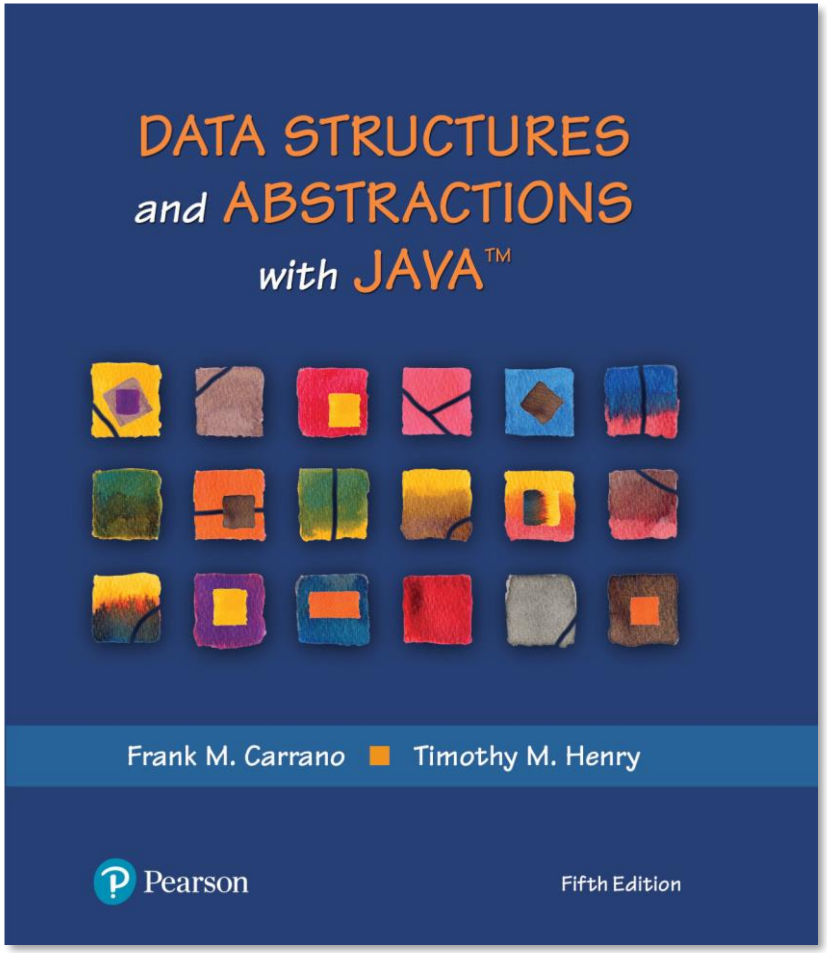
Data Structures and Abstractions with Java™

5th Edition



Chapter 19

Searching

The Problem

FIGURE 19-1 Searching is an everyday occurrence

Iterative Sequential Search of an Unsorted Array

```
public static <T> boolean inArray(T[] anArray, T anEntry)
{
   boolean found = false;
   int index = 0;
   while (!found && (index < anArray.length))
   {
      if (anEntry.equals(anArray[index]))
         found = true;
      index++;
   } // end while

   return found;
} // end inArray
```

Using a loop to search for a specific valued entry.

Iterative Sequential Search of an Unsorted Array

(a) A successful search for 8

Look at 9:

9	5	8	4	7

$8 \neq 9$, so continue searching.

Look at 5:

9	**5**	8	4	7

$8 \neq 5$, so continue searching.

