Sorting

Lecture Outline

- Selection sort
- Bubble sort
- Insertion sort
- Shell sort
- Merge sort
- Heapsort
- Quicksort

Lecture Outline (2)

- Understand the performance of these algorithms
  - Which to use for small arrays
  - Which to use for medium arrays
  - Which to use for large arrays

Conventions of Presentation

- Write algorithms for arrays of **`Comparable`** objects
- For convenience, examples show integers
  - These would be wrapped as **`Integer`**; or
  - You can implement separately for **`int`** arrays
- Generally use *n* for the length of the array
  - Elements *0* through *n-1*

Selection Sort

- A relatively easy to understand algorithm
- Sorts an array in *passes*
  - Each pass selects the next smallest element
  - At the end of the pass, places it where it belongs
- Efficiency is $O(n^2)$, hence called a *quadratic* sort
- Performs:
  - $O(n^2)$ comparisons
  - O(n) exchanges (swaps)

Selection Sort Example

35	65	30	60	20	scan 0-4, smallest 20
					swap 35 and 20
20	65	30	60	35	scan 1-4, smallest 30
					swap 65 and 30
20	30	65	60	35	scan 2-4, smallest 35
					swap 65 and 35
20	30	35	60	65	scan 3-4, smallest 60
					swap 60 and 60
20	30	35	60	65	done

Selection Sort Algorithm

1. for **fill** = 0 to n-2 do // steps 2-6 form a *pass*
2. set **posMin** to **fill**
3. for **next** = **fill**+1 to n-1 do
4. if item [**next]** < item [**posMin]**
5. set **posMin** to **next**
6. Exchange item at **posMin** with one at **fill**

Bubble Sort

- *Compares adjacent* array elements
  - *Exchanges* their values if they are *out of order*

- Smaller values *bubble up* to the top of the array
  - Larger values sink to the bottom

Bubble Sort Example

FIGURE 10.1
One Pass of Bubble Sort

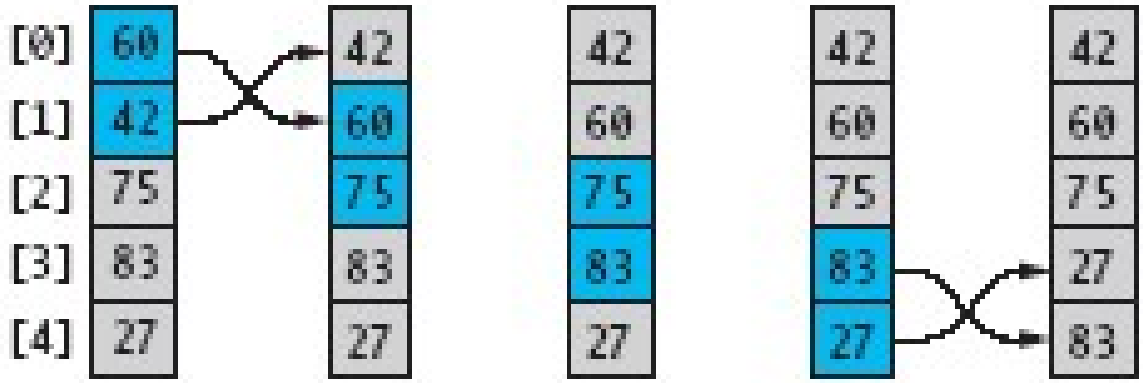
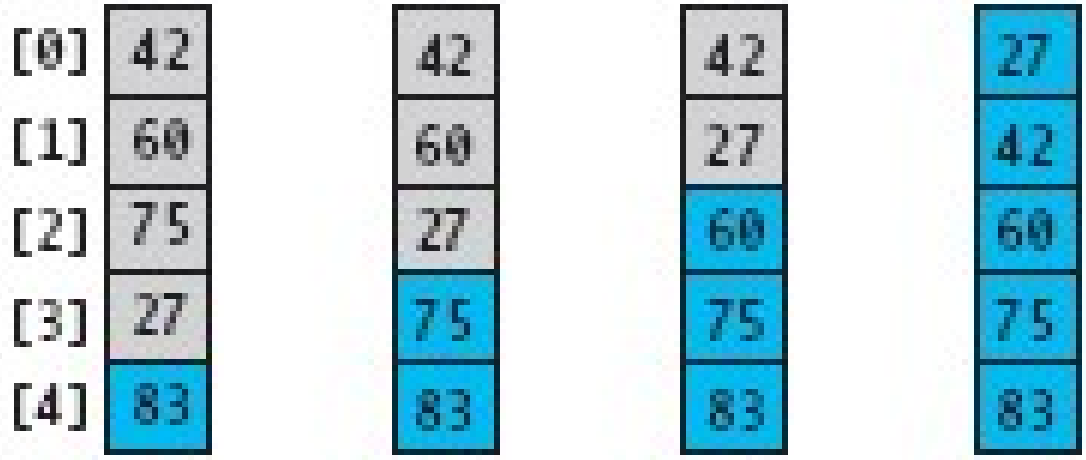


FIGURE 10.2
Array After Completion of Each Pass



Bubble Sort Algorithm

1. do
2. Initialize **exchanges** to **false**
3. for each pair of adjacent array elements
4. if values are out of order
5. Exchange the values
6. Set **exchanges** to **true**
7. while **exchanges**

Analysis of Bubble Sort

- Excellent performance *in some cases*
  - But *very poor* performance in others!
- Works ***best*** when array is nearly sorted to begin with
- Worst case number of comparisons: $O(n^2)$
- Worst case number of exchanges: $O(n^2)$
- *Best case* occurs when the array is already sorted:
  - O(n) comparisons
  - O(1) exchanges (none actually)

Insertion Sort

- Based on technique of card players to arrange a hand
  - Player keeps cards picked up so far in *sorted order*
  - When the player picks up a new card
    - Makes room for the new card
    - Then inserts it in its proper place

FIGURE 10.3
Picking Up a Hand of Cards

Insertion Sort Algorithm

- For each element from 2nd (nextPos = 1) to last:
  - Insert element at nextPos where it belongs
  - Increases sorted subarray size by 1

- To make room:
  - Hold nextPos value in a variable
  - Shuffle elements to the right until gap at right place

Insertion Sort Example

FIGURE 10.5
Inserting the Fourth Array Element

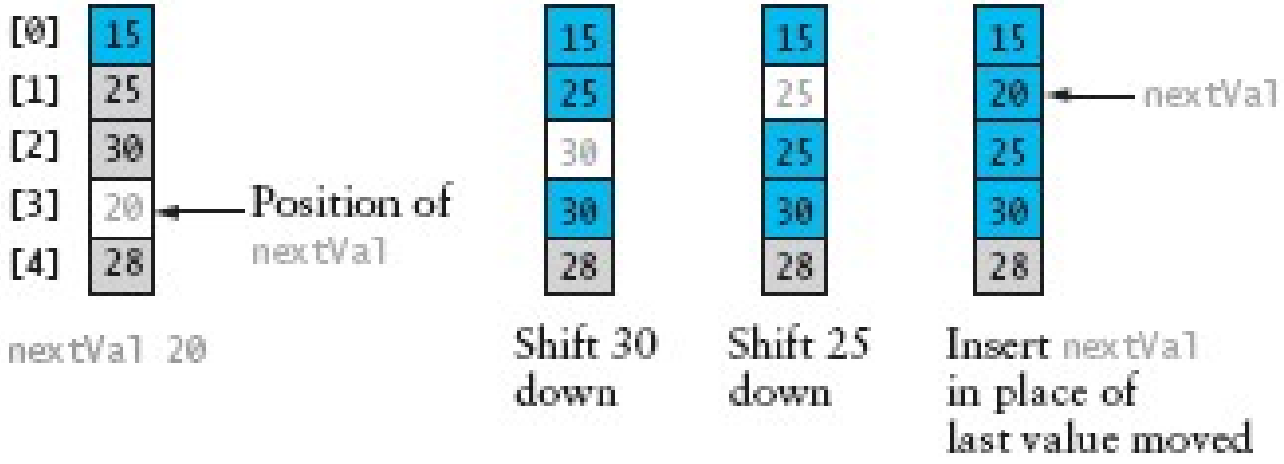
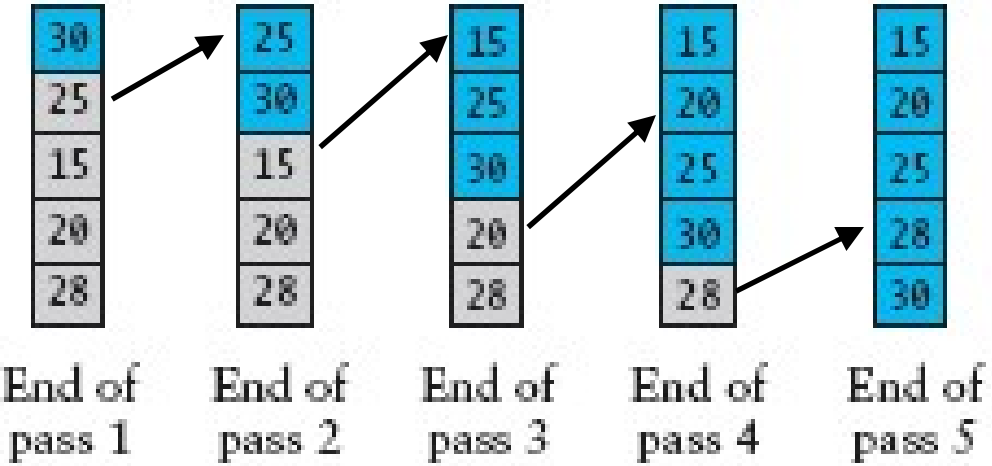


