


EECE7205: Fundamentals of Computer Engineering



Sorting Algorithms



Introduction

- We will study more algorithms to solve the following sorting problem:
 - **Input:** A sequence of n numbers (a_1, a_2, \dots, a_n)
 - **Output:** A permutation (*reordering*) $(a'_1, a'_2, \dots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.
- In practice, the numbers to be sorted are usually part of a collection of data called a **record**. This part to be sorted is called the record's **key**.
- Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms.
- A sorting algorithm sorts **in place** if only a constant number of elements of the input array are ever stored outside the array.



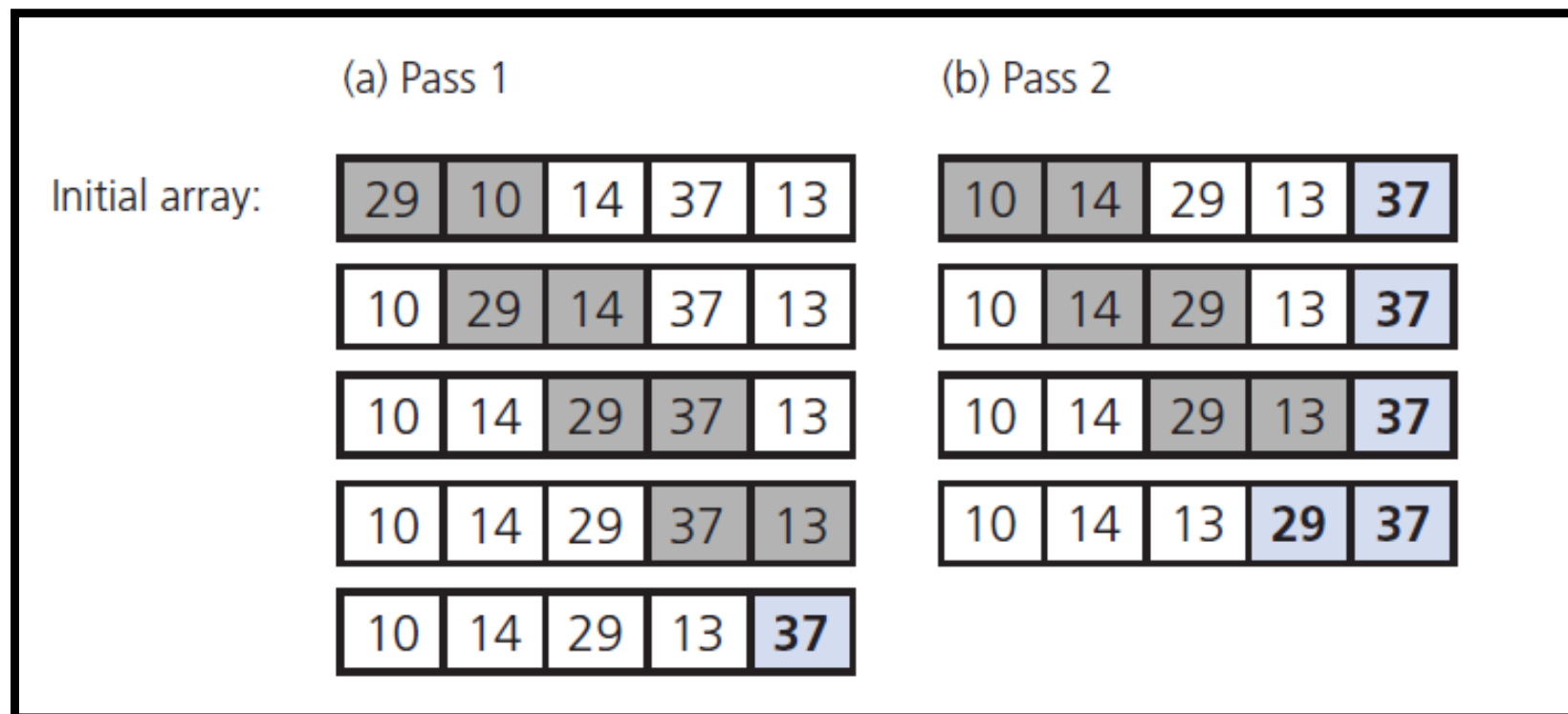
Insertion and Merge Sort

- We introduced insertion and merge sort.
- Insertion sort takes $\Theta(n^2)$ time in the worst case, however, it is a fast in-place sorting algorithm for small input sizes.
- Merge sort has a better asymptotic running time, $\Theta(n \log n)$, but its MERGE procedure does not operate in place.



The Bubble Sort

- Compares adjacent items
 - Exchanges them if out of order
 - Requires several passes over the data
- When ordering successive pairs
 - Largest item bubbles to end of the array





The Bubble Sort Analysis

- Worst case $O(n^2)$
- Best case (array already in order) is $O(n)$
- It operates in place.

