CHAPTER 19

Linked Lists

19.1 Dynamic Data Structure Concepts
19.2 Linked Lists
19.3 MATLAB Implementation of Linked Lists
19.3.1 The LLNode Class
19.3.2 The LinkedList Class
19.4 Processing Linked Lists Recursively
19.5 Implementing Linked List Methods in MATLAB
19.5.1 Building a Linked List
19.5.2 Traversing a Linked List
19.5.3 Mapping a Linked List
19.5.4 Filtering a Linked List
19.5.5 Folding a Linked List
19.5.6 Searching a Linked List
19.6 Applications of Linked Lists
19.6.1 Queues
19.6.2 Stacks
19.6.3 Priority Queues

Chapter Objectives

This chapter discusses:

- Dynamic data structures in general
- A simple dynamic data structure—the linked list
- The MATLAB implementation of a linked list
- Useful implementations of linked lists: queues, stacks, and priority queues

19.1 Dynamic Data Structure Concepts

Having dealt with the concept of modeling specific concrete or abstract objects, we turn to the process of defining dynamically sized collections of objects. The goal is to provide mechanisms that organize collections containing any kind of object—MATLAB basic classes, arrays or structures, cell arrays, or instances of our own classes as defined by the rules discussed in Chapter 18.

We will consider three concepts important to dynamic data structures: static memory allocation on the activation stack, dynamic memory allocation from the heap, and dynamically linking objects to create dynamic structures.

19.1.1 Static Memory Allocation

In Chapter 1 we discussed the allocation of memory to application programs on a processor in general. The first of these approaches is the activation stack (or simply the stack). The stack is used to allocate memory frames that contain data local to a script or function call. Each overall application program (such as MATLAB) is allocated one memory block for use as its stack. Each time a function is called, a frame of memory is allocated from the stack; when the function terminates, that memory is released from the stack as discussed in Chapter 9.

For example, in the middle of a solution to Programming Project 4 in Chapter 5, the activation stack might look like that shown in Figure 19.1. The script containing the variables `u`, `s`, and `g` calls the function roots with parameters `A`, `B`, and `C`. In the core of that function, the square root of a value is called. This would be the state of the stack before the `sqrt` function completes. The shaded blocks indicate static data storage on the stack for the variables allocated in the script and functions.

19.1.2 Dynamic Memory Allocation

The second source of memory discussed in Chapter 1 is the heap, which is a single block of memory available to all the applications to be allocated and deallocated from an application upon request. For example, when we created a `BankAccount` object for test purposes in Chapter 18, the command was as follows:

```
moola = BankAccount(1000)
```

sqrt x =	815.2					
roots A =	49.05	B =	-40	C =	40	
Ex6_1 u =	40	s=	40	g =	9.81	

Figure 19.1 *An activation stack*

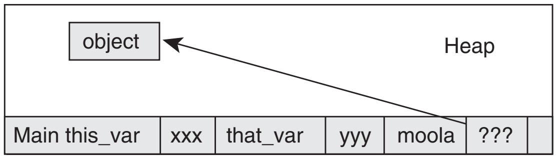

Figure 19.2 *Dynamically allocated memory*

Figure 19.2 illustrates what actually happens when that command is executed:

1. A block of memory large enough to hold the data structure for a `BankAccount` is requested from the heap
2. The data are initialized by the code in the constructor
3. The variable `moola` actually becomes a pointer, or reference, to that block of memory
4. The allocated memory is retained as long as there is a reference somewhere (not necessarily the same one)
5. When the last reference to that memory block is destroyed, the block is returned to the operating system's heap

We cannot leave this concept without considering the "ordinary" variables in Figures 19.1 and 19.2. Although it is convenient to introduce them initially as if they were statically allocated,[1] recall that all MATLAB entities are in fact instances of classes. Even single numbers are 1×1 arrays. Each of the variables illustrated should in fact be viewed in the same style as the variable `moola`, as shown in Figure 19.2.

19.1.3 Dynamically Linked Memory

The power of these structures becomes evident when you consider the "simple" structure shown in Figure 19.3. References to objects on the heap are not confined to stack frames. Dynamically created objects can contain references to other similar, or dissimilar, objects. This chapter will demonstrate the implementation of the simplest of these structures—the linked list.

19.2 Linked Lists

Linked lists are linear data structures implemented exactly as shown in Figure 19.3. Before we examine the details of this implementation, it is important to identify a watershed in the code presentation style used in this text. To this point, we have illustrated examples by showing all the code necessary to

[1] This is true in Java where primitive variables of type `int`, `double`, and `Boolean` are in fact stored on the stack.

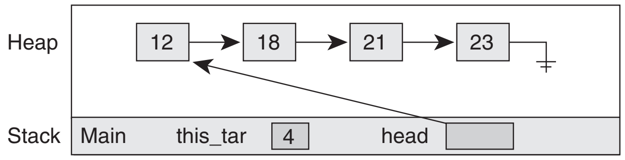

Figure 19.3 *Dynamically linked data*

implement the examples. So far this has been possible because the code has been quite simple, and necessary because there was no good metaphor for summarizing the code. Now the code will be more complex, but we have a metaphor for describing the code more concisely than including all the listings—the ADT.

Here we will review the ground rules under which we will use the ADTs as metaphors for the bulk of the routine code,[2] freeing us to concentrate on the methods of general interest. Consider one of the two basic classes necessary for implementing linked lists: the `LLNode` class, as shown in Figure 19.4. When we include an ADT in this form, the implementation implied by its presence will be as follows:

- A class with the given name (`LLNode`) stored in the appropriate directory (`@LLNode`)
- Data items with the names and order shown in the ADT
- Access methods with names built as shown, capitalizing the name of the data items
- A constructor expecting initial values of the data items in the order indicated (`data` then `next`); if any of the data items are not provided to the constructor, by default the values will be set to null (`[ ]`)
- Other "expected" methods (`char` or `display` in this example) may be discussed if they have any interesting content
- Other public methods will be identified on the left side of the ADT and the code presented and discussed

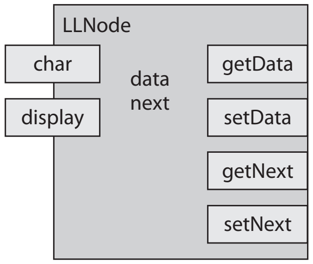

Figure 19.4 *ADT for the* `LLNode`

[2] The necessary source code is available at the Addison-Wesley Instructor Resource Center (www.aw.com/irc).

