***This chapter is a bonus Web chapter

CHAPTER 17
Sorting

Objectives

- To study and analyze time efficiency of various sorting algorithms (§§17.2–17.7).
- To design, implement, and analyze bubble sort (§17.2).
- To design, implement, and analyze merge sort (§17.3).
- To design, implement, and analyze quick sort (§17.4).
- To design and implement a heap (§17.5).
- To design, implement, and analyze heap sort (§17.5).
- To design, implement, and analyze bucket sort and radix sort (§17.6).

17.1 Introduction

<key point>
Sorting algorithms are good examples for algorithm design and analysis.
<end key point>

<margin note: why study sorting?>
When presidential candidate Obama visited Google in 2007, the Google CEO Eric Schmidt asked Obama the most efficient way to sort a million 32-bit integers (http://www.youtube.com/watch?v=k4RRi_ntQc8). Obama answered that bubble sort would be the wrong way to go. Is he right? We will examine it in this chapter.

<margin note: why study sorting?>
Sorting is a classic subject in computer science. There are three reasons to study sorting algorithms.

- First, sorting algorithms illustrate many creative approaches to problem solving, and these approaches can be applied to solve other problems.
- Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, functions, and lists.
- Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

<margin note: what data to sort?>
The data to be sorted might be integers, doubles, strings, or other items. Data may be sorted in increasing order or decreasing order. For simplicity, this chapter assumes:

(1) data to be sorted are integers,
(2) data are stored in a list, and
(3) data are sorted in ascending order.

There are many algorithms for sorting. You have already learned selection sort and insertion sort. This chapter introduces bubble sort, merge sort, quick sort, bucket sort, and radix sort.

17.2 Bubble Sort

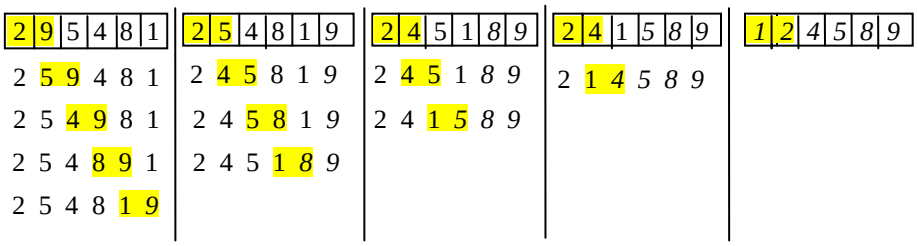
<key point>
Bubble sort sorts the list by sorting the neighboring elements in multiple phases.
<end key point>

The bubble sort algorithm makes several passes through the list. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or *sinking sort*, because the smaller values gradually "bubble" their way to the top and the larger values sink to the bottom. After first pass, the last element becomes the largest in the list. After the second pass, the second-to-last element becomes the second largest in the list. This process is continued until all elements are sorted.

<margin note: bubble sort illustration>
Figure 17.1a shows the first pass of a bubble sort of a list of six elements (2 9 5 4 8 1). Compare the elements in the first pair (2 and 9), and no swap is needed because they are already in order. Compare the elements in the second pair (9 and 5), and swap 9 with 5 because 9 is greater than 5. Compare the elements in the third pair (9 and 4), and swap 9 with 4. Compare the elements in the fourth pair (9 and 8), and swap 9 with 8. Compare the

elements in the fifth pair (9 and 1), and swap 9 with 1. The pairs being compared are highlighted and the numbers already sorted are italicized.

***margin note: see Supplement VI for bubble sort animation



(a) 1st pass (b) 2nd pass (c) 3rd pass (d) 4th pass (e) 5th pass

Figure 17.1
Each pass compares and orders the pairs of elements sequentially.

The first pass places the largest number (9) as the last in the list. In the second pass, as shown in Figure 17.1b, you compare and order pairs of elements sequentially. There is no need to consider the last pair, because the last element in the list is already the largest. In the third pass, as shown in Figure 17.1c, you compare and order pairs of elements sequentially except the last two elements, because they are already ordered. So in the *k*th pass, you don't need to consider the last *k* – 1 elements, because they are already ordered.

<margin note: algorithm>
The algorithm for bubble sort can be described in Listing 17.1:

Listing 17.1 Bubble Sort Algorithm

```
for k in range(1, len(list)):
    # Perform the kth pass
    for i in range(len(list) - k):
        if list[i] > list[i + 1]:
            swap list[i] with list[i + 1]
```

Note that if no swap takes place in a pass, there is no need to perform the next pass, because all the elements are already sorted. You may use this property to improve the preceding algorithm as in Listing 17.2.

Listing 17.2 Improved Bubble Sort Algorithm

```
needNextPass = True
k = 1
while k < len(list) and needNextPass:
    # List may be sorted and next pass not needed
    needNextPass = False
    # Perform the kth pass
    for i in range(len(list) - k):
        if list[i] > list[i + 1]:
            swap list[i] with list[i + 1]
            needNextPass = True  # Next pass still needed

    k += 1
```

The algorithm can be implemented as in Listing 17.3:

Listing 17.3 BubbleSort.py

<margin note line 8: perform one pass>

```
 1  def bubbleSort(list):
 2      needNextPass = True
 3
 4      k = 1
 5      while k < len(list) and needNextPass:
 6          # List may be sorted and next pass not needed
 7          needNextPass = False
 8          for i in range(len(list) - k):
 9              if list[i] > list[i + 1]:
10                  # swap list[i] with list[i + 1]
11                  list[i], list[i + 1] = list[i + 1], list[i]
12
13                  needNextPass = True # Next pass still needed
14
15  # A test function
16  def main():
17      list = [2, 3, 2, 5, 6, 1, -2, 3, 14, 12]
18      bubbleSort(list)
19      for v in list:
20          print(str(v) + " ", end = "")
21
22  main() # Call the main function
```

<Output>
```
-2 1 2 2 3 3 5 6 12 14
```
<End Output>

17.2.1 Bubble Sort Time

In the best-case, the bubble sort algorithm needs just the first pass to find that the list is already sorted. No next pass is needed. Since the number of comparisons is $n-1$ in the first pass, the best-case time for bubble sort is $O(n)$.

In the worst case, the bubble sort algorithm requires $n-1$ passes. The first pass takes $n-1$ comparisons; the second pass takes $n-2$ comparisons; and so on; the last pass takes 1 comparison. So, the total number of comparisons is:

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

Therefore, the worst-case time for bubble sort is $O(n^2)$.

17.3 Merge Sort

<key point>
The merge sort algorithm can be described recursively as follows: The algorithm divides the array into two halves and applies merge sort on each half recursively. After the two halves are sorted, merge them.
<end key point>

The algorithm is given in Listing 17.4.

Listing 17.4 Merge Sort Algorithm

<margin note line 2: base condition>
<margin note line 3: sort first half>
<margin note line 4: sort second half>
<margin note line 5: merge two halves>

```
1  def mergeSort(list):
2      if len(list) > 1:
3          mergeSort(list[0 ... len(list) / 2])
4          mergeSort(list[len(list) / 2 + 1 ... len(list)])
5          merge list[0 ... len(list) / 2] with
6              list[len(list) / 2 + 1 ... len(list)]
```

<margin note: merge sort illustration>

Figure 17.2 illustrates a merge sort of a list of eight elements (2 9 5 4 8 1 6 7). The original list is split into (2 9 5 4) and (8 1 6 7). Apply merge sort on these two sublists recursively to split (1 9 5 4) into (1 9) and (5 4) and (8 1 6 7) into (8 1) and (6 7). This process continues until the sublist contains only one element. For example, list (2 9) is split into sublists (2) and (9). Since list (2) contains a single element, it cannot be further split. Now merge (2) with (9) into a new sorted list (2 9); merge (5) with (4) into a new sorted list (4 5). Merge (2 9) with (4 5) into a new sorted list (2 4 5 9), and finally merge (2 4 5 9) with (1 6 7 8) into a new sorted list (1 2 4 5 6 7 8 9).

