Searching and Sorting Algorithms
Lecture 14

Introduction

- Using a search algorithm, you can:
  - Determine whether a particular item is in a list
  - If the data is specially organized (for example, sorted), find the location in the list where a new item can be inserted
  - Find the location of an item to be deleted

Search Algorithms

- Key of the item
  - Special member that uniquely identifies the item in the data set
- Key comparison: comparing the key of the search item with the key of an item in the list
  - Can count the number of key comparisons

Sequential Search

- Sequential search (linear search):
  - Same for both array-based and linked lists
  - Starts at first element and examines each element until a match is found
- Our implementation uses an iterative approach
  - Can also be implemented with recursion

Binary Search

- Binary search can be applied to sorted lists
- Uses the “divide and conquer” technique
  - Compare search item to middle element
  - If search item is less than middle element, restrict the search to the lower half of the list
    - Otherwise restrict the search to the upper half of the list

Binary Search (cont'd.)

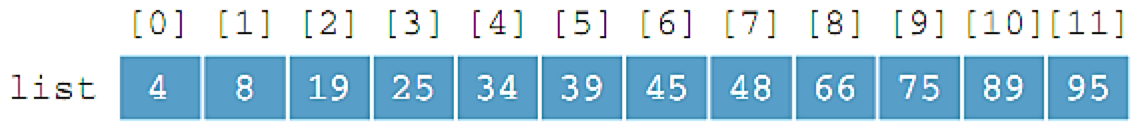

FIGURE 18-1 List of length 12

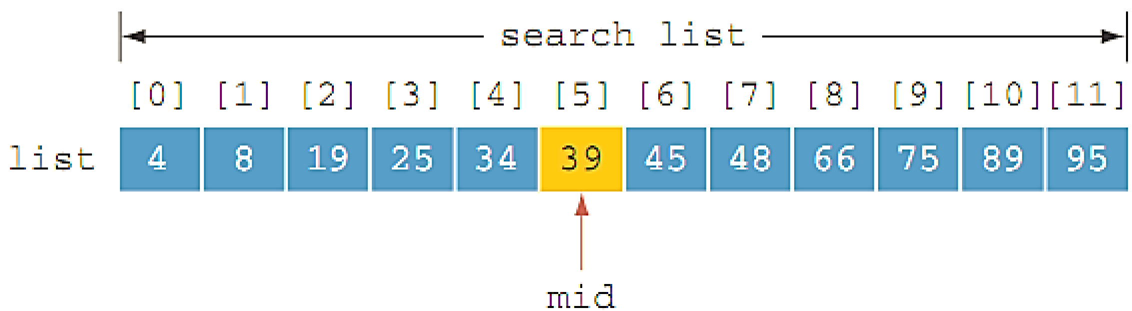

FIGURE 18-2 Search list, `list[0]...list[11]`

Binary Search (cont'd.)

- Search for value of 75:

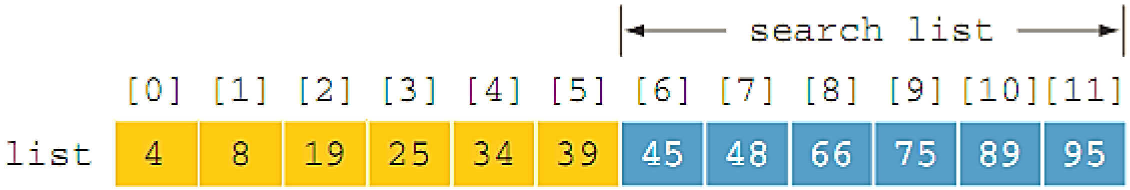

FIGURE 18-3 Search list, `list[6]...list[11]`

Asymptotic Notation: Big-O Notation (cont'd.)

TABLE 18-4 Growth Rate of Various Functions

n	$\log_2 n$	$n\log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4294967296

Asymptotic Notation: Big-O Notation (cont'd.)

