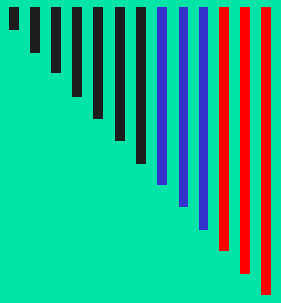


Chapter 12

Sorting

Objectives

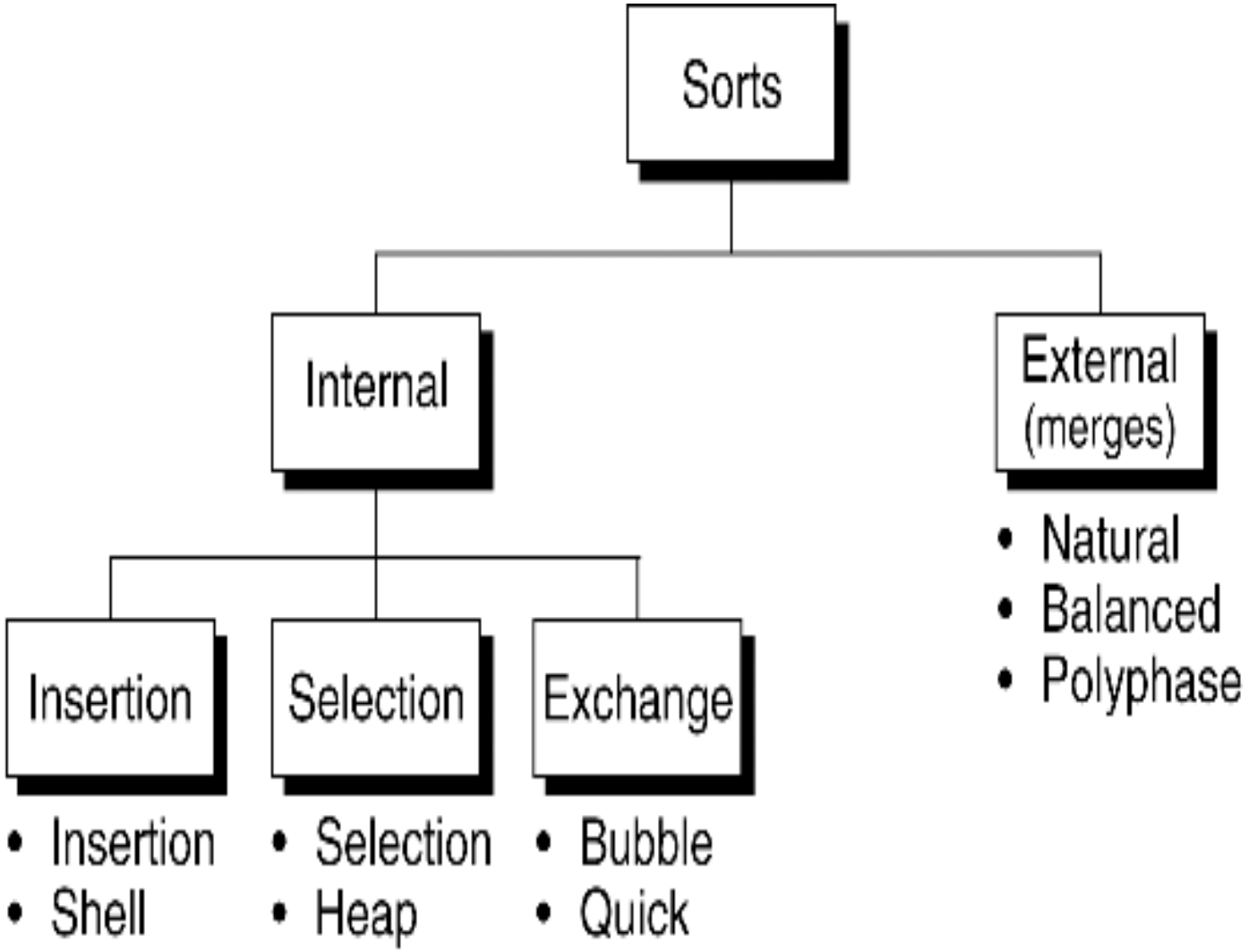
Upon completion you will be able to:

- **Understand the basic concepts of internal and external sorts**
- **Discuss the relative efficiency of different sorts**
- **Recognize and discuss selection, insertion and exchange sorts**
- **Discuss the design and operation of external sorts**
- **Recognize natural, balanced, and polyphase merge sorts**

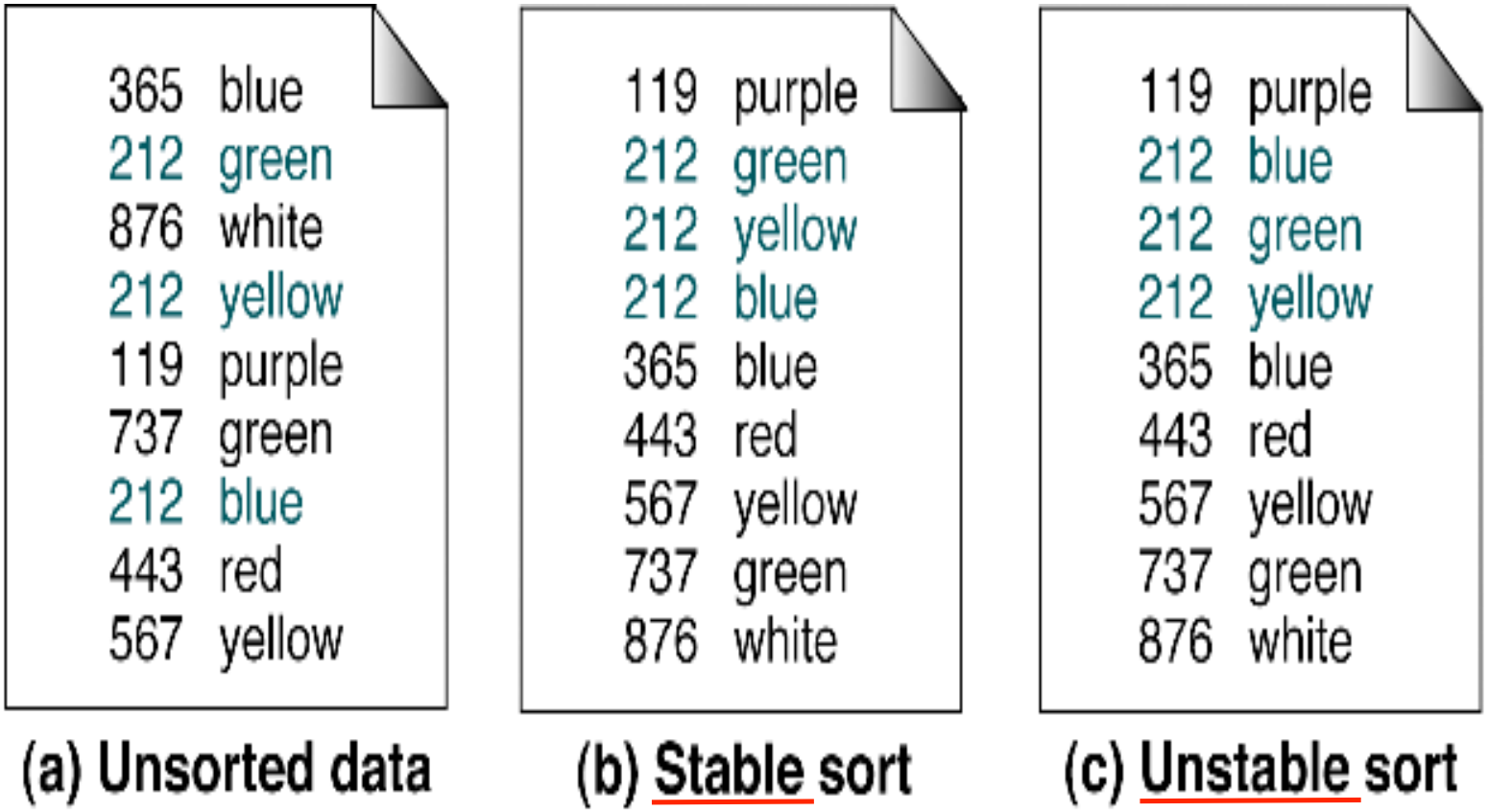
12-1 Sort Concepts

Sorts arrange data according to their value. In this chapter, we discuss both internal and external sorts. We divide internal sorts into three categories-insertion, selection, and exchange-and discuss two different sorts in each classification. At the end of the chapter we discuss natural, balanced, and polyphase external sorts.

Sort Classifications



Sort Stability

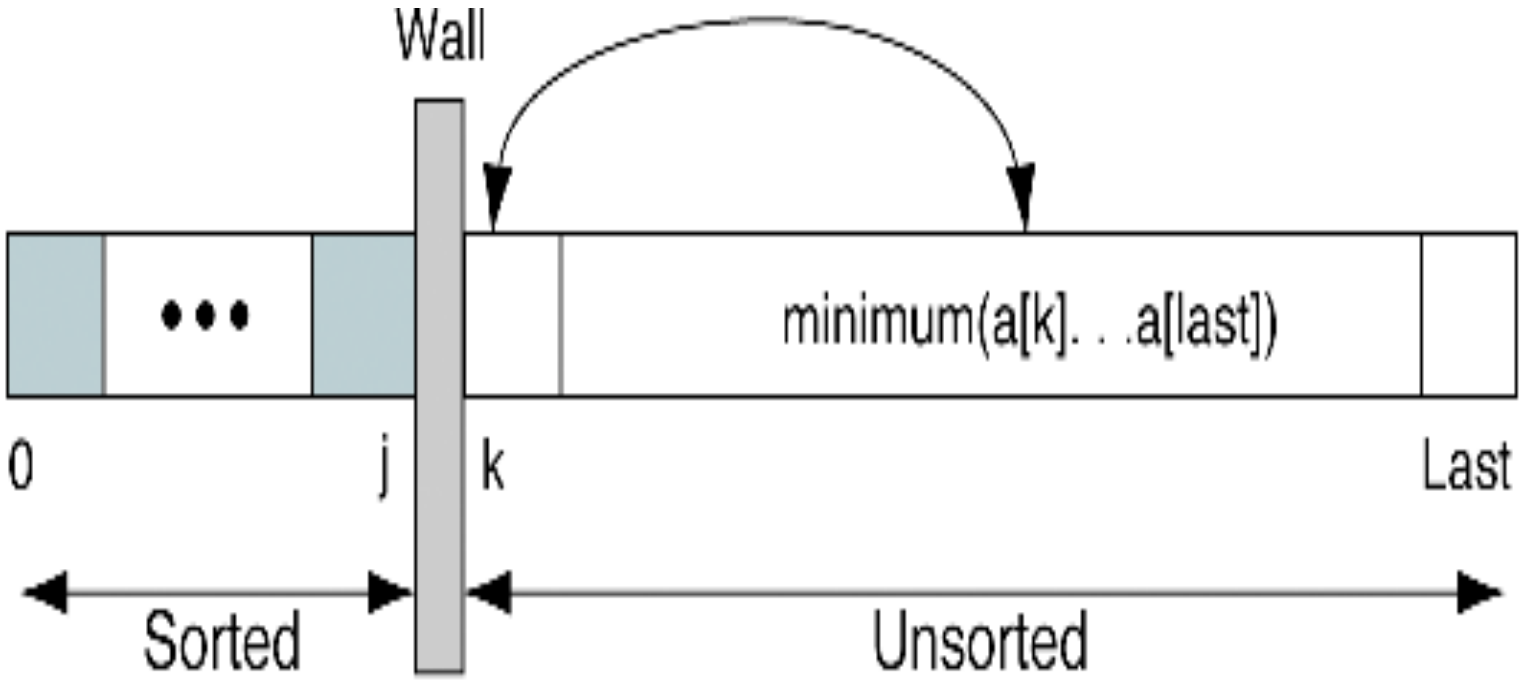


12-2 Selection Sorts

In each pass of the selection sort, the smallest element is selected from the unsorted sublist and exchanged with the element at the beginning of the unsorted list. We discuss two classic selection sorts, straight selection and heap sort.

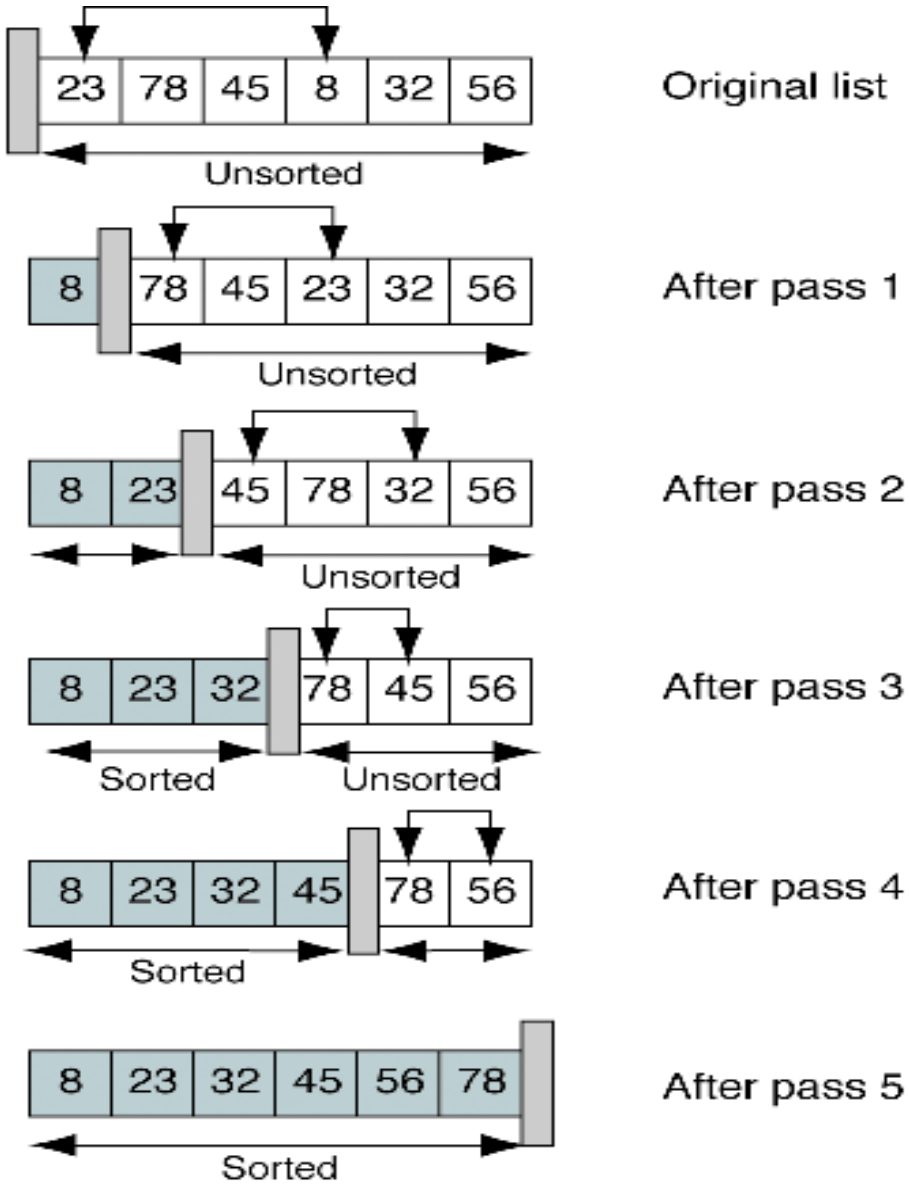
- **Straight Selection Sort**
- **Heap Sort**
- **Selection Sort Efficiency**
- **Selection Sort Implementation**

Selection Sort Concept



Selection Sort Example

一直去Unsorted
那邊挑最小的



Selection Sort

```
Algorithm selectionSort (list, last)
Sorts list array by selecting smallest element in
unsorted portion of array and exchanging it with element
at the beginning of the unsorted list.
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged smallest to largest
1  set current to 0
2  loop (until last element sorted)
   1   set smallest to current
```

Selection Sort (cont.)