19.3 MATLAB Implementation of Linked Lists

Two classes are necessary to implement linked lists. The links themselves are all instances of the `LLNode` class. The container of all the linked list methods is a `LinkedList` class.

19.3.1 The LLNode Class

The purpose of the `LLNode` class is to contain references to data items and connect them in a linear list. The data items can be of any type, with two possible constraints. If the data are of type `double`, we have to make a special case in the `char(...)` method. Some methods that process the contents of the linked list may put other requirements on the contained objects.

Hint

When you are debugging code by stepping through it with the debugger, it is easy to become confused about the class to which the current method belongs. You can make life easier in this respect by always putting the class to which this method belongs in the documentation lines. Here, we adopt the convention of specifying the directory containing the method in the second line of the method definition, as shown in Listing 19.1.

Since the `LLNode` class has no interesting methods, according to the ground rules mentioned earlier, we need to include no code listings except the `char(...)` method, as shown in Listing 19.1. Here, we have to make a special case of contents with class `double` because `char(d)` when `d` is type `double` attempts to convert that `double` value to the ASCII code equivalent. If `d` is of any other type, `char(d)` will convert it to a string as needed.

In Listing 19.1:

Lines 1–3: Show the typical method header.
Line 4: Determines whether the node contains a double value.
Line 5: If so, it uses the `%g` conversion to obtain a general string conversion.
Line 7: Otherwise, calls the `char(...)` method on the object data.

19.3.2 The LinkedList Class

The purpose of the `LinkedList` class is to contain the head of the linked list and all the methods for processing the list. Figure 19.5 shows the ADT of the

Listing 19.1 `LLNode char(...)` method

```
1. function s = char(n)
2. % @LLNode\char(n) is the string representation
3. %                     of n.data
4. if isa( n.data, 'double')
5.     s = sprintf('%g', n.data);
6. else
7.     s = char(n.data);
8. end
```

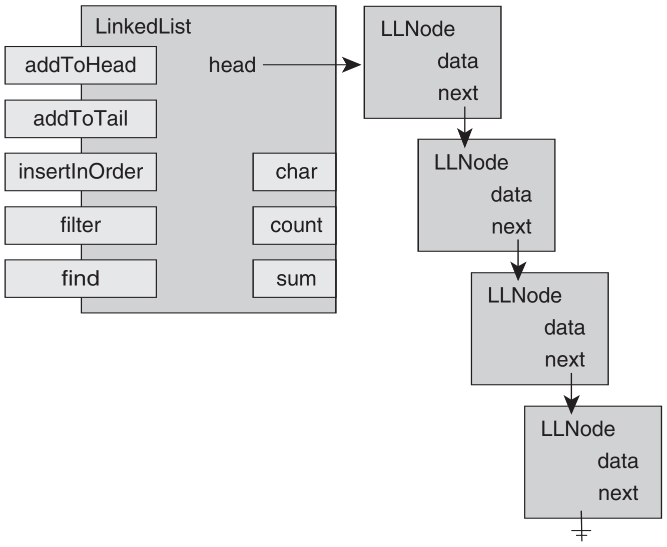

Figure 19.5 *ADT for the* `LinkedList` *class*

`LinkedList` class and its relationship to the `LLNode` class. The relationship is one of containment (HAS-A), not inheritance (IS-A). We will consider the methods of the `LinkedList` class in the context of the taxonomy of collection operations first introduced in Chapter 10.

19.4 Processing Linked Lists Recursively

We could view the definition of a linked list as one of the following:

- A linked list is empty (`[ ]`)
- A linked list is an `LLNode` containing some data and another linked list

Since the definition of a linked list is recursive in the same style as n! (see Chapter 9), we could expect that we should be processing the list recursively. Further, we could argue that we could model the template of a function for

Template 19.1 Template for processing a linked list

```
<function and return> processNode(here)
% recursive processing
if isempty(here)
      <reached the end>
else
      ... getData(here) ...
      ... processNode( getNext(here) ) ...
end
```

processing a linked list after that same structure, as shown in Template 19.1. See the notes that follow.

- The parameter passed to the recursive function could be named `here` to remind us that the recursive program is moving through the list
- We test for the empty state with MATLAB's built-in function `isempty(...)`
- There will be some actions to take when we reach the end of the list
- Until then, we may do something with the `data` value in the current node, and then recursively process using the `next` value of the node

We might also consider the view shown in Figure 19.6, where processing the whole list is accomplished by repetitively processing the values of a single node:

- Check for the end of the list (`here` is null)
- Make the recursive call via `next`
- Combine the result from the recursive call with the `data` at this node
- Return the result to the calling program

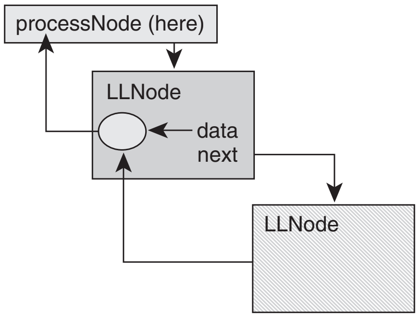

Figure 19.6 *Recursively processing a linked list*

19.5 Implementing Linked List Methods in MATLAB

This section discusses the implementation of the methods of a linked list and some derived classes.

19.5.1 Building a Linked List

There are three techniques for building a linked list. The one you use depends on how you intend the data to be inserted. The most straightforward is adding to the head of the list. However, this presents the last data item added as the first to be seen when traversing or searching the list. Consequently, we will also consider adding to the tail of the list and inserting in order. While adding to the head or tail of the list imposes no constraints on the nature of the data in the list, adding in order presumes that we have a means for comparing two of the objects being inserted.

Adding to the Head of a List Listing 19.2 shows the method for adding to the head of a list. As long as the user has supplied the parameter data, the method creates a new `LLNode` containing the `data` provided and the existing `head` of the list and makes that the new list `head`.

In Listing 19.2:

Lines 1–4: Show the method header.

Line 5: Checks that the caller supplied the data.

Line 6: All the logic for adding to the head of a list relates to the head of the list; therefore, recursion is not necessary. The `setHead(...)` and `getHead(...)` methods will be part of the class definition, but not specifically listed here. We create a new `LLNode` containing the `data` provided with the original list as its `next` field, and place this as the new head of the list.

Line 7: Returns the new linked list.

