Chapter 6

Mutable Lists

Chapter 5 describes immutable lists, that is, lists that cannot change. If you want a new list that is some alteration of a given list you must make a new list from the old list with the alteration incorporated in the construction process of the new list. Once that new list is constructed, it can also never change.

Immutable lists have many theoretical and practical advantages. One disadvantage, however, is low efficiency. In the above scenario, if you no longer care for the original list it must be garbage collected. A more efficient process is to allow the original list to be modified directly, which no longer requires copying or garbage collecting.

This chapter presents three mutable list implementations — the classic linked list, the Composite State list, and the Composite State Visitor list. The first data structure is the one used in practice. The second and third implementations are presented to teach two important OO design patterns — the Composite State pattern and the Composite State Visitor pattern.

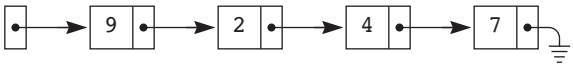
6.1 The Classic Linked Implementation

Figure 6.1 shows some common implementations of the classic linked list. Part (a) is the simplest. A node has two attributes, `_data` and `_next`.

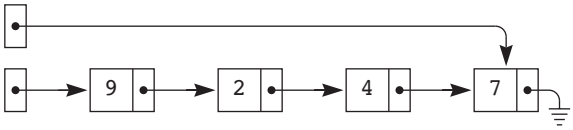
```
template<class T> class LNode {
private:
    T _data;
    shared_ptr<LNode<T>> _next;
```

Attribute `_data` has type `T`. Attribute `_next` is a pointer to a `LNode` and points to the next node in the linked list. The list itself consists of a single pointer to the first node at the head of the list.

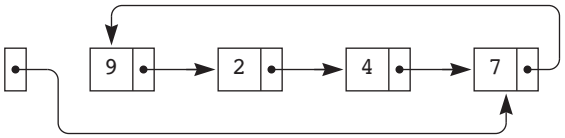
```
template<class T> class ListL :
    public enable_shared_from_this<ListL<T>> {
private:
    shared_ptr<LNode<T>> _head;
```



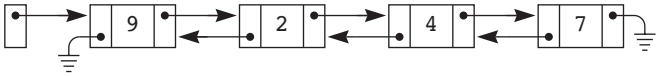
(a) A linked list.



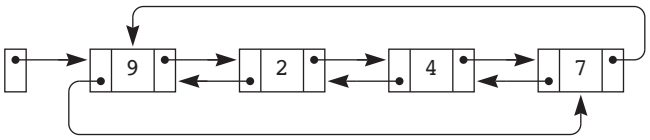
(b) A linked list with a second pointer to the last node.



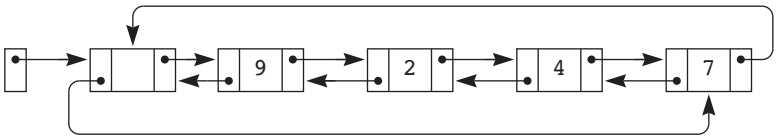
(c) A circular linked list.



(d) A doubly linked list.



(e) A doubly linked circular list.



(f) A doubly linked circular list with a head node.

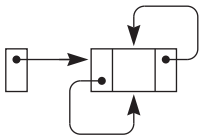
Figure 6.1 Classic linked lists. The lists in parts (b) and (c) have access to the last node in time $\Theta(1)$. The doubly linked lists allow traversal in both directions.

The last node in the list has a value of `nullptr` for `_next`, which is used to detect the end of the list. Class `ListL` inherits from the class `enable_shared_from_this` to allow the use of the iterator design pattern described in Section 6.2. Otherwise, this inheritance would not be necessary.

The advantage of storing data in a linked list as opposed to an array is that you do not need to specify the maximum number of data elements when you declare the list. You can efficiently allocate a node from the heap when you add an element and deallocate from the heap when you remove an element.

One disadvantage of the list in Figure 6.1(a) is the inefficiency in accessing its last

(a) The empty list for a list without a head node.



(b) The empty list for a doubly linked circular list with a head node.

Figure 6.2 Empty linked lists.

element, say if you want to append a value at the end of the list. To access the last element, you must traverse the list from beginning end following the links from node to node. The time to access the last element is $\Theta(n)$ where n is the number of elements in the list.

Part (b) shows an implementation that overcomes the inefficiency by providing a second pointer to the last node. The node is defined as before, but the list would now be defined as

```
template<class T> class ListL {
private:
    shared_ptr<LNode<T>> _head;
    shared_ptr<LNode<T>> _tail;
```

Because no loop is required to access the last element the time to do so is reduced from $\Theta(n)$ to $\Theta(1)$.

The list in part (b) is not an elegant solution to the problem of efficiently accessing the last node as it adds complexity to the definition of the list. A better solution is to make the list circular as in part (c). Instead of maintaining the `_next` field of the last node to be `nullptr`, maintain it to point to the first node. Instead of having two pointers in the definition of a list, have one pointer that points to the last node. No matter how many elements are in the list, the time to access the first node and the last node is $\Theta(1)$. It does not require a loop to get from the last node to the first node.

Figure 6.1(d) is another common implementation of a linked list called a doubly linked list. In such a list, each node has a pointer to the previous node as well as the next node.

```
template<class T> class LNode {
private:
    shared_ptr<LNode<T>> _prev;
    T _data;
    shared_ptr<LNode<T>> _next;
```

There are two advantages of this implementation. You can traverse the list in either direction, whereas with a singly linked list you can only traverse from beginning to end. Also, when you search a list by running a pointer down the nodes, some algorithms require access to the node just before the node to which the running pointer points. Having a pointer that points to the previous node can simplify the code in such algorithms.

The doubly linked list in part (d) has the disadvantage of requiring time $\Theta(n)$ to access the last node. That deficiency is corrected by the doubly linked circular list in

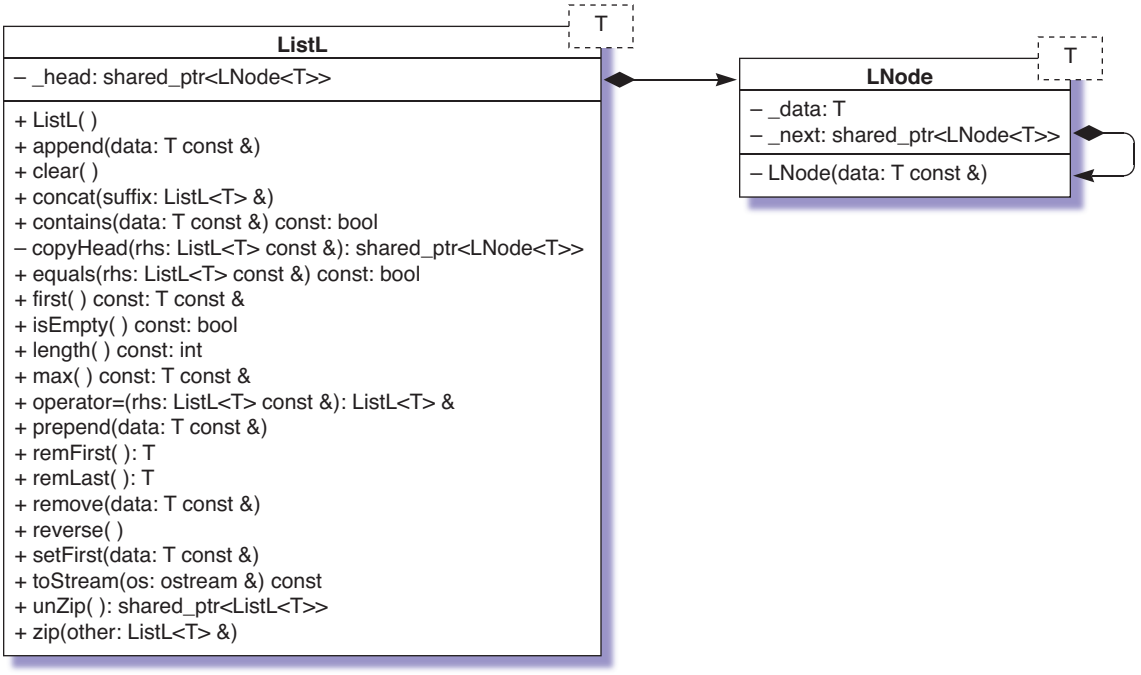


Figure 6.3 The UML class diagram for the linked list in the dp4ds distribution.

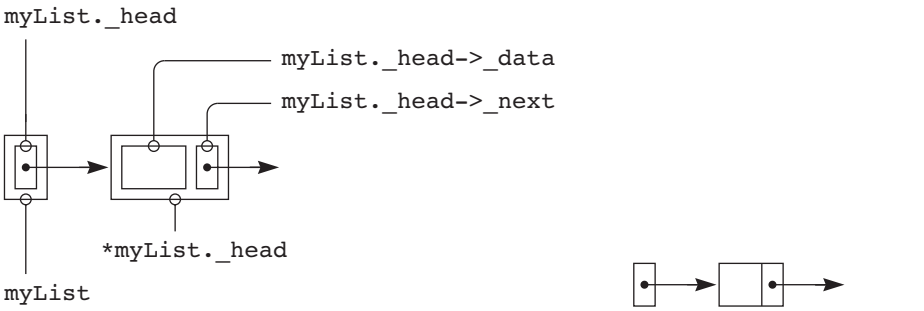
part (e). The list pointer need not point to the last node, because it can use the `_prev` pointer of the first node to access the last node in time $\Theta(1)$.

Figure 6.1(f) shows a common implementation of the classic linked list. Compared to the doubly linked circular list of part (e), it has an additional head node with no data. Figure 6.2(b) shows that even the empty list has a head node. Part (a) shows the empty list for an implementation without head nodes. The algorithms for lists without head nodes must frequently check if the list is empty and perform one kind of processing if it is empty and another kind of processing if it is not. Maintaining a head node even for empty lists simplifies the code because often no special cases are required for empty lists.

The dp4ds linked list

The dp4ds distribution software implements the classic linked list of Figure 6.1(a). As you work through the algorithms, you should keep in mind how they would be modified if the implementations were for one of the other variations in the figure.

Figure 6.3 shows the UML class diagram for the data structure. There are two classes, `ListL` for the list itself and `LNode` for the node. The symbol ◆– is the UML notation for class composition. In the code,



(a) A detailed rendering of a list and a node.

(b) An abbreviated rendering.

Figure 6.4 Rendering of the `ListL` list and the `NodeL` node.

```
template<class T> class ListL :
    public enable_shared_from_this<ListL<T>> {
private:
    shared_ptr<LNode<T>> _head;
```

an object of class `ListL` has an attribute that is a pointer to another object of class `LNode`. In other words, `ListL` is composed of `_head`. Corresponding to this composition, the arrow in the UML diagram points from the attribute `_head` in the `ListL` class to the class of `_head`, which is `LNode`. Class composition is known as a *has-a* relationship because, in this example, `ListL` *has a* pointer to an `LNode`.

Similarly, the code for an `LNode`

```
template<class T> class LNode {
private:
    T _data;
    shared_ptr<LNode<T>> _next;
```

shows that it is composed of the attribute `_next`, which is a pointer to an object of the same class. Hence, the composition arrow on the right in Figure 6.3 points from the `LNode` class to itself.

The two most important relationships in OO design are inheritance and class composition. Here is a comparison.

- Inheritance
  Symbol: △
  Relationship: *is-a*
  Example: In Figure 1.14, a `Rectangle` *is a* `AShape`.
- Class composition
  Symbol: ◆—
  Relationship: *has-a*
  Example: In Figure 6.1, a `ListL` *has a* `LNode`.

Figure 6.4 shows a rendering of a list and a node. The figure assumes `myList` has been declared a `ListL` as


```
void toStream(ostream &os) const;
// Post: A string representation of this list is returned.

bool isEmpty() const { return _head == nullptr; }
// Post: true is returned if this list is empty;
// Otherwise, false is returned.

T const &first() const;
// Pre: This list is not empty.
// Post: The first element of this list is returned.

int length() const;
// Post: The length of this list is returned.

T const &max() const;
// Pre: This list is not empty.
// Post: The maximum element of this list is returned.

bool contains(T const &data) const;
// Post: true is returned if data is contained in this list;
// Otherwise, false is returned.

bool equals(ListL<T> const &rhs) const;
// Post: true is returned if this list equals list rhs;
// Otherwise, false is returned.
// Two lists are equal if they contain the same number
// of equal elements in the same order.
```

Figure 6.5 Specifications for the output and characterization methods of the `ListL` data structure, and implementation of the `isEmoty()` method.

```
ListL myList;
```

You access each element in the figure as follows:

- `myList` is the list.
- `myList._head` is the attribute of the list, which is a pointer to a node.
- `*myList._head` is the object to which `myList._head` points, which is a node.
- `myList._head->_data` is the the data field of the node, which has type `T`.
- `myList._head->_next` is the the link field of the node, which is a pointer to a node.

Figure 6.4 (a) shows each above element in detail. Part (b) is a common abbreviation that merges some of the elements to keep the rendering uncluttered. Most figures in this section use the abbreviated rendering.

There is a double ambiguity on the interpretation of the first box in part (b). You can see in part (a) of the figure that this box could represent `myList`, which is a list,


```
// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, ListL<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void ListL<T>::toStream(ostream &os) const {
    os << "(";
    for (auto p = _head; p; p = p->_next) {
        if (p->_next) {
            os << p->_data << ", ";
        } else {
            os << p->_data;
        }
    }
    os << ")";
}
```

Figure 6.6 The output methods for the `ListL` data structure.

or it could represent `myList._head`, which is a pointer to a node. You can determine which it represents by the type of the identifier. For example, in Figure 6.13 on page 189 the box for `rhs` represents a list because `rhs` has type `ListL`. In the same figure, the box for `_result` represents a pointer to a node because `result` has type `LNode *`.

Figure 6.3 shows that `ListL` and `NodeL` have constructors, methods, and global functions. Each of these operations fall into one of the following four categories:

- Output and characterization
- Construction and insertion
- Destruction and removal
- Manipulation

The code in the dp4ds distribution specifies preconditions and postconditions for each of the methods. The following sections describe the methods, giving the code for some of them and leaving the code for others as exercises at the end of the chapter.

Output and characterization

These methods do not modify the list data structure, and so are the simplest to implement. They output a string representation of the list to the terminal, check if the list is empty, access the data value of the first node, compute the length of the list, determine the maximum value in the list, test if the list contains a specific value, and test if two lists are equal. Figure 6.5 shows the specification for this group of methods. Because the

implementation of `isEmpty()` is a single line of code, its implementation is given at the place of its specification.

Figure 6.6 shows the output methods for the `ListL` data structure. The definition of `operator<<` allows the client program to output the content of `myList` with the statement

```
cout << myList;
```

When the above statement executes, `cout` is the first actual parameter and `myList` is the second actual parameter. `cout` corresponds to the first formal parameter `os`, and `myList` corresponds to the second formal parameter `rhs`, which stands for "right hand side" because `myList` is on the right hand side of the `<<` operator in the above output statement.

The function `operator<<` is not a method of any class. It is a global function that takes a `ListL` as one of its parameters. The implementation makes a single call to the `ListL` method `toStream`, which formats the output. If `myList` has the values of Figure 6.1(a) then the output is

```
(9, 2, 4, 7)
```

Usually a `for` statement has an integer as the control variable. This method shows that a `for` statement can have a pointer `p` as the control variable. While an integer variable is usually incremented with the `++` operator each time through the loop, the control variable `p` is advanced to the next node of the list with

```
p = p->_next
```

each time through the loop.

Notice in the above output that each integer is followed by a comma except for the last integer 7. The statement

```
if (p->_next)
```

is true when `p->_next` is not `nullptr` and so detects whether the current node is the last node, in which case the comma is not printed.

The `equals()` method is complicated by the fact that either or both lists can be empty and may or may not be the same length. With all the possibilities, it is easy to write algorithms that crash with `nullptr` references. It is possible to write a correct algorithm with only one `while` loop having a conjunction of three short-circuit tests for loop termination — that the computation is not at the end of the first list, that the computation is not at the end of the second list, and that the respective `_data` values are not equal. After loop termination, you know the lists are equal if you reached the end of both lists together.

Construction and insertion

These methods modify the list data structure by constructing new lists from existing lists or by adding an element to an existing list and thereby increasing its length by one. How you write C++ code for the construction process depends crucially on the difference between a shallow copy and a deep copy. Figure 6.7 shows the difference between these two copy processes. Part (a) of the figure shows the initial values of two lists. `yourList` initially contains four elements, and and `myList` is initially empty.

Suppose you execute the assignment statement

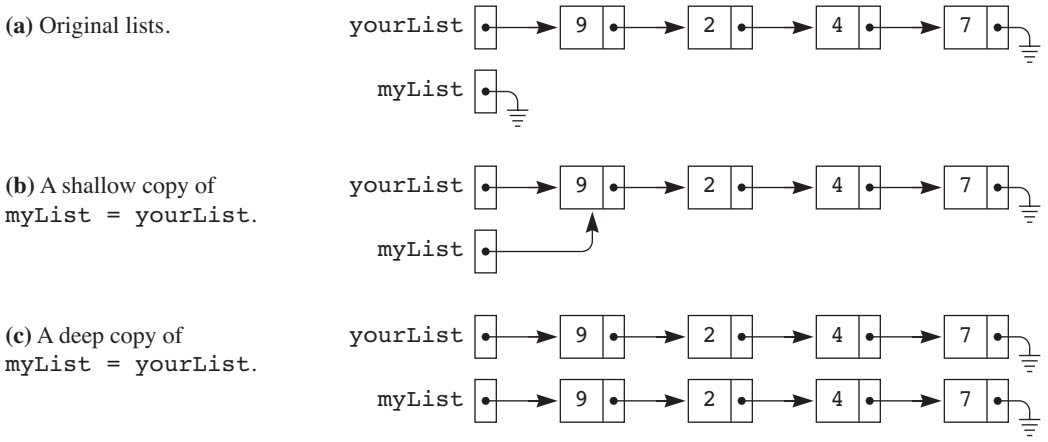


Figure 6.7 The difference between a shallow copy and a deep copy.

```
myList = yourList;
```

with the initial instantiations in part (a). The assignment statement gives a copy of `yourList` to `myList`. The preferred verbalization of the assignment statement is, "`myList` gets `yourList`." However, the result of the assignment depends on whether the copy is a shallow copy of `yourList` or a deep copy of `yourList`.

Figure 6.7(b) shows the final state if the copy is shallow. Because a `ListL` has a single attribute, which is a pointer to an `LNode`, the shallow copy makes a copy of the pointer in `yourList` and gives the pointer to `myList`. After the assignment, the pointer in `myList` points to the same node to which the pointer in `yourList` points. Both lists are sharing the same linked list of nodes.

