CHAPTER 6

Linked Structures

An array is the most basic sequence container used to store and access a collection of data. It provides easy and direct access to the individual elements and is supported at the hardware level. But arrays are limited in their functionality. The Python list, which is also a sequence container, is an abstract sequence type implemented using an array structure. It extends the functionality of an array by providing a larger set of operations than the array, and it can automatically adjust in size as items are added or removed.

The array and Python list can be used to implement many different abstract data types. They both store data in linear order and provide easy access to their elements. The binary search can be used with both structures when the items are stored in sorted order to allow for quick searches. But there are several disadvantages in the use of the array and Python list. First, insertion and deletion operations typically require items to be shifted to make room or close a gap. This can be time consuming, especially for large sequences. Second, the size of an array is fixed and cannot change. While the Python list does provide for an expandable collection, that expansion does not come without a cost. Since the elements of a Python list are stored in an array, an expansion requires the creation of a new larger array into which the elements of the original array have to be copied. Finally, the elements of an array are stored in contiguous bytes of memory, no matter the size of the array. Each time an array is created, the program must find and allocate a block of memory large enough to store the entire array. For large arrays, it can be difficult or impossible for the program to locate a block of memory into which the array can be stored. This is especially true in the case of a Python list that grows larger during the execution of a program since each expansion requires ever larger blocks of memory.

In this chapter, we introduce the linked list data structure, which is a general purpose structure that can be used to store a collection in linear order. The linked list improves on the construction and management of an array and Python list by requiring smaller memory allocations and no element shifts for insertions and

deletions. But it does eliminate the constant time direct element access available with the array and Python list. Thus, it's not suitable for every data storage problem. There are several varieties of linked lists. The singly linked list is a linear structure in which traversals start at the front and progress, one element at a time, to the end. Other variations include the circularly linked, the doubly linked, and the circularly doubly linked lists.

6.1 Introduction

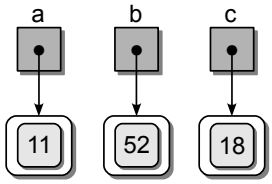
Suppose we have a basic class containing a single data field:

```
class ListNode :
  def __init__( self, data ) :
    self.data = data
```

We can create several instances of this class, each storing data of our choosing. In the following example, we create three instances, each storing an integer value:

```
a = ListNode( 11 )
b = ListNode( 52 )
c = ListNode( 18 )
```

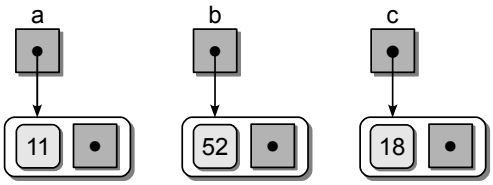
the result of which is the creation of three variables and three objects :



Now, suppose we add a second data field to the `ListNode` class:

```
class ListNode :
  def __init__( self, data ) :
    self.data = data
    self.next = None
```

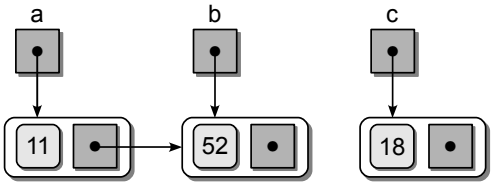
The three objects from the previous example would now have a second data field initialized with a null reference, as illustrated in the following:



Since the `next` field can contain a reference to any type of object, we can assign to it a reference to one of the other `ListNode` objects. For example, suppose we assign `b` to the `next` field of object `a`:

```
a.next = b
```

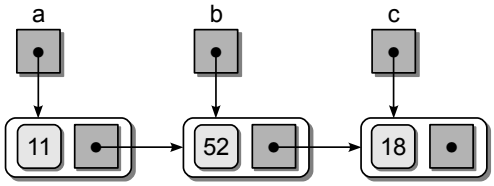
which results in object `a` being linked to object `b`, as shown here:



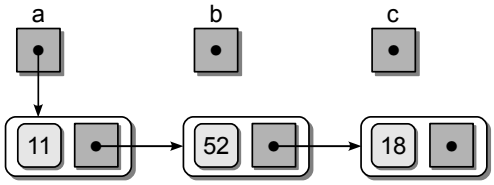
And finally, we can link object `b` to object `c`:

```
b.next = c
```

resulting in a chain of objects, as illustrated here:



We can remove the two external references `b` and `c` by assigning `None` to each, as shown here:



The result is a linked list structure. The two objects previously pointed to by `b` and `c` are still accessible via `a`. For example, suppose we wanted to print the values of the three objects. We can access the other two objects through the `next` field of the first object:

```
print( a.data )
print( a.next.data )
print( a.next.next.data )
```

A ***linked structure*** contains a collection of objects called ***nodes***, each of which contains data and at least one reference or ***link*** to another node. A ***linked list*** is a linked structure in which the nodes are connected in sequence to form a linear

list. Figure 6.1 provides an example of a linked list consisting of five nodes. The last node in the list, commonly called the ***tail node***, is indicated by a null link reference. Most nodes in the list have no name and are simply referenced via the link field of the preceding node. The first node in the list, however, must be named or referenced by an external variable as it provides an entry point into the linked list. This variable is commonly known as the head pointer, or ***head reference***. A linked list can also be empty, which is indicated when the head reference is null.

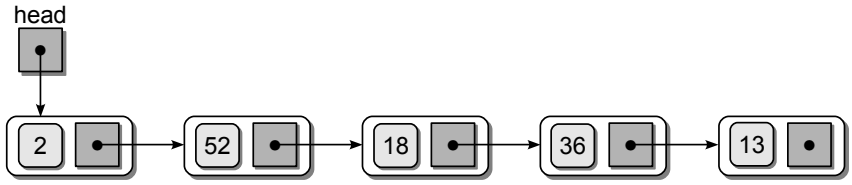


Figure 6.1: A singly linked list consisting of five nodes and a head reference.

Linked structures are built using fundamental components provided by the programming language, namely reference variables and objects. The linked list is just one of several linked structures that we can create. If more links are added to each node, as illustrated in Figure 6.2, we can connect the nodes to form any type of configuration we may need. The tree structure, which organizes the nodes in a hierarchical fashion, is another commonly used linked structure that we explore later in Chapters 13 and 14.

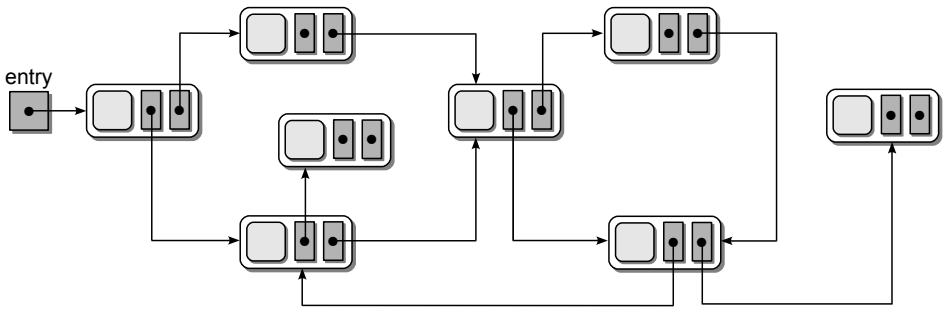


Figure 6.2: An example of a complex linked structure.

A linked list is a data structure that can be used to implement any number of abstract data types. While some languages do provide, as part of their standard library, a generic List ADT implemented using a linked list, we are going to create and work with linked lists directly. Some algorithms and abstract data types can be implemented more efficiently if we have direct access to the individual nodes within the ADT than would be possible if we created a generic linked list class.

In the next section, we explore the construction and management of a singly linked list independent of its use in the implementation of any specific ADT. In later sections we then present examples to show how linked lists can be used to implement abstract data types. We also include a number of exercises at the end of the chapter that provide practice in the construction and management of singly linked lists.

NOTE

External References. We use the term *external reference* to indicate those reference variables that point to a node but are not themselves contained within a node as is the case with the link fields. Some external references must be permanent or exist during the lifetime of the linked list in order to maintain the collection of nodes. Others are only needed on a temporary basis in order to perform a specific operation. These temporary external references should be local variables that disappear after the function or method has completed.

6.2 The Singly Linked List

A ***singly linked list*** is a linked list in which each node contains a single link field and allows for a complete traversal from a distinctive first node to the last. The linked list in Figure 6.1 is an example of a singly linked list.

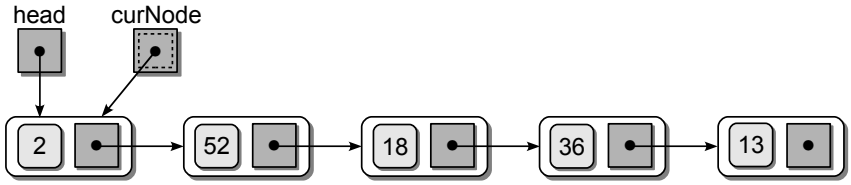
There are several operations that are commonly performed on a singly linked list, which we explore in this section. To illustrate the implementation of these operations, our code assumes the existence of a head reference and uses the `ListNode` class defined earlier. The data fields of the `ListNode` class will be accessed directly but this class should not be used outside the module in which it is defined as it is only intended for use by the linked list implementation.

6.2.1 Traversing the Nodes

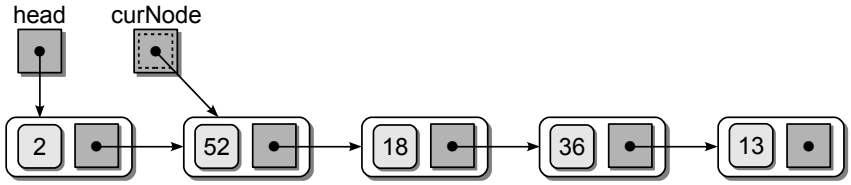
In the earlier example, we accessed the second and third nodes of our sample list by stringing together the `next` field name off the external reference variable `a`. This may be sufficient for lists with few nodes, but it's impractical for large lists. Instead, we can use a temporary external reference to traverse through the list, moving the reference along as we access the individual nodes. The implementation is provided Listing 6.1.

The process starts by assigning a temporary external reference `curNode` to point to the first node of the list, as illustrated in Figure 6.3(a). After entering the loop, the value stored in the first node is printed by accessing the data component stored in the node using the external reference. The external reference is then advanced to the next node by assigning it the value of the current node's link field, as illustrated in Figure 6.3(b). The loop iteration continues until every node in

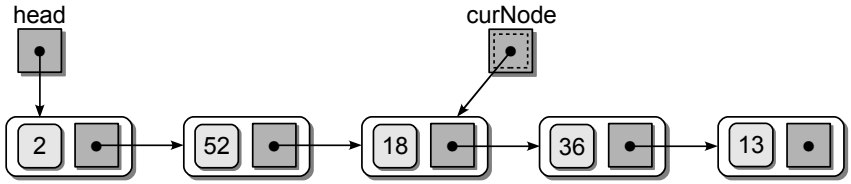
(a) After initializing the temporary external reference.



