Sorted Lists

Chapter 16

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

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
- Operations
  - **`add(newEntry)`**
  - **`remove(anEntry)`**
  - **`getPosition(anEntry)`**

Specifications for ADT Sorted List

- Additional Operations -- behave as they do for the ADT list
  - **`getEntry(givenPosition)`**
  - **`contains(anEntry)`**
  - **`remove(givenPosition)`**
  - **`clear()`**
  - **`getLength()`**
  - **`isEmpty()`**
  - **`toArray()`**

Specifications for ADT Sorted List

```
1  /** An interface for the ADT sorted list.
2      Entries in the list have positions that begin with 1.
3      @author Frank M. Carrano
4  /*
5  public interface SortedListInterface<T extends Comparable<? super T>>
6  {
7     /** Adds a new entry to this sorted list in its proper order.
8         The list's size is increased by 1.
9         @param newEntry  The object to be added as a new entry. */
10    public void add(T newEntry);
11
12    /** Removes the first or only occurrence of a specified entry
13        from this sorted list.
14        @param anEntry  The object to be removed.
15        @return  True if anEntry was located and removed; */
16                 otherwise returns false. */
17    public boolean remove(T anEntry);
18
19    /** Gets the position of an entry in this sorted list.
20        @param anEntry  The object to be found.
21        @return  The position of the first or only occurrence of anEntry
```


LISTING 16-1 The interface **`SortedListInterface`**

Specifications for ADT Sorted List

```
17    public boolean remove(T anEntry);
18
19    /** Gets the position of an entry in this sorted list.
20        @param anEntry  The object to be found.
21        @return  The position of the first or only occurrence of anEntry
22                 if it occurs in the list; otherwise returns the position
23                 where anEntry would occur in the list, but as a negative
24                 integer. */
25    public int getPosition(T anEntry);
26
27    // The following methods are described in Segment 12.9 of Chapter 12
28    // as part of the ADT list:
29
30    public T getEntry(int givenPosition);
31    public boolean contains(T anEntry);
32    public T remove(int givenPosition);
33    public void clear();
34    public int getLength();
35    public boolean isEmpty();
36    public T[] toArray();
37 } // end SortedListInterface
```


LISTING 16-1 The interface **`SortedListInterface`**

Linked Implementation

```
1  public class LinkedSortedList<T extends Comparable<? super T>>
2                implements SortedListInterface<T>
3  {
4     private Node firstNode; // Reference to first node of chain
5     private int  numberOfEntries;
6
7     public LinkedSortedList()
8     {
9        firstNode = null;
10       numberOfEntries = 0;
11    } // end default constructor
12
13    < Implementations of the sorted list operations go here.>
14    . . .
15    private class Node
16    {
17       private T    data;
18       private Node next;
19       < Constructors >
20       . . .
21       < Accessor and mutator methods: getData, setData, getNextNode, setNextNode >
22       . . .
23    } // end Node
24 } // end LinkedSortedList
```

Linked Implementation

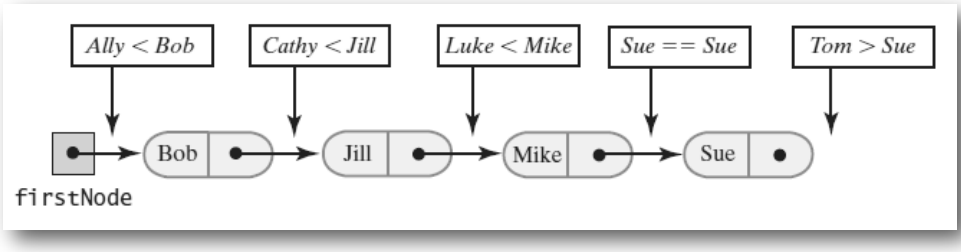

FIGURE 16-1 Places to insert names into a sorted chain of linked nodes

Linked 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 routine.

