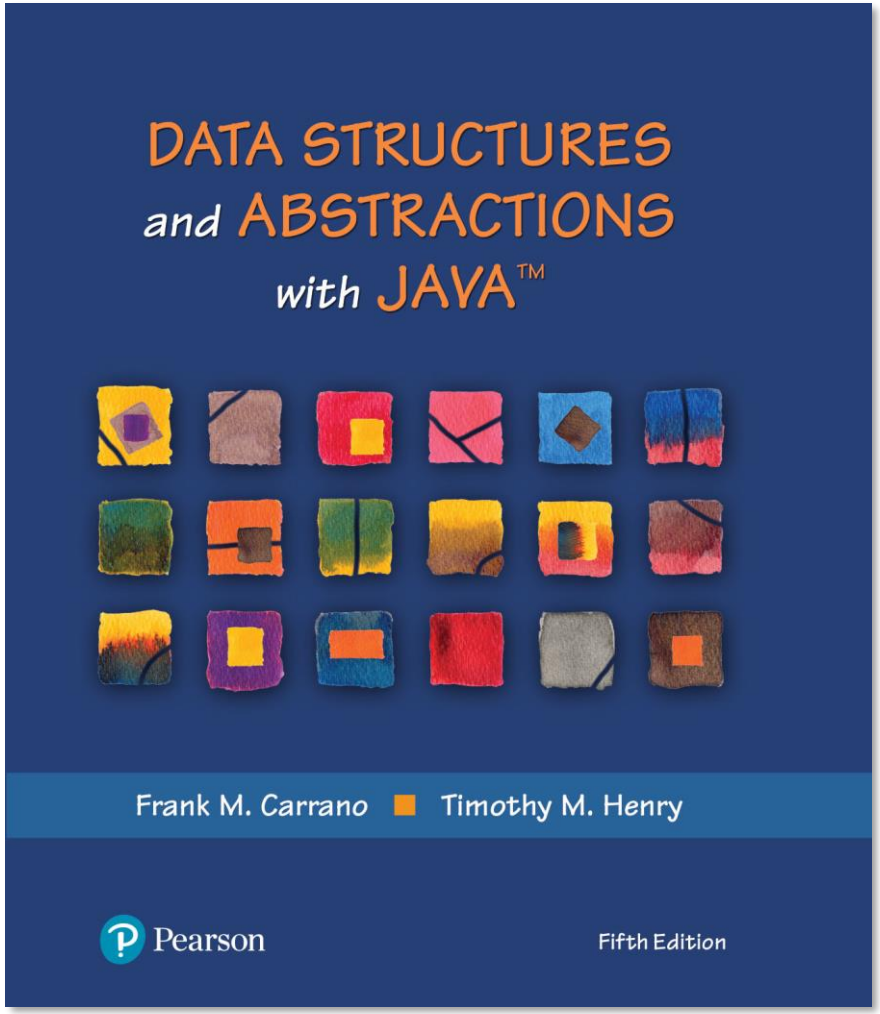
Data Structures and Abstractions with Java™

5th Edition



Chapter 17

Sorted Lists

List

- Entries in a list are ordered simply by positions within list
- Can add a sort operation to the ADT list
- Add an entry to, remove an entry from sorted list
  - Provide only the entry.
  - No specification where entry belongs or exists

Specifications for ADT Sorted List

- DATA
  - A collection of objects in sorted order and having the same data type
  - The number of objects in the collection
- We will simply reuse revised ListInterface and then use a subclass of AList to implement
- Operations requiring special consideration
  - **`add(newEntry) – just overridden for AList`**
  - **`removeEntry(anEntry)`**
  - **`getPosition(anEntry)`**

Specifications for ADT Sorted List

- Additional Operations
  - These behave as they do for the ADT List
    - `getEntry(givenPosition)`
    - `contains(anEntry)`
    - `remove(givenPosition)`
    - `clear()`
    - `size()`
    - `isEmpty()`
    - `toArray()`

