Sorting and Searching

Objectives

- To be able to explain and implement sequential search and binary search.
- To be able to explain and implement selection sort, bubble sort, merge sort, quick sort, insertion sort, and shell sort.
- To understand the idea of hashing as a search technique.
- To introduce the map abstract data type.
- To implement the map abstract data type using hashing.

Searching

- Searching is the algorithmic process of finding a particular item in a collection of items.
- A search typically answers either True or False as to whether the item is present.
- In Python, there is a very easy way to ask whether an item is in a list of items. We use the **in** operator.
- **Searching Algorithm:**
  - Sequential Search
  - Binary Search
  - Hashing

```
>>> 15 in [3,5,2,4,1]
False
>>> 3 in [3,5,2,4,1]
True
>>>
```

Even though this is easy to write,
However, there are many different ways to
search for the item.

Searching

Sequential Search: Unordered List

- Starting at the first item in the list,
- Move from item to item, following the underlying sequential ordering until we either:
    - find what we are looking for or
    - run out of items.
        - If we run out of items, we have discovered that the item we were searching for was not present.

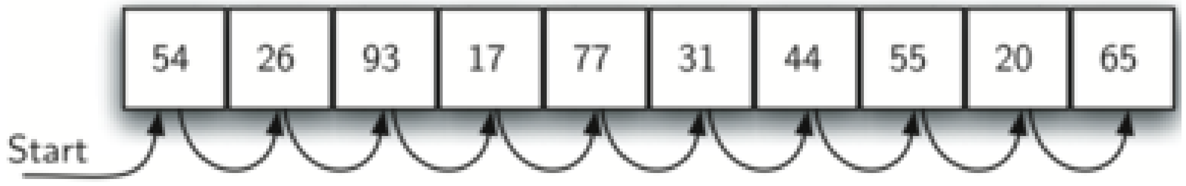


Figure 5.1: The Sequential Search of a List of Integers

Searching

Sequential Search: Unordered List (Cont.)

Implementation

```
def sequential_search(a_list, item):
    pos = 0
    found = False
    while pos < len(a_list) and not found:
        if a_list[pos] == item:
            found = True
        else:
            pos = pos+1
    return found

test_list = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequential_search(test_list, 3))      # False
print(sequential_search(test_list, 13))     # True
```

- The function needs the list and the item we are looking for
- and returns a boolean value as to whether it is present.
- The boolean variable found is initialized to False
- and is assigned the value True if we discover the item in the list.

Searching

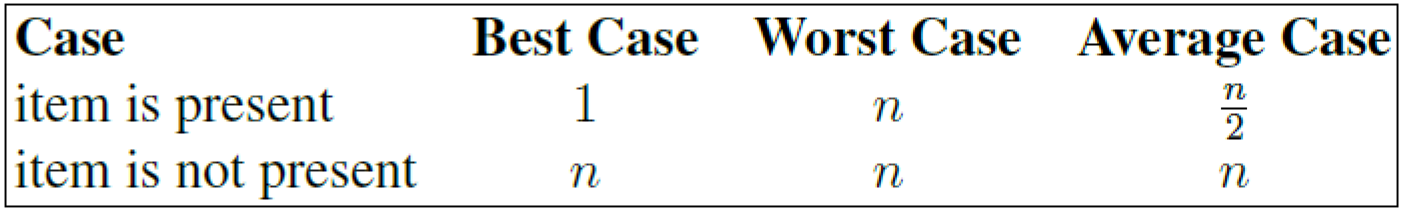
Analysis of Sequential Search for Unordered List

- If the item is **not in the list**,
  - The only way to know, it is to compare it against every item present.
  - If there are n items, then the sequential search requires n comparisons to discover that the item is not there.
- If the item is **in the list**: there are actually three different scenarios that can occur.
  - In the best case: we will find the item in the beginning of the list.
    - We will need **only one comparison**.
  - In the worst case: we will discover the item in the nth comparison.

Searching

Analysis of Sequential Search for Unordered List (Cont.)

- If the item is **not in the list,** we will compare n items
- If the item is **in the list,** we will compare n**/2** items in average**.**
- However, n is the large number, the coefficients is not significant in approximation, so the complexity of the sequential search, is $O(n)$.



Case	**Best Case**	**Worst Case**	**Average Case**
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

Table 5.1: Comparisons Used in a Sequential Search of an Unordered List

Searching

Sequential Search: Implementation (for Ordered List)

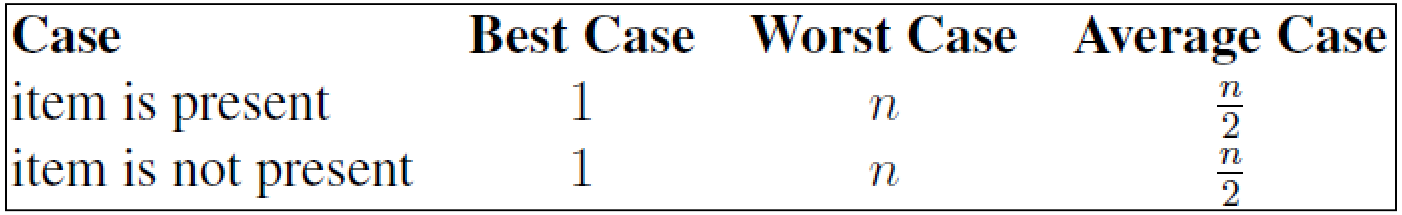
```
def ordered_sequential_search(a_list, item):
    pos = 0
    found = False
    stop = False
    while pos < len(a_list) and not found and not
stop:
        if a_list[pos] == item:
            found = True
        else:
            if a_list[pos] > item:
                stop = True
            else:
                pos = pos+1
    return found

test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(ordered_sequential_search(test_list, 3))
print(ordered_sequential_search(test_list, 13))
```

- If the item we are looking for is present in the list,
  - The chance of it being in any one of the n positions is still the same as before
  - We will still have the same number of comparisons to find the item as Unordered List.
- If the item is not present there is a slight advantage,
  - The algorithm can stop immediately after the item compared to the searched item is greater than the searched item.

Searching

Analysis of Sequential Search for Ordered List

- If the item is **not in the list,** we will compare n items
  - In the best case, we might discover that the item is not in the list by looking at **only one item**
  - In the worst case, we might discover that the item is not in the list by looking **at the nth item**
  - **On average**, we will know after looking through only n/2 items
- If the item is **in the list**: there are actually three different scenarios that can occur.
  - In the best case: we will find the item in the beginning of the list.
  - In the worst case: we will discover the item in the nth comparison.



Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	1	n	$\frac{n}{2}$

The complexity is $O(n)$

Table 5.2: Comparisons Used in Sequential Search of an Ordered List

Searching

Binary Search

1) Start by examining the middle item. If that item is the one we are searching for, we are done.
2) If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items:
   - If the item we are searching for is greater than the middle item, the entire lower half of the list as well as the middle item can be eliminated from further consideration.
   - If the item we are searching for is less than the middle item, the entire upper half of the list as well as the middle item can be eliminated from further consideration.
3) Repeat step 1 to 2 until either the searched item is found or not found

Searching

Binary Search: Implementation

Using while loop

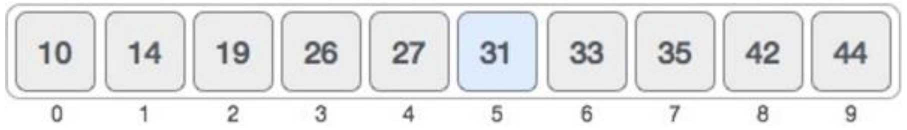
```
def binary_search(a_list, item):
    first = 0
    last = len(a_list) - 1
    found = False
    while first <= last and not found:
        midpoint = (first + last) // 2
        if a_list[midpoint] == item:
            found = True
        else:
            if item < a_list[midpoint]:
                last = midpoint - 1
            else:
                first = midpoint + 1
    return found

test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(test_list, 3))
print(binary_search(test_list, 13))
```

Using Recursion

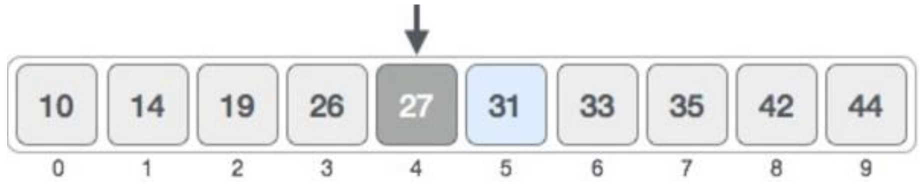
```
def binary_search(a_list, item):
    if len(a_list) == 0:
        return False
    else:
        midpoint = len(a_list) // 2
        if a_list[midpoint] == item:
            return True
        else:
            if item < a_list[midpoint]:
                return 
binary_search(a_list[:midpoint], item)
            else:
                return 
binary_search(a_list[midpoint + 1:], item)

test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(test_list, 3))
print(binary_search(test_list, 13))
```

Searching

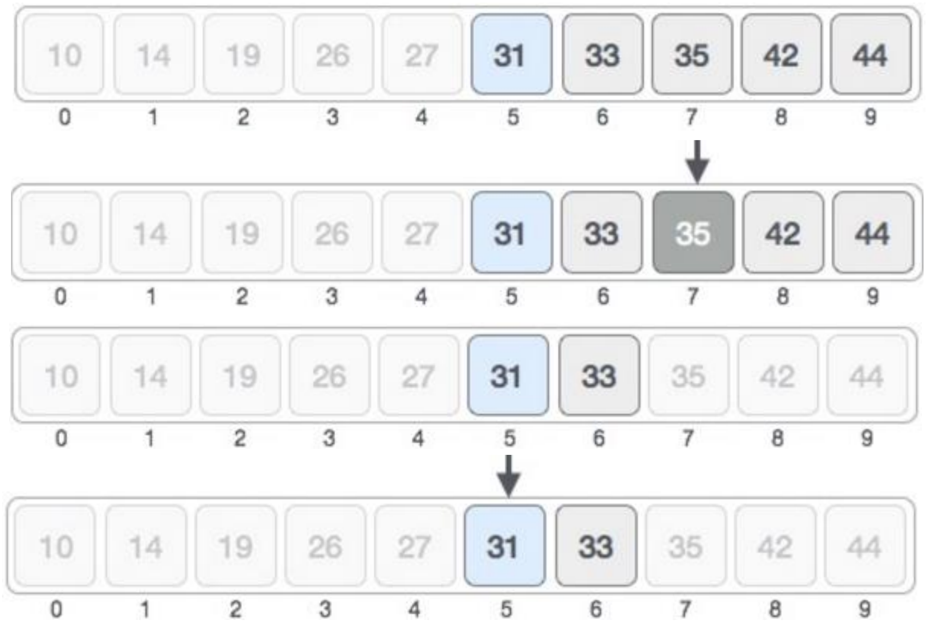
Binary Search: Step by Step (Ex. searching for 31)



mid = low + (high - low) / 2 = 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So 4 is the mid of the list.



