CHAPTER 9

Advanced Linked Lists

In Chapter 6, we introduced the linked list data structure and saw how it can be used to improve the construction and management of lists for certain types of applications. In that discussion, we limited the focus to the singly linked list in which traversals start at the front and progress, one element at a time, in a single direction. But there are a number of variations to the linked list structure based on how the nodes are linked and how many chains are constructed from those links. In this chapter, we introduce some of the more common linked list variations.

9.1 The Doubly Linked List

The singly linked list introduced in Chapter 6 consists of nodes linked in a single direction. Access and traversals begin with the first node and progress toward the last node, one node at a time. But what if we want to traverse the nodes in reverse order? With the singly linked list, this can be done but not efficiently. We would have to perform multiple traversals, with each traversal starting at the front and stopping one node earlier than on the previous traversal. Or consider the problem in which you are given a reference to a specific node and need to insert a new node immediately preceding that node. Since the predecessor of a given node cannot be directly accessed, we would have to use a modified insertion operation in which the list is traversed from the first node until the predecessor of the given node is found. Again, this is not an efficient solution. For these operations and a number of others, we need direct access to both the node following and immediately preceding a given node.

9.1.1 Organization

In a ***doubly linked list***, each node contains not only the data component and a link to the next node as in the singly linked list, but also a second link that points

to the preceding node, as illustrated in Figure 9.1. To create the individual nodes, we must add a third field to the node class, which we name `DListNode` to reflect its use with a doubly linked list, as shown in Listing 9.1.

Listing 9.1 Storage class for a doubly linked list node.

```
1 class DListNode :
2   def __init__( self, data ):
3     self.data = data
4     self.next = None
5     self.prev = None
```

A head reference is again used to reference the first node in the list. A tail reference, although optional, is commonly used with a doubly linked list to take advantage of the reverse chain, which allows for traversals from back to front. The last node is indicated by a null reference in the `next` link of the last node as was done in the singly linked list. But we must also indicate the first node since the list can be traversed in reverse order. This is done using a null reference in the `prev` link of the first node.

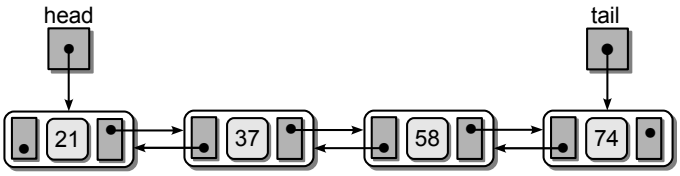

Figure 9.1: A doubly linked list with four nodes.

A doubly linked list can be sorted or unsorted depending on the specific application. The implementation of the various operations for an unsorted doubly linked list are very similar to those of the unsorted singly linked list. We leave the implementation of these operations as an exercise and only focus on the use and management of sorted doubly linked lists.

9.1.2 List Operations

The position of the nodes in a sorted doubly linked list are based on the key value of the corresponding data item stored in each node. The basic linked list operations can also be performed on a doubly linked list.

Traversals

The doubly linked list allows for traversals from front to back or back to front. The traversals are performed using a temporary external reference to mark the current node. The only difference is which node we start with and which link field is used to advance the temporary reference. Traversing the doubly linked list from

beginning to end is identical to that with a singly linked list. We start at the node referenced by `head` and advance the temporary reference, `curNode`, one node at a time, using the `next` link field. The reverse traversal, provided in Listing 9.2, starts at the node referenced by `tail` and advances `curNode`, one node at a time, using the `prev` link field. In either case, the traversal terminates when `curNode` is set to null, resulting in a $O(n)$ linear time operation.

Listing 9.2 Traversing a doubly linked list in reverse order.

```
1  def revTraversal( tail ):
2    curNode = tail
3    while curNode is not None :
4      print( curNode.data )
5      curNode = curNode.prev
```

Searching

Searching for a specific item in a doubly linked list, based on key value, can be implemented in the same fashion as for a singly linked list. Start with the first node and iterate through the list until we find the target item or we encounter a node containing a key value larger than the target. But a doubly linked list provides an advantage not available in the singly linked list. Since we can move forward or backward, the search operations do not have to begin with the first node.

Suppose we perform a sequence of search operations on the list from Figure 9.1 and we begin that sequence with a search for value 58. Using the external reference `probe` to iterate through the list, the target is found in the third node. If we leave `probe` pointing to the third node, as illustrated in Figure 9.2, we can begin the next search operation where the previous search left off instead of starting over from the beginning. Suppose the next search is for value 37. We can compare this target value to the item currently referenced by `probe` and determine if the search should proceed forward or backward starting from the `probe` node. In this case, we search backward, using `probe` to traverse the list. Thus, each time a search is performed, we leave the `probe` reference where the previous search ended and use it for subsequent searches. Note that `probe` is not a temporary reference variable as `curNode` was in the normal search operation, but must be maintained between operations in the same fashion as `head` and `tail`. Given the three external

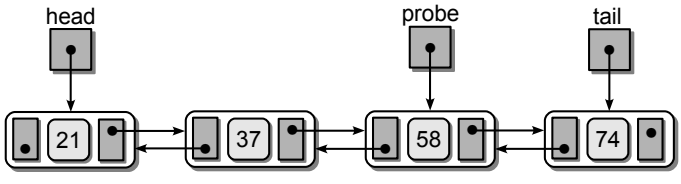

Figure 9.2: A probe reference positioned after searching for value 58.

references, `head`, `tail`, and `probe`, Listing 9.3 provides an implementation for searching a sorted doubly linked list using the probing technique.

Listing 9.3 Probing a doubly linked list.

```
# Given the head, tail, and probe references, probe the list for a target.

 # Make sure the list is not empty.
if head is None :
  return False
 # If probe is null, initialize it to the first node.
elif probe is None :
  probe = head

 # If the target comes before the probe node, we traverse backward;
 # otherwise traverse forward.
if target < probe.data :
  while probe is not None and target <= probe.data :
    if target == probe.data :
      return True
    else :
      probe = probe.prev
else :
  while probe is not None and target >= probe.data :
    if target == probe.data :
      return True
    else :
      probe = probe.next

 # If the target is not found in the list, return False.
return False
```

As with a singly linked list, all of the list operations must also work with an empty list, indicated by a null `head` reference. With the probe operation, the search simply fails if the list is empty. This can be quickly determined by examining the `head` reference.

The `probe` reference can also be null at the beginning of the search operation. This can occur when the first search is performed, since the `probe` reference has not been positioned within the list by a previous search operation. We could manage `probe` within the insert and deletion operations and make adjustments each time a new node is added or deleted. But performing this check within the search operation is just as easy, especially since this is the only operation in which `probe` is utilized. The `probe` reference can also become null when the previous search fails after the traversal has exhausted all possible nodes and the external reference "falls off the list." But unlike with a null head reference, the search does not necessarily fail. Instead, the `probe` reference must be initialized to point to the first node in the list to prepare for the current search.

The actual search is performed by either traversing forward or backward depending on the relation between the target and the key value in the node referenced by `probe`. If the target is smaller than `probe`'s key, then a reverse sorted list traver-

sal is performed; otherwise a normal forward traversal is used. If the target is found during the iteration of the appropriate loop, the function is terminated and `True` is returned. Otherwise, the loop will terminate after exhausting all possible nodes or stopping early when it's determined the target cannot possibly be in the list. In either case, `False` is returned as the last operation of the function.

We can add a quick test to determine if the target value cannot possibly be in the list. This is done by comparing the target to the first and last nodes in the list. If the target is smaller than the first item or larger than the last, we know the search will ultimately fail since the target is not in the list. This step can be omitted as the same results can be achieved by the search traversal follow, but it can help to improve the search time on large lists.

The search operation, which maintains and uses a third external reference variable, can improve the search time for a large number of sequential searches performed on large lists. But in the worst case, it remains a linear time operation since a complete traversal may have to be performed.

Adding Nodes

Adding new nodes to a sorted doubly linked list is done similar to that for the singly linked list. The only difference is we do not need to keep track of the preceding node. Once the position for the new node is located, we can access its predecessor using the appropriate `prev` field. Consider Figure 9.3, which illustrates inserting a new node into the middle of a sorted doubly linked list. The location of the new node is found by positioning a single temporary external reference variable to point to the node containing the first value larger than the new value. After this position is found, the new node can be linked into the list using the various `prev` and `next` node fields as illustrated. The resulting list after inserting the new node is illustrated in Figure 9.4 on the next page.

