12.1 Linear Lists and Linked Lists

As a first example of a dynamic data structure we will examine a new way of storing lists of items. It is certainly possible to implement a list using an array but, for many applications, arrays are inconvenient. If, for example, we want to keep the items in order based on some key, then insertion or deletion of an item might require us to move many other elements. Because such problems arise frequently, list manipulation is one of the most common areas for use of dynamic data structures. Before we look at new ways of storing and manipulating lists in a computer, however, it will be useful to take a closer look at what we mean by a list, without reference to a computer.

A general definition of a linear list might say that it is a sequence of nodes together with a set of operations on those nodes. The undefined term "node" can be thought of as something carrying some information; it might be a person's health record, a playing card, or simply an integer. The essential feature of a linear list is the fact that it is a sequence of nodes. If there are n 2': a nodes in the list, we can write the sequence as Xl, X2, ... , Xn where Xl is the first node, X2 is the second, and so on, with Xn being the last node. The operations that we might want to perform on a list will vary from one situation to another. They might include:

• examine the contents of the node at one end of the list

• insert or delete a node at one end of the list

• insert a new node before or after a node containing some key

• delete the kth node

• alter the contents of the kth node

• search for a node that contains some value

• sort the nodes of the list based on some key

These ideas about linear lists do not make any reference to the way that a list might be stored in a computer; they are concerned only with the abstract concept of a list. Because of this, we say that such a definition defines an abstract data type or ADT. In general, an ADT is a collection of data together with a set of operations that can be performed on the data.

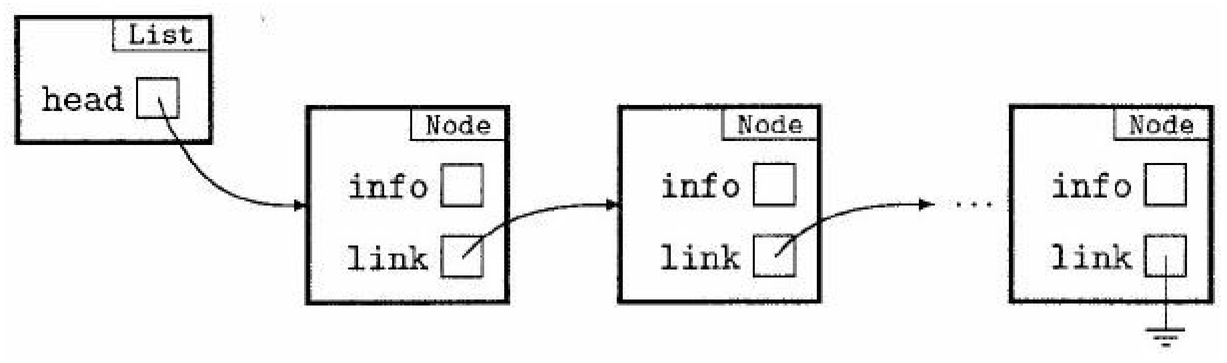
Example 1

```
If we wanted a list to be used by air traffic controllers keeping track of
planes waiting for takeoff, we might define the required linear list ADT
as having nodes that contain information about planes. The operations
```

```
that would be required might be: create a new, empty list (at the start of
the day), insert a new node at the rear of the list (as a plane leaves the
terminal), delete a node from the front of the list (as a plane takes off),
and check to see if there are any nodes in the list (if a runway becomes
available).
```

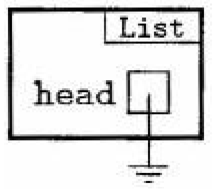
Once we have defined an ADT that is appropriate for a particular problem, we can then look at the best way of implementing it. For a linear list, an obvious way to implement it is to use an array and we have already seen many instances in which we used arrays for this purpose. As we noted above, however, there are many applications for which arrays are not convenient for list implementation.

An alternative that overcomes many of the problems associated with arrays is the use of linked lists as implementations of linear lists. In a linked list, each node contains not only the information required for the list but also a reference to the next node in the list. The structure of a linked list is shown in the next diagram.



As the diagram implies, we use (at least) two kinds of objects to create linked lists in Java: a single object of type List and a sequence of objects of type Node. The List object contains a reference to the first Node and each Node except the last contains a reference to the next Node in the list. In addition, the info fields shown in the diagram may also be references to objects containing the information carried by the list.