27 < 31 → the entire lower half of the list are eliminated



http://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm

Searching

Binary Search: Analysis of Algorithm

- Each comparison eliminates about half of the remaining items from consideration.
- If we start with n items, about $n/2$ items will be left after the first comparison.
- After the second comparison, there will be about $n/4$.
- Then $n/8$, $n/16$, and so on.
- When we split the list enough times, we end up with a list that has just one item. Either that is the item we are looking for or it is not. Either way, we are done.
- The number of comparisons necessary to get to this point is i where $n/2^i = 1$. Solving for i gives us $i = \log n$.
- The maximum number of comparisons is logarithmic with respect to the number of items in the list. Therefore, the binary search is $O(\log n)$.

Searching

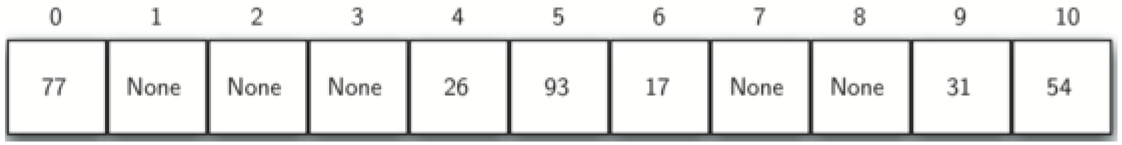
Binary Search: Analysis of Algorithm (Cont.)

- Even though a binary search is generally better than a sequential search, it is important to note that for small values of n, the additional cost of sorting is probably not worth it.
- In fact, we should always consider whether it is cost effective to take on the extra work of sorting to gain searching benefits.
  - If we can sort once and then search many times, the cost of the sort is not so significant.
  - However, for large lists, sorting even once can be so expensive that simply performing a sequential search from the start may be the best choice.

Searching

Hashing

- A data structure that can be searched in $O(1)$ time is **Hashing**.
- The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and $m - 1$, where m is number of slots.
- Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31.
  - Our first hash function, sometimes referred to as the "remainder method," simply takes an item and divides it by the table size, returning the remainder as its hash value (h(item) = item%11).
- Once the hash values have been computed, we can insert each item into



0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

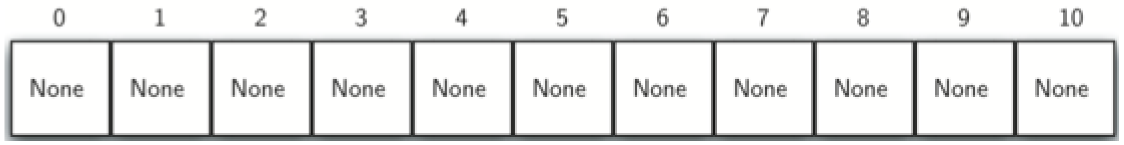
We have 11 slots (0-10)

Figure 5.5: Hash Table with Six Items

Searching

Hashing: Example

- h(item) = item%m
- h(item) = item%11



We have 11 slots (0-10)

Table 5.4: Simple Hash Function Using Remainders

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

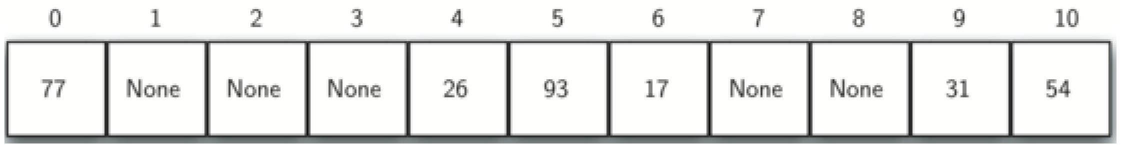
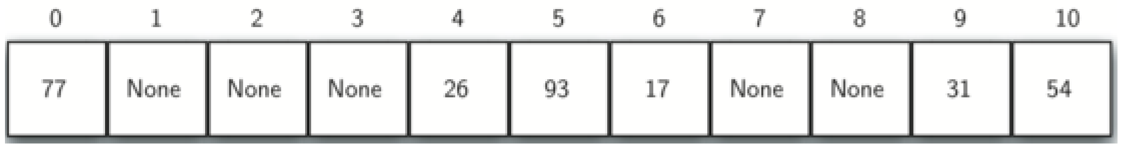


Figure 5.5: Hash Table with Six Items

Searching

Hashing: Collision

- According to the hash function, two or more items would need to be in the same slot. This is referred to as a **collision** (it may also be called a “clash”).
- Collisions create a problem for the hashing technique.
- For example, if we have a new item 44,
  - It would have a hash value of 0 (44%11 == 0).
  - **Slot 0 is already stored 77**, <u>now we have a problem</u>



0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Searching

Hashing: Hash Function

- Given an arbitrary collection of items, there is **no** systematic way to construct **a perfect hash function**, **maps each item into a unique slot**.
  - If we know the items and the collection will never change, then it is possible to construct a perfect hash function.
- **One way** to always have a perfect hash function is to **increase the size of the hash table** so that each possible value in the item range can be accommodated. This guarantees that each item will have a unique slot.
  - Although **this is practical for small numbers of items**, it is not feasible when the number of possible items is large.
  - For example, **if the items were nine-digit** Social Security numbers, this method **would require almost one billion slots**. **If we only want to store** data for a class of **25 students**, **we will be wasting an enormous amount of memory**.

Searching

Hashing: Hash Function (Cont.)

- To minimizes the number of collisions, distributes the items in the hash table.
  - **Folding Method**
  - **Mid-square Method**
- **Folding Method:**
  - Divide the item into equal-size pieces (the last piece may not be of equal size).
  - These pieces are then added together to give the resulting hash value.
  - For example, if our item was the phone number 436-555-4601,
    - We would take the digits and divide them into groups of 2 (43, 65, 55, 46, 01).
    - After the addition, 43 + 65 + 55 + 46 + 01, we get 210.
    - If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder.
    - In this case 210%11 is 1, so the phone number 436-555-4601 hashes to slot 1.
  - Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get 43 + 56 + 55 + 64 + 01 = 219 which gives 219%11 = 10.

Searching

Hashing: Hash Function (Cont.)

- **Mid-square Method:**
  - We first square the item, and then extract some portion of the resulting digits.
  - For example, if the item were 44, we would first compute 44^2 = 1, 936.
  - By extracting the middle two digits, 93, and performing the remainder step, we get 5 (93%11).

Table 5.5: Comparisons of Remainder and Mid-Square Methods

Item	Remainder	Mid-Square	
54	(54%11) = 10	3	
26	4	7	?
93	5	9	
17	6	8	?
77	0	4	
31	9	6	?

54^2 = 2916, 91%11 = 3

Searching

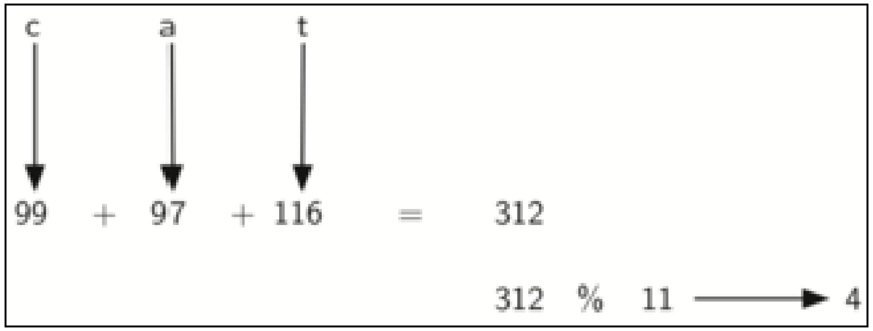
Hashing: Hash Function for String

- We can make use of ordinal value of a character, ord(*c*)
- **ord(*c*):** Given a string representing one Unicode character, return an integer representing the Unicode code point of that character.
  - For example, ord('a') returns the integer 97
  - and ord('€') (Euro sign) returns 8364.
  - This is the inverse of **chr()**. → chr(8364) = '€'

https://docs.python.org/3/library/functions.html#ord

Example: "cat"

- Take these three ordinal values, add them up, and use the remainder method to get a hash value



Searching

Hashing: Hash Function for String (Cont.)

- **Implementation (1)**
  - The code below shows a function called hash that takes a string and a table size
  - and returns the hash value in the range from 0 to table_size – 1.

```
def hash(a_string, table_size):
    sum = 0
    for pos in range(len(a_string)):
        sum = sum + ord(a_string[pos])

    return sum % table_size
```

Searching

Hashing: Hash Function for String (Cont.)

- **Implementation (2)**
  - It is interesting to note that when using this hash function, **anagrams** will always be given the same hash value. **(cinema, iceman)**
  - To remedy this, we could use the position of the character as a weight.
  - The modification to the hash function is left as an **exercise**.

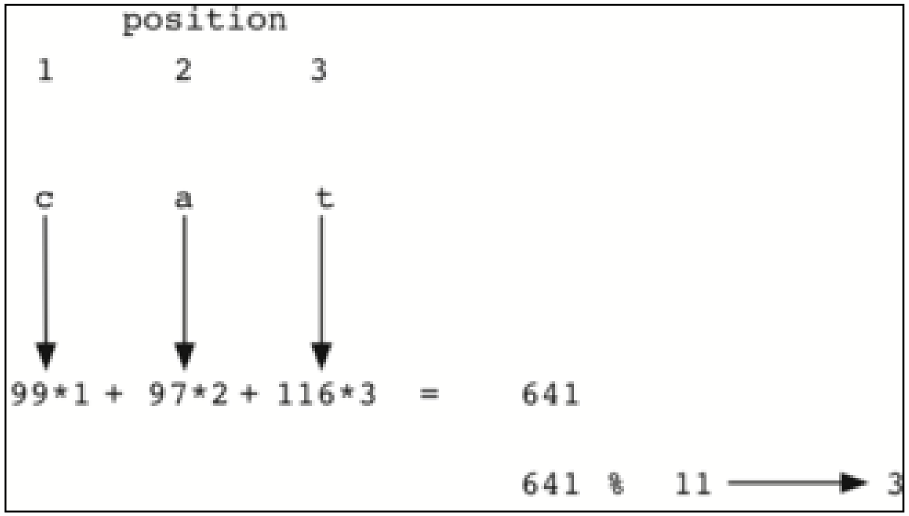


Figure 5.7: Hashing a String Using Ordinal Values with Weighting

Searching

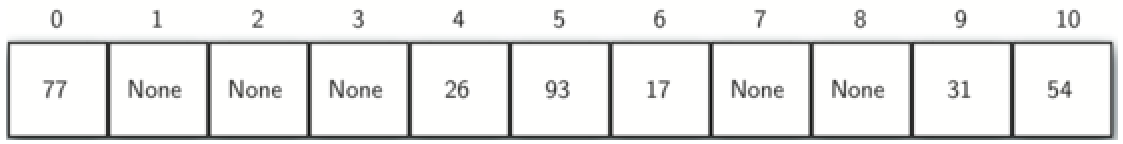
Hashing: Collision Resolution

- **Collision Resolution:** When two items hash to the same slot, we must have **a systematic method for placing the second (collision) item in the hash table**.
  - Try to find another open slot to hold the item that caused the collision, **open addressing**.
  - **Linear Probing:** find open slot sequentially, slot by slot, until we find an open position
    - **Disadvantage:**
      - The tendency (การโน้มเอียงไปสู่) for **clustering**; items become clustered in the table. ข้อมูลมีแนวโน้มที่จะไปกรองอยู่รวมกัน
    - **How to solve the clustering:**
      - We skip slots, thereby more evenly distributing the items that have caused collisions.
      - This will potentially reduce the clustering that occurs.
      - Figure 5.10 (Slide No. 27) shows the items when collision resolution is done with a “plus 3” probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.
      - The general name for this process of looking for another slot after a collision is **rehashing**.