FIGURE 10.4
An Insertion Sort



Insertion Sort Code

```
public static <T extends Comparable<T>>
    void sort (T[] a) {
  for (int nextPos = 1;
       nextPos < a.length;
       nextPos++) {
    insert(a, nextPos);
  }
}
```

Insertion Sort Code (2)

```
private static <T extends Comparable<T>>
    void insert (T[] a, int nextPos) {
  T nextVal = a[nextPos];
  while
    (nextPos > 0 &&
     nextVal.compareTo(a[nextPos-1]) < 0){
    a[nextPos] = a[nextPos-1];
    nextPos--;
  }
  a[nextPos] = nextVal;
}
```

Analysis of Insertion Sort

- Maximum number of comparisons: $O(n^2)$
- In the best case, number of comparisons: O(n)
- # shifts for an insertion = # comparisons - 1
  - When new value smallest so far, # comparisons
- A shift in insertion sort moves only *one item*
  - Bubble or selection sort exchange: *3 assignments*

Comparison of Quadratic Sorts

- None good for large arrays!

TABLE 10.2
Comparison of Quadratic Sorts

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

TABLE 10.3
Comparison of Rates of Growth

n	n^2	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

Shell Sort: A Better Insertion Sort

- Shell sort is a variant of insertion sort
  - It is named after Donald Shell
  - Average performance: $O(n^{3/2})$ or better
- Divide and conquer approach to insertion sort
  - Sort many smaller subarrays using insertion sort
  - Sort progressively larger arrays
  - Finally sort the entire array
- These arrays are elements separated by a *gap*
  - Start with *large gap*
  - Decrease the gap on each “pass”

Shell Sort: The Varying Gap

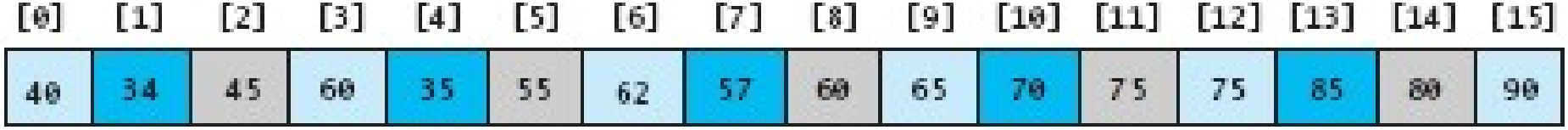
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	
Input data	62	83	18	53	07	17	95	86	47	69	25	28
After 5-sorting	17	28	18	47	07	25	83	86	53	69	62	95
After 3-sorting	17	07	18	47	28	25	69	62	53	83	86	95
After 1-sorting	07	17	18	25	28	47	53	62	69	83	86	95

- The first pass, 5-sorting, performs insertion sort on (a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10}). For instance, it changes the subarray (a_1, a_6, a_{11}) from (62, 17, 25) to (17, 25, 62).
- The next pass, 3-sorting, performs insertion sort on (a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12}).
- The last pass, 1-sorting, is an ordinary insertion sort of the entire array (a_1,..., a_{12}).

Shell Sort: The Varying Gap

Before and after sorting with gap = 7

Before and after sorting with gap = 3



Analysis of Shell Sort

- ***Intuition:***

  Reduces work by moving elements farther earlier
- Its general analysis is an open research problem
- Performance depends on sequence of gap values
- For sequence 2^k, performance is $O(n^2)$
- Hibbard's sequence (2^k-1), performance is $O(n^{3/2})$
- We start with n/2 and repeatedly divide by 2.2
  - Empirical results show this is $O(n^{5/4})$ or $O(n^{7/6})$
  - No *theoretical* basis (proof) that this holds

Shell Sort Algorithm

1. Set **gap** to n/2
2. while **gap** > 0
3. for each element from **gap** to end, by **gap**
4. Insert element in its **gap**-separated sub-array
5. if **gap** is 2, set it to 1
6. otherwise set it to **gap** / 2.2

Shell Sort Algorithm: Inner Loop

3.1 set **nextPos** to position of element to insert

3.2 set **nextVal** to value of that element

3.3 while **nextPos** > **gap** and
element at **nextPos-gap** is > **nextVal**

3.4 Shift element at **nextPos-gap** to **nextPos**

3.5 Decrement **nextPos** by **gap**

3.6 Insert **nextVal** at **nextPos**

Merge Sort

- A *merge* is a common data processing operation:
  - Performed on *two sequences* of data
    - Items in both sequences use same **`compareTo`**
    - Both sequences in ordered of this **`compareTo`**
- **Goal:** Combine the two sorted sequences in one larger sorted sequence

- *Merge sort* merges longer and longer sequences

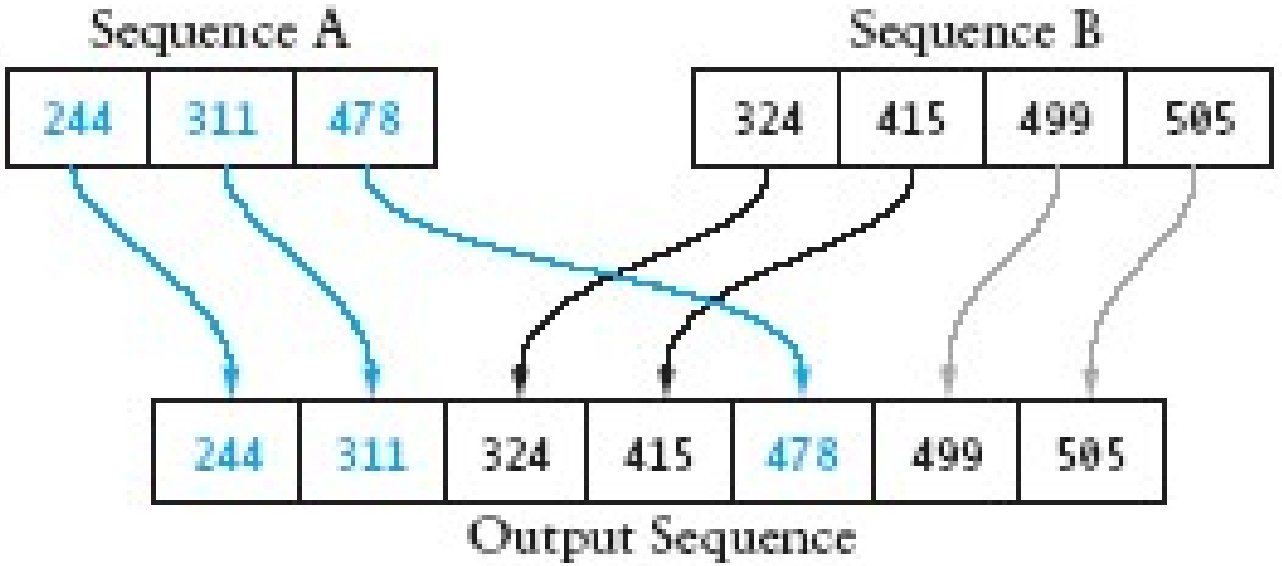
Merge Algorithm (Two Sequences)