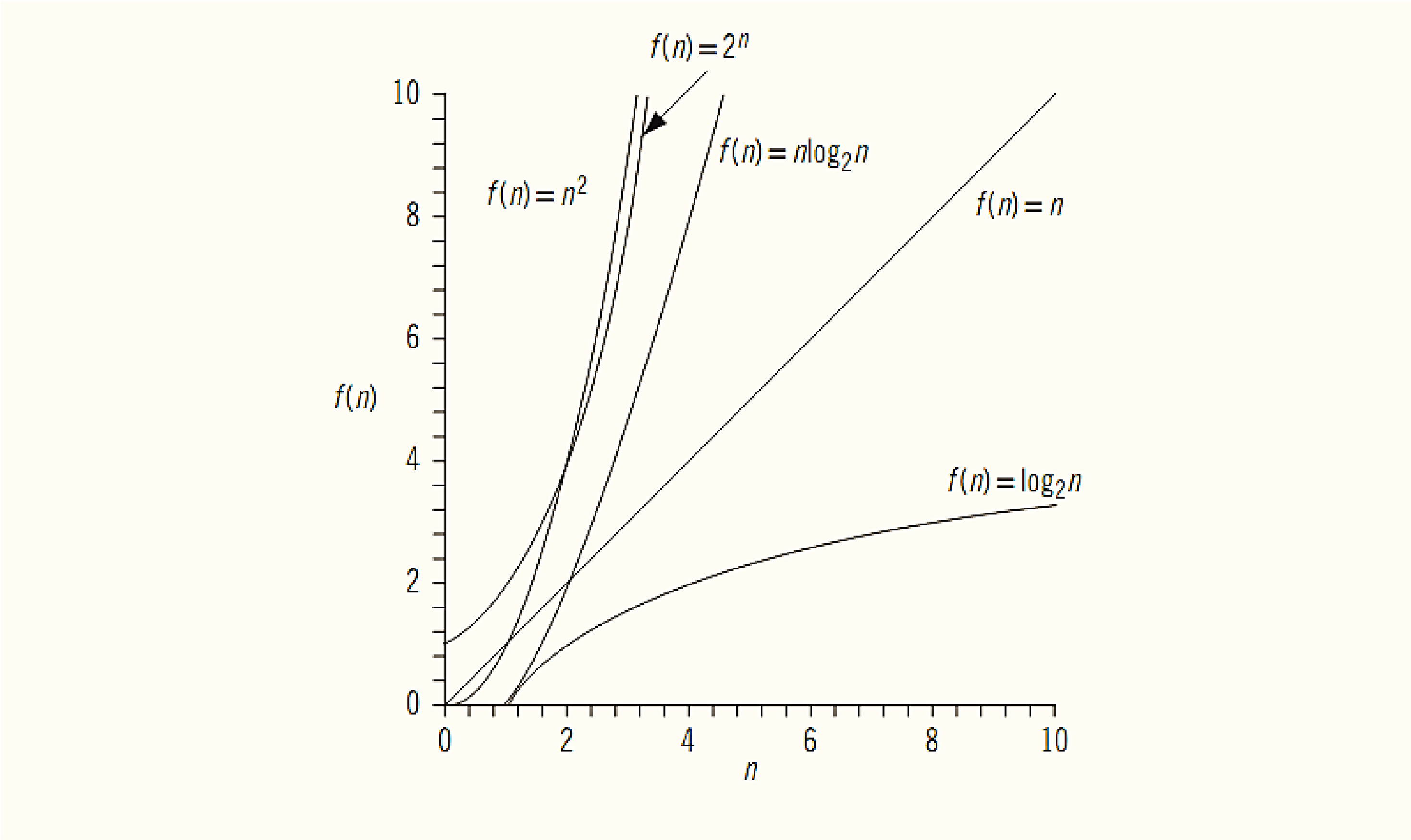

FIGURE 18-5 Growth rate of various functions

Asymptotic Notation: Big-O Notation (cont'd.)

TABLE 18-6 Growth Rate of n^2 and $n^2 + 4n + 20$

n	$g(n) = n^2$	$f(n) = n^2 + 4n + 20$
10	100	160
50	2500	2720
100	10000	10420
1000	1000000	1004020
10000	100000000	100040020

Asymptotic Notation: Big-O Notation (cont'd.)

TABLE 18-7 Some Big-O Functions That Appear in Algorithm Analysis

Function $g(n)$	Growth rate of $f(n)$
$g(n) = 1$	The growth rate is constant, so it does not depend on n, the size of the problem.
$g(n) = \log_2 n$	The growth rate is a function of $\log_2 n$. Because a logarithm function grows slowly, the growth rate of the function f is also slow.
$g(n) = n$	The growth rate is linear. The growth rate of f is directly proportional to the size of the problem.
$g(n) = n\log_2 n$	The growth rate is faster than the linear algorithm.
$g(n) = n^2$	The growth rate of such functions increases rapidly with the size of the problem. The growth rate is quadrupled when the problem size is doubled.
$g(n) = 2^n$	The growth rate is exponential. The growth rate is squared when the problem size is doubled.

Asymptotic Notation: Big-O Notation (cont'd.)

- We can use Big-O notation to compare sequential and binary search algorithms:

TABLE 18-8 Number of Comparisons for a List of Length n

Algorithm	Successful Search	Unsuccessful Search
Sequential search	$\frac{n+1}{2} = \frac{1}{2}n + \frac{1}{2} = O(n)$	$n = O(n)$
Binary search	$2\log_2 n - 3 = O(\log_2 n)$	$2\log_2 n = O(\log_2 n)$

Sorting Algorithms

- To compare the performance of commonly used sorting algorithms
  - Must provide some analysis of these algorithms
- These sorting algorithms can be applied to either array-based lists or linked lists

Sorting a List: Bubble Sort

- Suppose `list[0]...list[n-1]` is a list of *n* elements, indexed `0` to `n-1`
- <u>Bubble sort algorithm</u>:
  - In a series of `n-1` iterations, compare successive elements, `list[index]` and `list[index+1]`
  - If `list[index]` is greater than `list[index+1]`, then swap them

Sorting a List: Bubble Sort (cont'd.)

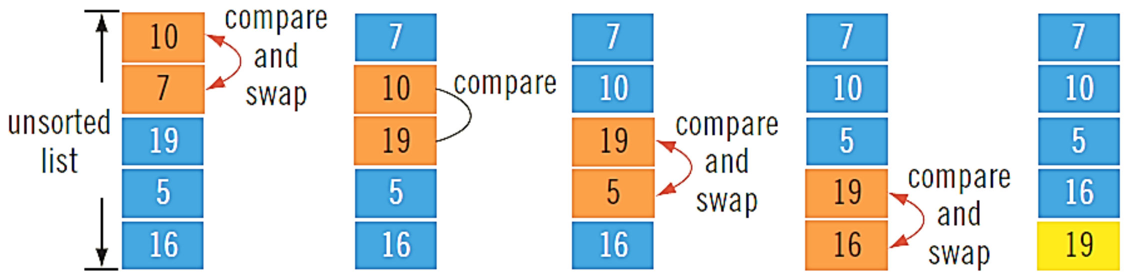

FIGURE 18-7 Elements of `list` during the first iteration

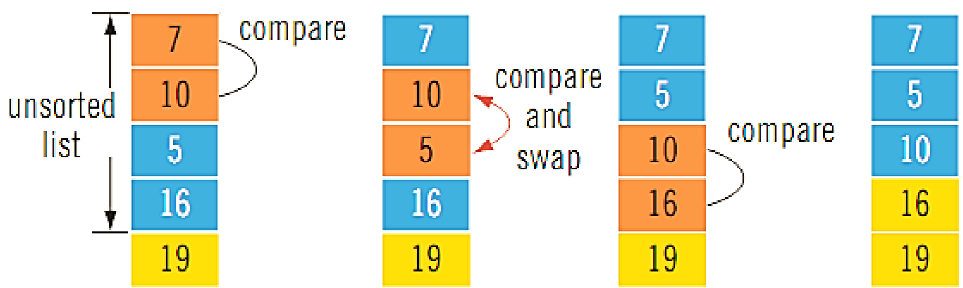

FIGURE 18-8 Elements of `list` during the second iteration

Sorting a List: Bubble Sort (cont'd.)