10	13	14	29	37
----	----	----	----	----



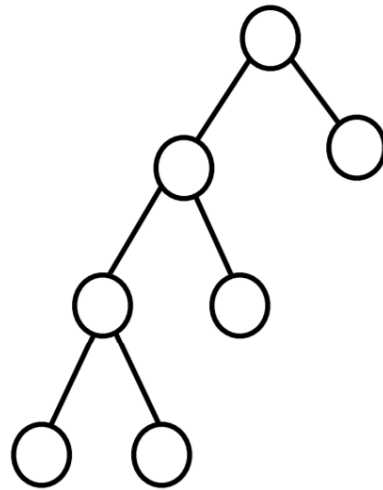
Heapsort

- Heapsort, like merge sort, has running time $\Theta(n \log n)$.
- It, like insertion sort, sorts in place.
- Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

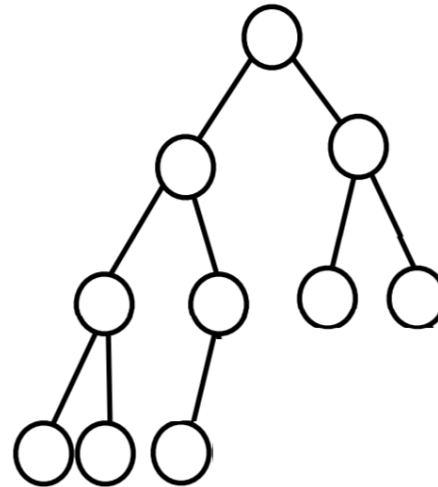


Binary Tree Types

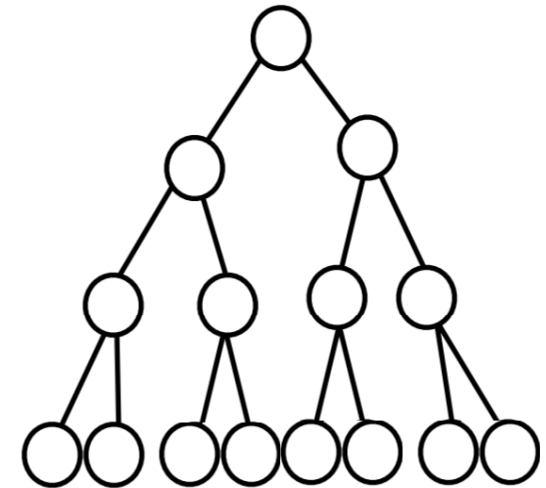
- In a **binary** tree, every node has 0, 1, or 2 children.
 - Nodes with 0 children all called **leaves**.
- In a **full** binary tree, every node has either 0 or 2 children.
- A **complete** binary tree is completely filled on all levels except the lowest level, which is filled from the left to right.
- A **perfect** binary tree is a full binary tree with all **leaves** at the same depth.



full



complete

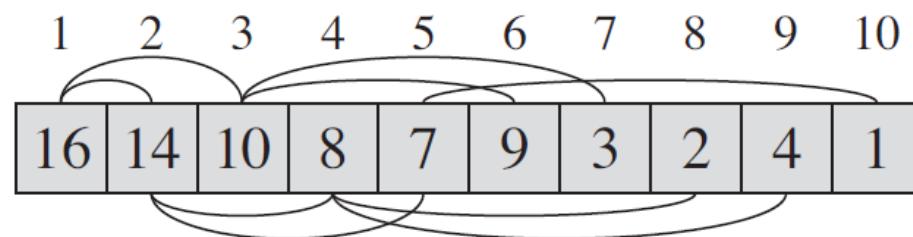
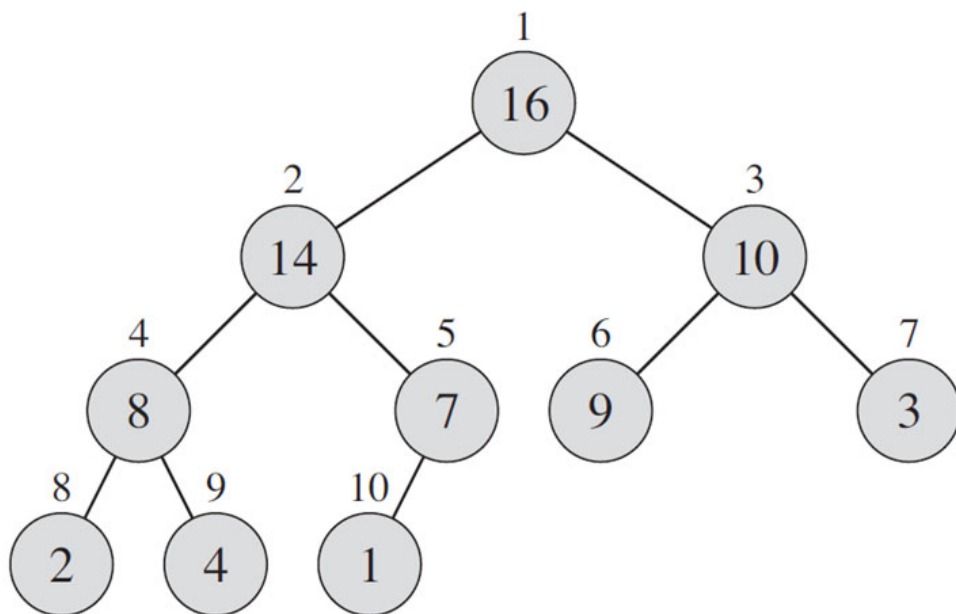


perfect



Heap

- The *(binary) heap* data structure is an array that represent a complete or perfect binary tree.
- Each node of the tree corresponds to an element of the array.