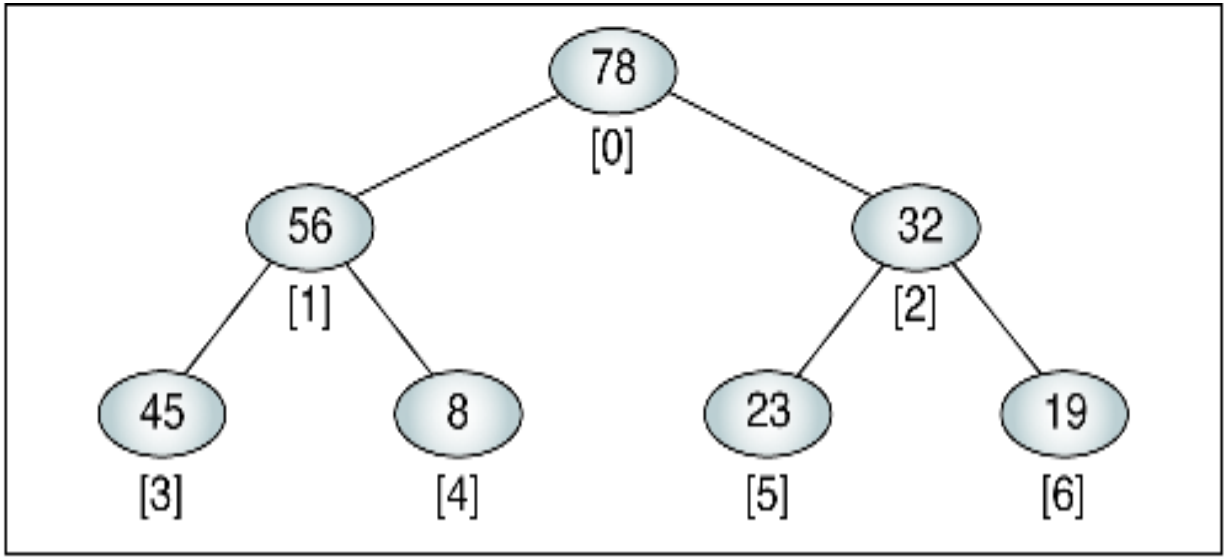
```
   2  set walker   to current + 1
   3  loop (walker <= last)
      1  if (walker key < smallest key)
         1  set smallest to walker
      2  increment walker
   4  end loop
      Smallest selected: exchange with current element.
   5  exchange (current, smallest)
   6  increment current
3  end loop
end selectionSort
```

Selection Sort Efficiency

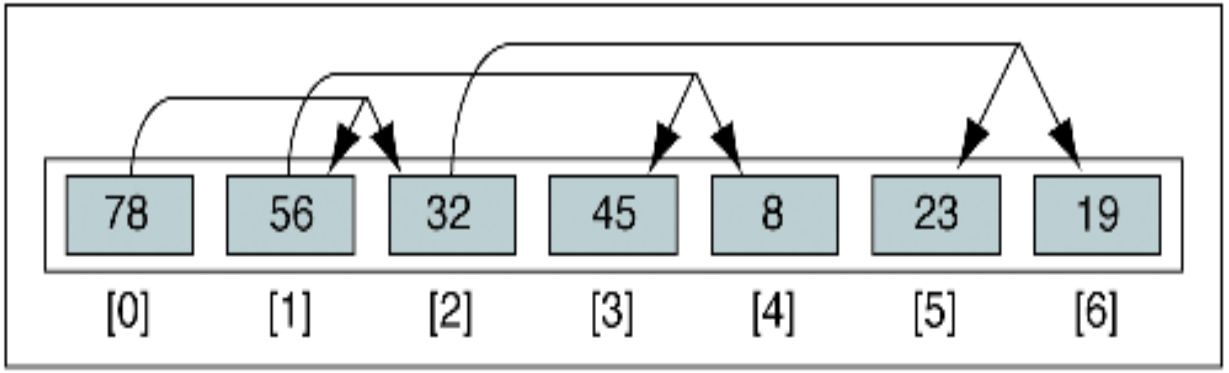
$$O(n^2)$$

```
2 loop (until last element sorted)        <-- n-1
   1  set smallest to current
   2  set walker   to current + 1
   3  loop (walker <= last)                <-- n-1
      1  if (walker key < smallest key)
         1  set smallest to walker
      2  increment walker
   4  end loop
      Smallest selected: exchange with current element.
   5  exchange (current, smallest)
   6  increment current
3 end loop
```

Heap Representations

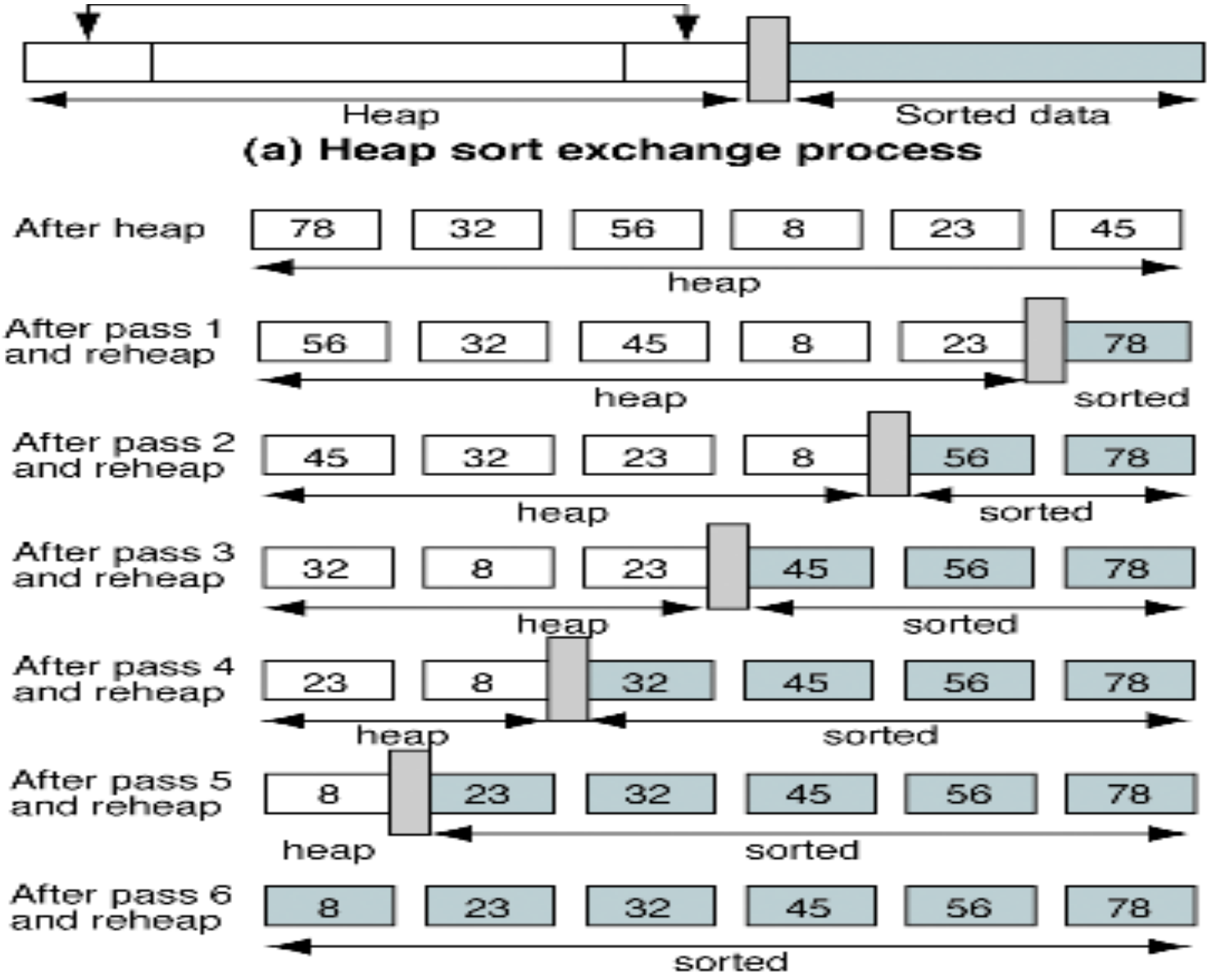


(a) Heap in its tree form



(b) Heap in its array form

Heap Sort


Heap
Sorted data
(a) Heap sort exchange process
After heap
78 32 56 8 23 45
heap
After pass 1 and reheap
56 32 45 8 23 78
heap
sorted
After pass 2 and reheap
45 32 23 8 56 78
heap
sorted
After pass 3 and reheap
32 8 23 45 56 78
heap
sorted
After pass 4 and reheap
23 8 32 45 56 78
heap
sorted
After pass 5 and reheap
8 23 32 45 56 78
heap
sorted
After pass 6 and reheap
8 23 32 45 56 78
sorted

Heap Sort

```
Algorithm heapSort (heap, last)
Sort an array, using a heap.
   Pre  heap array is filled
        last is index to last element in array
   Post heap array has been sorted
   Create heap
1  set walker to 1
2  loop (heap built)
   1   reheapUp (heap, walker)
   2   increment walker
3  end loop
   Heap created. Now sort it.
4  set sorted to last
5  loop (until all data sorted)
   1   exchange (heap, 0, sorted)
   2   decrement sorted
   3   reheapDown (heap, 0, sorted)
6  end loop
end heapSort
```

Heap Sort Efficiency

$$O(n \log n) =$$

$$O(n \log n) + O(n \times \log n)$$

Heap construction

All nodes

Reheap

Computation of Selection Sorts

n	Number of loops	
	Straight selection	Heap
25	625	116
100	10,000	664
500	250,000	4482
1000	1,000,000	9965
2000	4,000,000	10,965
	$O(n^2)$	$O(n \log n)$

Selection Sort

```
 1  /* ================== selectionSort ====================
 2     Sorts list [1…last] by selecting smallest element in
 3     unsorted portion of array and exchanging it with
 4     element at beginning of the unsorted list.
 5        Pre  list must contain at least one item
 6             last contains index to last list element
 7        Post list has been sorted smallest to largest
 8  */
 9  void selectionSort  (int list[ ], int last)
10  {
11  // Local Declarations
12     int smallest;
13     int holdData;
14
15  // Statements
16     for (int current = 0; current < last; current++)
17         {
18          smallest = current;
19          for (int walker = current + 1;
20                   walker <= last;
21                   walker++)
22              if (list[ walker ] < list[ smallest ])
```

Selection Sort (cont.)

```
23                  smallest = walker;
24
25              // Smallest selected: exchange with current
26              holdData         = list[ current ];
27              list[current]    = list[ smallest ];
28              list[smallest]   = holdData;
29            } // for current
30      return;
31  }  // selectionSort
```

Heap Sort

```
 1  /* ===================== heapSort ======================
 2     Sort an array, [list0 .. last], using a heap.
 3        Pre  list must contain at least one item
 4             last contains index to last element in list
 5        Post list has been sorted smallest to largest
 6  */
 7  void heapSort (int  list[ ], int  last)
 8  {
 9  // Local Definitions
10     int sorted;
11     int holdData;
12
13  // Statements
14     // Create Heap
15     for (int walker = 1; walker <= last; walker++)
16        reheapUp (list, walker);
17
18     // Heap created. Now sort it.
19     sorted = last;
20     while (sorted > 0)
21        {
22         holdData     = list[0];
23         list[0]      = list[sorted];
24         list[sorted] = holdData;
```

Heap Sort (cont.)

```
25          sorted--;
26          reheapDown (list, 0, sorted);
27        } // while
28     return;
29  }  // heapSort
30
31  /* ==================== reheapUp ====================
32     Reestablishes heap by moving data in child up to
33     correct location heap array.
34        Pre  heap is array containing an invalid heap
35             newNode is index location to new data in heap
36        Post newNode has been inserted into heap
37  */
```

Heap Sort (cont.)

```
38  void reheapUp (int* heap, int newNode)
39  {
40  // Local Declarations
41     int parent;
42     int hold;
43
44  // Statements
45     // if not at root of heap
46     if (newNode)
47        {
48         parent = (newNode - 1)/ 2;
49         if ( heap[newNode] > heap[parent] )
50            {
51             // child is greater than parent
52             hold           = heap[parent];
53             heap[parent]   = heap[newNode];
54             heap[newNode] = hold;
55             reheapUp (heap, parent);
56            } // if heap[]
57        } // if newNode
58     return;
59  }  // reheapUp
```

Heap Sort (cont.)

```
60
61  /* ==================== reheapDown ====================
62     Reestablishes heap by moving data in root down to its
63     correct location in the heap.
64        Pre  heap is an array of data
65             root is root of heap or subheap
66             last is an index to the last element in heap
67        Post heap has been restored
68  */
69  void reheapDown   (int* heap, int root, int last)
70  {
71  // Local Declarations
72     int hold;
```

Heap Sort (cont.)

```
73      int leftKey;
74      int rightKey;
75      int largeChildKey;
76      int largeChildIndex;
77
78  // Statements
79      if ((root * 2 + 1) <= last)
80          // There is at least one child
81         {
82          leftKey   = heap[root * 2 + 1];
83          if ((root * 2 + 2) <= last)
84             rightKey  = heap[root * 2 + 2];
85          else
86             rightKey  = -1;
87
88          // Determine which child is larger
```

Heap Sort (cont.)