Linked Sorted List Implementation

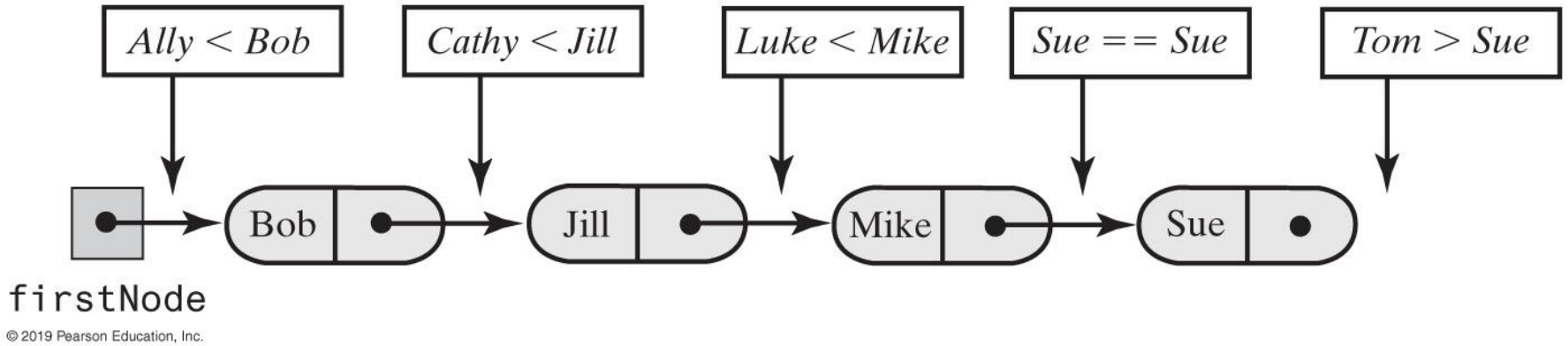


FIGURE 17-1 Places to insert additional names into a sorted chain of linked nodes

Linked Sorted List Implementation

Algorithm **add(newEntry)**

// Adds a new entry to the sorted list.

Allocate a new node containing newEntry

Search the chain until either you find a node containing newEntry *or you pass the point where it should be*

Let nodeBefore *reference the node before the insertion point*

if (*the chain is empty or the new node belongs at the beginning of the chain*)

  Add the new node to the beginning of the chain

else

  Insert the new node after the node referenced by nodeBefore

Increment the length of the sorted list

Algorithm for `add` method.