Figure 6.7(c) shows the final state if the copy is deep. Instead of simply passing a copy of the pointer from `yourList`, the assignment statement duplicates every node in `yourList`, links them up in the same order with `nullptr` in the `_next` field of the last node, and gives a pointer to the first node of the duplicated list to `myList`. In both cases, `myList._head` gets a pointer. With a shallow copy, `myList._head` points to the first node of `yourList`. With a deep copy, `myList._head` points to the first node of a copy of all the nodes of `yourList`.

One question at this point is, What are the advantages of each copy process? The advantage of a shallow copy is its efficiency, as the assignment statement executes in time $\Theta(1)$. A deep copy requires a loop to advance through the nodes of `yourList` one node at a time to construct the duplicated nodes. With a deep copy, the assignment statement executes in time $\Theta(n)$ where n is the number of elements in `yourList`.

The advantage of a deep copy is the safety that comes with not sharing lists. Working in a system with shared lists is notoriously difficult and error prone. After the assignment, suppose you append a new node at the end of `yourList`. In effect, you are appending it to `myList` as well because `_head` of both lists point to the same first node. However, suppose you prepend a new node at the beginning of `yourList`. To do so you must allocate the new node from the heap and make `yourList._head` point to it. But then, `myList._head` would no longer point to the first node of the shared list.


```
ListL(ListL<T> const &rhs) = delete;
// Copy constructor disabled.

ListL() = default;
// Post: This list is initialized to be empty.

void prepend(T const &data);
// Post: data is prepended to this list.

void append(T const &data);
// Post: data is appended to this list.

ListL<T> &operator=(ListL<T> const &rhs);
// Post: This list is a deep copy of rhs.

shared_ptr<LNode<T>> copyHead(ListL<T> const &rhs);
// Post: A deep copy of the head of rhs is returned.
```

Figure 6.8 Specifications for the construction and insertion methods of the `ListL` data structure.

This issue is at the heart of the difference between immutable lists and mutable lists. Recall that once you instantiate an immutable list you cannot change it. Therefore, the shared list scenario described above with all its complications cannot happen. The safety that comes with always using deep copies is guaranteed with immutable lists.

Another question is, What actually happens with mutable lists when you do an assignment, a shallow copy or a deep copy? The answer is, It depends. The above discussion is about the class `ListL`, which is provided in the dp4ds distribution. The distribution is a library of classes that work together to teach data structures and OO design patterns. In this library, the assignment statement is specified to be a deep copy.

All OO development environments provide a library of data structures and those with good documentation describe in detail whether shallow or deep copy is used in the scenarios that developers encounter during the development process. Sometimes you can choose one data structure over another based on this difference. You should be aware of these issues and plan accordingly when you use any OO development environment.

Figure 6.8 shows the specification for the construction and insertion methods of the `ListL` data structure. The first constructor has signature

```
ListL(ListL<T> const &rhs) = delete;
```

and is known as a copy constructor. In C++, a copy constructor has the same name as a class and has a single formal parameter with the same class and the `&` designation. The purpose of a copy constructor is to make a copy of the object in the actual parameter list.

This functionality is so common that C++ provides a copy constructor automatically even if not declared by the programmer. The implicit automatically-supplied constructor does a shallow copy. Because the dp4ds distribution specifies a deep copy, it declares this private `ListL` copy constructor but does not implement it. That prevents the com-


```
// ========= Constructors =========
template<class T>
LNode<T>::LNode(T const &data) :
    _data(data) {
}
```

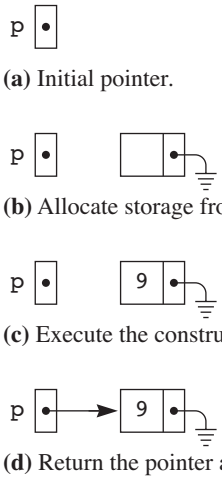


Figure 6.9 The constructors for the `ListL` data structure and the `LNode` class. The snapshots show the execution of `p = new LNode<T>(9)` for raw pointer `p`.

piler from supplying the default constructor automatically or the user of the library from writing an unsafe copy constructor for the `ListL` data structure.

The second constructor in Figure 6.8 has signature

```
ListL() = default;
```

The `ListL` class has a single attribute, `_head`, which is a shared pointer. Because shared pointers are always initialized to `nullptr` when they are allocated there is no need to write code for this constructor.

Figure 6.9 shows the constructor for the `LNode` class. The constructor use a C++ technique for initializing attributes called an *initialzer list*. The initialzer list begins with a colon `:` after the formal parameter and before the opening brace `{` for the constructor. Items in the initialzer list are separated with a comma `,` and consist of the attribute to be initialized and the value to which it is initialized in parentheses `( )`. The node constructor

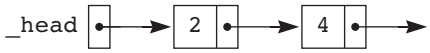
```
template<class T>
LNode<T>::LNode(T const &data) :
    _data(data) {
}
```

could alternatively be written without the initializer list as

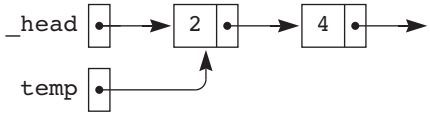
```
template<class T>
LNode<T>::LNode(T data) {
    _data = data;
}
```

where the initialization is explicit with the assignment operator =. Although the use of the initialzer list is optional in this example, some scenarios with inheritance and class

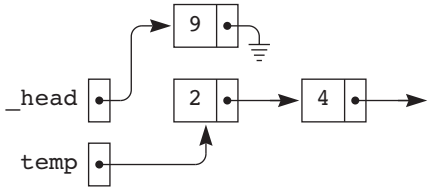

```
// ========= prepend =========
template<class T>
void ListL<T>::prepend(T const &data) {
    auto temp = _head;
    _head.reset(new LNode<T>(data));
    _head->_next = temp;
}
```



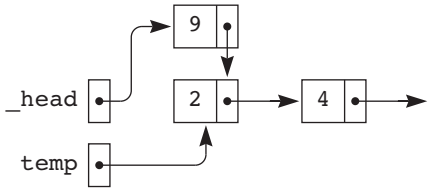
(a) Original list.



(b) `auto temp = _head;`



(c) `_head.reset(new LNode<T>(data));`



(d) `_head->_next = temp`

Figure 6.10 An insertion method for the `ListL` data structure. The snapshots show the execution of `myList.prepend(9)`.

composition require an initialzer list. It is easier to always use initialzer lists instead of trying to remember when they must be used.

The snapshots on the right show the action of the constructor when the statement

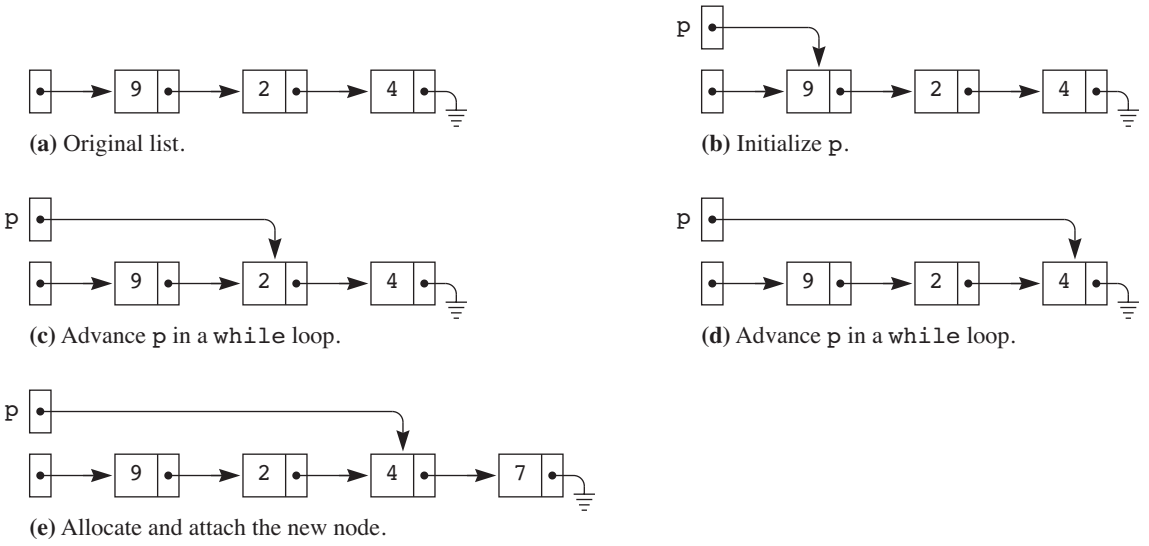
```
p = new LNode<T>(9);
```

executes for raw pointer `p`. When the `new` operator executes with a class having a constructor it does three things. It

- allocates storage from the heap for the attributes of the object,
- calls the constructor based on the number and types of parameters in the parameter list, and
- returns a pointer to the newly allocated storage.

Part (b) of the figure shows allocation from the heap. Part (c) does the implicit assignment of the initializer list. Part (d) returns the pointer, which the assignment statement gives to `p`. To make `p` a shared pointer you would execute

```
auto p = shared_ptr<LNode<T>>(new LNode<T>(9));
```

(a) Original list.

(b) Initialize `p`.

(c) Advance `p` in a `while` loop.

(d) Advance `p` in a `while` loop.

(e) Allocate and attach the new node.

Figure 6.11 The action of method `append()`. Implementation is left as an exercise. The figure shows the execution of `myList.append(7)`.

where you put the code to allocate the raw pointer in the constructor of the shared pointer.

Figure 6.10 shows the implementation of `prepend()`, one of the insertion methods of `ListL`. It only requires three C++ statements — one to set a temporary variable to the node at the head of the list, one to allocate the new node from the heap, and one to link the newly allocated node to the former first node of the list. The snapshots show the execution of

```
myList.prepend(9)
```

Because there is no loop or recursion, the time complexity of `prepend()` is $\Theta(1)$.

The other insertion method `append()` attaches the node at the end of the list. The implementation of `append()` is an exercise at the end of the chapter.

Figure 6.11 shows the action of `append()` with a list of three elements. Because `ListL` does not have direct access to the last node, you must find the last node by advancing a pointer down the list in a loop. To append a value to an empty list is a special case, which must be checked first with an `if` statement. The steps in the figure show the action of the algorithm when the list is not empty. The loop in your algorithm must stop with local variable `p` pointing to the last node. It must not be `nullptr`. Step (e) requires only one assignment statement outside the loop. Because of the loop, the time complexity of `append()` is $\Theta(n)$ where n is the number of elements in the list.

Figure 6.12 shows the two methods that provide a deep copy with the assignment statement. Method `operator=()` overloads the assignment operator so it can be used with `ListL` objects. Like method `operator<<`, which overloads the output streaming operator, the formal parameter `rhs` stands for right hand side because the corresponding actual parameter is on the right hand side of the assignment operator. In the assignment statement


```
// ========= operator= =========
template<class T>
ListL<T> &ListL<T>::operator=(ListL<T> const &rhs) {
    if (this != &rhs) { // In case someone writes myList = myList;
        _head = copyHead(rhs);
    }
    return *this;
}

// ========= copyHead =========
template<class T>
shared_ptr<LNode<T>> ListL<T>::copyHead(ListL<T> const &rhs) {
    if (rhs.isEmpty()) {
        return nullptr;
    } else {
        cerr << "ListL<T>::copyHead: Exercise for the student." << endl;
        // Three lines to set up the loop invariant,
        // followed by a single while with only three lines in the body.
        // No additional local variables.
        throw -1;
    }
}
```

Figure 6.12 Methods for providing a deep copy of a `ListL` with the assignment statement.

```
myList = yourList
```

actual parameter `yourList` corresponds to formal parameter `rhs`, `*this` corresponds to `myList`, and `_head` corresponds to `myList._head`.

The implementation in `operator=()` shows the standard C++ coding pattern for overloading the assignment operator. In general for a deep copy, the algorithm must deallocate the list on the left hand side, duplicate the list on the right hand side, set `_head` of the left hand side to point to the first node of the duplicate list, and return the object on the left hand side. With this implementation of `operator=()`, the use of shared pointers makes deallocation automatic. It duplicates the list on the right hand side by calling `copyHead()`. It points `_head` to the first node with the assignment statement. And it returns the object on the left side of the assignment statement by returning `*this`.

There is one unlikely scenario that you must guard against when you overload the assignment operator with a deep copy. Suppose the user of the `ListL` data structure writes

```
myList = myList
```

Admittedly, no user should write such an assignment statement as assigning a variable to its own value does not accomplish anything useful. However, there is nothing to prevent

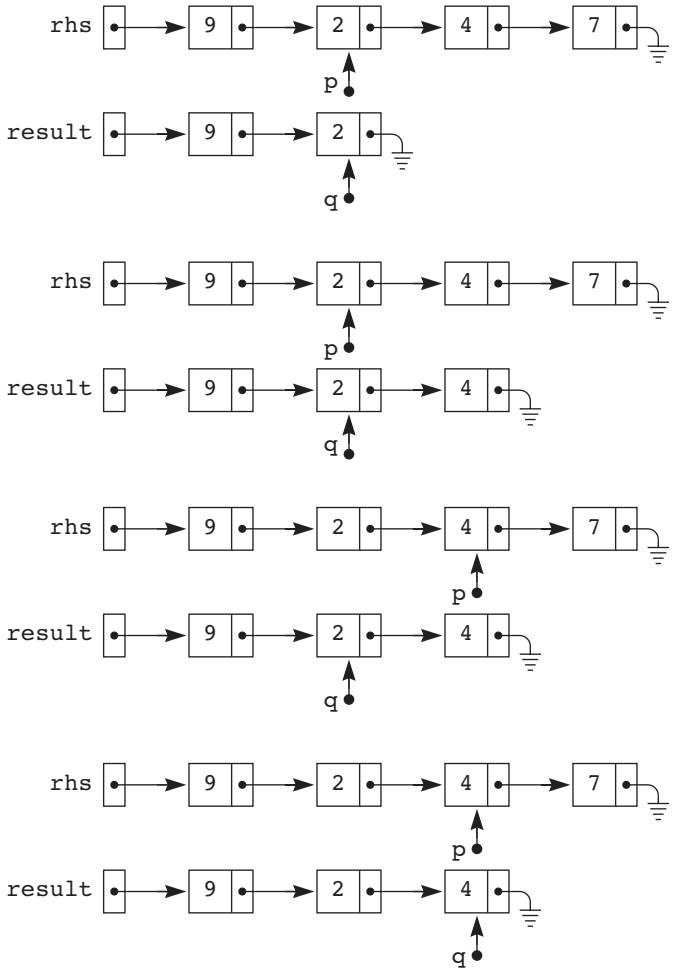


Figure 6.13 The action of method `copyHead()`. Implementation is left as an exercise.

this useless statement from being written, and if it is then the state of the computation should not change when it executes.

`operator=()` calls `copyHead()`, which does the actual duplication. Implementation of `copyHead()` is an exercise. Figure 6.13 shows the action of the algorithm, which requires a loop. You will obviously need a local pointer `p` to advance down the `rhs` list and a local variable `q` to mark the end of the list to attach the duplicate node.

Whenever you write a loop, you should always ask yourself, "What is the loop invariant?" The loop invariant for this algorithm has two parts, which describe the purposes of `p` and `q`.

- `p` points to the node in `rhs` preceding the node to be duplicated next.
- `q` points to the last node in `result`.


```
void clear();
// Post: This list is cleared to the empty list.

T remFirst();
// Pre: This list is not empty.
// Post: The first element is removed from this list and returned.

T remLast();
// Pre: This list is not empty.
// Post: The last element is removed from this list and returned.

void ListL<T>::remove(T const &data) {
// Post: If data is in this list, it is removed;
// Otherwise this list is unchanged.
```

Figure 6.14 Specifications for the destruction and removal methods of the `ListL` data structure.

To write a correct loop, you first initialize the variables to make the invariant true before executing the loop the first time. In the body of the loop, you perform the action necessary for computing one step of the algorithm. Then, you adjust the variables to re-establish the loop invariant.

Figure 6.13(a) shows the state after one of the executions of the loop body. You can see that `p` points to the node in `rhs` preceding the node to be duplicated next, and `q` points to the last node in `result`. Part (b) shows the action necessary for computing one step of the algorithm. At this point in the computation, the loop invariant is no longer true. However, after advancing `p` and `q` the invariant is re-established. That is, after step (d) it is again true that `p` points to the node in `rhs` preceding the node to be duplicated next, and `q` points to the last node in `result`. The correctness of the invariant before step (b) guarantees that the processing in step (b) is correct.

Algorithms with loops, especially those with pointers, can be difficult to implement. You should practice trying to formulate the proper loop invariant before you code the loop. A proper invariant can guide your decisions as you design your algorithm.

Destruction and removal

These methods modify the list data structure by removing one or more elements from the list. Figure 6.14 shows the specification for this group of methods. Because shared pointers provide automatic garbage collection, there is no need to have a destructor for `ListL`. There are methods to clear a list, remove the first element from a list, the last element from a list, and to search for a value in the list and remove that element if it exists.

Figure 6.15 shows the implementation of method `clear()`. The snapshots on the right side of the figure show what happens. Part (a) shows the initial list. Part (b) shows that `_head.reset()` has the effect of setting `_head` to `nullptr`. A shared pointer is an object whose attribute includes a raw pointer. It is this raw pointer attribute that is


```
// ========= clear =========
template<class T>
void ListL<T>::clear() {
    _head.reset();
}
```

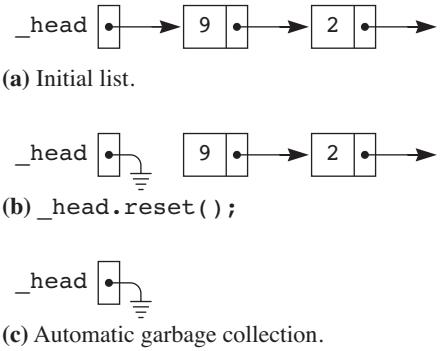


Figure 6.15 The `clear()` method for the `ListL` data structure.

set to `nullptr` by the `reset()` method.

Part (c) shows the automatic garbage collection. The reference count algorithm detects that there are no longer any pointers pointing to the first node. So, the first node is deallocated automatically. But then the reference count algorithm detects that there are no longer any pointers pointing to the second node. So, the second node is deallocated automatically, and so on down the chain. If there were no automatic garbage collection, the programmer would need to write the clear method with a loop, manually deleting each node in the list.

Implementation of method `remFirst()` is an exercise. Not only does it remove the first node in the list, it returns the value that was stored in the `_data` part of the node. To implement the method you should save the value in a temporary variable, say `result`, of type `T`. Then, after you delete the node, you can return `result`.