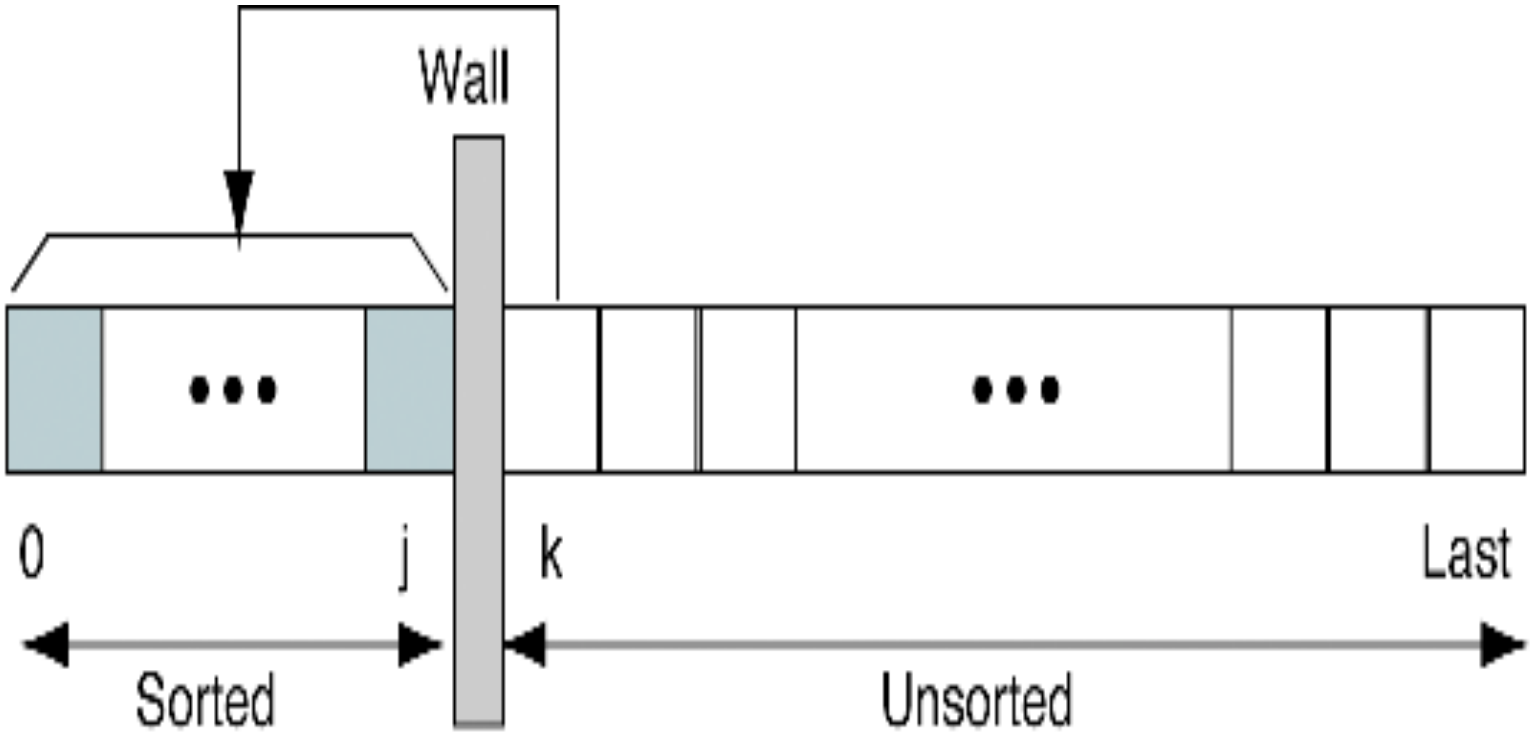
```
 89          if (leftKey > rightKey)
 90              {
 91               largeChildKey   = leftKey;
 92               largeChildIndex = root * 2 + 1;
 93              } // if leftKey
 94          else
 95              {
 96               largeChildKey   = rightKey;
 97               largeChildIndex = root * 2 + 2;
 98              } // else
 99          // Test if root > larger subtree
100          if (heap[root] < heap[largeChildIndex])
101               {
102                // parent < children
103                hold          = heap[root];
104                heap[root] = heap[largeChildIndex];
105                heap[largeChildIndex] = hold;
106                reheapDown (heap, largeChildIndex, last);
107               } // if root <
108      } // if root
109      return;
110  }   // reheapDown
```

12-3 Insertion Sorts

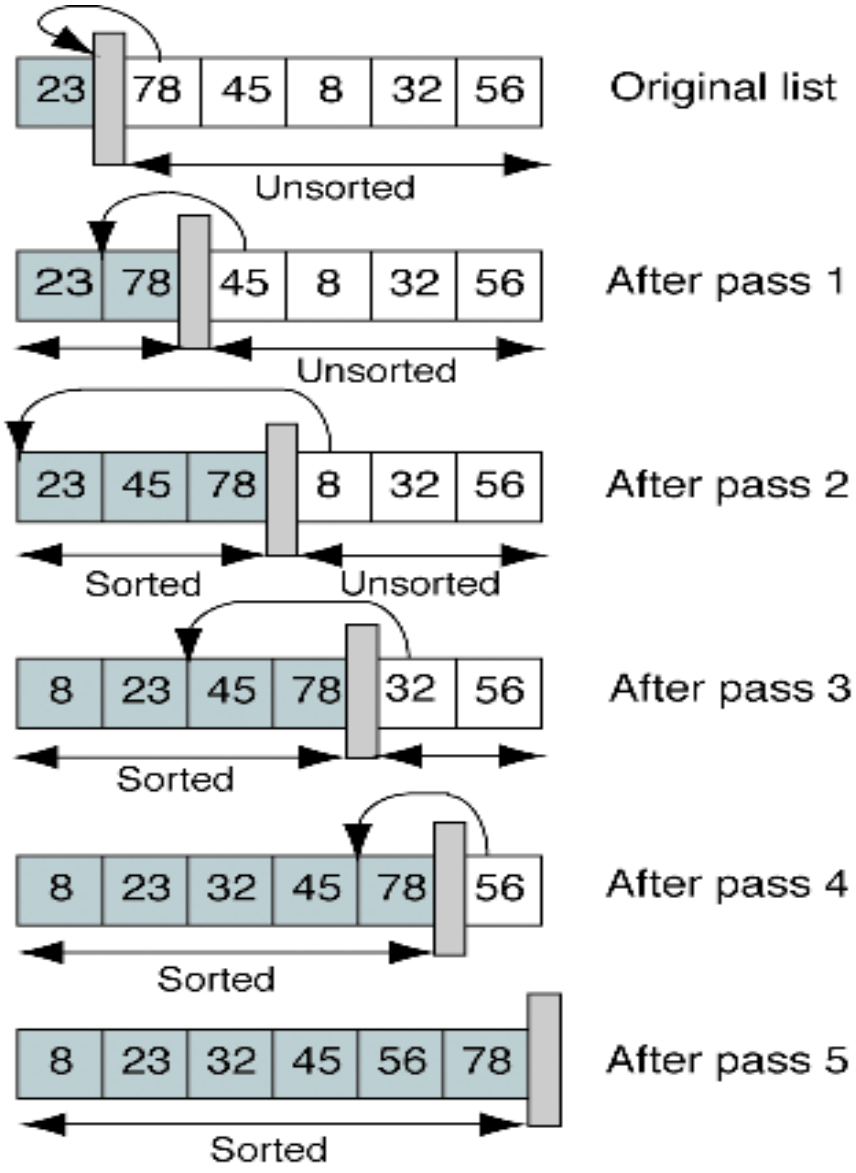
In each pass of an insertion sort, one or more pieces of data are inserted into their correct location in an ordered list. In this section we study two insertion sorts: the straight insertion sort and the shell sort.

- **Straight Insertion Sort**
- **Shell Sort**
- **Insertion Sort Efficiency**
- **Insertion Sort Implementation**

Insertion Sort Concept



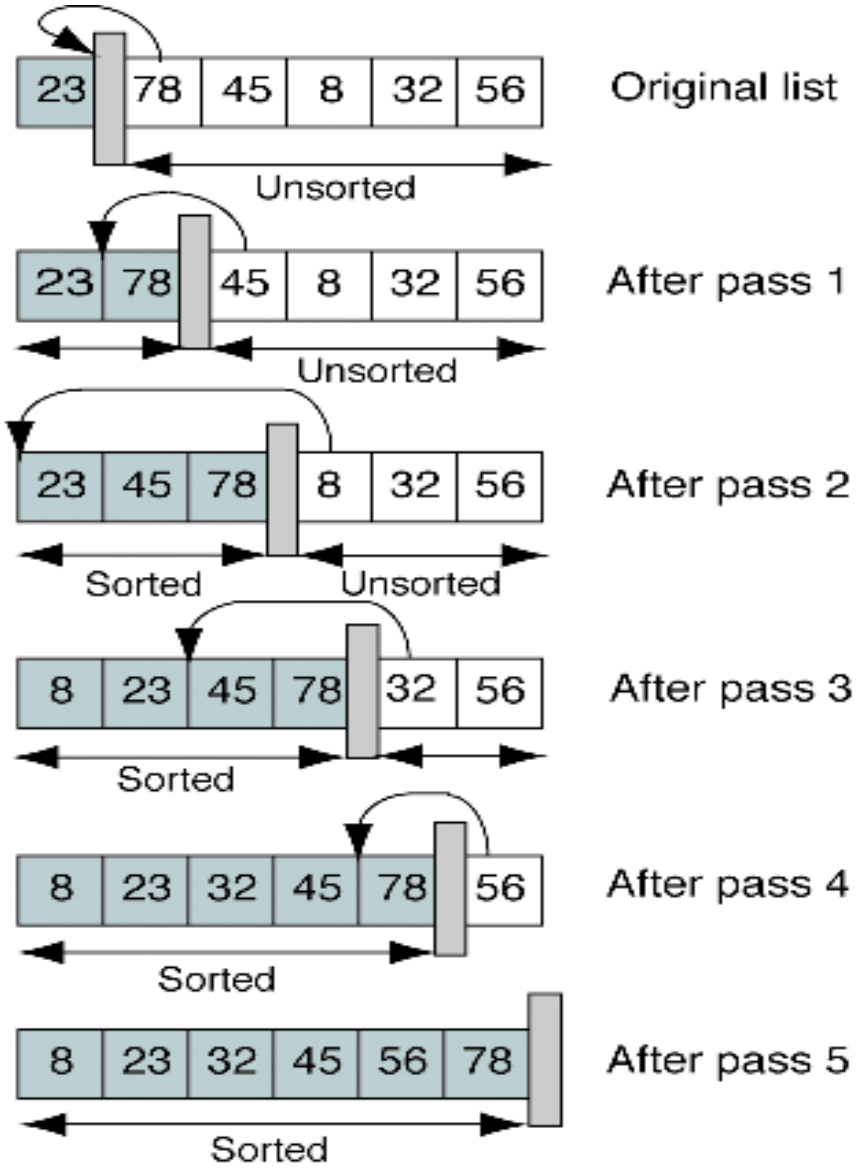
Insertion Sort Example



Straight Insertion Sort

```
Algorithm insertionSort (list, last)
Sort list array using insertion sort. The array is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list.
   Pre  list must contain at least one element
        last is an index to last element in the list
   Post list has been rearranged
1  set current to 1
2  loop (until last element sorted)
   1   move current element to hold
   2   set walker to current - 1
   3   loop (walker >= 0 AND hold key < walker key)
       1   move walker element right one element
       2   decrement walker
   4   end loop
   5   move hold to walker + 1 element
   6   increment current
3  end loop
end insertionSort
```

Insertion Sort Efficiency



n-1 passes

# of Executions:

Pass 1: 1

Pass 2: 2

Pass 3: 3

.

.

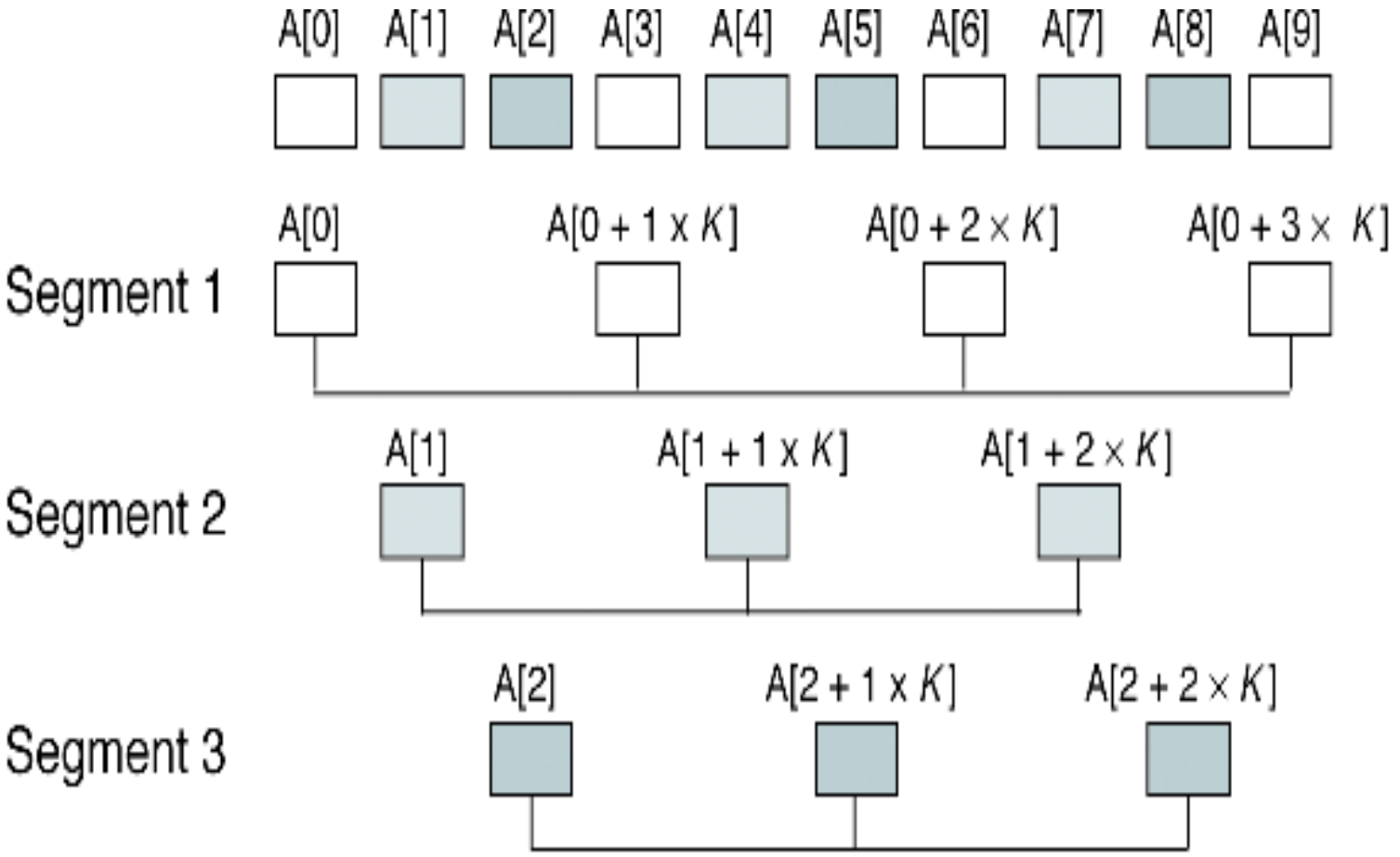
.