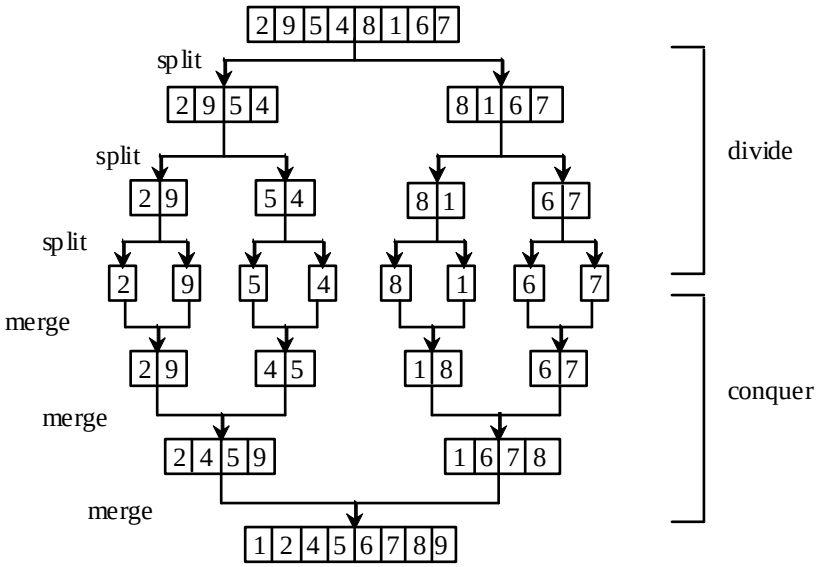


Figure 17.2
Merge sort employs a divide-and-conquer approach to sort the list.

The recursive call continues dividing the list into sublists until each sublist contains only one element. The algorithm then merges these small sublists into larger sorted sublists until one sorted list results.

The merge sort algorithm is implemented in Listing 17.5.

Listing 17.5 MergeSort.py

<margin note line 2: base case>
<margin note line 5: sort first half>
<margin note line 9: sort second half>
<margin note line 12: merge two halves>
<margin note line 22: list1 to temp>
<margin note line 26: list2 to temp>
<margin note line 30: rest of list1 to temp>
<margin note line 35: rest of list2 to temp>

```
1  def mergeSort(list):
2      if len(list) > 1:
3          # Merge sort the first half
```

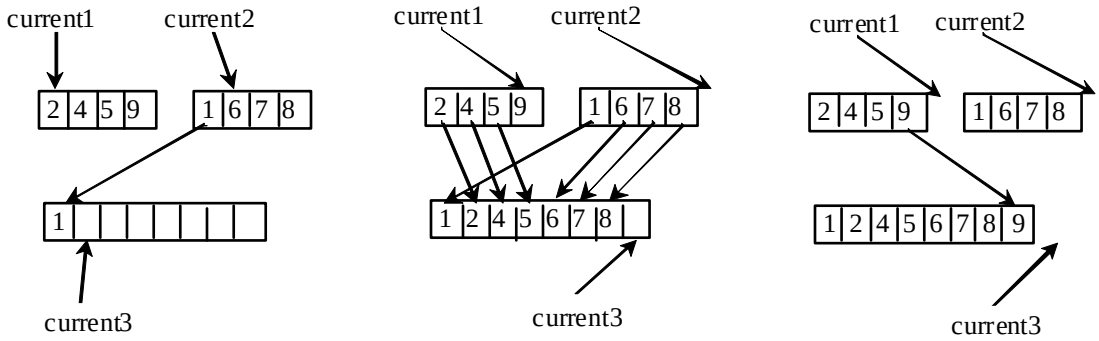
```
 4          firstHalf = list[ : len(list) // 2]
 5          mergeSort(firstHalf)
 6
 7          # Merge sort the second half
 8          secondHalf = list[len(list) // 2 : ]
 9          mergeSort(secondHalf)
10
11          # Merge firstHalf with secondHalf into list
12          merge(firstHalf, secondHalf, list)
13
14  # Merge two sorted lists */
15  def merge(list1, list2, temp):
16      current1 = 0  # Current index in list1
17      current2 = 0  # Current index in list2
18      current3 = 0  # Current index in temp
19
20      while current1 < len(list1) and current2 < len(list2):
21          if list1[current1] < list2[current2]:
22              temp[current3] = list1[current1]
23              current1 += 1
24              current3 += 1
25          else:
26              temp[current3] = list2[current2]
27              current2 += 1
28              current3 += 1
29
30      while current1 < len(list1):
31          temp[current3] = list1[current1]
32          current1 += 1
33          current3 += 1
34
35      while current2 < len(list2):
36          temp[current3] = list2[current2]
37          current2 += 1
38          current3 += 1
39
40  def main():
41      list = [2, 3, 2, 5, 6, 1, -2, 3, 14, 12]
42      mergeSort(list)
43      for v in list:
44          print(str(v) + " ", end = "")
45
46  main()
```

The `mergeSort` function (lines 1–12) creates a new list `firstHalf`, which is a copy of the first half of `list` (line 4). The algorithm invokes `mergeSort` recursively on `firstHalf` (line 5). The new list `secondHalf` was created to contain the second part of the original list (line 8). The algorithm invokes `mergeSort` recursively on `secondHalf` (line 9). After `firstHalf` and `secondHalf` are sorted, they are merged to `list` (line 12).

The `merge` function (lines 15–38) merges two sorted lists `list1` and `list2` into list `temp`. `current1` and `current2` point to the current element to be considered in `list1` and `list2` (lines 16–17). The function repeatedly compares the current elements from `list1` and `list2` and moves the smaller one to `temp`. `current1` is increased by `1` (line 23) if the smaller one is in `list1` and `current2` is increased by `1` (line 27) if the smaller one is in `list2`. Finally, all the elements in one of the lists are moved to `temp`. If there are still unmoved elements in `list1`, copy them to `temp` (lines 30–33). If there are still unmoved elements in `list2`, copy them to `temp` (lines 35–38).

***animation icon: merge animation

Figure 17.3 illustrates how to merge two lists **list1** (2 4 5 9) and **list2** (1 6 7 8). Initially the current elements to be considered in the lists are 2 and 1. Compare them and move the smaller element 1 to **temp**, as shown in Figure 17.3a. **current2** and **current3** are increased by 1. Continue to compare the current elements in the two lists and move the smaller one to **temp** until one of the lists is completely moved. As shown in Figure 17.3b, all the elements in **list2** are moved to **temp** and **current1** points to element 9 in **list1**. Copy 9 to **temp**, as shown in Figure 17.3c.



(a) After moving 1 to temp

(b) After moving all the elements in list2 to temp

(c) After moving 9 to temp

Figure 17.3

Two sorted lists are merged into one sorted list.

17.3.1 Merge Sort Time

<margin note: time analysis>

Let $T(n)$ denote the time required for sorting a list of n elements using merge sort. Without loss of generality, assume n is a power of 2. The merge sort algorithm splits the list into two sublists, sorts the sublists using the same algorithm recursively, and then merges the sublists. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + mergetime$$

The first $T(\frac{n}{2})$ is the time for sorting the first half of the list and the second $T(\frac{n}{2})$ is the time for sorting the second half. To merge two sublists, it takes at most $n-1$ comparisons to compare the elements from the two sublists and n moves to move elements to the temporary list. So, the total time is $2n-1$. Therefore,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 2n - 1 = O(n\log n)$$

<margin note: $O(n\log n)$ merge sort>

The complexity of merge sort is $O(n\log n)$. This algorithm is better than selection sort, insertion sort, and bubble sort.

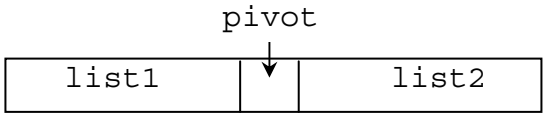
17.4 Quick Sort

<key point>
Quick sort works as follows: The algorithm selects an element, called the *pivot*, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part.
<end key point>

Quick sort was developed by C. A. R. Hoare (1962). The algorithm is described in Listing 17.6.