Searching

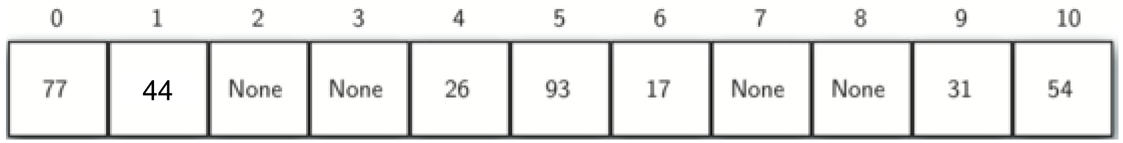
Hashing: Collision Resolution (Cont.)

- **Linear Probing: Steps**



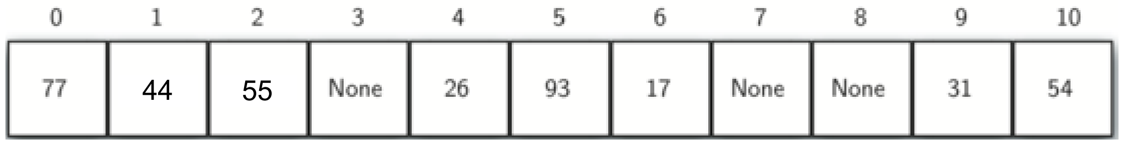
0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Next: **44**, 44%11 = 0, มี 77 แล้ว หา Slot ว่างถัดจากค่า Hash Value ดังนั้นได้ Slot 1



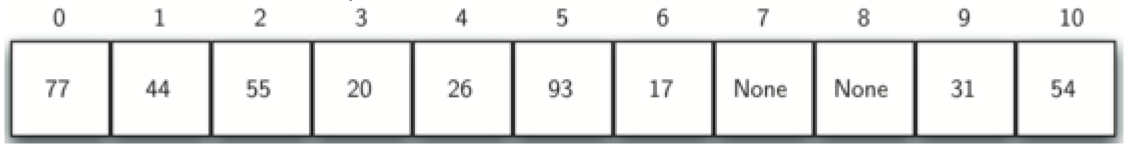
0	1	2	3	4	5	6	7	8	9	10
77	44	None	None	26	93	17	None	None	31	54

Next: **55**, 55%11 = 0, มี 77 แล้ว หา Slot ว่างถัดจากค่า Hash Value ดังนั้นได้ Slot 1 ไม่ว่าง (มี 44 อยู่) ดังนั้น ได้ Slot 2



0	1	2	3	4	5	6	7	8	9	10
77	44	55	None	26	93	17	None	None	31	54

Next: **20**, 20%11 = 9, มี 31 แล้ว หา Slot ว่างถัดจากค่า Hash Value เนื่องจาก Slot 10 ไม่ว่าง
ดังนั้น วกกลับไป 0, 1, 2 ก็ไม่ว่าง สุดท้ายได้ Slot 3



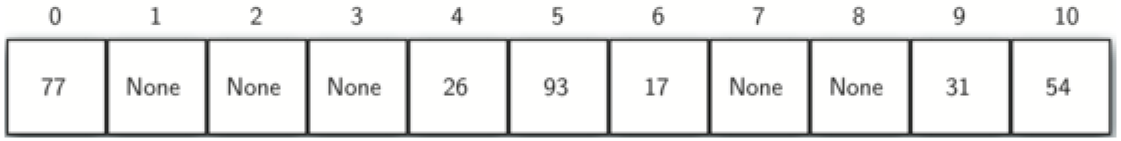
0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Figure 5.8: Collision Resolution with Linear Probing

Searching

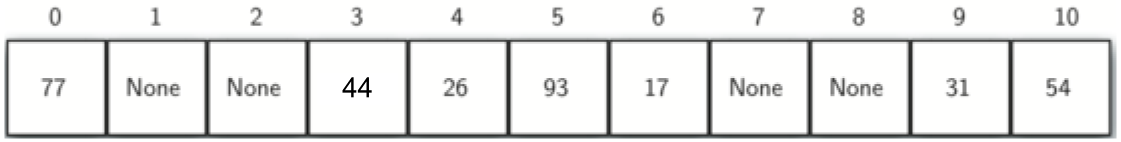
Hashing: Collision Resolution (Cont.)

- **Linear Probing: Reduce the Clustering**



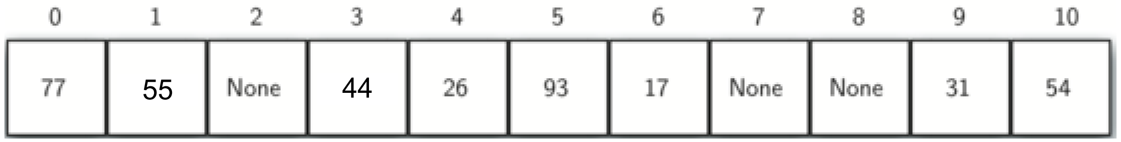
0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Next: **44**, 44%11 = 0, มี 77 แล้ว ดังนั้น Rehash = (0+3)%11 = 3 ว่าง ดังนั้นได้ Slot 3



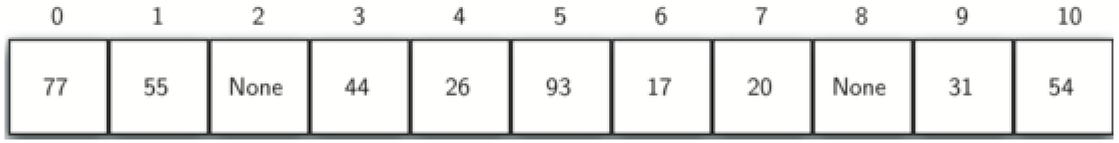
0	1	2	3	4	5	6	7	8	9	10
77	None	None	44	26	93	17	None	None	31	54

Next: **55**, 55%11 = 0, มี 77 แล้ว Rehash = (0+3)%11 = 3 ไม่ว่าง, Rehash = (3 + 3)%11 = 6 ไม่ว่าง,
Rehash = (6 +3)%11 = 9 ไม่ว่าง, Rehash = (9 +3)%11 = 1 ว่าง ได้ Slot 1



0	1	2	3	4	5	6	7	8	9	10
77	55	None	44	26	93	17	None	None	31	54

Next: **20**, 20%11 = 9, มี 31 แล้ว Rehash = (9+3)%11 = 1 ไม่ว่าง, Rehash = (12 + 3)%11 = 4 ไม่ว่าง,
Rehash = (15 +3)%11 = 7 ว่าง ได้ Slot 7



0	1	2	3	4	5	6	7	8	9	10
77	55	None	44	26	93	17	20	None	31	54

Figure 5.10: Collision Resolution Using "Plus 3"

Searching

Hashing: Collision Resolution

- **Rehashing:** a process of looking for another slot after a collision.

  new_hash_value = rehash(old_hash_value)

  where rehash(pos) = (pos + 1)%size_of_table.

  The "plus 3" rehash can be defined as

  rehash(pos) = (pos + 3)%size_of_table.

  In general,

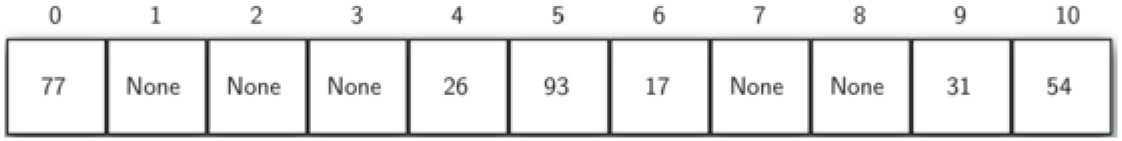
  rehash(pos) = (pos + skip)%sizeoftable.

- The size of the "skip" must be such that all the slots in the table will eventually be visited. Otherwise, part of the table will be unused.
  - To ensure this, it is often suggested that the table size be a prime number.
  - This is the reason we have been using 11 in our examples.

Searching

Hashing: Collision Resolution

- **Quadratic Probing:**
  - Instead of using a constant “skip” value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on.
  - This means that if the first hash value is h, the successive values are h+1, h+4, h+9, h+16, and so on.
  - In other words, quadratic probing uses a skip consisting of successive perfect squares.