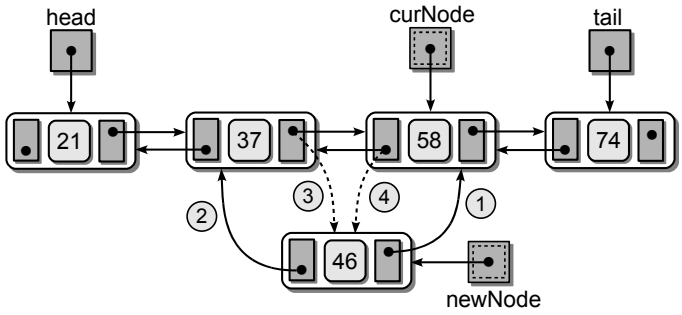

Figure 9.3: Linked modifications required to insert value 46 into the doubly linked list.

As with a singly linked list, inserting a node into a non-empty sorted doubly linked list can occur in one of three places: at the front, at the end, or somewhere in the middle. Inserting a new node into the middle of a list was illustrated above.

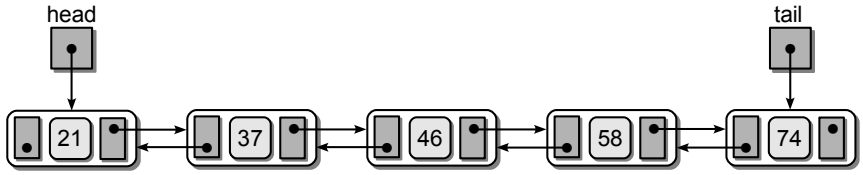

Figure 9.4: The result of inserting the new node into the doubly linked list.

Figure 9.5 illustrates the links required to insert a node at the front or end of a doubly linked list.

The code for adding a node to a sorted doubly linked list is provided in Listing 9.4. Since a tail reference is commonly used with a doubly linked list, we can provide a more efficient solution by dividing the operation into the four different cases. This reduces the need for loop traversal when the list is empty or the new node is prepended or appended to the list.

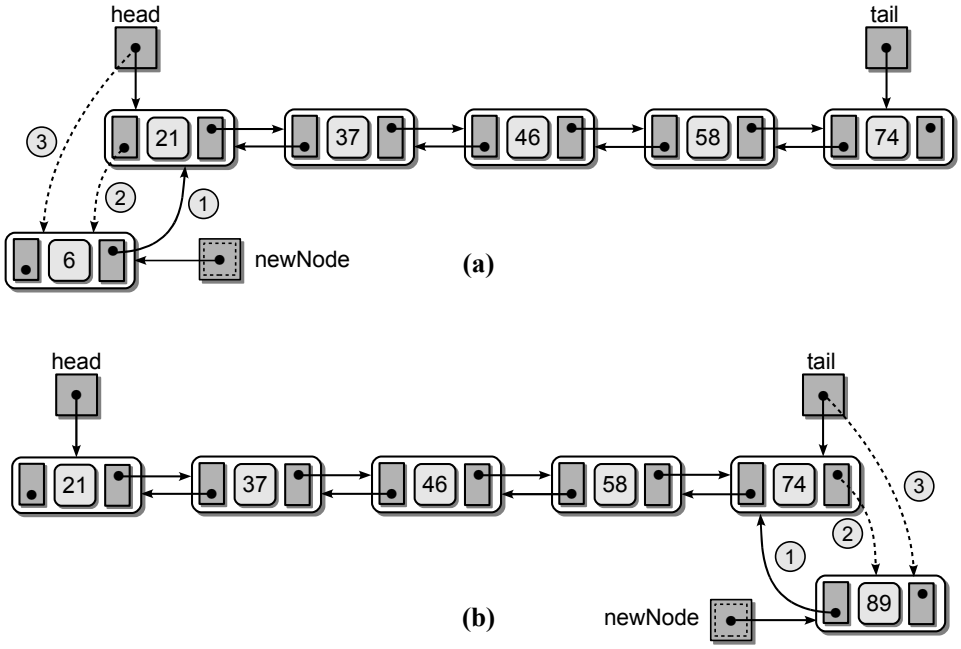

Figure 9.5: The links required to insert a new node into a sorted doubly linked list at the (a) front or (b) back.

Removing Nodes

Deleting a node from a doubly linked list is done in a similar fashion to that for a singly linked list. Again, there is no need to position a `pred` temporary external reference variable since the predecessor of a given node can be accessed using the appropriate `prev` fields. Implementation of the delete operation is left as an exercise.

Listing 9.4 Inserting a value into an ordered doubly linked list.

```
# Given a head and tail reference and a new value, add the new value to a
# sorted doubly linked list.

newnode = DListNode( value )
if head is None :         # empty list
  head = newnode
  tail = head
elif value < head.data :  # insert before head
  newnode.next = head
  head.prev = newnode
  head = newnode
elif value > tail.data :  # insert after tail
  newnode.prev = tail
  tail.next = newnode
  tail = newnode
else :                    # insert in the middle
  node = head
  while node is not None and node.data < value :
    node = node.next

  newnode.next = node
  newnode.prev = node.prev
  node.prev.next = newnode
  node.prev = newnode
```

9.2 The Circular Linked List

Another variation of the linked list structure is the ***circular linked list*** in which the nodes form a continuous circle. The circular linked list allows for a complete traversal of the nodes starting with any node in the list. It is also commonly used for applications in which data is processed in a round-robin fashion. For example, when scheduling the use of a CPU, the operating system must cycle through all of the processes running on a computer, allowing each an opportunity to use the processor for a short period of time. By using a circular linked list, the choice of the next process is made in a round-robin fashion.

9.2.1 Organization

The nodes in a circular linked list, as illustrated in Figure 9.6, are organized in a linear fashion the same as those in a singly or doubly linked list. In fact, a circular linked list can have single or double links. In a singly linked version, every node has a successor while in a doubly linked version every node has a successor and predecessor.

The nodes in a circular list have the same structure as those in the linear list versions. The only difference is the `next` field of the last node links to the first, and in the doubly linked version, the `prev` field of the first node links to the last.

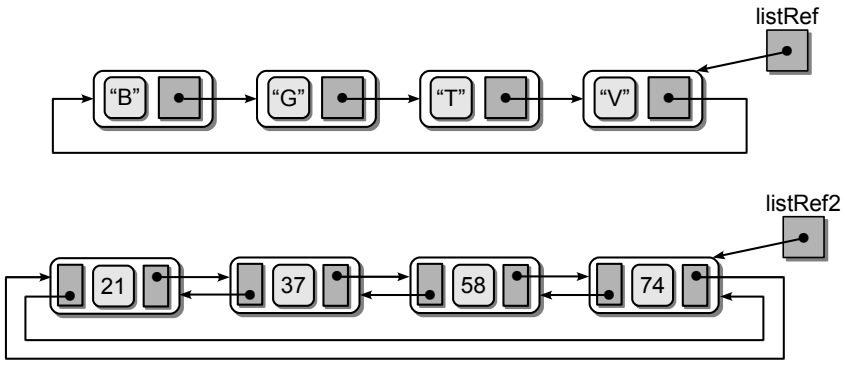

Figure 9.6: Examples of circular linked lists.

A single external reference variable is used to point to a node within the list. Technically, this could be any node in the list, but for convenience, the external reference is commonly set to the last node added to the list. By referencing that node, we have quick access to both the first node added and the last.

9.2.2 List Operations

The common set of operations performed on the linear versions of the linked list can also be performed on a circular linked list. However, some modifications are required in the algorithm for each operation to take into account the location of the external reference variable and the fact that none of the nodes have a null `next` field. In this section, we describe the various list operations for use with a sorted singly linked circular list. These operations require only slight modifications when applied to an unsorted circular list or the doubly linked version.

Traversals

Traversing a circular linked list requires a temporary external reference that is moved through the list, one node at a time, just like the linear versions. For the circular list, several modifications are required, all of which are necessary in order to perform a complete traversal while terminating the loop at the proper time. Terminating the loop is not as straightforward with the circular list since there are no nodes that have their `next` field set to `None`. Even in a single-node list, the link field of the last node always points back to the first node.