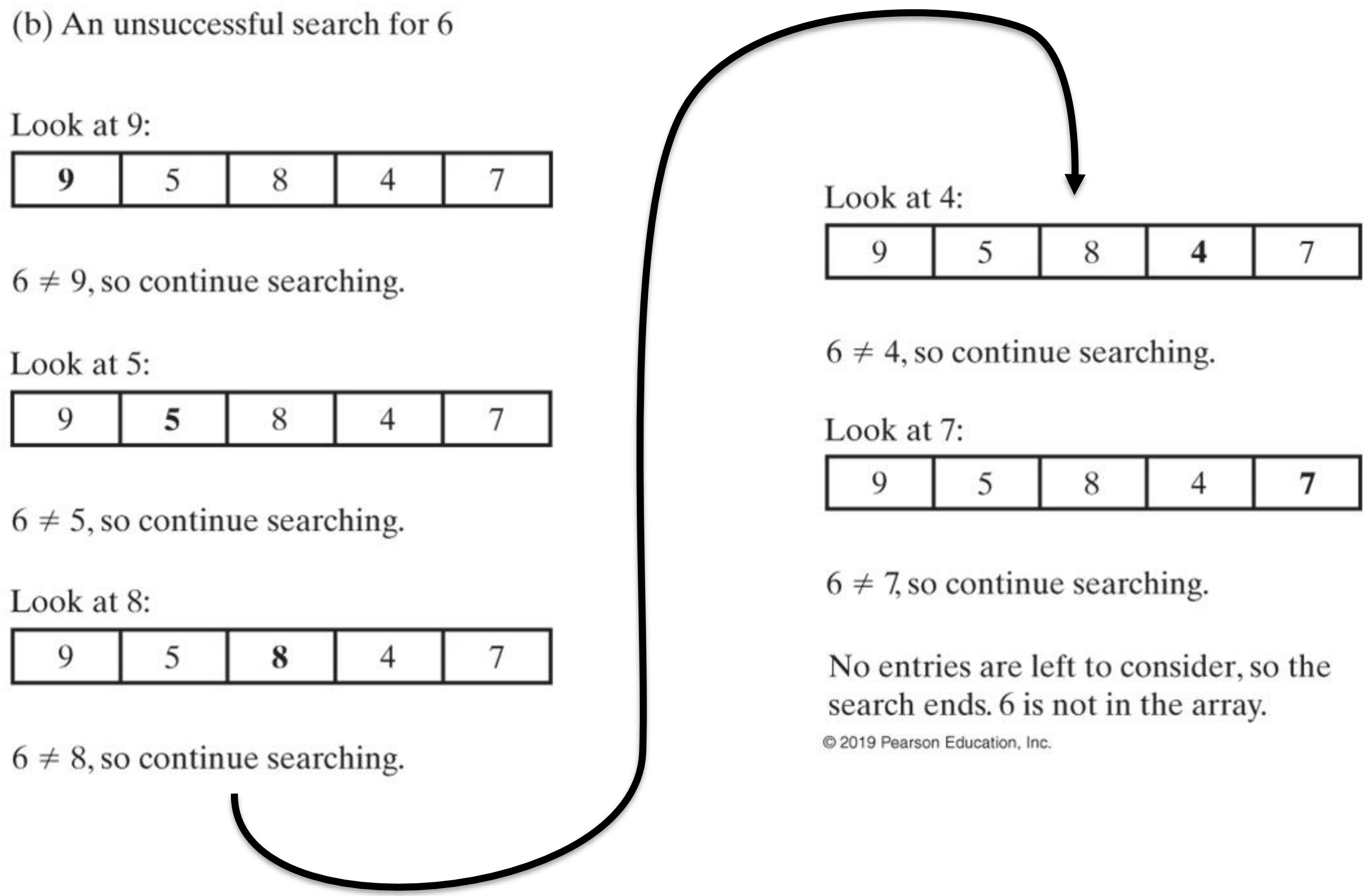
Look at 8:

9	5	**8**	4	7

$8 = 8$, so the search has found 8.

FIGURE 19-2a An iterative sequential search of an array

Iterative Sequential Search of an Unsorted Array



(b) An unsuccessful search for 6

Look at 9:

9	5	8	4	7

6 ≠ 9, so continue searching.

Look at 5:

9	**5**	8	4	7

6 ≠ 5, so continue searching.

Look at 8:

9	5	**8**	4	7

6 ≠ 8, so continue searching.

Look at 4:

9	5	8	**4**	7

6 ≠ 4, so continue searching.

Look at 7:

9	5	8	4	**7**

6 ≠ 7, so continue searching.

No entries are left to consider, so the search ends. 6 is not in the array.