Adding to the Tail of a List Theoretically it is possible to add to the tail of a linked list either iteratively or recursively. Recall the “myopic view” illustrated in Figure 19.6, and then apply that view when adding to the tail of a list. The `addToTail` method in the `LinkedList` class merely passes the head of the list to its recursive local function, `addToTailR`, and stores the result as the new head of the list when it returns. The helper function performs the addition of the data according to the logic described below.

All the processing is done by the recursive module using the myopic view illustrated in Figure 19.6 and Template 19.1. Consider, for example, the existing list shown in Figure 19.7.

If we wanted to add the value 10 to the tail of the list, the first recursive call comes by way of the head of the list to the first node, the one containing

Listing 19.2 Adding to the head of a linked list

```
1. function addToHead(ll, data)
2  % @LinkedList\addToHead.
3. %   addToHead(ll, data) adds data to
4. %             the head of a LinkedList object
5.     if nargin > 1
6.         setHead(ll, LLNode(data, getHead(ll) ));
7.         assignin('caller', inputname(1), ll);
8.     end
```

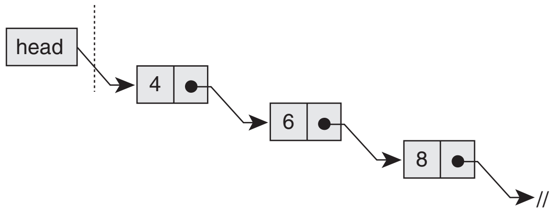

Figure 19.7 *The initial list*

the value 4 as shown in Figure 19.8. Since we are seeking the end of the list, the recursive calls continue.

Finally, the recursive calling reaches the end of the list, as illustrated in Figure 19.9. The action at the end of the list is to create a new node with the data provided (the 10) and null for the next node.

Each recursive call then returns to its caller a copy of its original node linked to the emerging list provided from the recursive call. Finally, as shown in Figure 19.10, the new list fully formed is provided to the wrapper function that stores the new list as its head value.

Hint

At first glance, this program looks a little strange. You might ask why we do not merely replace the final `next` field with a new `LLNode` containing the new data. One of the frustrations of working with MATLAB is its insistence on passing parameters by value so that every function and every method is working with a copy of the parameters passed. So it is certainly possible to replace that last `next` field in a copy of the last node, but not in the node itself. That copy must be returned to the caller and stored, and the circular argument continues back to the head of the list. When you follow that logic to its sad conclusion, you have to replace every node in every dynamic data structure that is changed by the current algorithm.

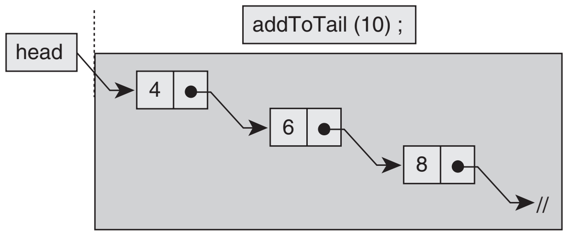

Figure 19.8 *The first recursive call*

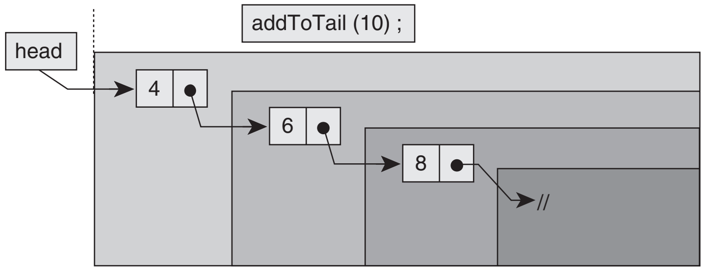

Figure 19.9 *At the end of the list*

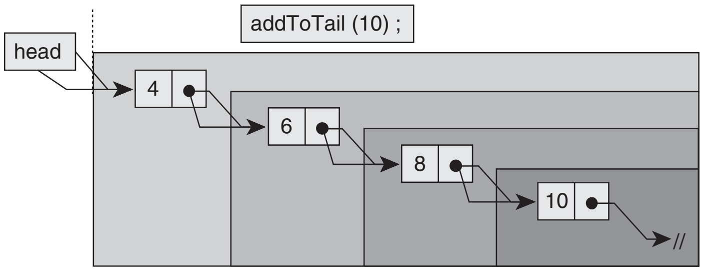

Figure 19.10 *Returning the new list*

Listing 19.3 shows the recursive method for adding to the tail of a linked list.

In Listing 19.3:

Lines 1–4: Show the method header.

Line 5: Makes sure the caller provided the `data` parameter.

Line 6: Passes the head of the list to the recursive helper function and sets the returned list as the new list head.

Line 7: Returns the new `LinkedList` object to the caller.

Line 9: Shows the recursive helper function. Every exit from this function must return an `LLNode` object.

Line 11: Checks for the terminating condition: the end of the list.

Line 12: Creates the new node for the end of the list.

Lines 14 and 15: Create a new node containing the data from the original node and the result of another recursive call as the next field.

Listing 19.3 Adding to the tail of a linked list

```
 1. function addToTail(ll, data)
 2. % @LinkedList\addToTail.
 3. %   addToTail(ll, data) adds data to the
 4. %                   tail of a LinkedList object
 5. if nargin > 1
 6.     setHead( ll, addToTailR(getHead(ll), data) );
 7.     assignin('caller', inputname(1), ll);
 8. end

 9. function nl = addToTailR(here, data)
10. % recursive add to tail
11. if isempty(here)
12.     nl = LLNode(data);
13. else
14.     nl = LLNode( getData(here), ...
15.                  addToTailR(getNext(here), data));
16. end
```

Adding to the List in Order Adding to the list in order is structurally similar to adding to its end, since we must allow for the possibility that the data must be added at its end. It must, however, also allow for finding a link that must come after the new data in the list. In this case, the method builds a new node containing the new data item, with the complete list from here to the end as the next link.

This process also places an additional demand on the data in the list. Whereas any data item can be added to the head or tail of the list, if the list is to be ordered, each data item must be able to respond to the test `A >= B`. Clearly, numbers and characters (including vectors) are equipped to do so. User-defined classes are enabled to respond to this operator by including the method `ge(A, B)`. Listing 19.4 shows the `ge(...)` method that enables a `BankAccount` (and any of its child classes) to be compared to another account, or to a numerical value.