Heap Indices Calculation

- The root of the tree is $A[1]$, and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

PARENT(i)

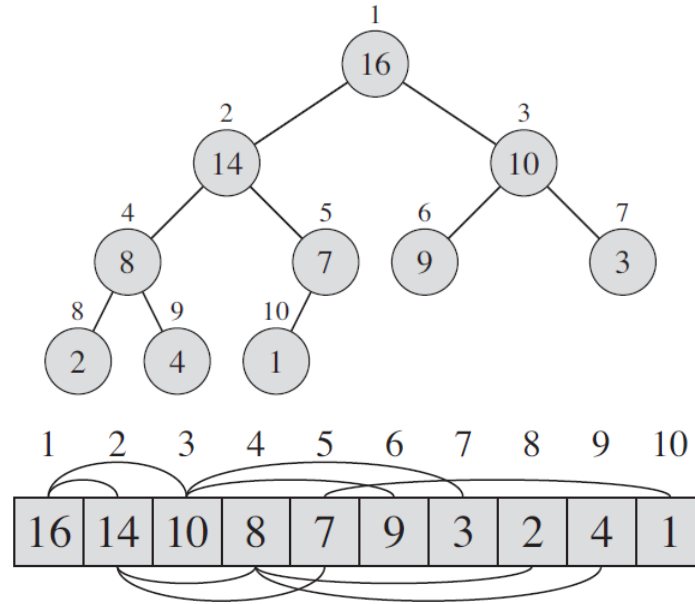
1 return $\lfloor i/2 \rfloor$

LEFT(i)

1 return $2i$

RIGHT(i)

1 return $2i + 1$

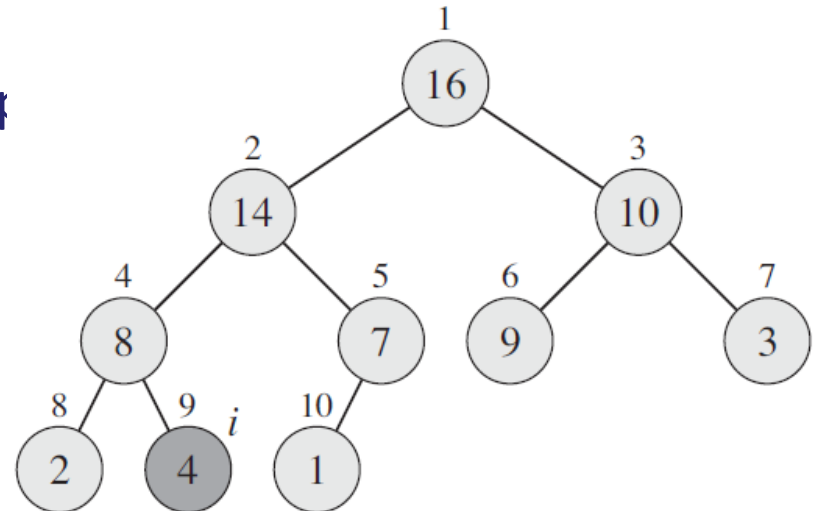


- Most computers can compute $2i$ in one instruction by **simply shifting** the binary representation of i left by one bit position. Similarly, $\lfloor i/2 \rfloor$ can be computed by shifting i right one-bit position.
- With the array representation of an n -element heap, the **leaves** are the nodes indexed by $\lfloor n/2 \rfloor + 1, \dots, n$. These nodes are the nodes with no children, since $2i$ and $2i + 1$ are past the end of the heap.



Heap Definitions

- The **max-heap property** is that for every node i other than the root, $A[\text{PARENT}(i)] \geq A[i]$, that is, the value of a node is at most the value of its parent.
 - Thus, the largest element in a max-heap is stored at the root, and
 - the root of any subtree has the largest element in the subtree.