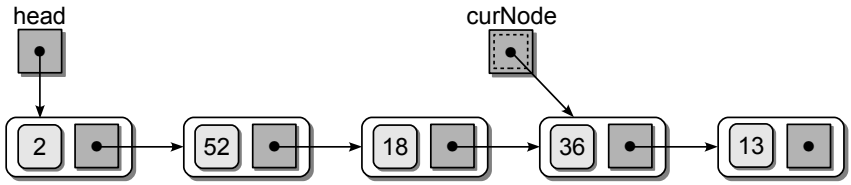
(b) Advancing the external reference after printing value 2.



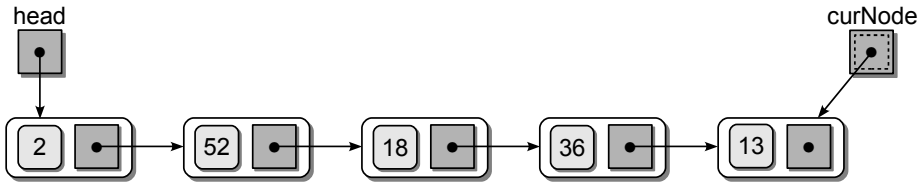
(c) Advancing the external reference after printing value 52.



(d) Advancing the external reference after printing value 18.



(e) Advancing the external reference after printing value 36.



(f) The external reference is set to None after printing value 13.

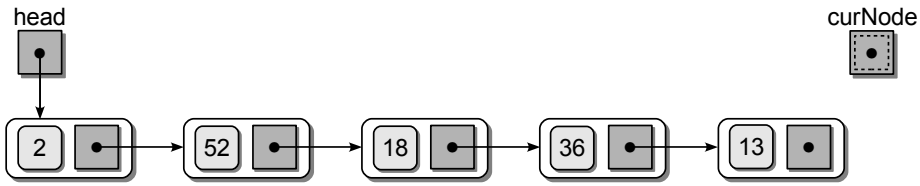


Figure 6.3: Traversing a linked list requires the initialization and adjustment of a temporary external reference variable.

Listing 6.1 Traversing a linked list.

```
1 def traversal( head ):
2   curNode = head
3   while curNode is not None :
4     print curNode.data
5     curNode = curNode.next
```

the list has been accessed. The completion of the traversal is determined when `curNode` becomes null, as illustrated in Figure 6.3(f). After accessing the last node in the list, `curNode` is advanced to the next node, but there being no next node, `curNode` is assigned `None` from the `next` field of the last node.

A correct implementation of the linked list traversal must also handle the case where the list is empty. Remember, an empty list is indicated by a null head reference. If the list were empty, the `curNode` reference would be set to null in line 2 of Listing 6.1 and the loop would not execute producing the correct result. A complete list traversal requires $O(n)$ time since each node must be accessed and each access only requires constant time.

6.2.2 Searching for a Node

A linear search operation can be performed on a linked list. It is very similar to the traversal demonstrated earlier. The only difference is that the loop can terminate early if we find the target value within the list. Our implementation of the linear search is illustrated in Listing 6.2.

Listing 6.2 Searching a linked list.

```
1 def unorderedSearch( head, target ):
2   curNode = head
3   while curNode is not None and curNode.data != target :
4     curNode= curNode.next
5   return curNode is not None
```

Note the order of the two conditions in the `while` loop. It is important that we test for a null `curNode` reference before trying to examine the contents of the node. If the item is not found in the list, `curNode` will be null when the end of the list is reached. If we try to evaluate the `data` field of the null reference, an exception will be raised, resulting in a run-time error. Remember, a null reference does not point to an object and thus there are no fields or methods to be referenced.

When implementing the search operation for the linked list, we must make sure it works with both empty and non-empty lists. In this case, we do not need a separate test to determine if the list is empty. This is done automatically by checking the traversal reference variable as the loop condition. If the list were empty, `curNode` would be set to `None` initially and the loop would never be entered. The linked list search operation requires $O(n)$ in the worst case, which occurs when the target item is not in the list.

6.2.3 Prepending Nodes

When working with an unordered list, new values can be inserted at any point within the list. Since we only maintain the head reference as part of the list structure, we can simply prepend new items with little effort. The implementation is provided in Listing 6.3. Prepending a node can be done in constant time since no traversal is required.

Listing 6.3 Prepending a node to the linked list.

```
# Given the head pointer, prepend an item to an unsorted linked list.
newNode = ListNode( item )
newNode.next = head
head = newNode
```

Suppose we want to add the value 96 to our example list shown in Figure 6.4(a). Adding an item to the front of the list requires several steps. First, we must create a new node to store the new value and then set its `next` field to point to the node currently at the front of the list. We then adjust `head` to point to the new node since it is now the first node in the list. These steps are represented as dashed lines in Figure 6.4(b). Note the order of the new links since it is important we first link the new node into the list before modifying the head reference. Otherwise, we lose

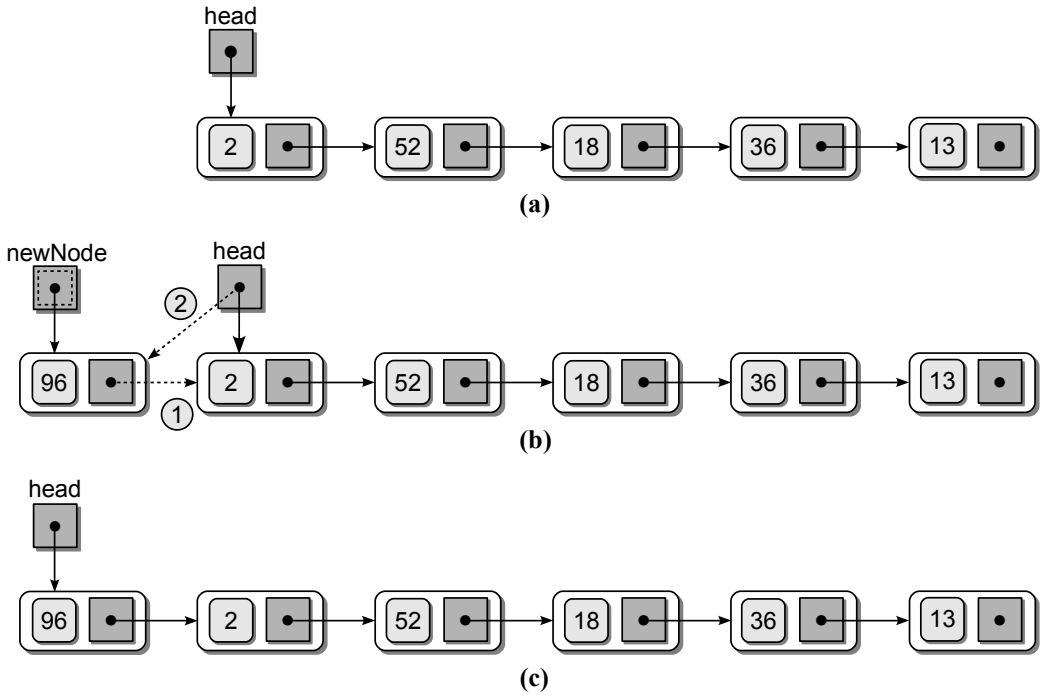


Figure 6.4: Prepending a node to the linked list: (a) the original list from Figure 6.1; (b) link modifications required to prepend the node; and (c) the result after prepending 96.

our external reference to the list and in turn, we lose the list itself. The results, after linking the new node into the list, are shown in Figure 6.4(c).

When modifying or changing links in a linked list, we must consider the case when the list is empty. For our implementation, the code works perfectly since the `head` reference will be null when the list is empty and the first node inserted needs the `next` field set to `None`.

6.2.4 Removing Nodes

An item can be removed from a linked list by removing or unlinking the node containing that item. Consider the linked list from Figure 6.4(c) and assume we want to remove the node containing 18. First, we must find the node containing the target value and position an external reference variable pointing to it, as illustrated in Figure 6.5(a). After finding the node, it has to be unlinked from the list, which entails adjusting the link field of the node's predecessor to point to its successor as shown in Figure 6.5(b). The node's link field is also cleared by setting it to `None`.

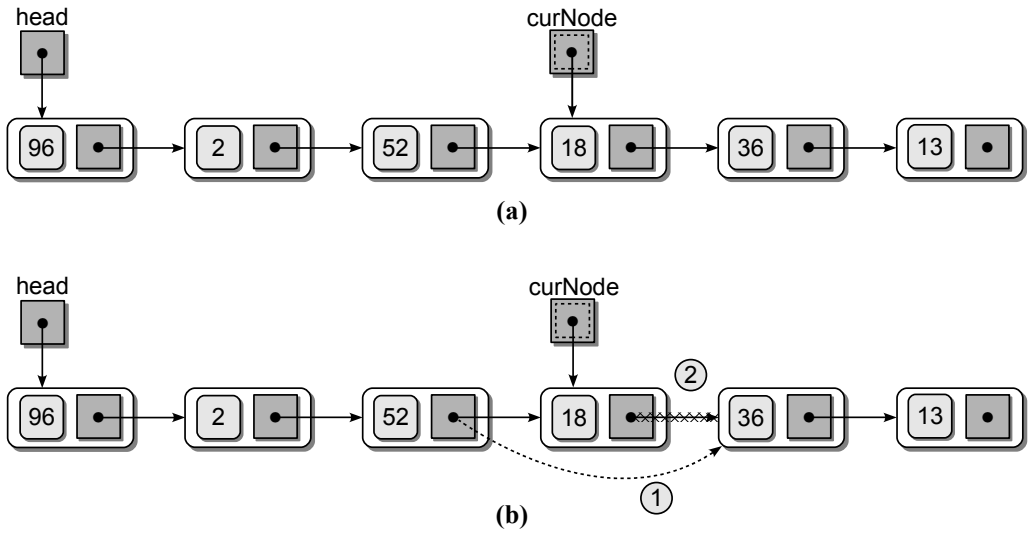


Figure 6.5: Deleting a node from a linked list: (a) finding the node to be removed and assigning an external reference variable and (b) the link modifications required to unlink and remove a node.