Recursive Add to Sorted List

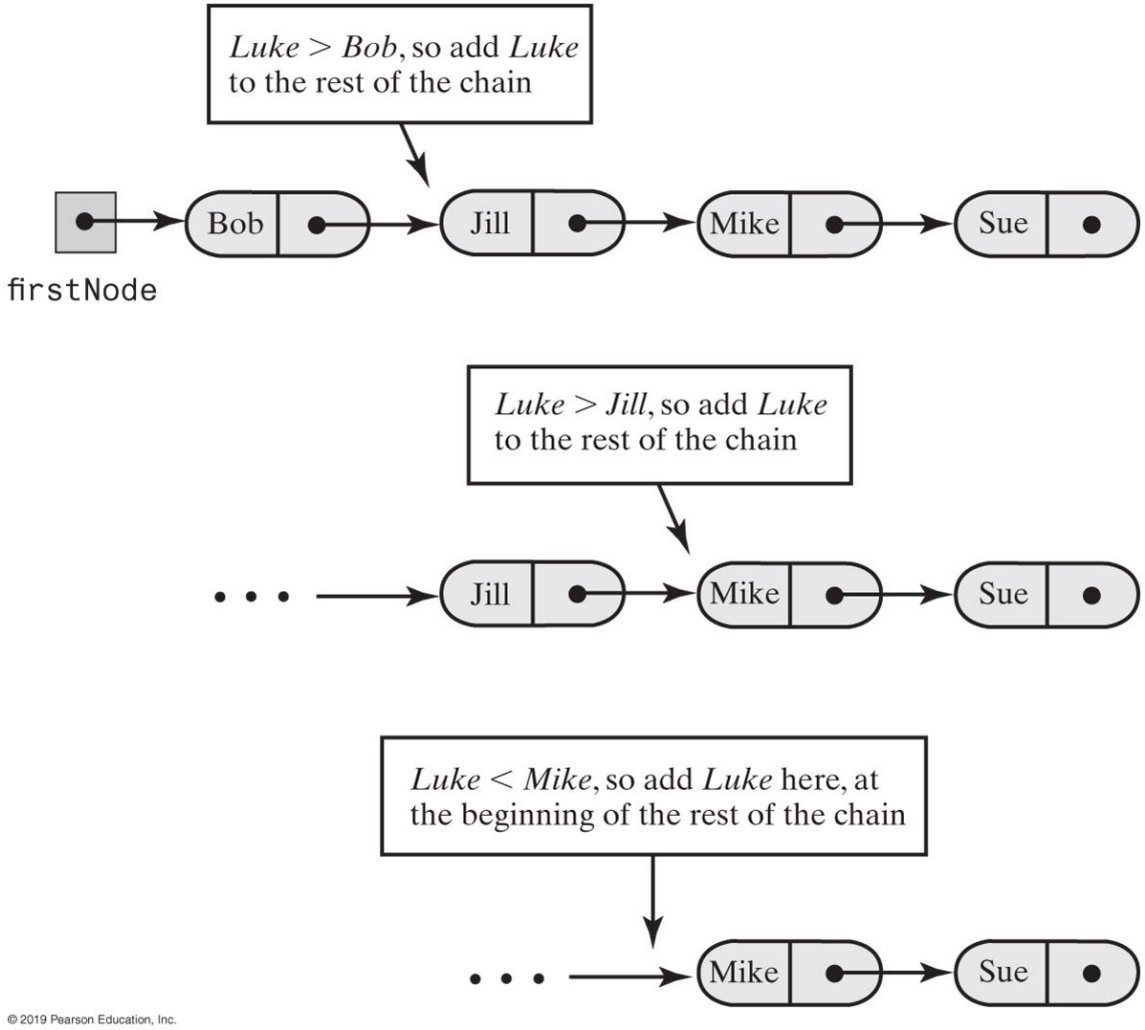


FIGURE 17-2 Recursively adding Luke to a sorted chain of names

Recursive Add to Sorted List (Part 1)

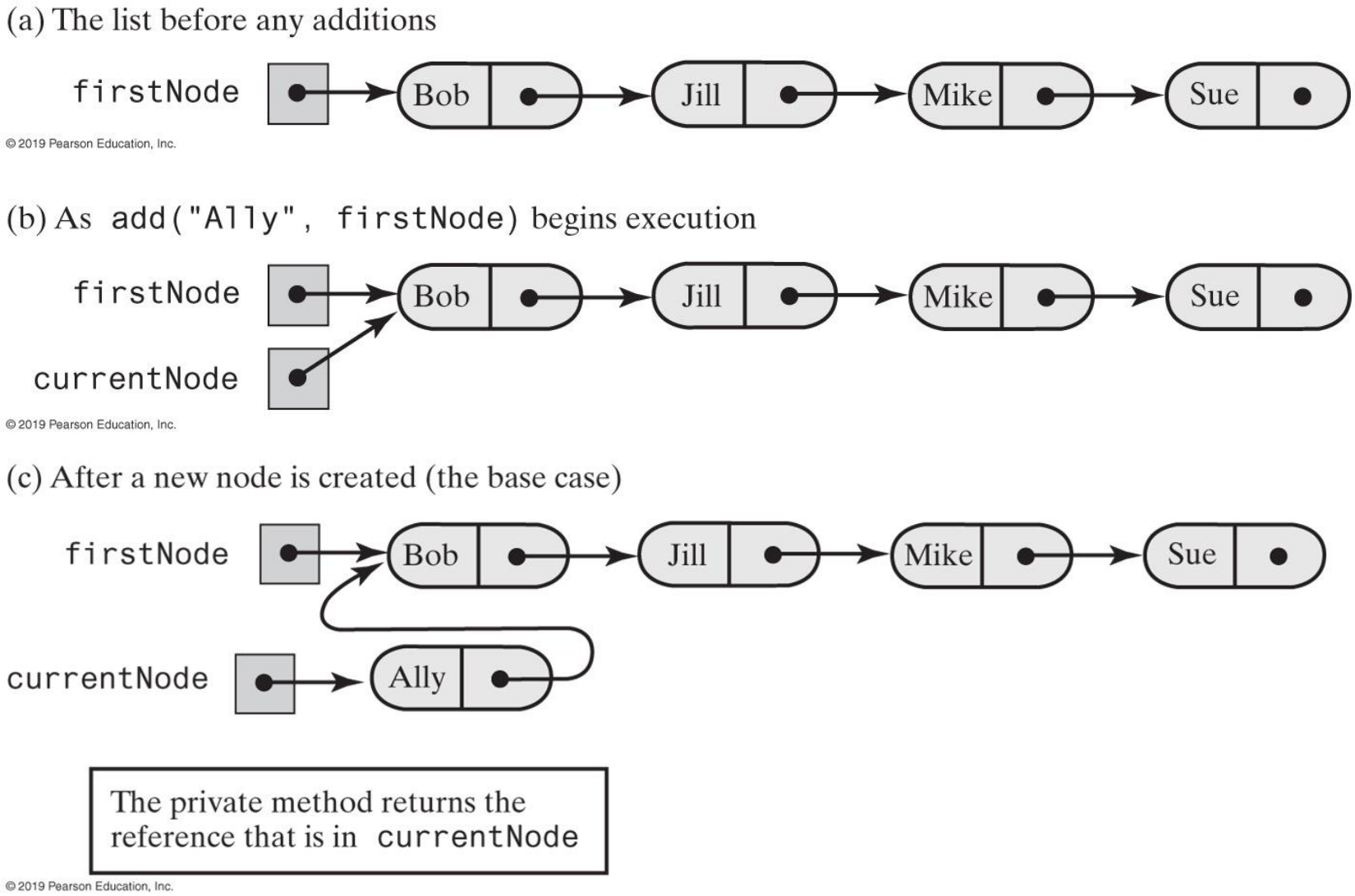
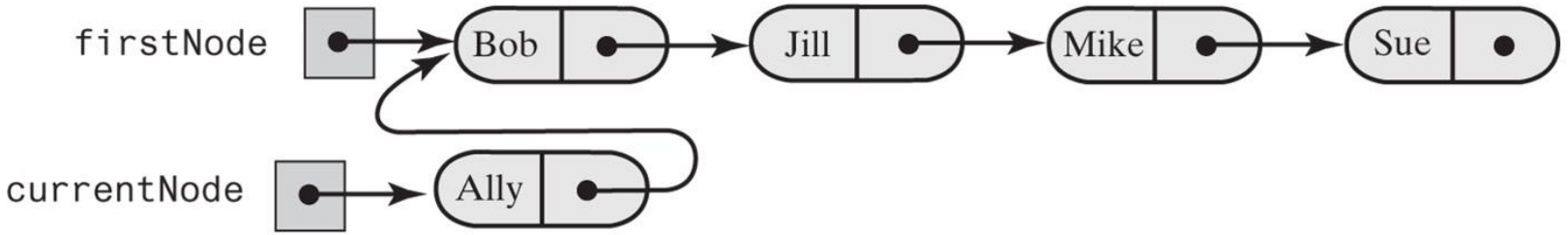