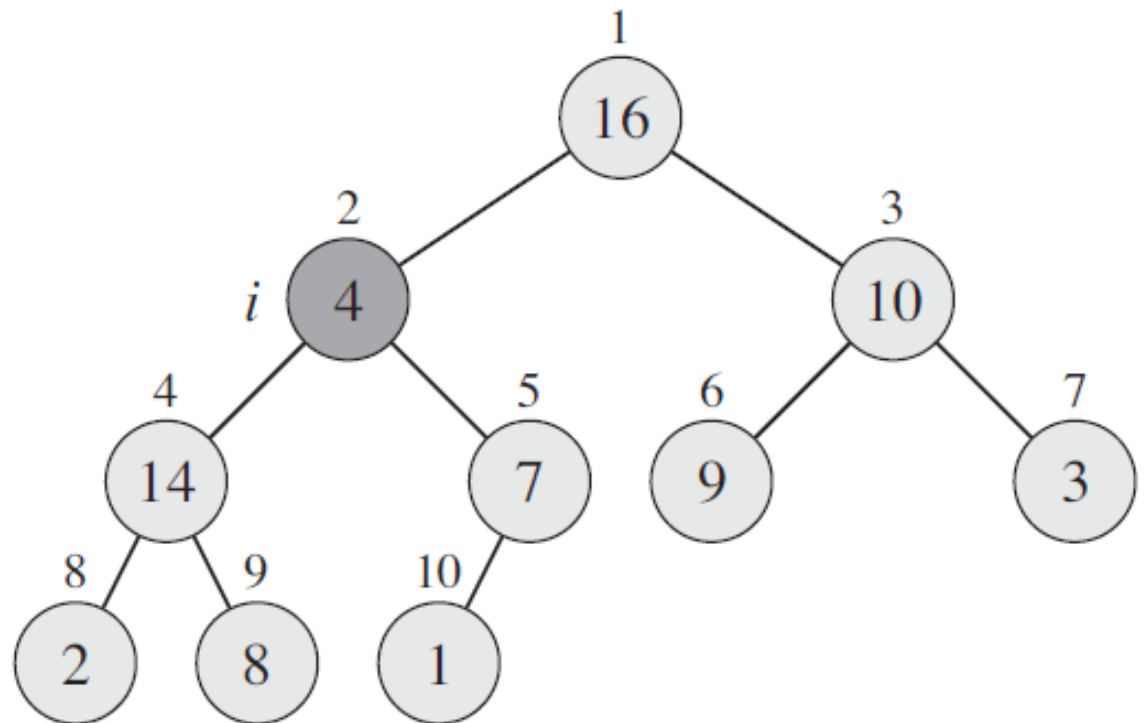


- The **height** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf,
 - The height of the heap to be the height of its root.
 - A perfect tree with height k has number of nodes $n = 2^{k+1} - 1$
 - Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\log n)$



Maintaining the Heap Property

- The procedure MAX-HEAPIFY maintains the max-heap property.
- MAX-HEAPIFY of node i assumes that the binary trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property.
- Example $A[2]$ in the shown figure.
- MAX-HEAPIFY lets the value at $A[i]$ “float down” in the max-heap so that the subtree rooted at index i obeys the max-heap property.



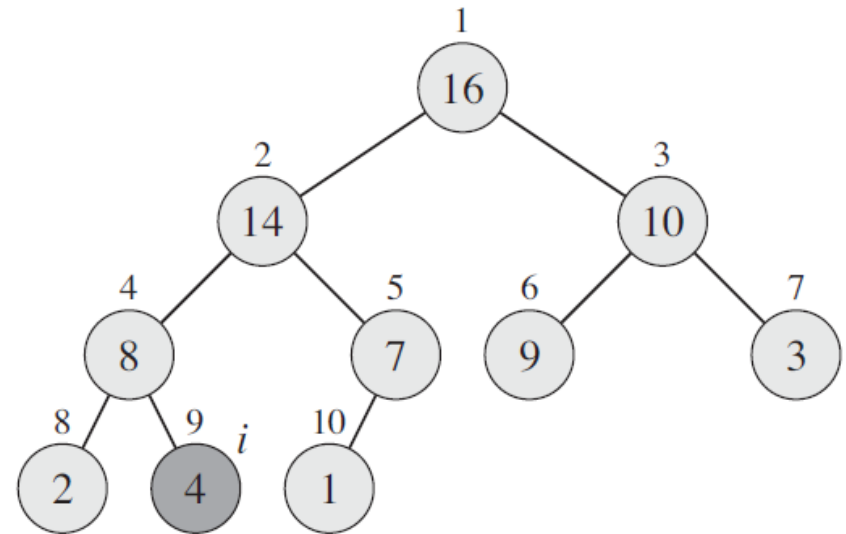
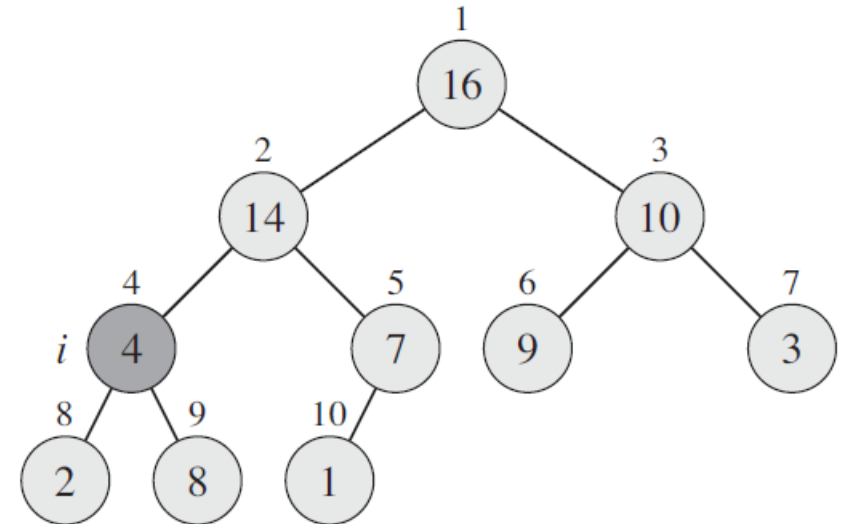
MAX-HEAPIFY Procedure