Accessing the node's successor is very simple using the `next` link of the node. But we must also access the node's predecessor in order to change its link. The only way we can do this is to position another external reference simultaneously during the search for the given node, as illustrated in Figure 6.6(a). The result after removing the node containing value 18 is shown in Figure 6.6(b).

Removing the first node from the list is a special case since the head pointer references this node. There is no predecessor that has to be relinked, but the head reference must be adjusted to point to the next node, as illustrated in Figure 6.7.

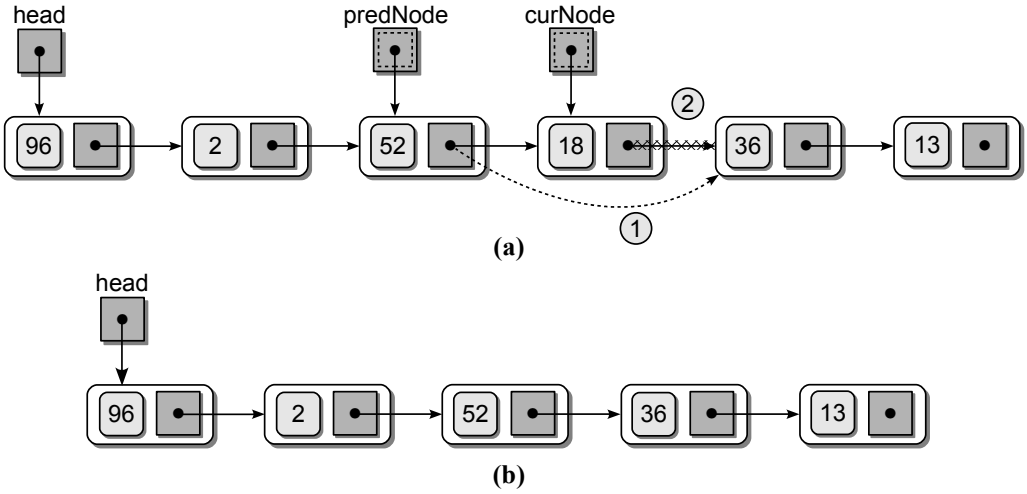


Figure 6.6: Using a second temporary reference to remove a node from a linked list: (a) positioning the second temporary reference variable `predNode`, and (b) the resulting list after removing 18 from the linked list.

We now step through the code required for deleting a node from a singly linked list, as illustrated in Listing 6.4. The `curNode` external reference is initially set to the first node in the list, the same as is done in the traversal and search operations. The `predNode` external reference is set to `None` since there is no predecessor to the first node in the list.

A loop is used to position the two temporary external reference variables as shown in lines 4–6 of Listing 6.4. As the `curNode` reference is moved along the list in the body of the loop, the `predNode` reference follows behind. Thus, `predNode` must be assigned to reference the same node as `curNode` before advancing `curNode` to reference the next node.

After positioning the two external references, there are three possible conditions: (1) the item is not in the list; (2) the item is in the first node; or (3) the item is somewhere else in the list. If the target is not in the list, `curNode` will be null, having been assigned `None` via the link field of the last node. This condition is evaluated in line 8. To determine if the target is in the first node, we can simply compare `curNode` to `head` and determine if they reference the same node. If they do, we set `head` to point to the next node in the list, as shown in lines 9–10.

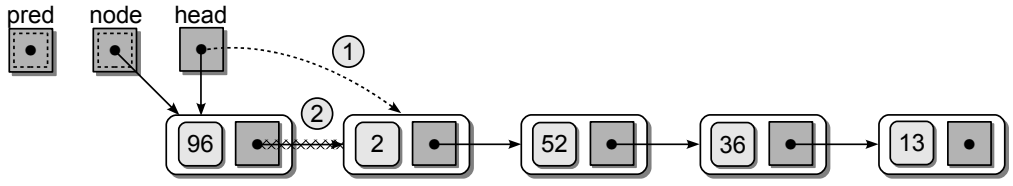


Figure 6.7: Modifications required to remove the first node of a linked list.

Listing 6.4 Removing a node from a linked list.

```
1  # Given the head reference, remove a target from a linked list.
2  predNode = None
3  curNode = head
4  while curNode is not None and curNode.data != target :
5    predNode = curNode
6    curNode = curNode.next
7
8  if curNode is not None :
9    if curNode is head :
10     head = curNode.next
11   else :
12     predNode.next = curNode.next
```

If the target is elsewhere in the list, we simply adjust the link field of the node referenced by `predNode` to point to the node following the one referenced by `curNode`. This step is performed in the `else` clause of the condition as shown in line 12 of Listing 6.4. If the last node is being removed, the same code can be used because the `next` field of the node pointed to by `predNode` will be set to `None` since `curNode` will be null. Removing a node from a linked list requires $O(n)$ time since the node could be at the end of the list, in which case a complete traversal is required to locate the node.

6.3 The Bag ADT Revisited

To illustrate the use of the linked list structure, we implement a new version of the Bag ADT, which we originally defined in Section 1.3. The new implementation is shown in Listing 6.5 on the next page.

6.3.1 A Linked List Implementation

We begin our discussion of the linked list implementation of the Bag ADT with the constructor. First, the `_head` field will store the head pointer of the linked list. The reference is initialized to `None` to represent an empty bag. The `_size` field is used to keep track of the number of items stored in the bag that is needed by the `__len__` method. Technically, this field is not needed. But it does prevent us from having to traverse the list to count the number of nodes each time the length is needed. Notice we only define a head pointer as a data field in the object. Temporary references such as the `curNode` reference used to traverse the list are not defined as attributes, but instead as local variables within the individual methods as needed. A sample instance of the new `Bag` class is illustrated in Figure 6.8.

The `contains()` method is a simple search of the linked list as described earlier in the chapter. The `add()` method simply implements the prepend operation, though we must also increment the item counter (`_size`) as new items are added. The `_BagListNode` class, used to represent the individual nodes, is also defined

Listing 6.5 The `llistbag.py` module.

```
# Implements the Bag ADT using a singly linked list.

class Bag :
   # Constructs an empty bag.
  def __init__( self ):
    self._head = None
    self._size = 0

   # Returns the number of items in the bag.
  def __len__( self ):
    return self._size

   # Determines if an item is contained in the bag.
  def __contains__( self, target ):
    curNode = self._head
    while curNode is not None and curNode.item != target :
      curNode = curNode.next
    return curNode is not None

   # Adds a new item to the bag.
  def add( self, item ):
    newNode = _BagListNode( item )
    newNode.next = self._head
    self._head = newNode
    self._size += 1

   # Removes an instance of the item from the bag.
  def remove( self, item ):
    predNode = None
    curNode = self._head
    while curNode is not None and curNode.item != item :
      predNode = curNode
      curNode = curNode.next

     # The item has to be in the bag to remove it.
    assert curNode is not None, "The item must be in the bag."

     # Unlink the node and return the item.
    self._size -= 1
    if curNode is head :
      self._head = curNode.next
    else :
      predNode.next = curNode.next
    return curNode.item

   # Returns an iterator for traversing the list of items.
  def __iter__( self ):
    return _BagIterator( self._head )

# Defines a private storage class for creating list nodes.
class _BagListNode( object ):
  def __init__( self, item ) :
    self.item = item
    self.next = None
```

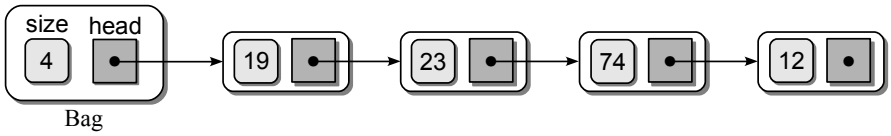


Figure 6.8: Sample instance of the `Bag` class.

within the same module. It is specified in lines 51–54 at the bottom of the module, but it is not intended for use outside the `Bag` class.

The `remove()` method implements the removal operation as presented in the previous section, but with a couple of modifications. The `if` statement that checked the status of the `curNode` variable has been replaced with an `assert` statement. This was necessary since the remove operation of the bag has a precondition that the item must be in the bag in order to be removed. If we make it pass the assertion, the item counter is decremented to reflect one less item in the bag, the node containing the item is unlinked from the linked list, and the item is returned as required by the ADT definition.

6.3.2 Comparing Implementations

The Python list and the linked list can both be used to manage the elements stored in a bag. Both implementations provide the same time-complexities for the various operations with the exception of the `add()` method. When adding an item to a bag implemented using a Python list, the item is appended to the list, which requires $O(n)$ time in the worst case since the underlying array may have to be expanded. In the linked list version of the Bag ADT, a new bag item is stored in a new node that is prepended to the linked structure, which only requires $O(1)$. Table 6.1 shows the time-complexities for two implementations of the Bag ADT.

In general, the choice between a linked list or a Python list depends on the application as both have advantages and disadvantages. The linked list is typically a better choice for those applications involving large amounts of ***dynamic data***, data that changes quite often. If there will be a large number of insertions and/or deletions, the linked structure provides a fast implementation since large amounts

Operation	Python List	Linked List
`b = Bag()`	$O(1)$	$O(1)$
`n = len(b)`	$O(1)$	$O(1)$
`x in b`	$O(n)$	$O(n)$
`b.add(x)`	$O(n)$	$O(1)$
`b.remove(x)`	$O(n)$	$O(n)$
traversal	$O(n)$	$O(n)$

Table 6.1: Comparing the Bag ADT implemented using a Python list and a linked list.

of data do not have to be shifted as is required by the Python list. This is especially true when prepending items. On the other hand, the Python list is a better choice in those applications where individual elements must be accessed by index. This can be simulated with a linked list, but it requires a traversal of the list, resulting in a linear operation whereas the Python list only requires constant time.

6.3.3 Linked List Iterators

Suppose we want to provide an iterator for our Bag ADT implemented using a linked list as we did for the one implemented using a Python list. The process would be similar, but our iterator class would have to keep track of the current node in the linked list instead of the current element in the Python list. We define a bag iterator class in Listing 6.6, which would be placed within the `llistbag.py` module that can be used to iterate over the linked list.

Listing 6.6 An iterator for the `Bag` class implemented using a linked list.

```
1  # Defines a linked list iterator for the Bag ADT.
2  class _BagIterator :
3    def __init__( self, listHead ):
4      self._curNode = listHead
5
6    def __iter__( self ):
7      return self
8
9    def next( self ):
10     if self._curNode is None :
11       raise StopIteration
12     else :
13       item = self._curNode.item
14       self._curNode = self._curNode.next
15       return item
```

When iterating over a linked list, we need only keep track of the current node being processed and thus we use a single data field `_curNode` in the iterator. This reference will be advanced through the linked list as the `for` loop iterates over the nodes. As was done with our Python list-based `Bag` class, the linked list version must include the `__iter__` method (shown in lines 47–48 of Listing 6.5), which returns an instance of the `_BagIterator` class.