FIGURE 19-2b An iterative sequential search of an array

Recursive Sequential Search of an Unsorted Array

```
Algorithm to search a[first] through a[last] for desiredItem
if (there are no elements to search)
return false
   else if (desiredItem equals a[first])
return true else
   return the result of searching a[first + 1] through a[last]
```

Pseudocode of the logic of our recursive algorithm.

Recursive Sequential Search of an Unsorted Array

```
/** Searches an array for anEntry. */
public static <T> boolean inArray(T[] anArray, T anEntry)
{
   return search(anArray, 0, anArray.length - 1, anEntry);
} // end inArray

// Searches anArray[first] through anArray[last] for desiredItem.
// first >= 0 and < anArray.length.
// last >= 0 and < anArray.length.
private static <T> boolean search(T[] anArray, int first, int last, T desiredItem)
{
   boolean found;
   if (first > last)
      found = false; // No elements to search
   else if (desiredItem.equals(anArray[first]))
      found = true;
   else
      found = search(anArray, first + 1, last, desiredItem);

   return found;
} // end search
```

Method that implements this algorithm will need parameters first and last.

Recursive Sequential Search of an Unsorted Array

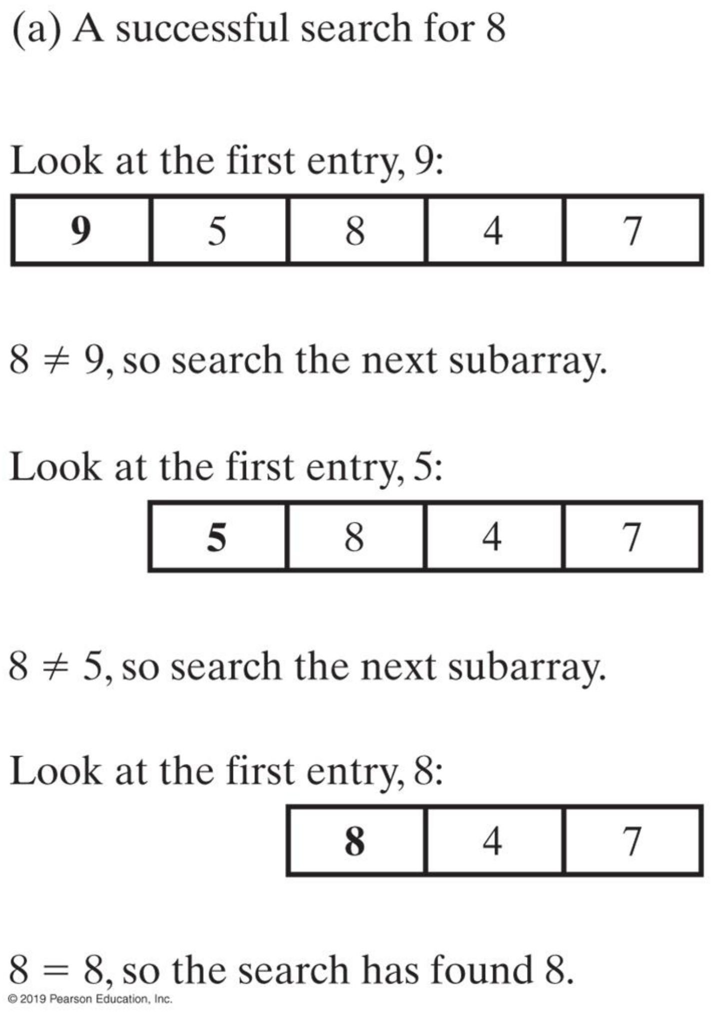
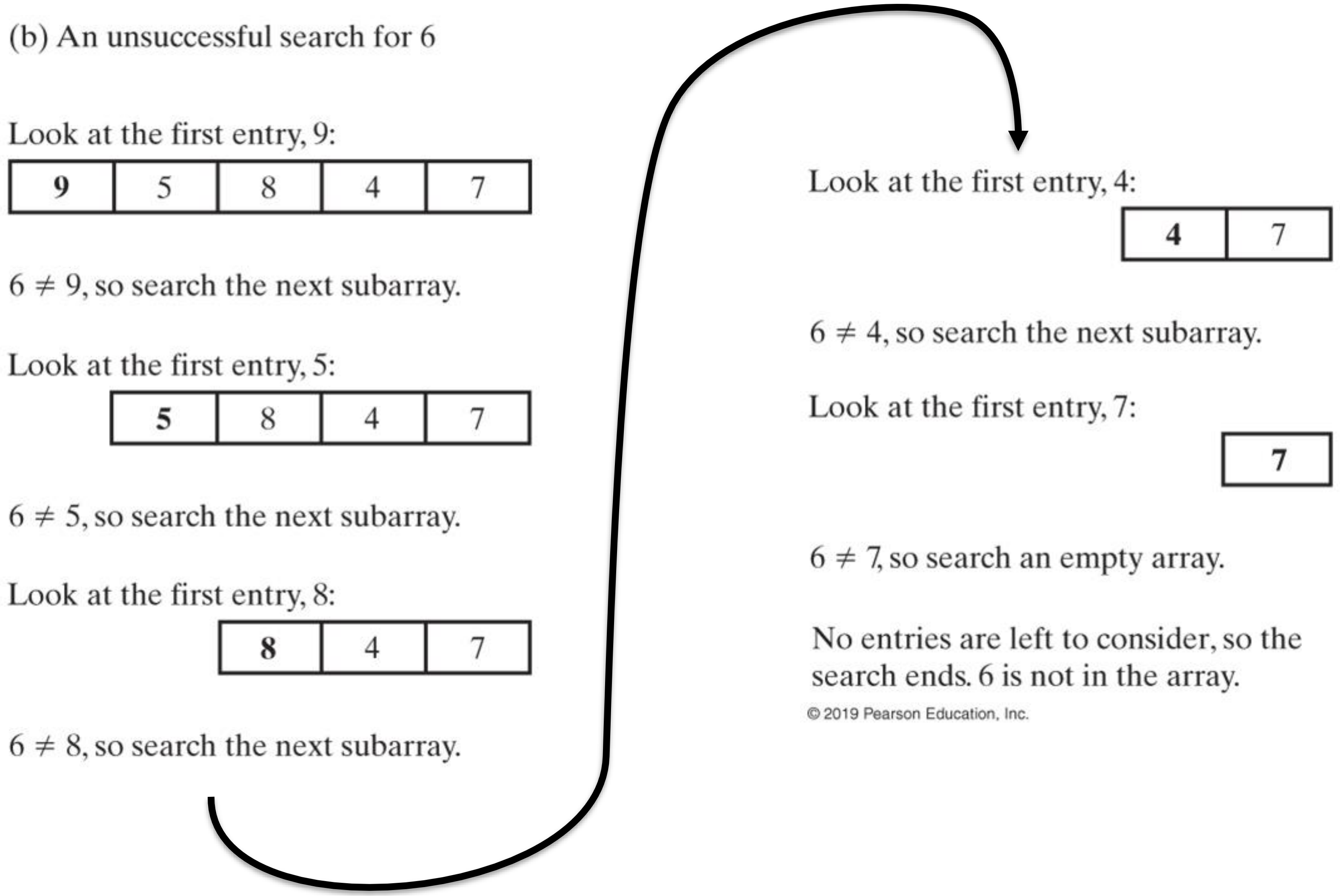

(a) A successful search for 8
Look at the first entry, 9:
9 5 8 4 7
8 ≠ 9, so search the next subarray.
Look at the first entry, 5:
5 8 4 7
8 ≠ 5, so search the next subarray.
Look at the first entry, 8:
8 4 7
8 = 8, so the search has found 8.
© 2019 Pearson Education, Inc.

FIGURE 19-3a A recursive sequential search of an array

Recursive Sequential Search of an Unsorted Array



(b) An unsuccessful search for 6

Look at the first entry, 9:

9	5	8	4	7

6 ≠ 9, so search the next subarray.

Look at the first entry, 5:

5	8	4	7

6 ≠ 5, so search the next subarray.