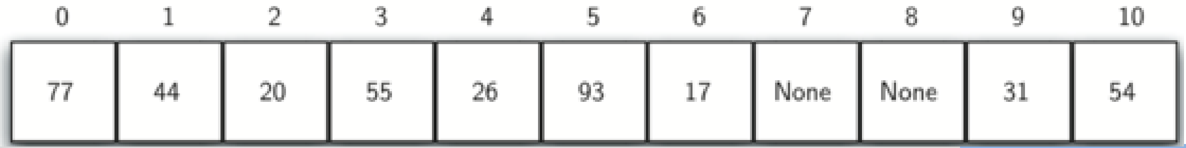


0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

44 = (0+1)%11 = 1 ว่าง

55 = (0+1)%11 = 1 ชน เพิ่มเป็น (0+4)%11 = 3 ชน … รอบที่ 5 $(0+5^2)$%11 = 3

20 = (9+1)%11 = 10 ชน เพิ่มเป็น (9+4)%11 = 2 ว่าง



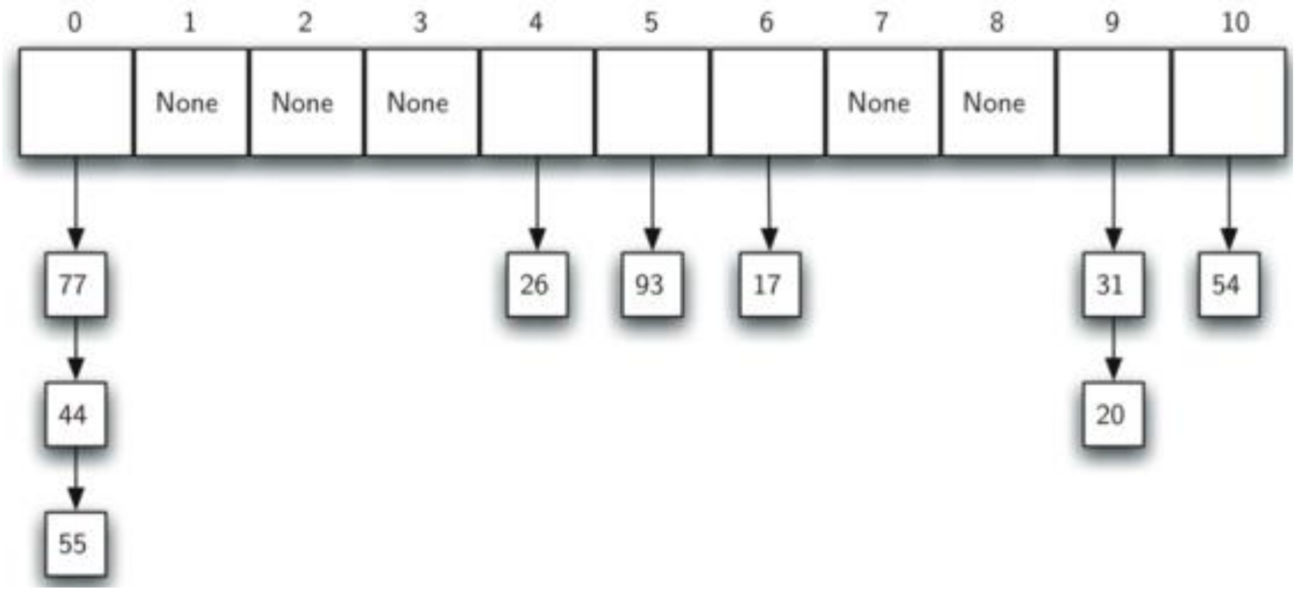
0	1	2	3	4	5	6	7	8	9	10
77	44	20	55	26	93	17	None	None	31	54

Figure 5.11: Collision Resolution with Quadratic Probing

Searching

Hashing: Collision Resolution

- **Chaining:** allows many items to exist at the same location.
  - When collisions happen, the item is still placed in the proper slot of the hash table.
  - As more and more items hash to the same location, the difficulty of searching for the item in the collection increases.



Searching

Hashing: Collision Resolution

Practice 1:

- Suppose you are given the following set of keys to insert into a hash table that holds exactly 11 values:

  113, 117, 97, 100, 114, 108, 116, 105, 99.

- Demonstrates the contents of the has table after all the keys have been inserted using **linear probing**?

Searching

Hashing: Implementing the Map Abstract Data Type

Map Abstract Data Type

- The structure is an unordered collection of associations between a key and a data value (Dictionary in python is a solution).
- The keys in a map are all unique so that there is a one-to-one relationship between a key and a value.

Searching

Hashing: Implementing the Map Abstract Data Type

Map Abstract Data Type (Cont.)

- **Map()**
  Create a new, empty map. It returns an empty map collection.
- **put(key,val)**
  Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- **get(key)**
  Given a key, return the value stored in the map or None otherwise.
- **del**
  Delete the key-value pair from the map using a statement of the form del map[key].
- **len()**
  Return the number of key-value pairs stored in the map.
- **in** Return True for a statement of the form key in map, if the given key is in the map, False otherwise.

Searching

Hashing: Implementing HashTable class

HashTable class

- Download HashTable class and testHashTable at 204700 web site
- Then run testHashTable.py, the results are

```
[77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]

['bird', 'goat', 'pig', 'chicken', 'dog', 'lion', 'tiger', None, None, 'cow', 'cat']
```

Searching

Hashing: Analysis of Hashing

- The best case hashing would provide a $O(1)$
- For collision, searching depends on load factor, λ.

$$\lambda = \text{number\_of\_items} / \text{table\_size}$$

- Therefore the worst case is $O(\lambda)$.

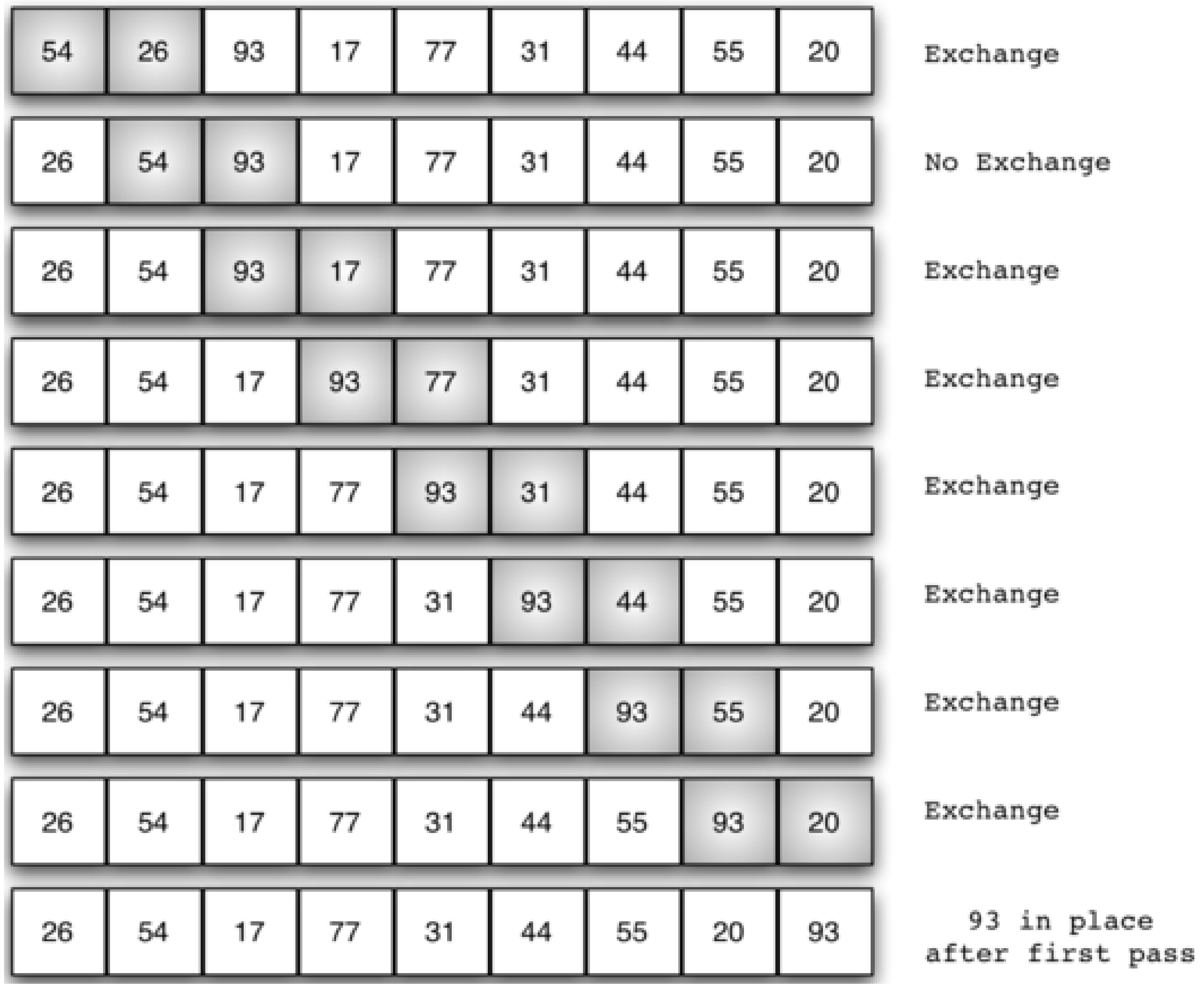
Sorting

- **Bubble Sort**
- **Selection Sort**
- **Insertion Sort**
- http://www.ee.ryerson.ca/~courses/coe428/sorting/bubblesort.html
- **Shell Sort**
- **Merge Sort**
- **Quick Sort**

Download python code ของการ Sort
ทุกประเภทจาก Web รายวิชา

Sorting

- **Bubble Sort**



54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

ต่อไปทำ 26 – 20

- - -

สุดท้ายเหลือแค่ 2 ตัว คู่หน้า

Sorting

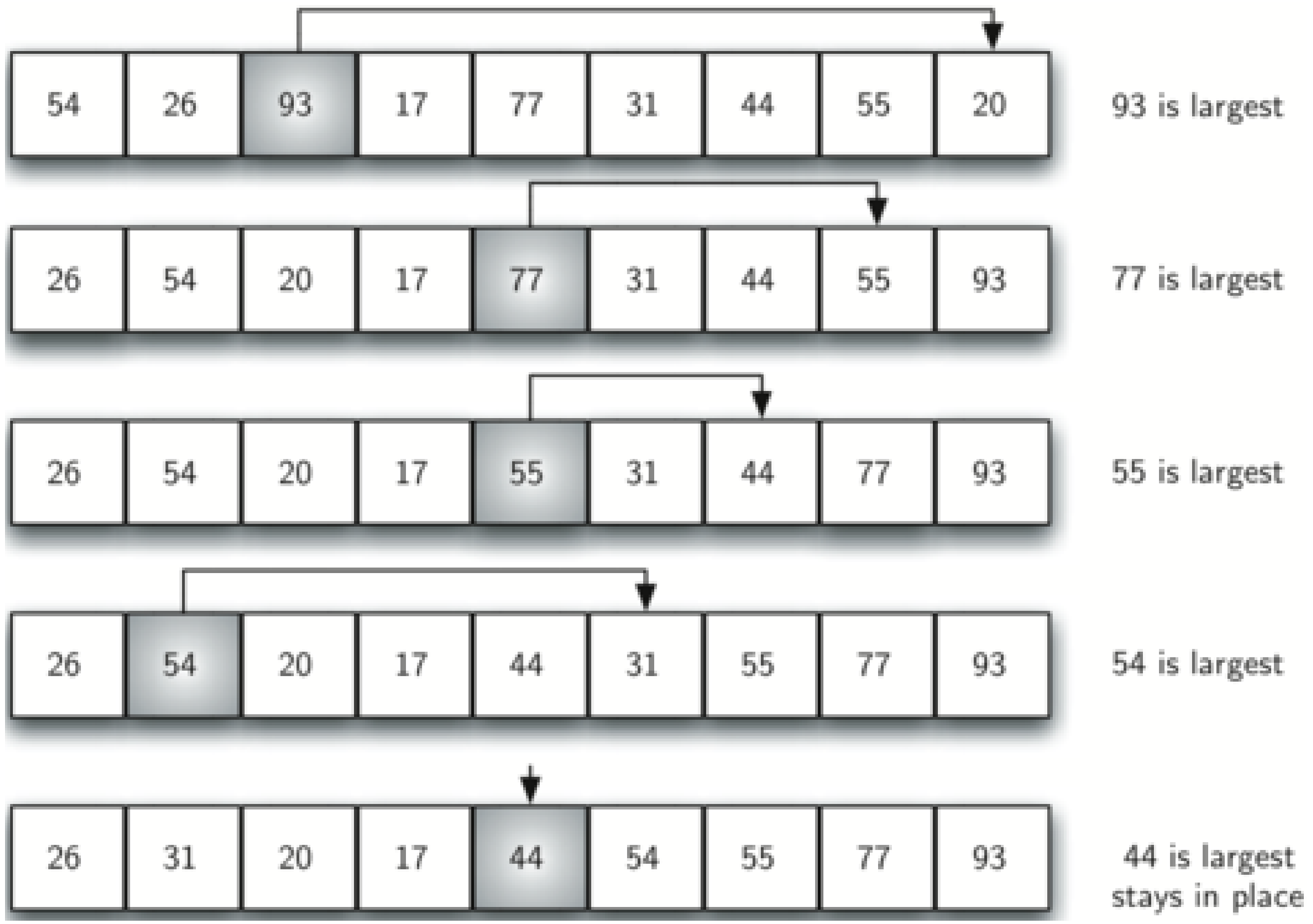
- **Bubble Sort: Analysis of Bubble Sort**