Figure 6.9 illustrates the `Bag` and `_BagIterator` objects at the beginning of the `for` loop. The `_curNode` pointer in the `_BagIterator` object is used just like the `curNode` pointer we used when directly performing a linked list traversal earlier in the chapter. The only difference is that we don't include a `while` loop since Python manages the iteration for us as part of the `for` loop. Note, the iterator objects can be used with any singly linked list configuration to traverse the nodes and return the data contained in each.

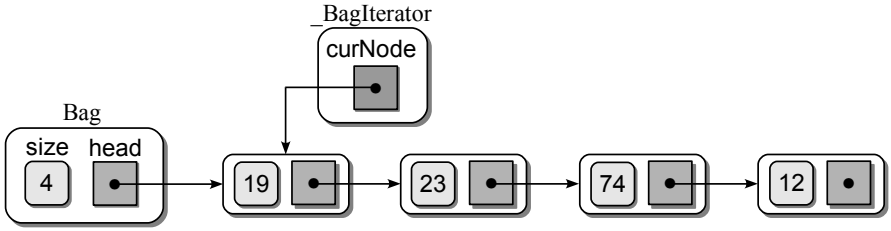


Figure 6.9: Sample `Bag` and `_BagIterator` objects at the beginning of the `for` loop.

6.4 More Ways to Build a Linked List

Earlier in the chapter, we saw that new nodes can be easily added to a linked list by prepending them to the linked structure. This is sufficient when the linked list is used to implement a basic container in which a linear order is not needed, such as with the Bag ADT. But a linked list can also be used to implement a container abstract data type that requires a specific linear ordering of its elements, such as with a Vector ADT. In addition, some implementations, such as in the case of the Set ADT, can be improved if we have access to the end of the list or if the nodes are sorted by element value.

6.4.1 Using a Tail Reference

The use of a single external reference to point to the head of a linked list is sufficient for many applications. In some instances, however, we may need to append items to the end of the list instead. Appending a new node to the list using only a head reference requires linear time since a complete traversal is required to reach the end of the list. Instead of a single external head reference, we can use two external references, one for the head and one for the tail. Figure 6.10 illustrates a sample linked list with both a head and a ***tail reference***.

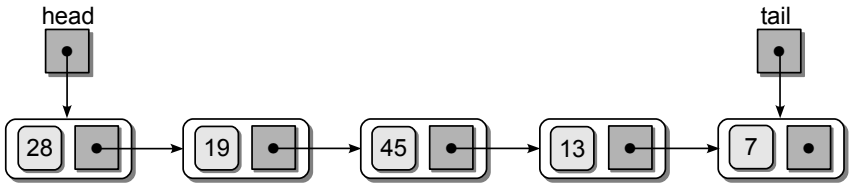


Figure 6.10: Sample linked list using both head and tail external references.

Appending Nodes

Adding the external tail reference to the linked list requires that we manage both references as nodes are added and removed. Consider the process of appending a new node to a non-empty list, as illustrated in Figure 6.11(a). First, a new node is created to store the value to be appended to the list. Then, the node is linked

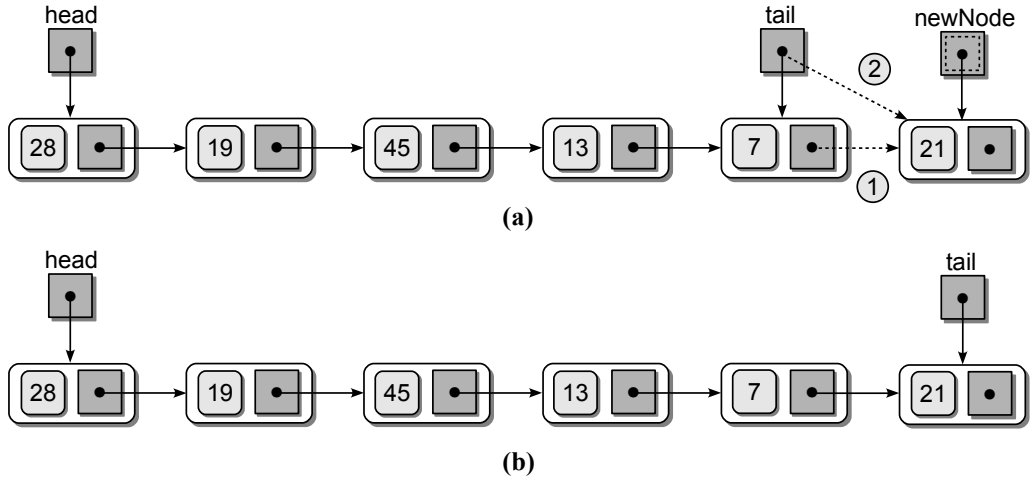


Figure 6.11: Appending a node to a linked list using a tail reference: (a) the links required to append the node, and (b) the resulting list after appending 21.

into the list following the last node. The `next` field of the node referenced by `tail` is set to point to the new node. The `tail` reference has to be adjusted to point to the new node since `tail` must always point to the last node in the list. The linked list resulting from appending 21 to the list is illustrated in Figure 6.11(b).

If the list is empty, there is no existing node in which the link field can be adjusted. Instead, both the `head` and `tail` references will be null. In this case, the new node is appended to the list by simply adjusting both external references to point to the new node. The code for appending a node to the linked list is provided in Listing 6.7. It assumes the existence of both the head and tail reference variables.

Listing 6.7 Appending a node to a linked list using a tail reference.

```
# Given the head and tail pointers, adds an item to a linked list.
newNode = ListNode( item )
if head is None :
  head = newNode
else :
  tail.next = newNode
tail = newNode
```

Removing Nodes

Removing a node from a linked list in which both head and tail references are used requires a simple modification to the code presented earlier in the chapter. Consider the sample list in Figure 6.12, in which we want to delete the node containing 21. After unlinking the node to be removed, we must check to see if it was at the end

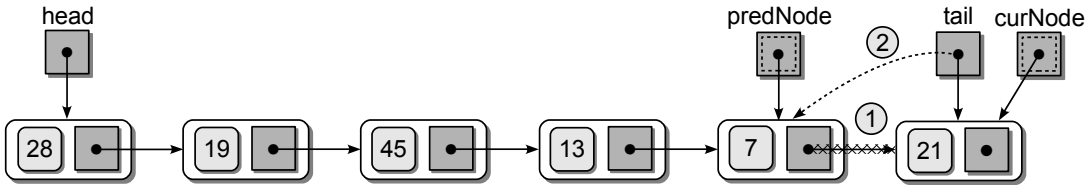


Figure 6.12: Deleting the last node in a list using a tail reference.

of the list. If it was, we must adjust the tail reference to point to the same node as `predNode`, which is now the last node in the list.

The code for removing an item from a linked list using a tail reference is shown in Listing 6.8. If the list contains a single node, the `head` reference will be assigned `None` when it is assigned the contents of the node's `next` field. The `tail` reference will also be set to `None` when it is set to `predNode`.

Listing 6.8 Removing a node from a linked list using a tail reference.

```
# Given the head and tail references, removes a target from a linked list.
predNode = None
curNode = head
while curNode is not None and curNode.data != target :
  predNode = curNode
  curNode = curNode.next

if curNode is not None :
  if curNode is head :
    head = curNode.next
  else :
    predNode.next = curNode.next
  if curNode is tail :
    tail = predNode
```

6.4.2 The Sorted Linked List

The items in a linked list can be sorted in ascending or descending order as was done with a sequence. Consider the sorted linked list illustrated in Figure 6.13. The sorted list has to be created and maintained as items are added and removed.

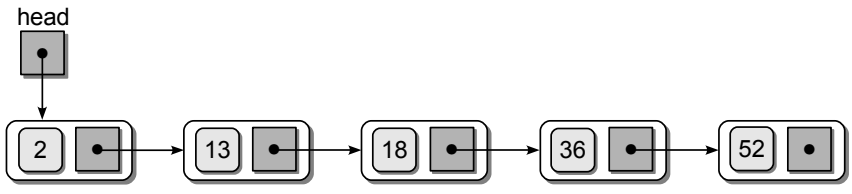


Figure 6.13: A sorted linked list with items in ascending order.

Linear Search

The linear search for use with the linked list can be modified to take advantage of the sorted items. The only change required is to add a second condition that terminates the loop early if we encounter a value larger than the target. The search routine for a sorted linked list is shown in Listing 6.9.

Listing 6.9 Searching a sorted linked list.

```
def sortedSearch( head, target ) :
  curNode = head
  while curNode is not None and curNode.data < target :
    if curNode.data == target :
      return True
    else :
      curNode = node.next
  return False
```

Inserting Nodes

Adding a new node to an unsorted linked list is simple because we can simply add it to the front or end of the list since its placement is not important. When adding a node to a sorted list, however, the correct position for the new value must be found and the new node linked into the list at that position. The Python implementation for inserting a value into a sorted linked list is provided in Listing 6.10.

As with the removal operation for the unsorted list, we must position two temporary external references by traversing through the linked list searching for the correct position of the new value. The only difference is the loop termination condition. To insert a new node, we must terminate the loop upon finding the first value larger than the new value being added.

Listing 6.10 Inserting a value into a sorted list.

```
# Given the head pointer, insert a value into a sorted linked list.
 # Find the insertion point for the new value.
predNode = None
curNode = head
while curNode is not None and value > curNode.data :
  predNode = curNode
  curNode = curNode.next

 # Create the new node for the new value.
newNode = ListNode( value )
newNode.next = curNode
 # Link the new node into the list.
if curNode is head :
  head = newNode
else :
  predNode.next = newNode
```

Three cases can occur when inserting a node into a sorted linked list, as illustrated in Figure 6.14: the node is inserted in the front, at the end, or somewhere in the middle. After finding the correct position, a new node is created and its `next` field is changed to point to the same node referenced by `curNode`. This link is required no matter where in the list the new node is inserted. If the new node is to be inserted in the front, then the operation is a simple prepend, as was done with an unsorted linked list, and `curNode` will be pointing to the first node. When the new value being added is the largest in the list and the new node is to be added at the end, `curNode` will be null and thus the `next` field will be null as it should be. When the new node is inserted elsewhere in the list, `curNode` will be pointing to the node that will follow the new node.