Look at the first entry, 8:

8	4	7

6 ≠ 8, so search the next subarray.

Look at the first entry, 4:

4	7

6 ≠ 4, so search the next subarray.

Look at the first entry, 7:

7

6 ≠ 7, so search an empty array.

No entries are left to consider, so the search ends. 6 is not in the array.

FIGURE 19-3b A recursive sequential search of an array

Efficiency of a Sequential Search of an Array

- The time efficiency of a sequential search of an array.
  - Best case $O(1)$
  - Worst case: $O(n)$
  - Average case: $O(n)$

Sequential Search of a Sorted Array

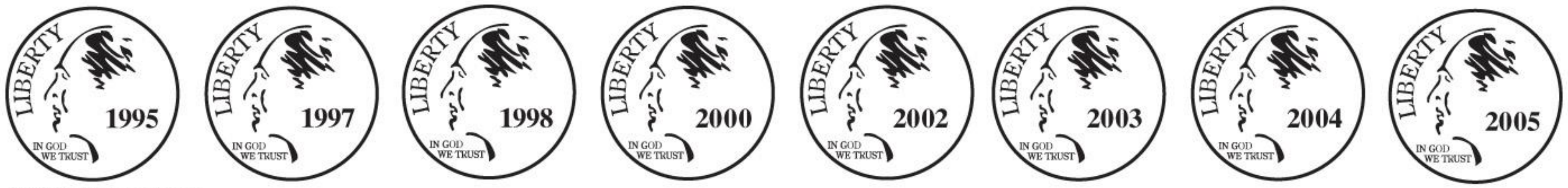


FIGURE 19-4 Coins sorted by their mint dates

Binary Search of a Sorted Array

I don't need this half of the book. I'll just throw it away.
© 2019 Pearson Education, Inc.

FIGURE 19-5 Ignoring one half of the data when the data is sorted

Binary Search of a Sorted Array

```
Algorithm to search a[0] through a[n − 1] for desiredItem
mid = approximate midpoint between 0 and n – 1
if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return the result of searching a[0] through a[mid – 1]
else if (desiredItem > a[mid])
   return the result of searching a[mid + 1] through a[n – 1]
```