We cannot simply initialize the `curNode` reference to the first node and traverse until it "falls off" the list as was done in the linear versions. Instead, we need a way to flag the end of the traversal independent of the link fields. The algorithm for traversing a circular linked list is provided in Listing 9.5.

The traversal process begins by initializing `curNode` to reference the last node in the list, as illustrated in Figure 9.7(a). The traversal reference begins at the last node instead of the first because we are going to terminate the traversal when it again reaches the last node, after iterating over the entire list. The termination

Listing 9.5 Traversing a circular linked list.

```
1 def traverse( listRef ):
2   curNode = listRef
3   done = listRef is None
4   while not done :
5     curNode = curNode.next
6     print curNode.data
7     done = curNode is listRef
```

of the loop is handled by the boolean variable `done`, which is initialized based on the status of `listRef`. If the list is empty, `done` will be set to `False` and the loop never executes. When the list contains at least one node, `done` will be initialized to `True` and the loop is executed.

In the linear versions of the linked list, the first step in the body of the loop was to visit the current node (i.e., print the contents of the `node.data` field) and then advance the traversal reference to the next node. In this version, we must first advance `curNode`, as illustrated in Figure 9.7(b), and then perform the visit operation. Remember, `curNode` is initialized to reference the last node in the list. In order to begin the traversal with the first node, the temporary reference must be advanced to the first node in the list.

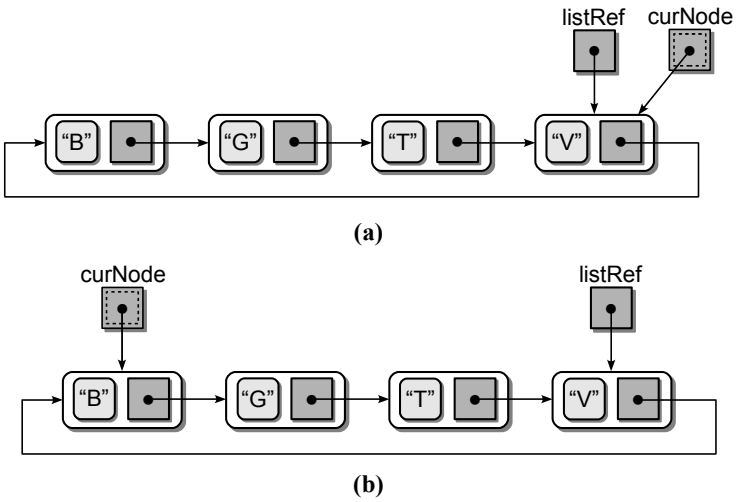

Figure 9.7: Traversing a circular linked list: (a) initial assignment of the temporary reference and (b) advancement of the temporary reference to the first node in the list.

Finally, `done` is updated by examining `curNode` to determine if it has again reached the last node in the list, as referenced by the `listRef` reference. `curNode` was initialized to be an alias of `listRef`, but it was advanced at the beginning of the loop. Thus, when `curNode` again reaches the end of the list, we know every node has been visited and the loop can terminate.

We must ensure this operation also works for a list containing a single node, as illustrated in Figure 9.8. When `curNode` is advanced to the next node, it actually still references itself. This is appropriate since the node is both the first and last node in the list. After the node is visited and the data printed in this example, `curNode` is advanced once again. When the alias test is performed at the bottom of the loop, `done` will be set to `True` and the loop will terminate properly.

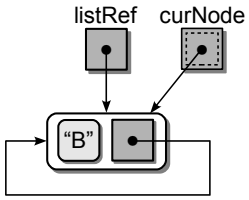

Figure 9.8: Traversing a circular linked list containing a single node.

Searching

The search operation requires a traversal through the list, although it can terminate early if we encounter the target item or reach an item larger than the target. The implementation, shown in Listing 9.6, closely follows that of the traversal operation from earlier.

Listing 9.6 Searching a circular linked list.

```
1  def searchCircularList( listRef, target ):
2    curNode = listRef
3    done = listRef is None
4    while not done :
5      curNode = curNode.next
6      if curNode.data == target :
7        return True
8      else :
9        done = curNode is listRef or curNode.data > target
10   return False
```

As with the traversal operation, the initialization of `done` handles the case where the list is empty and prevents the loop from ever being executed. The modification of the `done` flag at the bottom of the loop handles the termination of the traversal when the target is not found. A compound logical expression is used in order to test both conditions: a complete traversal was performed or we reached an item larger than the target. Finally, the condition in the `if` statement handles the case where the target item is found in the list.

Adding Nodes

Adding nodes to an ordered circular linked list is very similar to that of the ordered linear versions. The major difference is the management of the loopback link from the last node to the first. The implementation is much simpler if we divide it into four cases, as illustrated in Figure 9.9: (1) the list is empty when adding the node; (2) the new node is inserted at the front; (3) the new node is inserted at the end; or (4) the node is inserted somewhere in the middle. The first three cases are straightforward and only require a few adjustments in the links. The fourth case requires a search to find the proper location for the new node and the positioning of the two external reference variables. This requires a traversal similar to the one used earlier with the search operation. The Python code segment shown in Listing 9.7 provides the implementation of the insert operation. It should be noted again that the illustrations show the ordering in which the links should be made to avoid accidentally unlinking and destroying existing nodes in the list.

Listing 9.7 Inserting a node into an ordered circular linked list.

```
# Given a listRef pointer and a value, add the new value
# to an ordered circular linked list.

newnode = ListNode( value )
if listRef is None :              # empty list
  listRef = newNode
  newNode.next = newNode
elif value < listRef.next.data :  # insert in front
  newNode.next = listRef.next
  listRef.next = newNode
elif value > listRef.data :       # insert in back
  newNode.next = listRef.next
  listRef.next = newNode
  listRef = newNode
else :                            # insert in the middle
   # Position the two pointers.
  predNode= None
  curNode = listRef
  done = listRef is None
  while not done :
    predNode = curNode
    predNode = curNode.next
    done = curNode is listRef or curNode.data > target

   # Adjust links to insert the node.
  newNode.next = curNode
  predNode.next = newNode
```

Removing Nodes

Deleting a node from a circular linked list, for both ordered and unordered lists, closely follows the steps required for inserting a node into an ordered circular list. The implementation of this operation is left as an exercise.

