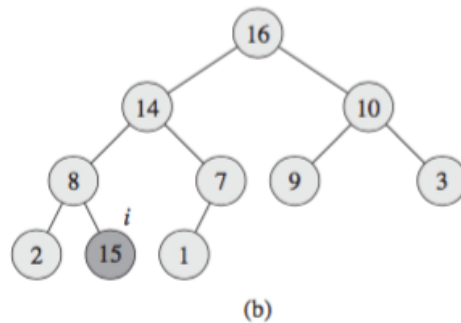
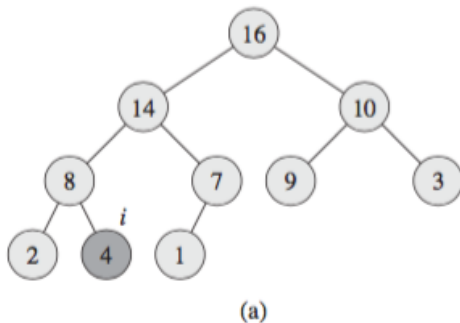
HEAP-EXTRACT-MAX(A)

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

- **INCREASE-KEY**(S, x, k): increases the value of element x 's key to new value k , which is assumed to be as large as x 's current value and runs in $O(\lg n)$ time

HEAP-INCREASE-KEY(A, i, key)

1. if $key < A[i]$
2. error "new key is smaller than current key"
3. $A[i] = key$
4. while $i > 1$ and $A[parent(i)] < A[i]$
5. exchange $A[i]$ with $A[parent(i)]$
6. $i = parent(i)$



The operation of HEAP-INCREASE-KEY. (a) The max-heap of the Figure with a node whose index is i heavily shaded. (b) This node has its key increased to 15. (c) After one iteration of the while loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. (d) The max-heap after one more iteration of the while loop. At this point, $A[parent(i)] \leq A[i]$. The max-heap property now holds and the procedure terminates.

Quicksort

- Worst case $\Theta(n^2)$
- Average case with expected running time of $\Theta(n \lg n)$
 - A good practical choice, remarkably efficient on average
 - Constant factors hidden in $\Theta(n \lg n)$ are quite small
- Performs sorting in place
- Works well even in virtual-memory environments
- Applies divide-and-conquer to sort a subarray $A[p..r]$
- **Divide:** Partition array $A[p..r]$ into two subarrays $A[p..q-1]$ and $A[q+1..r]$
 - Each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is less or equal to each element of $A[q+1..r]$
 - q is computed as part of the partitioning
- **Conquer:** Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort

- **Combine:** Subarrays are already sorted, no work needed to combine them

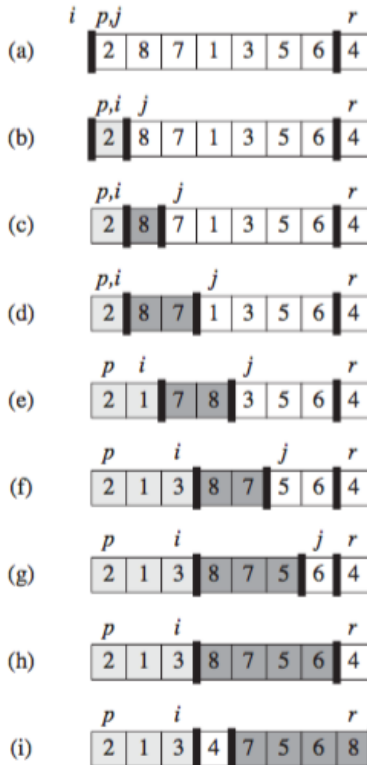
QUICKSORT(A, p, r)

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

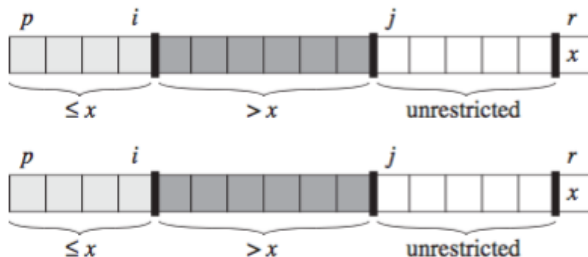
- There exist different algorithms for partitioning

PARTITION(A, q, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$



The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x . Lightly shaded array elements are all in the first partition with values no greater than x . Heavily shaded elements are in the second partition with values greater than x . The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x . (a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions. (b) The value 2 is “swapped with itself” and put in the partition of smaller values. (c)–(d) The values 8 and 7 are added to the partition of larger values. (e) The values 1 and 8 are swapped, and the smaller partition grows. (f) The values 3 and 7 are swapped, and the smaller partition grows. (g)–(h) The larger partition grows to include 5 and 6, and the loop terminates. (i) In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