First draft of an algorithm for a binary search of an array

Binary Search of a Sorted Array

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = approximate midpoint between first and last
if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid − 1, desiredItem)
else if (desiredItem > a[mid])
   return binarySearch(a, mid + 1, last, desiredItem)
```

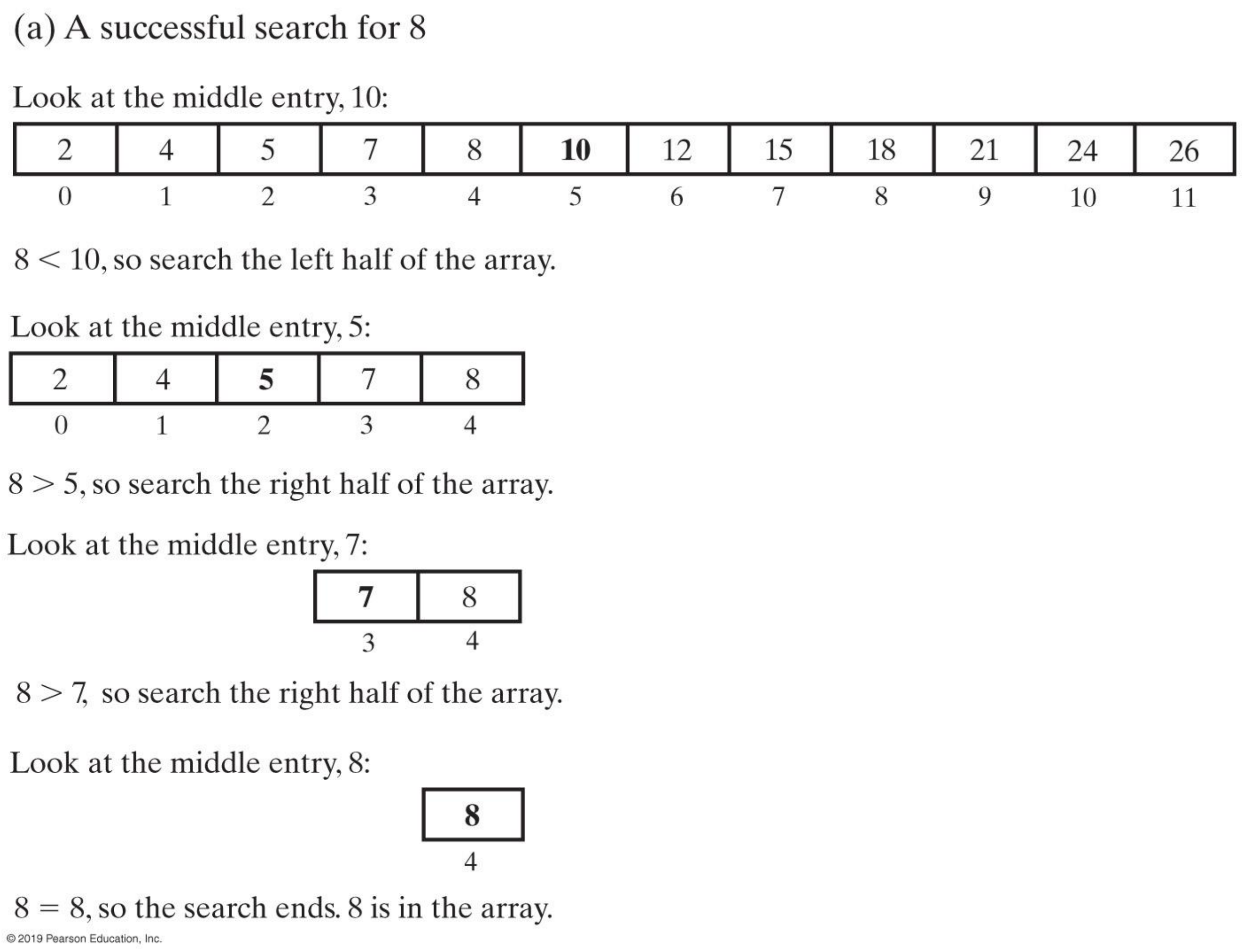
Revision of binary search algorithm as method

Binary Search of a Sorted Array