Implementation of method `remLast()` is also an exercise. As with `append()`, you must advance a pointer down the list before you remove the last element. However, there is an additional complication with `remLast()`. You not only need a pointer to the last node, you also need a pointer to the penultimate node because you must set that node's `_next` field to `nullptr`.

Figure 6.16 shows the action of method `remLast()`. It uses what is known as the inchworm technique. As usual, pointer `p` advances through the list one node at a time until it gets to the last node. An additional pointer `q` follows p down the list. The progress of `p` and `q` is like an inchworm, who steps with his hind feet raising his midsection and then steps with his front feet flattening his midsection. The loop invariant is

- `q` points to the node previous to the node to which `p` points, if there is such a node.

You can see that the invariant is not true at step (d) in the figure. However, it is true at step (e) and at each step just before execution of the loop body.

The last method in this group is `remove()` and is also an exercise. It is similar to `remLast()` with the inchworm technique except that you stop when you reach a node having the same `_data` field as `data` instead of stopping when you reach the end of the list. Also, instead of setting the `_next` field of the node to which `q` points to `nullptr`, you must set it to point the node to which `p->_next` points.

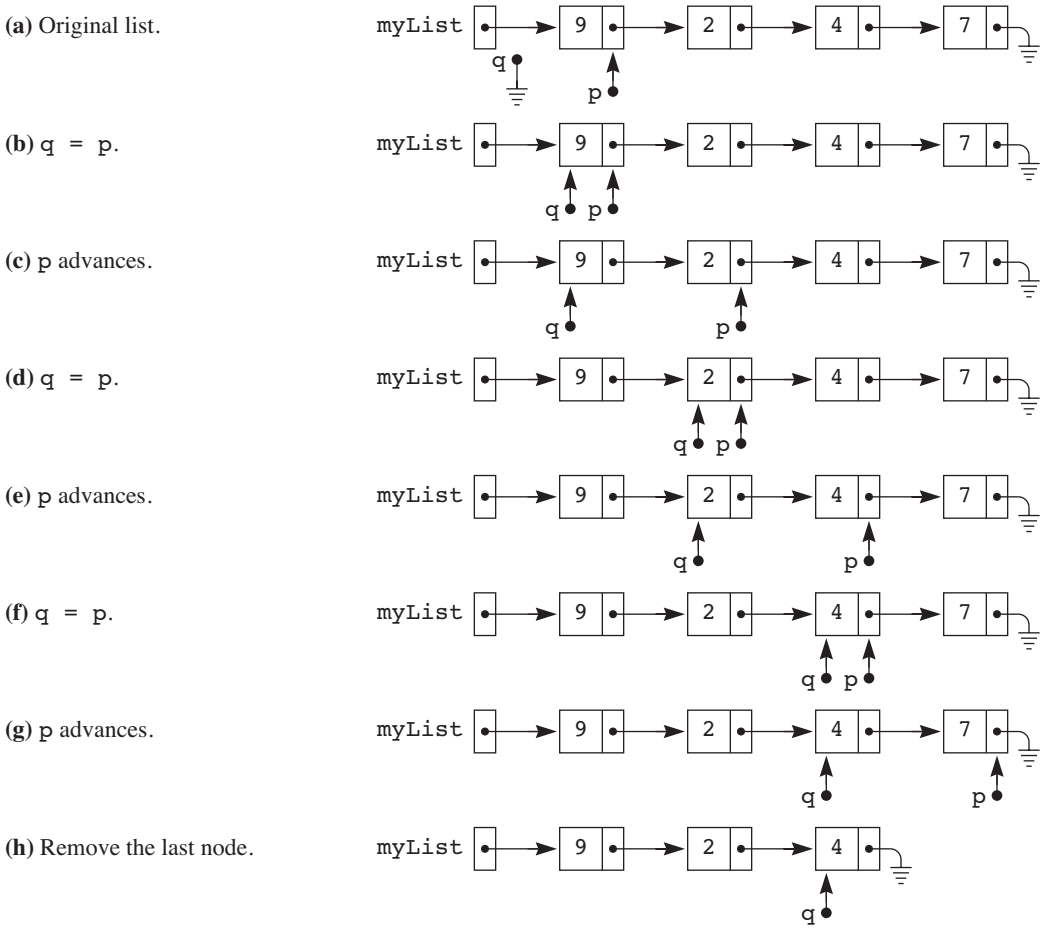


Figure 6.16 The action of method `remLast()`. It illustrates the inchworm coding technique. Implementation is left as an exercise.

Manipulation

These methods assume one or two lists have been previously instantiated and manipulate them. Figure 6.17 shows the specification for this group of methods. They set the first element to a given value, concatenate two lists, reverse the order of the elements in a list, zip two lists together, and unzip a single list producing two lists.

Figure 6.18 shows the implementation of `setFirst()`. Method `setFirst()` has a precondition that the list has at least one element and implements the precondition accordingly. An example of its use is

```
myList.setFirst(12)
```

which sets the `_data` field in the first node to 12.

Method `concat()` is an exercise. Its formal parameter is `suffix`, which is appended to `this` list. Figure 6.19 shows before and after snapshots when


```
void setFirst(T const &data);
// Pre: This list is not empty.
// Post: The first element of this list is changed to data.

void concat(ListL<T> &suffix);
// Post: suffix is appended to this list.
// suffix is empty (cut concatenate, as opposed to copy concatenate).

void reverse();
// Post: This list is reversed.

void zip(ListL<T> &other);
// Post: This list is the same perfect shuffle of this list and other,
// starting with the first element of this.
// other is the empty list (cut zip, as opposed to copy zip).

shared_ptr<ListL<T>> unZip();
// Post: This list is every other element of this list starting
// with the first.
// A pointer to a list with every other element of this list
// starting with the second is returned.
```

Figure 6.17 Specifications for the manipulation methods of the `ListL` data structure.

```
// ========= setFirst =========
template<class T>
void ListL<T>::setFirst(T const &data) {
    if (isEmpty()) {
        cerr << "setFirst precondition violated: "
             << "Cannot set first on an empty list."
             << endl;
        throw -1;
    }
    _head->_data = data;
}
```

Figure 6.18 Implementation of method `setFirst()` for the `ListL` data structure.

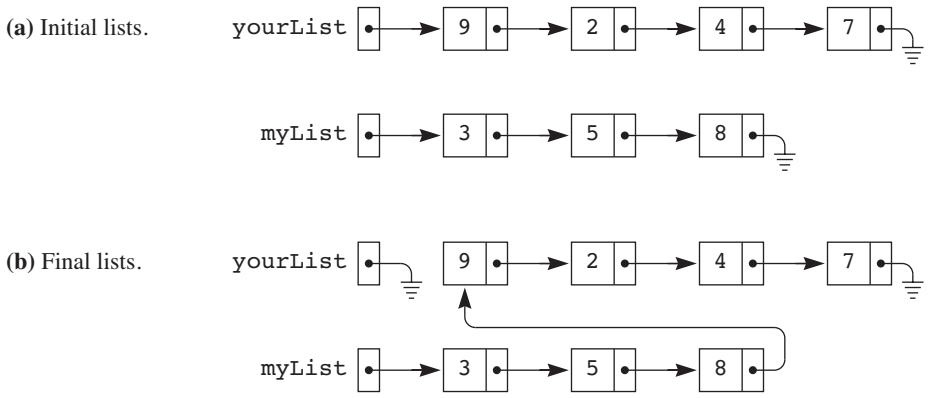


Figure 6.19 The action of method `concat()` for the `ListL` data structure. Implementation is left as an exercise.

```
myList.concat(yourList)
```

executes. Actual parameter `yourList` corresponds to formal parameter `suffix`, and `myList` corresponds to `this`.

Unlike with the assignment operator, no nodes are copied or allocated with the `new` or `make_shared` operations. Nor are any nodes deallocated with automatic garbage collection. All nodes are recycled. The algorithm simply finds the last node of `this` list, sets its `_next` field to point to the first node of `suffix`, and gives `nullptr` to the head of `suffix`. You can see that the execution time is $\Theta(m)$ where m is the number of elements in `this` list.

Method `reverse()` reverses the values in a list. If `myList` has values (9, 2, 4, 7, 3) and you execute

```
myList.reverse()
```

then `myList` will have values (3, 7, 4, 2, 9). One simple but inefficient way to reverse the list is to execute `remFirst()` in a loop removing all the values from the original list. In the body of the loop, prepend the value removed to a temporary list, which, at loop terminations contain the original values of the list but in reverse order.

The time to execute the above algorithm is $\Theta(n)$ where n is the number of elements in the list because the loop executes n times. It is obvious that no algorithm can do better than $\Theta(n)$ because every element in the list must be processed at least one time to reverse them all.

The reason for the inefficiency is that `remFirst()` deallocates a node from the heap and `prepend()` allocates a node from the heap with the same value. You can avoid the deallocation from the heap by unlinking the first node from the original list and avoid the allocation from the heap by linking the same node to the temporary list. The resulting algorithm is still $\Theta(n)$, but the constant coefficient of n is smaller.

Figure 6.20 shows the action of `reverse` using this technique. Part (a) shows the original list. Part (b) shows the state of the computation after two executions of the loop. Local variable `pReverse` is a pointer to the first node of the temporary list. The descriptions of the two temporary pointers in the figure are the two parts of the loop invariant for a list of n elements:

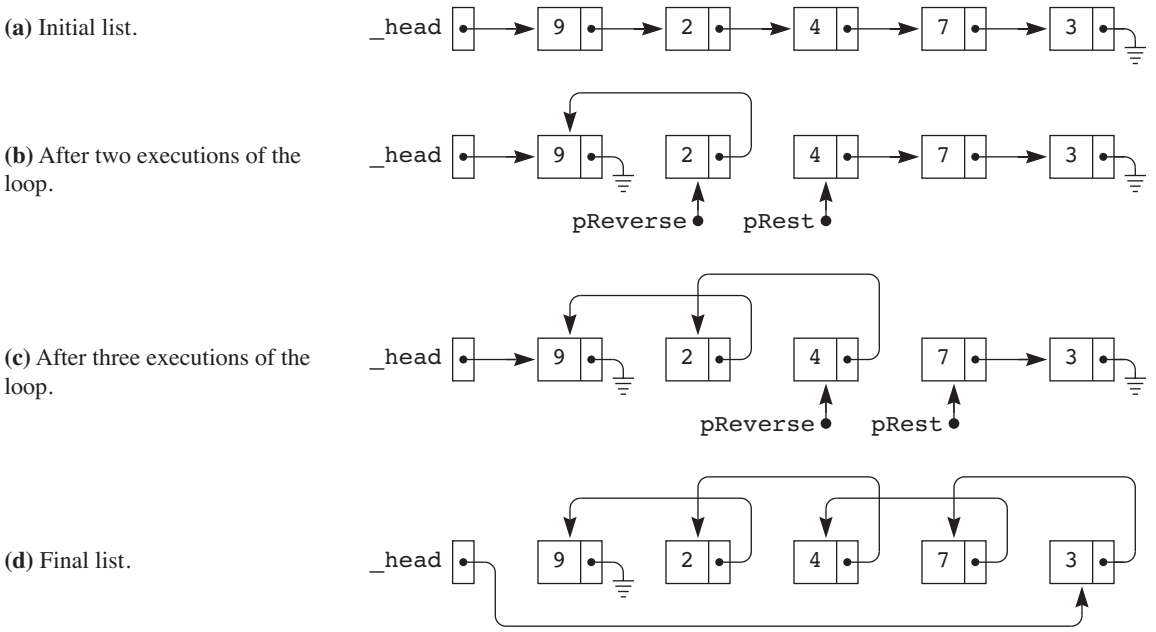


Figure 6.20 The action of method `reverse()` for the `ListL` data structure. Implementation is left as an exercise.

- `pRest` points to the first node in a list of the last m values of the original list.
- `pReverse` points to the first node of a list of the first $n - m$ values of the original list in reverse order.

For example, part (a) of the figure shows an original list of $(9, 2, 4, 7, 3)$. In part (b) of the figure,

- `pRest` points to $(4, 7, 3)$, which is a list of the last three values of the original list.
- `pReverse` points to $(2, 9)$, which is a list of the first two values of the original list in reverse order.

Part (c) shows that one more execution of the loop maintains the invariant, with `pRest` pointing to $(7, 3)$ and `pReverse` pointing to $(4, 2, 9)$.

Implementation of `reverse()` is an exercise. The initialization code should establish the loop invariant before the loop executes the first time. The code in the body of the loop should unlink the first node from `pRest`, prepend it to `pReverse`, and then re-establish the loop invariant.

To zip two lists together is to perform a perfect shuffle as if each value in the two lists are contained in two stacks of cards. To shuffle two stacks of cards you combine them by taking the first card from the first stack, the first card from the second stack,the second card from the first stack, the second card from the second stack, and so on. Figure 6.21 shows the action of

```
myList.zip(yourList)
```


(a) Initial lists.

(b) After one execution of the loop.

(c) After two executions of the loop.

(d) Final lists.

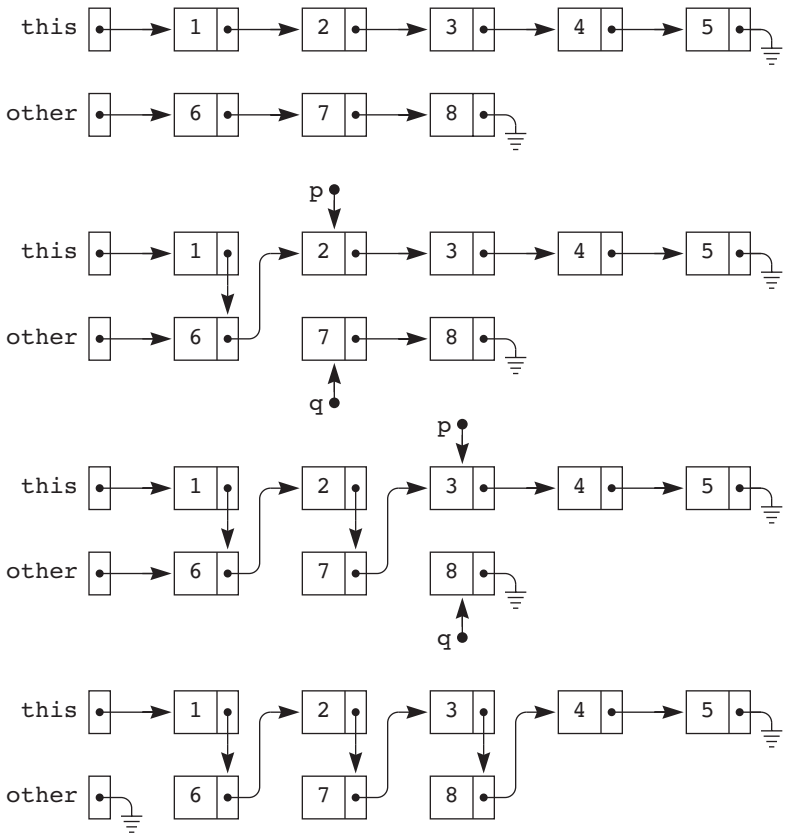


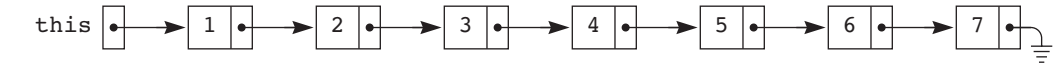
Figure 6.21 The action of method `zip()` for the `ListL` data structure. Implementation is left as an exercise.

when `myList` has values (1, 2, 3, 4, 5) and `yourList` has values (6, 7, 8). After the zip `myList` has values (1, 6, 2, 7, 3, 8, 4, 5) and `yourList` is empty.

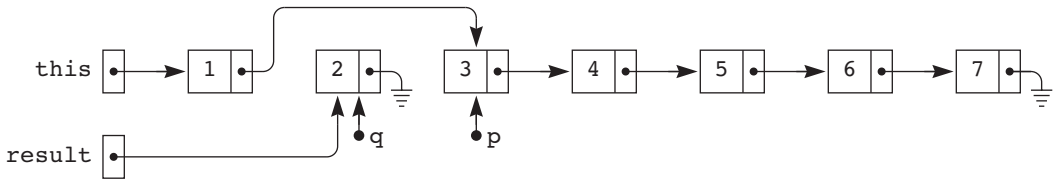
In the above example, `myList` has two more values than `yourList`. The algorithm makes no assumption about the lengths of the two lists. When processing gets to the end of one list the remaining members of the other list are the last members of the zipped list. For example, if `myList` has values (1, 2) and `yourList` has values (3, 4, 5, 6, 7, 8, 9) the zipped list should be (1, 3, 2, 4, 5, 6, 7, 8, 9).

Figure 6.21(b) shows the state of the computation after one execution of the loop. Suppose `this` list has n elements. Local variables `p` and `q` maintain the loop invariant, which has three parts.

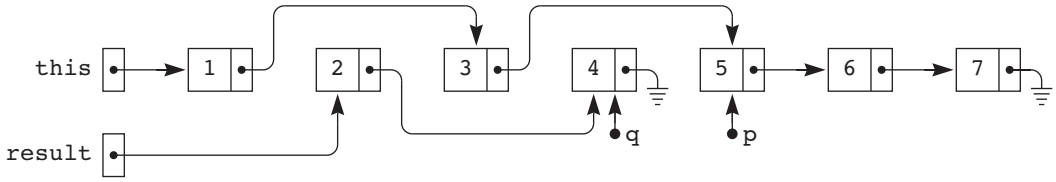
- `p` points to the first node in a list of the last m values of this list.
- The first $2 \cdot (n-m)$ items of `this` list are the first $(n-m)$ items of `this` and `other` zipped.
- `q` points to the $(n-m+1)$th value of the original other list.



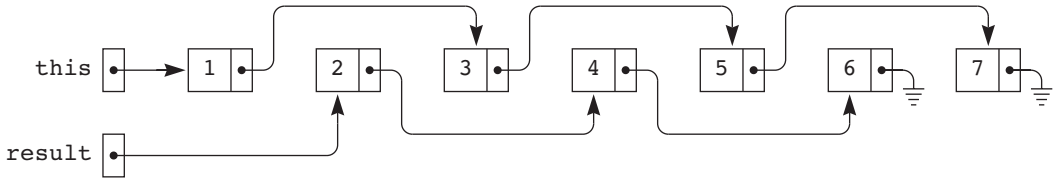
(a) Initial list.



(b) Before first execution of the loop.



(c) After one execution of the loop.



(d) Final list.