FIGURE 17-3 Recursively adding a node at the beginning of a chain

Recursive Add to Sorted List (Part 2)

(c) After a new node is created (the base case) *[from previous slide]*



(d) After the public `add` assigns the returned reference to `firstNode`

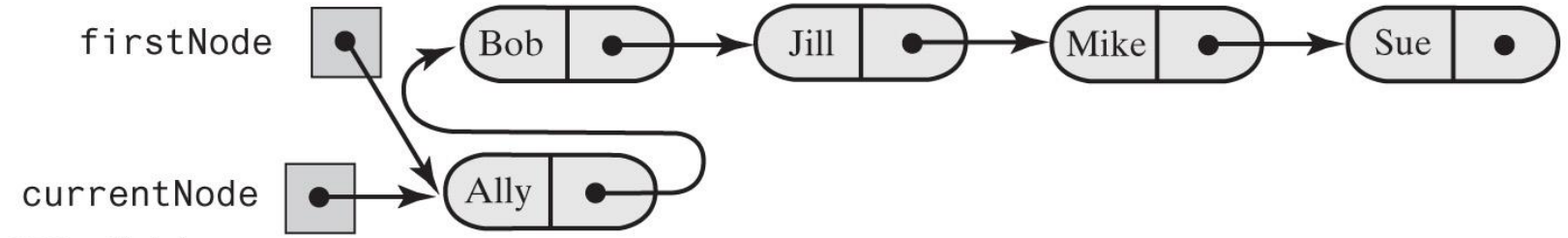


FIGURE 17-3 Recursively adding a node at the beginning of a chain

Recursive Add to Sorted List (Part 1)