Pass	Comparisons
1	$n-1$
2	$n-2$
3	$n-3$
$\ldots$	$\ldots$
$n-1$	1

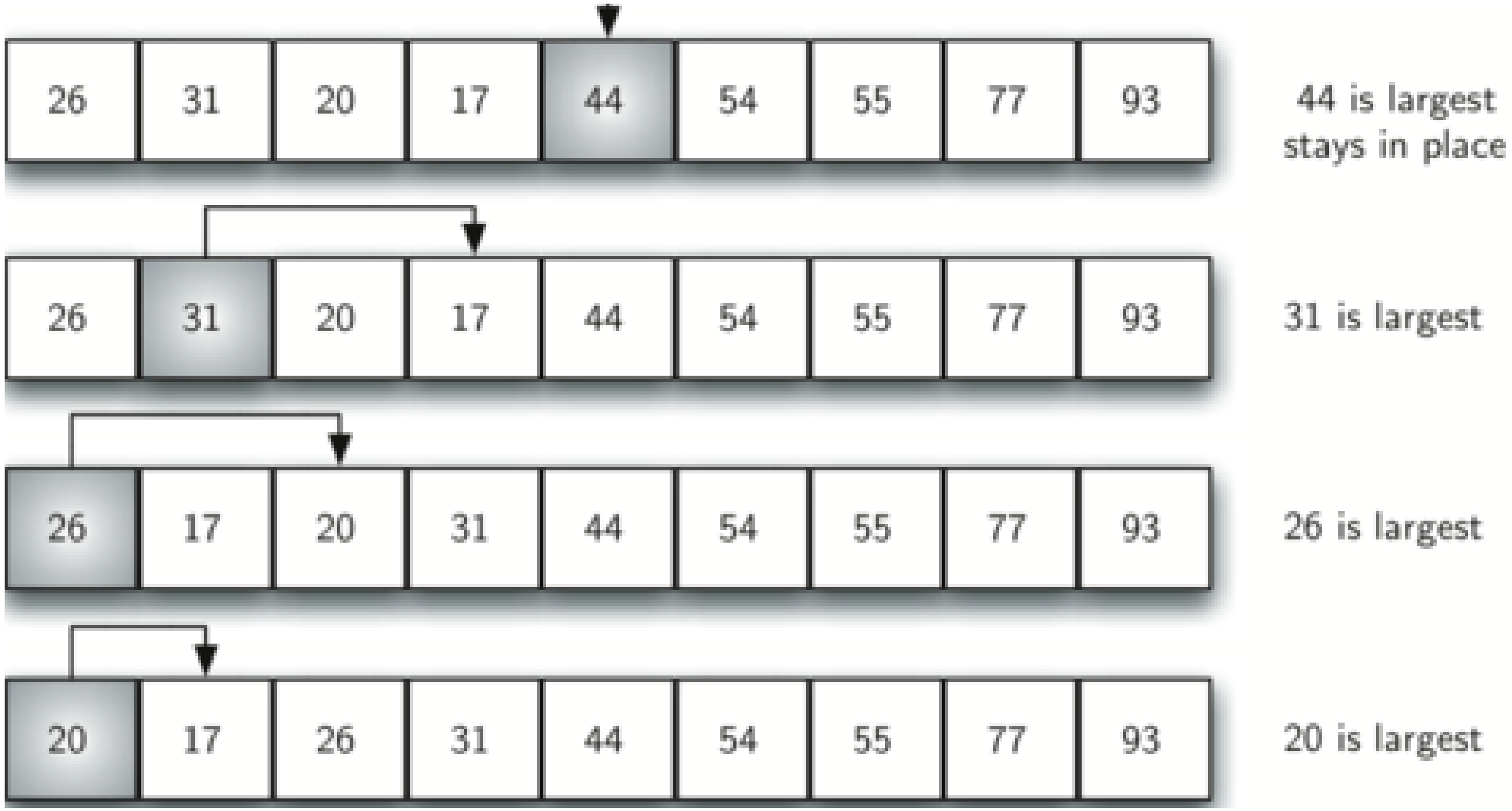
Table 5.6: Comparisons for Each Pass of Bubble Sort

จำนวนครั้งที่ทำ (n-1) + (n-2) + (n-3) + … + (n-(n-2)) + (n-(n-1)) บวกกัน (n-1) ครั้ง
ประมาณการสำหรับค่าที่มากที่สุด (n-1)*(n-1) = $n^2 - 2n + 1$ ประมาณ **O(n^2)**

Sorting: Selection Sort

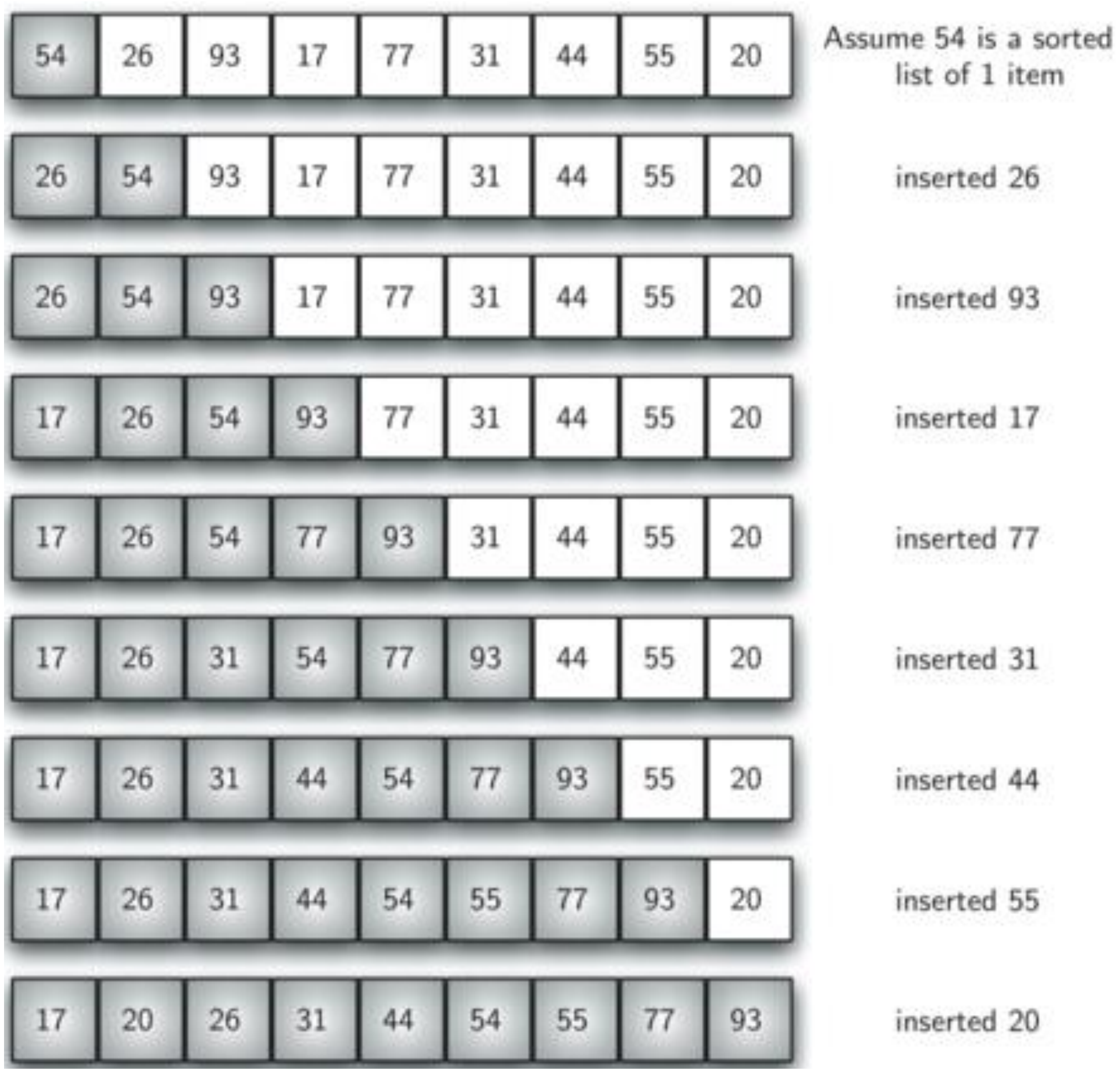


Sorting: Selection Sort (Cont.)