```
*Algorithm* binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // *Approximate midpoint*
if (first > last)
   return false
else if (desiredItem *equals* a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid − 1, desiredItem)
else // desiredItem > a[mid]
   return binarySearch(a, mid + 1, last, desiredItem)
```

Refine the logic a bit, get a more complete algorithm

Binary Search of a Sorted Array



(a) A successful search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

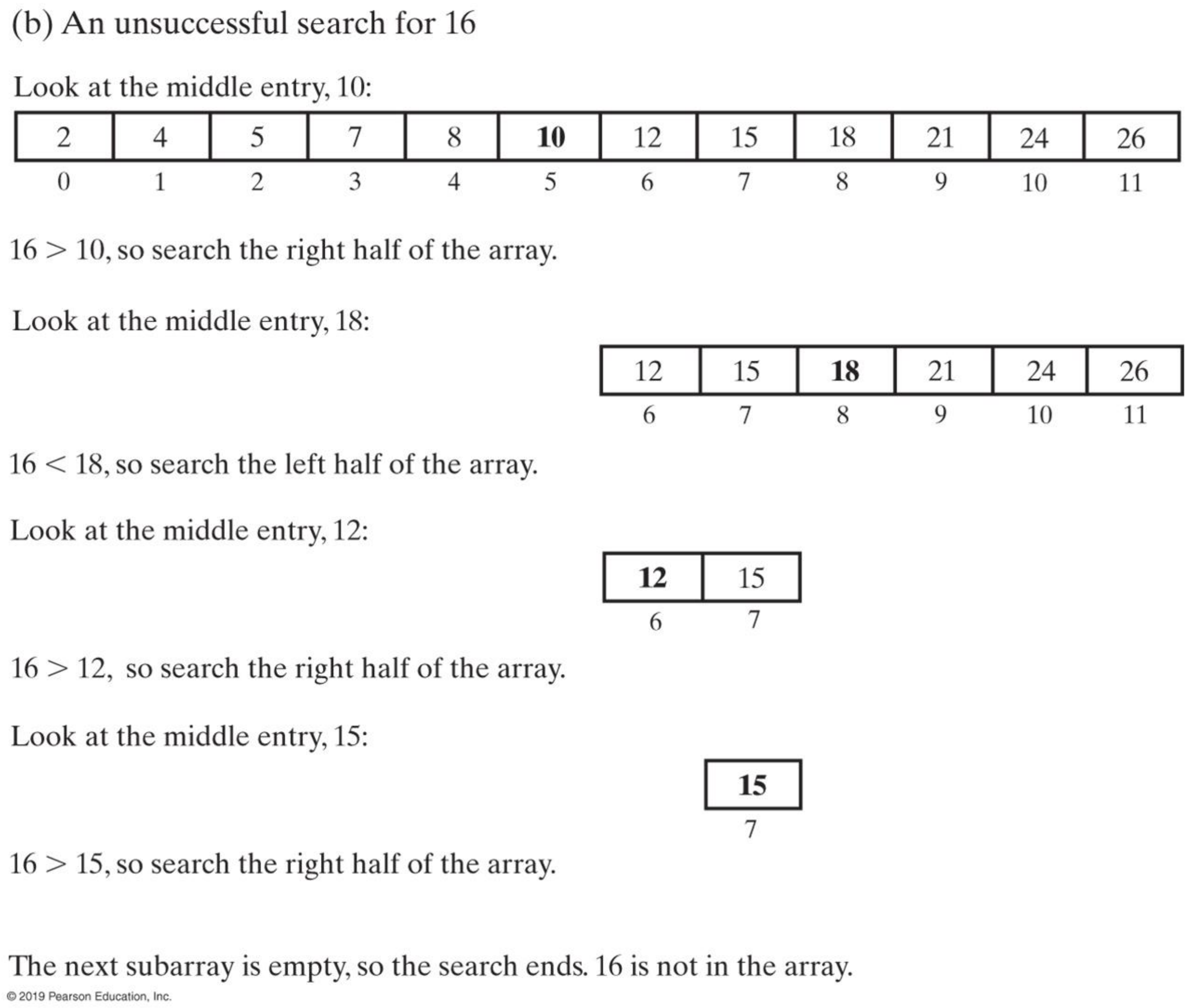
Look at the middle entry, 8:

8
4

$8 = 8$, so the search ends. 8 is in the array.

FIGURE 19-6a A recursive binary search of a sorted array

Binary Search of a Sorted Array



(b) An unsuccessful search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$16 > 10$, so search the right half of the array.

Look at the middle entry, 18:

12	15	**18**	21	24	26
6	7	8	9	10	11

$16 < 18$, so search the left half of the array.

Look at the middle entry, 12:

12	15
6	7

$16 > 12$, so search the right half of the array.

Look at the middle entry, 15:

15
7

$16 > 15$, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

FIGURE 19-6b A recursive binary search of a sorted array

Binary Search of a Sorted Array

```
private static <T extends Comparable<? super T>>
       boolean binarySearch(T[] anArray, int first, int last, T desiredItem)
{
  boolean found;
  int mid = first + (last - first) / 2;

  if (first > last)
    found = false;
  else if (desiredItem.equals(anArray[mid]))
    found = true;
  else if (desiredItem.compareTo(anArray[mid]) < 0)
    found = binarySearch(anArray, first, mid - 1, desiredItem);
  else
    found = binarySearch(anArray, mid + 1, last, desiredItem);

  return found;
} // end binarySearch

public static <T extends Comparable<? super T>> boolean inArray(T anEntry)
{
  return binarySearch(anArray, 0, anArray.length - 1, anEntry);
} // end inArray
```

Implementation of the method `binarySearch`

Java Class Library: The Method `binarySearch`