Figure 6.22 The action of method `unZip()` for the `ListL` data structure. Implementation is left as an exercise.

Implementation of `zip()` is an exercise. The initialization code should establish the loop invariant before the loop executes the first time. The code in the body of the loop should link the node to which `p` points to the node to which `q` points, link the node to which `q` points to the node to which `p` was pointing, and then re-establish the loop invariant.

Because you do not know whether `this` or `other` is longer, the test for the `while` loop to execute must guarantee that there are unzipped elements at both `p` and `q`. After the loop terminates, the algorithm must determine which list was longer and compute accordingly.

Method `unzip()` does the opposite of `zip()`. It takes a single list and creates two lists from it using every other value. Figure 6.22 shows the action of

```
yourList = myList.unZip()
```

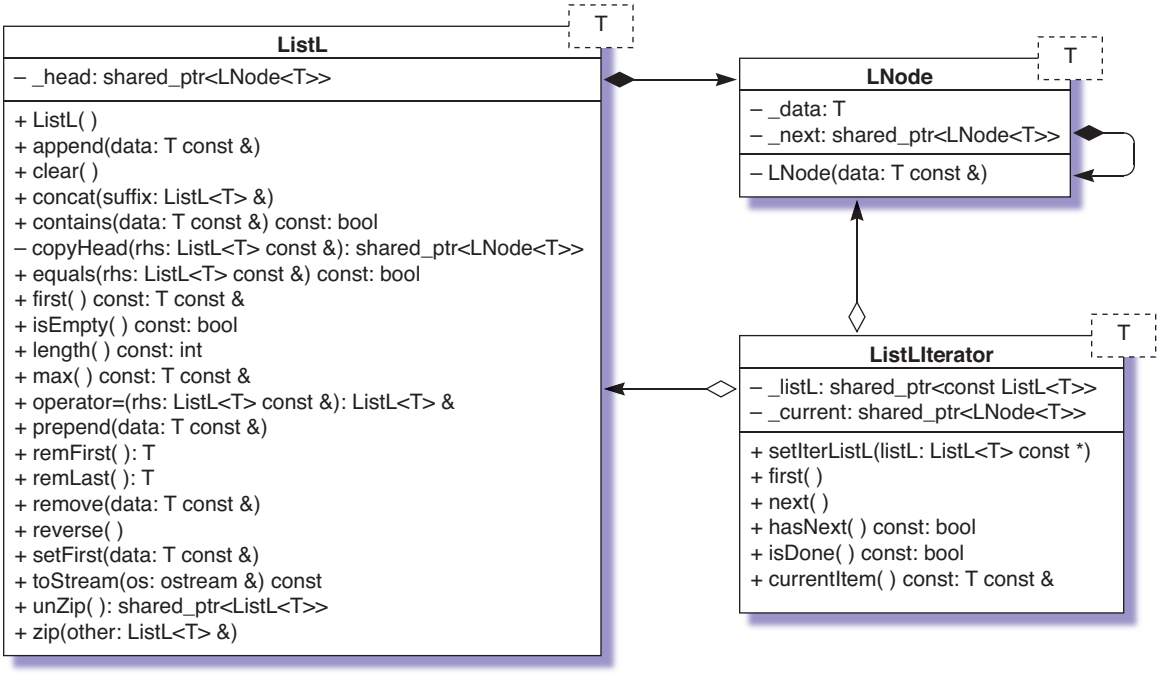



Figure 6.23 The UML class diagram for the iterator for the `ListL` data structure.

when `myList` has initial values (1, 2, 3, 4, 5, 6, 7). After execution, `myList` has values (1, 3, 5, 7) and `yourList` has values (2, 4, 6). In the figure, `result` is a local variable that is a pointer to a list. The box in the figure labeled `result` is `result->_head`.

A special case is `myList` initially empty or having only one value. In that case, `myList` is unchanged, and `yourList` is empty. Otherwise, you must set up the invariant before the loop executes, and maintain the invariant each time the loop executes. Implementation of `unZip()` is an exercise.

6.2 The Iterator Pattern

Abstraction is the hiding of detail. The `ListL` data structure allows the user to store and retrieve data without using the pointer details of the implementation, which are hidden. For example, to append the value 5 to a list the user need only write `myList.append(5)` with no pointer manipulations visible in the method call.

It would be impossible for the designer of a data structures library to anticipate all the processing requirements of the users. For example, suppose a user wanted some statistics from numeric values in the list such as the sum or the standard deviation. You could include a sum and standard deviation method in the library but whatever specialized methods you put in it, your user community would inevitably want others that you did not include.

The iterator design pattern solves the problem of hiding the pointer details of the


```
template<class T>
class ListLIterator {
private:
    shared_ptr<const ListL<T>> _listL;
    shared_ptr<LNode<T>> _current;
public:
    void setIterListL(shared_ptr<const ListL<T>> listL) {
        _listL = listL;
    }

    // Post: Positions the iterator to the first element.
    void first() { _current = _listL->_head; }

    // Post: Advances the current element.
    void next() { _current = _current->_next; }

    // Post: Checks whether there is a next element.
    bool hasNext() const { return bool(_current->_next); }

    // Post: Checks whether at end of the list.
    bool isDone() const { return !_current; }

    // Pre: The current element exists.
    // Post: The current element of this list is returned.
    T const &currentItem() const {
        if (!_current) {
            cerr << "currentItem precondition violated: "
                 << "Current element does not exist." << endl;
            throw -1;
        }
        return _current->_data;
    }
};
```

Figure 6.24 Specifications and implementation of the `ListL` iterator.

implementation while giving the user access to all the values in the list for custom processing. Figure 6.23 is the UML class diagram for the `ListLIterator`. It has two private attributes, `_listL`, which is a pointer to a `ListL`, and `_current`, which is a pointer to an `LNode`.

Inspection of the attributes of `ListLIterator` seems to indicate class composition. That is, it seems that the iterator is composed of a list and a node. The symbol ◇– with the hollow diamond indicates *aggregation* as opposed to composition. Aggregation is similar to composition as they both arise from one class having another class as its attribute. With composition, the class owns its attribute. With aggregation, the class has a link to an object that is owned by another entity. Composition is the *has-a*


```
// ========= toStream4 =========
template<class T>
void ListL<T>::toStream4(ostream &os) const {
    ListLIterator<T> iter;
    iter.setIterListL(this->shared_from_this());
    os << "(";
    for (iter.first(); !iter.isDone(); iter.next()) {
        if (iter.hasNext()) {
            os << iter.currentItem() << ", ";
        } else {
            os << iter.currentItem();
        }
    }
    os << ")";
}
```

Figure 6.25 The iterator version of `toStream()` in Figure 6.6.

relationship. Aggregation is the *has-a-link-to-but-is-not-the-owner-of* relationship.

To own an attribute is to be responsible for its allocation and deallocation. For example, the reason a `ListL` owns an `LNode` is because the list is responsible for allocating and deallocating the node. Figure 6.10 shows the code for a list allocating a node with `prepend()`, and Figure 6.15 shows the code for a list deallocating a node with `clear()`.

On the other hand, an iterator does not own a list or a node. You use an iterator with a pre-existing list. An iterator never allocates a node or a list. Figure 6.24 shows the specification and implementation of the iterator. The first method, `setIterListL`, does not allocate a new list. It assumes pre-existing list `listL` and sets its attribute to refer to that list.

The iterator maintains a current position in a list, somewhat like a cursor. Method `first()` sets the cursor to the first element, `next()` advances the cursor, `hasNext()` checks whether there is a next element in the list, `isDone()` checks if the cursor has reached the end of the list, and `currentItem()` returns the data value at the current position.

Figure 6.25 shows how to use the iterator to implement `toStream4()`, an alternate implementation of `toStream()`. The logic is identical to that in Figure 6.6 on page 181 but with no pointers visible. Iterator versions of `lenth()` and `max()` are exercises.

6.3 The Composite State Pattern

The `ListCS` data structure described in this section is one of five list implementations in the dp4ds distribution. The `ListL` data structure, or one of its variants in Figure 6.1, is the implementation used in practice. The other list implementations teach three important OO design patterns — the Composite pattern, the State pattern, and the Visitor pattern. Each list implementation other than `ListL` is a combination of one or more of

these three patterns. Here are the five list implementations of the dp4ds distribution.

- `ListL` Classic linked implementation, mutable
- `ListC` Composite pattern, immutable
- `ListCS` Composite and State patterns, mutable
- `ListCV` Composite and Visitor patterns, immutable
- `ListCSV` Composite, State, and Visitor patterns, mutable

The Composite pattern for a list is based on the following formal definition of a list.

- An empty list is a list.
- A nonempty list has two parts:
  - a data value of some type T, and
  - the rest of the list, which is a list.

This definition is recursive. That is, one of the parts of a nonempty list is itself a list. Even though `ListL` or one of its variants is common in practice, it does *not* match this formal definition of a list.

The reason `ListL` does not match the formal definition of a list is its definition of a node.

```
template<class T> class LNode {
private:
    T _data;
    shared_ptr<LNode<T>> _next;
```

In this definition, `_next` is not a list. Rather, it is a pointer to a node. In the Composite OO design pattern, a nonempty node in the `ListCS` data structure is declared as

```
template<class T> class NEcsNode : public AcsNode<T> {
private:
    T _data;
    shared_ptr<ListCS<T>> _rest;
```

A nonempty node inherits from `AcsNode<T>`, which is an abstract node. In place of `_next`, which is a pointer to a node, there is `_rest`, which is a pointer to a list. Because `_rest` is a list instead of a node, this implementation matches the formal definition of a list. All implementations of lists in the dp4ds distribution except for `ListL` incorporate the Composite OO design pattern.

The State OO design pattern gives a data structure state, which can be modified, which, in turn, makes the data structure mutable, that is, changeable. Note that dp4ds data structures `ListC` and `ListCV` do not incorporate the State design pattern and are immutable. Data structures `ListCS` and `ListCSV` do incorporate the State design pattern and are mutable.

Mutable lists incorporate the concept of a node, which can be modified, while immutable lists do not. For example, here is the definition of a nonempty list in the `ListC` data structure.

```
template<class T> class NEListC : public AListC<T> {
private:
    T const _first;
    shared_ptr<const AListC<T>> _rest;
```


A nonempty list inherits from `AListC<T>`, which is an abstract list. Data structure `ListCS` has a node and is mutable, but data structure `ListC` does not have a node and is, therefore, immutable.

This section describes the Composite State design pattern. The last section in this chapter describes the Visitor pattern, which is the basis of software plugin architectures.

The dp4ds Composite State list

Figure 6.26 is the UML class diagram for the `ListCS` data structure. As with `ListL`, `ListCS` has a `_head`, which in both data structures is a pointer to a node.

```
template<class T> class ListCS {
private:
    shared_ptr<AcsNode<T>> _head;
```

Unlike `ListL`, however, `_head` in the `ListCS` data structure is a pointer to abstract node `AcsNode` defined as follows.

```
template<class T> class AcsNode {
```

The abstract node has no attributes and is the superclass for two subclasses. The empty node `MTcsNode` inherits from `AcsNode` as follows

```
template<class T> class MTcsNode : public AcsNode<T> {
```

and has no attributes. The nonempty node `NEcsNode` inherits from `AcsNode` as follows

```
template<class T> class NEcsNode : public AcsNode<T> {
private:
    T _data;
    shared_ptr<ListCS<T>> _rest;
```

with two attributes `_data` and `_rest`. The UML diagram shows class composition where a `ListCS` *has a* `AcsNode`, inheritance where a `MTcsNode` *is a* `AcsNode`, inheritance where a `NEcsNode` *is a* `AcsNode`, and class composition where a `NEcsNode` *has a* `ListCS`.

The methods for the `ListCS` class are mostly identical to the methods for `ListL` except for three differences. `ListCS` has an additional constructor, method `setFirst()` is replaced by overloading `first()` in `ListCS`, and there are additional helper methods for `equals()`, `max()`, `remLast()`, `reverse()`, and `toStream()`. Specifications for the `ListCS` methods are identical to the corresponding ones for `ListL` and are not repeated here.

Parallel to the set of methods for list `ListCS` is a corresponding set of methods for abstract node `AcsNode`. The node methods are all abstract. Some node methods have the same signatures as their corresponding list methods. For example, the signature for the `equals()` method for both the list and the node is

```
bool equals(ListCS<T> const &rhs) const;
```

Other methods have different signatures. For example, the signature for the `remove()` method for the list is

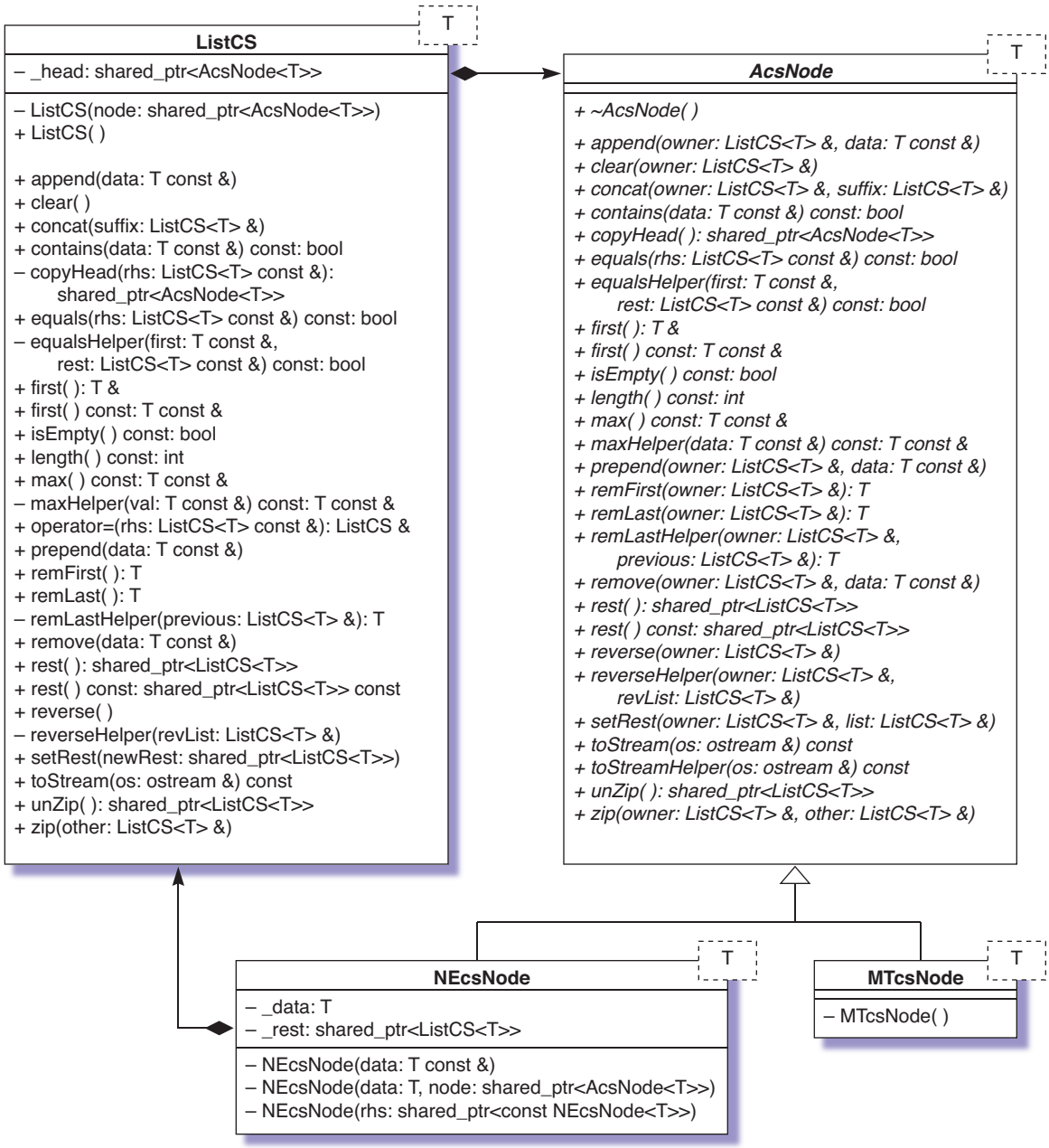
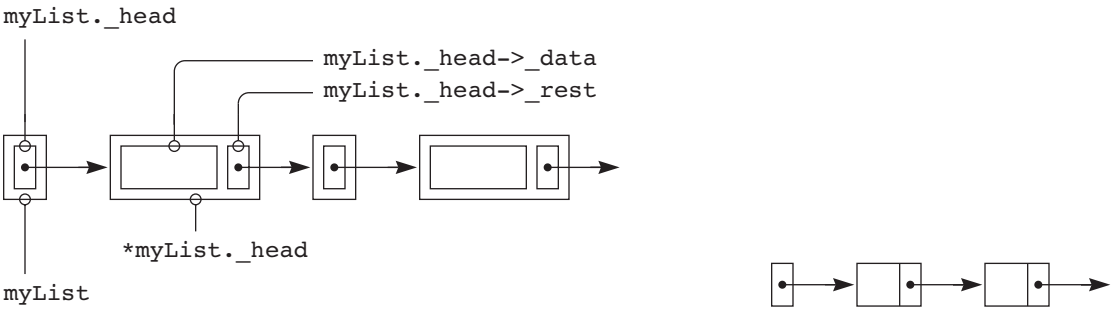


Figure 6.26 The UML class diagram for the `ListCS` data structure. It combines the Composite and the State OO design patterns.



(a) A detailed rendering of a list and a node.

(b) An abbreviated rendering.

Figure 6.27 Rendering of the `ListCS` list and the `NEcsNode` node.

```
void remove(T const &data);
```

while the signature for the `remove()` method for the node is

```
void remove(ListCS<T> &owner, T const &data);
```

Node methods that differ from their corresponding list methods do so by including an additional formal parameter named `owner` of type `ListCS<T>`, which is called by reference. Methods that do not change a list, like `equals()`, have the same signatures, while methods that do change a list, like `remove()`, have the additional `owner` parameter.

Figure 6.27(a) shows a detailed rendering of a `ListCS` and a `NEcsNode`. You can see that `myList._head->_rest` is not a pointer to a node, but rather a pointer to a list. The list to which it points, in turn, has its own `_head`. This is a graphic depiction of the Composite pattern because the `_rest` part of a nonempty node is a pointer to a list instead of a pointer to a node. One consequence of the Composite pattern is that there are two double ambiguities in the abbreviated rendering in part (b). The first box on the left could represent a list or a pointer to an abstract node. Also, the arrow from the small box in the nonempty node could represent the `_rest` pointer to the list or the `_head` pointer from that list to the node.