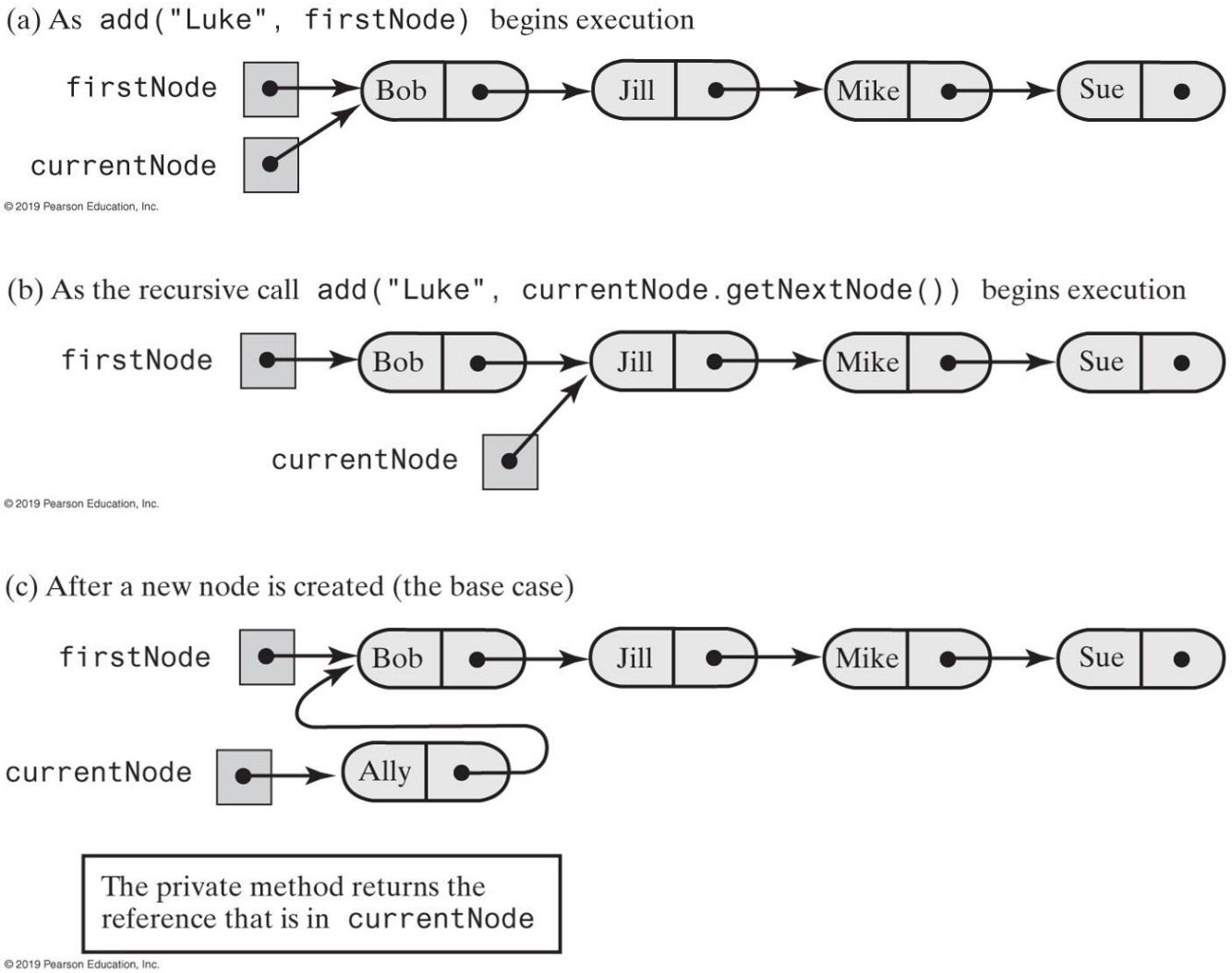
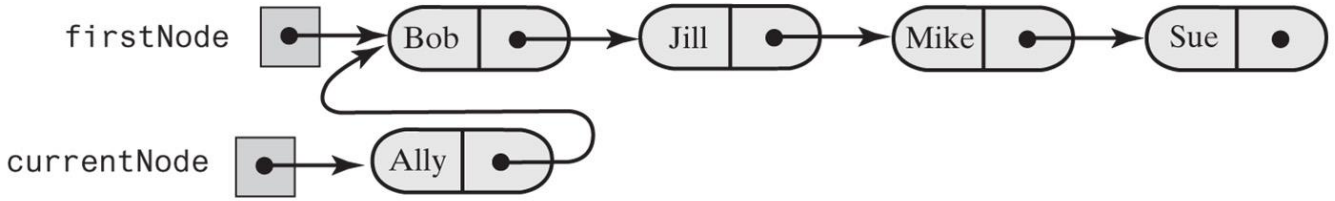