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 running time of MAX-HEAPIFY
on a subtree of size n is $O(\log n)$





Building a Max Heap

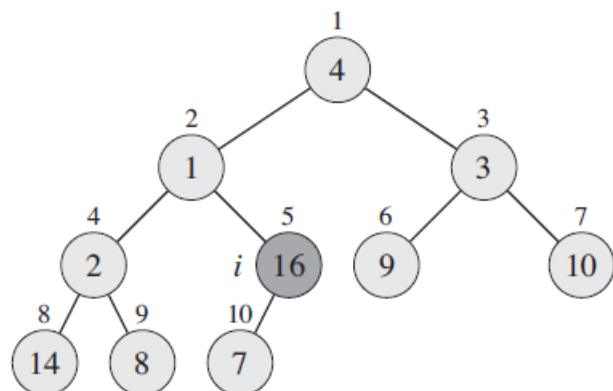
- The procedure BUILD-MAX-HEAP uses the procedure MAX-HEAPIFY in a bottom-up manner to convert an array $A[1..n]$, where $n = A.length$, into a max-heap.
- As we proved before that the elements in the subarray $A[(\lfloor n/2 \rfloor + 1), \dots, n]$ are all leaves of the tree, the procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.
- The running time of BUILD-MAX-HEAP is $O(n \log n)$

BUILD-MAX-HEAP(A)

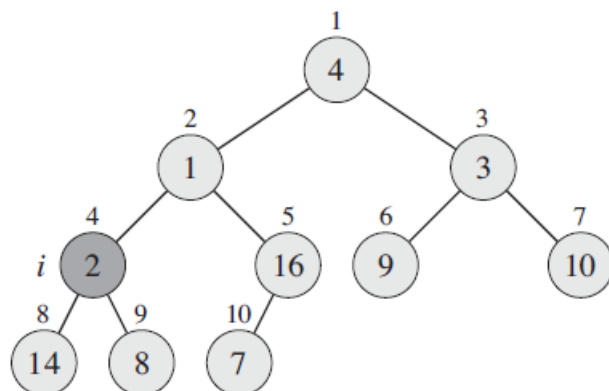
```
1   $A.heap-size = A.length$   
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1  
3      MAX-HEAPIFY( $A, i$ )
```



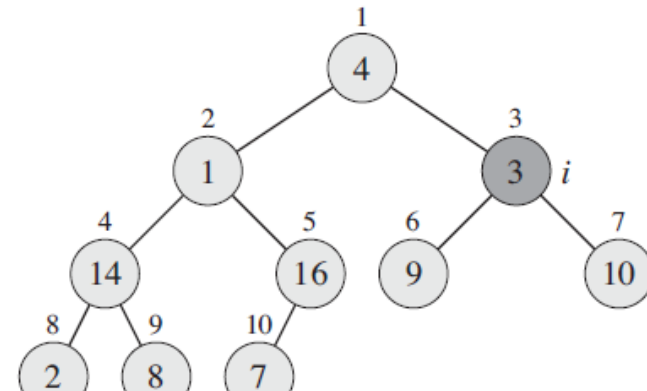
Building a Heap Example



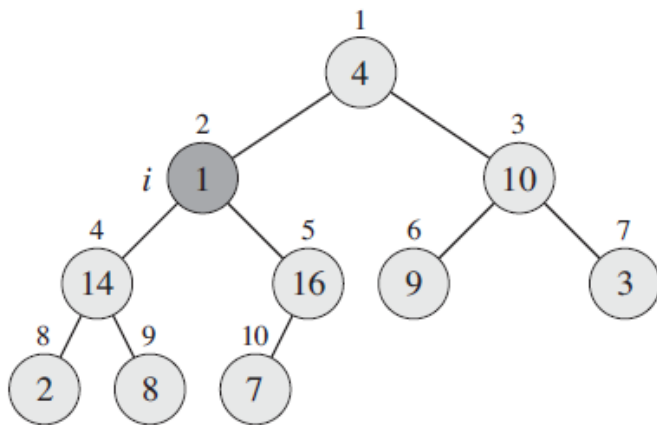
(a)