Linked Implementation

```
public void add(T newEntry)
{
   Node newNode = new Node(newEntry);
   Node nodeBefore = getNodeBefore(newEntry);

   if (isEmpty() || (nodeBefore == null))
   {
      // Add at beginning
      newNode.setNextNode(firstNode);
      firstNode = newNode;
   }
   else
   {
      // Add after nodeBefore
      Node nodeAfter = nodeBefore.getNextNode();
      newNode.setNextNode(nodeAfter);
      nodeBefore.setNextNode(newNode);
   } // end if

   numberOfEntries++;
} // end add
```


An iterative implementation of `add`

Linked Implementation

```
// Finds the node that is before the node that should or does
// contain a given entry.
// Returns either a reference to the node that is before the node
// that does or should contain anEntry, or null if no prior node exists
// (that is, if anEntry is or belongs at the beginning of the list).
private Node getNodeBefore(T anEntry)
{
   Node currentNode = firstNode;
   Node nodeBefore = null;

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

   return nodeBefore;
} // end getNodeBefore
```

The private method **`getNodeBefore`**

Recursive Linked Implementation

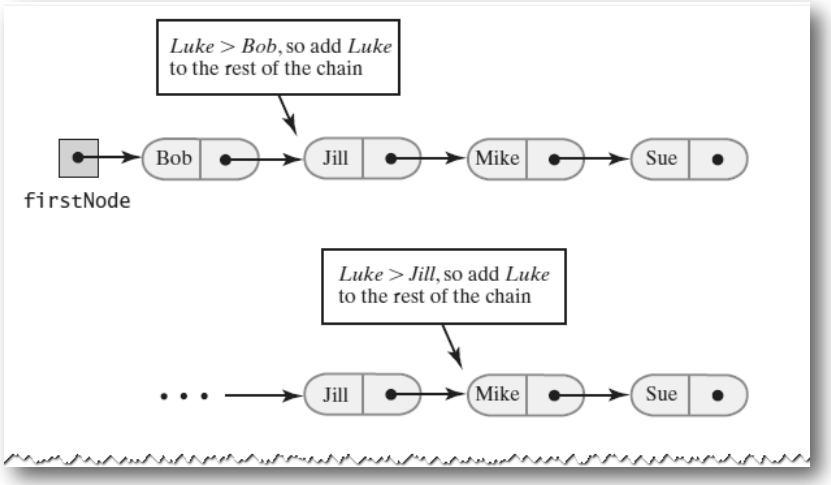

FIGURE 16-2 Recursively adding *Luke* to a sorted chain of names

Recursive Linked Implementation

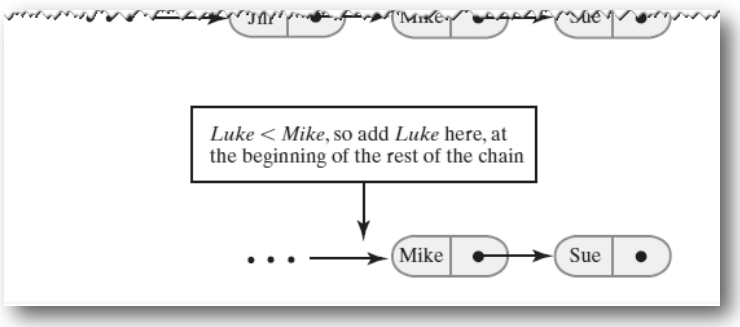

FIGURE 16-2 Recursively adding *Luke* to a sorted chain of names


```
public void add(T newEntry)
{
   firstNode = add(newEntry, firstNode);
   numberOfEntries++;
} // end add

private Node add(T newEntry, Node currentNode)
{
   if ( (currentNode == null) ||
        (newEntry.compareTo(currentNode.getData()) <= 0) )
   {
      currentNode = new Node(newEntry, currentNode);
   }
   else
   {
      Node nodeAfter = add(newEntry, currentNode.getNextNode());
      currentNode.setNextNode(nodeAfter);
   } // end if
   return currentNode;
} // end add
```