In Listing 19.4:

> Lines 1–3: Show the method header.
>
> Line 4: Checks to see if the second parameter is a `BankAccount` object or one of its children.
>
> Lines 6–8: Establish the value for comparing—either the balance of the given account or the data provided (presumed to be a number).
>
> Line 10: Returns the comparison. This is not recursive because the >= operator is applied here to numerical values for which this operator is built in.

As with `addToTail(...)`, the `addInOrder(...)` method hands the head of the list to a recursive local function and retrieves the new head of the list. Listing 19.5 shows the method for adding to a list in order. This method also uses a recursive helper function.

In Listing 19.5:

> Lines 1–4: Show the method header.
>
> Lines 5–7: Call the recursive helper function and return the result.

Listing 19.4 `ge(...) operator for the BankAccount` class

```
 1. function ans = ge(acct, data)
 2. % @BankAccount\ge
 3. % compare this account to another or to a number
 4. if isa( data, 'BankAccount')
 5.                 % children also respond to this
 6.     comparison = getBalance(data);
 7. else
 8.     comparison = data;
 9. end
10. ans = (getBalance(acct) >= comparison);
```

Listing 19.5 Linked list `addInOrder` method

```
 1. function addInOrder(ll, data)
 2. % @LinkedList\addInOrder.
 3. %   ll = addInOrder(ll, data) adds data in order
 4. %                             in a LinkedList object
 5. if nargin > 1
 6.     setHead( ll, addInOrderR(ll.head, data) );
 7.     assignin('caller', inputname(1), ll);
 8. end

 9. function nl = addInOrderR(here, data)
10. % recursive add in order
11. if isempty(here)
12.     nl = LLNode(data);
13. elseif getData(here) >= data
14.     nl = LLNode(data, here);
15. else
16.     nl = LLNode( getData(here), ...
                 addInOrderR(getNext(here), data));
17. end
```

Lines 9 and 10: Show the recursive helper function—a minor adaptation of `addtoTail(...)`.

Lines 11 and 12: Show the original terminating condition if this part of the list is empty. Recall that if the second parameter is not supplied to the `LLNode` constructor, its `next` field is presumed to be empty.

Lines 13 and 14: Show the added terminating condition where we discover that the current node contains data that ought to be ahead of the new data in the list. Here we create a new node with the remains of the list represented by `here` as its next field.

Line 16: Otherwise, we continue recursively down the list as before.

19.5.2 Traversing a Linked List

Since a traversal does not change the list, it is possible to perform a list traversal iteratively or recursively. In order to illustrate iterative processing on a list, we will implement the `char(...)` method on a `LinkedList` class iteratively using the `while` loop shown in Template 19.2.

Template 19.2 Iterative `LinkedList` processing template

```
<function and return> processLinkedList(theList)
% iterative processing
<initialize the loop exit test>
<initialize the data result>
while <stay in the loop>
      <process the data>
      <move forward>
```

continued on next page

```
end
<return the results>
```

Listing 19.6 shows the `char(...)` method code that directly matches Template 19.2:

- This function returns a character string `s`.
- The loop will move the variable here through the nodes of the list; its initial value will be the head of the list.
- The initial value of the result string is set to the identifier `'LL: '`.
- The loop continues as long as the `LLNode` reference here is not empty.
- We process the data by appending the `char(...)` conversion of the `LLNode` followed by a semicolon. Note that by asking for the `char(...)` of the `LLNode` rather than its data contents, we make it responsible for always returning a string. This permits the `LinkedList` to contain numbers—usually a problem because `char(n)` where `n` is class double will attempt an ASCII conversion of `n`.

In Listing 19.6:

Lines 1–3: Show the method header.

Line 4: Initializes the iterative variable that will move down the list.

Line 5: Initializes the output string.

Line 6: Uses the built-in `isempty(...)` test to terminate the iteration.

Line 7: Invokes the `char(...)` cast of the `LLNode` and concatenates the output with the emerging result.

Line 8: Moves down the list.

19.5.3 Mapping a Linked List

In general, mapping a list is performed not as a general service to all `LinkedList` users, but rather as a specialized utility written for derived classes that uses a `LinkedList` as their parent class. For example, suppose a bank

Listing 19.6 The `LinkedList char(...)` method

```
1. function s = char(ll)
2. % @LinkedList\char
3. % char(ll) is its string representation
4. here = getHead(ll);
5. s = 'LL: ';
6. while ~isempty(here)
7.     s = [s char(here) ';'];
8.     here = getNext(here);
9. end
```

keeps all its interest-bearing accounts in a class derived from the `LinkedList` class. Periodically, the bank needs to traverse that list and add the interest generated by each account. Because the length of the list remains unchanged but the contents do change, it is a mapping of the original list.

Listing 19.7 illustrates such an update method.

In Listing 19.7:

Lines 1–4: Show the method header.

Line 5: Computes the interest.

Line 6: Updates the account balance.

Line 7: Returns the new account object.

In order to update all the accounts in a list of `SavingsAccounts`, the mapping would follow the recursive template shown in Template 19.1 with the following tailoring required:

- As usual, the public method, `updateList`, merely launches the recursive helper, `updateR`, with the current head of the list and stores the new list returned as the new head
- The helper returns an empty list when it reaches the end of the old list
- Otherwise, it returns a new `LLNode` containing the result of updating the old data, using the recursive `updateR` call to update the rest of the list

The resulting code is shown in Listing 19.8.

Listing 19.7 `SavingsAccount update(...)` method

```
1. function update(acct)
2. % @SavingsAccount\update
3. %    update to the account by depositing
4. %    the interest due
5.     amount = calcInterest(acct);
6.     setBalance(acct, getBalance(acct) + amount);
7.     assignin('caller', inputname(1), acct);
```

Listing 19.8 Mapping a linked list

```
function updateList(acctList)
% @LinkedList\update
%    update to the list items by calling
%    its update method.
 setHead( acctList, updateR(getHead(acctList)) );
 assignin('caller', inputname(1), acctList);
```

continued on next page

```
 7. function newLst = updateR(here)
 8. % recursive update helper
 9. if isempty(here)
10.    newLst = [];
11. else
12.     item = getData(here);
13.     update(item);
14.     newLst = LLNode( item, ...
15.                      updateR( getNext(here) ) );
16. end
```

In Listing 19.8:

Lines 1–4: Show the method header.

Line 5: Invokes the recursive helper function and replaces the list head.

Line 6: Returns the new list object.

Lines 7 and 8: Show the header for the helper function.