After linking the new node to the list, we must determine if it is being inserted at the front of the list, in which case the `head` reference must be adjusted. We do this by comparing the `curNode` reference with the `head` reference. If they are aliases, the new node comes first in the linked list and we must adjust the `head`

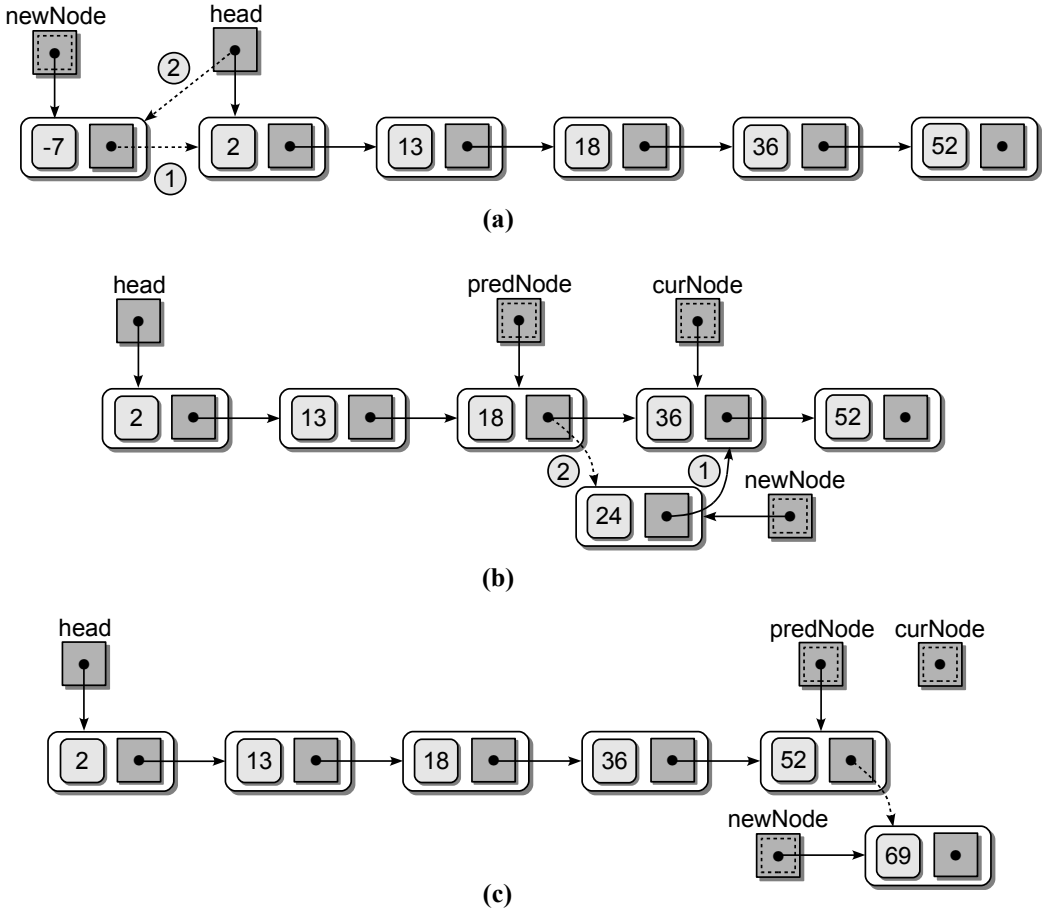


Figure 6.14: Inserting a new value into a sorted linked list: (a) inserting -7 at the front of the list; (b) inserting 24 in the middle of the list; (c) inserting 69 at the end of the list.

reference to point to the new node. If the two nodes are not aliases, then the node is inserted by setting the `next` field of the node referenced by `predNode` to point to the new node. This step is handled by lines 13–16 of Listing 6.10.

Traversing and Deleting

The traversal operation implemented for the unsorted linked list can be used with both sorted and unsorted linked lists since it is not dependent on the contents of the list itself. Deleting from a sorted linked list is the same operation used with an unsorted list with one exception. Searching for the node containing the target value can end early after encountering the first value larger than the one to be deleted.

6.5 The Sparse Matrix Revisited

In the previous chapter, we defined and implemented the Sparse Matrix ADT. Remember, a sparse matrix is a matrix containing a large number of zero elements. Instead of providing space for every element in the matrix, we need only store the non-zero elements. In our original implementation, we used a list to store the non-zero elements of the matrix, which were stored as `_MatrixElement` objects. This improved the time-complexity of many of the matrix operations when compared to the use of a 2-D array.

We can further improve the Sparse Matrix ADT by using the linked list structure. Instead of storing the elements in a single list, however, we can use an array of sorted linked lists, one for each row of the matrix. The non-zero elements for a given row will be stored in the corresponding linked list sorted by column index. The row index is not needed since it corresponds to a specific linked list within the array of linked lists. The sparse matrix from Figure 4.5 is illustrated in Figure 6.15 stored using an array of linked lists.

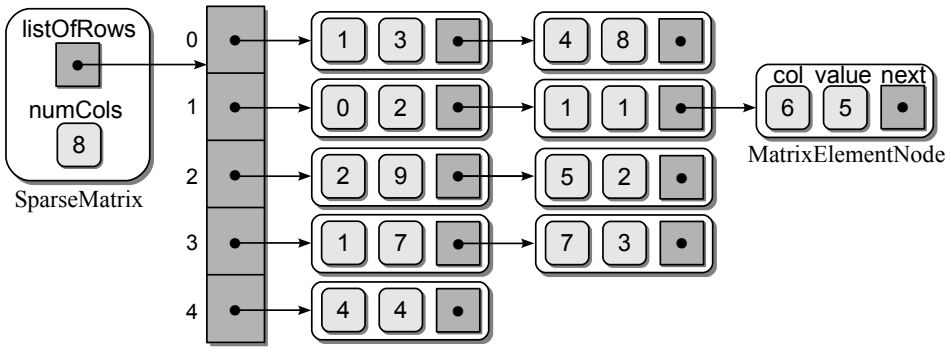


Figure 6.15: A sparse matrix implemented as an array of sorted linked lists.

6.5.1 An Array of Linked Lists Implementation

To implement the Sparse Matrix ADT using an array of sorted linked lists, we create a new `SparseMatrix` class, as shown in Listing 6.11. In the constructor, two class fields are created: one to store the number of columns in the matrix and another to store the array of head references to the linked lists in which the matrix elements will be stored. An array is created whose size is equal to the number of rows in the matrix. The individual elements are initialized to `None` to represent empty linked lists since there are no non-zero elements in the sparse matrix initially. Note we did not provide a field to store the number of rows as that information can be obtained from the length of the array. Thus, `numRows()` simply calls the array's length operation.

The `_MatrixElementNode` class, provided in lines 95–99, is a modified version of the `_MatrixElement` class used in Chapter 4. The row component has been removed while the `next` link field was added in order to use the objects as linked nodes. When elements are added to the sparse matrix, nodes will be added to the individual linked lists based on the row index of the element. Thus, the row component does not have to be stored in each node.

Changing Element Values

The `__setitem__` method is the main linked list management routine for the underlying structure. This method not only provides for the modification of element values, but it also handles node insertions for new non-zero elements and node removals when an element becomes zero. The three operations handled by this method can be combined to produce an efficient implementation.

The first step is to position the two external reference variables `predNode` and `curNode` along the linked list corresponding to the indicated row index. While only the `curNode` reference will be needed for a simple element value modification, `predNode` will be needed if we have to insert a new node or remove an existing node. After positioning the two references, we can then determine what action must be taken.

If the element corresponding to the given `row` and `col` indices is in the linked list, `curNode` will be pointing to a node and the `col` field of that node will be that of the element. In this case, either the value stored in the node is changed to the new non-zero value or the node has to be deleted when the new value is zero. Modifying the value only requires changing the `value` field of the `curNode`. Removing the element entry for a zero value is also straightforward since the two external references have been positioned correctly and the links can be changed as outlined in Section 6.2.4.

If the element is not represented by a node in the linked list of the corresponding row and the new value is non-zero, then a new node must be inserted in the proper position based on the `predNode` and `curNode` references. The only difference in the insertion operation from that used earlier in the chapter is that the head reference is stored in one of the elements of the `_listOfRows` array instead of its

own variable. If the element is already a zero-entry and the new value is zero, no action is required.

Setting the value of a matrix element requires $O(n)$ time in the worst case, where n is the number of columns in the matrix. This value is obtained by observing that the most time-consuming part is the positioning of the two references, `curNode` and `predNode`, which require a complete list traversal in the worst case. Since a linked list contains a single row, we know it will contain at most n nodes.

Listing 6.11 The `llistsparse.py` module

```
# Implementation of the Sparse Matrix ADT using an array of linked lists.
from array import Array

class SparseMatrix :
   # Creates a sparse matrix of size numRows x numCols initialized to 0.
  def __init__( self, numRows, numCols ):
    self._numCols = numCols
    self._listOfRows = Array( numRows )

   # Returns the number of rows in the matrix.
  def numRows( self ):
    return len( self._listOfRows )

   # Returns the number of columns in the matrix.
  def numCols( self ):
    return self._numCols

   # Returns the value of element (i,j): x[i,j]
  def __getitem__( self, ndxTuple ):
    ......

   # Sets the value of element (i,j) to the value s: x[i,j] = s
  def __setitem__( self, ndxTuple, value ):
    predNode = None
    curNode = self._listOfRows[row]
    while curNode is not None and curNode.col != col :
      predNode = curNode
      curNode = curNode.next

     # See if the element is in the list.
    if curNode is not None and curNode.col == col :
      if value == 0.0 :   # remove the node.
        if curNode == self._listOfRows[row] :
          self._listOfRows[row] = curNode.next
        else :
          predNode.next = curNode.next
      else :              # change the node's value.
        curNode.value = value

     # Otherwise, the element is not in the list.
    elif value != 0.0 :
      newNode = _MatrixElementNode( col, value )
      newNode.next == curNode
      if curNode == self._listOfRows[row] :
```

```
45            self._listOfRows[row] = newNode
46        else :
47          predNode.next = newnode
48
49     # Scales the matrix by the given scalar.
50    def scaleBy( self, scalar ):
51      for row in range( self.numRows() ) :
52        curNode = self._listOfRows[row]
53        while curNode is not None :
54          curNode.value *= scalar
55          curNode = curNode.next
56
57     # Creates and returns a new matrix that is the transpose of this matrix.
58    def transpose( self ):
59      ......
60
61     # Matrix addition: newMatrix = self + rhsMatrix.
62    def __add__( self, rhsMartrix ) :
63       # Make sure the two matrices have the correct size.
64      assert rhsMatrix.numRows() == self.numRows() and \
65             rhsMatrix.numCols() == self.numCols(), \
66             "Matrix sizes not compatable for adding."
67
68       # Create a new sparse matrix of the same size.
69      newMatrix = SparseMatrix( self.numRows(), self.numCols() )
70
71       # Add the elements of this matrix to the new matrix.
72      for row in range( self.numRows() ) :
73        curNode = self._listOfRows[row]
74        while curNode is not None :
75          newMatrix[row, curNode.col] = curNode.value
76          curNode = curNode.next
77
78       # Add the elements of the rhsMatrix to the new matrix.
79      for row in range( rhsMatrix.numRows() ) :
80        curNode = rhsMatrix._listOfRows[row]
81        while curNode is not None :
82          value = newMatrix[row, curNode.col]
83          value += curNode.value
84          newMatrix[row, curNode.col] = value
85          curNode = curNode.next
86
87       # Return the new matrix.
88      return newMatrix
89
90    # --- Matrix subtraction and multiplication ---
91    # def __sub__( self, rhsMatrix ) :
92    # def __mul__( self, rhsMatrix ) :
93
94  # Storage class for creating matrix element nodes.
95  class _MatrixElementNode :
96    def __init__( self, col, value ) :
97      self.col = col
98      self.value = value
99      self.next = None
```

Matrix Scaling

The `scaleBy()` method is very similar to the version used in the list implementation of the original Sparse Matrix ADT from Chapter 4. We need only traverse over each of the individual linked lists stored in the `_listOfRows` array, during which we scale the value stored in each node. Remember, this is sufficient as elements not represented by nodes in the linked lists have zero values and thus would not be affected by a scaling factor. The matrix scaling operation requires $O(k)$ time in the worst case since only the k non-zero elements are stored in the structure.