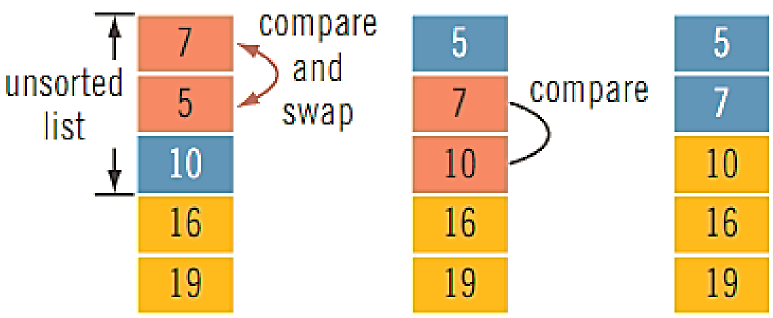

FIGURE 18-9 Elements of `list` during the third iteration

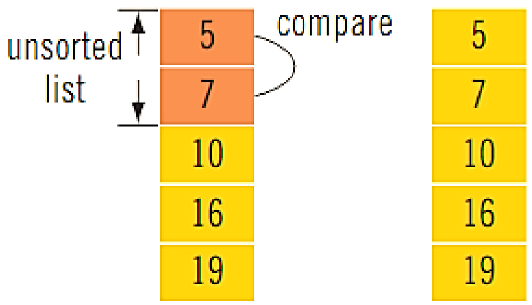

FIGURE 18-10 Elements of `list` during the fourth iteration

Analysis: Bubble Sort

- `bubbleSort` contains nested loops
  - Outer loop executes $n - 1$ times
  - For each iteration of outer loop, inner loop executes a certain number of times
- Total number of comparisons:

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$$

- Number of assignments (worst case):

$$3\frac{n(n-1)}{2} = \frac{3}{2}n^2 - \frac{3}{2}n = O(n^2)$$

Selection Sort: Array-Based Lists

- Selection sort algorithm: rearrange list by selecting an element and moving it to its proper position
- Find the smallest (or largest) element and move it to the beginning (end) of the list
- Can also be applied to linked lists

Analysis: Selection Sort

- function `swap`: does three assignments; executed $n-1$ times
  - $3(n - 1) = O(n)$
- function `minLocation`:
  - For a list of length k, $k-1$ key comparisons
  - Executed $n-1$ times (by `selectionSort`)
  - Number of key comparisons:

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = \frac{1}{2}n^2 + O(n) = O(n^2)$$

Insertion Sort: Array-Based Lists

- Insertion sort algorithm: sorts the list by moving each element to its proper place in the sorted portion of the list

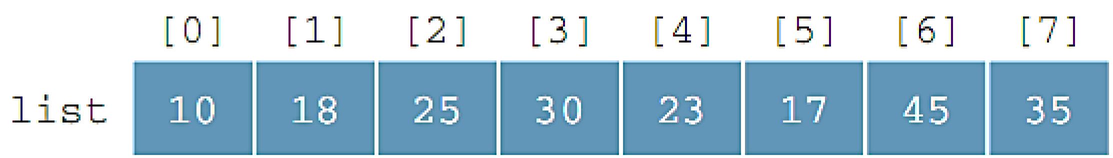

FIGURE 18-11 list

Insertion Sort: Array-Based Lists (cont'd.)

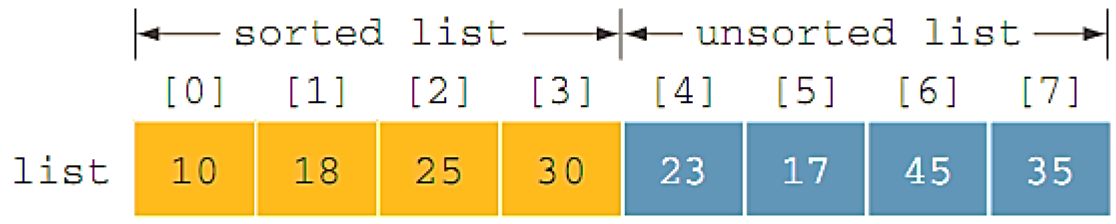

FIGURE 18-12 Sorted and unsorted portion of `list`

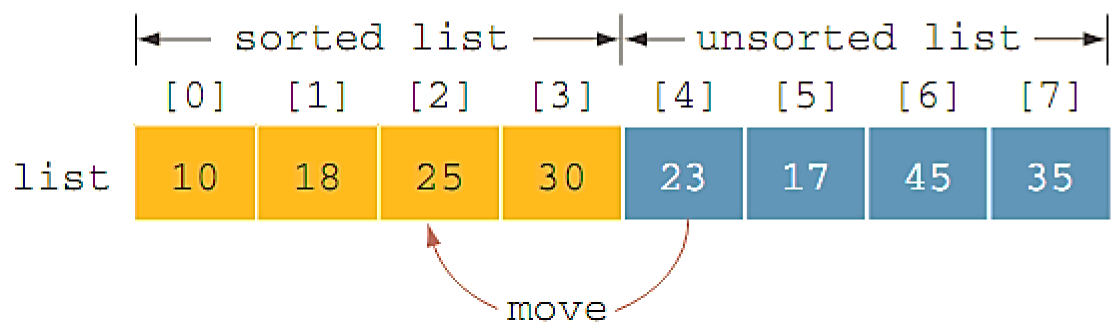

FIGURE 18-13 Move `list[4]` into `list[2]`

Insertion Sort: Array-Based Lists (cont'd.)

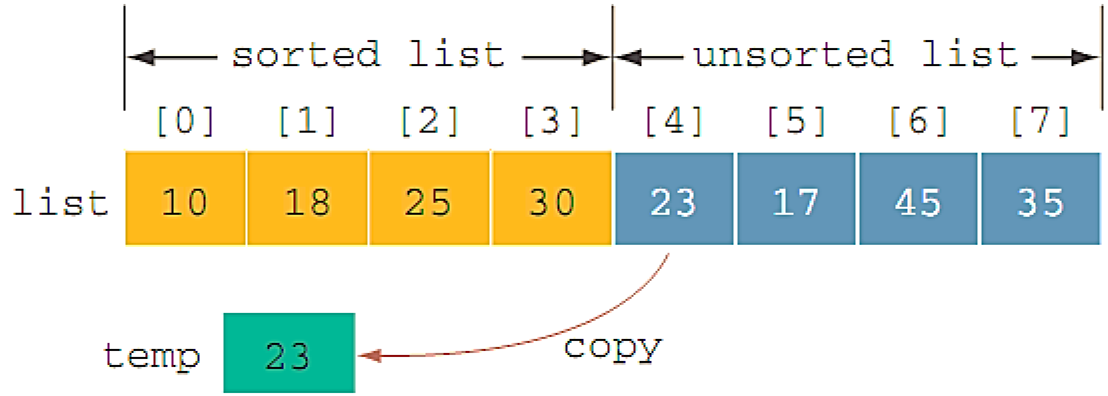

FIGURE 18-14 Copy `list[4]` into `temp`

Insertion Sort: Array-Based Lists (cont'd.)