Listing 17.6 Quick Sort Algorithm

<margin note line 2: base condition>
<margin note line 3: select the pivot>
<margin note line 4: partition the list>
<margin note line 7: sort first part>
<margin note line 8: sort second part>




```
1  def quickSort(list):
2      if len(list) > 1:
3          select a pivot
4          partition list into list1 and list2 such that
5              all elements in list1 <= pivot and
6              all elements in list2 > pivot
7          quickSort(list1)
8          quickSort(list2)
```

<margin note: how to partition>
Each partition places the pivot in the right place. The selection of the pivot affects the performance of the algorithm. Ideally, you should choose the pivot that divides the two parts evenly. For simplicity, assume the first element in the list is chosen as the pivot. Exercise 17.4 proposes an alternative strategy for selecting the pivot.

****animation icon: partition animation*
Figure 17.4 illustrates how to sort a list (5 2 9 3 8 4 0 1 6 7) using quick sort. Choose the first element 5 as the pivot. The list is partitioned into two parts, as shown in Figure 17.4b. The highlighted pivot is placed in the right place in the list. Apply quick sort on two partial lists (4 2 1 3 0) and then (8 9 6 7). The pivot 4 partitions (4 2 1 3 0) into just one partial list (0 2 1 3), as shown in Figure 17.4c. Apply quick sort on (0 2 1 3). The pivot 0 partitions it to just one partial list (2 1 3), as shown in Figure 17.4d. Apply quick sort on (2 1 3). The pivot 2 partitions it to (1) and (3), as shown in Figure 17.4e. Apply quick sort on (1). Since the list contains just one element, no further partition is needed.

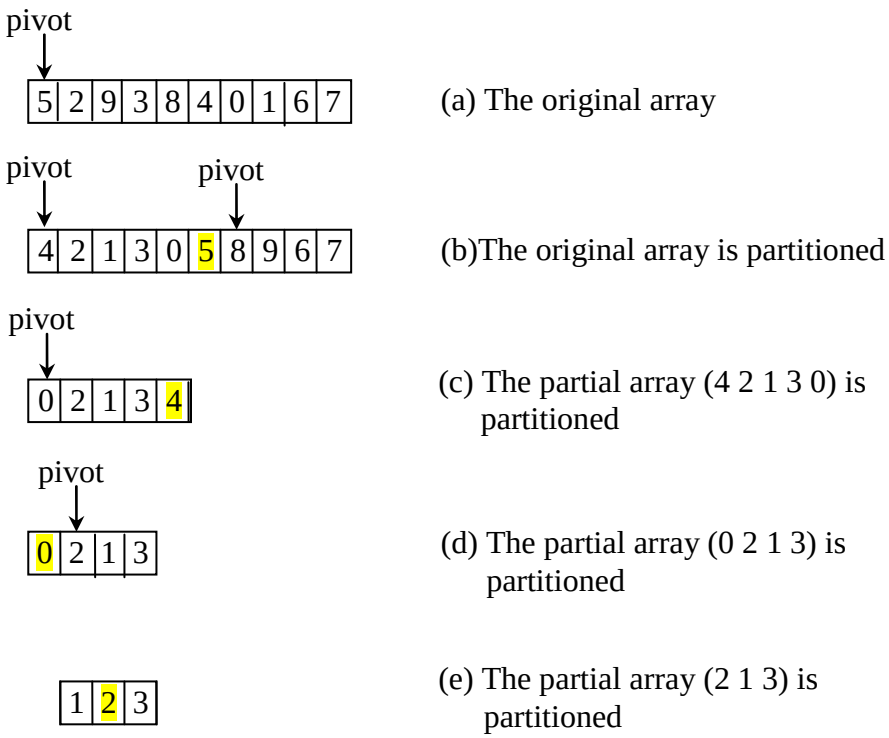


Figure 17.4

The quick sort algorithm is recursively applied to partial lists.

The quick sort algorithm is implemented in Listing 17.7. There are two overloaded quickSort functions in the class. The first function (line 2) is used to sort a list. The second is a helper function (line 6) that sorts a partial list with a specified range.

Listing 17.7 QuickSort.py

```
<margin note line 2: sort function>
<margin note: line 4: helper function>
<margin note: line 7: recursive call>
<margin note: line 8: recursive call>
<margin note line 18: forward>
<margin note line 22: backward>
<margin note line 27: swap>
<margin note line 35: place pivot>
<margin note line 36: pivot's new index>
<margin note line 38: pivot's original index>

1  def quickSort(list):
2      quickSortHelper(list, 0, len(list) - 1)
3
4  def quickSortHelper(list, first, last):
5      if last > first:
6          pivotIndex = partition(list, first, last)
7          quickSortHelper(list, first, pivotIndex - 1)
8          quickSortHelper(list, pivotIndex + 1, last)
9
10 # Partition list[first..last]
11 def partition(list, first, last):
12     pivot = list[first]  # Choose the first element as the pivot
13     low = first + 1  # Index for forward search
```

```
14      high = last  # Index for backward search
15
16      while high > low:
17          # Search forward from left
18          while low <= high and list[low] <= pivot:
19              low += 1
20
21          # Search backward from right
22          while low <= high and list[high] > pivot:
23              high -= 1
24
25          # Swap two elements in the list
26          if high > low:
27              list[high], list[low] = list[low], list[low]
28
29      while high > first and list[high] >= pivot:
30          high -= 1
31
32      # Swap pivot with list[high]
33      if pivot > list[high]:
34          list[first] = list[high]
35          list[high] = pivot
36          return high
37      else:
38          return first
39
40  # A test function
41  def main():
42      list = [2, 3, 2, 5, 6, 1, -2, 3, 14, 12]
43      quickSort(list)
44      for v in list:
45          print(str(v) + " ", end = "")
46
47  main()
```

<Output>
```
-2 1 2 2 3 3 5 6 12 14
```
<End Output>

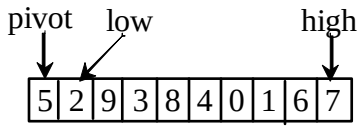
The `partition` function (lines 11–38) partitions `list[first..last]` using the pivot. The first element in the partial list is chosen as the pivot (line 12). Initially `low` points to the second element in the partial list (line 13) and `high` points to the last element in the partial list (line 14).

The function searches for the first element from left forward in the list that is greater than the pivot (lines 18–19), then search for the first element from right backward in the list that is less than or equal to the pivot (lines 22–23). Swap these two elements. Repeat the same search and swap operations until all the elements are searched in a while loop (lines 16–27).

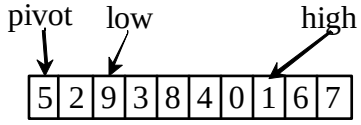
The function returns the new index for the pivot that divides the partial list into two parts if the pivot has been moved (line 36). Otherwise, return the original index for the pivot (line 38).

<margin note: partition illustration>
Figure 17.5 illustrates how to partition a list (5 2 9 3 8 4 0 1 6 7). Choose the first element 5 as the pivot. Initially `low` is the index that points to element 2 and `high` points to element 7, as shown in Figure 17.5a. Advance index `low` forward to search for the first element (9) that is greater than the pivot and move index `high` backward to search for the first element (1)

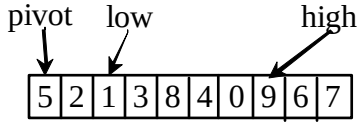
that is less than or equal to the pivot, as shown in Figure 17.5b. Swap 9 with 1, as shown in Figure 17.5c. Continue the search and move **low** to point to element 8 and **high** to point to element 0, as shown in Figure 17.5d. Swap element 8 with 0, as shown in Figure 17.5e. Continue to move **low** until it passes **high**, as shown in Figure 17.5f. Now all the elements are examined. Swap the pivot with element 4 at index **high**. The final partition is shown in Figure 17.5g. The index of the pivot is returned when the function is finished.