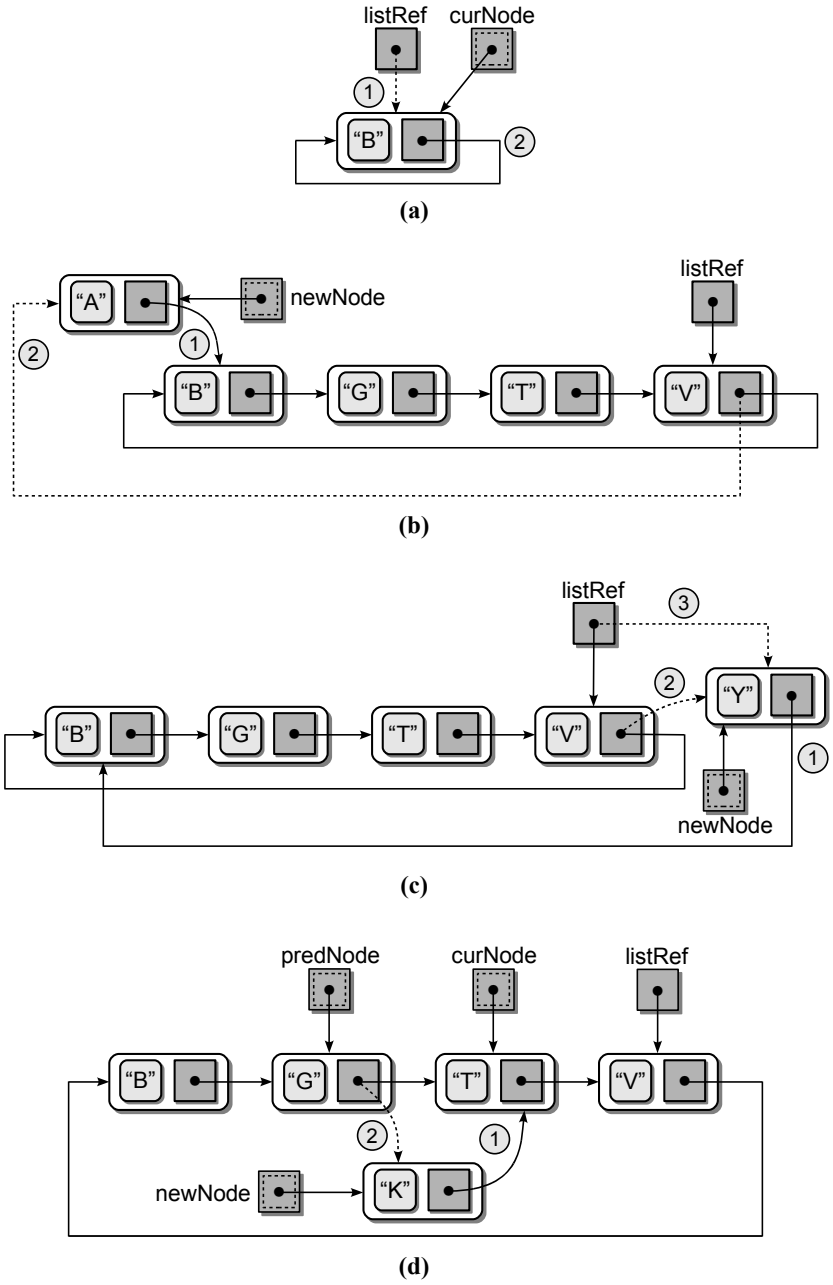

Figure 9.9: The links required to insert a new node into a circular list: (a) inserting the first node into an empty list; (b) prepending to the front; (c) appending to the end; and (d) inserting in the middle.

9.3 Multi-Linked Lists

The doubly linked list is a special instance of the more general multi-linked list. A ***multi-linked list*** is one in which each node contains multiple link fields which are used to create multiple chains within the same collection of nodes. In the doubly linked list, there are two chains through the collection of nodes, each using the key field to form the chains. The result is two singly linked lists in which one orders the nodes by key value in increasing order while the second orders the nodes in reverse order.

9.3.1 Multiple Chains

In a multi-linked list, chains can be created using multiple keys or different data components to create the multiple links. Consider the example of student records introduced in Chapter 1. Suppose we want to create a multi-linked list containing two chains in which the nodes are linked and sorted by id number in one chain and by name in the other, as illustrated in Figure 9.10. The first node in each chain is referenced by a separate head pointer. The `listById` reference indicates the first node in the chain sorted by id number while `listByName` indicates the first node in the chain sorted by name. For visual aid, the id chain is represented by solid lines and darker gray link fields while the name chain is represented by dashed lines and slightly lighter gray link fields.

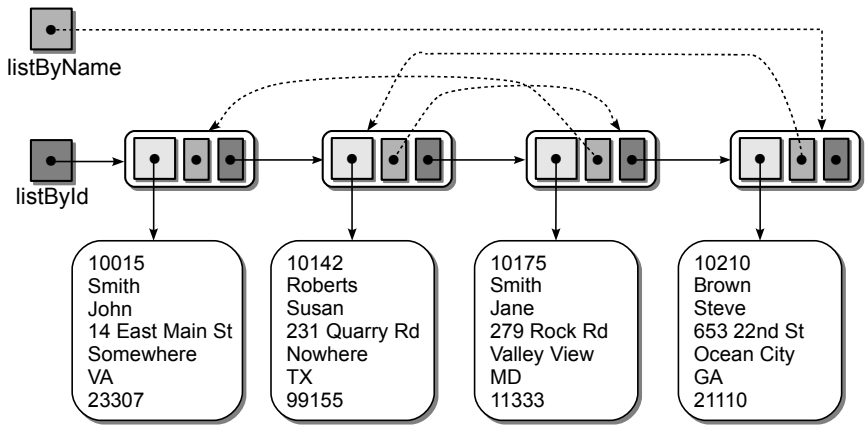

Figure 9.10: A multi-linked list storing student records with two complete chains.

The nodes in the multi-linked list can be traversed by either chain, the dashed links or the solid links, based on the desired order. It is up to the programmer to ensure he starts with the correct head node and follows the correct links. To create a multi-linked list, the nodes must contain multiple link fields, one for each chain. The `StudentMListNode` storage class in Listing 9.8 shows the definition used for the multi-linked list from Figure 9.10.

Listing 9.8 The node class for a multi-linked list.

```
1 class StudentMListNode :
2   def __init__( self, data ):
3     self.data = data
4     self.nextById = None
5     self.nextByName = None
```

When inserting nodes into the multi-linked list, a single node instance is created, but two insertions are required. After creating the new node, the multi-linked list is treated as two separate singly linked lists. Thus, the node must first be inserted into the chain ordered by id and then in the chain ordered by name. Similar action is required when deleting a node from the multi-linked list, but in this case, the node must be removed from both chains.

In the previous example, the two chains form two complete lists. That is, all of the nodes are part of both chains. The second chain could form a partial list instead. For example, suppose we wanted to create a multi-linked list of student records with one chain sorted by id and another containing only those students whose last name is "Smith," as illustrated in Figure 9.11. The former will contain all of the nodes and form a complete list, but the latter will only contain a subset of the nodes. Multi-linked lists can be organized in any number of ways with the resulting design based on the problem being solved.

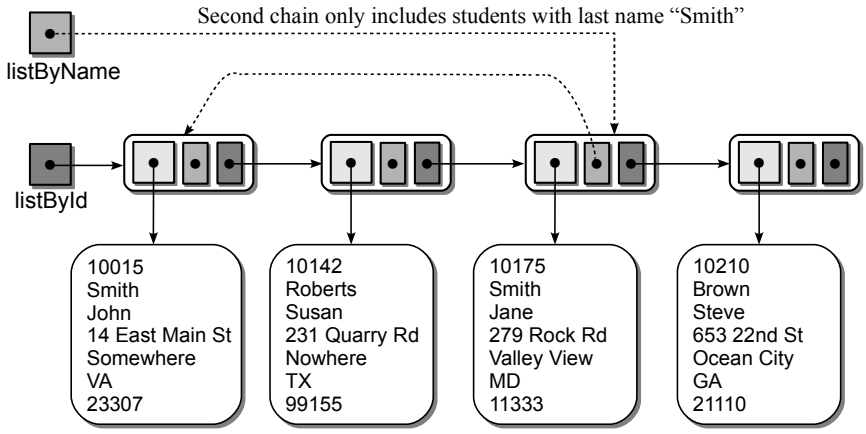

Figure 9.11: A multi-linked list storing with one complete chain and one partial chain.

9.3.2 The Sparse Matrix

A common use of a multi-linked list is an alternative implementation of the Sparse Matrix ADT, which was introduced in earlier chapters. In Chapter 4, we designed a solution for the sparse matrix using a Python list to store the non-zero elements. We improved on the original solution in Chapter 6 by using an array of linked lists to store the non-zero elements with a separate list for each row in the matrix. While both improved on the 2-D array implementation, the latter provided more

efficient solutions for many of the operations since traversals could be limited to a per-row basis instead of a complete traversal of all non-zero elements. Some matrix operations and applications, however, require traversals in column order instead of row order. By organizing the non-zero elements based on the matrix rows, the time required in these cases can actually be worse than with a single linked list.

To provide convenient traversals of both rows and columns, we can use a multi-linked organization, as illustrated in Figure 9.12. The nodes are linked by two chains along the respective row and column. This requires two arrays of linked lists, one for the row lists and another for the column lists.

