


Linked Implementation of Sorted List

Linked Implementation of Sorted List

```
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
```



LISTING 16-2 An outline of a linked implementation of the ADT sorted list

Linked Implementation

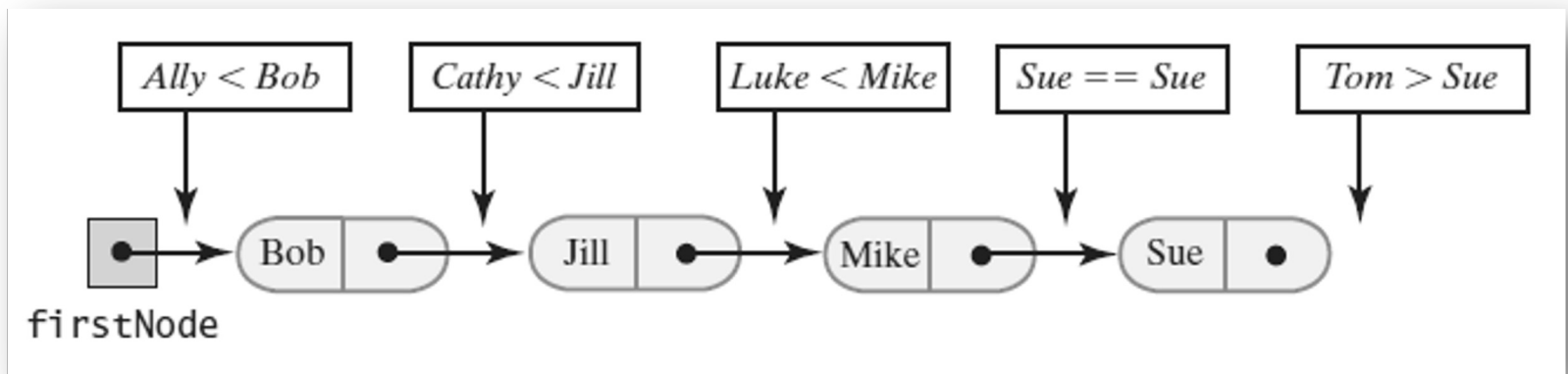


FIGURE 16-1 Places to insert names into a sorted chain of linked nodes

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.

Two approaches to Linked Implementations

- Iterative Linked Implementation
- Recursive Linked Implementation

Iterative 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**

Iterative 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

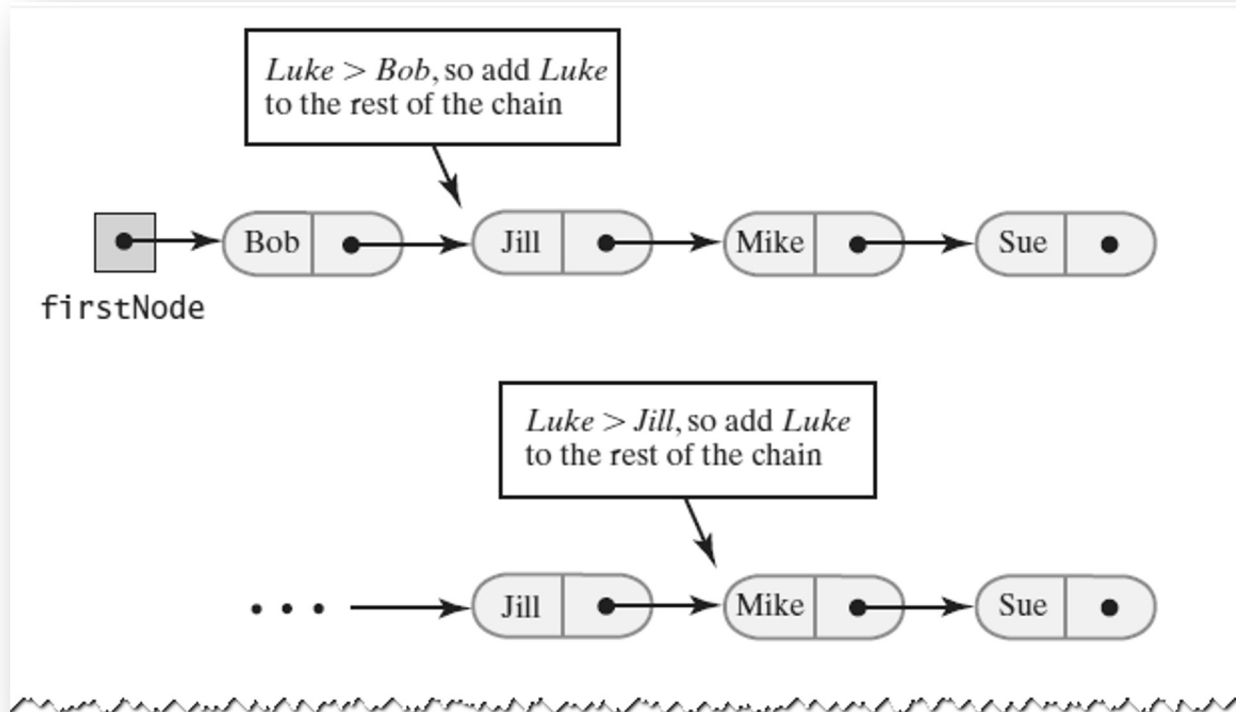


FIGURE 16-2 Recursively adding *Luke* to a sorted chain of names

Recursive Linked Implementation

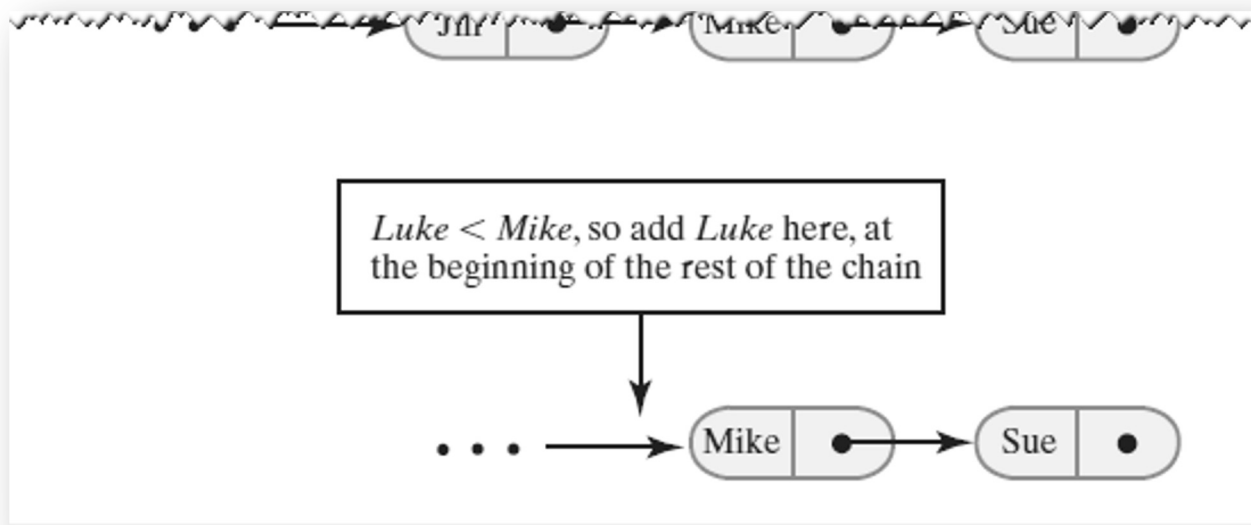


FIGURE 16-2 Recursively adding *Luke* to a sorted chain of names