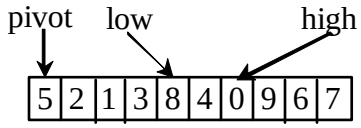
(a) Initialize pivot, low, and high



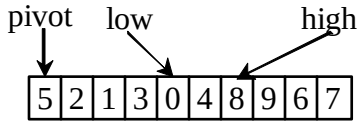
(b) Search forward and backward



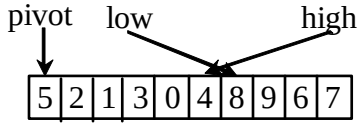
(c) 9 is swapped with 1



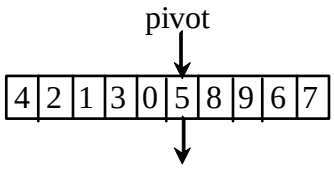
(d) Continue search



(e) 8 is swapped with 0



(f) When high < low, search is



(g) Pivot is in the right place

The index of the pivot is returned

Figure 17.5

The partition function returns the index of the pivot after it is put in the right place.

17.4.1 Quick Sort Time

<margin note: $O(n)$ partition time>

To partition a list of n elements, it takes n comparisons and n moves in the worst case. So, the time required for partition is $O(n)$.

<margin note: $O(n^2)$ worst-case time>

In the worst case, the pivot divides the list each time into one big sublist with the other empty. The size of the big sublist is one less than the one before divided. The algorithm requires $(n-1)+(n-2)+...+2+1=O(n^2)$ time.

<margin note: $O(n\log n)$ *best-case time>*

In the best case, the pivot divides the list each time into two parts of about the same size. Let $T(n)$ denote the time required for sorting a list of n elements using quick sort. So,

recursive quick sort on two sublists

partition time

$$T(n)=T(\frac{n}{2})+T(\frac{n}{2})+n.$$

Similar to the merge sort analysis, $T(n)=O(n\log n)$.

<margin note: $O(n\log n)$ *average-case time>*

On the average, each time the pivot will not divide the list into two parts of the same size or one empty part. Statistically, the sizes of the two parts are very close. So the average time is $O(n\log n)$. The exact average-case analysis is beyond the scope of this book.

<margin note: quick sort vs. merge sort>

Both merge sort and quick sort employ the divide-and-conquer approach. For merge sort, the bulk of work is to merge two sublists, which takes place *after* the sublists are sorted. For quick sort, the bulk of work is to partition the list into two sublists, which takes place *before* the sublists are sorted. Merge sort is more efficient than quick sort in the worst case, but the two are equally efficient in the average case. Merge sort requires a temporary list for sorting two sublists. Quick sort does not need additional list space. So, quick sort is more space efficient than merge sort.

<check point>

17.1

Use Figure 17.1 as an example to show how to apply bubble sort on [45, 11, 50, 59, 60, 2, 4, 7, 10].

17.2

Use Figure 17.2 as an example to show how to apply merge sort on [45, 11, 50, 59, 60, 2, 4, 7, 10].

17.3

Use Figure 17.4 as an example to show how to apply quick sort on [45, 11, 50, 59, 60, 2, 4, 7, 10].

17.4

What is wrong if lines 4-9 in Listing 17.5 MergeSort.py are replaced by the following code?

```
    firstHalf = list[ : len(list) // 2 + 1]
    mergeSort(firstHalf)

    # Merge sort the second half
```

```
secondHalf = list[len(list) // 2 + 1 : ]
mergeSort(secondHalf)
```

or replaced by the following code?

```
firstHalf = list[ : len(list) // 2]
mergeSort(firstHalf)

# Merge sort the second half
secondHalf = list[len(list) // 2 + 1 : ]
mergeSort(secondHalf)
```

<end check point>

17.5 Heap Sort

<key point>
Heap sort uses a binary heap. It first adds all elements to a heap and then removes the largest elements successively to obtain a sorted list.
<end key point>

<margin note: root>
<margin note: left subtree>
<margin note: right subtree>
<margin note: length>
<margin note: depth>
Heap sort uses a binary heap, which is a complete binary tree. A binary tree is a hierarchical structure. It either is empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*. The *length* of a path is the number of the edges in the path. The *depth* of a node is the length of the path from the root to the node.

A *heap* is a binary tree with the following properties:

- It is a complete binary tree.
- Each node is greater than or equal to any of its children.

<margin note: complete binary tree>
A binary tree is *complete* if each of its levels is full, except that the last level may not be full and all the leaves on the last level are placed leftmost. For example, in Figure 17.6, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap, because the root (39) is less than its right child (42).

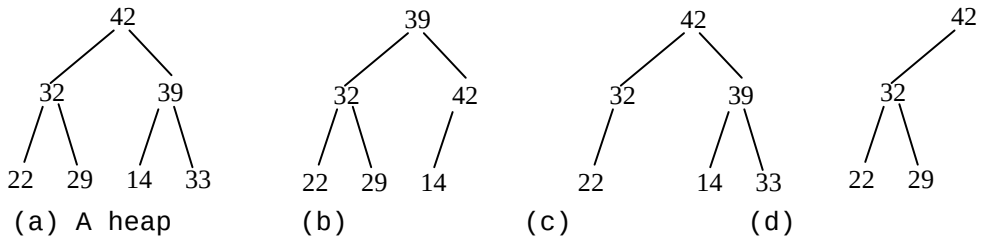


Figure 17.6
A heap is a special complete binary tree.

Pedagogical NOTE
<side remark: heap animation>

A heap can be implemented efficiently for inserting keys and for deleting the root. Follow the link www.cs.armstrong.edu/liang/animation/HeapAnimation.html to see how a heap works, as shown in Figure 17.7.

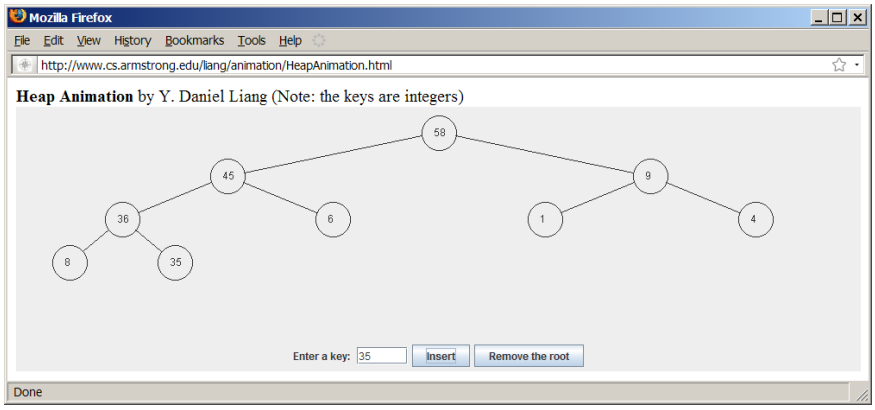
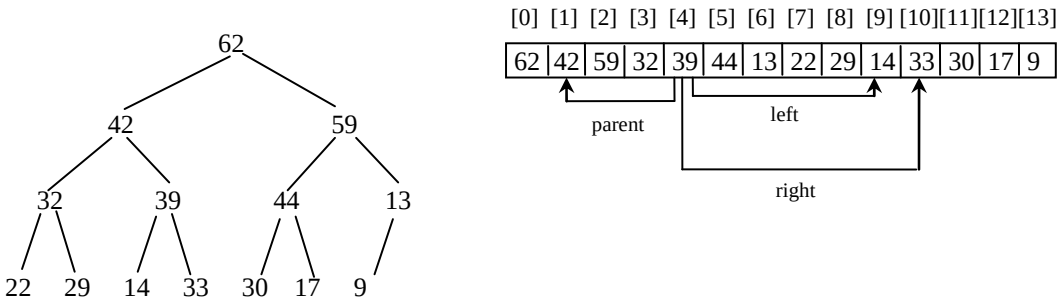


Figure 17.7
The animation tool enables you to insert a key and delete the root visually.