Merging two sequences:

1. Access the first item from both sequences
2. While neither sequence is finished
    1. Compare the current items of both
    2. Copy smaller current item to the output
    3. Access next item from that input sequence
3. Copy any remaining from first sequence to output
4. Copy any remaining from second to output

Picture of Merge

FIGURE 10.6
Merge Operation



Analysis of Merge

- Two input sequences, total length n elements
  - Must move each element to the output
  - Merge time is O(n)

- Must store *both input* and *output* sequences
  - An array cannot be merged *in place*
  - Additional space needed: O(n)

Merge *Sort* Algorithm

Overview:

- Split array into two halves
- Sort the left half (recursively)
- Sort the right half (recursively)
- Merge the two sorted halves

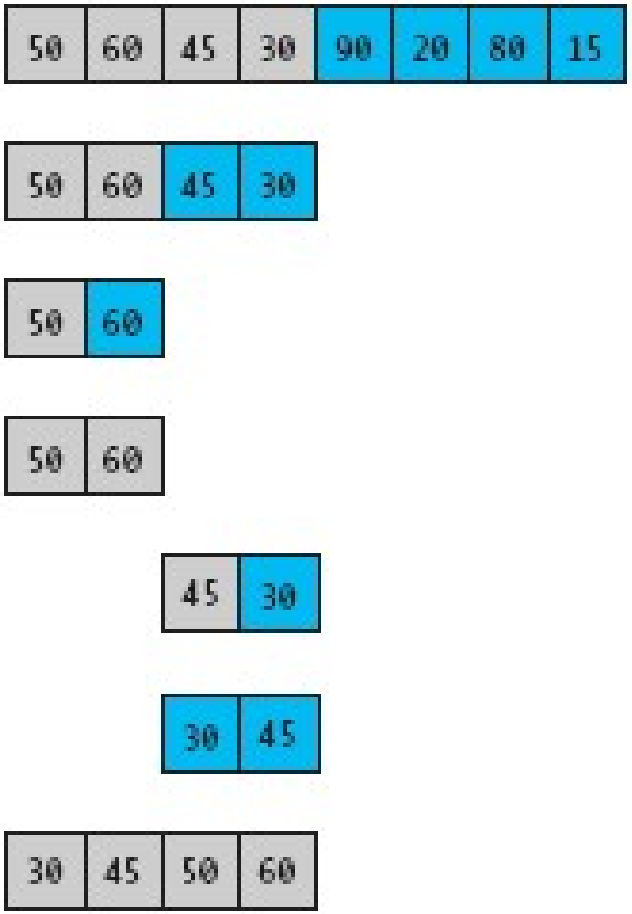
Merge *Sort* Algorithm (2)

Detailed algorithm:

- if **tSize** $\leq$ 1, return (no sorting required)
- set **hSize** to **tSize** / 2
- Allocate **LTab** of size **hSize**
- Allocate **RTab** of size **tSize** – **hSize**
- Copy elements 0 .. **hSize** – 1 to **LTab**
- Copy elements **hSize** .. **tSize** – 1 to **RTab**
- Sort **LTab** recursively
- Sort **RTab** recursively
- Merge **LTab** and **RTab** into **a**

Merge Sort Example

FIGURE 10.7
Trace of Merge Sort



1. *Split array into two 4-element arrays*
2. *Split left array into two 2-element arrays*
3. *Split left array (50, 60) into two 1-element arrays*
4. *Merge two 1-element arrays into a 2-element array*
5. *Split right array from Step 2 into two 2-element arrays*
6. *Merge two 1-element arrays into a 2-element array*
7. *Merge two 2-element arrays into a 4-element array*

Merge Sort Analysis

- Splitting/copying n elements to subarrays: O(n)
- Merging back into original array: O(n)
- Recursive calls: 2, each of size n/2
  - Their total non-recursive work: O(n)
- Next level: 4 calls, each of size n/4
  - Non-recursive work again O(n)
- Size sequence: n, n/2, n/4, ..., 1
  - Number of levels = log n
  - Total work: O(n log n)

Merge Sort Code

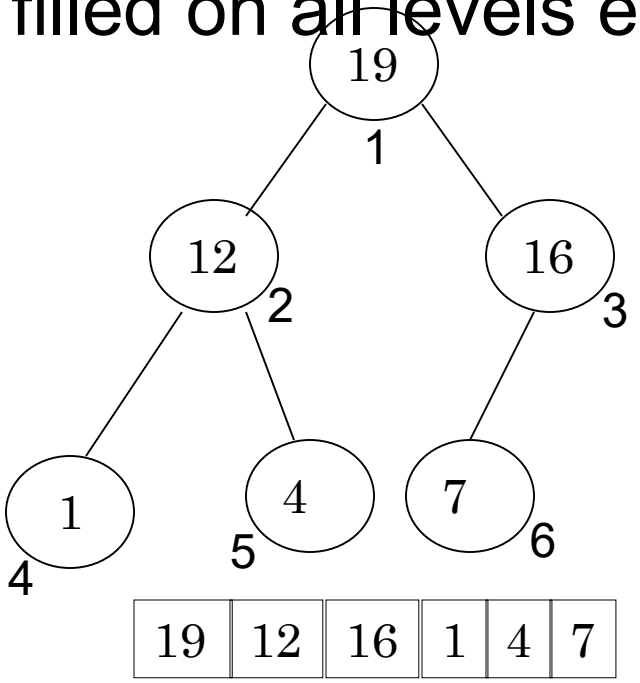
```
public static <T extends Comparable<T>>
    void sort (T[] a) {
  if (a.length <= 1) return;
  int hSize = a.length / 2;
  T[] lTab = (T[])new Comparable[hSize];
  T[] rTab =
    (T[])new Comparable[a.length-hSize];
  System.arraycopy(a, 0, lTab, 0, hSize);
  System.arraycopy(a, hSize, rTab, 0,
                   a.length-hSize);
  sort(lTab); sort(rTab);
  merge(a, lTab, rTab);
}
```

Merge Sort Code (2)

```
private static <T extends Comparable<T>>
    void merge (T[] a, T[] l, T[] r) {
  int i = 0;  // indexes l
  int j = 0;  // indexes r
  int k = 0;  // indexes a
  while (i < l.length && j < r.length)
    if (l[i].compareTo(r[j]) < 0)
      a[k++] = l[i++];
    else
      a[k++] = r[j++];
  while (i < l.length) a[k++] = l[i++];
  while (j < r.length) a[k++] = r[j++];
}
```

Heap

- The binary heap data structures is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is filled on all levels except possibly lowest.



Array A

Heap

- The root of the tree A[1] and given index *i* of a node, the indices of its parent, left child and right child can be computed

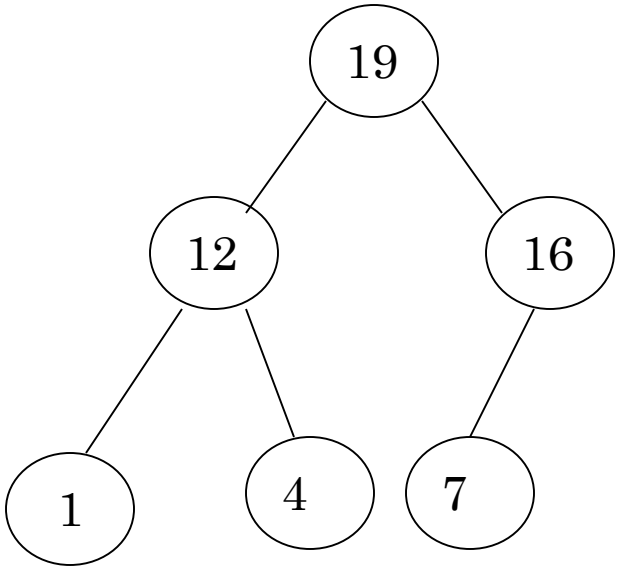
```
PARENT (i)
        return floor(i/2)
LEFT (i)
        return 2i
RIGHT (i)
        return 2i + 1
```

Heap order property

- For every node *v*, other than the root, the key stored in *v* is smaller or equal (greater or equal for max heap) than the key stored in the parent of *v*.