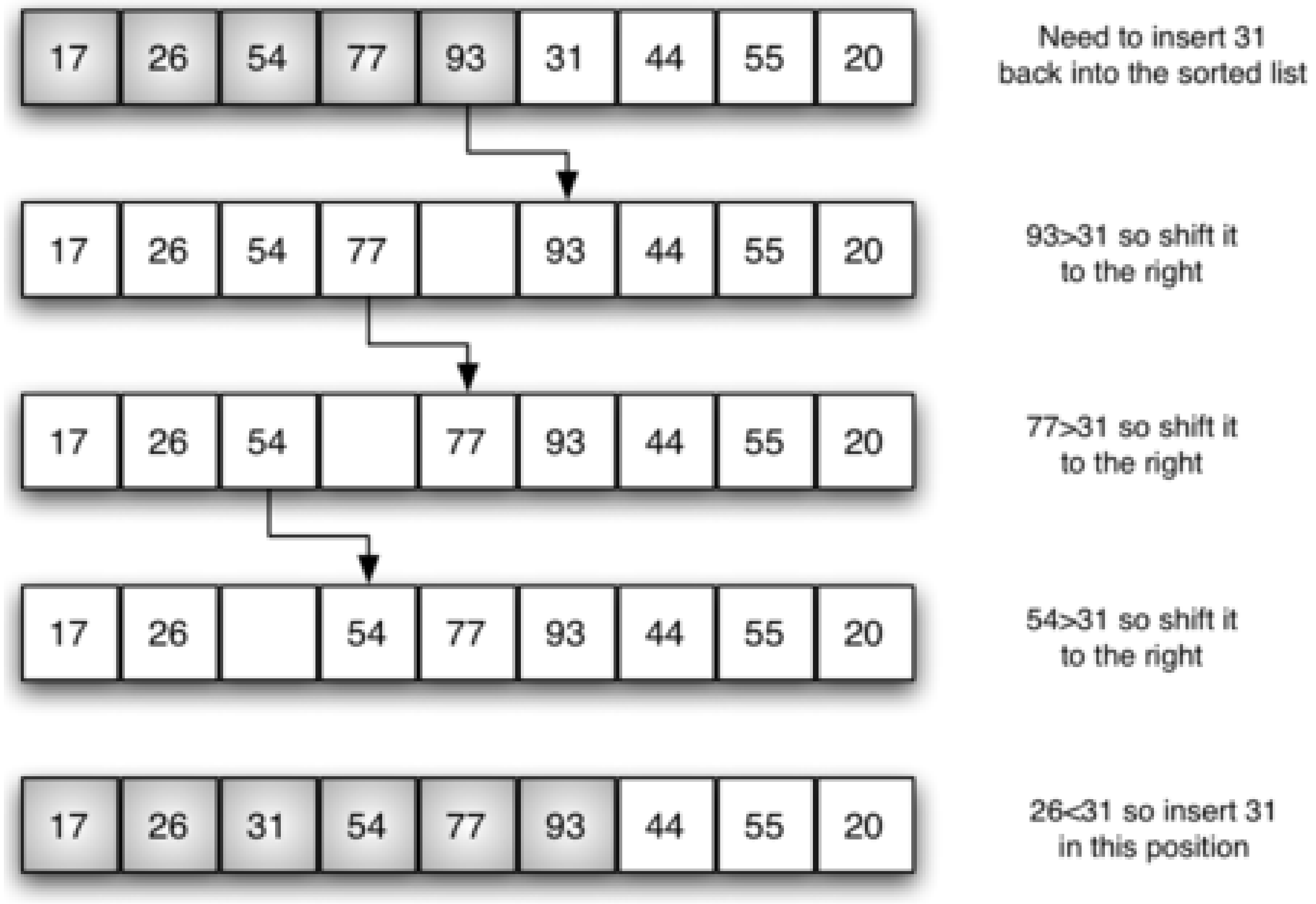
ทำ n รอบ แต่ละรอบ หา ค่า max เปรียบเทียบ (n-1) … 1 ดังนั้น ยังคง $O(n^2)$

Sorting: Insertion Sort



54 26 93 17 77 31 44 55 20 — Assume 54 is a sorted list of 1 item
26 54 93 17 77 31 44 55 20 — inserted 26
26 54 93 17 77 31 44 55 20 — inserted 93
17 26 54 93 77 31 44 55 20 — inserted 17
17 26 54 77 93 31 44 55 20 — inserted 77
17 26 31 54 77 93 44 55 20 — inserted 31
17 26 31 44 54 77 93 55 20 — inserted 44
17 26 31 44 54 55 77 93 20 — inserted 55
17 20 26 31 44 54 55 77 93 — inserted 20

Sorting: Insertion Sort


17 26 54 77 93 31 44 55 20
Need to insert 31 back into the sorted list
17 26 54 77 93 44 55 20
93>31 so shift it to the right
17 26 54 77 93 44 55 20
77>31 so shift it to the right
17 26 54 77 93 44 55 20
54>31 so shift it to the right
17 26 31 54 77 93 44 55 20
26<31 so insert 31 in this position

Still $O(n^2)$

Sorting: Shell Sort

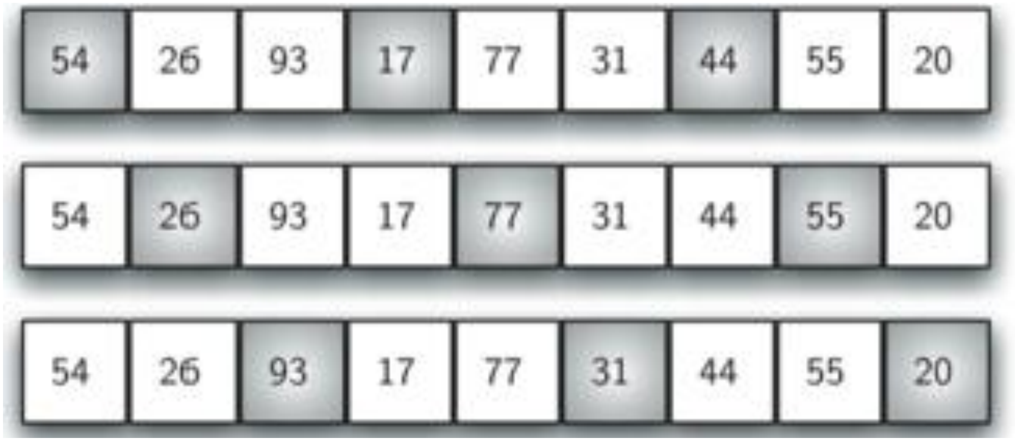
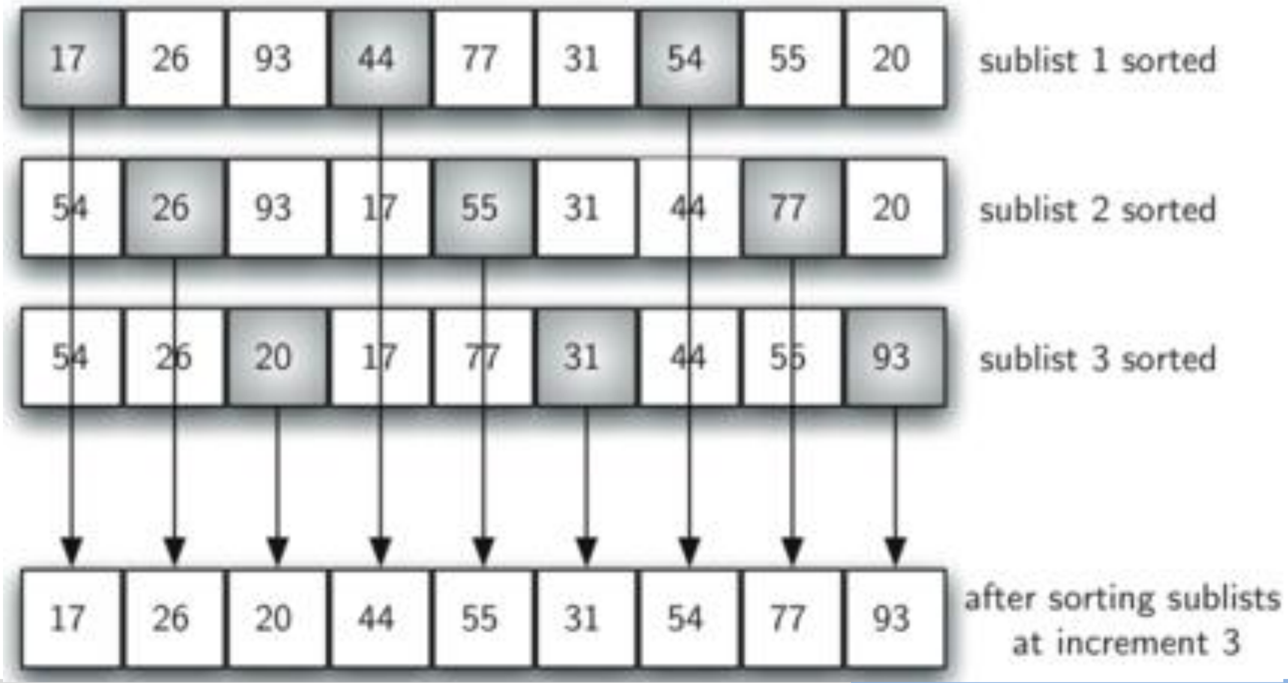


Figure 5.18: A Shell Sort with Increments of Three



Sorting: Shell Sort (Cont.)

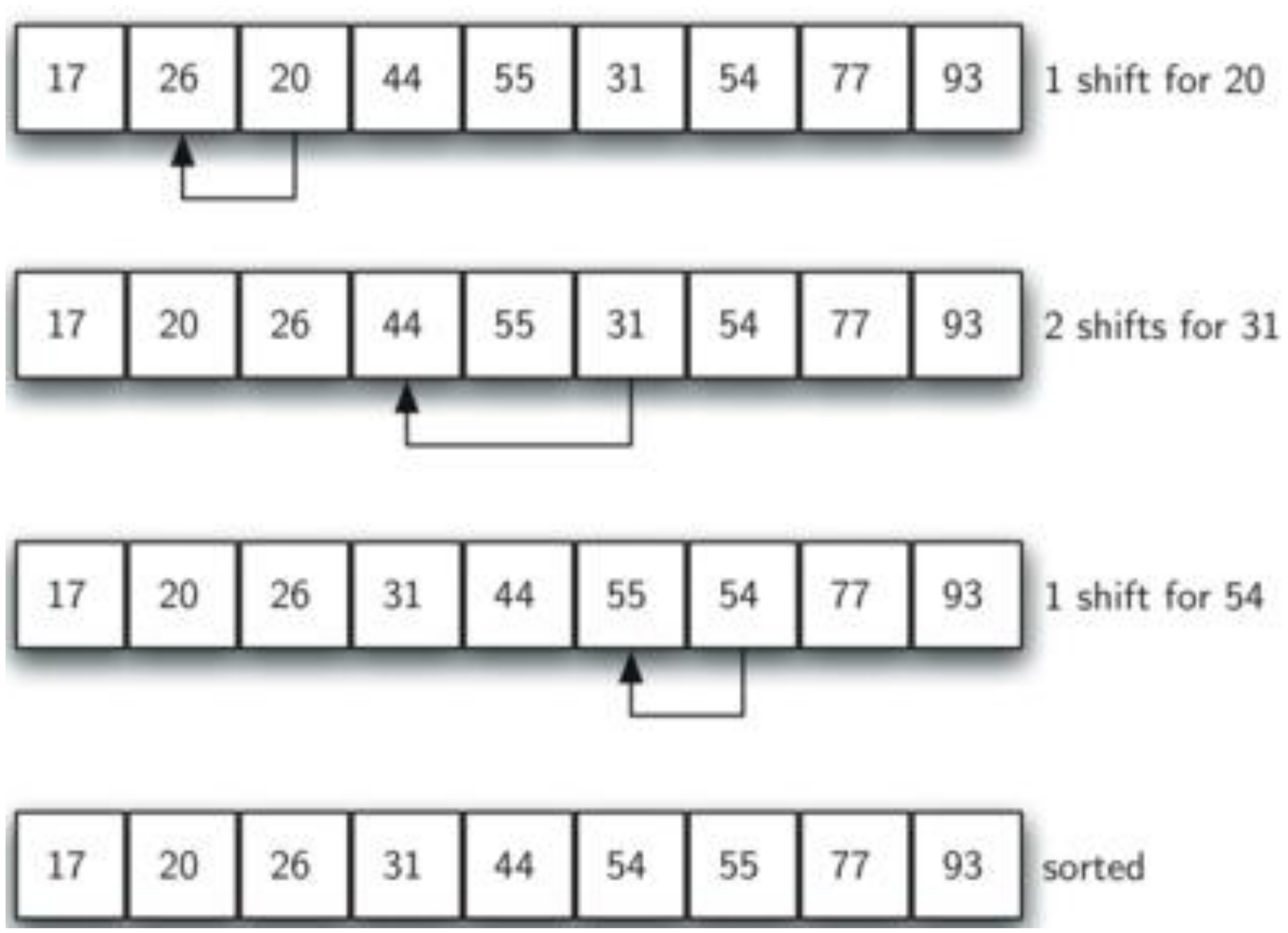


Figure 5.20: Shell Sort: A Final Insertion Sort with Increment of 1

More Example: http://www.tutorialspoint.com/data_structures_algorithms/shell_sort_algorithm.htm