Output and characterization

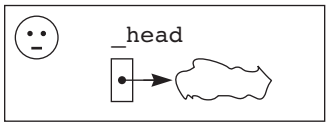
Figure 6.28 is implementation of the characterization method `isEmpty()`. If `myList` has values (4, 1, 7) then

```
myList.isEmpty()
```

returns false because the list is not empty.

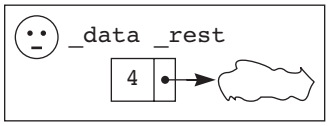
An object-oriented system is a collection of objects, each with its own local environment, that cooperate to perform the computation. The environment of an object consists of its attributes and parameters in the method it is executing. Figure 6.28 shows the standard coding pattern for a `ListCS` data structure. There are three cooperating objects — a list, an empty node, and a nonempty node. Parts (a), (b), and (c) show the local environments for these objects.


```
// ========= isEmpty =========
template<class T>
bool ListCS<T>::isEmpty() const {
    return _head->isEmpty();
}

template<class T>
bool MTcsNode<T>::isEmpty() const {
    return true;
}

template<class T>
bool NEcsNode<T>::isEmpty() const {
    return false;
}
```



(a) The environment of a list.

(b) The environment of an empty node.



(c) The environment of a nonempty node.

Figure 6.28 Implementation of method `isEmpty()` for the `ListCS` data structure.

Suppose the main program calls the method with list (4, 1, 7). Part (a) of the figure shows the environment for the list method `isEmpty()`. It is useful to think of an object as a person, represented by the face, who knows only what is in its environment. In the case of a list, it only has access to its attribute `_head`. Because `_head` is a pointer to an abstract node, the list does not know whether that node is empty or nonempty. That is, the list object does not even know whether it is an empty list.

Because it does not know whether it is empty it asks its `_head` node by executing

```
return _head->isEmpty();
```

Polymorphic dispatch determines which node method executes. In this scenario, `_head` is a pointer to the first node in the (4, 1, 7) list, so the nonempty method executes. Part (c) shows the environment of the nonempty node, which has 4 for the value of its `_data` because this object is the first node in the list. The nonempty node returns false to the list method, which passes it back to the main program.

There is no `if` statement to test whether a node is empty. Polymorphic dispatch eliminates the `if` statement by automatically executing the proper `isEmpty()` method based on the dynamic type of the node.

Figure 6.29 is the implementation of `rest()`. It returns a constant reference to the rest of a list. If `myList` has the value (4, 1, 2) then execution of

```
cout << myList.rest()
```

outputs the string `(1, 2)`.

A precondition for `rest()` is that the list is not empty. Again, the list object does not know whether it is empty, nor does it have direct access to its structure. So, it asks its abstract node with


```
// ========= rest const =========
template<class T>
shared_ptr<ListCS<T>> const ListCS<T>::rest() const {
    return _head->rest();
}

template<class T>
shared_ptr<ListCS<T>> const MTcsNode<T>::rest() const {
    cerr << "rest precondition violated: An empty list has no rest."
         << endl;
    throw -1;
}
template<class T>
shared_ptr<ListCS<T>> const NEcsNode<T>::rest() const {
    return _rest;
}
```

Figure 6.29 Implementation of constant method `rest()` for the `ListCS` data structure.

```
return _head->rest();
```

In this scenario, the nonempty node version of `rest()` executes. Figure 6.28(c) shows that `_rest` is in the environment of the first node. The nonempty node simply returns its `_rest`, which the list version passes back to the statement that called it.

Figure 6.29 also shows how to implement a precondition with a `ListCS` method. If the list is empty, the empty node version of `rest()` executes via polymorphic dispatch and terminates the program with an appropriate error message.

Figure 6.30 shows the implementation of characterization method `max()`. Its precondition is that it is not empty, because an empty list has no maximum value. If `myList` has values (2, 7, 1, 9, 3) then

```
myList.max()
```

returns 9.

The implementation shows how a helper function can assist in the computation. Here are the specifications of `max()` and `maxHelper()`.

```
T const &max() const;
// Pre: This list is not empty.
// Post: The maximum element of this list is returned.

T const &maxHelper(T const &val) const;
// Post: The maximum element of this list and val is returned.
```

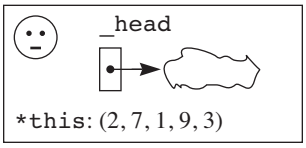
The difference is that `max()` has a precondition and analyzes only a list, while `max Helper()` has no precondition and has a parameter whose value is compared with the maximum value of a list.


```
// ========= max =========
template<class T>
T const &ListCS<T>::max() const {
    return _head->max();
}

template<class T>
T const &MTcsNode<T>::max() const {
    cerr << "max precondition violated: An empty list has no maximum."
         << endl;
    throw -1;
}

template<class T>
T const &NEcsNode<T>::max() const {
    return _rest->maxHelper(_data);
}

// --------- maxHelper ---------
template<class T>
T const &ListCS<T>::maxHelper(T const &val) const {
    return _head->maxHelper(val);
}

template<class T>
T const &MTcsNode<T>::maxHelper(T const &val) const {
    return val;
}

template<class T>
T const &NEcsNode<T>::maxHelper(T const &val) const {
    return _data > val ? _rest->maxHelper(_data) : _rest->maxHelper(val);
}
```

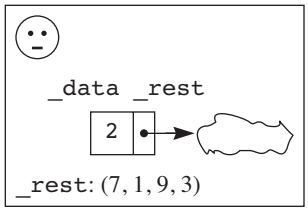
Figure 6.30 Implementation of method `max()` for the `ListCS` data structure.

Figure 6.31 is an execution trace of `max()`. Each part in the figure shows the environment of the object as the result of a method call from another object. Part (a) shows the `ListCS` object environment after the first call. It only has access to its pointer to an abstract node. Also shown in the environment is the value of `*this`, which is the original list of five elements.

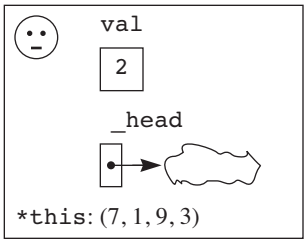
Part (b) shows the polymorphic dispatch call to the nonempty node. An object's environment consists of its attributes and parameters. Because there are no parameters with this call, this nonempty node has access to attributes `_data` and `_rest`. The value of `_data` is 2, as that is the first value in the original list. The environment also shows the value of `_rest` in an abbreviated form as (7, 1, 9, 3).



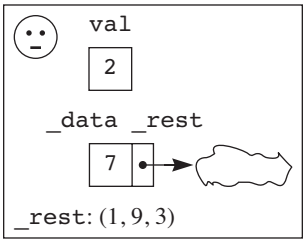
(a) `myList.max()`.



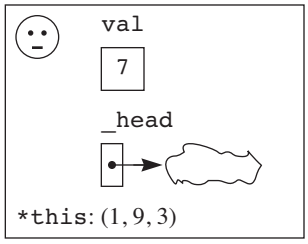
(b) `_head->max()`.



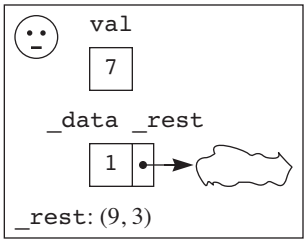
(c) `_rest->maxHelper(_data)`.



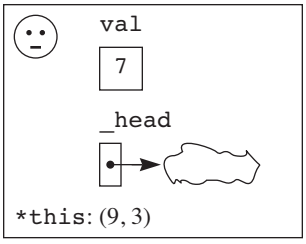
(d) `_head->maxHelper(val)`.



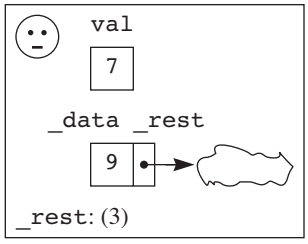
(e) `_rest->maxHelper(_data)`.



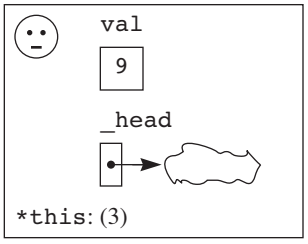
(f) `_head->maxHelper(val)`.



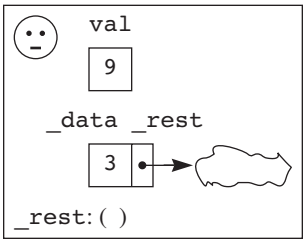
(g) `_rest->maxHelper(val)`.



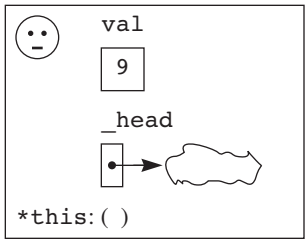
(h) `_head->maxHelper(val)`.



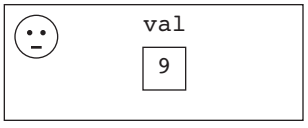
(i) `_rest->maxHelper(_data)`.



(j) `_head->maxHelper(val)`.



(k) `_rest->maxHelper(val)`.



(l) `_head->maxHelper(val)`.

Figure 6.31 An execution trace of `max()` for the `ListCS` data structure.

Part (c) shows the environment when the nonempty node calls the list version of `maxHelper()`. Now the environment includes not only a pointer to an abstract node, but also the formal parameter `val`, which has the value 2 of actual parameter `_data`.

Part (d) shows that the list simply passes the value to the node version of the helper. Here again, the parameter `val` is part of the environment along with attributes `_data` and `_rest`. The value of list `_rest` is (1, 9, 3) because that is the rest of the list `*this` in part (c). The code that executes in the nonempty node helper method is

```
return _data > val ?
    _rest->maxHelper(_data) : _rest->maxHelper(val);
```

which is where the comparison of two values occurs. In this scenario, `_data` has value 7 and `val` has value 2, and so

```
_rest->maxHelper(_data)
```

executes.

Part (e) shows the result of the execution. The recursive invariant is that `val` has the largest value from the first part of the list up until `*this`. In this case, 7 is the largest value from the first part of the list (2, 7, 1, 9, 3) up until (1, 9, 3).

Part (f) is the call by the list helper method to the node helper method and corresponds to part (d) in the previous iteration. The computation proceeds with pairs of mutually recursive calls. The list helper calls the node helper, which initiates the next cycle by calling the list helper. Sometimes the nonempty node helper calls the list helper with

```
_rest->maxHelper(val)
```

as in part (g), and sometimes the nonempty node helper calls the list helper with

```
_rest->maxHelper(_data)
```

as in part (i).

Part (l) shows the polymorphic dispatch to the empty node helper method when `*this` is empty in the list helper method. The empty node helper method simply executes

```
return val;
```

which returns the maximum value all the way up the call chain to the main program.

Figure 6.32 shows the implementation of output method `toStream()`. As with `ListL`, the `<<` operator is overloaded as in Figure 6.6, page 181 and is not shown here. Like `max()`, `toStream()` has a helper method, but unlike `max()` the helper method is not required for the list, but only the node.

There are no `if` statements and there is no loop. Polymorphic dispatch eliminates the `if` statements, and recursion eliminates the loop. The empty node version of `toStream()` streams `()` to `os`. On execution of

```
cout << myList;
```

with `myList` having value (2, 7, 9, 1, 3), the nonempty node version streams `(2` to `os`. In the environment of the nonempty node's `toStream()` method, `_rest` has value (7, 9, 1, 3). The call to the helper is polymorphically dispatched to the nonempty


```
// ========= toStream =========
template<class T>
void ListCS<T>::toStream(ostream &os) const {
    _head->toStream(os);
}

template<class T>
void MTcsNode<T>::toStream(ostream &os) const {
    os << "()";
}

template<class T>
void NEcsNode<T>::toStream(ostream &os) const {
    os << "(" << _data;
    _rest->_head->toStreamHelper(os);
}

// --------- toStreamHelper ---------
template<class T>
void MTcsNode<T>::toStreamHelper(ostream &os) const {
    os << ")";
}

template<class T>
void NEcsNode<T>::toStreamHelper(ostream &os) const {
    os << ", " << _data;
    _rest->_head->toStreamHelper(os);
}
```

Figure 6.32 Implementation of method `toStream()` for the `ListCS` data structure.

`toStreamHelper()` method, whose environment has a value of 7 and a `_rest` of (9, 1, 3). This method streams `, 7` to `os`.

The nonempty node version of `toStreamHelper()` continues to call itself recursively until `_rest` in its environment is the empty list. At that point, the empty node version gets called by polymorphic dispatch and streams the final `)` to `os`.

Like `max()` and `toStream()`, `equals()` also uses a helper method. Figure 6.33 shows the signatures and specifications for the `equals()` methods as does Figure 6.34 for the `equalsHelper()` methods.

In general, a helper method does processing similar to the original method, except that it has some extra data in its parameter list that affect the computation. For example, the specification of `max()` on page 206 states that it returns the maximum element of a list. But, the helper method returns the maximum element of the list and `val`, which is a parameter.

Similarly, the specification for `equals()` in Figure 6.33 states that it returns true


```
// ========= operator== =========
template<class T>
bool operator==(ListCS<T> const &lhs, ListCS<T> const &rhs) {
    return lhs.equals(rhs);
}

// ========= equals =========
// Post: true is returned if this list equals list rhs;
// Otherwise, false is returned.
// Two lists are equal if they contain the same number
// of equal elements in the same order.

template<class T>
bool ListCS<T>::equals(ListCS<T> const &rhs) const {
    cerr << "ListCS<T>::equals: Exercise for the student." << endl;
    throw -1;
}

template<class T>
bool MTcsNode<T>::equals(ListCS<T> const &rhs) const {
    cerr << "MTcsNode<T>::equals: Exercise for the student." << endl;
    throw -1;
}

template<class T>
bool NEcsNode<T>::equals(ListCS<T> const &rhs) const {
    cerr << "NEcsNode<T>::equals: Exercise for the student." << endl;
    throw -1;
}
```

Figure 6.33 Signatures and specification of method `equals()` for the `ListCS` data structure. Also shown is the overloaded == operator.

if the list is equal to parameter `rhs`. But, the specification for the helper in Figure 6.34 states that it returns true if `first` equals `this->first()` and `rest` equals `this->rest()`, where `first` and `rest` are the extra data in place of `rhs`.

In `max()`, the purpose of the extra parameter is to pass the previously computed maximum value to compare with the first element of the remaining part of the list that has not yet been processed. In `equals()`, the purpose of the extra parameters is to pass the first element and the rest of one list to test for equality with the first element and the rest of the other list.

Figure 6.35 is an execution trace of the statement

```
myList.equals(yourList)
```

when `myList` has values (4, 1, 7) and `yourList` has value (4). Implementation is an exercise, which requires code for the three `equals()` methods and the three helper


```
// --------- equalsHelper ---------
// Post: true is returned if first equals this->first()
// and rest equals this->rest();
// Otherwise, false is returned.

template<class T>
bool ListCS<T>::equalsHelper(T const &first, ListCS<T> const &rest) const {
    cerr << "ListCS<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}

template<class T>
bool MTcsNode<T>::equalsHelper(T const &first, ListCS<T> const &rest) const {
    cerr << "MTcsNode<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}

template<class T>
bool NEcsNode<T>::equalsHelper(T const &first, ListCS<T> const &rest) const {
    cerr << "NEcsNode<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}
```

Figure 6.34 Signatures and specification of methods `equalsHelper()` for the `ListCS` data structure. Implementation is left as an exercise.

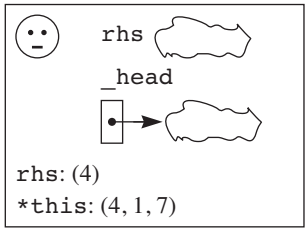
methods.

Each of the six methods has a local environment determined by its parameters and the attributes of its object. Figure 6.35(a) shows the environment after the original list call. Because the object is a list, the environment has its attribute `_head`. Because `rhs` is a parameter, the environment has it as well.

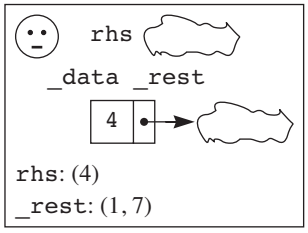
The list does not have direct access to any values, so it polymorphically dispatches the job to the node version in part (b). Because the object is a nonempty node, the environment has its attributes `_data` and `_rest`. It also has `rhs` as a parameter. The node now has access to the value 4 from `myList` in its `data` attribute. But, it does not have direct access to the first element of `yourList`. Fortunately, the helper returns true if `first` equals `this->first()` and `rest` equals `this->rest()`. So, it passes its `_data` and `_rest` to the list equals helper.

Part (c) shows the environment of the list equals helper. The object in this environment is `rhs` from the original call. This list object has the same problem that the original list object had in part (a). Namely, it does not have direct access to its first element to compare with `first`. So, it polymorphically dispatches the job to the nonempty node equals helper.

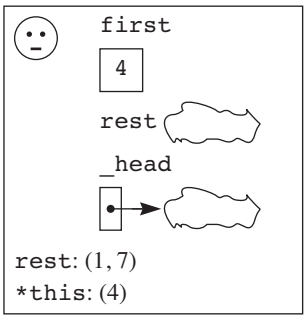
Part (d) shows the environment of this nonempty node, where comparison of the first elements of the two lists occurs. The object has access to its `_data` that it can compare with `first`. If these two values were not equal, no further recursive calls would be



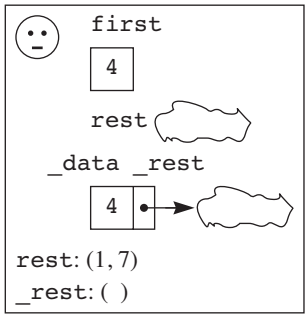
(a) List equals.



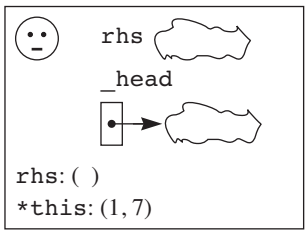
(b) Nonempty node equals.



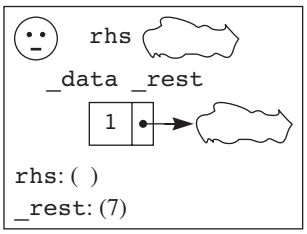
(c) List equals helper.



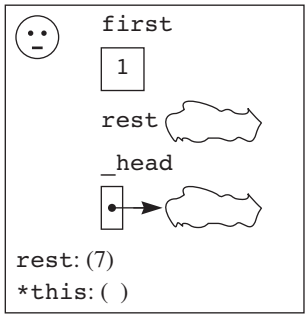
(d) Nonempty node equals helper.