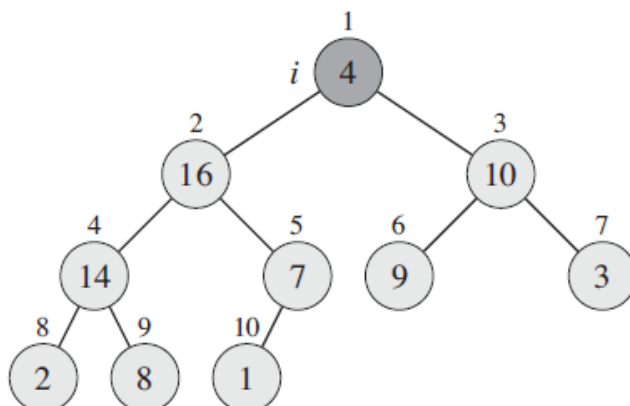
(b)



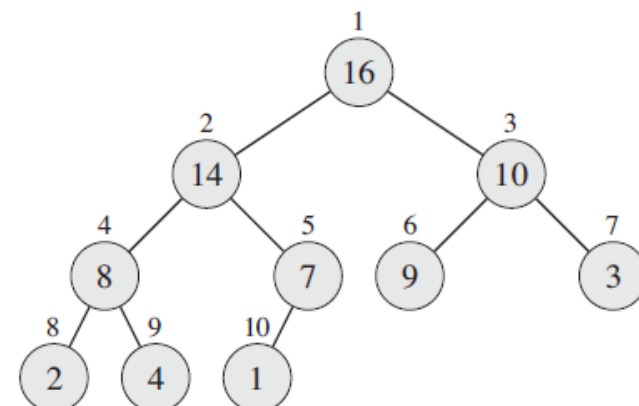
(c)



(d)



(e)



(f)



The Heapsort Algorithm

- The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array $A[1..n]$, where $n = A.length$.
- Since the maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position by exchanging it with $A[n]$.
- If we now discard node n by decrementing $A.heap-size$ and restore the max-heap property by calling MAX-HEAPIFY($A, 1$)
- Then repeats this process.
- The HEAPSORT procedure takes time $O(n \log n)$

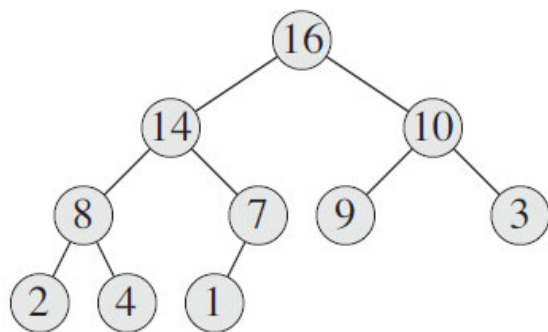
HEAPSORT(A)

```

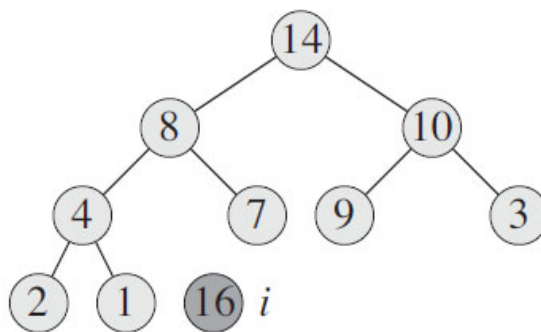
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
  
```



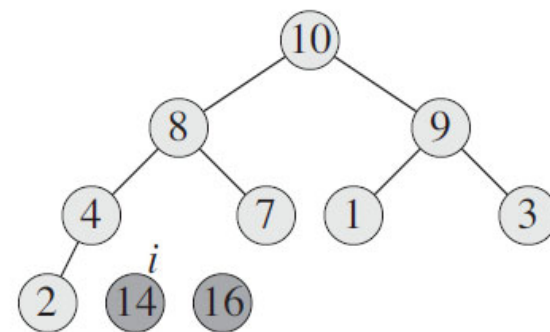

Heapsort Algorithm Example



(a)

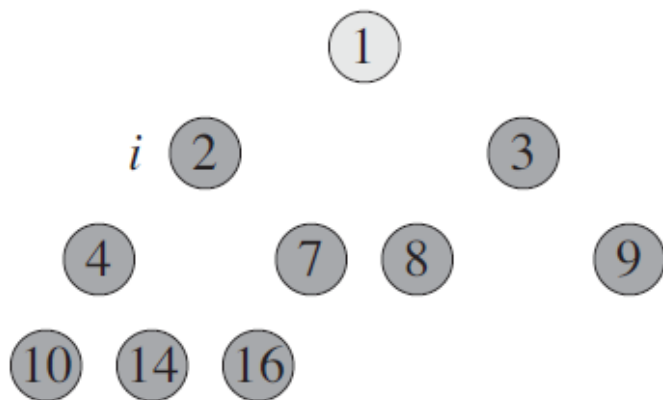


(b)