End NOTE

17.5.1 Storing a Heap

A heap can be stored in a list. The heap in Figure 17.8a can be stored using the list in Figure 17.8b. The root is at position 0, and its two children are at positions 1 and 2. For a node at position i, its left child is at position $2i+1$, its right child is at position $2i+2$, and its parent is $(i-1)/2$. For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 ($2\times4+1$), its right child (element 33) is at 10 ($2\times4+2$), and its parent (element 42) is at 1 ($(4-1)/2$).



(a) A heap (b) A heap stored in a list

Figure 17.8
A binary heap can be implemented using a list.

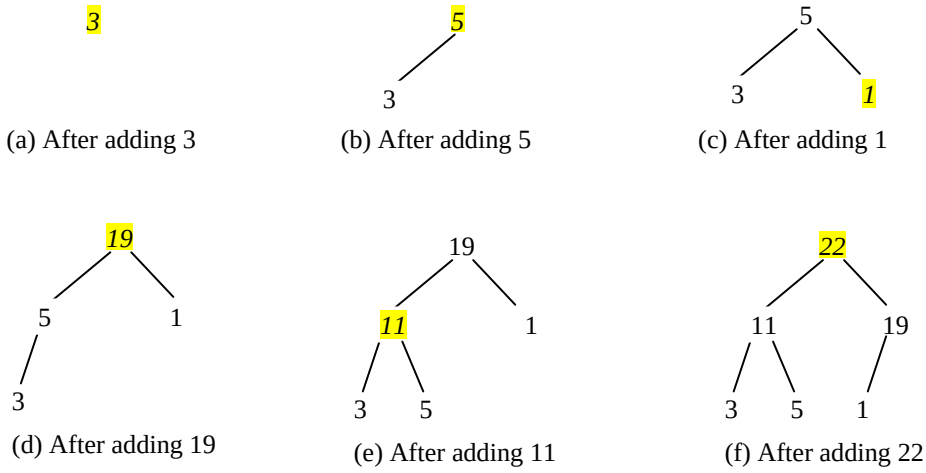
25.5.2 Adding a New Node

To add a new node to the heap, first add it to the end of the heap and then rebuild the tree as follows:

```
Let the last node be the current node
while the current node is greater than its parent:
    Swap the current node with its parent
```

```
        Now the current node is one level up
```

Suppose the heap is initially empty. The heap is shown in Figure 17.9, after adding numbers 3, 5, 1, 19, 11, and 22 in this order.

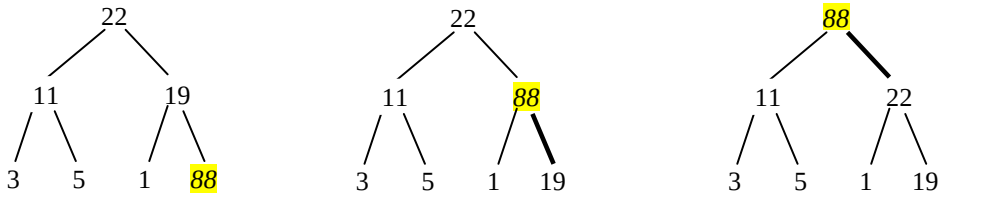


(a) After adding 3 (b) After adding 5 (c) After adding 1

(d) After adding 19 (e) After adding 11 (f) After adding 22

Figure 17.9
Elements 3, 5, 1, 19, 11, and 22 are inserted into the heap.

Now consider adding 88 into the heap. Place the new node 88 at the end of the tree, as shown in Figure 17.10a. Swap 88 with 19, as shown in Figure 17.10b. Swap 88 with 22, as shown in Figure 17.10c.



(a) Add 88 to a heap (b) After swapping 88 with 19 (c) After swapping 88 with 22

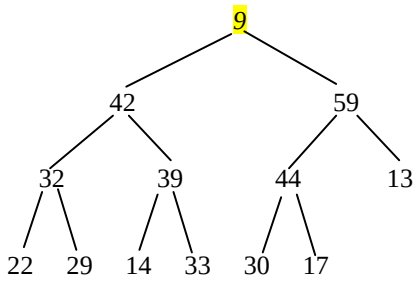
Figure 17.10
Rebuild the heap after adding a new node.

17.5.3 Removing the Root

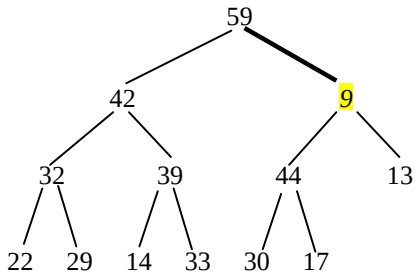
Often you need to remove the max element, which is the root in a heap. After the root is removed, the tree must be rebuilt to maintain the heap property. The algorithm for rebuilding the tree can be described as follows:

```
Move the last node to replace the root
Let the root be the current node
while (the current node has children and the current node is
       smaller than one of its children):
    Swap the current node with the larger of its children
    Now the current node is one level down
```

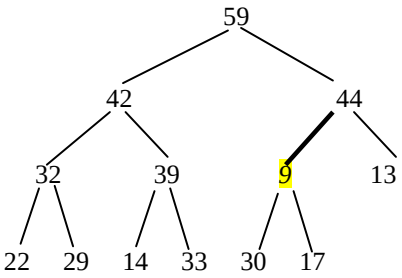
Figure 17.11 shows the process of rebuilding a heap after the root 62 is removed from Figure 17.8a. Move the last node 9 to the root as shown in Figure 17.11a. Swap 9 with 59 as shown in Figure 17.11b. Swap 9 with 44 as shown in Figure 17.11c. Swap 9 with 30 as shown in Figure 17.11d.



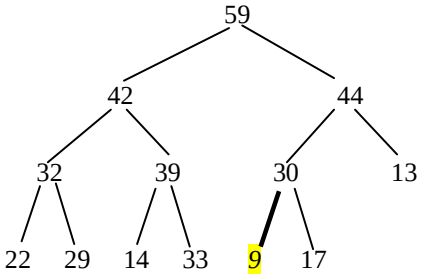
(a) After moving 9 to the root



(b) After swapping 9 with 59



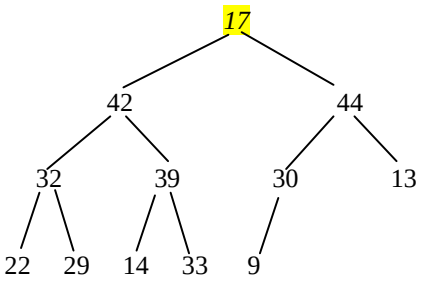
(c) After swapping 9 with 44



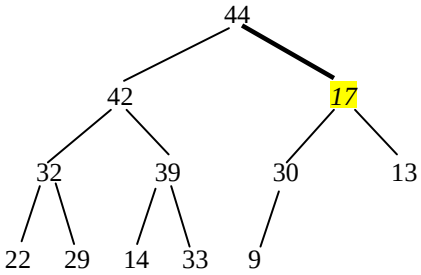
(d) After swapping 9 with 30

Figure 17.11
Rebuild the heap after the root 62 is removed.

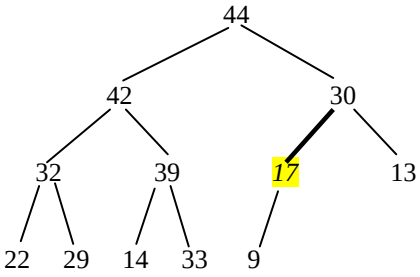
Figure 17.12 shows the process of rebuilding a heap after the root 59 is removed from Figure 17.11d. Move the last node 17 to the root, as shown in Figure 17.12a. Swap 17 with 44 as shown in Figure 17.12b. Swap 17 with 30 as shown in Figure 17.12c.



(a) After moving 17 to the root



(b) After swapping 17 with 44



(c) After swapping17 with 30

Figure 17.12
Rebuild the heap after the root 59 is removed.