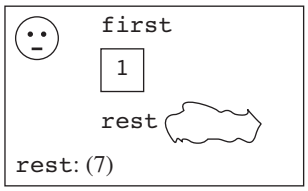
(e) List equals.



(f) Nonempty node equals.



(g) List equals helper.



(h) Empty node equals helper.

Figure 6.35 An execution trace of `equals()` for the `ListCS` data structure. Implementation is left as an exercise.

necessary, as the lists would not be equal. Because the two values are equal, the helper recursively calls the original list method with the two shorter lists.

The environment in part (e) is identical to that of part (a). The lists, however, are shorter by one element. Because `this` list is not empty, polymorphic dispatch yields the nonempty node environment of part (f). Although the nonempty node has access to its `_data` attribute, it does not have access to the first element of `rhs`.


```
// ========= Constructors =========
template<class T>
ListCS<T>::ListCS() :
    _head(new MTcsNode<T>()) {
}

// Post: _head points to node with no allocation.
template<class T>
ListCS<T>::ListCS(shared_ptr<AcsNode<T>> node) :
    _head(node) {
}

// Post: _data is data.
template<class T>
NEcsNode<T>::NEcsNode(T const &data) :
    _data(data) {
}

// Post: _data is data and _rest._head points to node.
template<class T>
NEcsNode<T>::NEcsNode(T data, shared_ptr<AcsNode<T>> node) :
    NEcsNode(data) {
    _rest->_head = node;
}

// Post: _data is rhs->_data and _rest is rhs->_rest.
template<class T>
NEcsNode<T>::NEcsNode(shared_ptr<const NEcsNode<T>> rhs) :
    _data(rhs->_data),
    _rest(rhs->_rest) {
}
```

The empty list created by `ListCS()`.

Figure 6.36 Constructors for `ListCS` and its nonempty node `NEcsNode`.

The list in part (g) has access to its `first` parameter, but it does not have direct access to its first element for comparison. So, it polymorphically dispatches the job to the empty node equals helper.

The environment for the empty node equals helper in part (h) is where the discovery is made that the lists are not equal. They cannot be equal in this environment, because the parameters are from a list that has at least one element, but this list is empty.

The remaining output and characterization methods are `first()`, `length()`, and `contains()`. Their implementations are also exercises at the end of this chapter.


```
// ========= prepend =========
template<class T>
void ListCS<T>::prepend(T const &data) {
    _head->prepend(*this, data);
}

template<class T>
void MTcsNode<T>::prepend(ListCS<T> &owner, T const &data) {
    // Assert: owner._head is a shared_ptr of this MTcsNode
    auto oldOwnerHead = owner._head;
    owner._head.reset(new NEcsNode<T>(data, oldOwnerHead));
}

template<class T>
void NEcsNode<T>::prepend(ListCS<T> &owner, T const &data) {
    auto oldOwnerHead = owner._head;
    owner._head.reset(new NEcsNode<T>(data, oldOwnerHead));
}
```

Figure 6.37 Implementation of `prepend()` for the `ListCS` data structure.

Construction and insertion

Figure 6.36 shows the two list constructors and three nonempty node constructors. As with the `ListL` data structure, the copy constructor for `ListCS` is disabled. The list constructor with no parameter initializes the list to the empty list, and the list constructor with a pointer to a node sets `_head` to point to the node with no allocation.

The figure also shows the empty list created by the first list constructor. Contrast this empty list with the empty list for `ListL` in Figure 6.2(a) on page 177. There is no occurrence of `nullptr` in the `ListCS` data structure. Instead of the `_next` field of the last node in `ListL` pointing to nothing with `nullptr`, the `_rest` field of the last node in `ListCS` is an empty list, whose `_head` points to an empty node.

A nonempty node constructor given a data value sets its `_data` attribute to the given value. A nonempty node constructor given a data value and a pointer to a node sets its `_data` attribute to the given value and its `_rest._head` to the given node. The node owns `_rest` and is responsible for its deallocation. A nonempty node constructor given list `rhs` does the same with the `_data` and `_rest` parts of `rhs`.

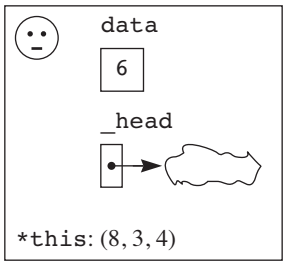
Figure 6.37 shows the implementation of `prepend()`. Figure 6.38 is an execution trace for the execution of

```
myList.prepend(6)
```

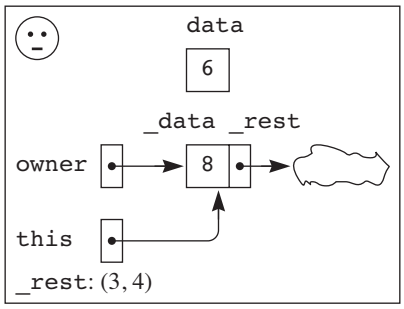
when `myList` has the values (8, 3, 4). Part (a) of the figure shows the environment of the list, which consists of its attribute `_head` and its parameter `data`.

Prepend is a method where the signature of the list method differs from the signature for the corresponding node method. The signature of the list method is

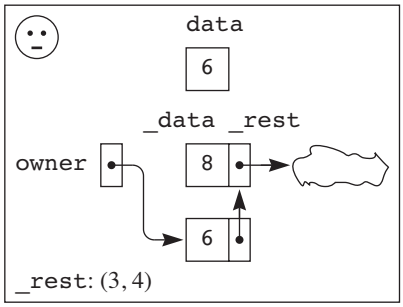
```
template<class T>
```

(a) `myList.prepend(6)`.



(b) `_head->prepend(*this, data)`



(c)
```
auto oldOwnerHead = owner._Head;
owner._head.reset(new NEcsNode<T>(data, oldOwnerHead));
```

Figure 6.38 An execution trace of `prepend()` for the `ListCS` data structure with a nonempty list.

```
void ListCS<T>::prepend(T const &data)
```

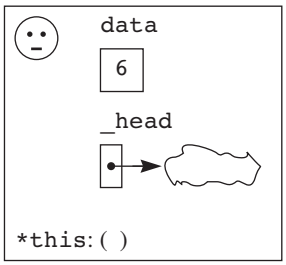
while the signature of the corresponding nonempty node method is

```
template<class T>
void NEcsNode<T>::prepend(ListCS<T> &owner, T const &data)
```

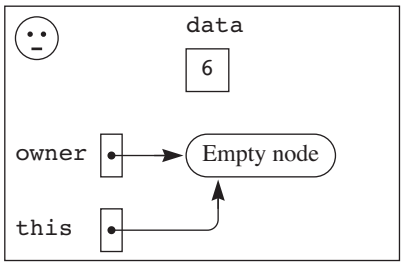
The additional formal parameter `owner` is a list, and corresponds to the actual parameter `*this` in the list method. The list method passes a reference to itself, because `prepend()` changes the list.

Part (b) shows the environment of the nonempty node, which contains its attributes `_data` and `_rest`. It also has `owner` and `data` because they are parameters. In C++, `this` is a pointer to the current object. Because part (b) shows the environment of a nonempty node, `this` is a pointer to the nonempty node that has 8 in its `_data` attribute.

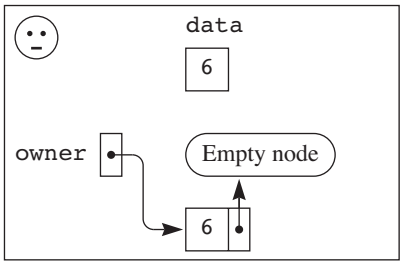
Part (c) shows the prepend operation. The nonempty node changes the `_head` field of its owner to point to a newly allocated nonempty node. The `new` operator allocates storage for a nonempty node then calls the constructor in Figure 6.36 with two parameters. The constructor sets the `_data` field of the new node to 6 and the `_rest._head` field of the new node to point to the same node to which `this` points. The nonempty node needs the reference to the original list in the `owner` parameter so it can change the state of the list.



(a) `myList.prepend(6)`.



(b) `_head->prepend(*this, data)`



(c)
```
auto oldOwnerHead = owner._Head;
owner._head.reset(new NEcsNode<T>(data, oldOwnerHead));
```

Figure 6.39 An execution trace of `prepend()` for the `ListCS` data structure with an empty list.

Figure 6.39 is an execution trace for the execution of

```
myList.prepend(6)
```

when `myList` is the empty list. Part (a) of the figure shows the environment of the list, which consists of its attribute `_head` and its parameter `data`.

The code for the empty node is identical to that of the nonempty node. The environment for the empty in part (b) has neither `_data` nor `_rest`, and `this` points to an empty node. Part (c) shows that the empty node simply assumes the role of the nonempty node in the previous scenario, which is why the code can be the same in both cases.

Figure 6.40 is an implementation of `copyHead()`. The nonempty node method uses the same constructor with two parameters that the prepend method uses. Instead of using a loop, the nonempty version calls `copyHead()` recursively in its actual parameter list.

The implementation of the overloaded assignment operator `operator=()` is identical to that of `ListL` in Figure 6.12 on page 188 and is not shown here. Implementation of insertion method `append()` is an exercise.

Destruction and removal

Removal method `clear()` of `ListCS` is an exercise. Clearly, no work is necessary for the empty node. The nonempty node recursively clears its `_rest` list, then resets its owner's `_head` to point to a new empty node. Only two lines of code in the nonempty node are required.


```
// ========= copyHead =========
// Post: A deep copy of the head of rhs is returned.
template<class T>
shared_ptr<AcsNode<T>> ListCS<T>::copyHead(ListCS<T> const &rhs) {
    return rhs._head->copyHead();
}

template<class T>
shared_ptr<AcsNode<T>>  MTcsNode<T>::copyHead() {
    return shared_ptr<MTcsNode<T>>(new MTcsNode<T>());
}

template<class T>
shared_ptr<AcsNode<T>>  NEcsNode<T>::copyHead() {
    return shared_ptr<NEcsNode<T>>(
        new NEcsNode<T>(_data, _rest->_head->copyHead()));
}
```

Figure 6.40 Implementation of `copyHead()` for the `ListCS` data structure.

Removal method `remFirst()` is an exercise. The empty node version implements the precondition. The nonempty node version saves the value to be returned in a local variable of type `T`, removes itself as the owner's first node, and then returns the saved value. Three lines of code in the nonempty node are required.

Both `remLast()` and `remove()` are exercises. You should implement method `remFirst()` before either of these, and then use `remFirst()` in the implementation of them.

Figure 6.41 shows the signatures and specifications for `remLast`, whose implementation is an exercise. As this algorithm uses a helper method, you must supply code for six different methods — a list version, an empty node version, and a nonempty node version for both the original method and the helper method. As with `remLast()` and `remove()`, you should implement `remFirst()` first and then use it in your implementation.

Figure 6.42 is an execution trace for the execution of

```
cout << myList.remLast()
```

when `myList` has values $(9, 5, 4, 6, 8)$. Part (a) is the environment of the list on the initial call. Because a list has the single attribute `_head` and the method has no parameters, `_head` is the only item in the list's environment. The list only knows that it has a pointer to an abstract node, and it does not know whether the head node is empty or is nonempty. So, it delegates the job polymorphically to the node.

Part (b) of the figure shows the environment of the nonempty node method. It has three things in its environment — `_data` and `_rest` because they are the the attributes of a nonempty node, and `owner` because it is passed as a parameter. As usual, the node method must have access to its owner in order to change the state of the owner. The nonempty node calls the list version of the helper method.


```
// ========= remLast =========
// Pre: This list is not empty.
// Post: The last element is removed from this list and returned.
template<class T>
T ListCS<T>::remLast() { ...

template<class T>
T MTcsNode<T>::remLast(ListCS<T> &owner) { ...

template<class T>
T NEcsNode<T>::remLast(ListCS<T> &owner) { ...

// --------- remLastHelper ---------
// Pre: previous.rest() is this list.
// Post: The last element of previous is removed and returned.
template<class T>
T ListCS<T>::remLastHelper(ListCS<T> &previous) { ...

template<class T>
T MTcsNode<T>::remLastHelper(ListCS<T> &owner, ListCS<T> &previous) { ...

template<class T>
T NEcsNode<T>::remLastHelper(ListCS<T> &owner, ListCS<T> &previous) { ...
```

Figure 6.41 The signatures and specifications of `remLast()` and its helper for the `ListCS` data structure. Implementation is left as an exercise.

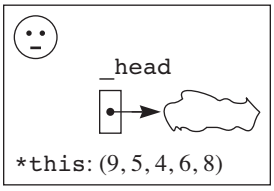
Part (c) shows the environment of the list helper method. Because this object is a list it has access to its attribute `_head`. The nonempty node method calls the list helper method in such a way that this list (5, 4, 6, 8) is the rest of the original list (9, 5, 4, 6, 8). Figure 6.41 shows a parameter named `previous` which is also available in the environment of the list helper. The precondition for this method is

```
// Pre: previous.rest() is this list.
```

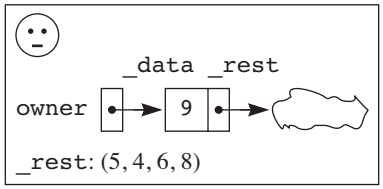
which is satisfied in part (c). As usual, the list does not know whether its `_head` is pointing to an empty or a nonempty node, so it calls the node helper with polymorphic dispatch.

Part (d) shows the environment of the nonempty node helper method. Because this object is a nonempty node it has access to its attributes `_data` and `_rest`. In addition, the object has access to two parameters in its environment, `owner` and `previous`. The list version of the helper calls the nonempty node version of the helper in such a way that `owner` is the owner of this node and `previous` is a list whose `_head` points to the nonempty node whose `_rest` is `owner`.

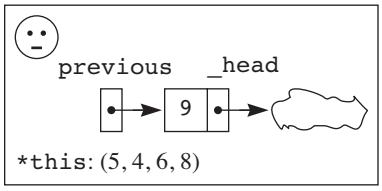
If this node were an empty node, it would be at the end of the original list. Furthermore, it would have access to `previous` and `owner` and would be able to remove the last element of the list. But, it is not an empty node, so it recursively calls the list version



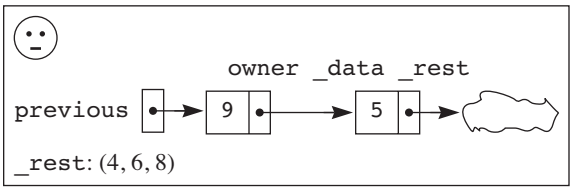
(a) List remove last.



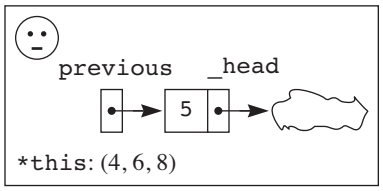
(b) Nonempty node remove last.



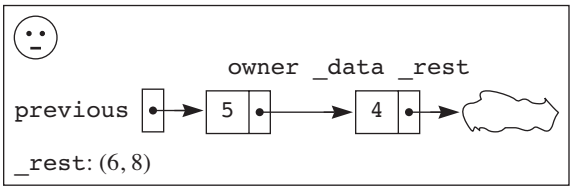
(c) List remove last helper.



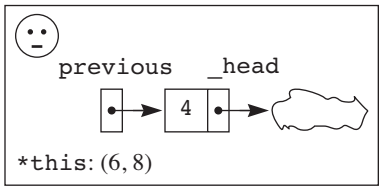
(d) Nonempty node remove last helper.



(e) List remove last helper.



(f) Nonempty node remove last helper.



(g) List remove last helper.

Figure 6.42 An execution trace of `remLast()` for the `ListCS` data structure. Implementation is left as an exercise.

of the helper method.

Part (e) shows the environment of the list helper. What corresponds to `owner` in part (d) corresponds to `previous` in part (e), and what corresponds to `rest` in part (d) corresponds to `_head` in part (e). Comparing parts (c), (e), and (g) shows that with each recursive call the computation gets one element closer to the end of the list when the deletion can finally take place.

Manipulation

Figure 6.43 shows the nonconstant version of `rest()`. Although the signatures differ, the code is identical to that of the constant version in Figure 6.29 on page 206. The difference between a constant and a nonconstant method that returns a reference to an object is that the nonconstant version may be used on the left side of an assignment to manipulate the data structure.

For example, the `ListL` data structure has a method named `setFirst()` that sets the first item to a given value. If `myList` is a `ListL`, to set the first element of the list to 99 you would execute


```
// ========= rest =========
template<class T>
ListCS<T> &ListCS<T>::rest() {
    return _head->rest();
}

template<class T>
ListCS<T> &MTcsNode<T>::rest() {
    cerr << "rest precondition violated: An empty list has no rest."
         << endl;
    throw -1;
}

template<class T>
ListCS<T> &NEcsNode<T>::rest() {
    return _rest;
}
```

Figure 6.43 Implementation of the nonconstant version of `rest()`.

```
myList.setFirst(99)
```

However, `ListCS` has no such method. Instead, it has a version of `first()` without the `const` designation in its signature. If `yourList` is a `ListCS`, to set the first element of the list to 99 you would execute

```
yourList.first() = 99
```

Implementation of the nonconstant version of `first()` is an exercise.

Implementation of manipulation method `concat()` is an exercise. The empty node version does the actual concatenation by setting the owner to the suffix and then setting the suffix to the empty list. The algorithm gets to the empty node by recursively concatenating the `suffix` list to the rest of the original list.

Figure 6.44 shows the specifications of `reverse()` and its helper, whose implementations are an exercise. If the original list empty then no work is necessary. If the original list is not empty then its nonempty node method sets up an empty list named `revList`. It then calls the list helper, which prepends `owner` to `revList` in reverse order, making `owner` empty. Then, it executes

```
owner = revList;
```

to give the reversed list back to `owner`.

For example, if `owner` has values (9, 2, 4, 7, 3) then the nonempty node method sets up a local `revList` with initial value ( ). After the call to the helper, `owner` has the value ( ), and `revList` has values (3, 7, 4, 2, 9). After the assignment to `owner`, `owner` has values (3, 7, 4, 2, 9). Because `revList` is a local variable it will be garbage collected when the `reverse()` method returns.


```
// ========= reverse =========
// Post: This list is reversed.
template<class T>
void ListCS<T>::reverse() { ...

template<class T>
void MTcsNode<T>::reverse(ListCS<T> &owner) { ...

template<class T>
void NEcsNode<T>::reverse(ListCS<T> &owner) { ...

// --------- reverseHelper ---------
// Post: This list is prepended to revList in reverse order,
// and this list is empty.
template<class T>
void ListCS<T>::reverseHelper(ListCS<T> &revList) { ...

template<class T>
void MTcsNode<T>::reverseHelper(ListCS<T> &owner, ListCS<T> &revList) { ...

template<class T>
void NEcsNode<T>::reverseHelper(ListCS<T> &owner, ListCS<T> &revList) { ...
```

Figure 6.44 The signatures and specifications of `reverse()` and its helper for the `ListCS` data structure. Implementation is left as an exercise.

