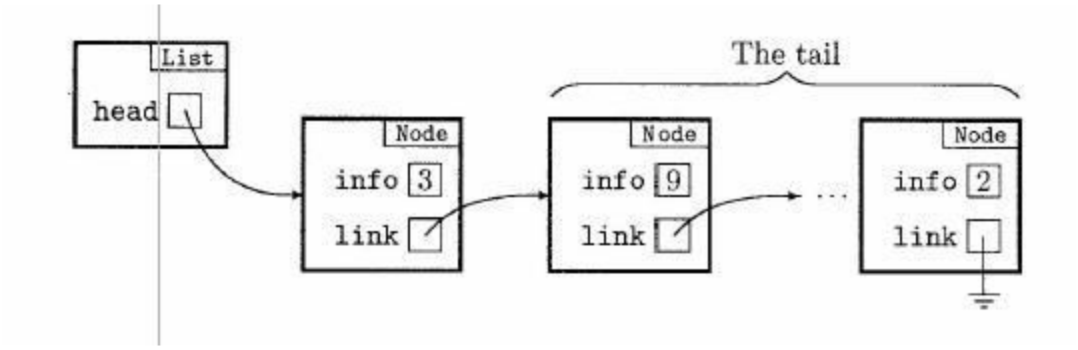
12.4 Recursive List Processing

You may recall that every recursive process has two parts: a case in which the process is defined in terms of a simpler version of itself and a simple case that terminates the recursion. It is often useful to think of *structures* in a similar way. To do so with linked lists, we can think of a linked list as a structure that is either empty (the simple case) or consists of a node followed by a linked list - the tail of the original linked list.



Recursive algorithms for list processing reflect this structure by processing the node referred to by the head directly and processing the tail recursively with an empty list acting as a stopping condition.

Example 1

If a linked list has nodes consisting of `info` and `link` fields, then, to print the contents of the `info` fields, we can use an algorithm of the following form.

```
To PRINT THE LIST
   if the list is not empty
      print the info field of the first node
      PRINT THE LIST that forms the tail
```

Using this outline, we can write the method in Java. As a first attempt, we could try the following method in the List class:

```
public void printList ()
{
   // a first attempt
   if (head != null)
   {
      System.out.println(head.info);
         head.link.printList();
   }
}
```

This appears to follow the algorithm but it doesn't work! The problem is that, since `printList` is an instance method of the class `List`, it must be called with an implicit `List` object. Unfortunately, the statement

```
head.link.printList();
```
attempts to make a call to `printList` with the object `head.link`, which is of type `Node`. There are a number of ways to get around this problem. The one that we have chosen is to add a version of `printList` to our `Node` class. In the `List` class, we use a non-recursive method that checks to see if the list is not empty and, if it is, the method then calls the recursive helper version in the `Node` class. That method is guaranteed to have a non-empty list so its algorithm is:
```
To PRINT A NON-EMPTYLIST
   print the info field of the first node
   if the tail is not empty
      PRINT THE NON-EMPTYLIST that forms the tail
```
The results are shown in the following example.

Example 2

The printList methods shown here illustrate correct recursive printing of a linked list.
```
class List
{
   private Node head;
   public void printList ()
   {
      if (head != null)
         head.printList();
   }

   class Node // an inner class
   {
      int info;
      Node link;

      Node (int i, Node n)
      {
         info = i;
         link = n;
      }

      void printList ()
      {
         System.out.println(info);
         if (link != null)
            link.printList();
      }
   }
}
```

Using recursion can simplify the structure of many methods. To illustrate this, consider the problem of inserting a new node in a list in which the `item` fields are in ascending order (with smallest values at the front of the list). We solved this problem in Example 1 of Section 12.2 by using two auxiliary references to the nodes of the list. Recursion allows us to look at the problem in a much different and simpler way.

Example 3

This is a recursive algorithm for insertion of a node containing `item` in the `info` field in an ordered list.

```
To INSERT IN THE LIST
   if the list is empty or item < first node's info field
      insert a node containing item here
   else
      INSERT IN THE LIST referred to by the first node
```

Once again, we will not use the algorithm in this simple form. Instead, we will create two versions of the insertion method - one for the `List` class and one for the inner `Node` class. The details are given in the next example.

Example 4

This non-recursive method, for the `List` class, will start the process of insertion of a node in an ordered list. The method calls the recursive version only if the new item should be inserted somewhere *after* the first node; otherwise, it does the insertion itself.

```
public void insert (int item)
{
   if (head == null || item < head.info)
      // insert item as first node of original list
      head = new Node(item,head);
   else
      // call recursive version to
      // insert item in tail of non-empty list
      head.insert(item);
}
```

This recursive helper method, for the `Node` class, will insert `item` in its correct position in the tail of a non-empty ordered list. It should never be called if the new item is to be the first node in the list.

```
void insert (int item)
{
   if (link == null || item < link.info)
      // insert item right after first node
      link = new Node(item,link);
   else
      // insert after first node of non-empty tail
      link.insert(item);
}
```

Exercise 12.4

1. In Example 2, in the `printList` method (in the `Node` class), what would be the effect

of placing the `println` statement *after* the statement that calls `printList`?