Matrix Addition

The `__add__` method for this version of the sparse matrix, which is provided in lines 62–88 of Listing 6.11, also follows the four steps outlined in Section 4.5.1. We first create a new `SparseMatrix` object that will contain the new matrix resulting from the addition. Then, the contents of the `self` or lefthand-side matrix is copied to the new matrix, one element at a time. Finally, we traverse over the non-zero elements of the righthand-side matrix and add the values of its non-zero elements to the new matrix.

This implementation of the addition operation, which requires $O(kn)$ time in the worst case, is not the most efficient. Instead of using the `__getitem__` and `__setitem__` operations, we can use temporary traversal reference variables with each matrix to directly access the non-zero values in the two source matrices and to store the resulting non-zero values in the new matrix. A new implementation can be devised that only requires $O(k)$ time in the worst case.

6.5.2 Comparing the Implementations

The Sparse Matrix ADT implemented as an array of linked lists can be evaluated for any of the three cases: best, average, and worst. The analysis, which is left as an exercise, depends on the relationship between the total number of non-zero elements, k, and the number of columns, n, in the matrix.

We implemented the Matrix ADT using a 2-D array that can be used to store a sparse matrix and we implemented the Sparse Matrix ADT using two different data structures: a list of `_MatrixElement` objects, and an array of linked lists. Table 6.2 provides a comparison of the worst case time-complexities between the three implementations for several of the matrix operations. The 2-D array implementation offers the best advantage of quick element access with the `__getitem__` and `__setitem__` methods, but the other matrix operations are more costly. While both the Python list and the array of linked lists implementations provide similar times, the array of linked lists version will typically provide better times since the efficiency for many of the operations is based on the number of columns in the matrix and not the total number of non-zero elements.

Operation	2-D Array	Python List	Linked Lists
constructor	$O(1)$	$O(1)$	$O(1)$
`s.numRows()`	$O(1)$	$O(1)$	$O(1)$
`s.numCols()`	$O(1)$	$O(1)$	$O(1)$
`x = s[i,j]`	$O(1)$	$O(k)$	$O(n)$
`s[i,j] = x`	$O(1)$	$O(k)$	$O(n)$
`s.scaleBy(x)`	$O(n^2)$	$O(k)$	$O(k)$
`r = s + t`	$O(n^2)$	$O(k^2)$	$O(kn)$

Table 6.2: Comparison of the Matrix and Sparse Matrix ADT implementations.

6.6 Application: Polynomials

Polynomials, which are an important concept throughout mathematics and science, are arithmetic expressions specified in terms of variables and constants. A polynomial in one variable can be expressed in expanded form as

$$a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x^1 + a_0$$

where each $a_i x^i$ component is called a *term*. The a_i part of the term, which is a scalar that can be zero, is called the *coefficient* of the term. The exponent of the x^i part is called the *degree* of that variable and is limited to whole numbers. For example,

$$12x^2 - 3x + 7$$

consists of three terms. The first term, $12x^2$, is of degree 2 and has a coefficient of 12; the second term, $-3x$, is of degree 1 and has a coefficient of -3; the last term, while constant, is of degree 0 with a coefficient of 7.

Polynomials can be characterized by degree (i.e., all second-degree polynomials). The degree of a polynomial is the largest single degree of its terms. The example polynomial above has a degree of 2 since the degree of the first term, $12x^2$, has the largest degree.

In this section, we design and implement an abstract data type to represent polynomials in one variable expressed in expanded form. The discussion begins with a review of polynomial operations and concludes with a linked list implementation of our Polynomial ADT.

6.6.1 Polynomial Operations

A number of operations can be performed on polynomials. We review several of these operations, beginning with addition.

Addition

Two polynomials of the same variable can be summed by adding the coefficients of corresponding terms of equal degree. The result is a third polynomial. Consider the following two polynomials:

$$5x^2 + 3x - 10$$
$$2x^3 + 4x^2 + 3$$

which we can add to yield a new polynomial:

$$(5x^2 + 3x - 10) + (2x^3 + 4x^2 + 3) = 2x^3 + 9x^2 + 3x - 7$$

Subtraction is performed in a similar fashion but the coefficients are subtracted instead. Another way to view polynomial addition is to align terms by degree and add the corresponding coefficients:

$$\begin{array}{rcccc} & & 5x^2 & 3x & -10 \\ + & 2x^3 & 4x^2 & & 3 \\ \hline & 2x^3 & 9x^2 & 3x & -7 \end{array}$$

Multiplication

The product of two polynomials is also a third polynomial. The new polynomial is obtained by summing the result from multiplying each term of the first polynomial by each term of the second. Consider the two polynomials from the previous example:

$$(5x^2 + 3x - 10)(2x^3 + 4x^2 + 3)$$

The second polynomial has to be multiplied by each term of the first polynomial:

$$5x^2(2x^3 + 4x^2 + 3) + 3x(2x^3 + 4x^2 + 3) + -10(2x^3 + 4x^2 + 3)$$

We then distribute the terms of the first polynomial to yield three intermediate polynomials:

$$(10x^5 + 20x^4 + 15x^2) + (6x^4 + 12x^3 + 9x) + (-20x^3 - 40x^2 - 30)$$

Finally, the three polynomials are summed, resulting in

$$10x^5 + 26x^4 - 8x^3 - 25x^2 + 9x - 30$$

Evaluation

The easiest operation by far is the evaluation of a polynomial. Polynomials can be evaluated by assigning a value to the variable, commonly called the unknown. By making the variable known in specifying a value, the expression can be computed, resulting in a real value. If we assign value 3 to the variable x in the equation

$$10x^5 + 26x^4 - 8x^3 - 25x^2 + 9x - 30$$

the result will be

$$10(3)^5 + 26(3)^4 - 8(3)^3 - 25(3)^2 + 9(3) - 30 = 4092$$

6.6.2 The Polynomial ADT

Given the overview of polynomials, we now turn our attention to defining the Polynomial ADT.

Define **Polynomial ADT**

A *polynomial* is a mathematical expression of a variable constructed of one or more terms. Each term is of the form $a_i x^i$ where a_i is a scalar coefficient and x^i is the unknown variable of degree i.

- `Polynomial()`: Creates a new polynomial initialized to be empty and thus containing no terms.
- `Polynomial(degree, coefficient)`: Creates a new polynomial initialized with a single term constructed from the `degree` and `coefficient` arguments.
- `degree()`: Returns the degree of the polynomial. If the polynomial contains no terms, a value of -1 is returned.
- *`getitem`*`(degree)`: Returns the coefficient for the term of the provided degree. Thus, if the expression of this polynomial is $x^3 + 4x + 2$ and a degree of 1 is provided, this operation returns 4. The coefficient cannot be returned for an empty polynomial.
- `evaluate(scalar)`: Evaluates the polynomial at the given `scalar` value and returns the result. An empty polynomial cannot be evaluated.
- *`add`*`(rhsPolynomial)`: Creates and returns a new `Polynomial` that is the result of adding this polynomial and the `rhsPoly`. This operation is not defined if either polynomial is empty.
- *`subtract`*`(rhsPoly)`: Creates and returns a new `Polynomial` that is the result of subtracting this polynomial and the `rhsPoly`. This operation is not defined if either polynomial is empty.
- *`multiply`*`(rhsPoly)`: Creates and returns a new `Polynomial` that is the result of multiplying this polynomial and the `rhsPoly`. This operation is not defined if either polynomial is empty.

Two constructors were specified for this abstract data type. Most object-oriented languages provide a mechanism to construct an object in various ways. In Python, we define a single constructor and supply default values for the arguments.

6.6.3 Implementation

To implement the Polynomial ADT, we must determine how best to represent the individual terms and how to store and manage the collection of terms. In

earlier chapters, we were limited to the use of a list or dictionary. But with the introduction of the linked list in this chapter, we now have an additional option. The linked list has the advantage of requiring fewer shifts and no underlying array management as is required with the Python list. This is especially important when working with dynamic polynomials.

Linked List Structure

We are going to implement our Polynomial ADT using a singly linked list. Given this choice, we must decide how the data should be stored and organized within the linked list. Since a polynomial is constructed as the sum of one or more non-zero terms, we can simply store an individual term in each node of the list as defined by the `PolyTermNode` class, shown in lines 65–69 of Listing 6.12.

Next, we must decide whether to order the nodes within the linked list. Upon analysis of the polynomial operations, it becomes clear an ordered list would be better since many of those operations are based on the degree of the terms. For example, the degree of the polynomial is the degree of the largest term. If the list is ordered, finding the polynomial degree is rather simple. Likewise, you will soon see that ordering the terms allows for a more efficient implementation of the addition and multiplication operations. Unlike previous examples, we are going to order the nodes in descending order based on degree since polynomials are typically written with the terms ordered from largest degree to smallest. A sample linked list structure for the polynomial $5x^2 + 3x - 10$ is illustrated in Figure 6.16.