The helper's job is to prepend `this` list to `revList` in reverse order, which it does recursively one element at a time. For example, suppose part way through the computation that `this` has values $(4, 7, 3)$ and `revList` has values $(2, 9)$. Because $(4, 7, 3)$ is not empty more work needs to be done. The algorithm removes first element 4 from $(4, 7, 3)$ and prepends it to $(2, 9)$ yielding the lists $(7, 3)$ and $(4, 2, 9)$. The values of the lists evolve from the beginning as follows.

$(9, 2, 4, 7, 3)$	$( )$
$(2, 4, 7, 3)$	(9)
$(4, 7, 3)$	$(2, 9)$
$(7, 3)$	$(4, 2, 9)$
(3)	$(7, 4, 2, 9)$
$( )$	$(3, 7, 4, 2, 9)$

The specification of `zip()` for `ListCS` is identical to that for `ListL` in Figure 6.17 on page 193. The Composite pattern makes possible a short elegant implementation with no helper methods. The idea is to switch the roles of `this` and `other` with each recursive call.

For example, if `this` has values $(1, 2, 3, 4, 5)$ and `other` has values $(6, 7, 8)$ the final value of `this` should be $(1, 6, 2, 7, 3, 8, 4, 5)$. If `this` were empty, it could be set to `other` with the assignment statement


```
owner = other;
```

followed my making `other` the empty list. But, it is not empty. So, in the environment of the nonempty node, `owner` is the list (1, 2, 3, 4, 5), `_rest` is the list (2, 3, 4, 5), and `other` is the list (6, 7, 8). Now, switch the roles and recursively zip (6, 7, 8) into (2, 3, 4, 5) yielding (6, 2, 7, 3, 8, 4, 5) for `other` and (1) for `owner`. Then, use `setRest()` to set the rest of `owner` to `other`.

The specification of `unZip()` for `ListCS` is identical to that for `ListL` in Figure 6.17 on page 193. The implementation of this method is unusual because, even though the list is changed, the node methods do not require an `owner` parameter. No `owner` parameter is required because `head` of the original list still points to the same node, and hence does not change.

For example, if the original list is (1, 2, 3, 4, 5, 6, 7) then after the call it is (1, 3, 5, 7) and a pointer is returned to the list (2, 4, 6). The modified list still has 1 as its first value, and so `head` of the original list does not change. As with `zip()`, the Composite pattern makes possible an elegant implementation with no helper methods.

If the original list is empty you can simply return a pointer to a new empty list. But, in the above example, it is not empty. So, its nonempty node exists in an environment where `_rest` is the list (2, 3, 4, 5, 6, 7). The algorithm sets up a pointer to a local temporary list named `resultOfRestUnzip` by recursively unzipping `_rest`. The temporary list then has values (3, 5, 7), and in the process `_rest` is changed to (2, 4, 6).

The algorithm must return a pointer to a list, so it allocates a new empty list from the heap and gives the shared pointer to a local `result` variable. It then executes the assignment statement

```
*result = *_rest;
```

to set `result` to (2, 4, 6), a deep copy because the assignment is a list assignment. One execution of `setRest()` gives the owner's `_rest` its proper values. The algorithm terminates by returning `result`.

6.4 The Composite State Visitor Pattern

The Visitor object-oriented design pattern is the basis of plugin software architectures. Examples are plugins for browsers that allow you to add capabilities to the browser, and plugins for image processing applications that allow you to add various effects to images. Many integrated development environments (IDEs) are based on plugin architectures.

The advantage of a plugin architecture is the ability to add features to the application without modifying and re-compiling the class library. Because the original library does not need to be changed to add new features, the plugins can even be supplied by third-party developers. The ability for third-party developers to add features to an application frees the developer of the core application from having to anticipate and build in every feature that the users might want. Users can customize the core application by their selection of plugins.

The problem that must be solved in plugin architectures is how to allow a third-party developer to write an arbitrary method that has access to the underlying data structure of the application without knowing in advance the signature of the method the user wants to use. One user might want a non-void method with no parameters that returns a value

of some type, while another user might want a void method with several parameters. It would be impossible for the developer of the core application to supply a library of blank methods for the third-party developer to implement having every possible combination of all the possible types of parameters and return values that the user might want. The Visitor design pattern solves this problem.

The dp4ds Composite State Visitor list

The `ListCSV` data structure combines the Composite and State patterns of the `ListCS` data structure with the Visitor pattern. Imagine that the `ListCS` data structure is a commercial application and the features of the application are the methods of the UML class diagram for `ListCS` in Figure 6.26 on page 203. Suppose some user requests a method to determine if the values in the list are sorted. As there is no method named `isSorted()` that returns type boolean, your only alternative with `ListCS` is to write the additional method, recompile the application, and release an update including the new feature.

Figure 6.45 is a UML diagram for `ListCSV` that uses a plugin architecture. `ListCSV` does not have the extensive list of methods that `ListCS` has. Instead, it has a minimal complete set of operations that any plugin developer can use to build the desired plugin. The minimal complete set includes `first()`, `prepend()`, `remFirst()`, `rest()`, and `setRest()`. The methods in `ListCS` that are missing in `ListCSV` are implemented as plugins. So, if a user wants a new method that determines if the list is sorted, that feature can be implemented as a plugin with no modification to the core application.

No library of data structures in any collection other than dp4ds implements a linked list with the Composite State Visitor pattern. The sole purpose for presenting it here is to teach the object-oriented Visitor design pattern, which is common in large applications but not in data structures.

In addition to the core application, the core library provides the separate abstract visitor class `AListCSVVis`, which stands for "abstract `ListCSV` visitor". A plugin is an object that inherits from this abstract class. Figure 6.45 shows three plugins that implement the functions `isEmpty()`, `clear()`, and `max()` in `ListCS`. In `ListCSV`, they are not part of the core application. All the methods of `ListCS` are in `ListCSV`. Many are not shown in Figure 6.45.

The abstract visitor class is provided by the core application, so the application's code can be compiled knowing the signatures of its methods. The plugin developer also has access to the abstract visitor class. When a concrete visitor inherits from the abstract visitor, it implements the abstract methods of the visitor. The four methods implemented by the plugin developer are the empty case and nonempty case for the constant and nonconstant lists.

```
virtual void emptyCase(ListCSV<T> &host) = 0;
virtual void nonEmptyCase(ListCSV<T> &host) = 0;
virtual void emptyCase(ListCSV<T> const &host) = 0;
virtual void nonEmptyCase(ListCSV<T> const &host) = 0;
```

In C++ terminology, these are pure virtual methods. Each of these methods takes a `ListCSV` as a parameter.

Figure 6.46 is the implementation of the special `accept()` methods of the core application. The plugin code calls the list version of the `accept()` method, which

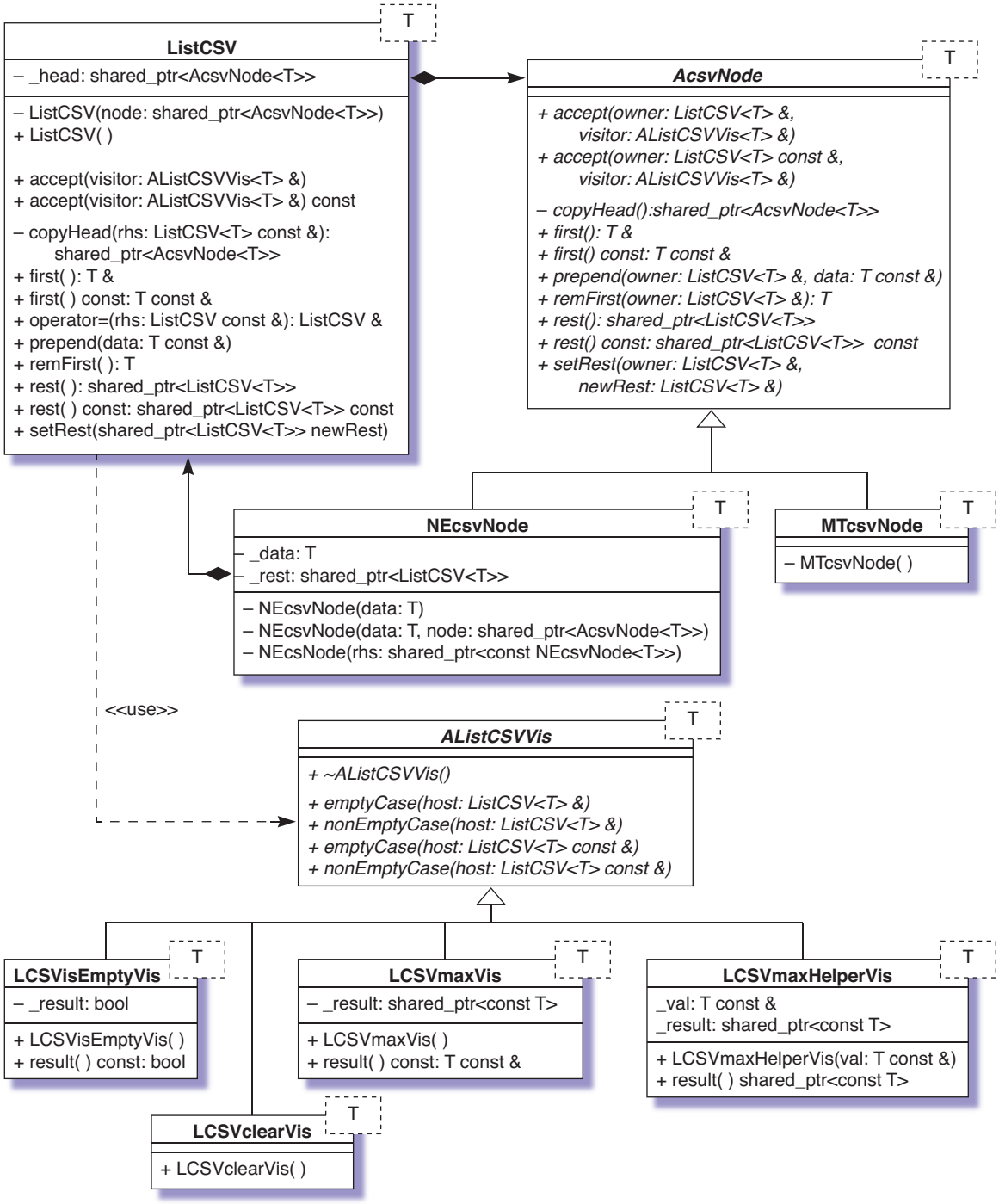


Figure 6.45 The UML class diagram for the `ListCSV` data structure. It combines the Composite, the State, and the Visitor OO design patterns.


```
// ========= accept =========
template<class T>
void ListCSV<T>::accept(AListCSVVis<T> &visitor) {
    _head->accept(*this, visitor);
}

template<class T>
void MTcsvNode<T>::accept(ListCSV<T> &owner, AListCSVVis<T> &visitor) {
    visitor.emptyCase(owner);
}

template<class T>
void NEcsvNode<T>::accept(ListCSV<T> &owner, AListCSVVis<T> &visitor) {
    visitor.nonEmptyCase(owner);
}

// ========= accept const =========
template<class T>
void ListCSV<T>::accept(AListCSVVis<T> &visitor) const {
    _head->accept(*this, visitor);
}

template<class T>
void MTcsvNode<T>::accept(ListCSV<T> const &owner, AListCSVVis<T> &visitor) {
    visitor.emptyCase(owner);
}

template<class T>
void NEcsvNode<T>::accept(ListCSV<T> const &owner, AListCSVVis<T> &visitor) {
    visitor.nonEmptyCase(owner);
}
```

Figure 6.46 Implementation of `accept()` in `ListCSV`.

then calls the empty node or the nonempty node version of `accept()` via polymorphic dispatch. The list version of `accept()` cannot know at compile time whether the visitor will change the list. So, it does not know whether it needs to pass itself with `*this` as `owner` to the node version. Passing `*this` as `owner` for all visitors does no harm for those that do not modify the list and makes possible the modification for those that do.

The relationship between `ListCSV` and `AListCSVVis` is neither one of inheritance nor of class composition. Instead, the dashed line in Figure 6.45 is the UML designation for the *dependency* relation. Class `ListCSV` depends on class `AListCSVVis` because the methods `accept()` in `ListCSV` use parameters of type `AListCSVVis`. The designation «use» next to the dependency arrow is the UML standard way to specify this type of dependency. In UML terminology, `ListCSV` is the client of the dependency and `AListCSVVis` is the supplier of the dependency. A dependency exists when a change

in the supplier can affect the client. If any of the code in `AListCSVVis` changes, those changes would affect `ListCSV`, and `ListCSV` would have to be recompiled.

This list plugin architecture fixes the signatures of `accept()` in `ListCSV` and of `emptyCase()` and `nonEmptyCase()` in `AListCSVVis`. Once these signatures are fixed, the core application is compiled once and for all and the `AListCSVVis` abstract class is made available to the plugin developers. The code for the methods in the core application — `copyHead()`, `first()`, `operator=()`, `prepend()`, `remFirst()`, `rest()`, `setRest()` — is identical to the code in `ListCS` and is not repeated here.

There are three visitor techniques that govern how a native method in the original monolithic class translates to a visitor method in a plugin architecture.

- Input parameters in the native method are attributes in the visitor initialized in the constructor for the visitor.
- Output parameters in the native method are reference variable attributes in the visitor.
- Nonvoid methods in the native class require an additional method named `result()` that returns the same type.

These three techniques solve the problem of not being able to anticipate all the possible signatures that a user would want for a new method. When you subclass an abstract class you are free to add whatever attributes you want to the concrete class. So, the plugin developer can add an attribute to the concrete visitor for each parameter in the original method, and those attributes are transparent to the core application. Furthermore, for nonvoid methods the plugin developer is free to write a `result()` method to return whatever type is needed, which is also transparent to the core application.

Given the original method in the large monolithic original application, it is a mechanical translation to write the visitor version. The following sections show how to translate a method to its corresponding visitor version.

The isEmpty( ) visitor

Figure 6.47 shows the implementation of the `isEmpty()` visitor `LCSVisEmptyVis`. Code for the constant version is identical to the code for the nonconstant version and is not shown. It is a translation of `isEmpty()` for `ListCS` in Figure 6.28 on page 205. Figure 6.28 (a) shows the environment of a list when the user calls the method from within an `if` statement as follows.

```
if (myList.isEmpty()) {
```

If `myList` has values (4, 1, 7) the nonempty node gets called by polymorphic dispatch. Within the environment of part (c) the node object knows to return false to the list.

The calling sequence is different when you use the visitor version. With this visitor, the equivalent calling code is the following.

```
LCSVisEmptyVis<T> isEmptyVis;
myList.accept(isEmptyVis);
if (isEmptyVis.result()) {
```

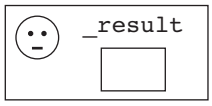
The first line declares a local visitor named `isEmptyVis`. The second line passes the visitor as a parameter to the `accept` method of `myList`. The effect of this call is to set


```
// ========= LCSVisEmptyVis =========
template<class T>
class LCSVisEmptyVis : public AListCSVVis<T> {
private:
    bool _result; // Output result.

public:
    // ========= Constructor =========
    LCSVisEmptyVis() = default;

    // ========= visit =========
    void emptyCase(ListCSV<T> &host) override {
        _result = true;
    }

    void nonEmptyCase(ListCSV<T> &host) override {
        _result = false;
    }

    // ========= result =========
    // Pre: This visitor has been accepted by a host list.
    // Post: true is returned if the host list is empty;
    // Otherwise, false is returned.
    bool result() const {
        return _result;
    }
};

// Global function for convenience.
template<class T>
bool isEmpty(ListCSV<T> const &list) {
    LCSVisEmptyVis<T> isEmptyVis;
    list.accept(isEmptyVis);
    return isEmptyVis.result();
}
```

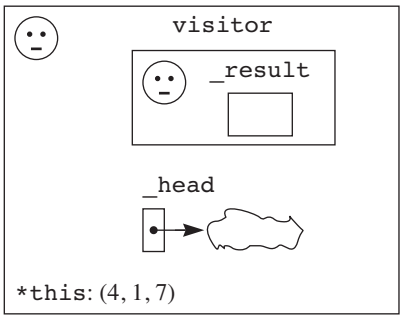
Figure 6.47 Implementation of the `LCSVisEmptyVis` plugin in `ListCSV`. Code for the constant version is identical to the code for the nonconstant version and is not shown.

an attribute in the visitor to false if `myList` has values (4, 1, 7). The third line uses the result of the computation by calling the `result()` method of the visitor.

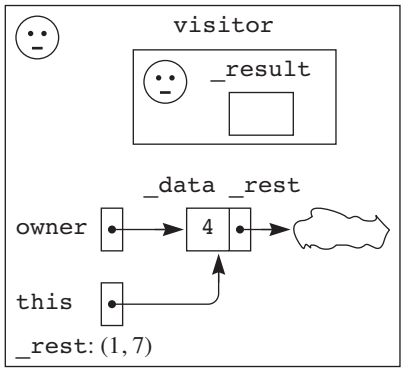
Figure 6.48 shows the environments in the method calls during execution with the visitor to test if a `ListCSV` is empty. Part (a) shows the environment of the visitor object `isEmptyVis` on its allocation as a local variable with



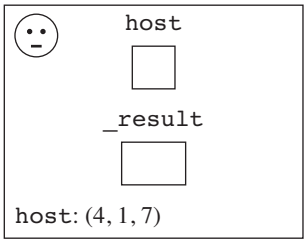
(a) The environment of visitor `isEmptyVis`.



(b) The environment of `myList` after `myList.accept(isEmptyVis)`.



(c) The environment of the nonempty node after `_head->accept(*this, visitor)`.



(d) The environment of the visitor after `visitor.nonEmptyCase(owner)`.

Figure 6.48 The environments in the method calls during execution with the visitor to test if a `ListCSV` is empty.

```
LCSVisEmptyVis<T> isEmptyVis;
```

It has access to its attribute `_result`. Because its constructor in Figure 6.47 does no initialization, the value of its attribute is not set.

Part (b) shows the environment of `myList` after execution of the method call

```
myList.accept(isEmptyVis);
```

The environment of an object during execution of one of its methods consists of its attributes and its parameters, as these are the items to which it has access to carry out its computation. Because `myList` is a `ListCSV` it has access to its attribute `_head` and its parameter named `visitor`, which corresponds to the actual parameter `isEmptyVis`. Because `visitor` is itself an object, its environment is passed along in the call.