Pass n-1: n-1

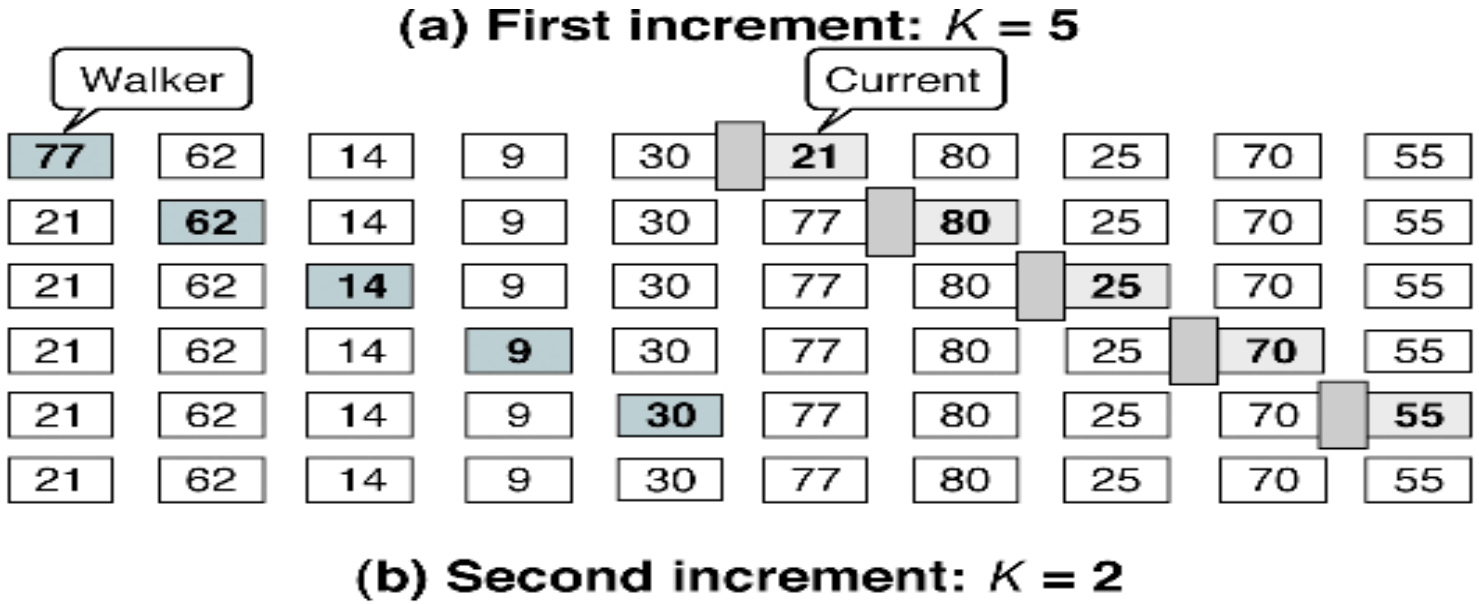
On average, n/2 executions are needed for each pass

$$O(n^2)$$

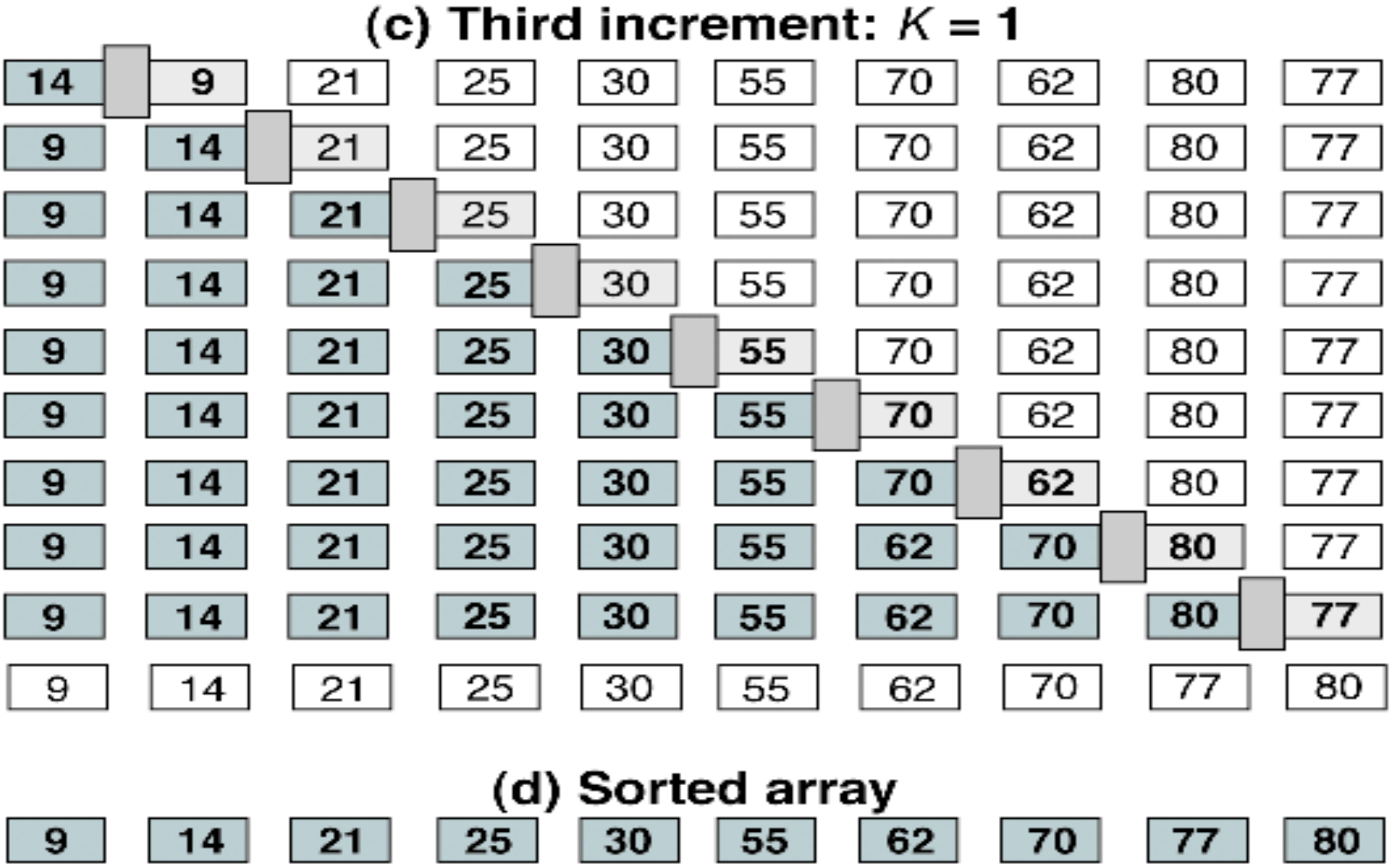
Segmented Array



Diminishing Increments in Shell Sort



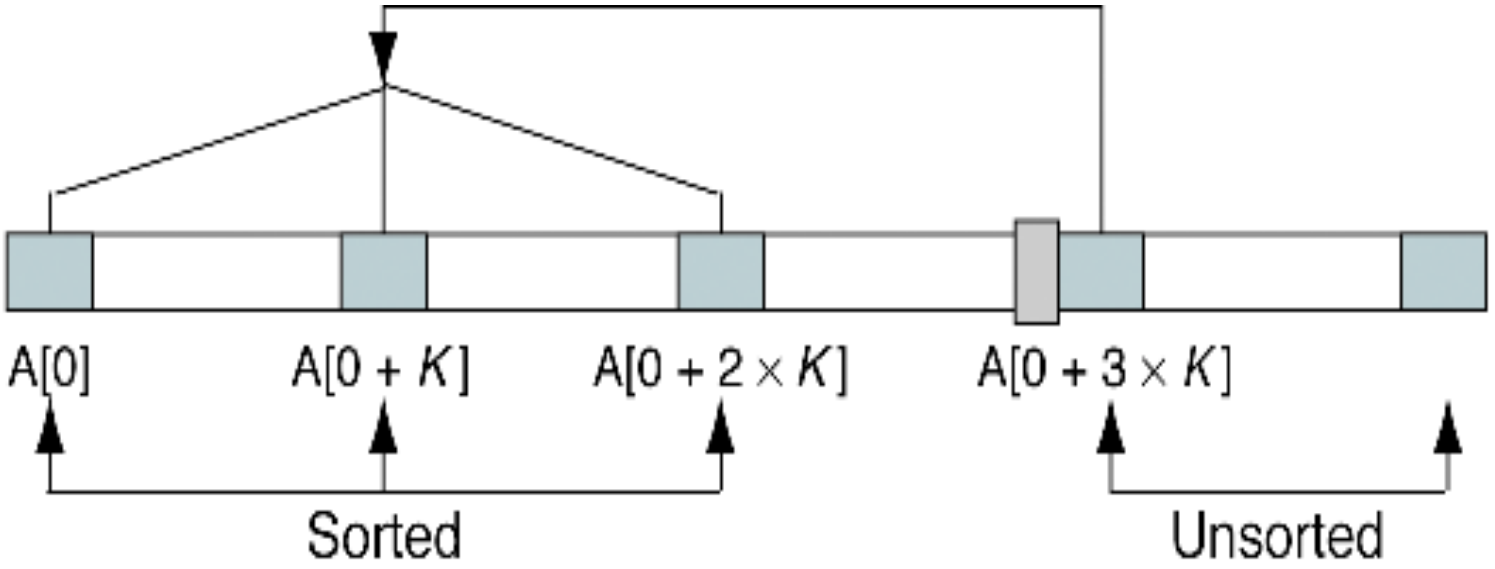
Diminishing Increments in Shell Sort (cont.)



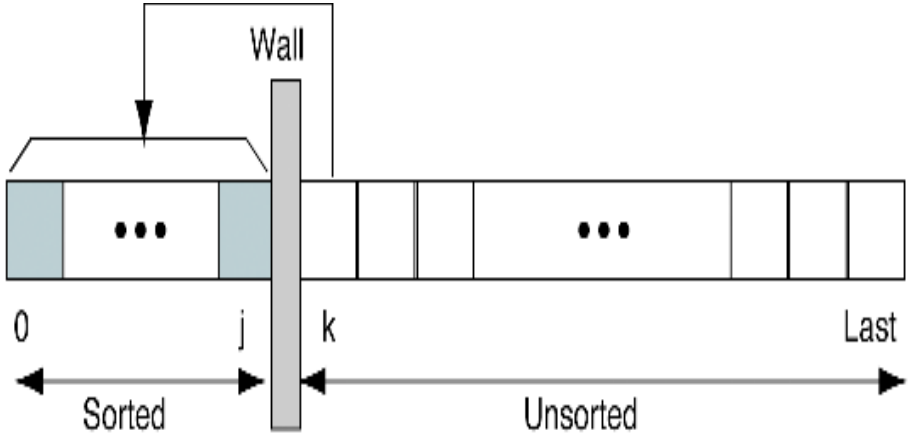
Shell Sort

```
Algorithm shellSort (list, last)
Data in list array are sorted in
place. After the sort, their keys will be in order,
list[0] <= list[1] <= … <= list[last].
   Pre  list is an unordered array of records
        last is index to last record in array
   Post list is ordered on list[i].key
 1 set incre to last / 2
   Compare keys "increment" elements apart.
 2 loop (incre not 0)
   1   set current to incre
   2   loop (until last element sorted)
       1   move current element to hold
       2   set walker to current - incre
       3   loop (walker >= 0 AND hold key < walker key)
              Move larger element up in list.
           1  move walker element one increment right
              Fall back one partition.
           2  set walker to walker - incre
       4   end loop
           Insert hold record in proper relative location.
       5   move hold to walker + incre element
       6   increment current
   3   end loop
       End of pass--calculate next increment.
   4   set incre to incre / 2
 3 end loop
end shellSort
```

Ordered Segment in a Shell Sort

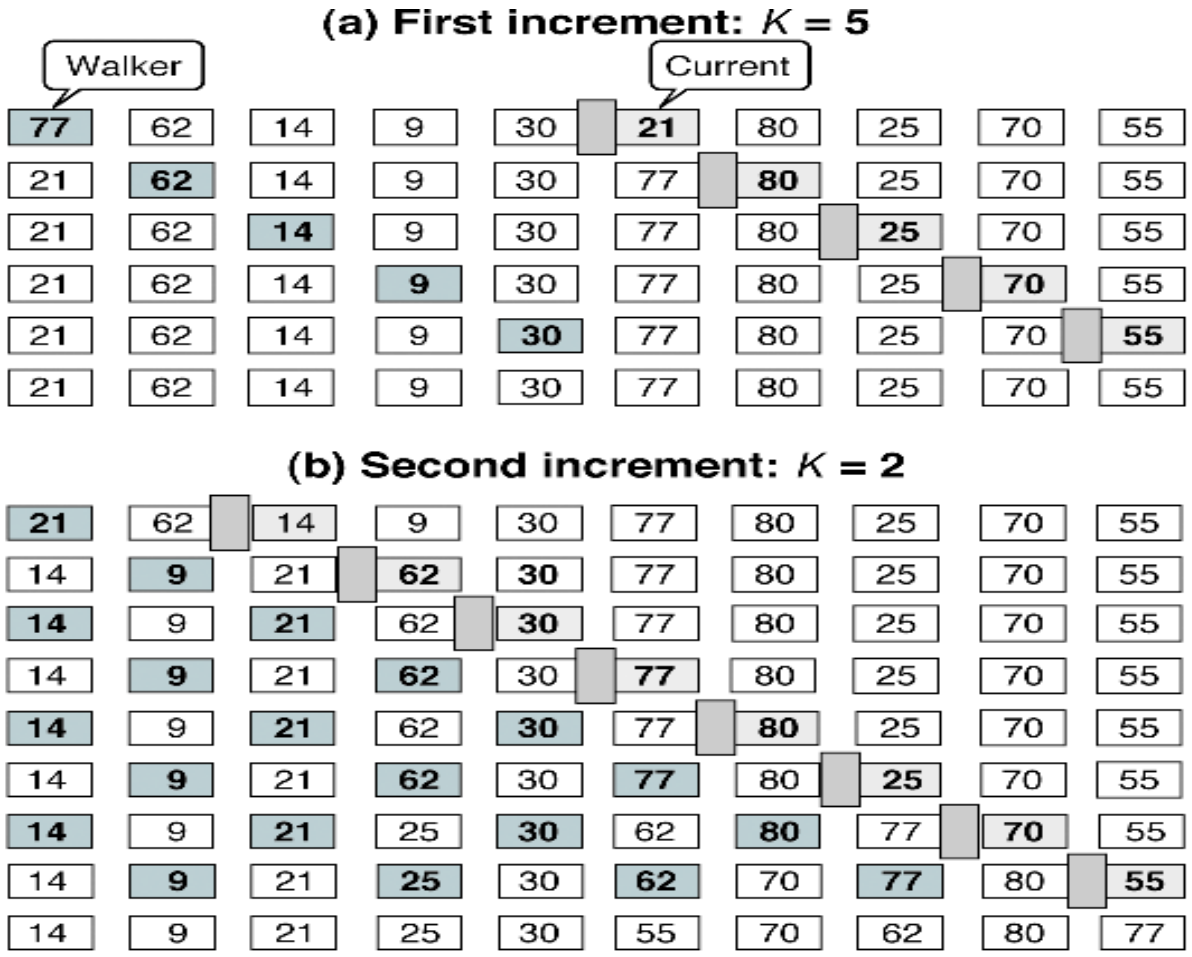


Insertion Sort



Shell Sort Efficiency

More than $O(n \times \log n)$ **,but less than** $O(n^2)$



Comparison of Insertion and Selection Sorts

n	Number of loops		
	Straight insertion Straight selection	Shell	Heap
25	625	55	116
100	10,000	316	664
500	250,000	2364	4482
1000	1,000,000	5623	9965
2000	4,000,000	13,374	10,965
	$O(n^2)$	$O(n^{1.25})$	$O(n \log n)$

Insertion Sort

```
 1  /* ================== insertionSort ==================
 2     Sort using Insertion Sort. The list is divided into
 3     sorted/unsorted list. In each pass, first element
 4     in unsorted list is inserted into sorted list.
 5        Pre  list must contain at least one element
 6             last contains index to last element in list
 7        Post list has been rearranged
 8  */
 9  void insertionSort (int list[], int last)
10  {
11  // Local Definitions
12     int hold;
13     int walker;
14
15  // Statements
16     for (int current = 1; current <= last; current++)
17        {
18         hold = list[current];
19         for (walker = current - 1;
```

Insertion Sort (cont.)

```
20              walker >= 0 && hold < list[walker];
21              walker--)
22                 list[walker + 1] = list[walker];
23
24          list [walker + 1] = hold;
25        } // for current
26     return;
27  }  // insertionSort
```

Shell Sort

```
 1  /* List[1], list[2], …, list[last] are sorted in place
 2     so that the keys are ordered, list[1].key <=
 3     list[2].key, <= … <= list[last].key.
 4        Pre  list is an unordered array of integers
 5             last is index to last element in array
 6        Post list is ordered
 7  */
 8  void shellSort (int list [], int last)
 9  {
10  // Local Definitions
11     int hold;
12     int incre;
13     int walker;
14
15  // Statements
16     incre = last / 2;
17     while (incre != 0)
18        {
19         for (int curr = incre; curr <= last; curr++)
20            {
21             hold = list [curr];
22             walker = curr - incre;
23             while (walker >= 0 && hold < list [walker])
24                {
25                 // Move larger element up in list
26                 list [walker + incre] = list [walker];
27                 // Fall back one partition
```

Shell Sort (cont.)