- In this case the maximum value is stored in the root

Max Heap Example



19	12	16	1	4	7

Array A

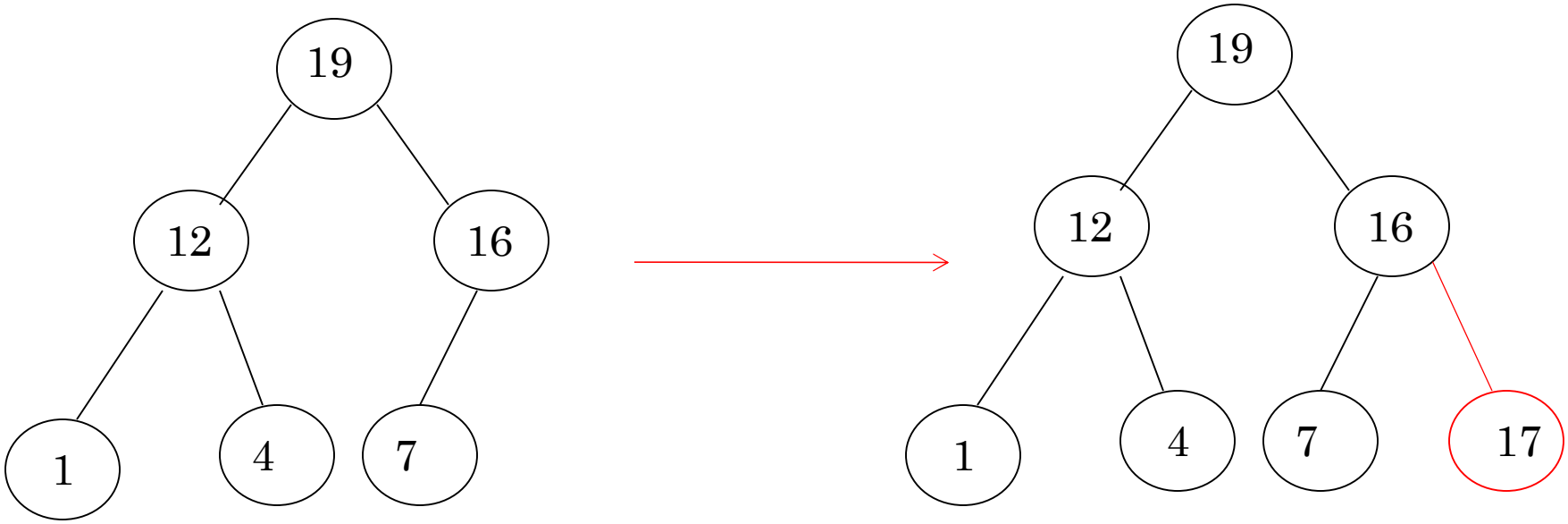
Insertion

- Algorithm
  1. Add the new element to the next available position at the lowest level
  2. Restore the max-heap property if violated
     - General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child

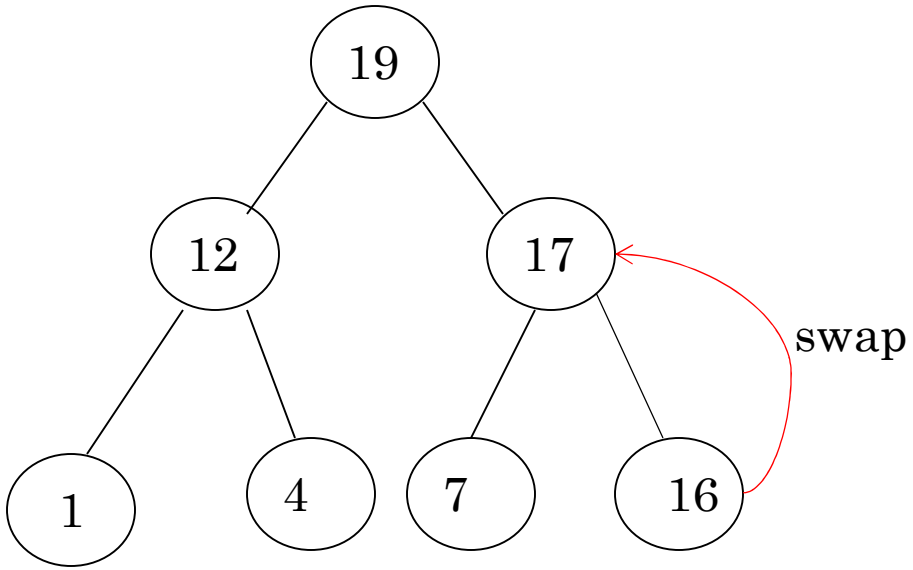
OR

Restore the min-heap property if violated

- General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.



Insert 17



Percolate up to maintain the heap property

Deletion

- Delete max
  - Copy the last number to the root (overwrite the maximum element stored there).
  - Restore the max heap property by percolate down.

- Delete min
  - Copy the last number to the root (overwrite the minimum element stored there).
  - Restore the min heap property by percolate down.

Heap Sort

A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap

Procedures on Heap

- Heapify
- Build Heap
- Heap Sort

Heapify

- Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

```
Heapify(A, i)
{
    l ← left(i)
    r ← right(i)
    if l <= heapsize[A] and A[l] > A[i]
       then largest ←l
       else largest ← i
    if r <= heapsize[A] and A[r] > A[largest]
       then largest ← r
    if largest != i
       then swap A[i] ←→ A[largest]
          Heapify(A, largest)
}
```

Build Heap

- We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

```
Buildheap(A)
{
      heapsize[A] ←length[A]
      for i ← length[A]/2  down to 1
         do Heapify(A, i)
}
```

Heap Sort Algorithm

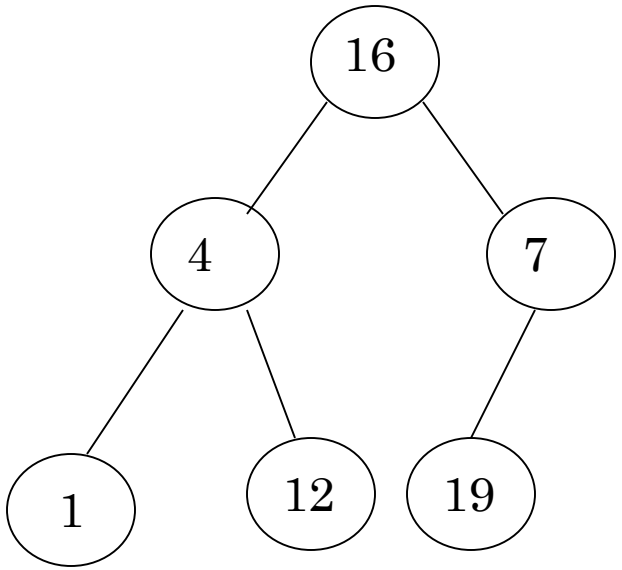
- The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . *n*]. Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