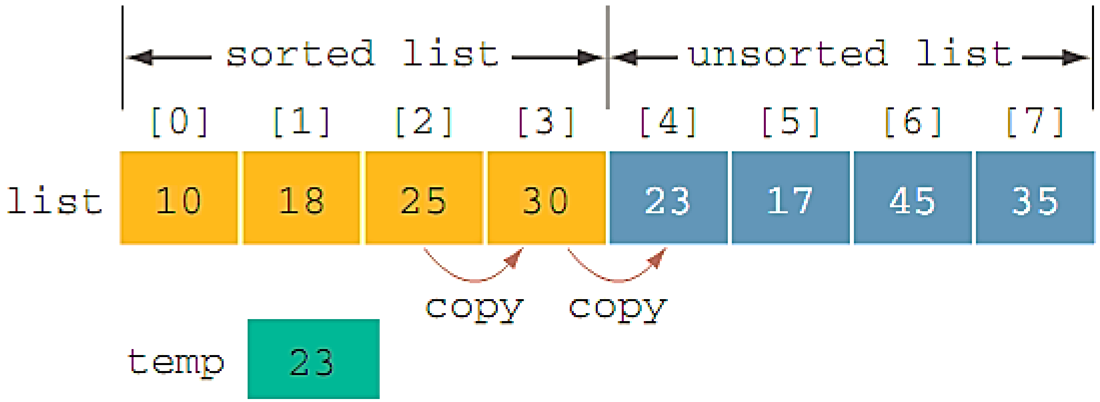

FIGURE 18-15 List before copying `list[3]` into `list[4]` and then `list[2]` into `list[3]`

Insertion Sort: Array-Based Lists (cont'd.)

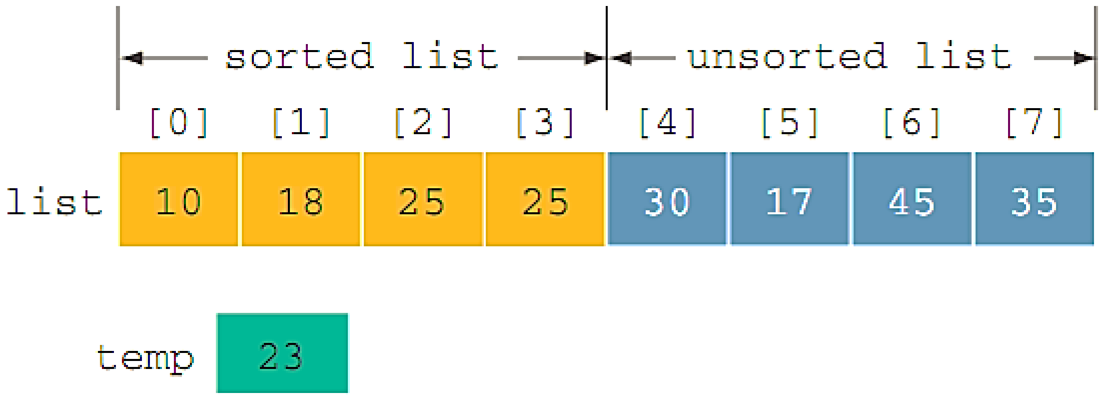

FIGURE 18-16 List after copying `list[3]` into `list[4]` and then `list[2]` into `list[3]`

Insertion Sort: Array-Based Lists (cont'd.)

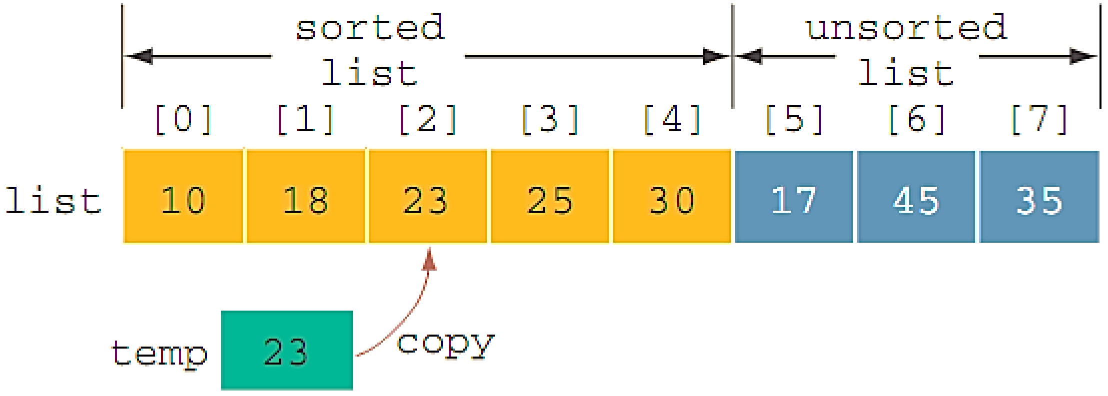

FIGURE 18-17 List after copying `temp` into `list[2]`

Analysis: Insertion Sort

- The `for` loop executes $n - 1$ times
- Best case (list is already sorted):
  - Key comparisons: $n - 1 = O(n)$
- Worst case: for each `for` iteration, `if` statement evaluates to `true`
  - Key comparisons: $1 + 2 + \ldots + (n - 1) = n(n - 1) / 2 = O(n^2)$
- Average number of key comparisons and of item assignments: $\frac{1}{4} n^2 + O(n) = O(n^2)$

Analysis: Insertion Sort (cont'd.)

TABLE 18-9 Average Case Behavior of the Bubble Sort, Selection Sort, and Insertion Sort Algorithms for a List of Length n

Algorithm	Number of Comparisons	Number of Swaps
Bubble sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{4} = O(n^2)$
Selection sort	$\frac{n(n-1)}{2} = O(n^2)$	$3(n-1) = O(n)$
Insertion sort	$\frac{1}{4}n^2 + O(n) = O(n^2)$	$\frac{1}{4}n^2 + O(n) = O(n^2)$

Quick Sort: Array-Based Lists

- Quick sort: uses the divide-and-conquer technique
  - The list is partitioned into two sublists
  - Each sublist is then sorted
  - Sorted sublists are combined into one list in such a way that the combined list is sorted
  - All of the sorting work occurs during the partitioning of the list

Quick Sort: Array-Based Lists (cont'd.)

- **`pivot`** element is chosen to divide the list into: `lowerSublist` and `upperSublist`
  - The elements in `lowerSublist` are < `pivot`
  - The elements in `upperSublist` are ≥ `pivot`
- Pivot can be chosen in several ways
  - Ideally, the pivot divides the list into two sublists of nearly- equal size

Quick Sort: Array-Based Lists (cont'd.)