```
/** Searches an entire array for a given item.
 @param array    An array sorted in ascending order.
 @param desiredItem The item to be found in the array.
 @return Index of the array entry that equals desiredItem;
        otherwise returns –belongsAt – 1, where belongsAt is
 the index of the array element that should contain desiredItem. */
public  static  int  binarySearch(type[]  array,  type  desiredItem);
```

Static method `binarySearch` specification

Efficiency of a Binary Search of an Array

- The time efficiency of a binary search of an array
  - Best case: $O(1)$
  - Worst case: $O(\log n)$
  - Average case: $O(\log n)$

Iterative Sequential Search of an Unsorted Chain

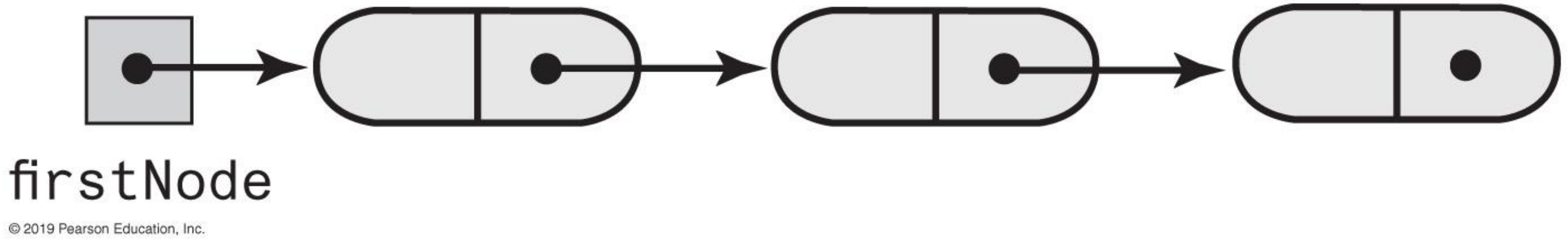


FIGURE 19-7 A chain of linked nodes that contain the entries in a list

Iterative Sequential Search of an Unsorted Chain

```
public boolean contains(T anEntry)
{
   boolean found = false;
   Node currentNode = firstNode;

   while (!found && (currentNode != null))
   {
      if (anEntry.equals(currentNode.getData()))
         found = true;
      else
         currentNode = currentNode.getNextNode();
   } // end while

   return found;
} // end contains
```

Implementation of iterative search in `contains`

Recursive Sequential Search of an Unsorted Chain

```
private boolean search(Node currentNode, T desiredItem)
{
   boolean found;

   if (currentNode == null)
      found = false;
   else if (desiredItem.equals(currentNode.getData()))
      found = true;
   else
      found = search(currentNode.getNextNode(), desiredItem);

   return found;
} // end search

public boolean contains(T anEntry)
{
   return search(firstNode, anEntry);
} // end contains
```

Implementation of recursive search in `search`

Iterative Sequential Search of a Sorted Chain

```
public boolean contains(T anEntry)
{
   Node currentNode = firstNode;

   while ( (currentNode != null) &&
           (anEntry.compareTo(currentNode.getData()) > 0) )
   {
      currentNode = currentNode.getNextNode();
   } // end while

   return (currentNode != null) &&
           anEntry.equals(currentNode.getData());
} // end contains
```

Implementation of iterative search in `contains`

Binary Search of a Sorted Chain

- First find middle of the chain:
  - You must traverse the whole chain
  - Then traverse one of the halves to find the middle of that half
- Conclusion
  - Hard to implement
  - Less efficient than sequential search

Choosing between Iterative Search and Recursive Search

Operation	Best Case	Average Case	Linked
`Sequential Search (unsorted data)`	O(1)	O(*n*)	O(*n*)
`Sequential Search (sorted data)`	O(1)	O(*n*)	O(*n*)
`Binary Search (sorted array)`	O(1)	O(log *n*)	O(log *n*)

FIGURE 19-8 The time efficiency of searching, expressed in Big Oh notation

Choosing between Iterative Search and Recursive Search

- Iterative Searches
  - Can save some time and space
- Recursive Searches
  - Will not require much additional space for the recursive calls
  - Coding binary search recursively is easier

End

Chapter 19