Lines 9 and 10: At the end of the list, return empty—we are not adding to the list.

Lines 12 and 13: Extract and update the item. Recall that since the update function tunnels back, it must be called with a simple variable.

Lines 14 and 15: Create a new node with the updated item and the usual recursive call.

Style Points

It is really important as you build more complex programs to use good abstraction. In the illustration of mapping, it is tempting to incorporate the update function into the `updateList` method because both are rather simple in form. However, the effort that creates separate, single-purpose functions is rewarded in code that is both easy to understand and reusable.

19.5.4 Filtering a Linked List

As with mapping a list, filtering can be performed only as a service to specialized collections derived from the `LinkedList` class. The contents of the derived class must provide a method `keep(...)` that determines whether to keep a particular item on the list.

For example, we continue the idea of a list of bank accounts in a `LinkedList`. Since we are not making any use of child account features, we can use any type of `BankAccount` and store the `keep(...)` method in the `BankAccount` class. Perhaps the bank wishes to remove all accounts with a negative balance for this list. They would merely have to provide the `keep(...)` method shown in Listing 19.9 and then implement the generic `LinkedList` filter.

Listing 19.9 `BankAccount keep(...)` method

```
1. function ans = keep(acct)
2. % @BankAccount\keep
```

continued on next page

```
 3. %    keep an account with a non-negative balance
 4.     ans = getbalance(acct) >= 0;
```

In Listing 19.9:

Lines 1–3: Show the method header.

Line 4: Shows the logic for keeping an account on the list. In more complex situations, this test can be tailored to a specific set of requirements.

To filter items in a generic list, we will again follow the recursive template shown in Template 19.1 with the following tailoring required:

- As usual, the public method, `filter`, launches the recursive helper, `filterR`, with the current head of the list and stores the new list returned
- The helper returns an empty list when it reaches the end of the old list
- Otherwise, if it should keep this data item, it returns a new `LLNode` containing the old data, using the recursive `updateR` call to update the rest of the list
- If it should not keep this data item, it returns the result from applying `updateR` recursively to the rest of the list

The resulting code is shown in Listing 19.10.

In Listing 19.10:

Lines 1–6: Show the usual wrapper function to set up the recursion.

Lines 7–9: Show the filtering helper function header.

Listing 19.10 Generic `LinkedList filter` method

```
 1. function filter(ll)
 2. % @LinkedList\filter
 3. %    filter the list by removing all items
 4. %    returning false from their keep method
 5.     setHead( ll, filterR(getHead(ll)) );
 6.     assignin('caller', inputname(1), ll);

 7. function newLst = filterR(here)
 8. % recursive filter helper
 9. if isempty(here)
10.     newLst = [];
11. elseif keep(getData(here))
12.     newLst = LLNode( getData(here), ...
                         filterR( getNext(here) ) );
13. else
14.     newLst = filterR( getNext(here) );
15. end
```

Lines 9 and 10: Show the terminating condition returning empty.

Line 11: Checks whether we need to keep this item.

Lines 11 and 12: If so, we make a new node with the current item and the recursive call.

Line 14: Otherwise, we delete the current item by returning only the result of the recursive call without creating a new node.

19.5.5 Folding a Linked List

Unlike the previous list operations, folding can be performed as a service in the `LinkedList` class. We illustrate this by the `sum(...)` method. It constrains the contents of the list to those classes that respond to the method `plus(...)`. In general, `plus(...)` permits the programmer to add the current object to any other object. We will assume here that if the second object is not the same class as ours, it is a `double` value.

We continue the idea of a list of bank accounts in a `LinkedList`. Since we are not making any use of child account features, we can use any type of `BankAccount` and store the `plus(...)` method in the `BankAccount` class. Perhaps the bank wishes to sum all accounts in this list. They would merely have to provide the `plus(...)` method shown in Listing 19.11, and then implement the generic `LinkedList` sum.

Hint

We have used two specific examples of a general principle. The statement `'A >= B'` results in MATLAB calling `ge(A, B)`, and `'A + B'` results in a call to `plus(A, B)`. In general, all MATLAB operators have a "shadow" function that is actually called when the operator is applied. See MATLAB help for a full listing of these functions.

In Listing 19.11:

Lines 1–5: Show the method header.

Line 6: Checks whether the addend is a `BankAccount` (or child thereof).

Line 7: If so, extracts its balance.

Line 9: Otherwise, assumes the data item is a number.

Line 11: Returns the required sum.

Listing 19.11 `BankAccount plus(...)` method

```
 1. function ans = plus(ba, item)
 2. % @BankAccount\plus
 3. % add this bank account to another item:
 4. %  either another BankAccount, or something
 5. %  else that can be added to a double
 6. if isa(item,'BankAccount')
 7.     addend = getBalance(item);
 8. else
 9.     addend = item;
10. end
11. ans = addend + getBalance(ba);
```

To sum items in a generic list, we will again follow the recursive template shown in Template 19.1 with the following tailoring required:

- As usual, the public method, `sum`, merely launches the recursive helper, `sumR`, with the current head of the list and passes the total on to its caller.
- The helper returns `0` when it reaches the end of the old list.
- Otherwise, it returns the result from adding the current data value to the result of applying `sumR` recursively to the rest of the list.

The resulting code is shown in Listing 19.12.

In Listing 19.12:

Lines 1–3: Show the method header.

Line 4: Invokes the recursive sum and returns the result.

Lines 5 and 6: Show the header for the recursive helper.

Lines 7 and 8: Show the terminating condition—return zero when there are no nodes.

Lines 10 and 11: We should spend a line or two on the stunning elegance of this apparently simple statement. MATLAB is set up so that whenever the + operator is invoked, it just calls the `plus(...)` method of the first item with the second operand as the second parameter. Since `sumR(...)` always returns a number, this line of code will result in calling the `plus` method on whatever item is in the list with the current sum as the second parameter. If the list is a list of `BankAccount` objects, the `BankAccount plus(...)` method will be invoked. If it is a list of numbers, the built-in `plus` method for doubles will be called. This same `sum(...)` method could be applied to a linked list containing vectors as long as they all had the same length!