FIGURE 17-4 Recursively adding a node between existing nodes in a chain

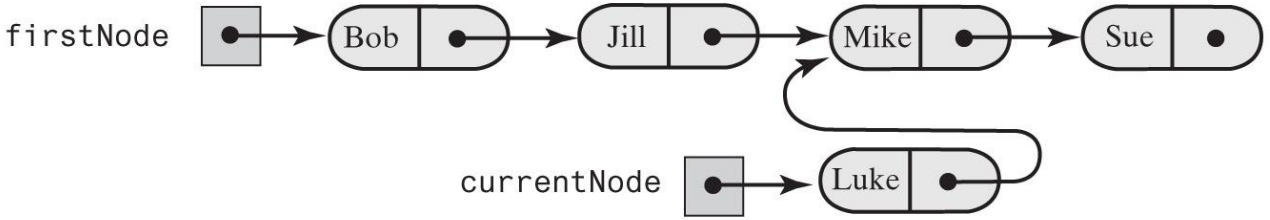
Recursive Add to Sorted List (Part 2)

(c) After a new node is created (the base case)



The private method returns the reference that is in `currentNode`

(d) After a new node is created (the base case)

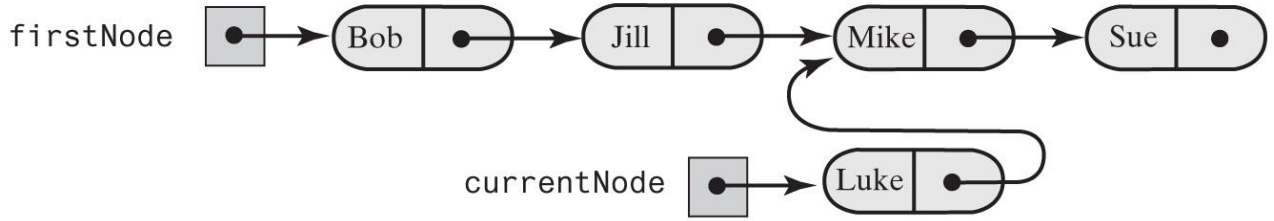


The private method returns the reference that is in `currentNode`

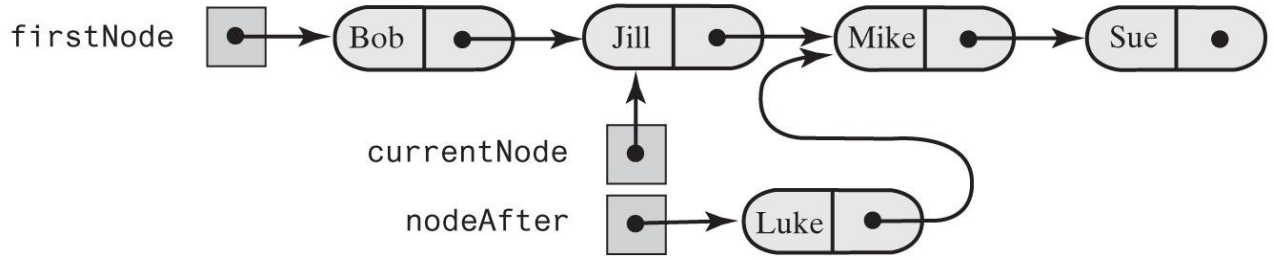
FIGURE 17-4 Recursively adding a node between existing nodes in a chain

Recursive Add to Sorted List (Part 2)

(d) After a new node is created (the base case)



(e) After the returned reference is assigned to `nodeAfter`



(f) After `currentNode.setNextNode(nodeAfter)` executes

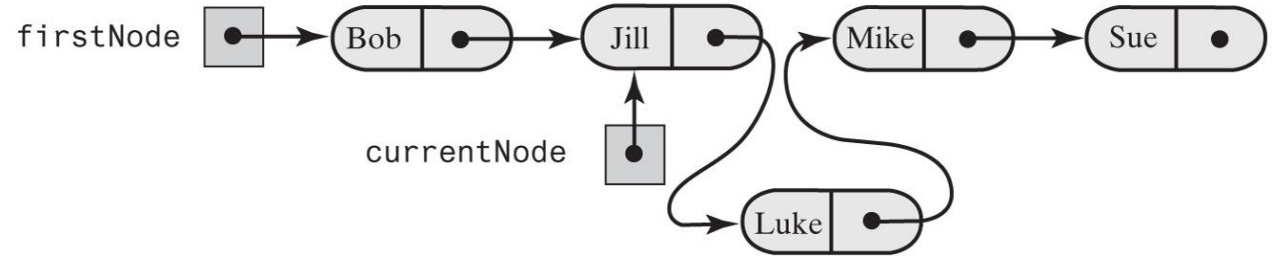


FIGURE 17-4 Recursively adding a node between existing nodes in a chain

Comparison of Implementations

Operation	Array	Linked
`add(newEntry)`	O(n)	O(n)
`remove(anEntry)`	O(n)	O(n)
`getPosition(anEntry)`	O(n)	O(n)
`getEntry(givenPosition)`	O(1)	O(n)
`contains(anEntry)`	O(n)	O(n)
`remove(givenPosition)`	O(n)	O(n)
`display()`	O(n)	O(n)
`clear(), getLength(), isEmpty()`	O(1)	O(1)