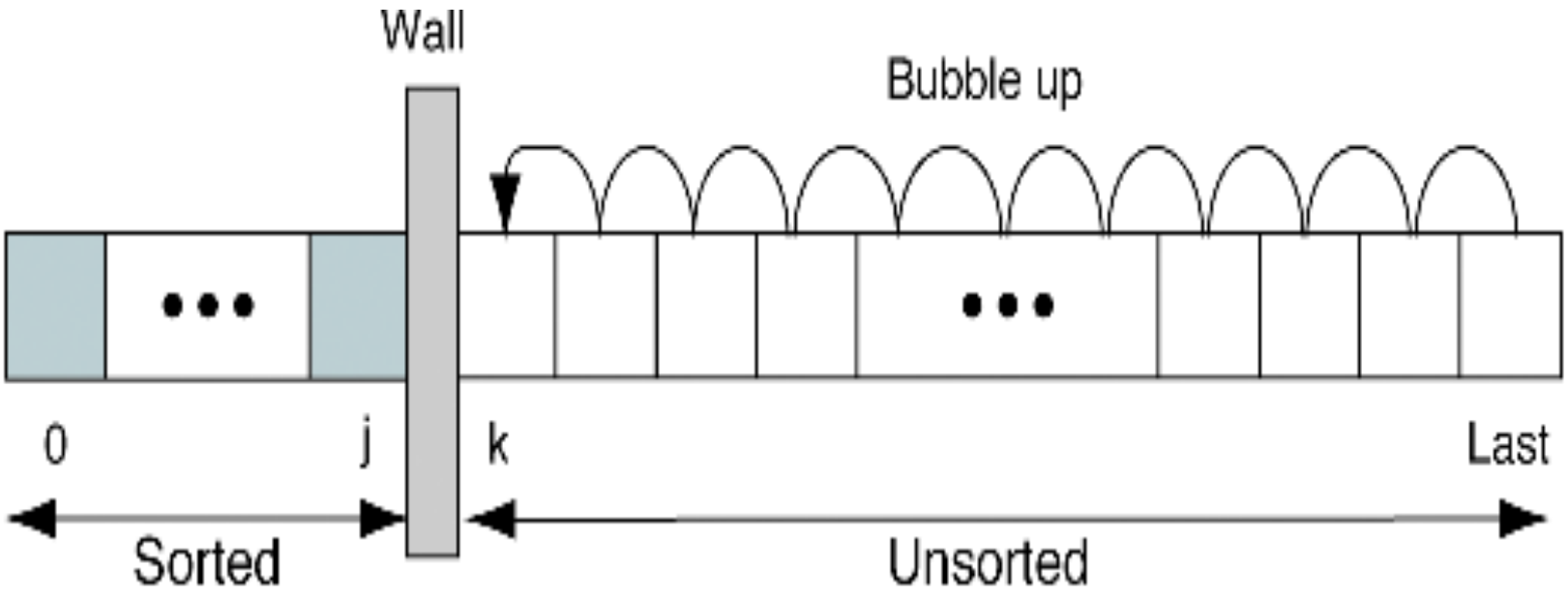
```
28                walker = ( walker - incre );
29               } //  while
30              // Insert hold in proper position
31              list [walker + incre] = hold;
32             } // for walk
33           // End of pass--calculate next increment.
34           incre = incre / 2;
35         } // while
36     return;
37 } // shellSort
```

12-4 Exchange Sorts

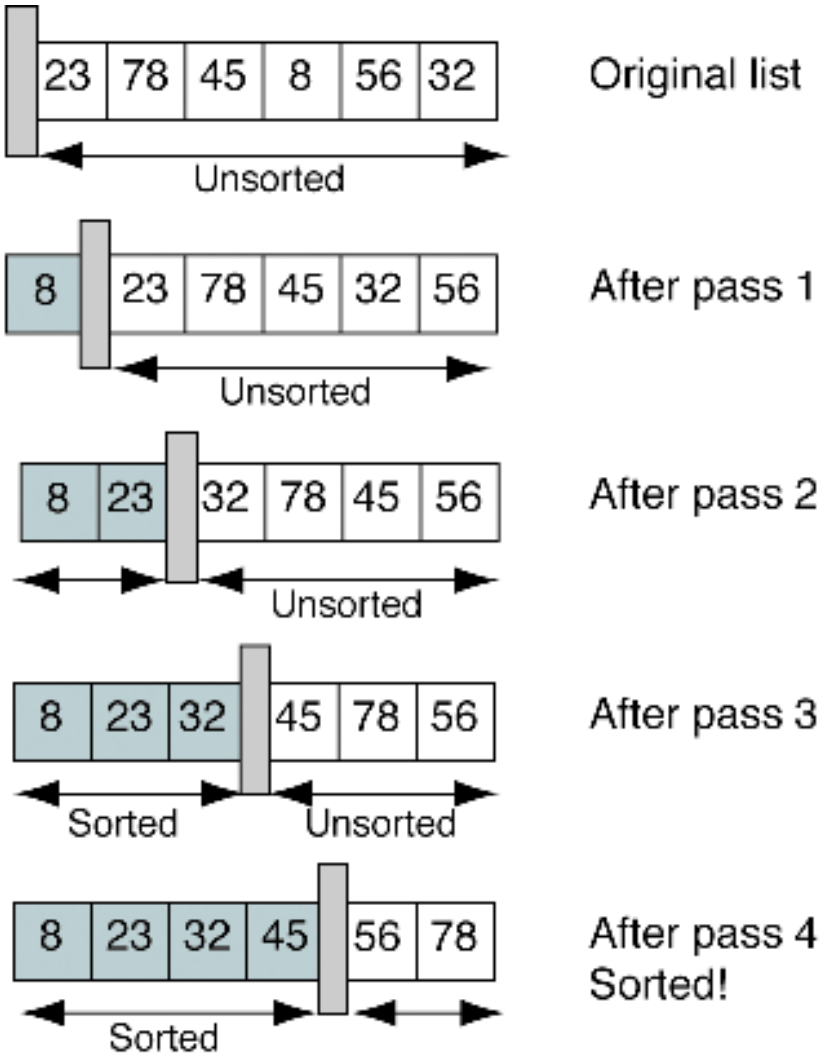
This section discusses the four basic queue operations. Using diagrammatic figures, it shows how each of them work. It concludes with a comprehensive example that demonstrates each operation.

- Bubble Sort
- Quick Sort
- Exchange Sort Efficiency
- Sort Summary
- Exchange Sort Implementation

Bubble Sort Concept



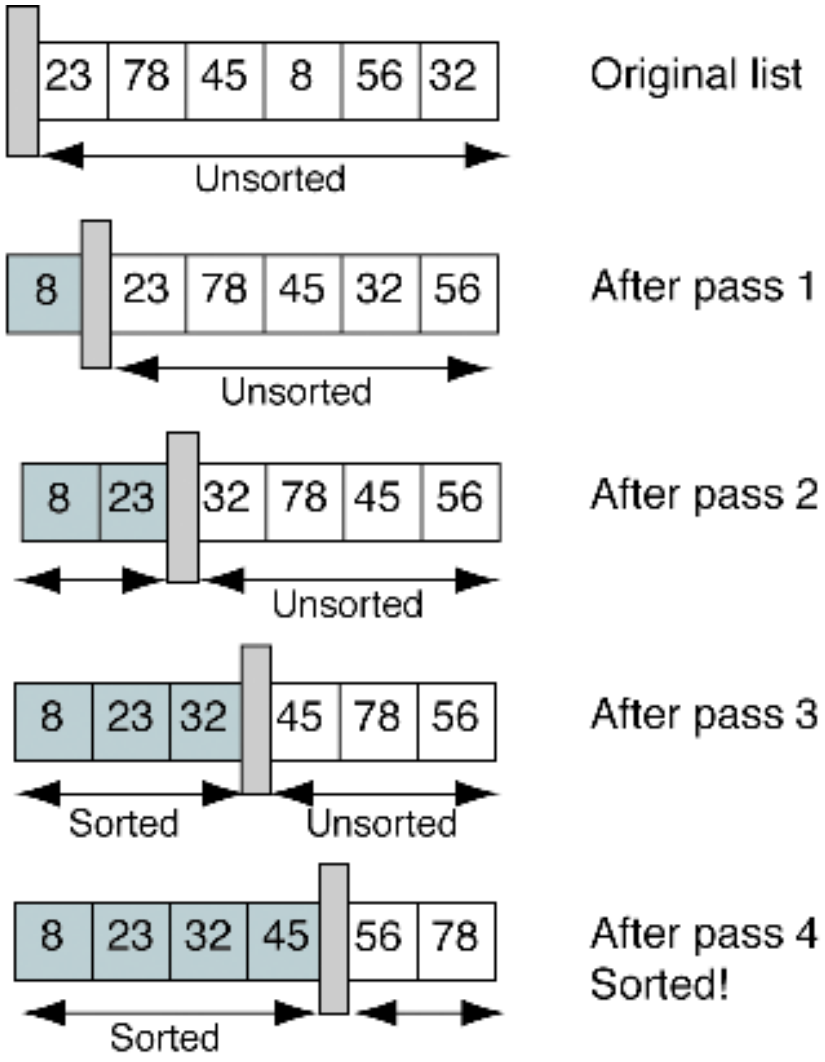
Bubble Sort Example



Bubble Sort

```
Algorithm  bubbleSort (list, last)
Sort an array using bubble sort. Adjacent
elements are compared and exchanged until list is
completely ordered.
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged in sequence low to high
1  set current to 0
2  set sorted  to false
3  loop (current <= last AND sorted false)
      Each iteration is one sort pass.
   1  set walker to last
   2  set sorted to true
   3  loop (walker > current)
      1  if (walker data < walker - 1 data)
            Any exchange means list is not sorted.
         1  set sorted to false
         2  exchange (list, walker, walker - 1)
      2  end if
      3  decrement walker
   4  end loop
   5  increment current
4  end loop
end bubbleSort
```

Bubble Sort Efficiency

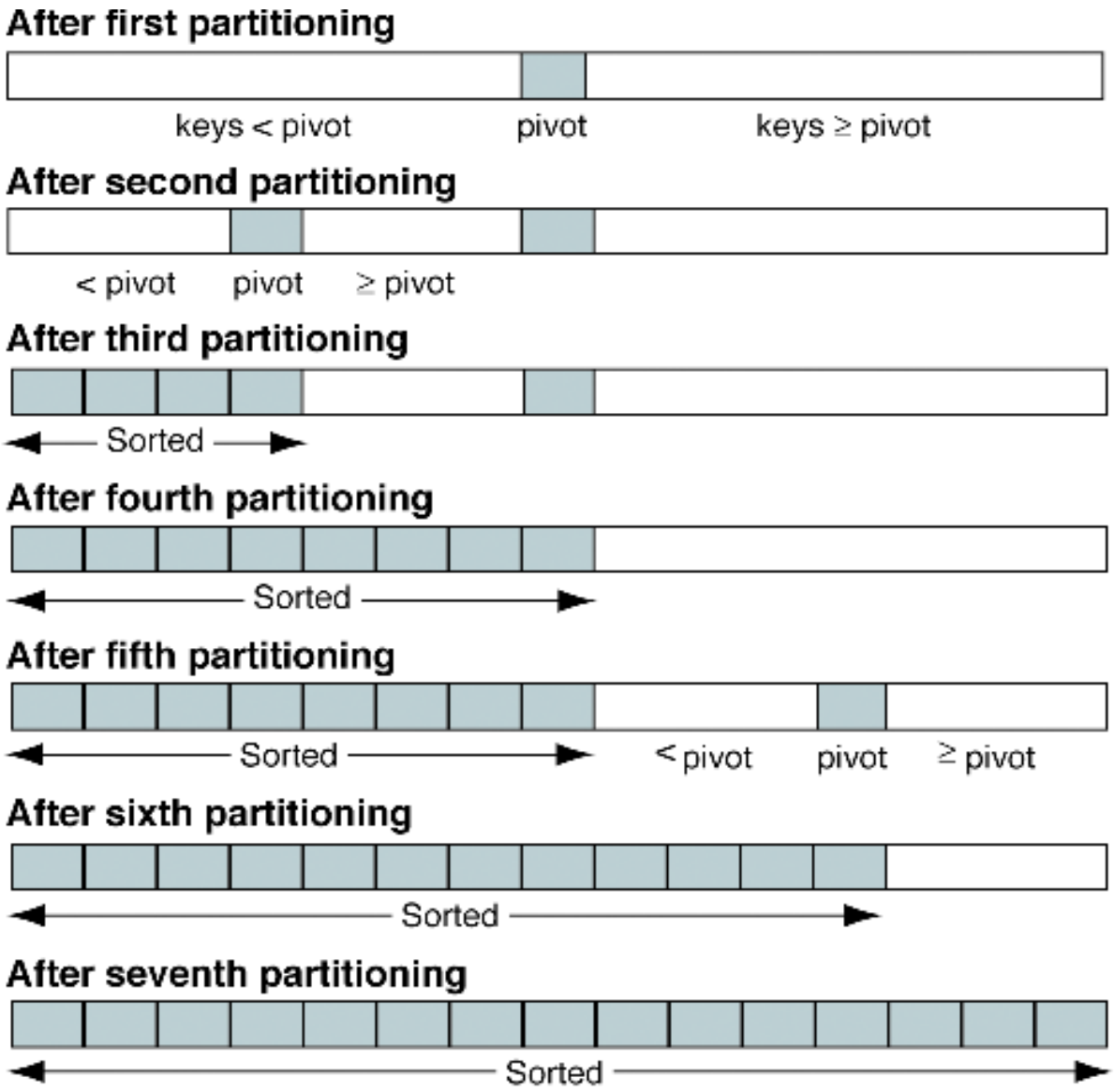


- **n passes**
- **# of executions:**
  pass 1: n-1
  pass 2: n-2
  pass 3: n-3
  ...
  pass n: 0