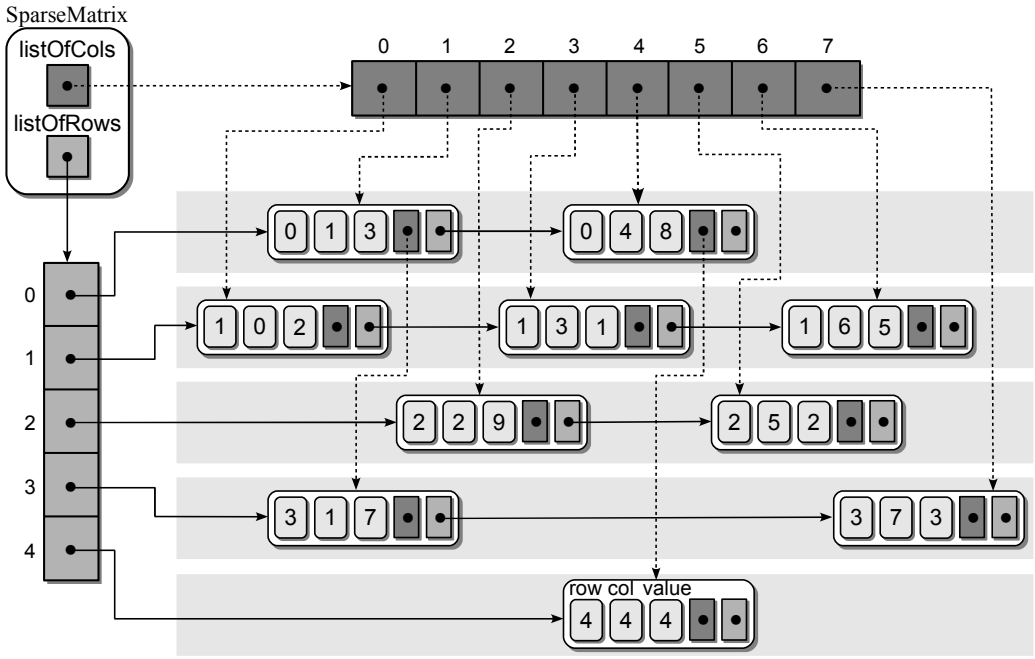

Figure 9.12: A multi-linked list implementation of the sparse matrix from Figure 4.5.

An individual element can be accessed by traversing the row chain or the column chain as needed. This dual access requires that each node contain both the row and column indices in addition to the two link fields, as shown in Listing 9.9. The implementation of the Sparse Matrix ADT using a multi-linked list is left as an exercise.

Listing 9.9 The multi-linked nodes for implementing the Sparse Matrix ADT.

```
1  class MatrixMListNode :
2    def __init__( self, row, col, value ) :
3      self.row = row
4      self.col = col
5      self.value = value
6      self.nextCol = None
7      self.nextRow = None
```

9.4 Complex Iterators

The iterators designed and used in earlier chapters were examples of simple iterators since we only needed to maintain a single traversal variable. A more complex example would be the addition of an iterator to the `SparseMatrix` class implemented as an array of linked lists in the previous chapter or the multi-linked list version defined in this chapter. Both of those implementations used an array of linked lists to store the matrix elements. To build an iterator in this case, we must keep track of the array of linked lists and both the current row within the array and the current node within the linked list for the given row. Listing 9.10 provides an implementation of an iterator for the Sparse Matrix ADT implemented using a multi-linked list.

When the iterator is first created, the constructor must initialize the `_curRow` and `_curNode` fields to reference the first node in the first non-empty row. Since

Listing 9.10 An iterator for the Sparse Matrix ADT implemented using a multi-linked list.

```
# Defines a Python iterator for the Sparse Matrix ADT implemented using
# an array of linked lists.

class _SparseMatrixIterator :
  def __init__( self, rowArray ):
    self._rowArray = rowArray
    self._curRow = 0
    self._curNode = None
    self._findNextElement()

  def __iter__( self ):
    return self

  def next( self ):
    if self._curNode is None :
      raise StopIteration
    else :
      value = self._curNode.value
      self._curNode = self._curNode.next
      if self._curNode is None :
        self._findNextElement()
      return value

  def _findNextElement( self ):
    i = self._curRow
    while i < len( self._rowArray ) and
          self._rowArray[i] is None :
      i += 1

    self._curRow = i
    if i < len( self._rowArray ) :
      self._currNode = self._rowVector[i]
    else :
      self._currNode = None
```

every row in the sparse matrix does not necessarily contain elements, we must search for the first node. This same operation will be required when advancing through the sparse matrix as we reach the end of each row, so we include the `_findNextElement()` helper method to find the next node in the array of linked lists.

To find the next node, the helper method starts at the current row and increments a counter until the next non-empty row is found or all rows have been processed. If the counter references a non-empty row, `_curNode` is set to the first node in that list. Otherwise, it is set to `None`, which will signal the end of the traversal loop.

The `next()` method of the `_SparseMatrixIterator` class need only examine the `_curNode` field to determine if the `StopIteration` exception should be raised or a value is to be returned. If `_curNode` is not null, we first save the value from the current node and then advance the reference to the next node in the list. When reaching the end of the linked list representing the current row, the `_findNextElement()` must be called to search for the next non-empty row in the array of linked lists. The operation concludes by returning the value saved before advancing `_curNode` to the next node.

9.5 Application: Text Editor

If you have used a computer, it's very likely you have used a text editor. Text editors allow us to enter text to create documents or to enter information into form fields like those found on some web pages. Editors can be very complex like a word processor or rather simple like Microsoft's *Notepad*™. No matter the complexity, all text editors must maintain and manage the text as the user enters, deletes, and manipulates characters and lines. In this section, we define an Edit Buffer ADT, which can be used by a simple text editor to store and manage the text of a document being manipulated by the actual editor. The ADT is defined independent of the text editor and only contains operations necessary for the storage and manipulation of plain text. This ADT is not meant to be used by more complex editors such as those that perform syntax highlighting or even by word processors, which must also maintain character properties like font style and size. We conclude with an implementation of our Edit Buffer ADT and discuss other alternatives.

9.5.1 Typical Editor Operations

While the design and implementation of an actual text editor is beyond the scope of this textbook, the examination of an edit storage buffer is not. We begin our discussion by examining the workings of typical text editors that will aide in the design of our ADT.

Layout

Text editors typically work with an abstract view of the text document by assuming it is organized into rows and columns as illustrated in Figure 9.13. The physical storage of a text document depends on the underlying data structure, though when stored on disk, the text file is simply a sequential stream of characters, as illustrated in Figure 9.14. The end of each line in a text document contains a newline character (`\n`) resulting in rows of varying sizes of unlimited length. An empty document is assumed to contain a single blank line where a blank line would consist of a single newline character. As a visual aid, the illustrations in the two diagrams use the ↩ symbol to indicate the actual location of the newline character and a small bullet (·) to indicate blank spaces.

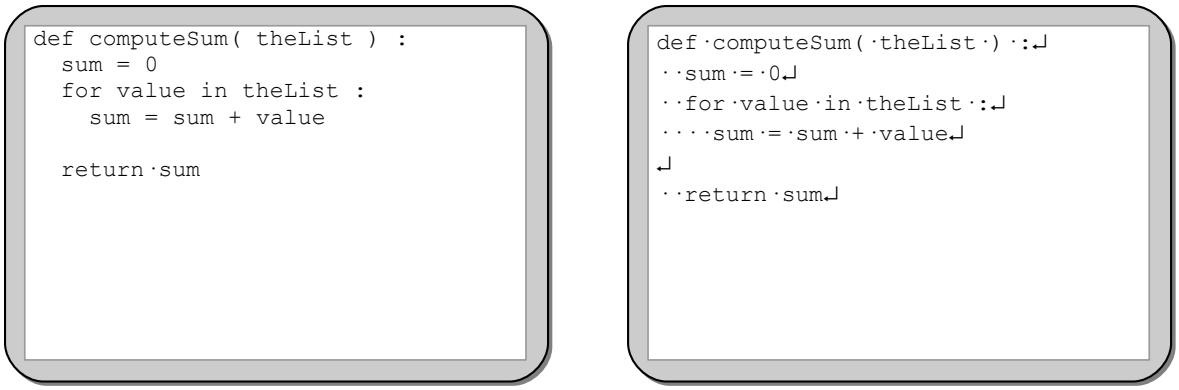

Figure 9.13: The abstract view of a text document as used by most text editors (top) and the same with special characters inserted for the blank spaces and newlines (bottom).

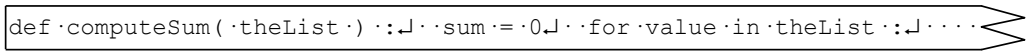

Figure 9.14: The file or stream view of the text document from Figure 9.13.