17.5.4 The Heap Class

Now you are ready to design and implement the Heap class. The class diagram is shown in Figure 17.a. Its implementation is given in Listing 17.8.

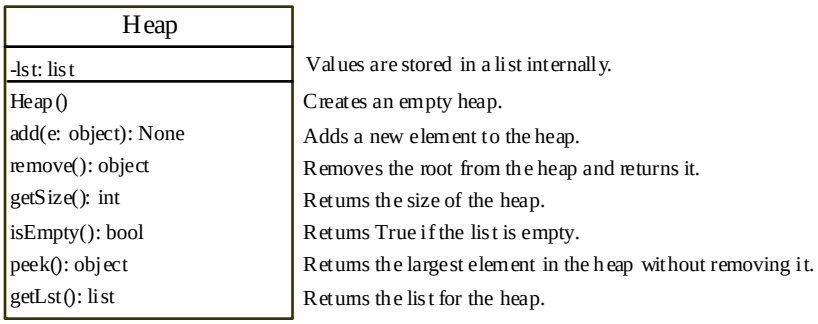


Figure 17.13
Heap provides the operations for manipulating a heap.

Listing 17.8 Heap.py

```
<margin note line 2: constructor>
<margin note line 3: internal heap representation>
<margin note line 6: add a new element>
<margin note line 7: append the object>
<margin note line 15: swap with parent>
<margin note line 18: heap now>
<margin note line 23: remove the root>
<margin note line 25: empty heap>
<margin note line 27: root>
<margin note line 28: new root>
<margin note line 29: remove the last>
<margin note line 32: adjust the tree>
<margin note line 32: compare two children>
<margin note line 45: swap with the larger child>
<margin note line 55: getSize>
```

```
1  class Heap:
 2      def __init__(self):
 3          self.__lst = []
 4
 5      # Add a new item into the lst
 6      def add(self, e):
 7          self.__lst.append(e)  # Append to the lst
 8          # The index of the last node
 9          currentIndex = len(self.__lst) - 1
10
11          while currentIndex > 0:
12              parentIndex = (currentIndex - 1) // 2
13              # Swap if the current item is greater than its parent
14              if self.__lst[currentIndex] > self.__lst[parentIndex]:
15                 self.__lst[currentIndex], self.__lst[parentIndex] = \
16                     self.__lst[parentIndex], self.__lst[currentIndex]
17              else:
18                  break  # The tree is a lst now
19
20              currentIndex = parentIndex
21
22      # Remove the root from the lst
```

```
23      def remove(self):
24          if len(self.__lst) == 0:
25             return None
26
27          removedItem = self.__lst[0]
28          self.__lst[0] = self.__lst[len(self.__lst) - 1]
29          self.__lst.pop(len(self.__lst) - 1) # Remove the last item
30
31          currentIndex = 0
32          while currentIndex < len(self.__lst):
33              leftChildIndex = 2 * currentIndex + 1
34              rightChildIndex = 2 * currentIndex + 2
35
36              # Find the maximum between two children
37              if leftChildIndex >= len(self.__lst):
38                  break  # The tree is a lst
39              maxIndex = leftChildIndex
40              if rightChildIndex < len(self.__lst):
41                  if self.__lst[maxIndex] < self.__lst[rightChildIndex]:
42                      maxIndex = rightChildIndex
43
44              # Swap if the current node is less than the maximum
45              if self.__lst[currentIndex] < self.__lst[maxIndex]:
46                  self.__lst[maxIndex], self.__lst[currentIndex] = \
47                      self.__lst[currentIndex], self.__lst[maxIndex]
48                  currentIndex = maxIndex
49              else:
50                  break  # The tree is a lst
51
52          return removedItem
53
54      # Returns the size of the heap
55      def getSize(self):
56          return len(self.__lst)
57
58      # Returns True if the heap is empty
59      def isEmpty(self):
60          return self.getSize() == 0
61
62      # Returns the largest element in the heap without removing it
63      def peek(self):
64          return self.__lst[0]
65
66      # Returns the list in the heap
67      def getLst(self):
68          return self.__lst
```

A heap is represented using a list internally (line 3). You may change it to other data structures, but the `Heap` class contract will remain unchanged.

The `add(e)` method (lines 6-20) appends the element to the tree and then swaps it with its parent if it is greater than its parent. This process continues until the new object becomes the root or is not greater than its parent.

The `remove()` method (lines 23-52) removes and returns the root. To maintain the heap property, the method moves the last object to the root position and swaps it with its larger child if it is less than the larger child. This process continues until the last object becomes a leaf or is not less than its children.

17.5.5 Sorting Using the Heap Class

To sort a list using a heap, first create an object using the **Heap** class, add all the elements to the heap using the **add** method, and remove all the elements from the heap using the **remove** method. The elements are removed in descending order. Listing 17.9 gives an algorithm for sorting a list using a heap.

Listing 17.9 HeapSort.py

<margin note line 5: create a Heap>
<margin note: line 9: add element>
<margin note: line 13: remove element>
<margin note line 17: invoke sort method>

```
 1  from Heap import Heap
 2
 3  def heapSort(list):
 4      heap = Heap() # Create a Heap
 5
 6      # Add elements to the heap
 7      for v in list:
 8          heap.add(v)
 9
10      # Remove elements from the heap
11      for i in range(len(list)):
12          list[len(list) - 1 - i] = heap.remove()
13
14  def main():
15      list = [-44, -5, -3, 3, 3, 1, -4, 0, 1, 2, 4, 5, 53]
16      heapSort(list)
17      for v in list:
18          print(str(v) + " ", end = " ")
19
20  main()
```

<Output>

```
-44 -5 -4 -3 0 1 1 2 3 3 4 5 53
```

<End Output>

17.5.6 Heap Sort Time Complexity

<margin note: height of a heap>

Let us turn our attention to analyzing the time complexity for the heap sort. Let h denote the height for a heap of n elements. Since a heap is a complete binary tree, the first level has 1 node, the second level has 2 nodes, the kth level has 2^{k-1} nodes, the $(h-1)$th level has 2^{h-2} nodes, and the hth level has at least 1 and at most 2^{h-1} nodes. Therefore,

$$1+2+...+2^{h-2} < n \le 1+2+...+2^{h-2}+2^{h-1}$$

i.e.,

$$2^{h-1}-1 < n \le 2^h - 1$$
$$2^{h-1} < n+1 \le 2^h$$
$$h-1 < \log(n+1) \le h$$

Thus, $h < \log(n+1)+1$ and $\log(n+1) \le h$. Therefore, $\log(n+1) \le h < \log(n+1)+1$. Hence, the height of the heap is $O(\log n)$.

<margin note: $O(n \log n)$ *worst-case time>*

Since the **add** function traces a path from a leaf to a root, it takes at most h steps to add a new element to the heap. So, the total time for constructing an initial heap is $O(n\log n)$ for a list of n elements. Since the **remove** function traces a path from a root to a leaf, it takes at most h steps to rebuild a heap after removing the root from the heap. Since the **remove** function is invoked n times, the total time for producing a sorted list from a heap is $O(n\log n)$.

<margin note: heap sort vs. merge sort>
Both merge sort and heap sort requires $O(n\log n)$ time. Merge sort requires a temporary list for merging two sublists. Heap sort does not need additional list space. So, heap sort is more space efficient than merge sort.

<check point>
Section 17.5
17.5
What is a complete binary tree? What is a heap? Describe how to remove the root from a heap and how to add a new item to a heap.

17.6
What is the return value from invoking the **remove** function if the heap is empty?

17.7
Add the elements **4**, **5**, **1**, **2**, **9**, **3** into a heap in this order. Draw the diagrams to show the heap after each element is added.