Part (c) shows the polymorphic dispatch call to the nonempty node with

```
_head->accept(*this, visitor);
```


As is the case with the `ListCS` class, the nonempty node has access to its attributes `_data` and `_next` and its parameter `owner` as the object that owns this node. In addition, it has access to parameter `visitor`.

Part (d) shows the environment of `visitor` after execution of

```
visitor.nonEmptyCase(owner);
```

The nonempty node can access `visitor` because `visitor` is a parameter of its method. Because `visitor` is an object, it has its own set of methods that it can execute. So, the nonempty node executes the `nonEmptyCase()` of `visitor`. The nonempty node knows that `visitor` has a method named `nonEmptyCase()` because all concrete subclasses of `AListCSVVis` must implement such methods.

The environment of `visitor` has access to `_result` because it is an attribute, and it has access to `host` because it is the formal parameter whose actual parameter corresponds to `owner` in part (c), the environment of the nonempty node. The name `host` is more appropriate than the name `owner` because the visitor is not a nonempty node that has an owner. The nonempty node in part (c) has access to `_data` and `_rest` as attributes, but the visitor in part (d) has no such direct access. However, it does have access to `host` as a parameter, and it can access them indirectly as `host.first()` and `host.rest()`.

Now the following code that the plugin developer wrote in `nonEmptyCase()` executes.

```
_result = false;
```

The resulting environment looks the same as part (d), but with `false` in the cell for `result`. Control then returns to the nonempty node with an environment like part (c), but with `false` in the cell for `result`. It returns similarly to the `accept()` method of the list and then to the calling environment, which looks like part (a), but with a value of `false` for `_result`.

The user of the visitor now executes

```
if (isEmptyVis.result()) {
```

to test if the list is empty. The `result()` method of the visitor simply returns attribute `_result`, which was stored as the result of the computation.

In general, to use this visitor is a three step process.

- Declare a local visitor.
- Pass it as a parameter in the data structure's `accept()` method.
- Call the visitor's `result()` method to get the result of the computation.

The third step is not required for visitors that correspond to `void` methods. Rather than require the user to perform these steps, the visitors in the dp4ds distribution supply a global convenience function that does the necessary setup and call. Figure 6.47 shows the implementation of the convenience function `isEmpty()` for the `LCSVisEmptyVis` visitor. With this convenience function, the user simply executes

```
if (isEmpty(myList)) {
```


with all the details of the visitor protocol hidden. This global function cannot be a method of the list, because then the core application would have to supply it, which would defeat the whole purpose of the plugin architecture design.

Setting up the environment of Figure 6.48(d) is the heart of plugin architectures. The core application defines the abstract visitor and gives the plugin developer one or more abstract methods to implement to perform the computation of the plugin. A key feature of the architecture is that the plugin developer writes code for a method, but does not write the code that calls the method. In `ListCSV`, the plugin developer completes the code for `emptyCase()` and `nonEmptyCase()` but the core application calls these methods from its node classes. This calling protocol is known as a *callback* or a *hook*. Systems that use this protocol are known as *frameworks*.

Some mobile app development systems are in essence similar frameworks. You can think of the app as a plugin for the mobile device. In these frameworks, the Integrated Development Environment (IDE) sets up the main program for the app, usually an event loop, which is invisible to the developer. The developer then completes the methods that the framework calls.

The toStream( ) visitor

Figures 6.49 and 6.50 are the implementation of the visitor version of `toStream()` for `ListCS` in Figure 6.32 on page 210. The diagram in Figure 6.49 shows the environment of the helper as a result of the programmer executing

```
cout << myList
```

with `myList` having values (4, 1, 7). As with the `LCSVisEmptyVis` visitor in Figure 6.48(d), the `LCSVtoStreamVis` visitor has access to `host` and to its attribute, in this case `_os`. While the environment is different for the empty node versus the nonempty node in `ListCS`, the environments are the same for the empty case and the nonempty case in `ListCSV`. Both cases have access to `host` and the attributes of the visitor.

The convenience function is `operator<<()` with formal parameter `os` corresponding to actual parameter `cout`. The function sets up the local visitor with

```
LCSVtoStreamVis<T> toStreamVis(os);
```

which in turn calls the constructor for the visitor. Then, in the constructor, `_os` is initialized via the initializer list as follows.

```
LCSVtoStreamVis(ostream &os): _os(os) { }
```

Now the visitor has a reference to `cout` in the reference variable `_os` as the diagram in Figure 6.49 shows.

The general technique for a method with a parameter is to make the parameter an attribute in the visitor and initialize the attribute in the constructor. The only difference between call by reference and call by value is that in call by reference the attribute is a reference variable.

The original `toStream()` method for `ListCS` has the code

```
os << "(" << _data;
_rest._head->toStreamHelper(os);
```



```
// ========= LCSVtoStreamVis =========
template<class T>
class LCSVtoStreamVis : public AListCSVVis<T> {
private:
    ostream &_os; // Output parameter.

public:
    // ========= Constructor =========
    explicit LCSVtoStreamVis(ostream &os):
        _os(os) {
    }

    // ========= visit =========
    // Pre: This LCSVtoStreamVis visitor has been accepted by a host list.
    // Post: A string representation of this list is returned.
    void emptyCase(ListCSV<T> &host) override {
        _os << "()";
    }

    void nonEmptyCase(ListCSV<T> &host) override {
        _os << "(" << host.first();
        LCSVtoStreamHelperVis<T> toStreamHVis(_os);
        host.rest()->accept(toStreamHVis);
    }
```

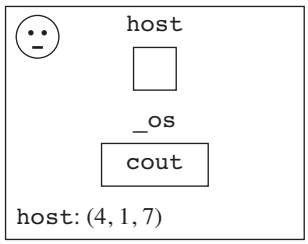


Figure 6.49 Implementation of the `LCSVtoStreamVis` plugin in `ListCSV`. Code for the constant version is identical to the code for the nonconstant version and is not shown. The diagram shows the environment of the visitor, which applies to both the empty case and the nonempty case. The listing continues in the next figure.

for the nonempty node. The corresponding code for the nonempty case for `ListCSV` is

```
_os << "(" << host.first();
LCSVtoStreamHelperVis<T> toStreamHelperVis(_os);
host.rest()->accept(toStreamHelperVis);
```

for the visitor.

The first lines are identical except for two differences. The visitor uses attribute `_os` where the original uses parameter `os`, and the visitor uses `host.first()` where the original nonempty node uses `_data`.

The second line of the original calls the helper for `_rest` passing `os` as a parameter. Because the helper is itself a visitor, this action takes two lines in the visitor — one to set up the helper visitor and initialize the output parameter, and one to call `accept()`. Attribute `_rest` in the original corresponds to `host.rest()` in the visitor. These two examples illustrate the following general rule from `ListCS` to `ListCSV`.

- Use `host.first()` in a `ListCSV` visitor to correspond to `_data` in a `ListCS` nonempty node.


```
// ========= LCSVtoStreamHelperVis =========
template<class T>
class LCSVtoStreamHelperVis : public AListCSVVis<T> {
private:
    ostream &_os; // Output parameter.

public:
    // ========= Constructor =========
    explicit LCSVtoStreamHelperVis(ostream &os):
        _os(os) {
    }

    // ========= visit =========
    // Pre: This LCSVtoStreamHelperVis visitor has been accepted
    // by a host list.
    // Post: A string representation of this list is returned
    // except for the leading open parenthesis "(", which is omitted.
    void emptyCase(ListCSV<T> &host) override {
        _os << ")";
    }

    void nonEmptyCase(ListCSV<T> &host) override {
        _os << ", " << host.first();
        host.rest()->accept(*this);
    }
};

// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, ListCSV<T> const &rhs) {
    LCSVtoStreamVis<T> toStreamVis(os);
    rhs.accept(toStreamVis);
    return os;
}
```

Figure 6.50 (continued) Implementation of the `LCSVtoStreamVis` helper and convenience function in `ListCSV`. Code for the constant version is identical to the code for the nonconstant version and is not shown. This concludes the implementation.

- Use `host.rest()` in a `ListCSV` visitor to correspond to `_rest` in a `ListCS` nonempty node.

The original `toStream()` helper method for `ListCS` has the code

```
_rest._head->toStreamHelper(os);
```

for the nonempty node. This is a recursive call to the helper method. Normally, the translation of this call would require two lines in the visitor version — one to set up the


```
// ========= LCSVclearVis =========
template<class T>
class LCSVclearVis : public AListCSVVis<T> {
public:
    // ========= Constructor =========
    LCSVclearVis() = default;

    // ========= visit =========
    // Pre: This visitor has been accepted by a host list.
    // Post: This list is cleared to the empty list.
    void emptyCase(ListCSV<T> &host) override {
    }

    void nonEmptyCase(ListCSV<T> &host) override {
        host.rest()->accept(*this);
        host.remFirst();
    }

    // ========= visit const =========
    void emptyCase(ListCSV<T> const &host) override {
        cerr << "LCSVclearVis precondition violated: "
             << "Cannot clear a const list." << endl;
        throw -1;
    }

    void nonEmptyCase(ListCSV<T> const &host) override {
        cerr << "LCSVclearVis precondition violated: "
             << "Cannot clear a const list." << endl;
        throw -1;
    }
};

// Global function for convenience.
// Post: list is cleared to the empty list.
template<class T>
void clear(ListCSV<T> &list) {
    LCSVclearVis<T> clearVis;
    list.accept(clearVis);
}
```

Figure 6.51 Implementation of the `LCSVclearVis` plugin in `ListCSV`.

helper visitor and one to call its `accept()` method. The corresponding code for the nonempty case for `ListCSV` is

```
host.rest().accept(*this);
```



```
// ========= LCSVmaxVis =========
template<class T>
class LCSVmaxVis : public AListCSVVis<T> {
private:
    shared_ptr<const T> _result;  // Output result.

public:
    // ========= Constructor =========
    LCSVmaxVis() = default;

    // ========= visit =========
    void emptyCase(ListCSV<T> &host) override {
        cerr << "max precondition violated: "
             << "An empty list has no maximum." << endl;
        throw -1;
    }

    void nonEmptyCase(ListCSV<T> &host) override {
        LCSVmaxHelperVis<T> maxHelperVis(host.first());
        host.rest()->accept(maxHelperVis);
        _result = maxHelperVis.result();
    }

    // ========= result =========
    // Pre: This visitor has been accepted by a host list.
    // Pre: This list is not empty.
    // Post: The maximum element of this list is returned.
    T const &result() const {
        return(*_result);
    }
};
```

Figure 6.52 Implementation of the `LCSVmaxVis` plugin in `ListCSV`. Code for the constant version is identical to the code for the nonconstant version and is not shown. The implementation continues in the next figure.

for the visitor. It consists of only one line, and illustrates a visitor optimization technique. In this implementation, there is no need to declare a local helper visitor on the runtime stack with its own local attribute `_os` because that attribute would simply reference the original stream object. Because `this` is a pointer to the current object, and the current object is a helper visitor, `*this` is the current visitor and has an `accept()` method. The one-line code recursively calls its own accept method instead of setting up an additional local visitor and calling its accept method. This optimized code is both shorter to write and faster to execute.


```
// ========= LCSVmaxHelperVis =========
template<class T>
class LCSVmaxHelperVis : public AListCSVVis<T> {
private:
    T const &_val; // Input parameter.
    shared_ptr<const T> _result; // Output result.

public:
    // ========= Constructor =========
    explicit LCSVmaxHelperVis(T const &val):
        _val(val) {
    }

    // ========= visit =========
    void emptyCase(ListCSV<T> &host) override {
        _result = make_shared<const T>(_val);
    }

    void nonEmptyCase(ListCSV<T> &host) override {
        if (host.first() > _val) {
            LCSVmaxHelperVis<T> maxHelperVis(host.first());
            host.rest()->accept(maxHelperVis);
            _result = maxHelperVis.result();
        }
        else {
            LCSVmaxHelperVis<T> maxHelperVis(_val);
            host.rest()->accept(maxHelperVis);
            _result = maxHelperVis.result();
        }
    }
```

Figure 6.53 (continued) Implementation of the `LCSVmaxVis` helper and convenience function in `ListCSV`. Code for the constant version is identical to the code for the nonconstant version and is not shown. The implementation continues in the next figure.

The clear( ) visitor

Figure 6.51 is the implementation of the `clear()` visitor `LCSVclearVis`. It is the translation of a `void` method with no parameters, and so it has no attributes. It also requires no `return()` method for the global convenience function to call.

This visitor corresponds to the removal method `clear()` of `ListCS`. Any method that removes items from a list or modifies a list must implement the precondition for constant lists, which cannot be changed. Accordingly, the code for a constant list in Figure 6.51 does not duplicate the code for the nonconstant list. Instead, it aborts the program with an appropriate error message.


```
    // ========= result =========
    // Pre: This LCSVmaxHelperVis visitor has been accepted by a host list.
    // Post: A shared_ptr of the maximum element of this list is returned.
    shared_ptr<const T> result() const {
        return _result;
    }
};

// Global function for convenience.
template<class T>
T const max(ListCSV<T> const &list) {
    LCSVmaxVis<T> maxVis;
    list.accept(maxVis);
    return maxVis.result();
}
```

Figure 6.54 (continued) Implementation of the `LCSVmaxVis` helper and convenience function in `ListCSV`. Code for the constant version is identical to the code for the nonconstant version and is not shown. This concludes the implementation.

The max( ) visitor

Figures 6.52, 6.53, and 6.54 show the implementation of the `max()` visitor. Code for the constant version is identical to the code for the nonconstant version and is not shown.

The implementation uses all the techniques of the previous examples. In the visitor, `host.first()` corresponds to `_data`, and `host.rest()` corresponds to `_rest`. The helper has an input parameter `val`, and so the visitor helper has a corresponding attribute `_val` that is initialized with an initializer list in its constructor.

An important feature of a plugin architecture is the additional level of abstraction that it provides for the programmer. Each of the dp4ds distribution projects of this chapter are at successive levels of abstraction where more details are hidden. The `ListL` project is at the lowest level. Code for linked lists at this level manipulates pointers, tests for `nullptr`, and distinguishes between nodes and lists. The `ListCS` project is at the next higher level. Direct manipulation of pointers is reduced and there are no references to `nullptr`, which is subsumed by the concept of an empty node. The `ListCSV` project is at the highest level of abstraction. Plugin developers have no need to access nodes at all, as all computations can be accomplished with methods `first()` and `rest()` and the insertion and deletion methods.

Exercises

6–1 What are the best case and worse case $\Theta(n)$ execution times, where n is the number of elements in the list, of the following `ListL` methods? **(a)** `append()`. **(b)** `concat()`, where m is the number of elements in `suffix`. **(c)** `contains()`. **(d)** `equals()`, where m is the number of elements in `rhs`. **(e)** `first()`. **(f)** `isEmpty()`. **(g)** `length()`. **(h)** `prepend()`. **(i)** `remFirst()`. **(j)** `remLast()`. **(k)** `remove()`.

(l) `reverse()`. **(m)** `unZip()`. **(n)** `zip()`, where *m* is the number of elements in `other`.

6–2 Complete the implementation of the following member functions for the `ListL` class in Figure 6.3. The dp4ds distribution software contains unit tests for each method, which you should use to test the correctness of your implementation.

(a) `first()`. Remember to implement the precondition.

(b) `length()`.

(c) `max()`. Remember to implement the precondition.

(d) `contains()`.

(e) `equals()`. See page 182 for a discussion of the implementation.

(f) `append()`. See Figure 6.11 for the action of `append()`.

(g) `copyHead()`. See Figure 6.13 for the action of `copyHead()`.

(h) `remFirst()`. Remember to implement the precondition. See page 191 for a discussion of the implementation.

(i) `remLast()`. Remember to implement the precondition. See Figure 6.16 for the action of `remLast()`.

(j) `remove()`. This method has no precondition. See page 191 for a discussion of the implementation.

(k) `concat()`. See Figure 6.19 for the action of `concat()`.

(l) `reverse()`. See Figure 6.20 for the action of `reverse()`.

(m) `zip()`. See Figure 6.21 for the action of `zip()`.

(n) `unZip()`. See Figure 6.22 for the action of `unZip()`.

6–3 Complete the implementation of the following member functions for the `ListL` class using the `ListLIterator` class in Figure 6.24. There must be no use of any pointers, including `nullptr`, in any of your code. The dp4ds distribution software contains unit tests for each method, which you should use to test the correctness of your implementation.

(a) `length2()`.

(b) `max3()`. Remember to implement the precondition.

6–4 Complete the implementation of the following member functions for the `ListCS` class in Figure 6.26. The dp4ds distribution software contains unit tests for each method, which you should use to test the correctness of your implementation.

(a) `first()`, both the constant and the nonconstant versions. Remember to implement the precondition.

(b) `length()`.

(c) `contains()`.

(d) `equals()`. See Figure 6.35 for the action of `equals()`.

(e) `append()`.

(f) `clear()`. See page 217 for a discussion of the implementation.

(g) `remFirst()`. Remember to implement the precondition. See page 218 for a discussion of the implementation.

(h) `remLast()`. See Figure 6.41 for the specifications and signatures of the methods and Figure 6.42 for a trace of the algorithm. Remember to implement the precondition.

(i) `remove()`. This method has no precondition. See page 218 for a discussion of the implementation.

(j) `concat()`. See page 221 for a discussion of the implementation.

(k) `reverse()`. See Figure 6.44 for the specifications and signatures of the methods.

(l) `zip()`. See page 222 for a discussion of the implementation.

(m) `unZip()`. See page 223 for a discussion of the implementation.

6–5 Incorporate your solution for `first()` and `remFirst()` from `ListCS` into the core application `ListCSV` in Figure 6.45. Then, complete the implementation of the following visitors for the `ListCSV` class. The dp4ds distribution software contains unit tests for each visitor, which you should use to test the correctness of your implementation.

(a) `LCSVlengthVis`, the visitor version of `length()`.

(b) `LCSVcontainsVis`, the visitor version of `contains()`.

(c) `LCSVequalsVis`, the visitor version of `equals()`.

(d) `LCSVappendVis`, the visitor version of `append()`.

(e) `LCSVremLastVis`, the visitor version of `remLast()`.

(f) `LCSVremoveVis`, the visitor version of `remove()`.

(g) `LCSVconcatVis`, the visitor version of `concat()`.

(h) `LCSVreverseVis`, the visitor version of `reverse()`.

(i) `LCSVzipVis`, the visitor version of `zip()`.

(j) `LCSVunZipVis`, the visitor version of `unZip()`.