Text Cursor

Text editors also use the concept of a cursor to mark the current position within the document. The cursor position is identified by row and an offset or character position within the given row. All basic insertion and deletion operations are performed relative to the cursor. Note that for this example we consider the cursor to be the keyboard or text cursor and not the cursor related to the use of the mouse.

The cursor can be moved and positioned anywhere within the document, but only to a spot currently containing a character. Thus you cannot randomly place characters within a document as if you were writing on a piece of paper. To position

text within a document, you must fill the document with blank lines and spaces as necessary. The minimum cursor movements available with most text editors include:

- Vertical movement in which the cursor is moved one or more lines up or down from the current line. When the cursor is moved in the vertical direction, it typically maintains the same character position on the line to which it was moved. If this line is shorter than the one from which the cursor was moved, the cursor is typically positioned at the end of the line.
- Horizontal movement along the current line in which the cursor moves either forward or backward. When the cursor reaches the beginning of the line, it wraps backward to the end of the previous line; when it reaches the end of the line, it wraps forward to the beginning of the next line. Thus, the cursor moves about the text document as if moving within a stream of text like that stored in a text file.
- Movement to the document and line extremes. The cursor can typically be moved to the front or back of the current line or to the beginning or end of the document. In the latter case, the cursor is typically moved to the front of the last row in the document.

Inserting Text

Most text editors use an entry mode of either "insert," in which new characters are inserted into the text, or "overwrite," in which new characters replace existing characters. In either mode, characters are inserted into the document at the cursor position. When using insert mode, all characters on the same line from the cursor to the end of the line are shifted down and the new character is inserted at the current position. This differs from overwrite mode, in which new characters simply replace or overwrite the character at the cursor position. In both cases, the cursor is then shifted one position to the right on the same line.

The newline characters that are used to indicate the end of a line can be treated like any other character. If new characters are inserted immediately before a newline character, the line is automatically extended. When inserting text after a newline character, you are technically inserting the text at the beginning of the next line in front of the first character on that line. When a newline character is inserted, a line break is created at the insertion point, resulting in the current line being split into two lines. The new line is always inserted immediately following the line being split.

Deleting Text

Most text editors provide both a delete and a rub-out operation for removing individual characters from a document. These are typically mapped to the Delete and Backspace keys, respectively. Both operations delete a character and shift all following characters on the same line forward one position. The difference between

the two operations is which character is deleted and what happens to the cursor afterward. In the delete operation, the character at the cursor position is removed and the cursor remains at the same position. The rub-out operation, on the other hand, removes the character preceding the cursor and then moves the cursor one position to the left.

When a newline character is deleted, the current line and the one immediately following are merged. The newline character at the end of the last line in the document cannot be deleted.

Most text editors provide other text manipulation operations such as truncating a line from the cursor position to the end of the line. This operation typically does not remove the newline character at the end of the line, but instead simply deletes the non-newline characters from the cursor position to the end of the line. Some editors may also provide an operation to delete the entire line on which the cursor is positioned.

9.5.2 The Edit Buffer ADT

We are now ready to define an Edit Buffer ADT that can be used with a simple text editor. Our definition is based on the description of text editors presented in the previous section. In order to keep the example simple, we limit the operations available with our ADT.

Define EditBuffer ADT

An *edit buffer* is a text buffer that can be used with a text editor for storing and manipulating the text as it is being edited. The buffer stores plain text with no special formatting or markup codes, other than the common ASCII special characters like tabs and newlines. Individual lines are terminated with a newline character. The current cursor position is maintained and all operations are performed relative to this position. The cursor can only be positioned where a character currently exists. The cursor can never be moved outside the bounds of the document. The buffer can insert characters in either insert or overwrite entry mode.

- `EditBuffer()`: Creates a new and empty edit buffer. An empty buffer always contains a single blank line and the cursor is placed at the first position of this blank line. The entry mode is set to insert.
- `numLines()`: Returns the number of lines in the text buffer.
- `numChars()`: Returns the length of the current line that includes the newline character.
- `lineIndex()`: Returns the line index of the line containing the cursor. The first line has an index of 0.
- `columnIndex()`: Returns the column index of the cursor within the current line. The first position in each line has an index of 0.

- `setEntryMode( insert )`: Sets the entry mode to either insert or overwrite based on the value of the boolean argument `insert`.
- `toggleEntryMode()`: Toggles the entry mode to either insert or overwrite based on the current mode.
- `inInsertMode()`: Returns true if the current entry mode is set to insert and false otherwise.
- `getChar()`: Returns the character at the current cursor position.
- `getLine()`: Returns the contents of the current line as a string that includes the newline character.
- `moveUp( num )`: Moves the cursor up `num` lines. The cursor is kept at the same character position unless the new line is shorter, in which case the cursor is placed at the end of the new line. The `num` is negative, and the cursor position is not changed.
- `moveDown( num )`: The same as `moveUp()` except the cursor is moved down.
- `moveDocHome()`: Moves the cursor to the document's home position, which is the first line and first character position in that line.
- `moveDocEnd()`: Moves the cursor to the document's end position, which is the last line and first character position in that line.
- `moveLeft()`: Moves the cursor to the left one position. The cursor is wrapped to the end of the previous line if it is currently at the front of a line.
- `moveRight()`: Moves the cursor to the right one position. The cursor is wrapped to the beginning of the next line if it is currently positioned at the end of a line.
- `moveLineHome()`: Moves the cursor to the front of the current line at the first character position.
- `moveLineEnd()`: Moves the cursor to the end of the current line.
- `breakLine()`: Starts a new line at the cursor position. A newline character is inserted at the current position and all characters following are moved to a new line. The new line is inserted immediately following the current line and the cursor is adjusted to be at the first position of the new line.
- `deleteLine()`: Removes the entire line containing the cursor. The cursor is then moved to the front of the next line. If the line being deleted is the last line, the cursor is moved to the front of the previous line.
- `truncateLine()`: Removes all of the characters at the end of the current line starting at the cursor position. The newline character is not removed and the cursor is left at the end of the current line.

- `addChar( char )`: Inserts the given character into the buffer at the current position. If the current entry mode is insert, the character is inserted and the following characters on that line are shifted down; in overwrite mode, the character at the current position is replaced. If the cursor is currently at a newline character and the entry mode is overwrite, the new character is inserted at the end of the line. The cursor is advanced one position. If `char` is the newline character, then a line break occurs, which is the same as calling `breakLine()`.

- `deleteChar()`: Removes the character at the current position and leaves the cursor at the same position.

- `ruboutChar()`: Removes the character preceding the current position and moves the cursor left one position. If the cursor is currently at the front of the line, the newline character on the preceding line is removed and the current line and the preceding line are merged into a single line.

- `deleteAll()`: Deletes the entire contents of the buffer and resets it to the same state as in the constructor.

The definition of the ADT includes the operations sufficient for use with a simple text editor. All of the information needed by an editor to display or manipulate the text is available through the defined operations.

9.5.3 Implementation

Many different data structures can be used to implement the Edit Buffer ADT. The choice can depend on the type of editor with which the buffer will be used. For example, when editing a field within a dialog box or web page, the editing capabilities and the buffer size are usually limited. But a full-blown text edit like *JEdit* or *Notepad*™ requires a dynamic buffer, which can grow and shrink in size.

For our implementation, we are going to use a doubly linked list of Python lists, which provides dynamic capabilities in terms of growing and shrinking the buffer as needed while providing quick line insertions and deletions. The individual lines of text will be stored in the nodes of the doubly linked list while the individual characters within the lines will be stored in Python lists. Figure 9.15 illustrates a sample buffer using this organization.

The use of a linked list provides fast line insertions and deletions as text is added and removed. The doubly linked version allows for quick movement within the buffer both forward and backward as the user navigates among the lines of text. The choice of the vector to store the individual characters allows for quick modifications as characters are added and deleted. Existing characters can be directly modified or deleted and new characters inserted without the overhead required when using strings. Although the Python list does require resizing, it's typical for individual lines of text on average to be relatively short.

