A LINKED LIST CLASS

As we will see in the next section, the Standard Template Library has a `list<T>` class template that stores list elements in a linked list (but which have a more complex structure than we have been considering). Like most of the other STL containers, `list<T>` provides many list operations, because it is intended for use in a wide variety of list-processing problems. As we have noted before, there are times when one doesn't need or want all of the operations, and a "lean and mean" linked-list class that contains the basic list operations would be more suitable. We will now indicate how such a class might be designed and implemented.

Nodes. Before we can build a linked list class, we must first consider how to implement the nodes that make up a linked list. Since each node has two different parts—a data part and a next part—it is natural to have a `Node` class with two instance variables, `data` and `next`. The instance variable `data` will be of a type that is appropriate for storing a list element, and the `next` member will be a pointer to the node that stores the successor of this element:

```
class Node
{
 public:
   . . .
   DataType data;
   Node * next;
};
```

Note that this definition of a `Node` is a *recursive (or self-referential) definition* because it uses the name `Node` in its definition: The `next` member is defined as a pointer to a `Node`.

You might wonder why we have made the instance variables of `Node` public. This is because the declaration of class `Node` will be placed inside another class `LinkedList`. Making the instance variables of `Node` public makes them accessible to all of the methods and friend functions of `Linked List`.[9] However, the `Node` class will be inside the private section of the class `LinkedList`, so these instance variables will not be accessible outside the class:

```
#ifndef LINKEDLIST
#define LINKEDLIST

template <typename DataType>
class LinkedList
{
 private:
   /*** Node class ***/
   class Node
   {
    public:
     // Node's operations
        . . .
```

[9] Instead of a class, we could use a *struct*, whose members are public by default. However, it is common practice to use structs for C-style structs that contain no function members and classes when there are. See also Footnote 10.

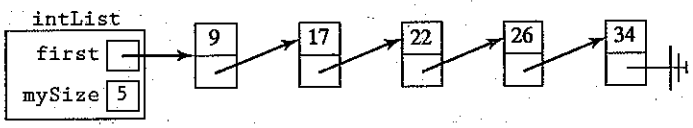
```
      // Node's instance variables
       DataType data;
       Node * next;
    };
    typedef Node* NodePointer;

  public:
   // LinkedList's methods
     . . .
  private:
   // LinkedList's instance variables
   . . .
};
. . .
#endif
```

Only one instance variable for the `LinkedList` class is needed: a pointer to the first node. However, several operations are easier to implement if we add another instance variable to keep a count of the list elements. Thus we would put the following declarations in the class:

```
// LinkedList's instance variables
  NodePointer first;     // points to first node
  int mySize;            // number of nodes
```

A typical `LinkedList<int>` object `intList` might then be pictured as follows:



As we will see in the next section where we look at STL's version of a linked-list class, there are many different operations on linked lists. Here we will describe only some of the basic ones. To save space, we will leave most of the details of implementing these with methods and functions as exercises.

Constructor. The constructor creates an empty list. It need only make `first` a null pointer and initialize `mySize` to 0:

```
//--- Constructor
template <typename DataType>
inline LinkedList::LinkedList()
{
   first = 0;
   mySize = 0;
}
```

List Traversals. Several list operations such as sorting and searching requiring **traversing** the list; that is, starting with the first node, process the data part of a node in the

required manner, and then move to the next, repeating this as long as necessary. The following statements show the basic technique for traversing the entire list:

```
ptr = first;       // ptr is of type NodePointer
while (ptr != 0)
{
   /* Appropriate statements to process
      ptr->data are inserted here */
   ptr = ptr->next;
}
```

Replacing the comment inside the loop with an output statement would produce an output operation for `LinkedList`.

Insert and Delete. The descriptions we gave earlier for the insert and delete operations lend themselves easily to algorithms and then to definitions of methods for these operations. Recall that they both require having a pointer positioned at the node that precedes the point of insertion or deletion. Positioning this pointer is done with an appropriate version of list traversal.

Assuming that `predptr` has been positioned, we can insert a node with code like the following. It assumes that the class `Node` has an explicit-value constructor that creates a node containing a specified data value and sets its `next` pointer to 0:

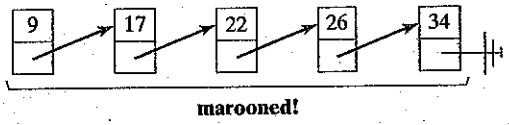
```
newptr = new Node(dataVal);    // newptr of type NodePointer
if (predptr != 0)              // Not inserting at front
{
   newptr->next = predptr->next;
   predptr->next = newptr;
{
else                           // Inserting at front
{
   newptr->next = first;
   first = newptr;             // reset first
}
```

And we can delete the specified node with code like the following:

```
if (predptr != 0)              // Not deleting first node
{
   ptr = predptr->next;
   predptr->next = ptr->next; // bypass
{
else                           // Deleting first node
{
   ptr = first;
   first = ptr->next;          // reset first
}
delete ptr;                    // return node to free store
```

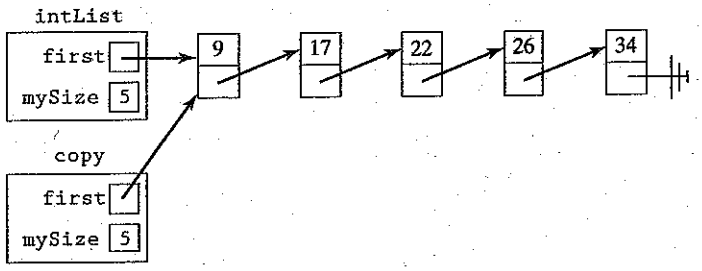
Destructor. We would need to add a destructor to `LinkedList` for the same reason as for run-time arrays. If we don't provide one, the default destructor used by the compiler

for a linked list will cause memory leaks. The compiler deallocates memory for the instance variables `first` and `mySize`, but the nodes in the linked list will be marooned:



There are several ways to implement a destructor, but perhaps the most natural one is to do a traversal, deleting each node as we go.

Copy Constructor and Assignment. A copy constructor also is needed for the same reasons as for run-time arrays. If we don't provide one, the default copy constructor (which just does byte-by-byte copying) used by the compiler for a linked list like `intList` will produce



For the same reason, an assignment operator must be provided. Both of these are list traversals where we copy each list element into a new node and link the nodes together.