As usual, the electrical ground symbol on the last node indicates a null reference. An empty list has the form:



To define such a structure in Java, we can start with the following:

```
class List
```

```
{
  private Node head;
}
```

We want head to be private so that it cannot be manipulated directly from outside the List class. Notice that the type of head is Node because it will be used to refer to objects of type Node.

We can start to define the Node class by defining structures for the data that we are going to store (shown by the info fields in the diagram) and a reference to the next node in the list (shown by the link fields in the diagram). The info field will; in practice, usually be a reference to an object that actually stores the information but, for the purpose of demonstrating the structure of a linked list, we will simply store integers in these fields.

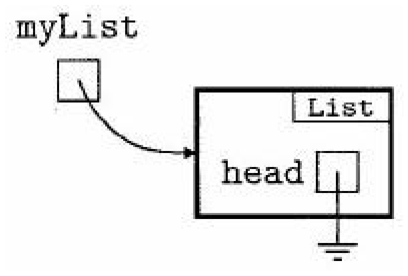
We have a bit of a problem in deciding whether the fields of the Node class should be public or private. If we declare them to be private, then the head field will not be able to access them directly. On the other hand, if we make them public, then they can be seen and modified by anybody. Java provides a nice solution to this problem through the use of inner classes which are, as the name implies, classes defined within other classes. Such classes are only visible from within the class in which they are defined; fields and methods of an inner class are only visible from within the inner class or its containing class. In the following code, we have added an inner Node class to our List class. The inner class has a constructor that permits us to create nodes with given info and link fields.

```
class List
{
  private Node head;

  class Node
  {
    int info;
    Node link;

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

Having decided on the structure of a linked list, how do we actually create one? We can create a new, empty list with its head field initialized to null by using the default constructor of the List class. To create an empty list called myList, we could write:

```
List myList = new List();
```

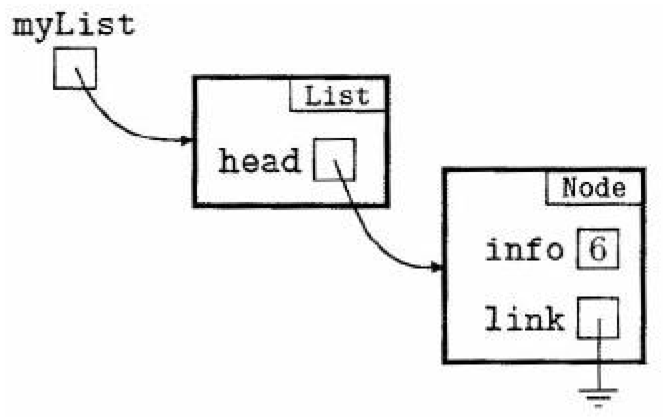
The result of this statement is illustrated in the next diagram.



If we now wanted to insert a single node containing the value item into this empty list, we could write the following method for the List class.

```
public void insertFirst (int item)
{
  head = new Node(item,null);
}
```

To execute the method, Java would first evaluate the right side of the assignment. This would cause a new node to be constructed with the info field having the value 6 and the link field having the value null. The assignment would then alter head so that it would refer to the new node as shown in the next diagram.



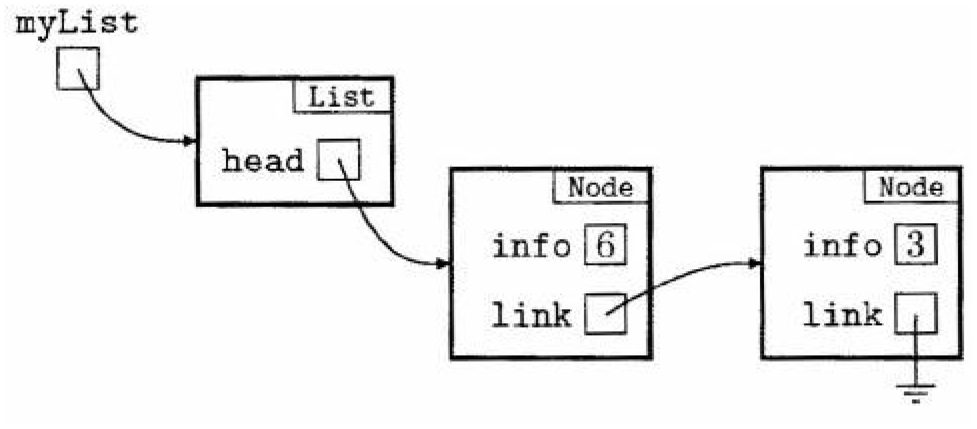
This process could be continued with a method insertSecond that inserts a second node in a list. Such a method would have to create a new node and have the link field of the existing node refer to it. The existing node itself is referred to by head so the link field of this node has the identifier head. link. The method could take the following form:

```
public void insertSecond (int item)
{
  head.link = new Node(item,null);
}
```

If we had created the structure shown in the previous diagram and then executed the statement:

```
myList.insertSecond(3);
```

The result would be the following structure in which a new node has been created and head. link, the link field from the first node, has been set to refer to this new node.



To be able to add a third node to the list, we could write another method that would create a new node and attach it to the link field of the second node. The second node is referred to by head. link, the link field of the first node. Thus, the link field of the second node has the identifier head . link . link. Using this, we could then create the method insert- Third:

```
public void insertThird (int item)
{
  head.link.link = new Node(item,null);
}
```

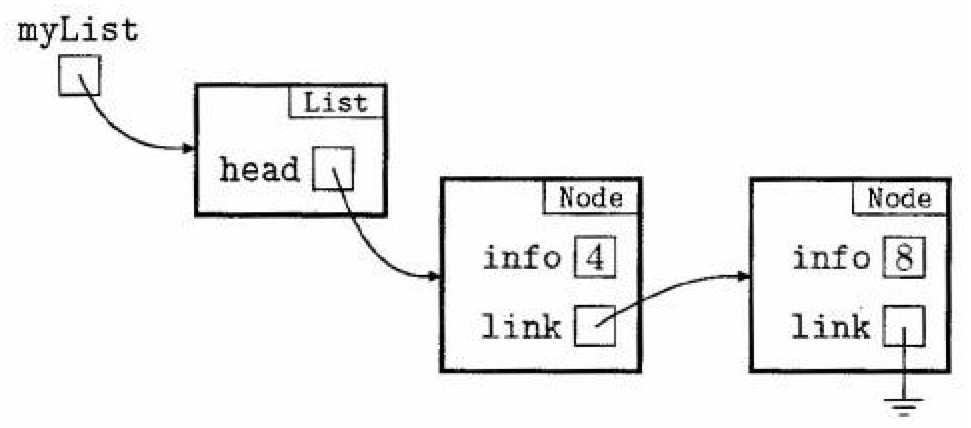
This, obviously, is not a great way to proceed. We want some process that is general and does not involve expressions that get increasingly long as a list grows. There are a number of solutions to this problem. One of them is to always add nodes at the beginning of the list rather than the end. You will be asked to examine another approach in the exercises. The next example shows a method, for the List class, that could be used repeatedly to build a linked list, one node at a time, by adding items at the front of the list.

Example 2

```
This method adds a node at the front of a linked list.
```

```
public void addAtFront (int item)
{
  head = new Node(item,head);
}
```

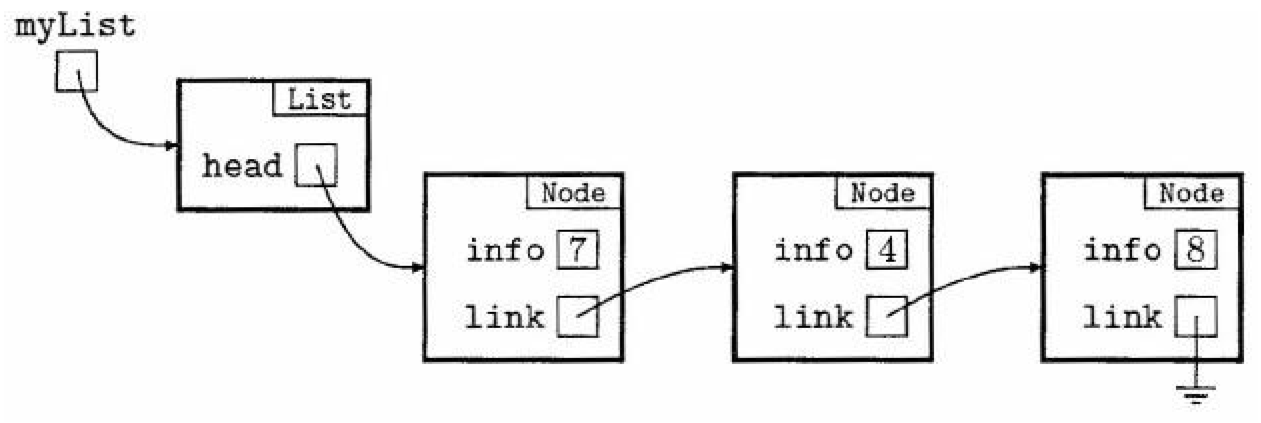
To execute the method, Java evaluates the right side of the assignment, causing a **new** node to be constructed with the link field referring to the same place that head refers the front of the list. The assignment then alters head so that it now refers to the **new** node. If, before the method was called, a list had the form:



Then, after a call of the form:

```
myList.addAtFront(7);
```

The list would become:



For many operations on linked lists, we need to "walk" through the structure, node by node, a process known as a traversal of a list. To do this, we can use a reference variable that moves along the list, from node to node. To start, we might write:

```
Node temp = head;
```

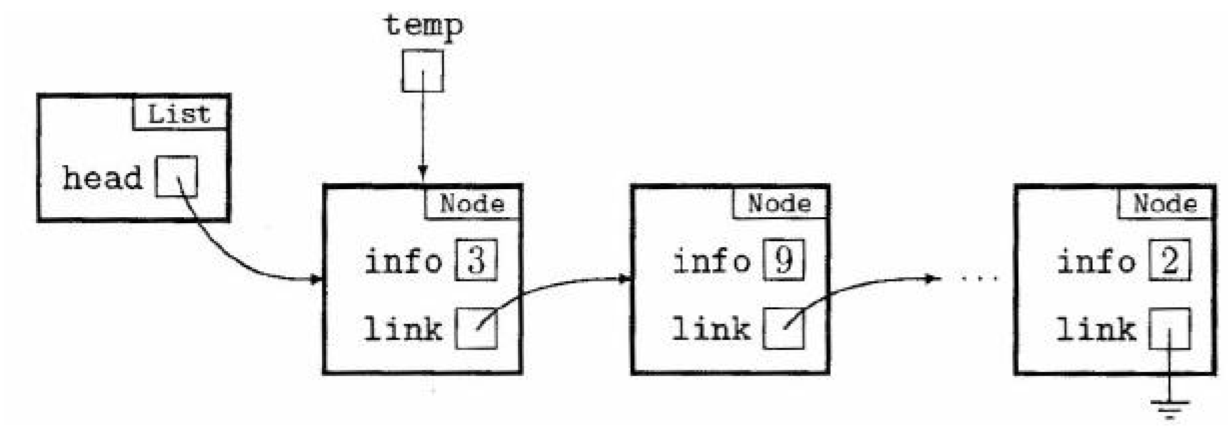
So that temp refers to the first node of the list (if there is one). To get temp to move from one node to the next, we can write:

```
temp = temp. link;
```

We show how this works in the next example.

Example 3

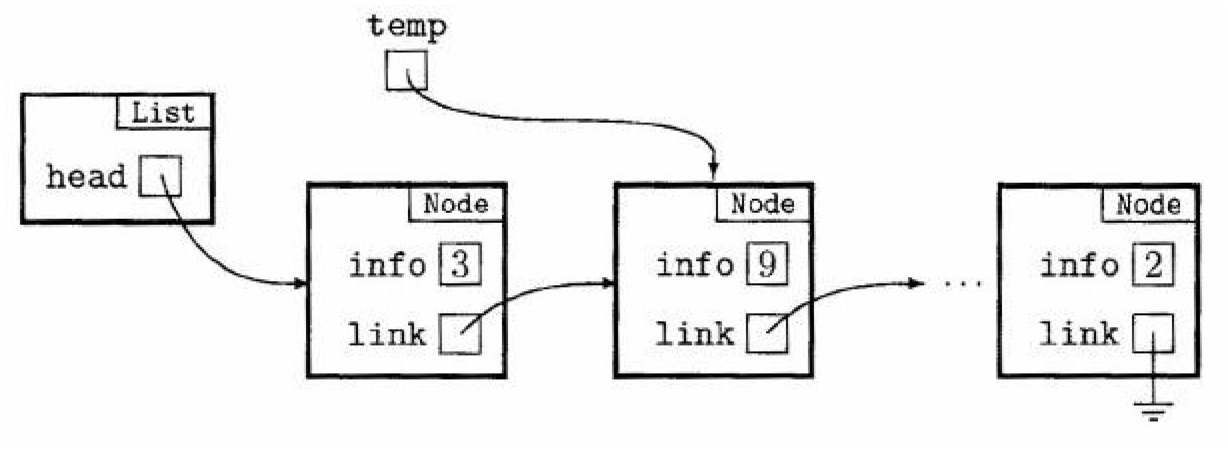
```
The method printList (in the List class) traverses a linked list of the
form shown in this section, printing the contents of the info field of each
node in the list.