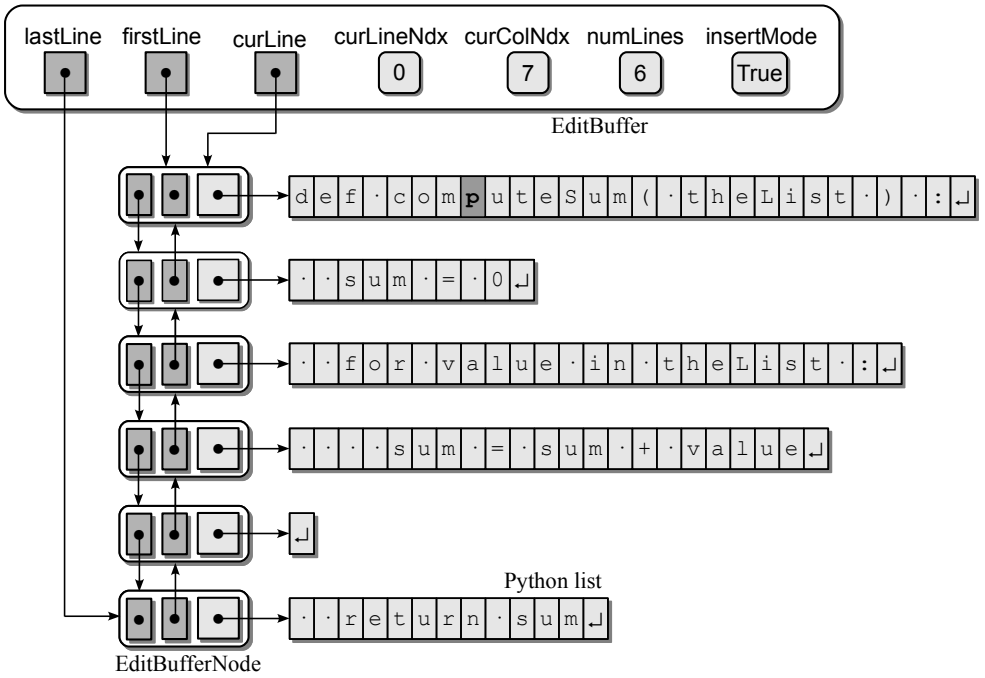

Figure 9.15: The doubly linked list of vectors used to implement the Edit Buffer ADT representing the text document from Figure 9.13.

A partial implementation of the `EditBuffer` class is provided in Listing 9.11. The implementation of some operations and helper methods, many of which can be implemented in a similar fashion to those provided, are left as exercises. Several of the operations simply provide status information and are straightforward in their implementation. These are provided in the listing without commentary. Others are more involved and require a brief discussion, which is provided the following sections.

Listing 9.11 The `editbuffer.py` module.

```
1  # Implements the Edit Buffer ADT using a doubly linked list of vectors.
2  class EditBuffer :
3     # Constructs an edit buffer containing one empty line of text.
4    def __init__( self ):
5      self._firstLine = _EditBufferNode( ['\n'] )
6      self._lastLine = self._firstLine
7      self._curLine = self._firstLine
8      self._curLineNdx = 0
9      self._curColNdx = 0
10     self._numLines = 1
11     self._insertMode = True
12
13    # Returns the number of lines in the buffer.
14   def numLines( self ):
15     return self._numLines
```

(Listing Continued)

Listing 9.11 Continued ...

```
16
17    # Returns the number of characters in the current line.
18    def numChars( self ):
19      return len( self._curLine.text )
20
21    # Returns the index of the current row (first row has index 0).
22    def lineIndex( self ):
23      return self._curRowNdx
24
25    # Returns the index of the current column (first col has index 0).
26    def columnIndex( self ):
27      return self._curColNdx
28
29    # Sets the entry mode based on the boolean value insert.
30    def setEntryMode( self, insert ):
31      self._insertMode = insert
32
33    # Toggles the entry mode between insert and overwrite.
34    def toggleEntryMode( self ):
35      self._insertMode = not self._insertMode
36
37    # Returns true if the current entry mode is insert.
38    def inInsertMode( self ):
39      return self._insertMode == True
40
41    # Returns the character at the current cursor position.
42    def getChar( self ):
43      return self._curLine.text[ self._curColNdx ]
44
45    # Returns the current line as a string.
46    def getLine( self ):
47      lineStr = ""
48      for char in self._curLine.text :
49        lineStr .= char
50      return lineStr
51
52    # Moves the cursor up num lines.
53    def moveUp( self, nlines ):
54      if nlines <= 0 :
55        return
56      elif self._curLineNdx - nlines < 0 :
57        nlines = _curLineNdx
58
59      for i in range( nlines ) :
60        self._curLine = self._curLine.prev
61
62      self._curLineNdx -= nlines
63      if self._curColNdx >= self.numChars() :
64        self.moveLineEnd()
65
66    # Moves the cursor left one position.
67    def moveLeft( self ):
68      if self._curColNdx == 0 :
69        if self._curRowNdx > 0 :
```

```
70          self.moveUp( 1 )
71          self.moveLineEnd()
72        else :
73          self._curColNdx -= 1
74
75      # Moves the cursor to the front of the current line.
76      def moveLineHome( self ) :
77        self._curColNdx = 0
78
79      # Moves the cursor to the end of the current line.
80      def moveLineEnd( self ):
81        self._curColNdx = self.numChars() - 1
82
83      # Starts a new line at the cursor position.
84      def breakLine( self ):
85          # Save the text following the cursor position.
86        nlContents = self._curLine.text[self._curColNdx:]
87          # Insert newline character and truncate the line.
88        del self._curLine.text[self.__curColNdx:]
89        self._curLine.text.append( '\n' )
90          # Insert the new line and increment the line counter.
91        self._insertNode( self._curLine, nlContents )
92          # Move the cursor.
93        self._curLine = newLine
94        self._curLineNdx += 1
95        self._curColNdx = 0
96
97      # Inserts the given character at the current cursor position.
98      def addChar( self, char ):
99        if char == '\n' :
100         self.breakLine()
101       else :
102         ndx = self._curColNdx
103         if self.inInsertMode() :
104           self._curLine.text.insert( ndx, char )
105         else :
106           if self.getChar() == '\n' :
107             self._curLine.text.insert( ndx, char )
108           else :
109             self._curLine.text[ ndx ] = char
110         self._curColNdx += 1
111
112     # Removes the character preceding the cursor; cursor remains fixed.
113     def deleteChar( self ):
114       if self.getChar() != '\n' :
115         self._curLine.text.pop( self._curColNdx )
116       else :
117         if self._curLine is self._lastLine :
118           return
119         else :
120           nextLine = self._curLine.next
121           self._curLine.text.pop()
122           self._curLine.text.extend( nextLine.text )
123           self._removeNode( nextLine )
124
```

(Listing Continued)

Listing 9.11 Continued ...

```
125 # Defines a private storage class for creating the list nodes.
126 class _EditBufferNode :
127   def __init__( self, text ):
128     self.text = text
129     self.prev = None
130     self.next = None
```

Constructor

The constructor is defined in lines 5–11 of Listing 9.11. The `_firstLine` and `_lastLine` reference variables act as the head and tail pointers for the doubly linked list. The number of lines in the buffer is maintained using `_numLines` and the current entry mode is specified by the boolean `_insertMode`. The current cursor position within the buffer must be tracked, which is done using `_curLine` and `_curColNdx`. The former is a linked list external reference since it points to a node in the doubly linked list and the latter is an index value referencing a position within the vector for the current line. The line number on which the cursor is currently positioned is also maintained since it will be needed by the `getLine()` method.

The ADT definition calls for a newly created buffer to be initially set to empty. While an empty buffer contains no visible characters, it will contain a single blank line that consists of a lone newline character. A node is created and initialized with a vector containing a single newline character. The nodes in the doubly linked list will be instances of the `EditBufferNode` class, the definition of which is provided in lines 126–130. The cursor is initially placed at the home position, which in an empty document corresponds to the newline character in the newly created blank line. Finally, the initial entry mode is set to insert mode.

Cursor Movement

A number of operations in the Edit Buffer ADT handle movement of the cursor, which allows for character manipulation at different points in the buffer. These routines modify the current cursor position by appropriately adjusting the `_curLine`, `_curLineNdx`, and `_curColNdx` fields.