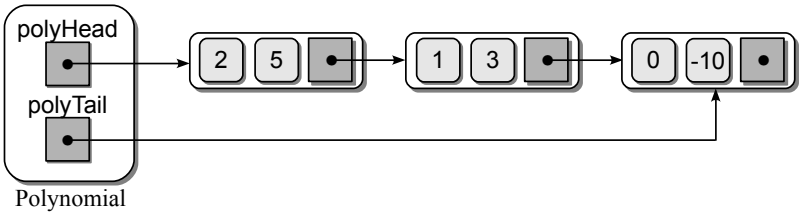


Figure 6.16: A `Polynomial` object for the polynomial $5x^2 + 3x - 10$.

Finally, we need to decide whether our implementation can benefit from the use of a tail pointer or if a head pointer alone will suffice. A rule of thumb in making this decision is whether we will be appending nodes to the list or simply inserting them in their proper position. If you need to append nodes to a linked list, you should use a tail pointer. The implementation of some of our polynomial operations can be improved if we append nodes directly to the end of the linked list. Thus, we will use and manage a tail pointer in our implementation of the Polynomial ADT.

Listing 6.12 Partial implementation of the `polynomial.py` module.

```
1  # Implementation of the Polynomial ADT using a sorted linked list.
2  class Polynomial :
3     # Create a new polynomial object.
4    def __init__(self, degree = None, coefficient = None):
5      if degree is None :
6        self._polyHead = None
7      else :
8        self._polyHead = _PolyTermNode(degree, coefficient)
9      self._polyTail = self._polyHead
10
11     # Return the degree of the polynomial.
12   def degree( self ):
13     if self._polyHead is None :
14       return -1
15     else :
16       return self._polyHead.degree
17
18     # Return the coefficient for the term of the given degree.
19   def __getitem__( self, degree ):
20     assert self.degree() >= 0,
21        "Operation not permitted on an empty polynomial."
22     curNode = self._polyHead
23     while curNode is not None and curNode.degree >= degree :
24       curNode = curNode.next
25
26     if curNode is None or curNode.degree != degree :
27       return 0.0
28     else :
29       return curNode.degree
30
31     # Evaluate the polynomial at the given scalar value.
32   def evaluate( self, scalar ):
33     assert self.degree() >= 0,
34         "Only non-empty polynomials can be evaluated."
35     result = 0.0;
36     curNode = self._polyHead
37     while curNode is not None :
38       result += curNode.coefficient * (scalar ** curNode.degree)
39       curNode = curNode.next
40     return result
41
42   # Polynomial addition: newPoly = self + rhsPoly.
43   def __add__( self, rhsPoly ):
44     ......
45
46     # Polynomial subtraction: newPoly = self - rhsPoly.
47   def __sub__( self, rhsPoly ):
48     ......
49
50     # Polynomial multiplication: newPoly = self * rhsPoly.
51   def __mul__( self, rhsPoly ):
52     ......
53
```

(Listing Continued)

Listing 6.12 Continued ...

```
54     # Helper method for appending terms to the polynomial.
55    def _appendTerm( self, degree, coefficient ) :
56      if coefficient != 0.0 :
57        newTerm = _PolyTermNode( degree, coefficient )
58        if self._polyHead is None :
59          self._polyHead = newTerm
60        else :
61          self._polyTail.next = newTerm
62        self._polyTail = newTerm
63
64  # Class for creating polynomial term nodes used with the linked list.
65  class _PolyTermNode( object ):
66    def __init__( self, degree, coefficient ):
67      self.degree = degree
68      self.coefficient = coefficient
69      self.next = None
```

Basic Operations

The Polynomial ADT calls for two constructors, one for creating an empty polynomial and the other that can be used to create a polynomial initialized with a single term supplied as an argument. In Python, we can provide multiple constructors with the use of default values. The constructor, shown in lines 4–9 of Listing 6.12, defines two data fields, the head and tail pointers, for use with the linked list implementation. These references are either initialized to `None` or set to point to the first node in the list depending on how the constructor was called.

The `degree()` method is simple to implement as it returns either the degree of the largest term that is stored in the first node or -1 if the polynomial is not defined. For our ADT, a polynomial is not defined if it does not contain any terms, which is indicated in our implementation by an empty list.

The get operation, which we implement using the subscript operator, returns the coefficient corresponding to a specific term of the polynomial identified by degree. A linear search of the linked list is required to find the corresponding term. Since the nodes are sorted by degree, we can terminate the search early if we encounter a node whose degree is smaller than the target. After the loop terminates, there are two possible conditions. If there is no non-zero term with the given degree, then `curNode` will either be `None` or pointing to a list node whose degree is smaller than the target. In this case, we must return a value of 0 since by definition a zero-term has a coefficient of 0. Otherwise, we simply return the coefficient of the corresponding term pointed to by `curNode`.

A polynomial is evaluated by supplying a specific value for the variable used to represent each term and then summing the terms. The `evaluate()` method is easily implemented as a list traversal in which a sum is accumulated, term by term. The result is a $O(n)$ time operation, where n is the degree of the polynomial.

Appending Terms

We included a tail reference in our linked list implementation for use by several of the polynomial arithmetic operations in order to perform fast append operations. While the Polynomial ADT does not define an append operation, we want to provide a helper method that implements this operation. It will be used by other methods in the class for creating efficient operations. The `_appendTerm()` helper method in lines 55–62 of Listing 6.12 accepts the degree and coefficient of a polynomial term, creates a new node to store the term, and appends the node to the end of the list. Since we only store the non-zero terms in the linked list, we must ensure the supplied coefficient is not zero before creating and appending the new node.

Polynomial Addition

The addition of two polynomials can be performed for our linked list implementation using a simple brute-force method, as illustrated in the code segment below:

```
class Polynomial :
# ...
  def simple_add( self, rhsPoly ):
    newPoly = Polynomial()
    if self.degree() > rhsPoly.degree() :
      maxDegree = self.degree()
    else
      maxDegree = rhsPoly.degree()

    i = maxDegree
    while i >= 0 :
      value = self[i] + rhsPoly[i]
      self._appendTerm( i, value )
      i += 1

    return newPoly
```

The new polynomial is created by iterating over the two original polynomials, term by term, from the largest degree among the two polynomials down to degree 0. The element access method is used to extract the coefficients of corresponding terms from each polynomial, which are then added, resulting in a term for the new polynomial. Since we iterate over the polynomials in decreasing degree order, we can simply append the new term to the end of the linked list storing the new polynomial.

This implementation is rather simple, but it's not very efficient. The element access method, which is used to obtain the coefficients, requires $O(n)$ time. Assuming the largest degree between the two polynomials is n, the loop will be executed n times, resulting in quadratic time in the worst case.

The polynomial addition operation can be greatly improved. Upon close examination it becomes clear this problem is similar to that of merging two sorted

lists. Consider the linked lists in Figure 6.17 representing three polynomials with the nodes positioned such that corresponding terms are aligned. The top two lists represent the two polynomials $5x^2 + 3x - 10$ and $2x^3 + 4x^2 + 3$ while the bottom list is the polynomial resulting from adding the other two.

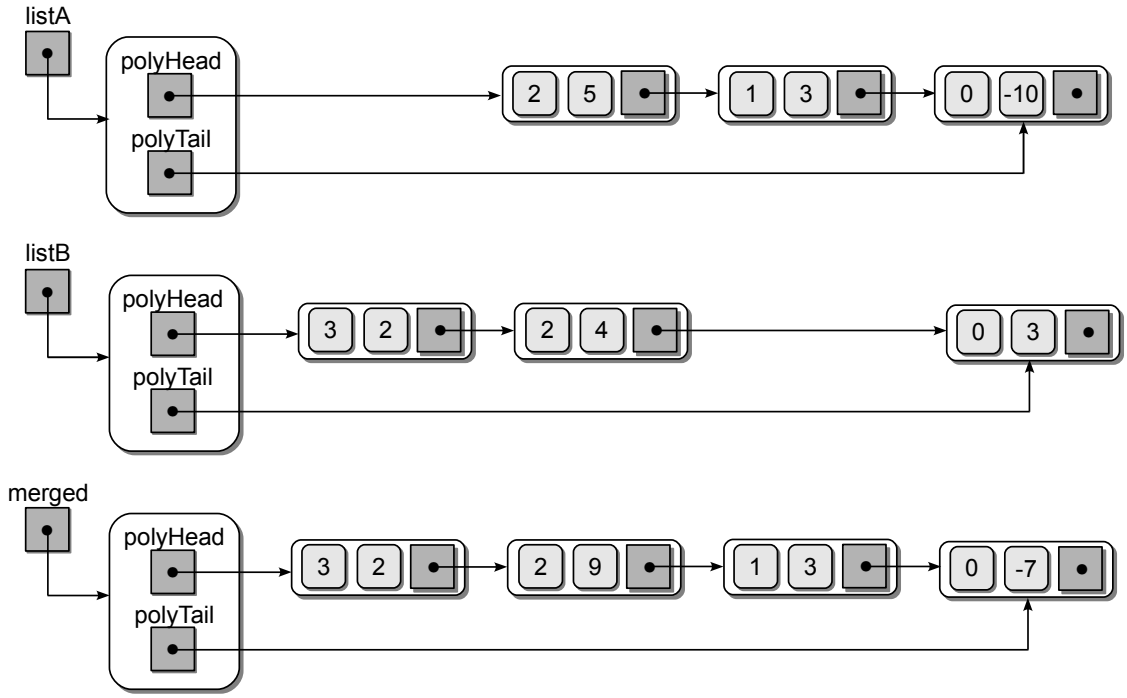


Figure 6.17: The top two linked lists store the two polynomials $5x^2 + 3x - 10$ and $2x^3 + 4x^2 + 3$. The bottom list is the resulting polynomial after adding the two original polynomials.

In Chapter 4, we discussed an efficient solution for the problem of merging two sorted lists. We also saw how that solution could be used for the set union operation, which required a new Python list containing nonduplicate items. If we use a similar approach, combining duplicate terms by adding their coefficients, we can produce a more efficient solution for our current problem of polynomial addition.

Merging two sorted arrays or Python lists, as was done in the previous chapter, is rather simple since we can refer to individual elements by index. Merging two sorted linked list requires several modifications. First, we must use temporary external references to point to the individual nodes of the two original polynomials. These references will be moved along the two linked lists as the terms are processed and merged into the new list. Next, we must utilize the `_appendTerm()` helper method to append new nodes to the resulting merged list. The implementation of the `add()` method using the list merge technique is provided in Listing 6.13.