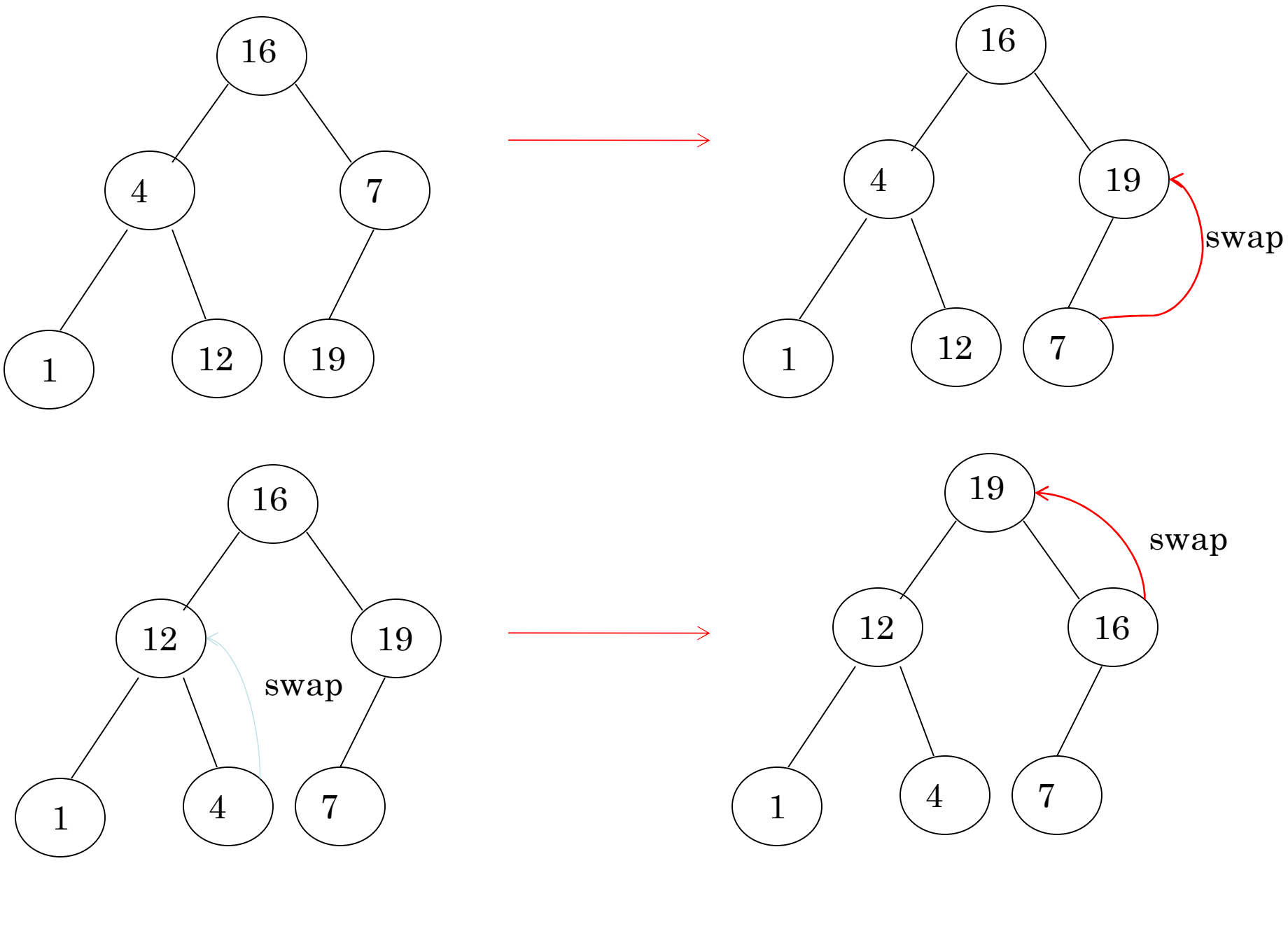
```
Heapsort(A)
{
        Buildheap(A)
        for i ← length[A] //down to 2
            do swap A[1] ←→ A[i]
            heapsize[A] ← heapsize[A] - 1
            Heapify(A, 1)
}
```

Example: Convert the following array to a heap

16	4	7	1	12	19

Picture **the array as a complete binary tree:**

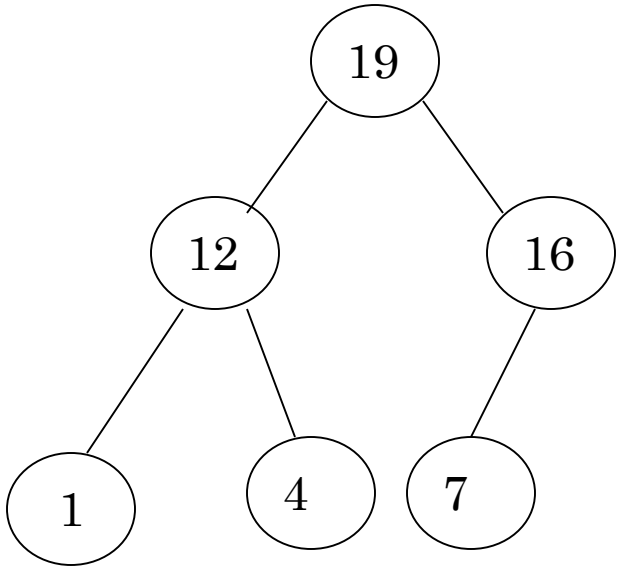



16
4
7
1
12
19
16
4
19
swap
1
12
7
16
12
19
swap
1
4
7
19
swap
12
16
1
4
7

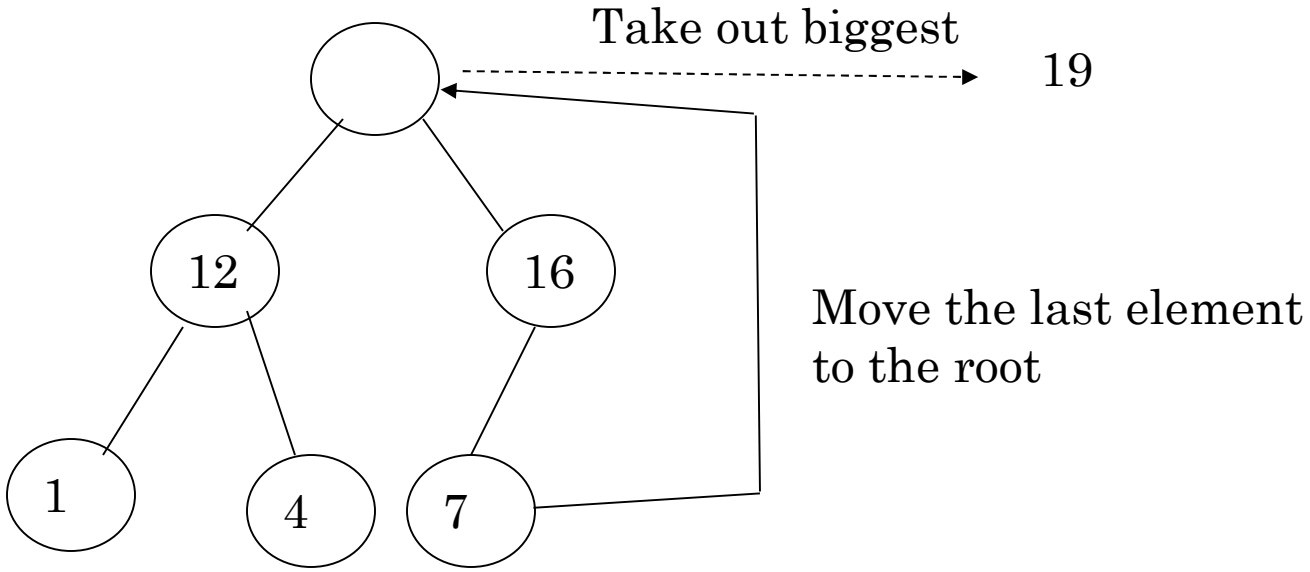
Heap Sort

- The heapsort algorithm consists of two phases:
  - build a heap from an arbitrary array
  - use the heap to sort the data

- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap



Example of Heap Sort



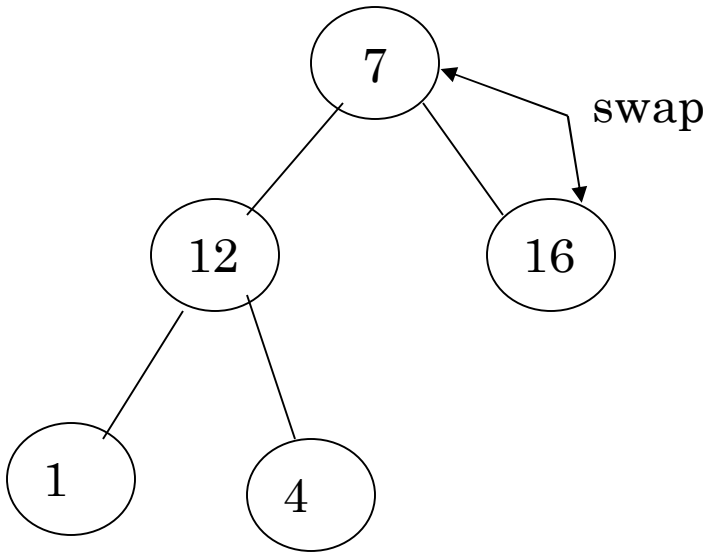
Array A

12	16	1	4	7

Sorted:

19

HEAPIFY()

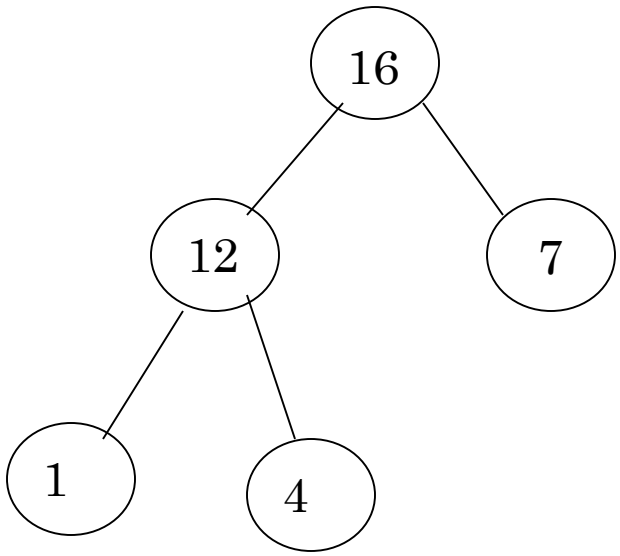


Sorted:

Array A

7	12	16	1	4

19

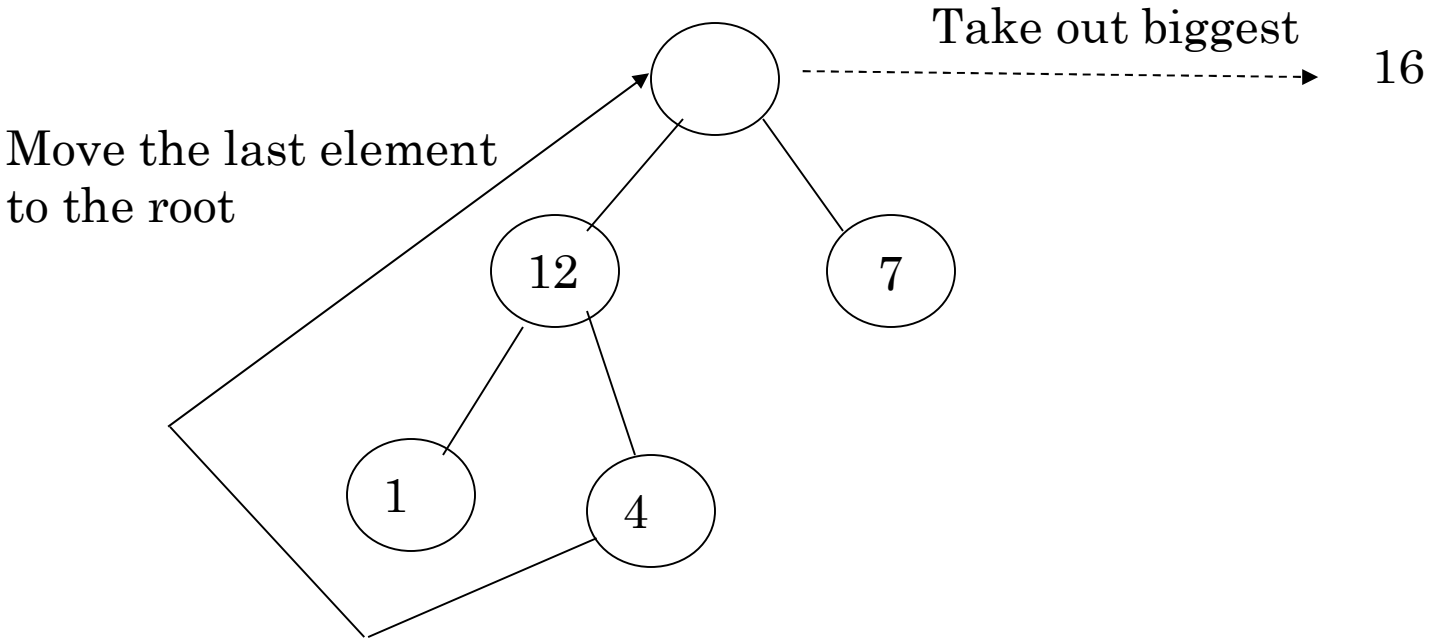


Sorted:

Array A

16	12	7	1	4

19

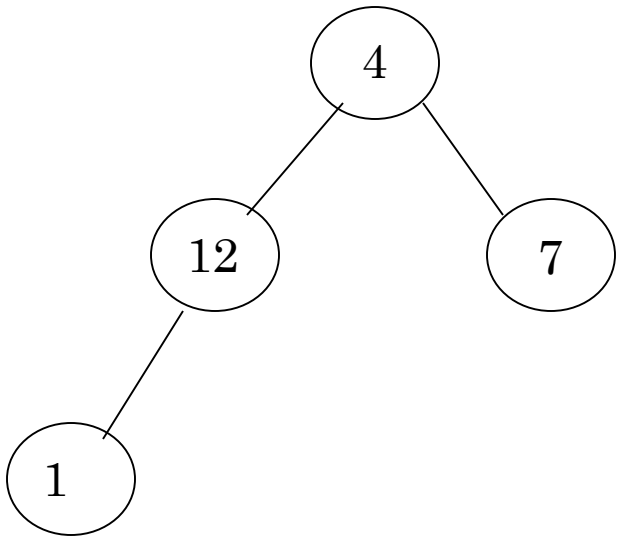


Sorted:

Array A

12	7	1	4

16	19



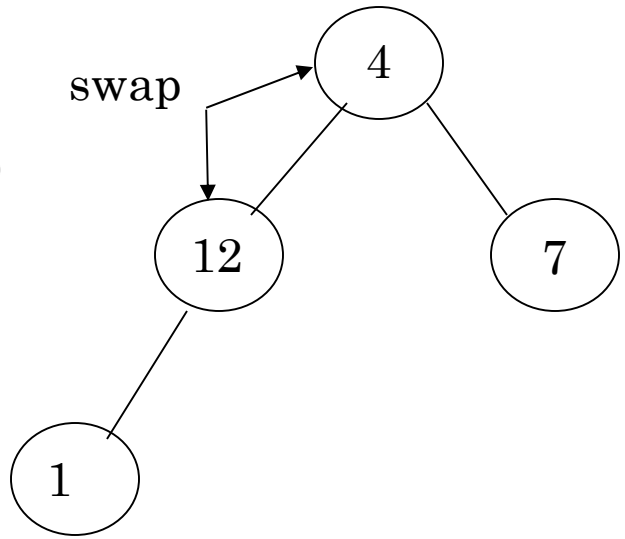
Array A

4	12	7	1

Sorted:

16	19

HEAPIFY()

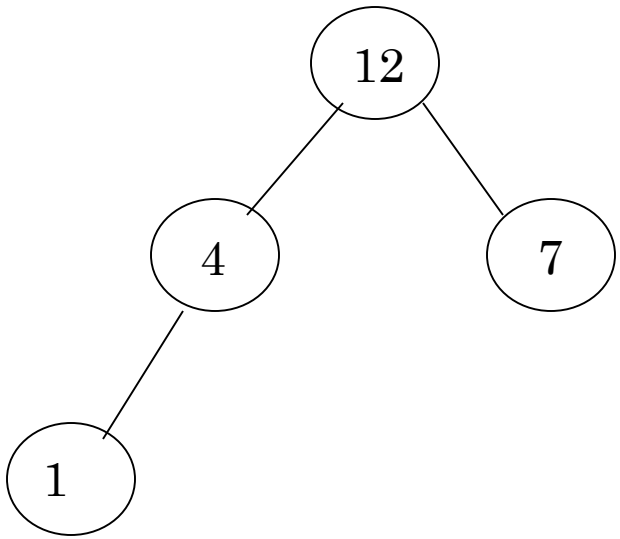


Array A

4	12	7	1

Sorted:

16	19

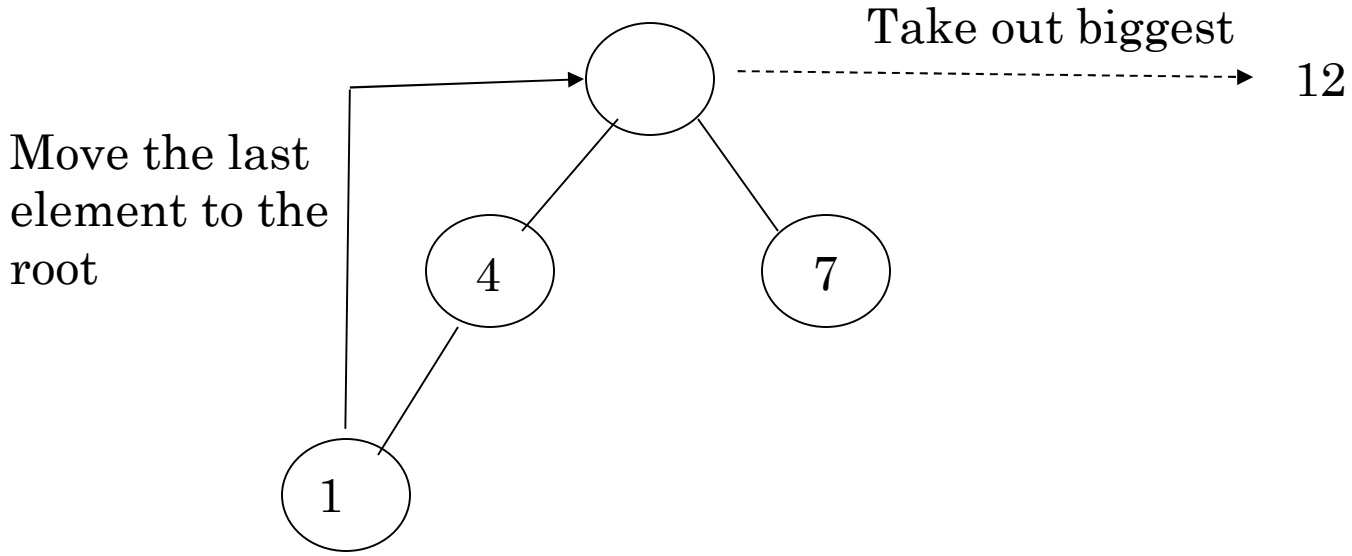


Array A

12	4	7	1

Sorted:

16	19

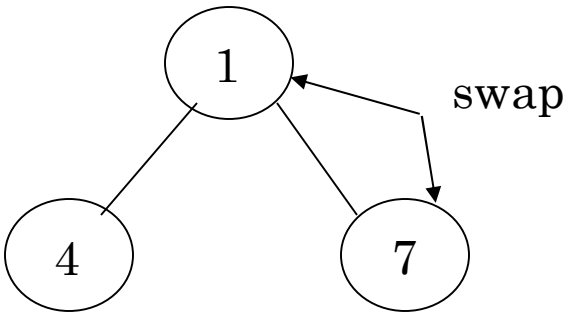


Array A

4	7	1

Sorted:

12	16	19

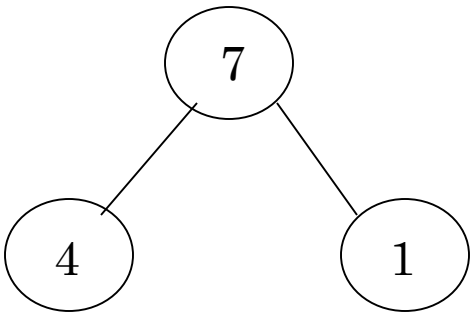


Array A

1	4	7

Sorted:

12	16	19

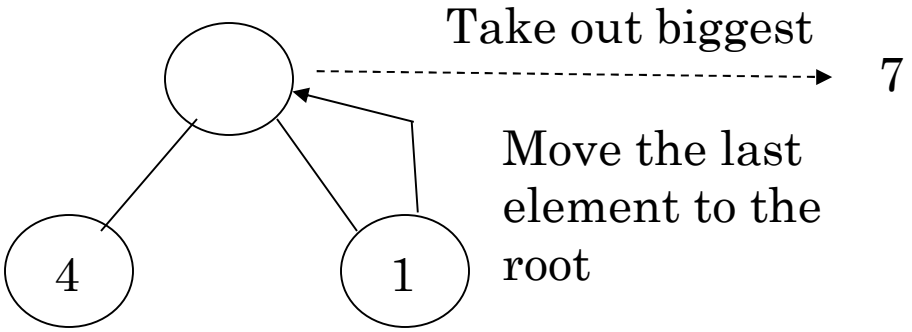


Array A

7	4	1

Sorted:

12	16	19



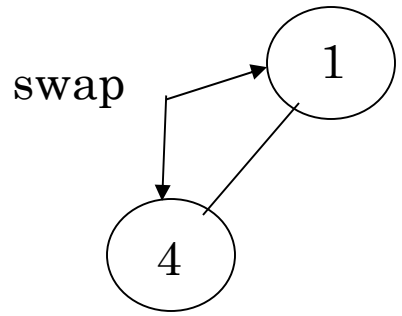
Array A

1	4

Sorted:

7	12	16	19

HEAPIFY()

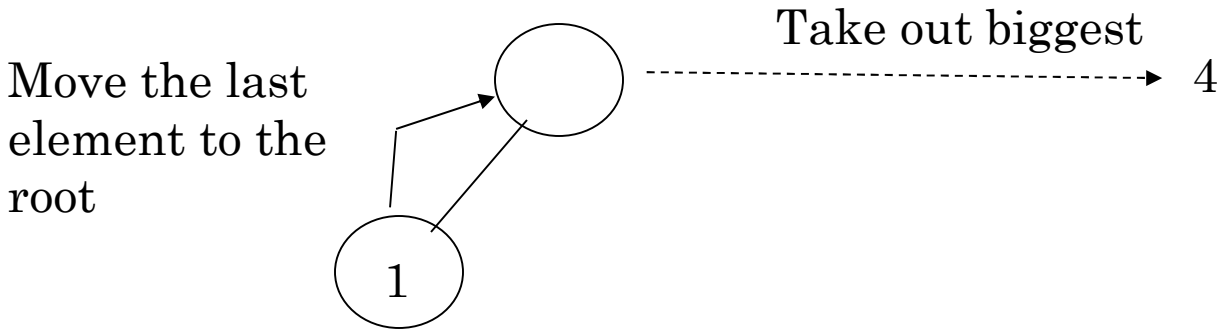


Array A

4	1

Sorted:

7	12	16	19

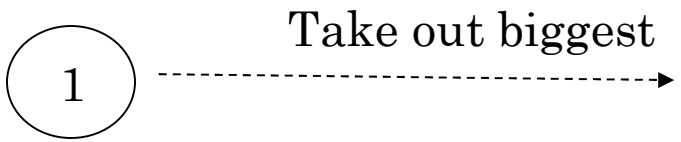


Array A

Sorted:

1

4	7	12	16	19

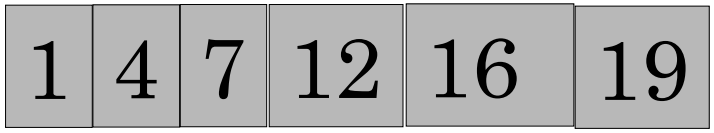


Array A

Sorted:

1	4	7	12	16	19

Sorted:



Heapsort

- Merge sort time is O(n log n)
  - ***But*** requires (temporarily) n extra storage items
- *Heapsort*
  - Works *in place:* no additional storage
  - Offers same O(n log n) performance

Heapsort Analysis

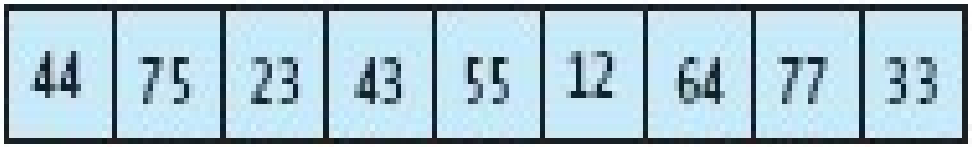
- Insertion cost is log i for heap of size i
  - Total insertion cost = log(n)+log(n-1)+...+log(1)
  - This is O(n log n)
- Removal cost is also log i for heap of size i
  - Total removal cost = O(n log n)