(c)

▪
▪
▪



(j)

A

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

(k)



The Quicksort Algorithm

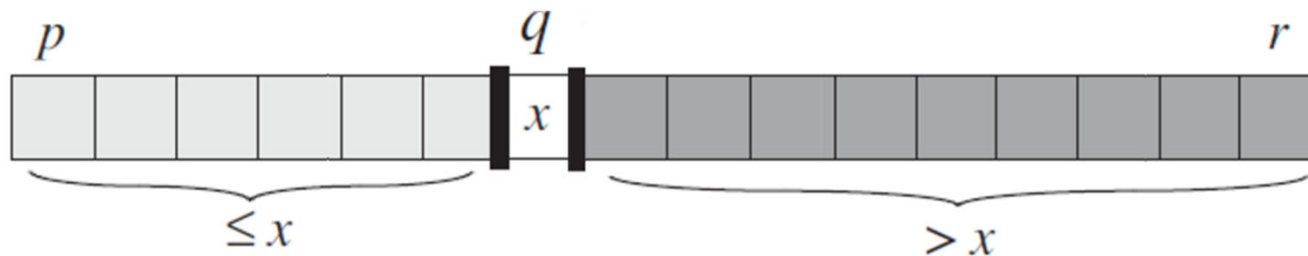
- Quicksort, like merge sort, applies the **divide-and-conquer** paradigm
 - **Divide:** Partition the array $A[p .. r]$ into two subarrays $A[p .. q-1]$ and $A[q+1 .. r]$ such that each element of $A[p .. q-1]$ is less than or equal to $A[q]$ and $A[q]$ is less than or equal to each element of $A[q+1 .. r]$. 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.



The Quicksort Procedure

QUICKSORT(A, p, r)

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





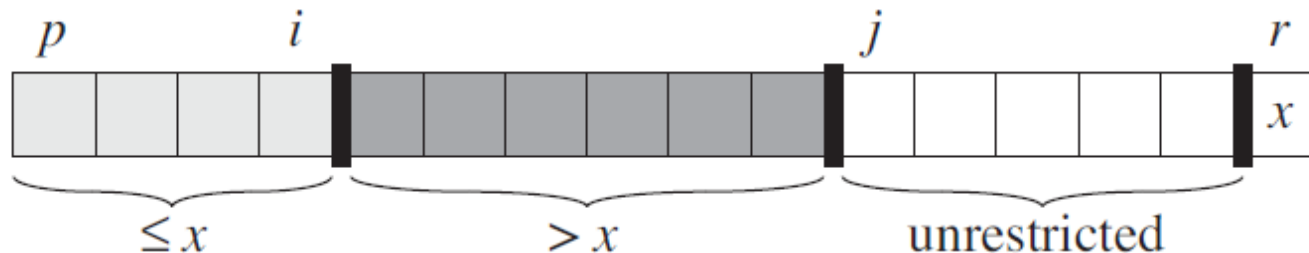
The Partition Procedure

- PARTITION selects an element $x = A[r]$ as the **pivot** element around which to partition the subarray $A[p..r]$
- As the procedure runs, it partitions the array into four regions.

PARTITION(A, p, r)