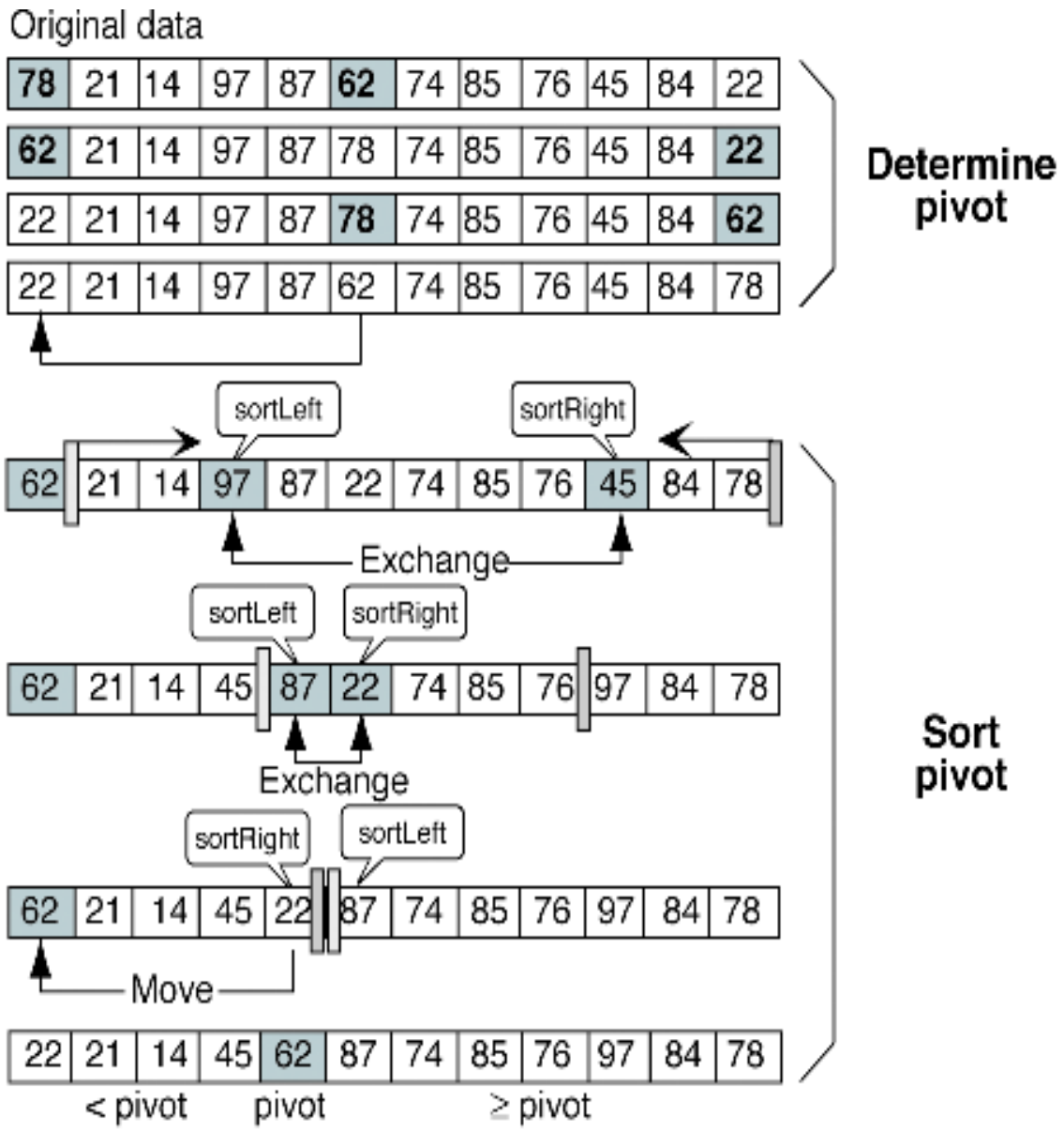
On average, n/2 executions are needed for each pass

$\Rightarrow O(n^2)$

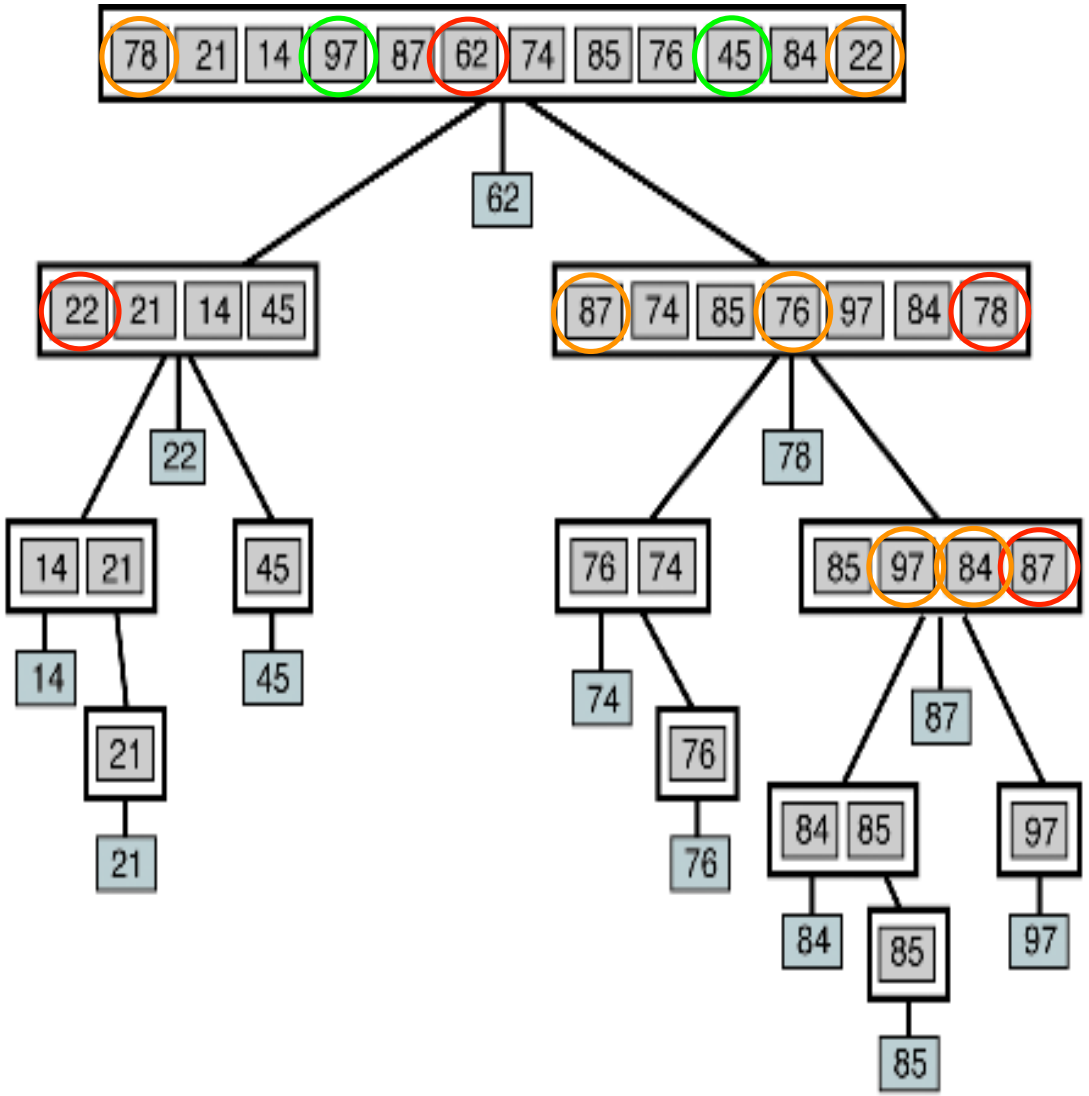
Quick Sort Partitions



Quick Sort Pivot



Quick Sort Operation



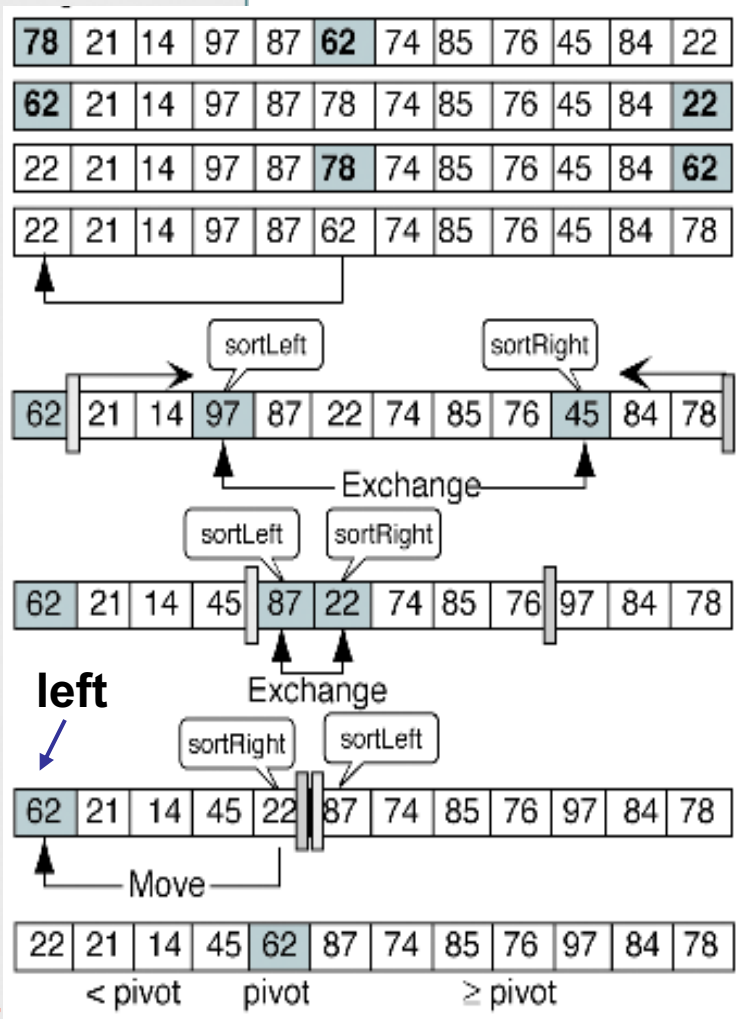
Quick Sort

```
Algorithm quickSort (list, left, right)
An array, list, is sorted using recursion.
   Pre  list is an array of data to be sorted
        left and right identify the first and last
        elements of the list, respectively
   Post list is sorted
1  if ((right - left) > minSize)
      Quick sort
   1  medianLeft (list, left, right)
   2  set pivot     to left element
   3  set sortLeft  to left + 1
   4  set sortRight to right
   5  loop (sortLeft    <= sortRight)
         Find key on left that belongs on right
```

Quick Sort (cont.)

```
      1  loop (sortLeft key < pivot key)
         1  increment sortLeft
      2  end loop
         Find key on right that belongs on left
      3  loop (sortRight key >= pivot key)
         1  decrement sortRight
      4  end loop
      5  if (sortLeft <= sortRight)
         1  exchange(list, sortLeft, sortRight)
         2  increment sortLeft
         3  decrement sortRight
      6  end if
   6  end loop
      Prepare for next pass
   7  move sortLeft - 1 element to left element
   8  move pivot element to sortLeft - 1 element
   9  if  (left < sortRight)
      1  quickSort (list, left, sortRight - 1)
  10  end if
  11  if (sortLeft < right)
      1  quickSort (list, sortLeft, right)
  12  end if
 2  else
    1  insertionSort (list, left, right)
 3  end if
end quickSort
```

partition夠小的時候，用簡單的方法做完就好



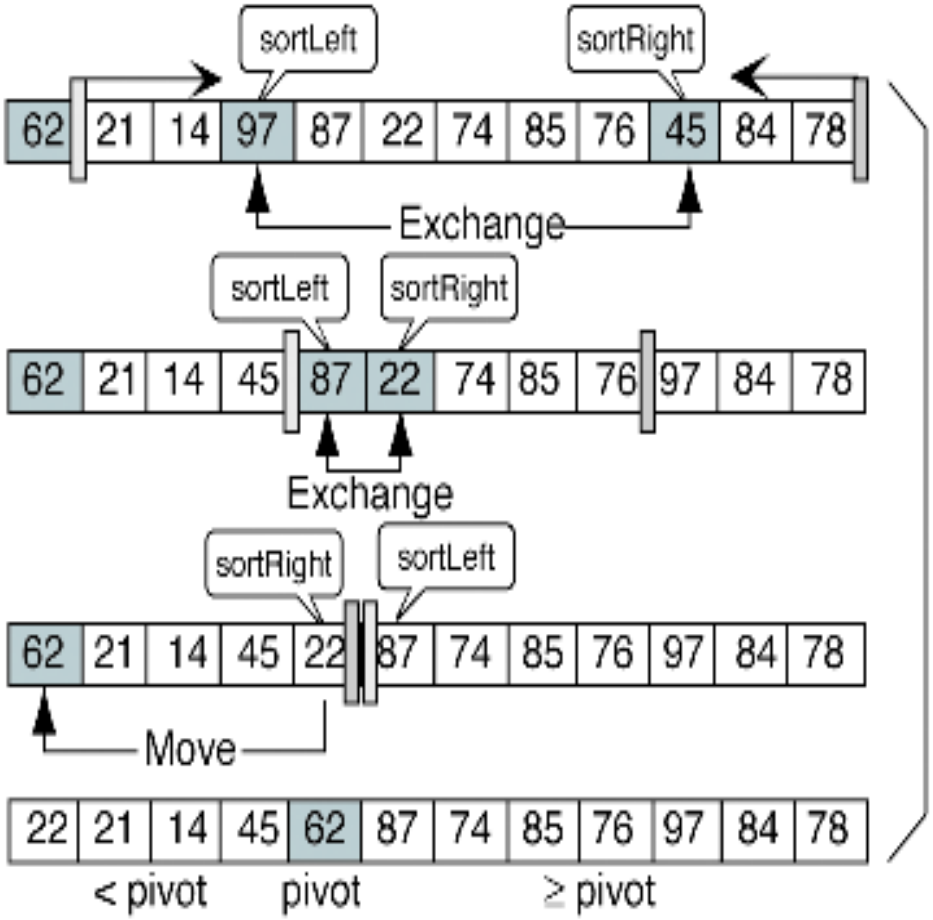
Quick Sort's Straight Insertion Sort

```
Algorithm quickInsertion (list, first, last)
Sort array list using insertion sort. The list is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list. This is a special version of the insertion
sort modified for use with quick sort.
   Pre  list must contain at least one element
        first is an index to first element in the list
        last is an index to last element in the list
   Post list has been rearranged
 1 set current to first + 1
 2 loop (until current partition sorted)
   1  move current element to hold
   2  set walker to current - 1
   3  loop (walker >= first AND hold key < walker key)
      1  move walker element one element right
      2  decrement walker
   4  end loop
   5  move hold to walker + 1 element
   6  increment current
 3 end loop
end quickInsertion
```

Median Left

```
Algorithm medianLeft (sortData, left, right)
Find the median value of an array and place it in the first
(left) location.
   Pre   sortData is an array of at least three elements
         left and right are the boundaries of the array
   Post  median value located and placed left
   Rearrange sortData so median value is in middle location.
1  set mid to (left + right) / 2
2  if (left key > mid key)
   1   exchange (sortData, left, mid)
3  end if
4  if (left key > right key)
   1   exchange (sortData, left, right)
5  end if
6  if (mid key > right key)
   1   exchange (sortData, mid, right)
7  end if
   Median is in middle location. Exchange with left.
8  exchange (sortData, left, mid)
end medianLeft
```

Quick Sort Efficiency (Rough)



1. Each pass requires *n* steps

2. *Log(n)* passes are required

⇩

O(nLog(n))