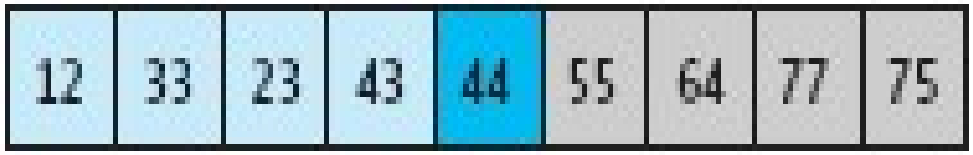
- Total cost is O(n log n)

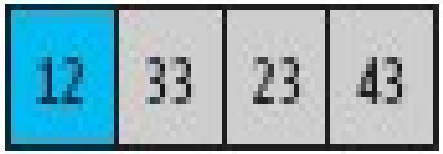
Quicksort

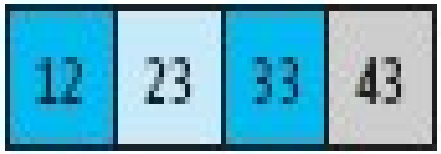
- Developed in 1962 by C. A. R. Hoare
- Given a *pivot value:*
  - Rearranges array into two parts:
    - Left part $\leq$ pivot value
    - Right part > pivot value
- Average case for Quicksort is O(n log n)
  - Worst case is $O(n^2)$

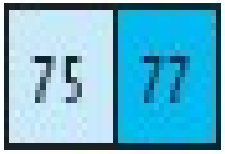
Quicksort Example











Algorithm for Quicksort

first and **last** are end points of region to sort

- if **first** < **last**
- Partition using **pivot**, which ends in **pivIndex**
- Apply Quicksort recursively to left subarray
- Apply Quicksort recursively to right subarray

Performance: O(n log n) provide **pivIndex** not always too close to the end

Performance $O(n^2)$ when **pivIndex** always near end

Quicksort Code

```
public static <T extends Comparable<T>>
    void sort (T[] a) {
  qSort(a, 0, a.length-1);
}
private static <T extends Comparable<T>>
    void qSort (T[] a, int fst, int lst) {
  if (fst < lst) {
    int pivIndex = partition(a, fst, lst);
    qSort(a, fst, pivIndex-1);
    qSort(a, pivIndex+1, lst);
  }
}
```

Algorithm for Partitioning

1. Set pivot value to a[fst]
2. Set **up** to fst and **down** to lst
3. do
4. Increment **up** until a[**up**] > pivot or **up** = lst
5. Decrement **down** until a[**down**] <= pivot or **down** = fst
6. if **up** < **down**, swap a[**up**] and a[**down**]
7. while **up** is to the left of **down**
8. swap a[fst] and a[**down**]
9. return **down** as pivIndex

Trace of Algorithm for Partitioning

FIGURE 10.14
Locating First Values to Exchange

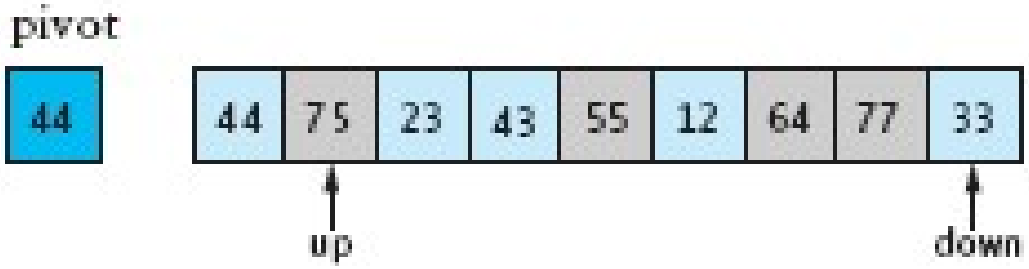


FIGURE 10.15
Array After the First Exchange

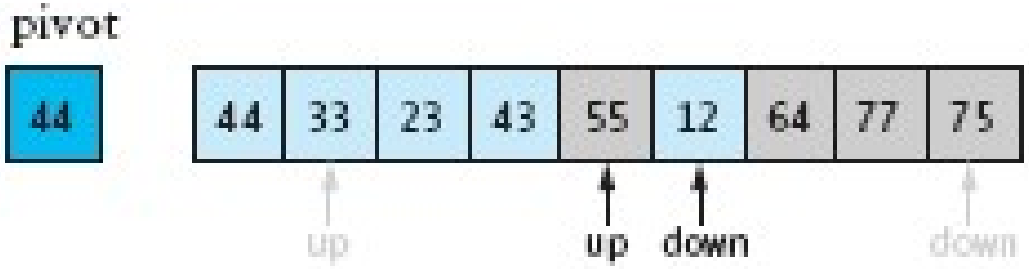
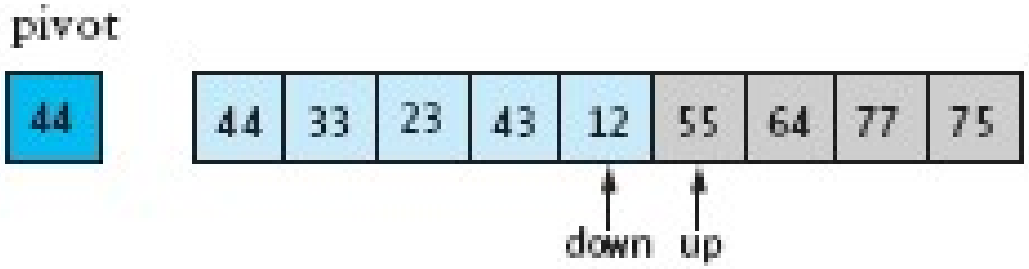


FIGURE 10.16
Array After the Second Exchange



Partitioning Code

```
private static <T extends Comparable<T>>
    int partition
       (T[] a, int fst, int lst) {
  T pivot = a[fst];
  int u = fst;
  int d = lst;
  do {
    while ((u < lst) &&
           (pivot.compareTo(a[u]) >= 0))
      u++;
    while (pivot.compareTo(a[d]) < 0)
      d++;
    if (u < d) swap(a, u, d);
  } while (u < d);
```

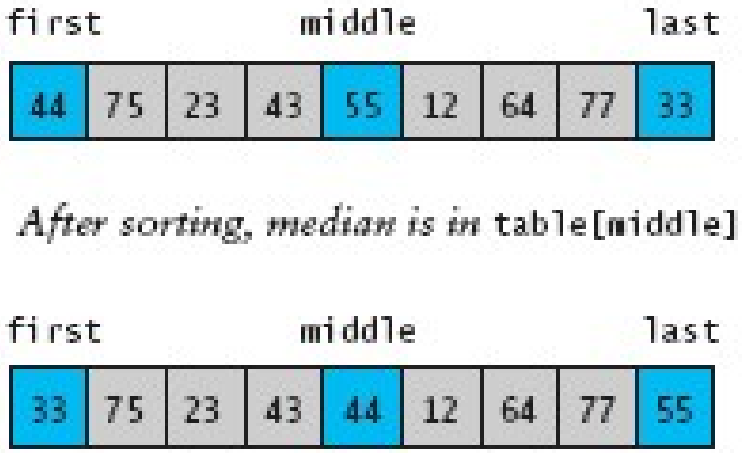
Partitioning Code (2)

```
    swap(a, fst, d);
    return d;
}
```

Revised Partitioning Algorithm

- Quicksort is $O(n^2)$ when each split gives 1 empty array
- This happens when the array is already sorted
- Solution approach: pick better pivot values
- Use three “marker” elements: first, middle, last
- Let pivot be one whose value is between the others

FIGURE 10.20
Sorting First, Middle, and Last Elements in Array



Summary

- Three quadratic sorting algorithms:
  - Selection sort, bubble sort, insertion sort
- Shell sort: good performance for up to 5000 elements
- Quicksort: average-case O(n log n)
  - If the pivot is picked poorly, get worst case: $O(n^2)$
- Merge sort and heapsort: guaranteed O(n log n)
  - Merge sort: space overhead is O(n)

Summary

TABLE 10.4
Comparison of Sort Algorithms

	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$