17.7
Show the heap after the root in the heap in Figure 17.12c is removed.

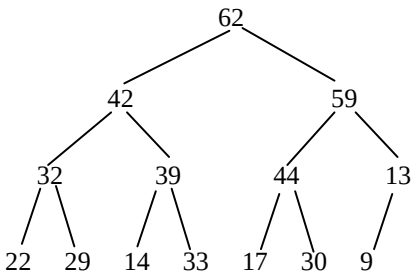
17.8
What is the time complexity of inserting a new element into a heap and what is the time complexity of deleting an element from a heap?

17.9

Show the steps of creating a heap using [45, 11, 50, 59, 60, 2, 4, 7, 10].

17.10

Given the following heap, show the steps of removing all nodes from the heap.



<end check point>

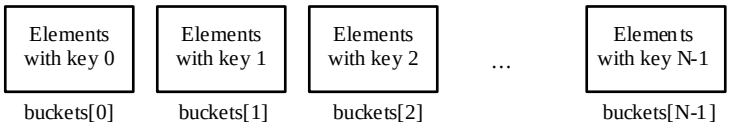
17.6 Bucket Sort and Radix Sort
<key point>

Bucket sort and radix sort are efficient for sorting integers.
<end key point>

All sort algorithms discussed so far are general sorting algorithms that work for any types of keys (e.g., integers, strings, and any comparable items). These algorithms sort the elements by comparing their keys. The lower bound for general sorting algorithms is $O(n \log n)$. So, no sorting algorithms based on comparisons can perform better than $O(n \log n)$. However, if the keys are small integers, you can use bucket sort without having to compare the keys.

The bucket sort algorithm works as follows. Assume the keys are in the range from 0 to N-1. We need N buckets labeled 0, 1, ..., and N-1. If an element's key is i, the element is put into the bucket i. Each bucket holds the elements with the same key value.

<margin note: buckets>



You can use a list to implement a bucket. The bucket sort algorithm for sorting a list of elements can be described as follows:

```
def bucketSort(list):
    buckets = N * [0]

    # Distribute the elements from list to buckets
    for i in range(len(list)):
        key = list[i].getKey()

        if buckets[key] is empty:
            buckets[key] = []

        buckets[key].append(list[i])

    # Now move the elements from the buckets back to list
    k = 0 # k is an index for list
    for i in range(len(buckets)):
        if buckets[i] is not empty:
            for j in range(len(buckets[i])):
                list[k] = buckets[i][j]
                k += 1
```

Clearly, it takes $O(n+N)$ time to sort the list and uses $O(n+N)$ space, where n is the list size.

Note that if N is too large, bucket sort is not desirable. You can use radix sort. Radix sort is based on bucket sort, but it uses only ten buckets.

<margin note: stable>
It is worthwhile to note that bucket sort is *stable*, meaning that if two elements in the original list have the same key value, their order is not changed in the sorted list. That is, if element e_1 and element e_2 have the same key and e_1 precedes e_2 in the original list, e_1 still precedes e_2 in the sorted list.

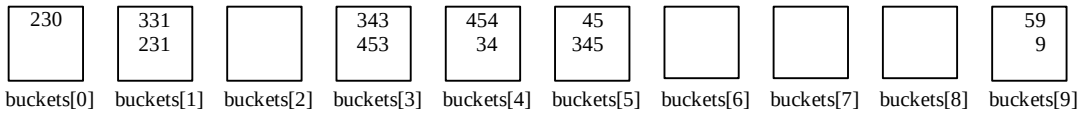
Again assume that the keys are positive integers. The idea for the radix sort is to divide the keys into subgroups based on their radix positions. It applies bucket sort repeatedly for the key values on radix positions, starting from the least-significant position.

Consider sorting the elements with the keys:

331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

Apply the bucket sort on the last radix position. The elements are put into the buckets as follows:

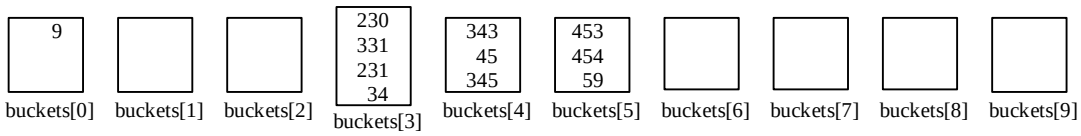
****margin note: see Supplement VI for radix sort animation*
<margin note: queue>



After being removed from the buckets, the elements are in the following order:

230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9

Apply the bucket sort on the second-to-last radix position. The elements are put into the buckets as follows:
<margin note: queue>

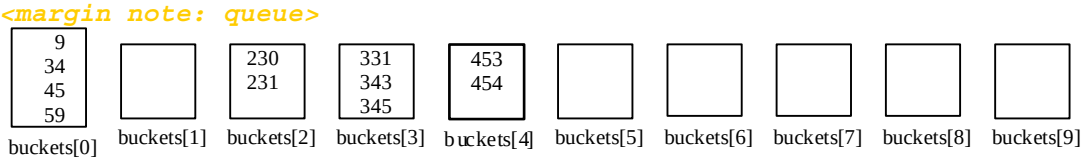


After being removed from the buckets, the elements are in the following order:

9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59

(Note that 9 is 009.)

Apply the bucket sort on the third-to-last radix position. The elements are put into the buckets as follows:
<margin note: queue>



After being removed from the buckets, the elements are in the following order:

9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454

The elements are now sorted.

In general, radix sort takes $O(dn)$ time to sort n elements with integer keys, where d is the maximum number of the radix positions among all keys.

<check point>

17.12
Can you sort a list of strings using bucket sort?

17.13
Show how radix sort works using the numbers 454, 34, 23, 43, 74, 86, and 76.
<end check point>

Key Terms

- **bubble sort**
- **bucket sort**
- **heap**
- **heap sort**
- **merge sort**
- **quick sort**
- **radix sort**

Chapter Summary

1. The worst-case complexity for selection sort, insertion sort, bubble sort, and quick sort is $O(n^2)$.
2. The average-case and worst-case complexity for merge sort is $O(n\log n)$. The average time for quick sort is also $O(n\log n)$.
3. Heaps are a useful data structure for designing efficient algorithms such as sorting. You learned how to define and implement a heap class, and how to insert and delete elements to/from a heap.
4. The time complexity for heap sort is $O(n\log n)$.
5. Bucket sort and radix sort are specialized sorting algorithms for integer keys. These algorithms sort keys using buckets rather than comparing keys. They are more efficient than general sorting algorithms.

Multiple-Choice Questions

See multiple-choice questions online at liang.armstrong.edu/selftest/selftestpy?chapter=17.

Programming Exercises

17.1
(*Improving quick sort*) The quick sort algorithm presented in the book selects the first element in the list as the pivot. Revise it by selecting the median among the first, middle, and last elements in the list.

17.2
(*Checking order*) Write the following functions that check whether a list is ordered in ascending order or descending order. By default, the function checks ascending order. To check descending order, pass `descending` to the `ord` argument in the function.

```
def ordered(list, ord = "ascending"):
```

Write a test program that prompts the user to enter a list of integers and checks if the list is in ascending order.

17.3
(*Min-heap*) The heap presented in the text is also known as a *max-heap*, in which each node is greater than or equal to any of its children. A

min-heap is a heap in which each node is less than or equal to any of its children. Revise the heap sort program to use a min-heap. Write a test program that prompts the user to enter a list of integers and uses the min-heap to sort the integers.

17.4*
(*Radix sort*) Write a program that randomly generates 1000000 integers and sorts them using radix sort.

17.5*
(*Execution time for sorting*) Write a program that obtains the execution time of selection sort, bubble sort, merge sort, quick sort, heap sort, and radix sort for input size 50000, 100,000, 150,000, 200,000, 250,000, and 300,000. Your program should create data randomly and print a table like this:

Array size	Selection Sort	Bubble Sort	Merge Sort	Quick Sort	Heap Sort	Radix Sort
50000						
100000						
150000						
200000						
250000						
300000						

The text gives a recursive quick sort. Write a nonrecursive version in this exercise.