Sorting: Merge Sort

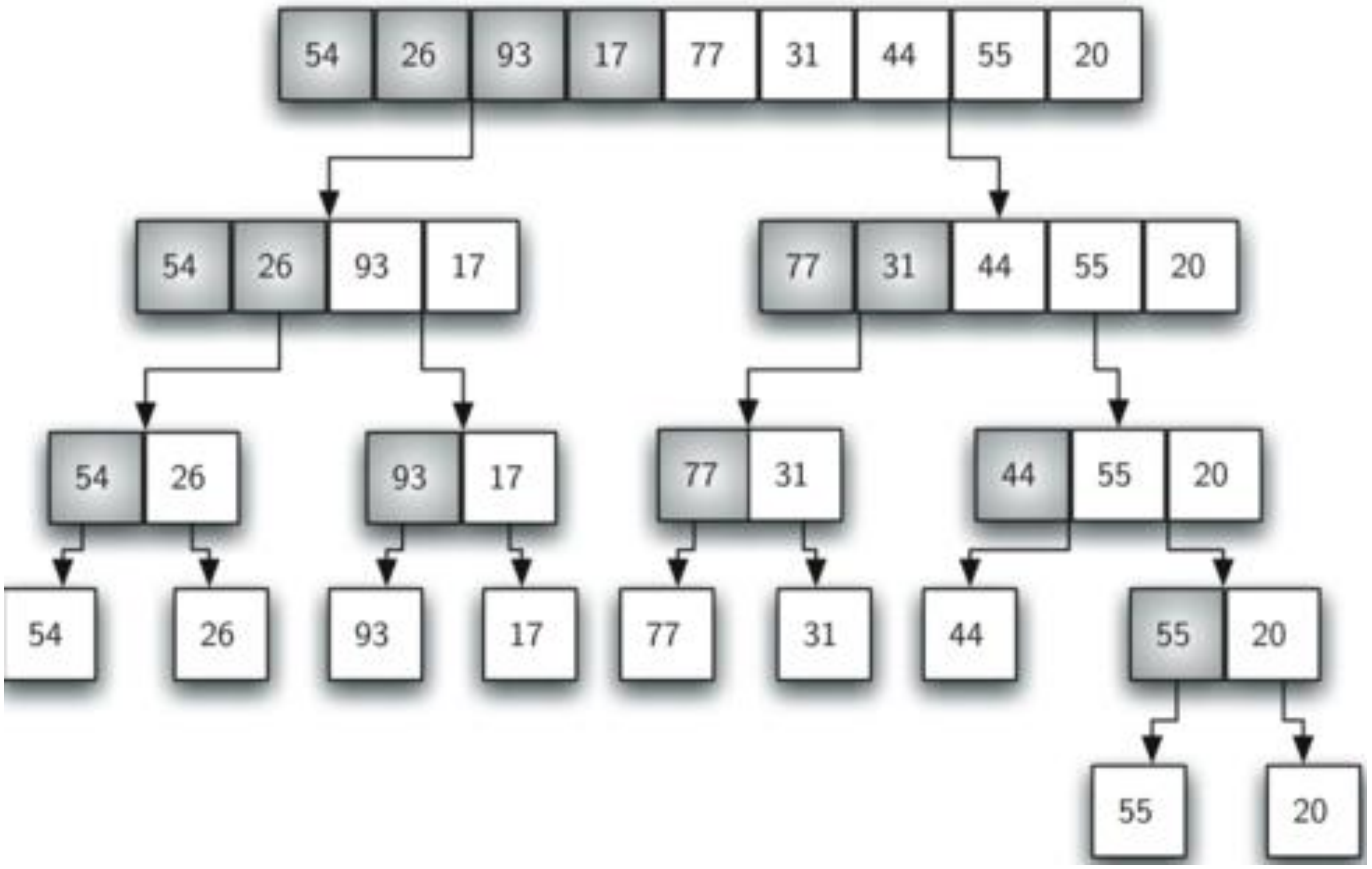


Figure 5.22: Splitting the List in a Merge Sort

Sorting: Merge Sort (Cont.)