Listing 6.13 Efficient implementation of the polynomial add operation.

```
1  class Polynomial :
2  # ...
3    def __add__( self, rhsPoly ):
4      assert self.degree() >= 0 and rhsPoly.degree() >= 0,
5           "Addition only allowed on non-empty polynomials."
6
7      newPoly = Polynomial()
8      nodeA = self._termList
9      nodeB = rhsPoly._termList
10
11      # Add corresponding terms until one list is empty.
12     while nodeA is not None and nodeB is not None :
13       if nodeA.degree > nodeB.degree :
14         degree = nodeA.degree
15         value = nodeA.coefficient
16         nodeA = nodeA.next
17       elif listA.degree < listB.degree :
18         degree = nodeB.degree
19         value = nodeB.coefficient
20         nodeB = nodeB.next
21       else :
22         degree = nodeA.degree
23         value = nodeA.coefficient + nodeB.coefficient
24         nodeA = nodeA.next
25         nodeB = nodeB.next
26       self._appendTerm( degree, value )
27
28      # If self list contains more terms append them.
29     while nodeA is not None :
30       self._appendTerm( nodeA.degree, nodeA.coefficient )
31       nodeA = nodeA.next
32
33      # Or if rhs contains more terms append them.
34     while nodeB is not None :
35       self._appendTerm( nodeB.degree, nodeB.coefficient )
36       nodeB = nodeB.next
37
38     return newPoly
```

Multiplication

Computing the product of two polynomials requires multiplying the second polynomial by each term in the first. This generates a series of intermediate polynomials, which are then added to create the final product. To aide in this operation, we create a second helper method, `_termMultiply()`, as shown in lines 23–39 of Listing 6.14, which creates a new polynomial from multiplying an existing polynomial by another term.

Using this helper method, we can now easily create a solution for the multiplication operation that simply implements the individual steps outlined earlier for multiplying two polynomials. As with the earlier `simple_add()` method, this

method is quite simple but not very efficient. The implementation of the polynomial multiplication is provided in lines 3–19 of Listing 6.14. We leave as an exercise the proof that the `__mul__` method requires quadratic time in the worst case as well as the development of a more efficient implementation.

Listing 6.14 Implementation of the polynomial multiply operation.

```
1  class Polynomial :
2  # ...
3    def multiply( self, rhsPoly ):
4      assert self.degree() >= 0 and rhsPoly.degree() >= 0,
5      "Multiplication only allowed on non-empty polynomials."
6
7       # Create a new polynomial by multiplying rhsPoly by the first term.
8      node = self._polyHead
9      newPoly = rhsPoly._termMultiply( node )
10
11      # Iterate through the remaining terms of the poly computing the
12      # product of the rhsPoly by each term.
13     node = node.next
14     while node is not None :
15       tempPoly = rhsPoly._termMultiply( node )
16       newPoly = newPoly.add( tempPoly )
17       node = node.next
18
19     return newPoly
20
21    # Helper method for creating a new polynomial from multiplying an
22    # existing polynomial by another term.
23   def _termMultiply( self, termNode ):
24     newPoly = Polynomial()
25
26      # Iterate through the terms and compute the product of each term and
27      # the term in termNode.
28     curr = curr.next
29     while curr is not None :
30        # Compute the product of the term.
31       newDegree = curr.degree + termNode.degree
32       newCoeff = curr.coefficient * termNode.coefficient
33
34        # Append it to the new polynomial.
35       newPoly._appendTerm( newDegree, newCoeff )
36
37        # Advance the current pointer.
38       curr = curr.next
39     return newPoly
```

Exercises

6.1 Implement the following functions related to the singly linked list:

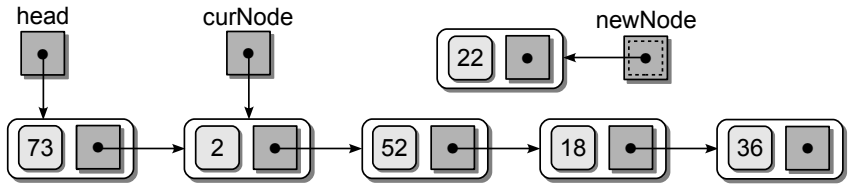
(a) The `removeAll(head)` function, which accepts a head reference to a singly linked list, unlinks and remove every node individually from the list.

(b) The `splitInHalf(head)` function, which accepts a head reference to a singly linked list, splits the list in half and returns the head reference to the head node of the second half of the list. If the original list contains a single node, `None` should be returned.

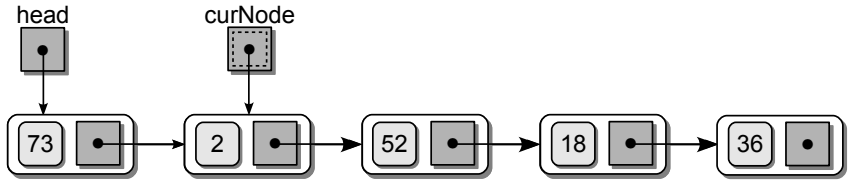
6.2 Evaluate the following code segment which creates a singly linked list. Draw the resulting list, including the external pointers.

```
box = None
temp = None
for i in range( 4 ) :
  if i % 3 != 0 :
    temp = ListNode( i )
    temp.next = box
    box = temp
```

6.3 Consider the following singly linked list. Provide the instructions to insert the new node immediately following the node containing 45. Do not use a loop or any additional external references.



6.4 Consider the following singly linked list. Provide the instructions to remove the node containing 18. Do not use a loop or any additional external references.



6.5 The following questions are related to the Sparse Matrix ADT.

(a) Implement the remaining methods of the `SparseMatrix` class presented in the chapter using the array of sorted linked lists: `__getitem__`, `transpose()`, `__sub__`, and `__mul__`.

(b) Determine the time-complexity for each of the `SparseMatrix` methods implemented in part (a).

(c) Prove or show that the matrix addition operation of the `SparseMatrix` class, as implemented in the chapter using an array of sorted linked lists, has a worst case run time of $O(kn)$.

(d) As you proved in part (c), the implementation of the `SparseMatrix` `__add__` method presented in the chapter is $O(kn)$. A more efficient implementation is possible without the use of the `__getitem__` and `__setitem__` methods. Design and implement a new version of the `__add__` method that has a run time of no more than $O(k)$.

(e) Show that your implementation of the `__add__` method from part(c) has a worst case run time of $O(k)$.

(f) What advantages are there to using sorted linked lists with the Sparse Matrix ADT instead of unsorted linked lists?

6.6 In Programming Project 4.1, you implemented the Sparse Life Grid ADT that creates a game grid of unlimited size for use with the game of Life. That implementation used a single Python list to store the individual live cells, which was similar to the technique we used with the Sparse Matrix ADT. Explain why the array of linked lists structure used to implement the Sparse Matrix ADT in this chapter cannot be used to implement the Sparse Life Grid ADT.

6.7 Prove or show that the worst case time for the `__mul__` method of the `Polynomial` class implemented in this chapter is $O(n^2)$.

Programming Projects

6.1 We have provided two implementations of the Set ADT in Chapter 1 and Chapter 4.

(a) Implement a new version of the Set ADT using an unsorted linked list.

(b) Implement a new version of the Set ADT using a sorted linked list.

(c) Evaluate your new implementations to determine the worst case run time of each operation.

(d) Compare the run times of your new versions of the Set ADT to those from Chapter 1 and Chapter 4.

6.2 Consider the Vector ADT from Programming Project 2.1:

(a) Implement a new version of the ADT using an unsorted linked list.

(b) Evaluate your new implementation to determine the worst case run time of each operation.

(c) Compare the run times of your new version of the Vector ADT to that of the original in Programming Project 2.1.

(d) What are the advantages and disadvantages of using a linked list to implement the Vector ADT?

6.3 Consider the Map ADT from Section 3.2:

(a) Implement a new version of the Map ADT using an unsorted linked list.

(b) Implement a new version of the Map ADT using a sorted linked list.

(c) Evaluate your new implementations to determine the worst case run time of each operation.

(d) Compare the run times of your new versions of the Map ADT to those from Section 3.2 and Programming Project 5.2.

6.4 Implement the `__sub__` method for the `Polynomial` class implemented in the chapter.

6.5 The implementation of the `Polynomial __mul__` method is $O(n^2)$ in the worst case. Design and implement a more efficient solution for this operation.

6.6 Provide a new implementation of the Polynomial ADT to use a Python list for storing the individual terms.

6.7 Integer values are implemented and manipulated at the hardware-level, allowing for fast operations. But the hardware does not supported unlimited integer values. For example, when using a 32-bit architecture, the integers are limited to the range -2,147,483,648 through 2,147,483,647. If you use a 64-bit architecture, this range is increased to the range -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807. But what if we need more than 19 digits to represent an integer value?

In order to provide platform-independent integers and to support integers larger than 19 digits, Python implements its integer type in software. That means the storage and all of the operations that can be performed on the values are handled by executable instructions in the program and not by the hardware. Learning to implement integer values in software offers a good example of the need to provide efficient implementations. We define the Big Integer ADT below that can be used to store and manipulate integer values of any size, just like Python's built-in `int` type.

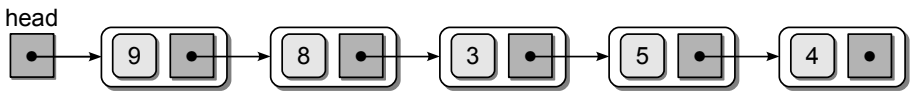
- `BigInteger( initValue = "0" )`: Creates a new big integer that is initialized to the integer value specified by the given string.
- *`toString`*`()`: Returns a string representation of the big integer.
- *`comparable`*`( other )`: Compares this big integer to the `other` big integer to determine their logical ordering. This comparison can be done using any of the logical operators: `<, <=, >, >=, ==, !=`.
- *`arithmetic`*`( rhsInt )`: Returns a new `BigInteger` object that is the result of performing one of the arithmetic operations on the `self` and `rhsInt` big integers. Any of the following operations can be performed:

```
+    -    *    //    %     **
```

- *`bitwise-ops`*`( rhsInt )`: Returns a new `BigInteger` object that is the result of performing one of the bitwise operators on the `self` and `rhsInt` big integers. Any of the following operations can be performed:

```
|    &    ^     <<    >>
```

(a) Implement the Big Integer ADT using a singly linked list in which each digit of the integer value is stored in a separate node. The nodes should be ordered from the least-significant digit to the largest. For example, the linked list below represents the integer value 45,839:



(b) Implement the Big Integer ADT using a Python list for storing the individual digits of an integer.

6.8 Modify your implementation of the Big Integer ADT from the previous question by adding the assignment combo operators that can be performed on the `self` and `rhsInt` big integers. Allow for any of the following operations to be performed:

```
+=    -=    *=    //=    %=     **=
<<=   >>=   |=    &=     ^=
```