FIGURE 17-5 The worst-case efficiencies of the operations on the ADT sorted list for two implementations

Implementation that uses ADT List

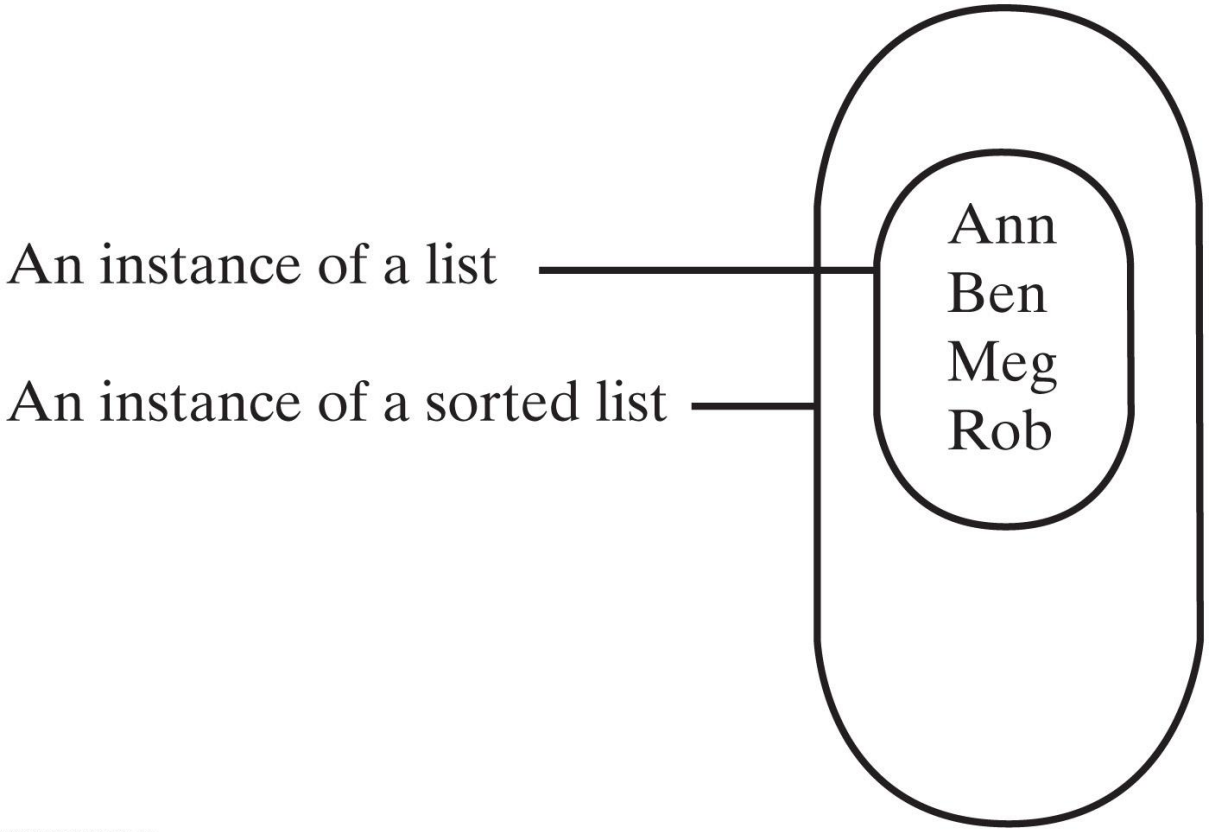


FIGURE 17-6 An instance of a sorted list that contains a list of its entries

Implementation That Uses the ADT List

```
public void add(T newEntry)
{
  int newPosition = Math.abs(getPosition(newEntry));
  list.add(newPosition, newEntry);
} // end add
```

The method add.

Implementation That Uses the ADT List

```
public boolean remove(T anEntry)
{
  boolean result = false;
  int position = getPosition(anEntry);

  if (position > 0)
  {
    list.remove(position);
    result = true;
  } // end if

  return result;
} // end remove
```

The method `remove`.