The effect of placing the `println` statement *after* the statement that calls `printList` is that the List gets printer backwards.

```
        /*
        * This method prints the list BACKWARDS
        * pre: none
        * post: none
        */
       void printList ()
       {
         if (link != null)
             link.printList();
         System.out.println(info);

       }
```

2. In Example 4, if we had written the first `if` statement in the form

```
if (item < head.info || head == null) ...
```

then the method would not work correctly. Why?

This statement will eventually result in a java.lang.NullPointerException error when `head = null,` as the statement tries to compare the `info` field of a Node that hasn't been initialized yet.

A general way to avoid such problems is to place the condition that is "most likely to be true" on the left. However, this may not always be the case.

```
   /*
    * This method uses recursion to insert items into the list in
ascending order
    * pre:none
    * post: returns java.lang.NullPointerException error
    */
   public void insert (int item)
   {
      if (item < head.info || head == null)         // insert item as
first node of original list
         head = new Node(item,head);
      else
         // call recursive version to
         // insert item in tail of non-empty list
         head.insert(item);

   }
```

3. Complete the following outline of the algorithm used by the recursive version of `insert` given in the text:

```
   To INSERT IN THE TAIL OF A NON-EMPTY LIST ...
```

```
To INSERT IN THE TAIL OF A NON-EMPTY LIST
   If head is empty
      Insert the item in head
   Else
      Pass item into insert method until head.next is empty, once
the head.next is empty, add item into head.next.
```

LIST CLASS:

```
   /*
    * This method uses recursion to insert items in the tail of a non-
empty list
    * pre: none
    * post: none
   */
   public void insert (int item)
   {
      if (head == null)
         head = new Node (item, null);
      else {
         head.insert(item);
      }
   }
```

NODE CLASS

```
      /*
       * This method inserts items in the tail of a non-empty list
       * pre: none
       * post: none
      */
      void insert (int item)
      {
         if (link == null)
            // insert item right after first node
            link = new Node(item,null);
         else
            // insert after first node of non-empty tail
            link.insert(item);
      }
```

4. Write a class `List` that implements the `printList` and `insert` methods discussed in this section. Test your methods by writing a `main` method that repeatedly prompts a user for non-negative integers using zero as a sentinel to stop input. The non-zero values should be inserted, in ascending order, into a linked list. Once the list has been

created, its contents should be printed.

```
import java.util.Scanner;
public class Question4 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Scanner input = new Scanner (System.in);
		List4 list = new List4 ();
		int integer;
		do {
			System.out.println("Enter a non-negative integer:");
			integer = input.nextInt();
			if (integer < 0)
				System.out.println("Invalid integer ! Please 
try again.");
			else
				list.insert(integer);
		}
		while (integer != 0);
		list.printList();
	}

}

class List4
{
   private Node head;

  /*
   * This method uses recursion to print the List
   * pre: none
   * post:none
   */
   public void printList ()
   {
      if (head != null)
         head.printList();
   }
   /*
    * This method uses recursion to insert items into the list in 
ascending order
    * pre:none
    * post: none
    */
   public void insert (int item)
   {
      if (head == null || item < head.info)
         // insert item as first node of original list
         head = new Node(item,head);
      else
```

```
          // call recursive version to
          // insert item in tail of non-empty list
          head.insert(item);
    }

    class Node // an inner class
    {
       int info;
       Node link;

       Node (int i, Node n)
       {
          info = i;
          link = n;
       }

       /*
        * This method prints the list
        * pre: none
        * post: none
        */
       void printList ()
       {
         System.out.println(info);
         if (link != null)
             link.printList();
       }

       /*
        * This method inserts items into the List in ascending order
        * pre: none
        * post: none
        */
       void insert (int item)
       {
          if (link == null || item < link.info)
             // insert item right after first node
             link = new Node(item,link);
          else
             // insert after first node of non-empty tail
             link.insert(item);
       }
    }
}
```

5. Write a pair of methods, similar in structure to the ones shown in this section, to delete the last node in a linked list.

LIST CLASS
```
/*
 * This method deletes the last node in the List
 * pre: none
 * post: none
*/
public void delete () {
        if (head == null)
                System.out.println("List is already empty");
        else if (head.link == null)
                head = null;
        else
                head.delete();

}
```

NODE CLASS:
```
/*
 * This method uses recursion to delete the last node in the List
 * pre: none
 * post: none
*/
void delete () {
   if (link.link == null)
           link = null;
   else
           link.delete();

}
```

6. We can use recursion to make a copy of a linked list. To do so, we could start by writing the method shown here for the `List` class. It creates a new `List` object and then, if necessary, calls a recursive helper method in the `Node` class to create the required nodes for the copy.

```
public List copy ()
{
   List other = new List();
   if (head != null)
      other.head = head.copy();
      return other;
}
```

Complete the solution by writing the recursive `copy` method for the `Node` class.

NODE CLASS

```
     /*
       * This method uses recursion to copy the item of this List to
another list
       * pre: none
       * post: returns Node
      */
      Node copy () {
        if (this != null)
              return this;
        else
              return null;

      }
```