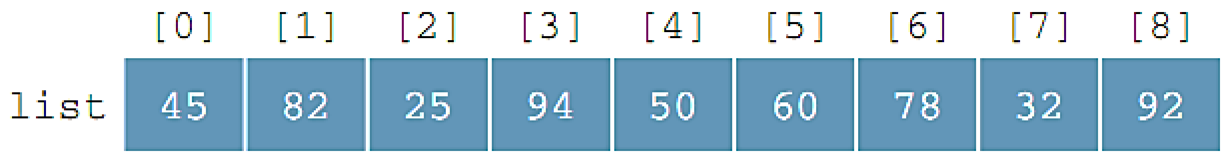

FIGURE 18-19 `list` before the partition

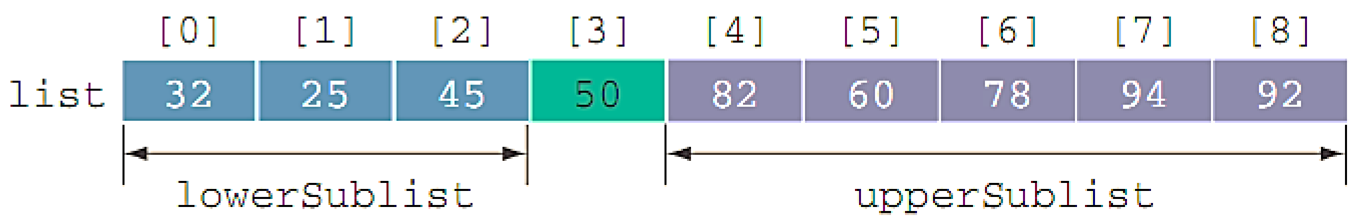

FIGURE 18-20 `list` after the partition

Quick Sort: Array-Based Lists (cont'd.)

- Partition algorithm (assumes that `pivot` is chosen as the middle element of the list):
  1. Determine `pivot`; swap it with the first element of the list
  2. For the remaining elements in the list:
     - If the current element is less than pivot, (1) increment `smallIndex`, and (2) swap current element with element pointed by `smallIndex`
  - Swap the first element (`pivot`), with the array element pointed to by `smallIndex`

Quick Sort: Array-Based Lists (cont'd.)

- Step 1 determines the pivot and moves `pivot` to the first array position
- During Step 2, list elements are arranged

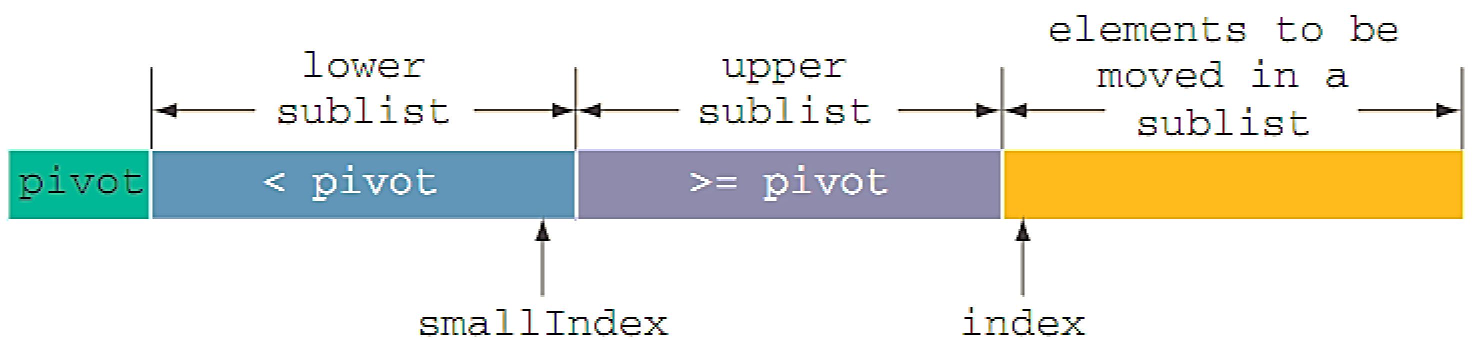

FIGURE 18-21 List during the execution of Step 2

Quick Sort: Array-Based Lists (cont'd.)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
32	55	87	13	78	96	52	48	22	11	58	66	88	45

FIGURE 18-22 List before sorting

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
52	55	87	13	78	96	32	48	22	11	58	66	88	45
`pivot`													

FIGURE 18-23 List after moving `pivot` to the first array position

Quick Sort: Array-Based Lists (cont'd.)