Listing 19.13 is a script to test the filtering and folding functions. Note that as far as outside appearances are concerned, filtering is done "in place" in the

Listing 19.12 Generic `LinkedList sum` method

```
 1. function total = sum( list )
 2. % @LinkedList\sum
 3. % total the items in a list
 4. total = sumR( getHead(list) );

 5. function sum = sumR(here)
 6. % recursive list adder
 7. if isempty(here)
 8.     sum = 0;
 9. else
10.     sum = getData(here)...
11.         + sumR(getNext(here));
12. end
```

`LinkedList` class. So to keep a copy of all the accounts before applying the filter, we use the copy constructor as follows:

```
goodAccts = LinkedList(BAList);
```

In Listing 19.13:

Line 1: Creates an empty list.
Lines 2–4: Add three random accounts.
Lines 5–8: Add an overdrawn `DeluxSavingsAccount`.
Line 9: Uses the `char(...)` method to print the complete list.
Line 10: Makes a copy of the original account list.
Line 11: Filters the accounts.
Line 12: Uses the `LinkedList char(...)` method to display the filtered list.
Line 13: Folds the original list with the `sum(...)` method.

Listing 19.14 shows the result of this test.

A Powerful Thought Notice that the total of the original list is in fact the total of the four original account balances. This draws attention to a powerful

Listing 19.13 Testing the `filter` function

```
 1. BAList = LinkedList;
 2. addToHead(BAList, BankAccount)
 3. addToHead(BAList, BankAccount(-1000))
 4. addToHead(BAList, SavingsAccount(2000))
 5. dsa = DeluxSavingsAccount(2000);
 6. allowOverdraft(dsa, true);
 7. gets = withdraw(dsa, 3000);
 8. addToHead(BAList, dsa)
 9. fprintf('accounts: %s\n', char(BAList) )
10. goodAccts = LinkedList(BAList);
11. filter(goodAccts);
12. fprintf('\ngood accounts: %s\n',char(goodAccts) )
13. total = sum(BAList)
```

Listing 19.14 Result of testing the `filter` function

```
accounts: Delux Savings Account with $-1020.00
 with overdraft OK
Savings Account with $2000.00
Account with $-1000.00
Account with $0.00
good accounts: Savings Account with $2000.00
Account with $0.00
total =
   -20
```

idea. We should ask whether it is sufficient for the `LinkedList` copy constructor to do as all previous copy constructors have done and return just a copy of the object provided. Follow this thought chain:

> *First Answer:* "Oops, no—you cannot just return a copy of the `LinkedList` object, because it contains a reference to a chain of `LLNodes` in memory. If you just copy the head of that chain, you will end up with two `LinkedList` objects pointing to the same `LLNode` chain. If you modify either of these `LinkedList` objects, they will both have their `LLNode` chains changed." True or False?
>
> *Second Answer:* "Oops, no—False. Whenever you change the `LLNode` chain, MATLAB makes you copy it anyway, so it is actually perfectly safe to copy just the original list head. Whoever changes that chain gets back a new chain, not a modification of the original."
>
> *Sad Reality:* The current implementation of the `LinkedList` copy constructor actually does the deep copy by building a new `LLNode` chain.[3]

19.5.6 Searching a Linked List

Unlike the previous list operations, searching can be performed as a service in the `LinkedList` class. We illustrate this by the `find(...)` method. It constrains the contents of the list to those classes that respond to the `eq(...)` method. The `eq(...)` method is invoked whenever a program uses the `'=='` operator, and can be applied to primitive data types and added to all more complex classes. The user of this operation is required only to state how a user class will determine whether an object of that class responds to the equality test for some data. Bank account equality could be determined in general by account number, telephone number, and so on. In our simple example, we will look for a number in a list of numbers.

To find an item in a list, we will again follow the recursive template shown in Template 19.1 with the following tailoring required:

- As usual, the public method, `find`, launches the recursive helper, `findR`, with the current head of the list and the item to be found, `item`. When the recursive call completes, it passes the result to its caller.
- The helper returns `[ ]` when it reaches the end of the old list.
- Otherwise, if the item sought matches this item in the list, it returns that item.
- Otherwise, it returns the result from finding the item on the rest of the list, using `findR` recursively.

The resulting code is shown in Listing 19.15.

[3] If you can explain this to your instructor, you deserve an A for the class and to be exempted from all future assignments.

Listing 19.15 `LinkedList find` method

```
 1. function result = find( list, item )
 2. %
 3. % find an item on a list
 4. result = findR( getHead(list), item );

 5. function result = findR(here, item)
 6. % recursive list adder
 7. if isempty(here)
 8.     result = [];
 9. elseif getData(here) == item
10.     result = getData(here);
11. else
12.     result = findR(getNext(here), item);
13. end
```

In Listing 19.15:

Lines 1–3: Show the method header as a wrapper function.

Line 4: Returns the result of the recursive call. Rather than returning `true` or `false`, this function should do the more useful thing of returning *a copy of* the object found or `empty` if it is not there. (You cannot do what you really want to do and return a reference to the object actually in the list.)

Lines 5 and 6: Show the header for the recursive helper.

Lines 7 and 8: Show the terminating condition—if you reach the end of the list, return `empty`.

Line 9: Shows another elegant situation. This calls the `eq(...)` method on whatever item is in the list. In our test case, this is a number with a built-in `eq(...)` method, but exactly the same code will also call the `eq(...)` method on user-defined classes.

Line 10: Shows the object found.

Line 12: If not found, continues with the recursive calls.

Listing 19.16 shows the code to test the `find` method. It populates a `LinkedList` with random numbers between 0 and 19, and then checks to see if the value 10 is on the list. Running it repeatedly will verify that the `find` method is operating correctly.

Listing 19.16 Testing the `find` method

```
lst = LinkedList;
for it = 1:20
    num = floor( 20*rand );
    addToHead( lst, num );
end
```

continued on next page

```
 6. fprintf('list is %s\n', char(lst));
 7. ans = find(lst, 10);
 8. if isempty(ans)
 9.     ans = -1;
10. end
11. fprintf('find 10 returned %d\n', ans );
```

In Listing 19.16:

Line 1: Makes a new linked list.

Lines 2–5: Add 20 random integers.

Line 6: Displays the list.

Line 7: Finds the value 10 on the list.

Lines 8–10: Detect the failure case to avoid problems with `fprintf`.

Line 11: Shows the result.

19.6 Applications of Linked Lists

Aside from being a good introductory example of dynamic data structures, there are three useful children classes of the `LinkedList` class—the `Queue`, the `Stack`, and the `PriorityQueue`. In Chapter 17 we saw that two of these classes, the `Queue` and the `PriorityQueue`, are necessary to the algorithms for searching for paths in a graph. We might ask what is wrong with the queue and priority queue implementations that used cell arrays to store the queue data. There are two answers: one is weak because of the MATLAB OO implementation, but the other is very strong.