Quick Sort Efficiency – Worst Case

***f(n)*: number of comparisons to sort an n-element-array**

***i*: the chosen pivot index**

$$\Rightarrow \begin{cases} f(n) = 0 & \text{if } n < 2 \\ f(n) = f(i) + f(n-i-1) + n & \text{if } n \geq 2 \end{cases}$$

Worst Case : no Singleton's mod. and the data were already sorted

前面的medium-left就是在避免這件事

$$f(n) = f(i) + f(n-i-1) + n$$

$$\Rightarrow f(n) = f(n-1) + n \quad \because i = 0 \text{ and } f(i) = f(0) = 0$$

$$\Rightarrow \begin{cases} f(n) = f(n-1) + n \\ f(n-1) = f(n-2) + n-1 \\ f(n-2) = f(n-3) + n-2 \\ \cdots \\ f(1) = f(0) + 1 \\ f(0) = 0 \end{cases}$$

$$\Rightarrow f(n) = n+(n-1)+(n-2)+\ldots+1+0$$

$$\Rightarrow f(n) = n(n+1)/2$$

$$\Rightarrow O(n^2)$$

Quick Sort Efficiency – Best Case

Best Case : the pivot belongs exactly in the middle of the array

$\because n = 2^k - 1$ and $i = 2^{k-1} - 1$

f(n) = f(i) + f(n-i-1) + n

$$\Rightarrow f(2^k - 1) = f(2^{k-1} - 1) + f(2^k - 1 - (2^{k-1} - 1) - 1) + 2^k - 1$$

$$\Rightarrow f(2^k - 1) = 2f(2^{k-1} - 1) + 2^k - 1$$

$$\Rightarrow \begin{cases} f(2^k - 1) = 2^1 f(2^{k-1} - 1) + 2^0(2^k - 1) \\ 2^1 f(2^{k-1} - 1) = 2^2 f(2^{k-2} - 1) + 2^1(2^{k-1} - 1) \\ 2^2 f(2^{k-2} - 1) = 2^3 f(2^{k-3} - 1) + 2^2(2^{k-2} - 1) \\ \ldots \\ 2^{k-2} f(2^{2-1} - 1) = 2^{k-1} f(2^1 - 1) + 2^{k-1}(2^1 - 1) \end{cases}$$

$$\Rightarrow f(2^k - 1) = 2^{k-1} f(1) + 2^0(2^k - 1) + 2^1(2^{k-1} - 1) + \ldots + 2^{k-1}(2^1 - 1)$$

Quick Sort Efficiency – Best Case

$\because f(1)=0$ $\quad f(2^k-1)=(2^k-1)+(2^k-2^1)+\ldots+(2^k-2^{k-1})$

$=(k-1)2^k-(2^0+2^1+2^2+\ldots+2^{k-1})$

$=(k-1)2^k-(2^k-1)$

Let $n=2^k$-1 *or* $k=\log(n+1)$

$f(n)=(\log(n+1)-1)(n+1)-n$

$=(n+1)\log(n+1)-2n-1$

$\Rightarrow O(n\log n)$

Quick Sort Efficiency – Average Case

Average Case : the pivot index *i* can be anywhere

$f(n) = f(i) + f(n-i-1) + n$

$\Rightarrow f(n) = (1/n)\{f(0) + f(1) + \ldots + f(n-1)\}$
$+ (1/n)\{f(n-1) + f(n-2) + \ldots + f(0)\} + n$

$\Rightarrow f(n) = (2/n)\{f(0) + f(1) + \ldots + f(n-1)\} + n$

$\Rightarrow nf(n) = 2\{f(0) + f(1) + \ldots + f(n-1)\} + n^2$ .....(a)

$\Rightarrow (n-1)f(n-1) = 2\{f(0) + f(1) + \ldots + f(n-2)\} + (n-1)^2$ ....(b)

(a)-(b)

$\Rightarrow nf(n) - (n-1)f(n-1) = 2f(n-1) + n^2 - (n-1)^2$

$\Rightarrow nf(n) = (n+1)f(n-1)+2n-1$

Divide by $n(n+1)$ $\Rightarrow$ $\frac{1}{n+1}f(n) = \frac{1}{n}f(n-1) + \frac{2}{n+1} - \frac{1}{n(n+1)}$

Quick Sort Efficiency – Average Case

$$\Rightarrow \begin{cases} \frac{1}{n+1}f(n) = \frac{1}{n}f(n-1) + \frac{2}{n+1} - \frac{1}{n(n+1)} \\ \frac{1}{n}f(n-1) = \frac{1}{n-1}f(n-2) + \frac{2}{n} - \frac{1}{(n-1)n} \\ \frac{1}{n-1}f(n-2) = \frac{1}{n-2}f(n-3) + \frac{2}{n-1} - \frac{1}{(n-2)(n-1)} \\ \ldots \\ \frac{1}{3}f(2) = \frac{1}{2}f(1) + \frac{2}{3} - \frac{1}{(2)(3)} \end{cases}$$

$\Rightarrow$ ***(1/(n+1))f(n) = (1/2)f(1)+***
(2/(n+1)+2/n-1)+…+2/3-
{(1/n(n+1)+1/(n-2)(n)+1/(n-2)(n-1)+…+1/6)}

Quick Sort Efficiency – Average Case

$f(1)=0$

$\Rightarrow (1/(n+1))f(n) = (2/(n+1)+2/n-1)+\ldots+2/3)-$
$\{(1/n(n+1)+1/(n-2)(n)+1/(n-2)(n-1)+\ldots+1/6)\}$

$\Rightarrow f(n) = (n+1)(2/(n+1)+2/n-1)+\ldots+2/3)-$
~~$\{(n+1)(1/n(n+1)+1/(n-2)(n)+1/(n-2)(n-1)+\ldots+1/6)\}$~~

$\Rightarrow f(n) = (n+1)\ln n$

$$\Rightarrow O(n \log n)$$

Sort Comparisons

n	Number of loops		
	Straight insertion Straight selection Bubble sorts	Shell	Heap and quick
25	625	55	116
100	10,000	316	664
500	250,000	2364	4482
1000	1,000,000	5623	9965
2000	4,000,000	13,374	10,965
	O(n^2)	O(n^1.25)	O(nlogn)

Bubble Sort

```
 1  /* ================== bubbleSort ====================
 2     Sort list using bubble sort. Adjacent elements are
 3     compared and exchanged until list is ordered.
 4        Pre  list must contain at least one item
 5             last contains index to last list element
 6        Post list has been sorted in ascending sequence
 7  */
 8  void bubbleSort (int list [], int last)
 9  {
10  // Local Definitions
11     int temp;
12
```

Bubble Sort (cont.)

```
13  // Statements
14     // Each iteration is one sort pass
15     for (int current = 0, sorted = 0;
16               current <= last && !sorted;
17               current++)
18          for (int walker = last, sorted = 1;
19                    walker > current;
20                    walker--)
21            if (list[ walker ]  < list[ walker - 1 ])
22                 // Any exchange means list is not sorted
23                 {
24                  sorted = 0;
25                  temp               = list[walker];
26                  list[walker]       = list[walker - 1];
27                  list[walker - 1] = temp;
28                 } // if
29     return;
30  }  // bubbleSort
```

Quick Sort

```
 1  /* ==================== quickSort ====================
 2     Array, sortData[left..right] sorted using recursion.
 3        Pre  sortData is an array of data to be sorted
 4             left identifies first element of sortData
 5             right identifies last element of sortData
 6        Post sortData array is sorted
 7  */
 8  void quickSort (int sortData[ ], int left, int right)
 9  {
10  #define MIN_SIZE 16
11
12  // Local Definitions
13  int sortLeft;
14  int sortRight;
15  int pivot;
16  int hold;
17
```

Quick Sort (cont.)

```
18  // Statements
19     if ((right - left) > MIN_SIZE)
20       {
21        medianLeft (sortData, left, right);
22        pivot     = sortData [left];
23        sortLeft  = left + 1;
24        sortRight = right;
25        while (sortLeft <= sortRight)
26          {
27           // Find key on left that belongs on right
28           while (sortData [sortLeft] < pivot)
29              sortLeft = sortLeft + 1;
30           // Find key on right that belongs on left
31           while (sortData[sortRight] >= pivot)
32              sortRight = sortRight - 1;
```

Quick Sort (cont.)

```
33            if (sortLeft <= sortRight)
34              {
35               hold                  = sortData[sortLeft];
36               sortData[sortLeft]    = sortData[sortRight];
37               sortData[sortRight]   = hold;
38               sortLeft              = sortLeft  + 1;
39               sortRight             = sortRight - 1;
40              } // if
41          } // while
42        // Prepare for next pass
43        sortData [left]          = sortData [sortLeft - 1];
44        sortData [sortLeft - 1] = pivot;
45        if (left < sortRight)
46            quickSort (sortData, left, sortRight - 1);
47        if (sortLeft < right)
48            quickSort (sortData, sortLeft, right);
49      } // if right
50    else
51      quickInsertion (sortData, left, right);
52     return;
53  } // quickSort
```

Modified Insertion Sort

```
 9              last is index to last element in data
10         Post data has been rearranged
11  */
12  void quickInsertion (int data[], int first, int last)
13  {
14  // Local Definitions
15     int hold;
16     int walker;
17
18  // Statements
19     for (int current = first + 1;
20              current <= last;
21              current++)
22        {
23         hold   = data[current];
24         walker = current - 1;
25         while (walker >= first
26             && hold < data[walker])
27               {
28                data[walker + 1] = data[walker];
29                walker = walker - 1;
30               } // while
31         data[walker + 1] = hold;
32        } // for
33     return;
34  }  // quickInsertion
```

Medium Left for Quick Sort

```
 1  /* ================== medianLeft ====================
 2     Find the median value of an array,
 3     sortData[left..right], and place it in the
 4     location sortData[left].
 5        Pre  sortData is array of at least three elements
 6             left and right are boundaries of array
 7        Post median value placed at sortData[left]
 8  */
 9  void medianLeft (int sortData[], int left, int right)
10  {
11  // Local Definitions
12     int mid;
13     int hold;
14
15  // Statements
16     // Rearrange sortData so median is middle location
17     mid = (left + right) / 2;
```

Medium Left for Quick Sort (cont.)

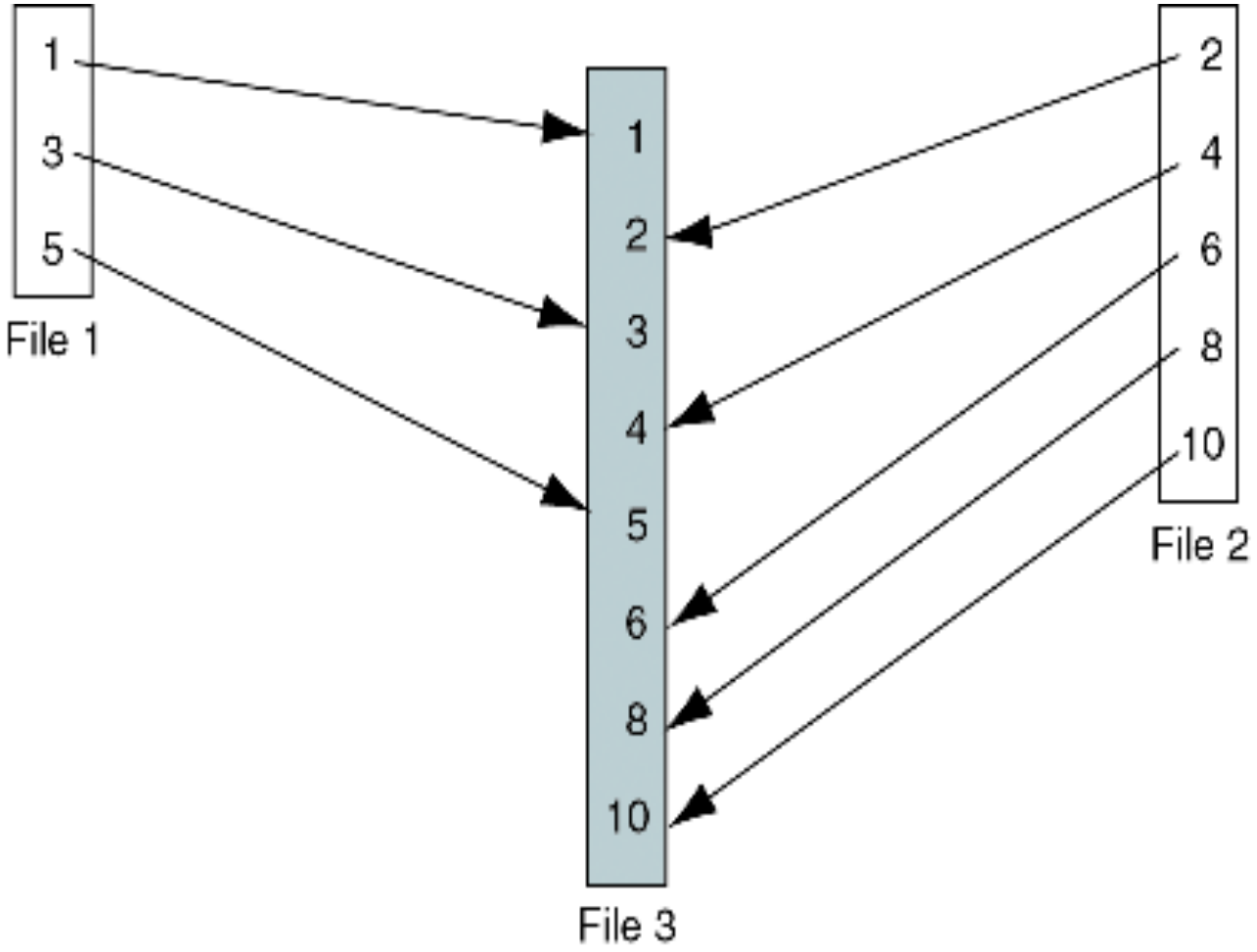
```
18      if (sortData[left] > sortData[mid])
19         {
20          hold             = sortData[left];
21          sortData[left]   = sortData[mid];
22          sortData[mid]    = hold;
23         } // if
24      if (sortData[left] > sortData[right])
25         {
26          hold             = sortData[left];
27          sortData[left]   = sortData[right];
28          sortData[right] = hold;
29         } // if
30      if (sortData[mid]  > sortData[right])
31         {
32          hold             = sortData[mid];
33          sortData[mid]    = sortData[right];
34          sortData[right] = hold;
35         } // if
36
37      // Median is in middle. Exchange with left
38      hold             = sortData[left];
39      sortData[left] = sortData[mid];
40      sortData[mid]  = hold;
41
42      return;
43  }  // medianLeft
```

12-5 External Sorts

In external sorting portions of the data may be stored in secondary memory during the sorting process. Included in this section is a discussion of file merging and three external sort approaches-natural, balanced, and polyphase.