public void printList ()
{
  for (Node temp = head; temp != null; temp = temp. link)
    System.out.println(temp.info);
}


Notice the way that the loop is controlled. If the list is empty, then
temp will have an initial value of null, the expression temp != null will
never be true, the loop will never be entered, and nothing will be printed.
If the list does contain any items, then the initial assignment


temp = head;


makes temp refer to the same object that head is referring to the first
node in the list.
```



After a value has been printed by the print In statement, the statement

```
temp = temp. link;
```

in the header of the loop causes the value in temp . link to be copied into temp. Since temp . link is a reference to the next node in the list, temp will now refer to that node also, as shown in the next diagram.



Each time we go through the loop, temp moves along to refer to the next node, eventually traversing the entire list. When the value of the last node's link field is assigned to temp, it will become **null** and the loop will terminate, as required.

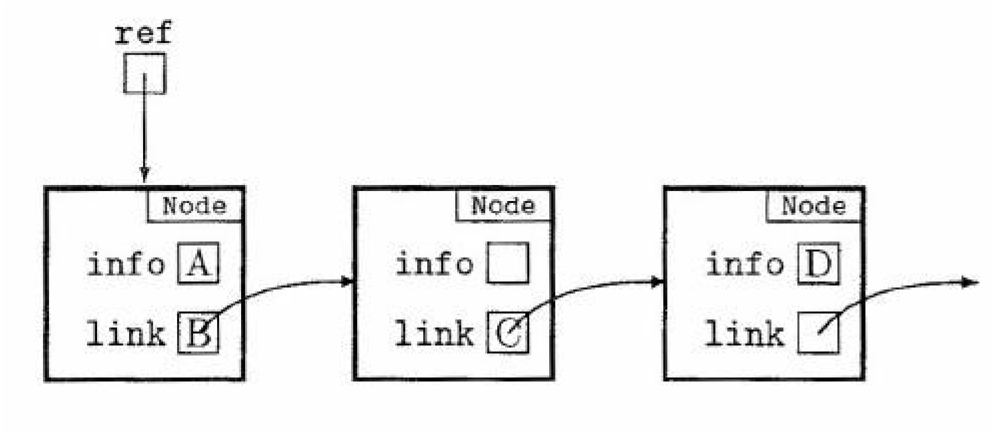
Once we have finished with a list, it is easy to get rid of the nodes. All that we have to do is write

```
head = null;
```

This will cause the reference to the first node to be lost and, since the only way that we can get to the other nodes is via the links starting from the first node, execution of the statement causes us to lose contact with the entire list. In some languages these now useless cells would clutter up memory for the rest of the life of the program unless the programmer explicitly disposed of them but, in Java, a program known as a "garbage collector" automatically frees up the space occupied by any inaccessible objects. Luckily, we do not have to worry about the details (in this course).

Exercise 12.1

1. In the diagram, ref is a reference to an object of the type Node discussed in this section. The. field marked A can be referred to as ref. info. Find similar names for the fields marked B, C, and D.



2. Rewrite the method printList shown in Example 3 so that, as well as printing the values in the list, it also prints a message if the list is empty.

3. Write an instance method sum for the List class that returns the sum of the values in the info fields of its implicit List object.

4. Write an instance method deleteFirst for the List class that deletes the first node of a linked list. If the list is empty, the method should print a warning message.

5. Write an instance method deleteLast for the List class that deletes the last node of a linked list. If the list is empty, the method

should print a warning message.

6. Write a toString method for the List class. The method should return a string that contains all the inf 0 fields of the list, separated by / /. For example, if the list contained the integers 3, 5, and 8 in its info fields, the method should return "3//5//8".
7. Write an instance method addAtRear for the List class. The method should have a single int parameter called item. The method should first locate the last node in a list and then create and attach a new node, containing the value of item, at that end of the list. In writing your method, be sure that it works correctly for an empty list.