1. The slightly weak argument is that the efficiency of every cell array operation—adding to the head or tail or removing from the head—is always O(N) because the cell array must be copied. The efficiency of removing from the head of a linked list is O(1), and adding to the tail is also O(1) in other implementations that use references, such as Java.
2. The stronger argument is that in the OO implementation, the data are completely encapsulated. The user has no access at all to the data except by way of the methods made public to the user.

The following sections describe the OO implementation of the `Queue`, `MStack`, and `PriorityQueue` classes. Unfortunately, although MATLAB does not currently implement the `Queue and PriorityQueue` classes, it does have an internal `Stack` class that forces us to change the name of our stack implementation.

19.6.1 Queues

Here we begin to see some concrete return on the investment made to create the `LinkedList` class. Once that `LinkedList` class has been created with all

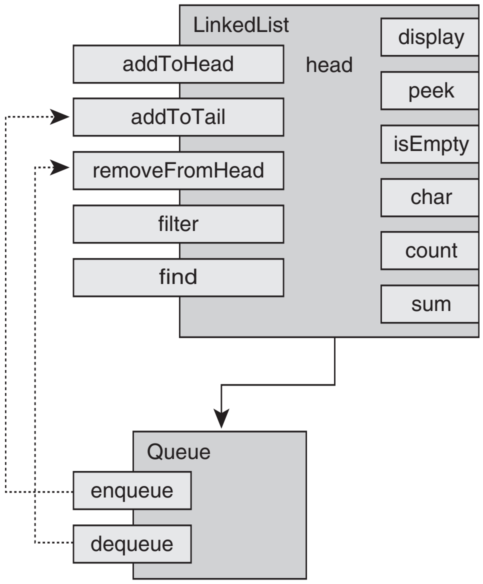

Figure 19.11 *Queue implementation*

the data support methods likely to be required, a `Queue` class can quickly be derived as a child class, as shown in Figure 19.11. All we need to do is implement the child class framework and two methods, each of which invokes the appropriate parent methods. To `enqueue`, we need to add to the tail of the list, and to `dequeue`, we remove from the head of the list.

The complete code for the `Queue` class is therefore its standard infrastructure plus the simple `enqueue(...)` and `dequeue(...)` methods, as shown in Listings 19.17 and 19.18. Since the method `peek(...)` is appropriate for all three child classes, this method (Listing 19.19) was added to the `LinkedList` class.

Listing 19.17 `Queue enqueue` method

```
function enqueue(q, data)
% @Queue\enqueue
% enqueue onto a queue
   addToTail(q, data);
   assignin('caller', inputname(1), q);
```

Listing 19.18 `Queue dequeue` method

```
function ans = dequeue(q)
% @Queue\dequeue
% dequeue from a Queue
   ans = removeFromHead(q);
   assignin('caller', inputname(1), q);
```

Listing 19.19 `LinkedList peek` method

```
function ans = peek(lst)
% @LinkedList\peek
% peek at the head of the list
   ans = getData(getHead(lst));
```

Listing 19.20 Testing the `Queue` class

```
 1. q = Queue;
 2. fprintf('is queue empty? %d\n', isEmpty(q) );
 3. for ix = 1:10
 4.     enqueue(q, ix);
 5. end
 6. fprintf('is queue empty? %d\n', isEmpty(q) );
 7. q
 8. fprintf('dequeue -> %d leaving %s\n', ...
        dequeue(q), char(q) );
 9. fprintf('peek at queue -> %d leaving %s\n', ...
        peek(q), char(q) );
```

Listing 19.20 shows the code to test the `Queue` class.

In Listing 19.20:

Line 1: Creates an empty queue.
Line 2: Verifies that it is empty.
Lines 3–5: Put the numbers 1:10 into the queue.
Line 6: Verifies that the queue is not empty.
Line 7: Forces its display method to show us the queue.
Line 8: Shows that the first item into the queue was the first item out.
Line 9: Shows that the second item would be the next out, but is not removed by the `peek(...)` method.

Listing 19.21 shows the result of this test.

19.6.2 Stacks

Once we have a working `LinkedList` class, implementing a stack is also trivial, as illustrated in Figure 19.12. The `push(...)` method merely calls

Listing 19.21 Results from testing the `Queue` class

```
is queue empty? 1
is queue empty? 0
Queue: 1;2;3;4;5;6;7;8;9;10;
dequeue -> 1 leaving Queue: 2;3;4;5;6;7;8;9;10;
peek at queue -> 2 leaving Queue: 2;3;4;5;6;7;8;9;10;
```

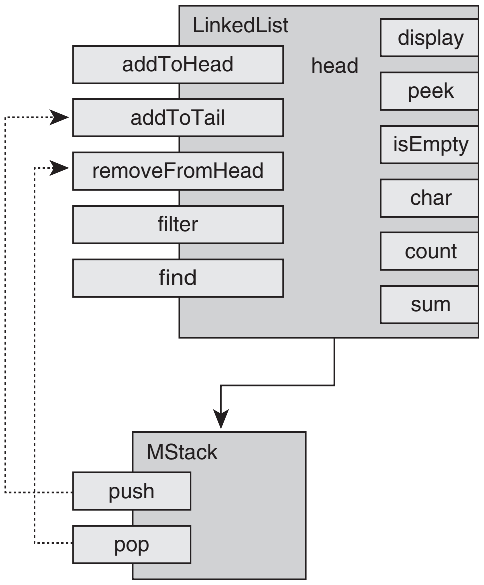

Figure 19.12 `MStack` *implementation*

`addtoHead(...)`, and the `pop(...)` method calls `removeFromHead(...)`. As previously noted, we need to name our stack class something other than `Stack` to avoid the MATLAB name conflict.

Listings 19.22 and 19.23 show the `push(...)` and `pop(...)` methods without the need for further explanation.

Listing 19.24 shows the code to test the `MStack` class, which is intended to follow on from the `Queue` class test script.