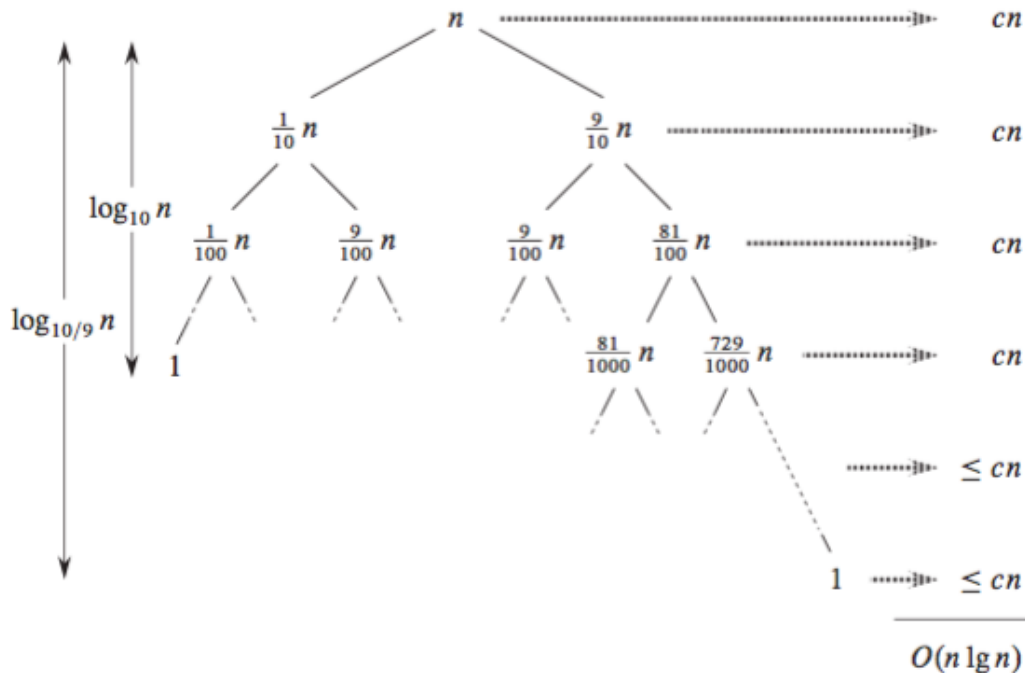


The four regions maintained by the procedure PARTITION on a subarray $A[p \dots r]$. The values in $A[p \dots i]$ are all less than or equal to x , the values in $A[i+1 \dots j-1]$ are all greater than x , and $A[r] = x$. The subarray $A[j \dots r-1]$ can take on any values.

The two cases for one iteration of procedure PARTITION. (a) If $A[j] > x$, the only action is to increment j , which maintains the loop invariant. (b) If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Performance of Quicksort

- Depends on ...
 - Whether the partition is or not balanced
 - What element was used to partition
 - Balanced partition (Fast as MergeSort)
 - Unbalanced (Slow as InsertionSort)
- Worst case partitioning (previously ordered input)
 - when partitioning produces 1 region with $n-1$ elements and the other with 0 elements
 - Assumes that this happens in each step of the algorithm
 - Cost to partition $\Theta(n)$
 - Recursive call on array size 0 just returns, $T(0) = \Theta(1)$ $T(1) = \Theta(1)$
 - Recurrence $T(n) = T(n-1) + \Theta(n)$
 - If we sum costs at each level of the recursion we get an arithmetic series (eq. A.2) that evaluates to $\Theta(n^2)$
 - $T(n) = T(n-1) + \Theta(n)$ has solution $T(n) = \Theta(n^2)$
 - $\sum_{k=1}^n k = \frac{1}{2}n(n+1) = \Theta(n^2)$
- Best case in partition
 - Regions of size $n/2$
 - Recurrence $T(n) = 2T(n/2) + \Theta(n)$
 - Case 2. master theorem $\leftarrow T(n) = \Theta(n \lg n)$
- Average case closer to the best case than to the worst case
 - Suppose a split in proportion 9-1 \leftarrow looks too unbalanced
 - Recurrence $T(n) = T(9n/10) + T(n/10) + n$
 - See recursion tree, balanced split case
- Balanced partitioning
 - Each level has cost n
 - Boundary condition at depth $\log_{10} n = \Theta(\lg n)$ (Levels have a cost at most n)
 - Recursion ends at depth $\log_{10/9} n = \Theta(\lg n)$
 - Total cost of QuickSort $\leftarrow = \Theta(n \lg n)$ (For each split with constant proportion)
- Intuition for the average case
 - Partition produces good and bad splits
 - Randomly distributed
 - Suppose good and bad splits alternate (in the tree)
 - * Good \rightarrow splits of the best case
 - * Bad \rightarrow splits of the worst case
 - Running time is still $\Theta(n \lg n)$
 - * With larger constant hidden by the O -notation



A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.