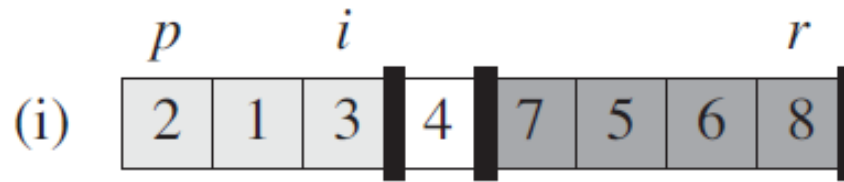
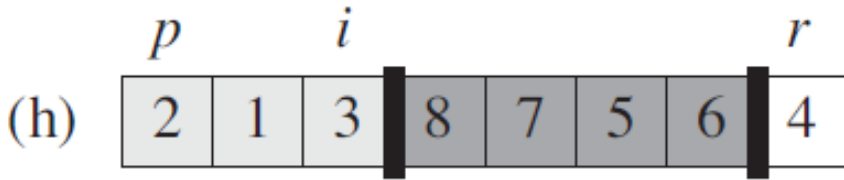
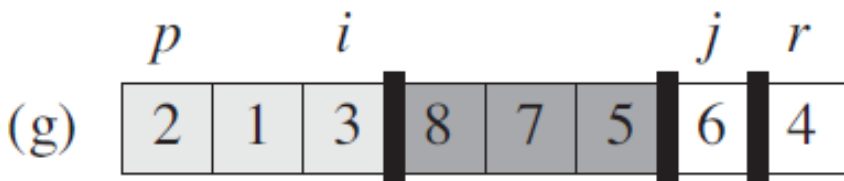
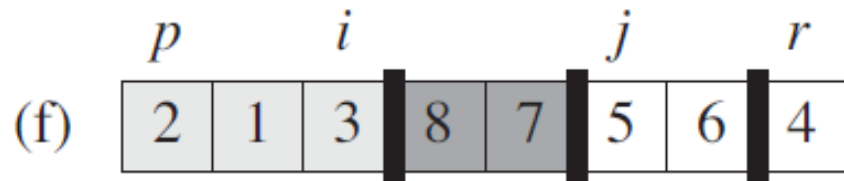
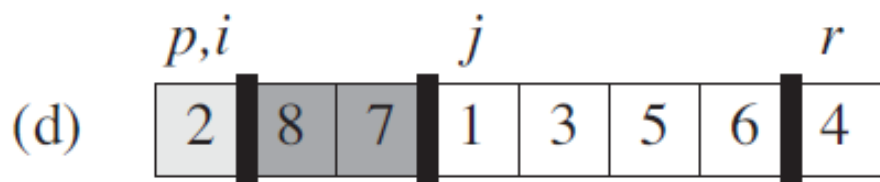
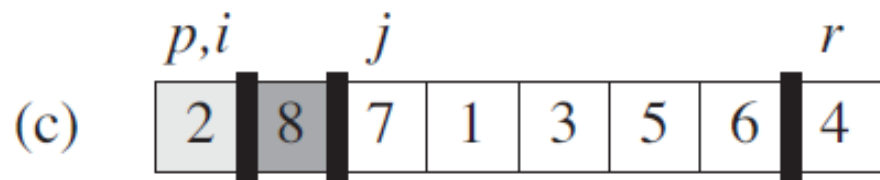
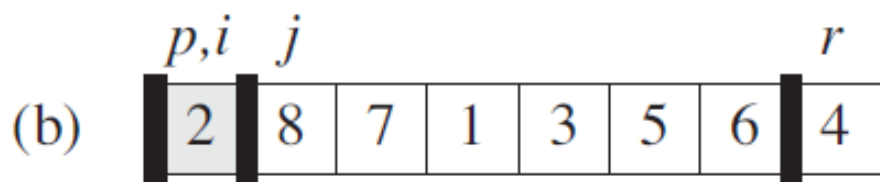
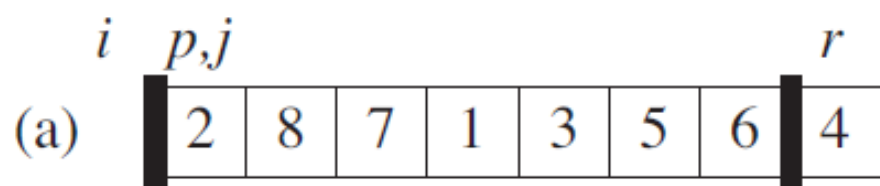
```

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  $i + 1$ 
  
```





PARTITION Operation Example





Quicksort Efficiency

- The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers.
- Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average.
- Its expected running time is $\Theta(n \log n)$ and the constant factors hidden in the $\Theta(n \log n)$ notation are quite small.
- It also has the advantage of sorting **in place**, and it works well even in virtual-memory environments as it helps in minimizing page replacement by increasing data locality.



Sorting in Linear Time

- We have now introduced several algorithms that can sort n numbers in $O(n \log n)$ time.
 - Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average.
- These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements*. We call such sorting algorithms **comparison sorts**.
- Other sorting algorithms (e.g., counting sort, radix sort, and bucket sort) run in linear time. These algorithms use operations other than comparisons to determine the sorted order.



Counting Sort

- **Counting sort** assumes that each of the n input elements is an integer in the range 0 to k , for some integer k . When $k = O(n)$, the sort runs in $\Theta(n)$ time.
- Counting sort determines, for each input element x , the number of elements less than x . It uses this information to place element x directly into its position in the output array.
- In the algorithm procedure, we assume that the input is an array $A[1 .. n]$, and thus $A.length = n$.
- We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[0..k]$ provides temporary working storage to hold the occurrences of each element in A .



Counting Sort Procedure

COUNTING-SORT(A, B, k)

```

1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  = number of
   // elements equal to  $i$ .
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9  //  $C[i]$  = number of
   // elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 

```

	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

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

	1	2	3	4	5	6	7	8
B							3	

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



Counting Sort Stability

- An important property of counting sort is that it is *stable*: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array.
- The property of stability is important in a case like given an array of students sorted by name and re-sort it by year of graduation. Using a stable sorting algorithm to sort the array by year will ensure that within each year the students will remain sorted by name.



Counting Sort Analysis

- Given a list of n integers to sort where these integers range from 0 to k .
- The four loops in the algorithm require a total of $2n + 2k$ iterations.
- For a k that is $O(n)$, the counting sort runs in $\Theta(n)$ time