Comprehensive

17.6**
(*Turtle: selection sort animation*) Write a program that animates the selection sort algorithm. Create a list that consists of 20 distinct numbers from 1 to 20 in a random order. The list elements are displayed in a histogram, as shown in Figure 17.13. If two elements are swapped, redisplay them in the histogram.

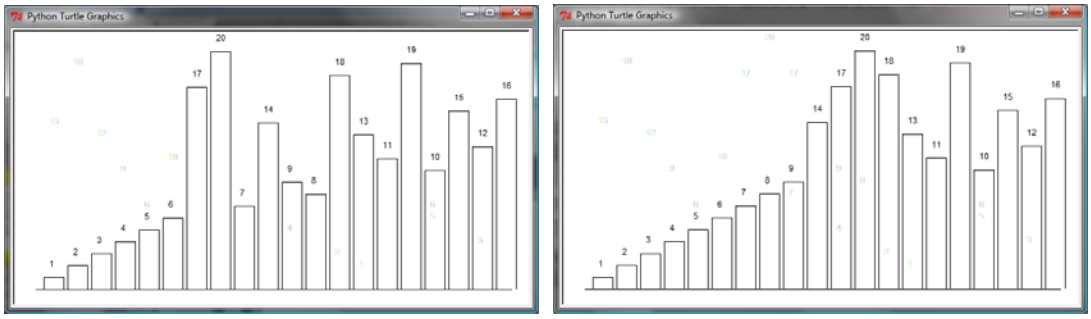


Figure 17.13

The program animates selection sort.

17.7**
(*Turtle: insertion sort animation*) Write a program that animates the insertion sort algorithm. Create a list that consists of 20 distinct

numbers from 1 to 20 in a random order. The list elements are displayed in a histogram, as shown in Figure 17.17. If two elements are swapped, redisplay them in the histogram.

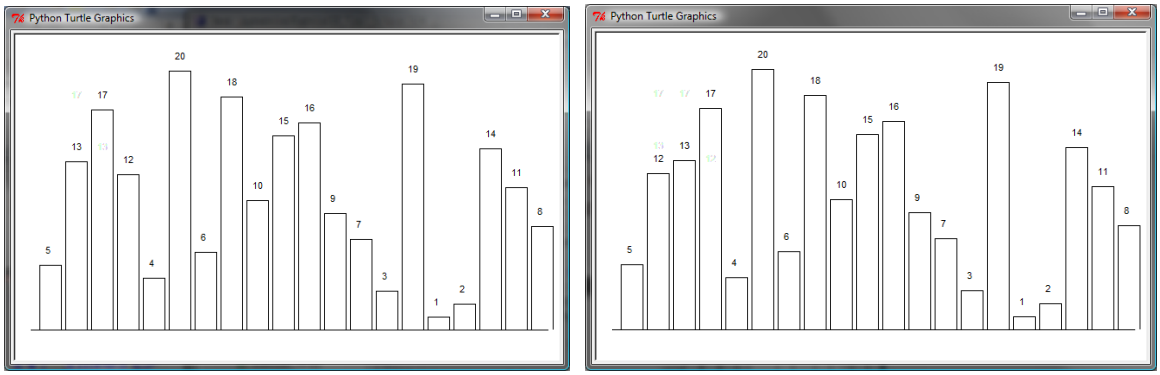


Figure 17.14

The program animates insertion sort.

17.8*
(*Turtle: bubble sort animation*) Write a program that animates the bubble sort algorithm. Create a list that consists of 20 distinct numbers from 1 to 20 in a random order. The list elements are displayed in a histogram, as shown in Figure 17.15. If two elements are swapped, redisplay them in the histogram.

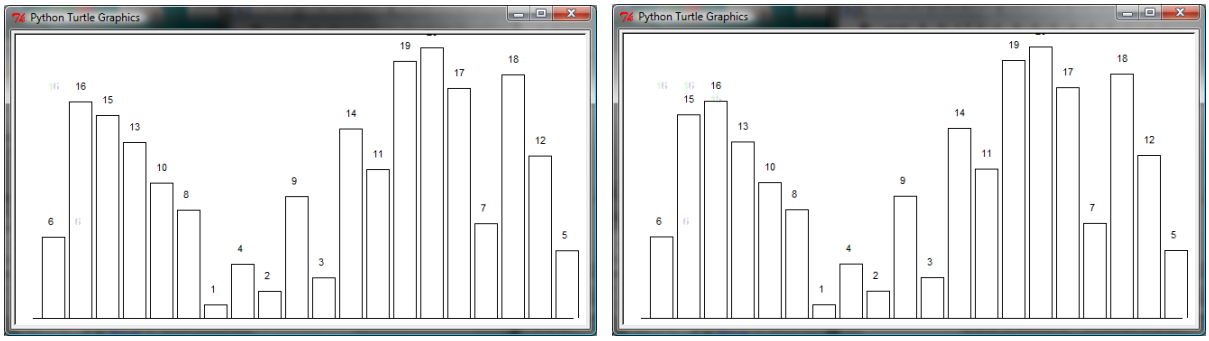


Figure 17.15

The program animates bubble sort.

17.9*
(*Radix sort animation*) Write a program that animates the radix sort algorithm. Create a list that consists of 20 random numbers from 0 to 1000. The list elements are displayed, as shown in Figure 17.17. Clicking the *Step* button causes the program to place a number to a bucket. The number that has just been placed is displayed in red. Once all numbers are placed in the buckets, clicking the *Step* button collects all the numbers from the buckets back to the list. When the algorithm is finished, clicking the *Step* button displays a dialog box to inform the user. Clicking the *Reset* button creates a new random list for a new start.

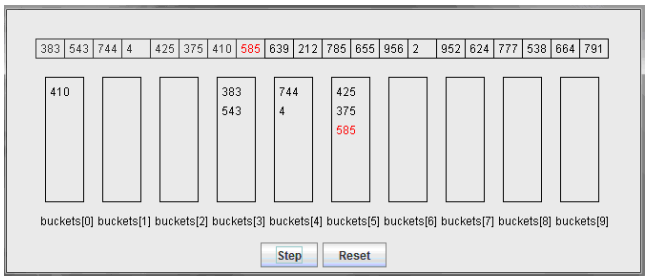
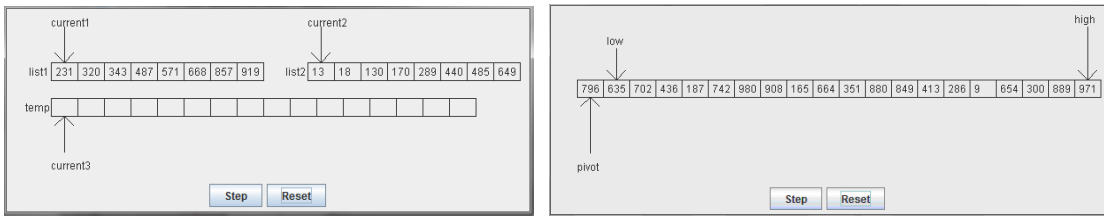


Figure 17.16

The program animates radix sort.

17.10*
(*Merge animation*) Write a program that animates the merge of two sorted lists. Create two lists **list1** and **list2**, each consists of 8 random numbers from 1 to 999. The list elements are displayed, as shown in Figure 17.17a. Clicking the *Step* button causes the program to move an element from **list1** or **list2** to **temp**. Clicking the *Reset* button creates two new random lists for a new start. When the algorithm is finished, clicking the *Step* button displays a dialog box to inform the user.



(a) *(b)*

Figure 17.17

(a) The program animates merge of two sorted lists. (b) The program animates partition for quick sort.

17.11*
(*Quick sort partition animation*) Write a program that animates the partition for a quick. The program creates a list that consists of 20 random numbers from 1 to 999. The list is displayed, as shown in Figure 17.17b. Clicking the *Step* button causes the program to move **low** to the right or **high** to left, or swap the elements at **low** or **high**. Clicking the *Reset* button creates a new list of random numbers for a new start. When the algorithm is finished, clicking the *Step* button displays a dialog box to inform the user.