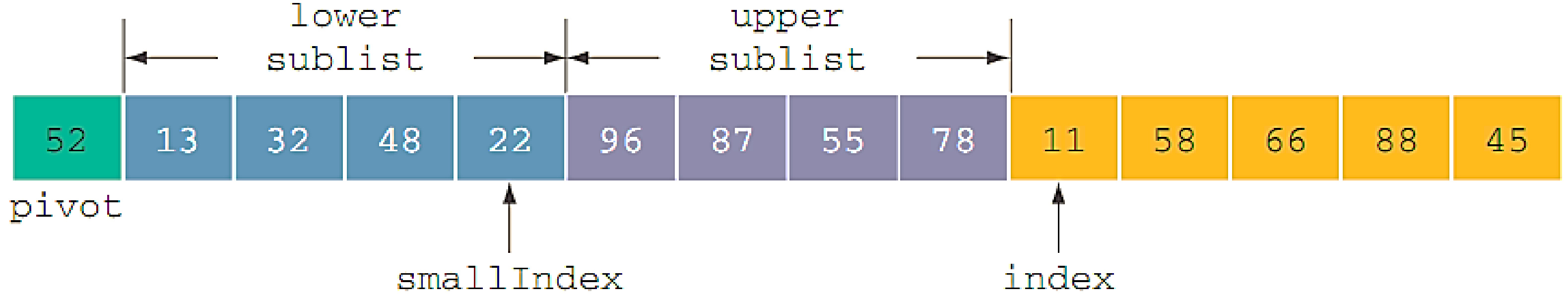

FIGURE 18-24 List after a few iterations of Step 2

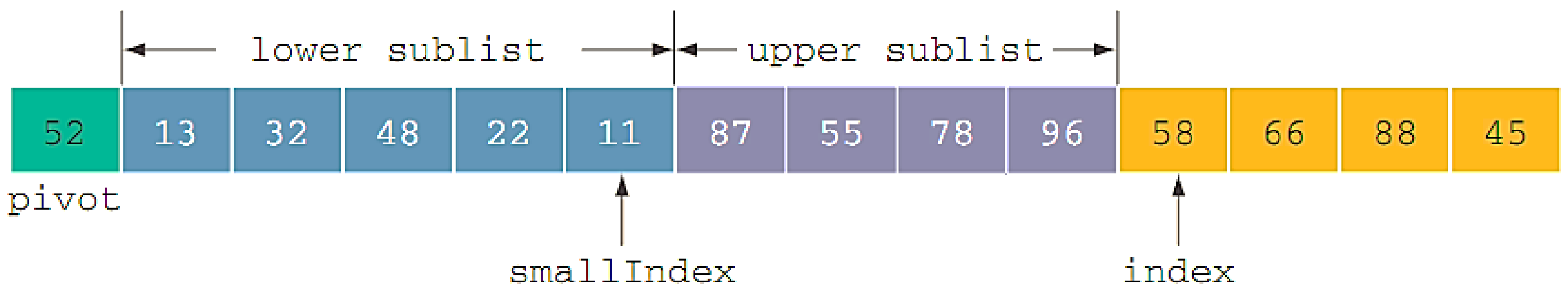

FIGURE 18-25 List after moving 11 into the lower sublist

Quick Sort: Array-Based Lists (cont'd.)

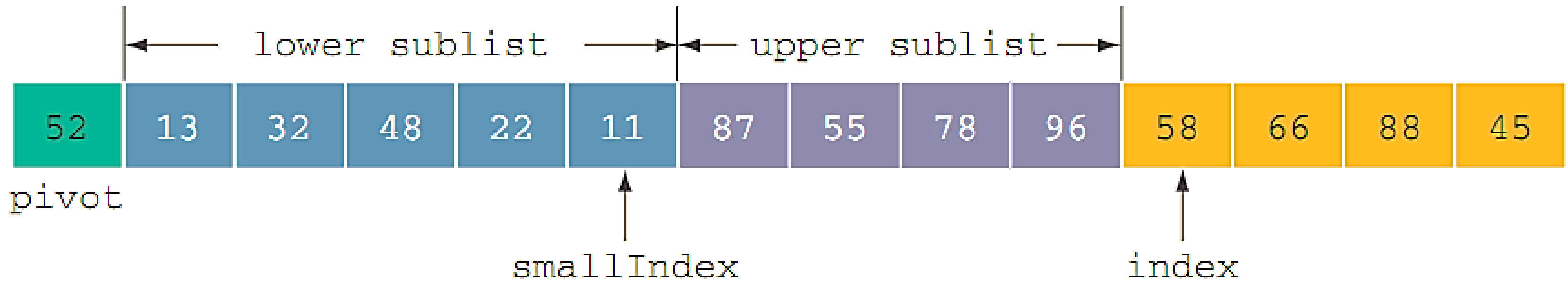

FIGURE 18-26 List before moving 58 into a sublist

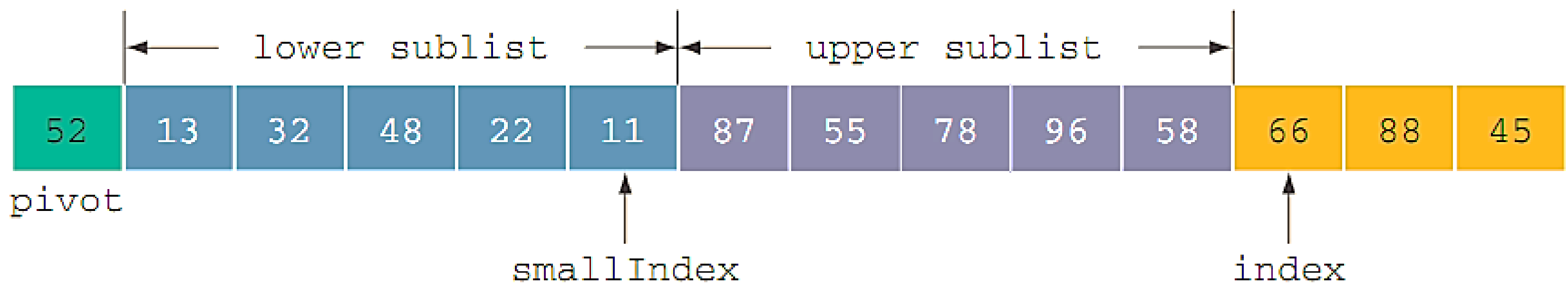

FIGURE 18-27 List after moving 58 into the upper sublist

Quick Sort: Array-Based Lists (cont'd.)

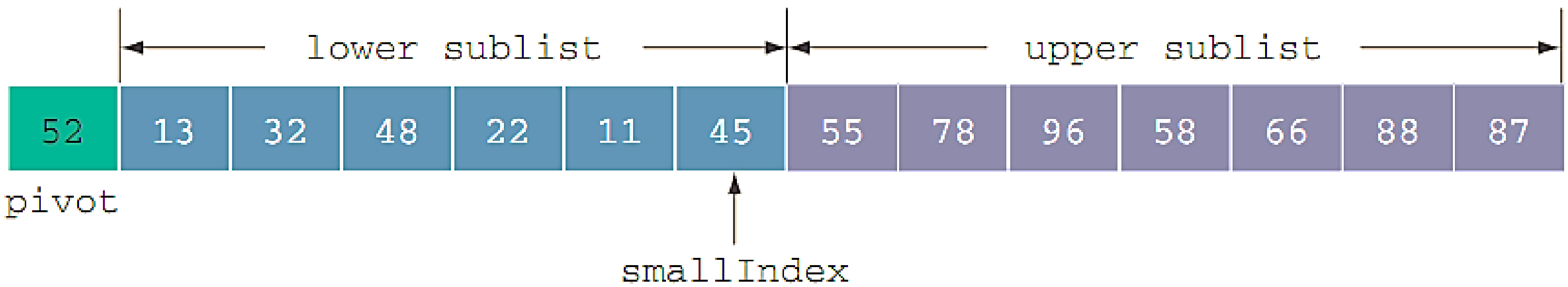

FIGURE 18-28 List elements after arranging into the lower sublist and upper sublist