Listing 19.22 The `Stack push` method

```
function push(stk, data)
% @MStack\push
% push onto a stack
   addToHead(stk, data);
   assignin('caller', inputname(1), stk);
```

Listing 19.23 The `Stack pop` method

```
function ans = pop(stk)
% @MStack\pop
% pop from a stack
   ans = removeFromHead(stk);
   assignin('caller', inputname(1), stk);
```

Listing 19.24 Testing the `MStack` class

```
 1. stk = MStack;
 2. fprintf('is stack empty? %d\n', isEmpty(stk) );
 3. for ix = 1:10
 4.     push(stk, ix);
 5. end
 6. fprintf('is stack empty? %d\n', isEmpty(stk) );
 7. stk
 8. fprintf('pop from stack -> %d leaving %s\n', ...
        pop(stk), char(stk) );
 9. fprintf('peek stack -> %d leaving %s\n', ...
        peek(stk), char(stk) );
19. while ~isEmpty(q)
20.     push(stk, dequeue(q));
21. end
22. fprintf('stack with whole queue is %s\n', ...
                             char(stk) );
```

In Listing 19.24:

Line 1: Creates an `MStack` object.

Line 2: Checks that the empty test works.

Lines 3–5: Push the numbers 1 to 10.

Line 6: Checks that the stack is not empty.

Line 7: Forces the `display(...)` method to show the stack contents.

Line 8: The first out should be the last item in: the value 10, leaving the other 9 numbers on the stack.

Line 9: `peek(...)` should show the next out, but keep it on the stack.

Lines 19–21: Empty the queue (Listing 19.20) into the stack.

Line 22: Shows the whole stack with the queue coming out first in reverse order.

Listing 19.25 shows the results of this test.

19.6.3 Priority Queues

It is not surprising that implementing a priority queue can be accomplished by extending the `Queue` class, and invoking a different `LinkedList` method

Listing 19.25 Results from testing the `MStack` class

```
is stack empty? 1
is stack empty? 0
MStack: 10;9;8;7;6;5;4;3;2;1;
pop from stack -> 10 leaving MStack:
                        9;8;7;6;5;4;3;2;1;
peek stack -> 9 leaving MStack: 9;8;7;6;5;4;3;2;1;
stack with whole queue is MStack:
             10;9;8;7;6;5;4;3;2;9;8;7;6;5;4;3;2;1;
```

for the `PriorityQueue enqueue(...)` method. This is an excellent example of extension by redefinition—allowing the parent classes to provide all the infrastructure and operational requirements except the one specific new capability required. Figure 19.13 illustrates this implementation, and Listing 19.26 illustrates the one method required.

Listing 19.27 shows a suitable script for testing the `PriorityQueue` class.

In Listing 19.27:

Line 1: Creates a priority queue.

Lines 2–6: Generate 10 random numbers between 1 and 100, display them, and enqueue them on the priority queue.

Line 7: Displays the priority queue.

Listing 19.28 shows the results from running this test script. Notice that the values are in priority order.

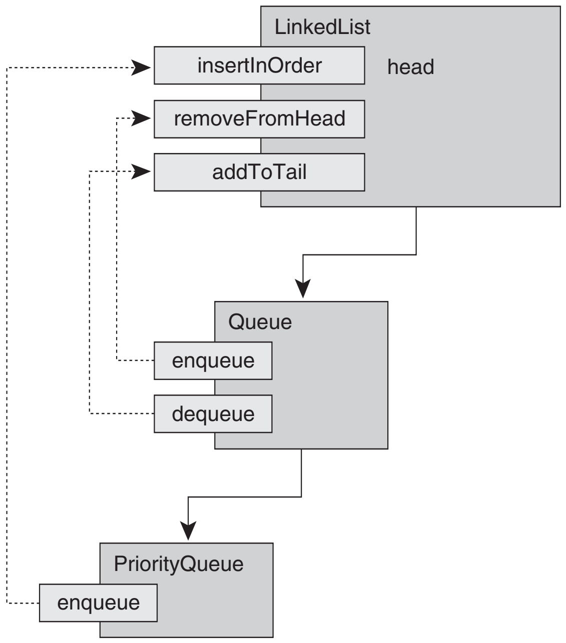

Figure 19.13 `PriorityQueue` *implementation*

Listing 19.26 `PriorityQueue enqueue` method

```
function enqueue(pq, data)
% @PriorityQueue\enqueue
%     enqueue onto a queue
   addInOrder(pq, data);
   assignin('caller', inputname(1), pq);
```

Listing 19.27 `PriorityQueue` test script

```
pq = PriorityQueue;
for ix = 1:10
    value = floor(100*rand);
    fprintf(' %g:', value );
    enqueue(pq, value );
end
fprintf('\npriority queue is %s\n', char(pq) );
```

Listing 19.28 `PriorityQueue` test results

```
82: 91: 11: 81: 90: 15: 12: 76: 72: 65:
priority queue is PriorityQueue:
                     11;12;15;65;72;76;81;82;90;91;
```

Chapter Summary

This chapter discussed:

- The overall concept of dynamic data structures
- The linked list as a specific structure
- The MATLAB implementation of a linked list
- Queues, stacks, and priority queues as useful applications created by extending the `LinkedList` class

Programming Projects

1. Write and test a method that copies all the entries from a `LinkedList` into a cell array.
   a. What kind of process is this? (Folding? Mapping?)
   b. Is this restricting the nature of the data in any way?
2. Write another method to copy the data, inserting each item *in order* into the cell array.
   a. Is this now restricting the nature of the data in any way?
3. Rewrite the `char(...)` method for the `LinkedList` class in recursive form. Call it `rchar(...)` and test it thoroughly.
4. Rewrite the recursive `rchar(...)` method for the `LinkedList` class, concatenating the recursive result before the local data instead of after it. Test this version thoroughly. What do you learn about the ordering of the data?

5. Write and test a method to find the number of items in a `LinkedList`.
6. Write and test a method to find the average of the items in a `LinkedList`.
7. Write and test a method to find the bank account with the largest balance in a `LinkedList` of `BankAccounts`.
8. Write an iterative version of `find(...)` for the `LinkedList` class using a `while` loop.
9. Write an iterative version of `sum(...)` for the `LinkedList` class using a `while` loop.
10. Write the class `ArrayQueue` that conforms to the `Queue` ADT, but is implemented using a cell array instead of inheriting from the `LinkedList` class.
11. Write the class `ArrayStack` that conforms to the `Stack` ADT, but is implemented using a cell array instead of inheriting from the `LinkedList` class.
12. Attempt to write an iterative version of `addToTail` for a `LinkedList` class using a `while` loop to reach the end of the list.

Hint

Don't spend too long trying—it probably cannot be done. What inherent MATLAB behavior makes this impossible?