Implementation That Uses the ADT List

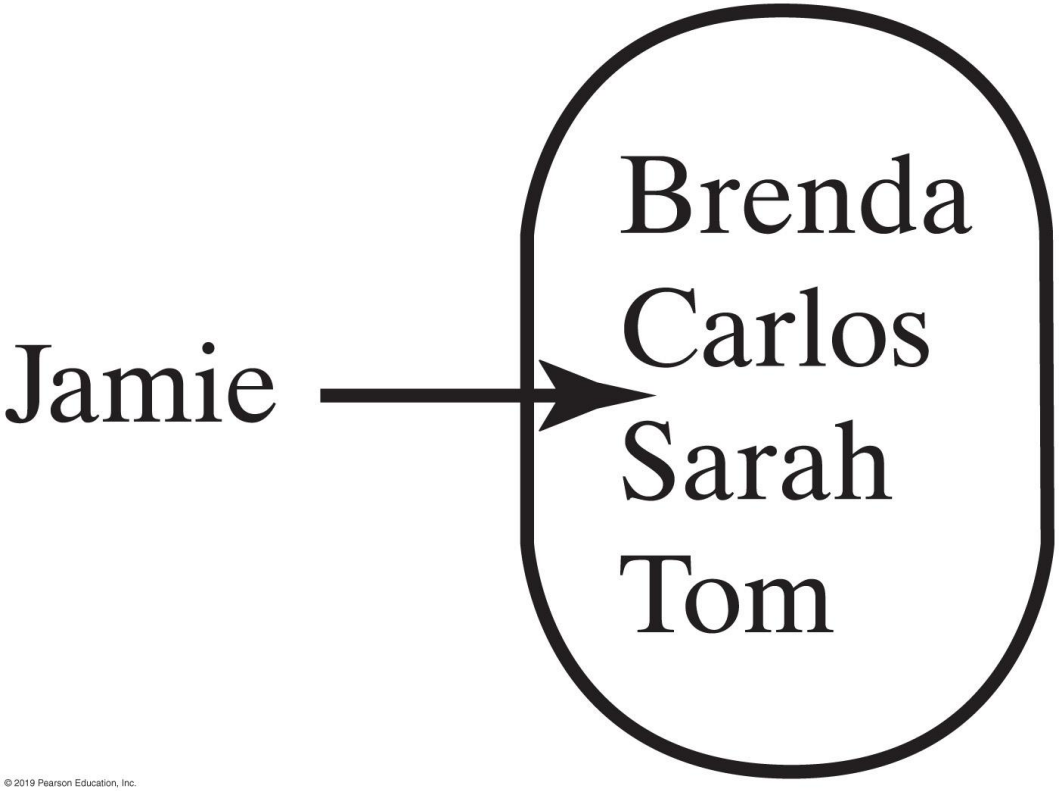


FIGURE 17-7 A sorted list in which Jamie belongs after Carlos but before Sarah

Comparison of Implementations

Operation	Array	Linked
`add(newEntry)`	O(*n*)	O(*n*)
`remove(anEntry)`	O(*n*)	O(*n*)
`getPosition(anEntry)`	O(*n*)	O(*n*)
`getEntry(givenPosition)`	O(1)	O(*n*)
`contains(anEntry)`	O(*n*)	O(*n*)
`remove(givenPosition)`	O(*n*)	O(*n*)
`display()`	O(*n*)	O(*n*)
`clear(), getLength(), isEmpty()`	O(1)	O(1)

FIGURE 17-8 The worst-case efficiencies of the operations on the ADT sorted list for two implementations

Comparison of Implementations

Operation	Array	Linked	ArraryList	LList
`add(newEntry)`	O(n)	O(n)	O(n)	O(n^2)
`remove(anEntry)`	O(n)	O(n)	O(n)	O(n^2)
`getPosition(anEntry)`	O(n)	O(n)	O(n)	O(n^2)
`getEntry(givenPosition)`	O(1)	O(n)	O(1)	O(n)
`contains(anEntry)`	O(n)	O(n)	O(n)	O(n)
`remove(givenPosition)`	O(n)	O(n)	O(n)	O(n)
`display()`	O(n)	O(n)	O(n)	O(n)
`clear(), getLength(), isEmpty()`	O(1)	O(1)	O(1)	O(1)

FIGURE 17-9 The worst-case efficiencies of the ADT sorted list operations when implemented using an instance of the ADT list

End

Chapter 17