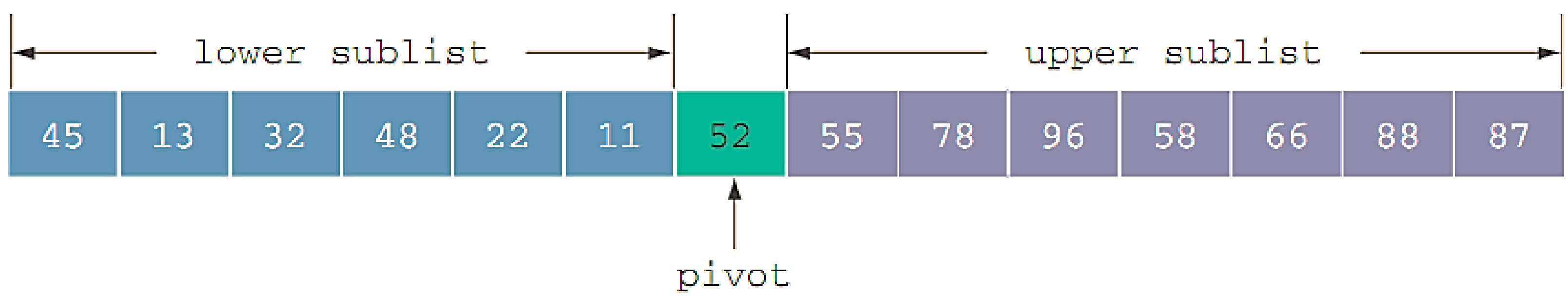

FIGURE 18-29 List after swapping 52 with 45

Merge Sort: Linked List-Based Lists

- Quick sort: $O(n\log_2 n)$ average case; $O(n^2)$ worst case
- Merge sort: always $O(n\log_2 n)$
  - Uses the divide-and-conquer technique
    - Partitions the list into two sublists
    - Sorts the sublists
    - Combines the sublists into one sorted list
  - Differs from quick sort in how list is partitioned
    - Divides list into two sublists of nearly equal size

Merge Sort: Linked List-Based Lists (cont'd.)

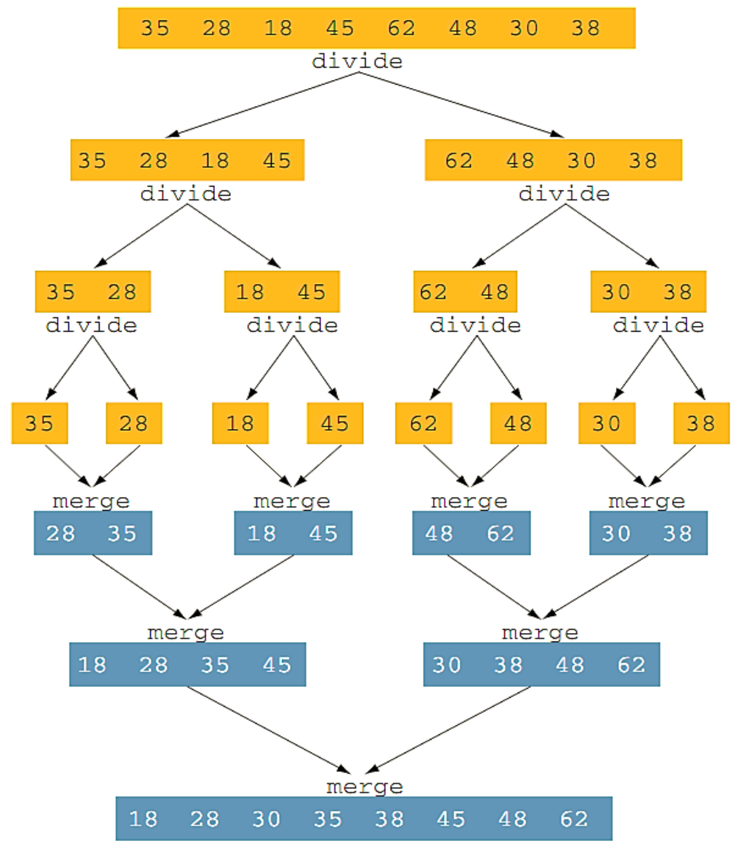

FIGURE 18-30 Merge sort algorithm

Merge Sort: Linked List-Based Lists (cont'd.)

- General algorithm:

```
if the list is of a size greater than 1
{
   a. Divide the list into two sublists.
   b. Merge sort the first sublist.
   c. Merge sort the second sublist.
   d. Merge the first sublist and the second sublist.
}
```

- Uses recursion

Divide

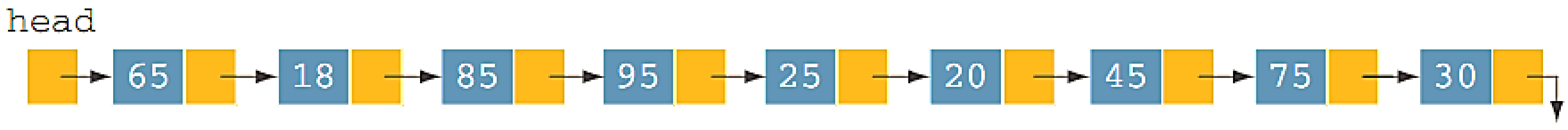

FIGURE 18-31 Unsorted linked list

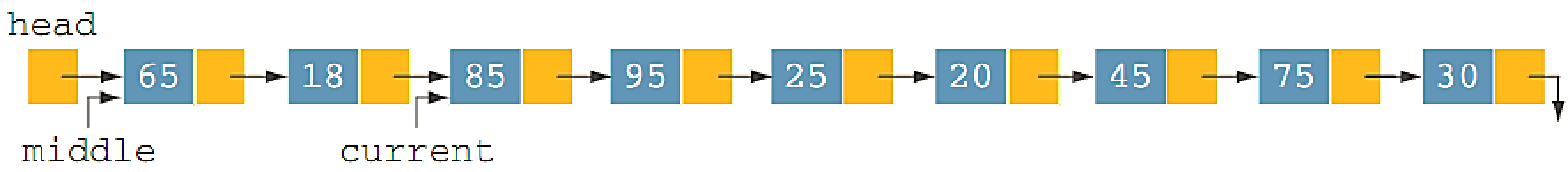

FIGURE 18-32 `middle` and `current` before traversing the list

Divide (cont'd.)