- **Merging Ordered Files**
- **Merging Unordered Files**
- **The Sorting Process**
- **Sort Phase Revisited**

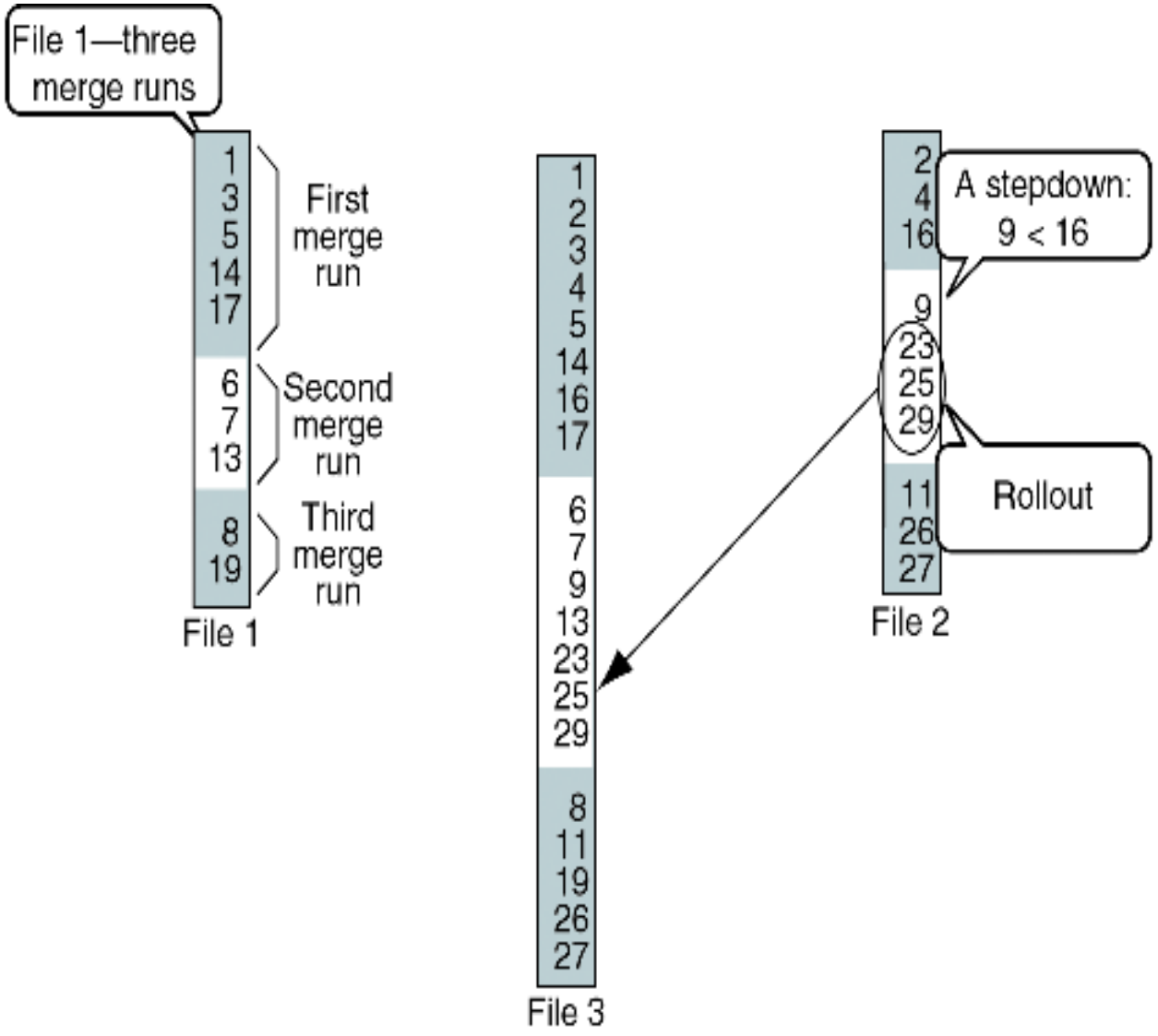
Simple Merge



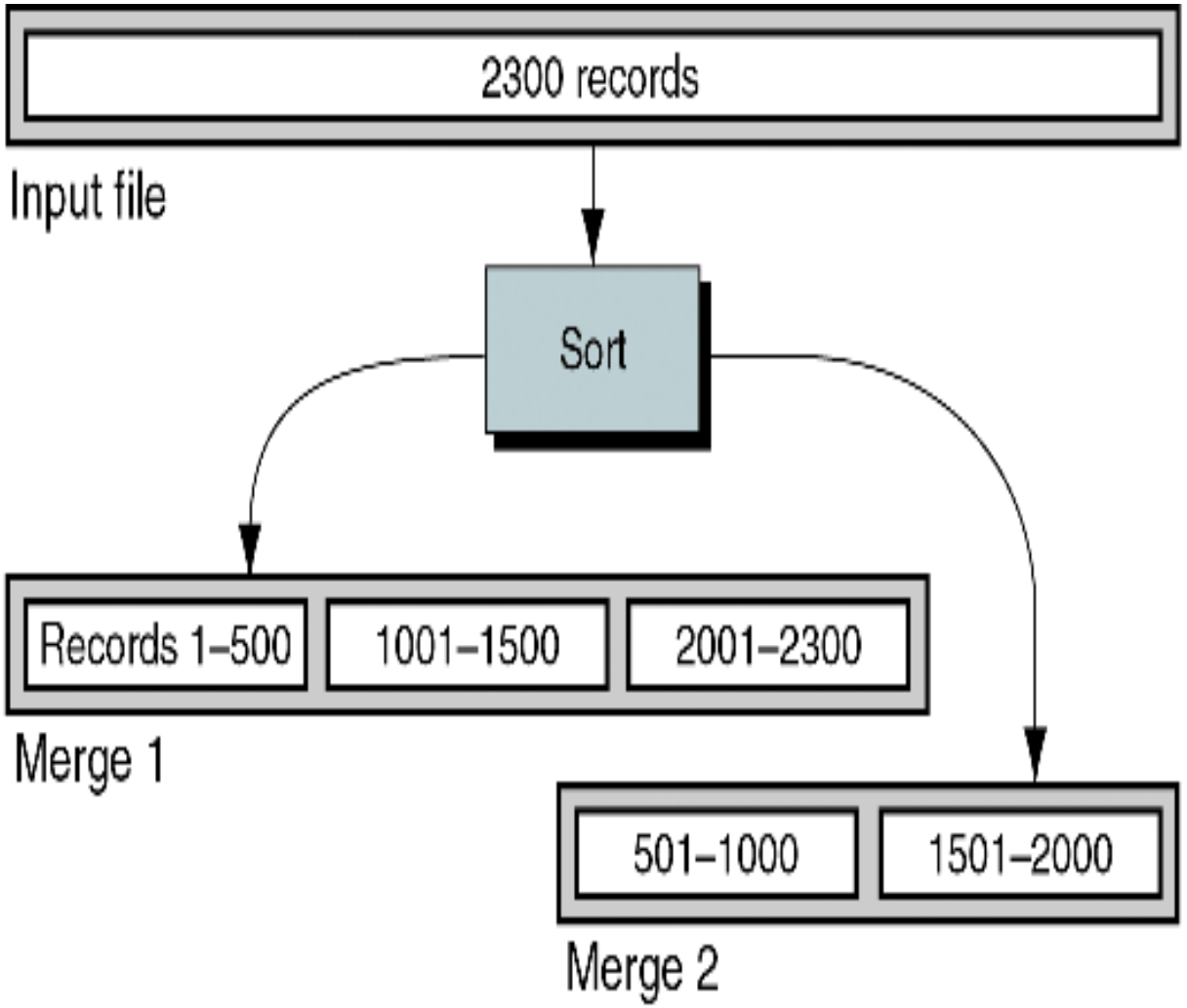
Merge Files

```
Algorithm mergeFiles
Merge two sorted files into one file.
   Pre    input files are sorted
   Post   input files sequentially combined in output file
 1 open files
 2 read (file1 into record1)
 3 read (file2 into record2)
 4 loop (not end file1 OR not end file2)
   1   if (record1 key <= record2 key)
       1  write  (record1 to file3)
       2  read   (file1 into record1)
       3  if (end of file1)
          1  set record1 key to infinity
       4  end if
   2   else
       1  write  (record2 to file3)
       2  read   (file2 into record2)
       3  if (end of file2)
          1  set record2 key to infinity
       4  end if
   3   end if
 5 end loop
 6 close files
end mergeFiles
```

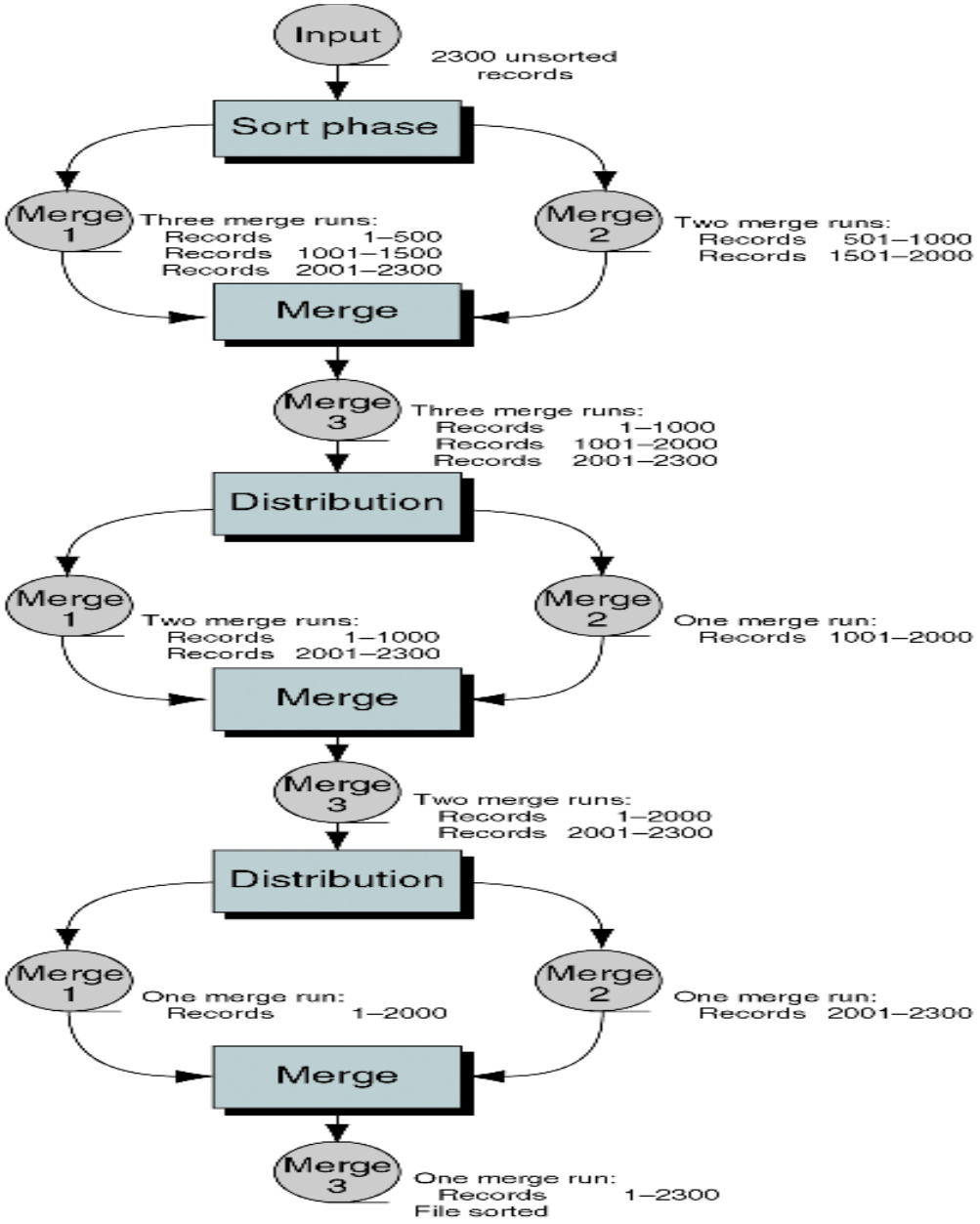
Merge Files Example



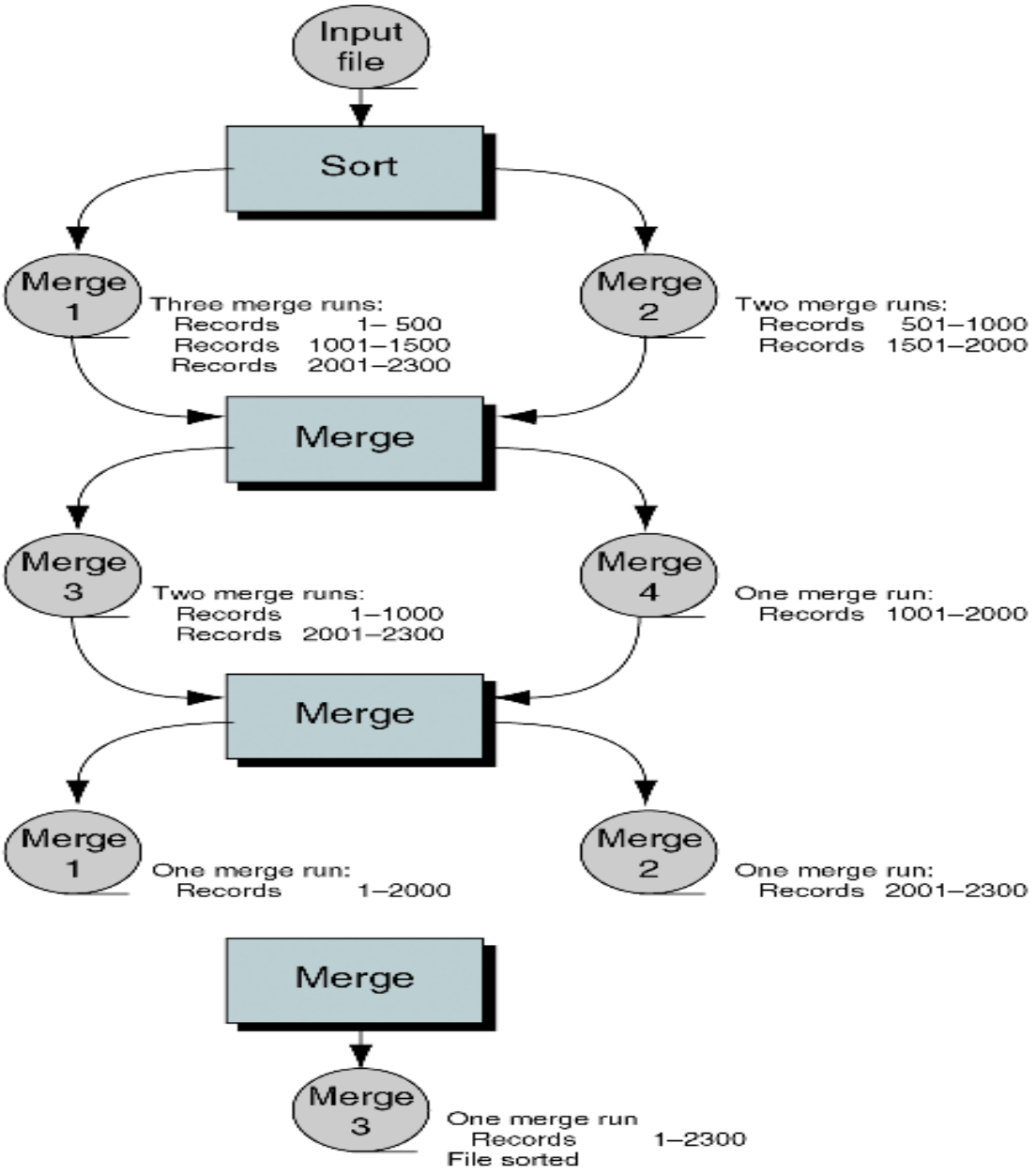
Sort Phase in an External Sort



Natural Two-way Merge Sort



Balanced Two-way Merge Sort



External Sort Phase Using Heap Sort