A recursive implementation of **`add`**

Recursive Linked Implementation

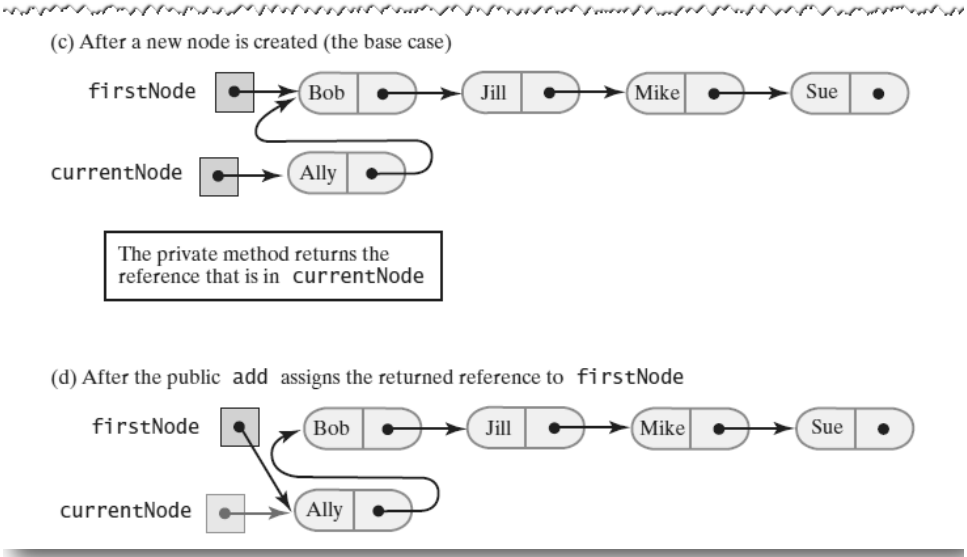

FIGURE 16-3 Recursively adding a node at the beginning of a chain

Recursive Linked Implementation

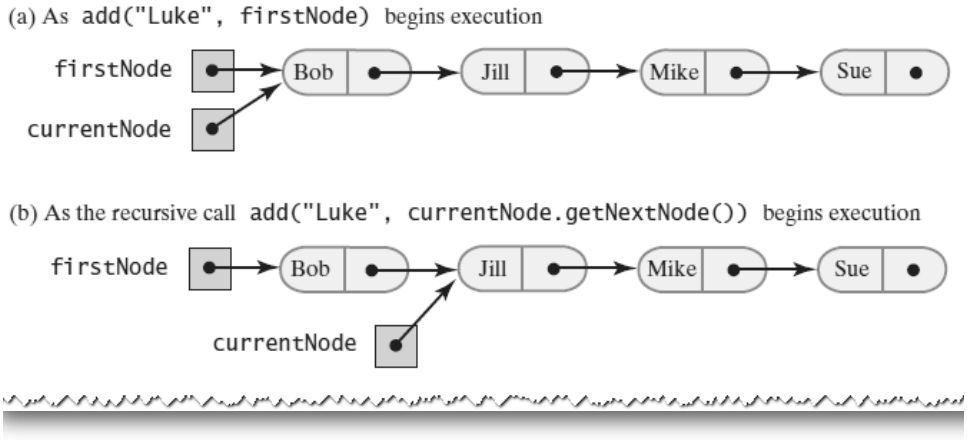

FIGURE 16-4 Recursively adding a node between existing nodes in a chain

Recursive Linked Implementation

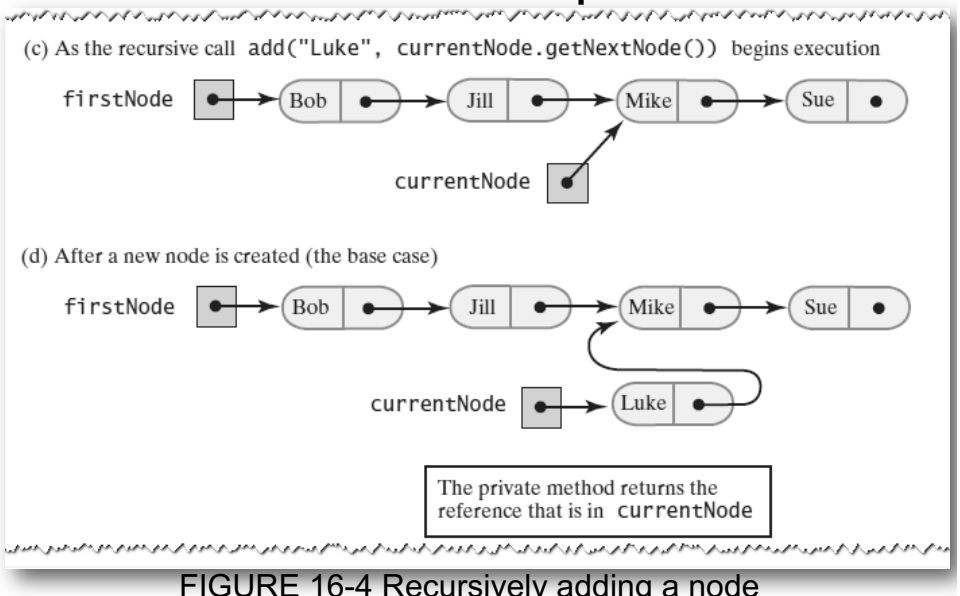

FIGURE 16-4 Recursively adding a node between existing nodes in a chain

Recursive Linked Implementation

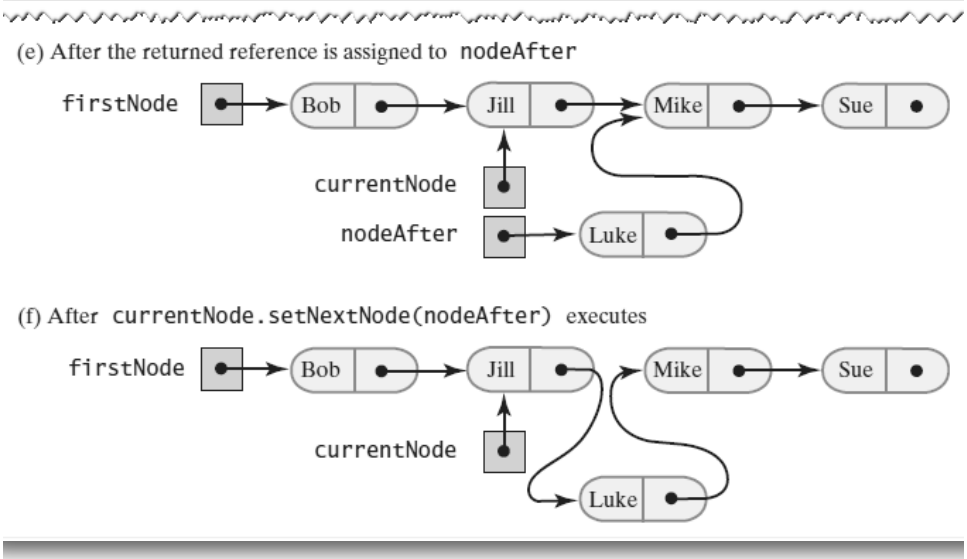

FIGURE 16-4 Recursively adding a node between existing nodes in a chain

Implementation That Uses the ADT List

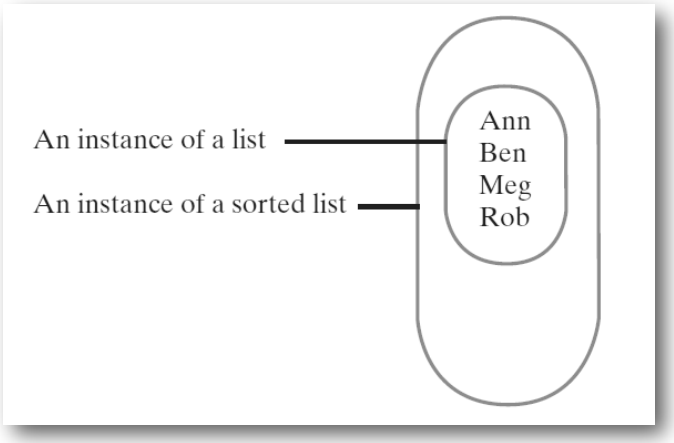

FIGURE 16-6 An instance of a sorted list that contains a list of its entries

Implementation That Uses the ADT List

```
public class SortedList<T extends Comparable<? super T>>
             implements SortedListInterface<T>
{
   private ListInterface<T> list;
   public SortedList()
   {
      list = new LList<>();
   } // end default constructor
      . . .
} // end SortedList
```

Our class **`SortedList`** will implement the interface **`SortedListInterface`**

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

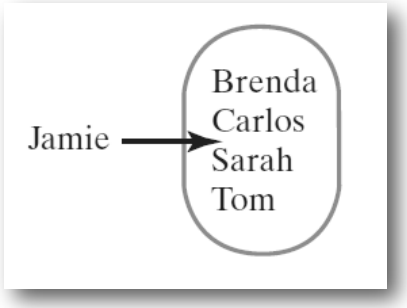

FIGURE 16-7 A sorted list in which *Jamie* belongs after *Carlos* but before *Sarah*

Implementation That Uses the ADT List

```
public int getPosition(T anEntry)
{
   int position = 1;
   int length = list.getLength();

   // Find position of anEntry
   while ( (position <= length) &&
           (anEntry.compareTo(list.getEntry(position)) > 0) )
   {
      position++;
   } // end while

   // See whether anEntry is in list
   if ( (position > length) ||
        (anEntry.compareTo(list.getEntry(position)) != 0) )
   {
      position = -position; // anEntry is not in list
   } // end if

   return position;
} // end getPosition
```

End

Chapter 16