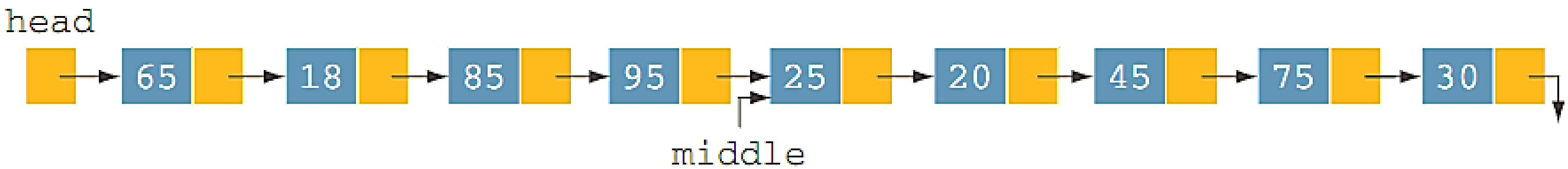

FIGURE 18-33 `middle` after traversing the list

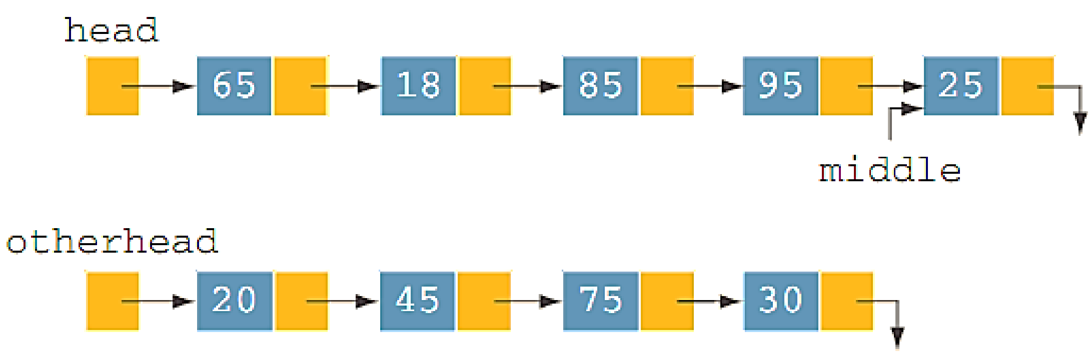

FIGURE 18-34 List after dividing it into two lists

Merge

- Sorted sublists are merged into a sorted list
  - Compare elements of sublists
  - Adjust pointers of nodes with smaller `info`

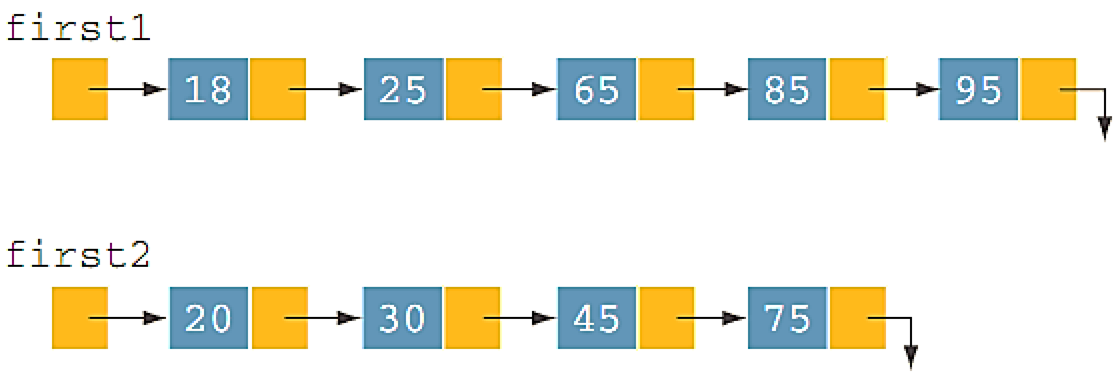

FIGURE 18-35 Sublists before merging

Merge (cont'd.)

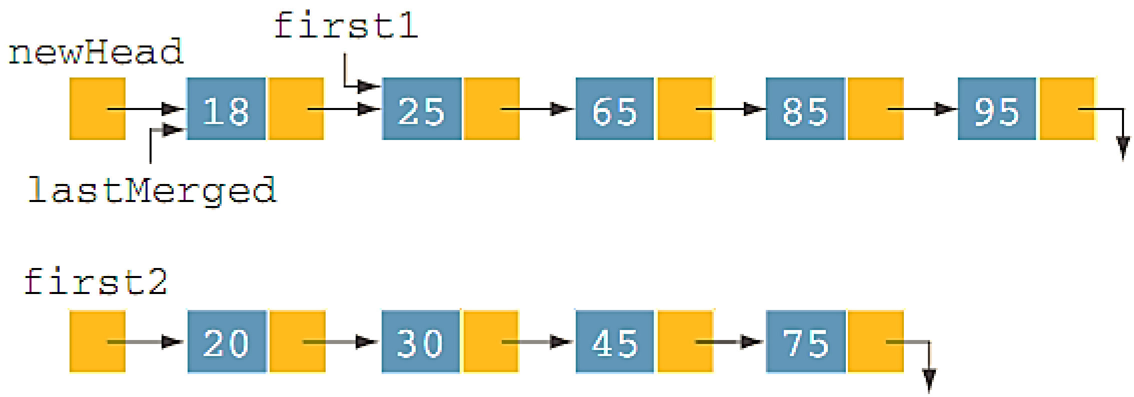

FIGURE 18-36 Sublists after setting `newHead` and `lastMerged` and advancing `first1`

Merge (cont'd.)

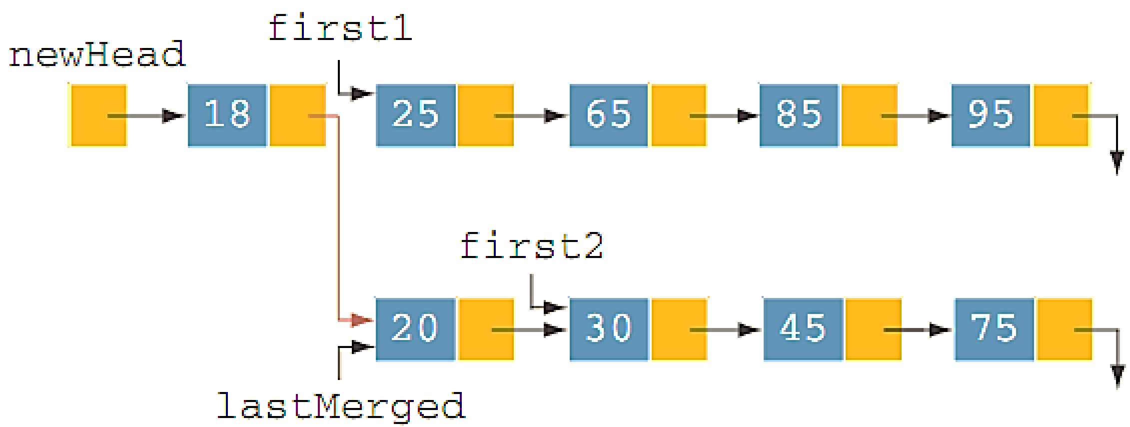

FIGURE 18-37 Merged list after putting the node with `info` 20 at the end of the merged list