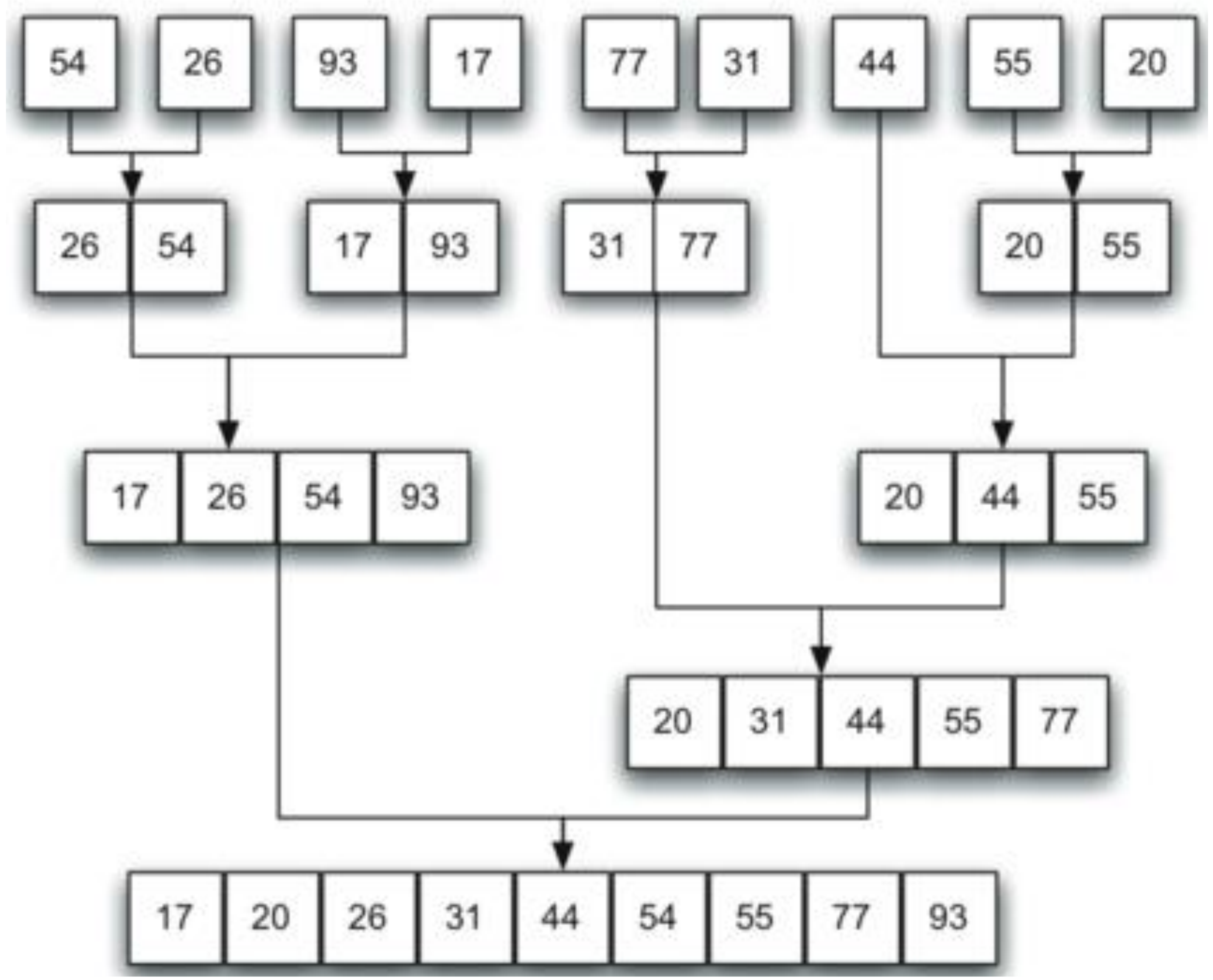


Figure 5.23: Lists as They Are Merged Together

More Example: http://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm

Sorting: Quick Sort

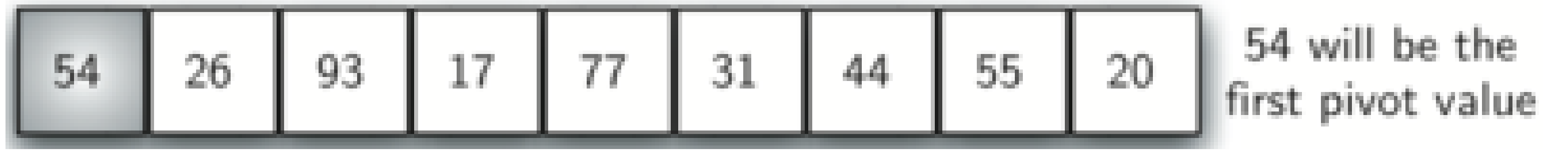
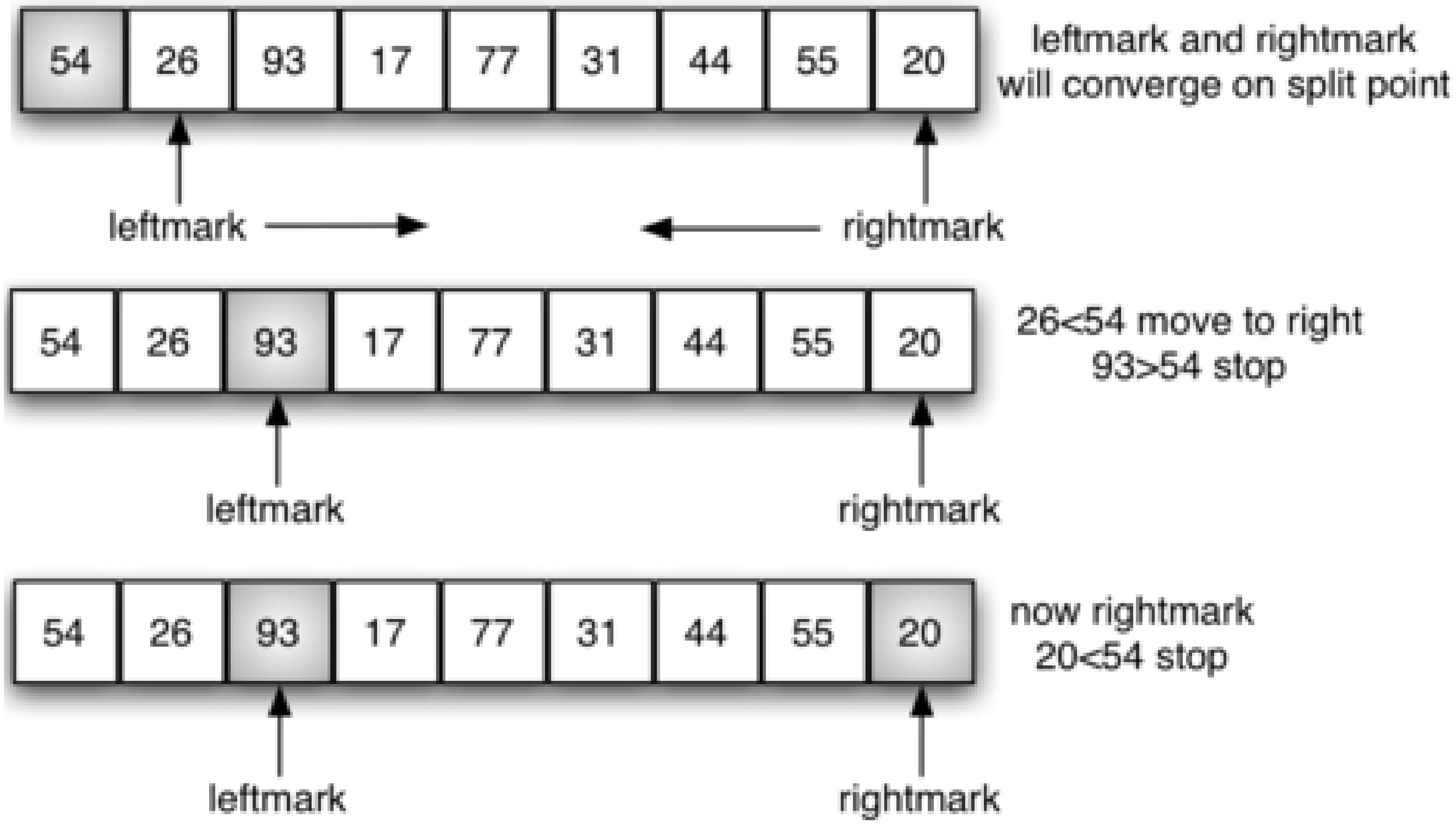


Figure 5.24: The First Pivot Value for a Quick Sort



Sorting: Quick Sort (Cont.)

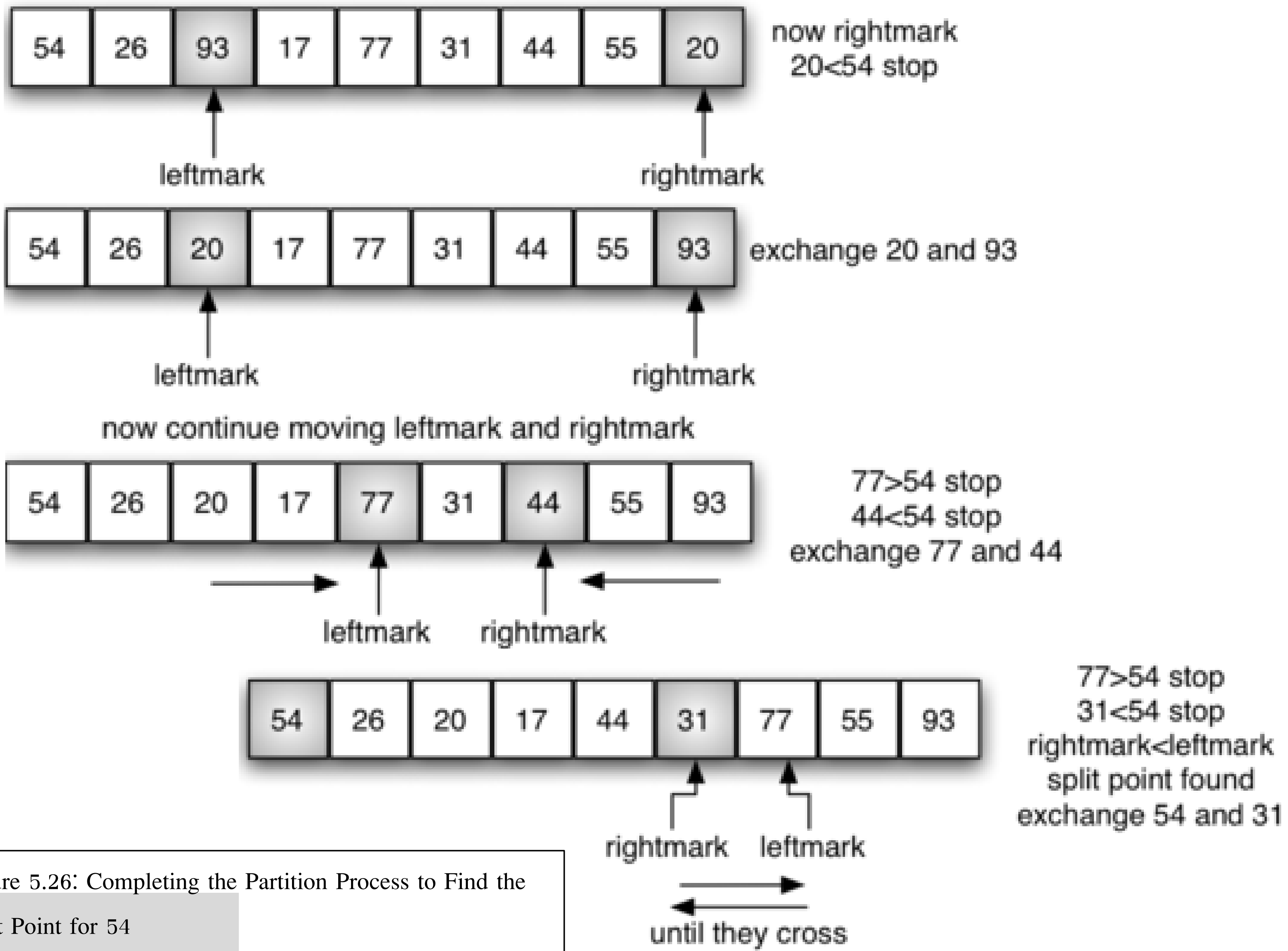
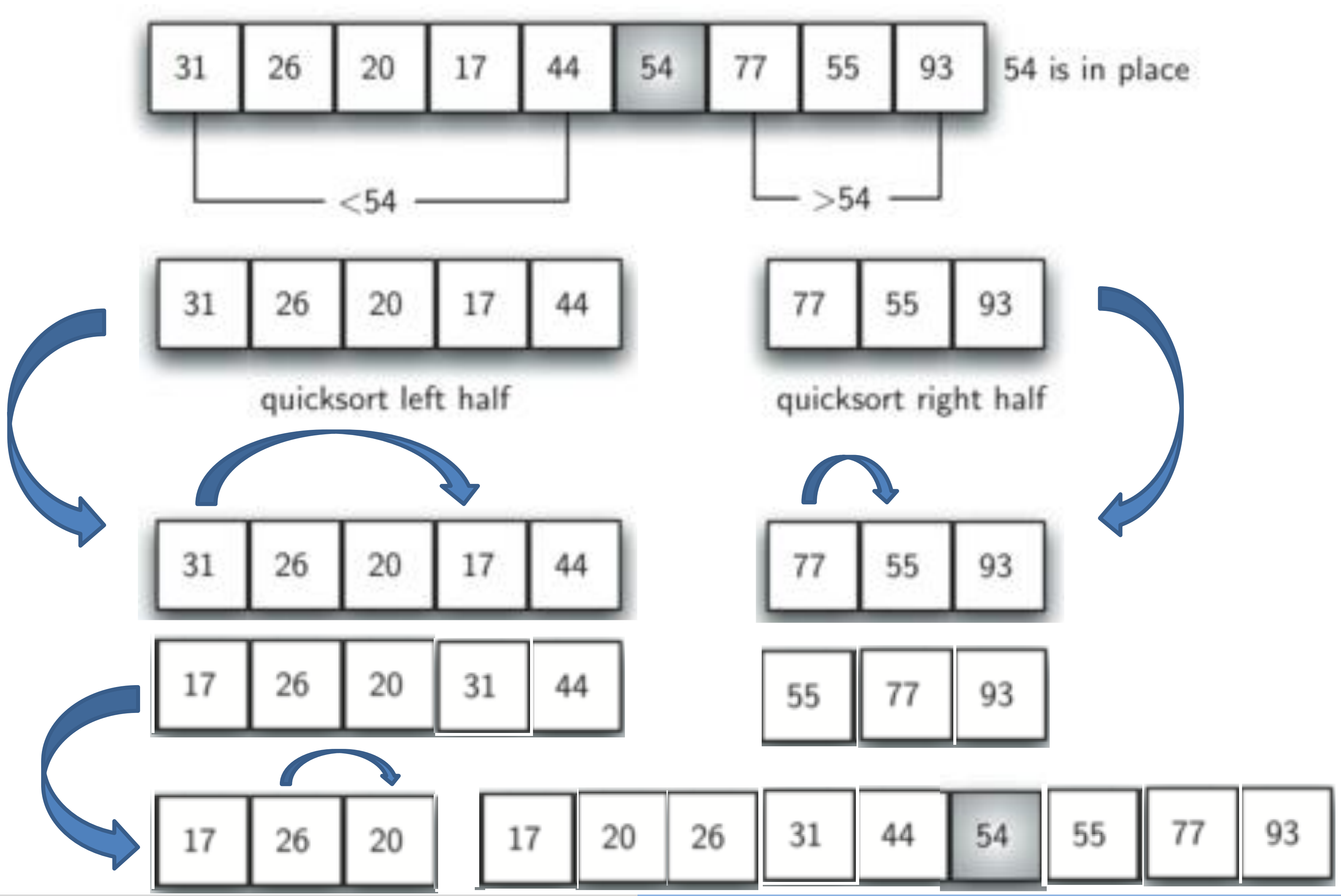


Figure 5.26: Completing the Partition Process to Find the Split Point for 54

Sorting: Quick Sort (Cont.)


31 26 20 17 44 54 77 55 93
54 is in place
<54
>54
31 26 20 17 44
quicksort left half
77 55 93
quicksort right half
31 26 20 17 44
77 55 93
17 26 20 31 44
55 77 93
17 26 20
17 20 26 31 44 54 55 77 93