Vertical movement of the cursor is handled by four methods: `moveDocHome()`, `moveDocEnd()`, `moveDown()`, and `moveUp()`. The cursor can be moved up or down one line or multiple lines at a time but it can never be moved outside the valid range. Implementation of the `moveUp()` method is provided in lines 53–64 of Listing 9.11. By definition, the number of lines the cursor is to be moved must be positive, otherwise the cursor is not moved. In addition, the cursor cannot be moved further up than the first line in the document. These conditions are evaluated by the first two logical expressions. If `nlines` would result in the cursor being moved beyond the first line, `nlines` has to be adjusted to limit the movement to the first line. Next, a `for` loop is used to move the `_curLine` reference up the indicated number of

lines followed by _curLineNdx being adjusted appropriately. Finally, we determine if the horizontal position of the cursor must be adjusted. If the line to which the cursor has been moved is shorter than the previous line, then the cursor must be positioned at the end of the new line.

Horizontal movement of the cursor is managed by four methods: moveLeft(), moveRight(), moveLineHome(), and moveLineEnd(). Moving the cursor to the beginning or end of a line simply requires modifying the value of _curColNdx. Moving the cursor horizontally one space can be as simple as adjusting _curColNdx by one, except when the cursor is at the beginning or ending of the line when moving left or right, respectively.

Implementation of the moveLeft() method is provided in lines 67–73. When moving left one space and the cursor is at the beginning of the line, the cursor must be moved to the end of the previous line. This can be accomplished by moving the cursor up one line and then to the end of that line. Of course this is only done if the current line is not the first line in the buffer. Moving right one space can be implemented in a similar fashion. The implementation of the remaining cursor movement methods is left as an exercise.

Modifying the Buffer

Moving the cursor is somewhat easier than modifying the contents of the buffer. Most modifications must consider the newline character, which may require splitting a line or merging two lines depending on the specific operation. We begin our look at buffer modifications with the operation of adding a single character.

The action taken when adding a character in the addChar() method, shown in lines 98–110 of Listing 9.11, depends on the entry mode and what character is being added. In either mode, when adding a newline character, the result is the same as breaking a line at the current position and thus this can be done with the breakLine() method.

When the current entry mode is set to insert, the character is always inserted into the vector at the current cursor position, which causes the following characters on the same line to shift down one place. When inserting a character at the end of the line, we are actually inserting it immediately preceding the newline character. Thus, the newline character will always be the last character in the vector. The action taken when adding a character in overwrite mode depends on the position of the cursor. When the cursor is at the end of line, characters are inserted in the same fashion as if the entry mode was set to insert. The newline character is never overwritten. Otherwise, the new character simply replaces the character at the cursor position. After adding a new character, the cursor is always moved to the right one place. This can be done by simply adding one to _curColNdx since the cursor never moves to the next line during an add operation unless a newline character is being added. The latter case is handled by the breakLine() method.

Deleting a character from the buffer is straightforward if the character is not the newline character. It only requires removing or popping the character at the current cursor position within the vector containing the current line. The implementation of the deleteChar() method is provided in lines 113–123 of Listing 9.11.

Deleting a newline character requires the merging of two lines, the current line with the following one. Of course the newline character of the last line in the buffer cannot be deleted. Thus, this condition must first be checked before merging the two lines. Merging the two lines requires several steps. First, the newline character on the current line must be removed and the current line extended by appending to it the contents of the next line. Then, the node and buffer entry for the next line must be removed from the doubly linked list. This step is performed using the `_removeNode()` helper method, the implementation of which is left as an exercise. The helper method also handles the final step in merging the two lines, which is to decrement the line count by one.

Splitting a Line

We conclude our discussion with the `breakLine()` method, the implementation of which is provided in lines 84–95 of Listing 9.11. This operation splits a line into two lines at the current cursor position with the character at the cursor position becoming part of the new line. Figure 9.16 illustrates the sample buffer after splitting the third line at the letter "i" of the word "in."

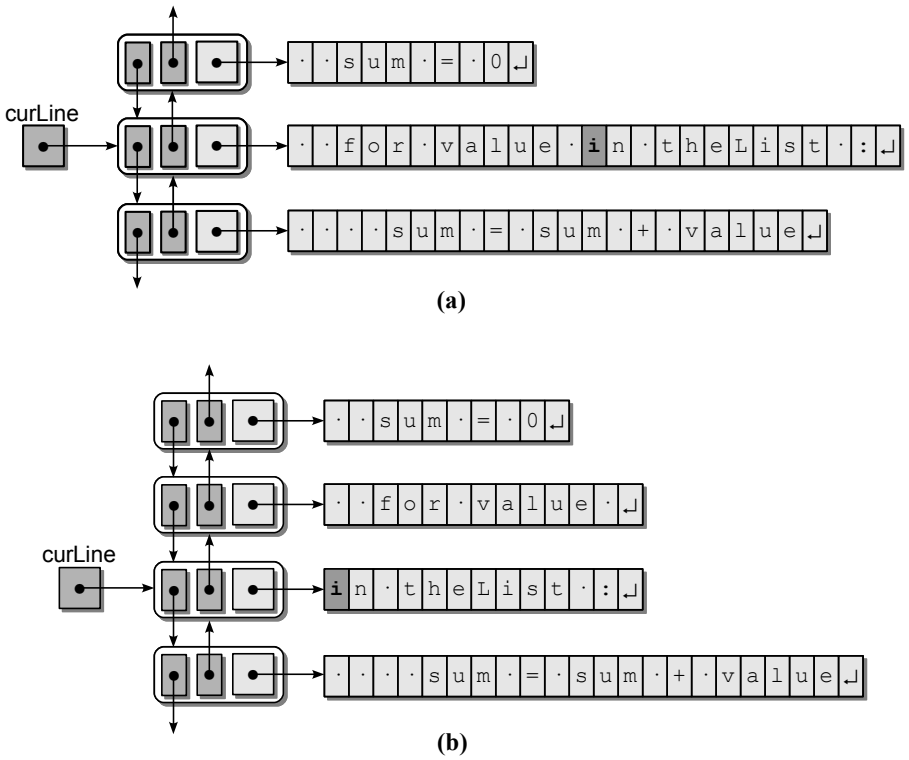

Figure 9.16: Breaking a line of text at the cursor position: (a) the sample buffer before the split and (b) the result after the split with the cursor position adjusted.

The contents at the end of the current line vector starting at the cursor position will form the new line. We first extract and save this text by creating a slice from the current line. The part of the vector from which we created the slice is then deleted and a newline character is appended. We use the `_insertNode()` helper method to then insert a new line into the buffer. The helper method inserts a new node following the one pointed to by `_curLine` and containing the supplied contents. The last step required in breaking the current line is to move the cursor to the front of the new line.

Exercises

9.1 Evaluate the four basic linked list operations for a sorted doubly linked list and indicate the worst case time-complexities of each.

9.2 Evaluate the worst case time-complexity for the search operation on a doubly linked list using a probe pointer.

9.3 Evaluate the worst case time-complexity of the Sparse Matrix ADT implemented in Programming Project 9.7.

9.4 Provide the code to implement the four basic operations — traversal, search, insertion, deletion — for an unsorted doubly linked list.

9.5 Provide the code to delete a node from a sorted doubly linked list.

9.6 Provide the code to delete a node from a sorted singly linked circular list.

Programming Projects

9.1 Complete the implementation of the `EditBuffer` class from Section 9.5.

9.2 Modify the `moveLeft()` and `moveRight()` methods to accept an integer to indicate the number of spaces to move in the respective direction.

9.3 Modify the `EditBuffer` class to include the following operations:

- `getPage( first, last )`: Returns a vector of strings consisting of the rows [first...last], which can be used for displaying a page of text.
- `insertString( str )`: Inserts a string of text within the current line starting at the current cursor position.
- `moveTo( lineNdx, colNdx )`: Moves the cursor to the indicated position within the buffer. If `lineNdx` is out of range, no action is taken. If `colNdx` is larger than the current line, then the cursor is positioned at the end of that line.

- `searchFor( str )`: Searches the buffer and returns a tuple containing the (line, col) position of the first occurrence of the given search string. `None` is returned if the buffer does not contain the search string.
- `searchForAll( str )`: The same as `searchFor()` but returns a vector of tuples indicating all occurrences of the given search string.

9.4 Modify the `EditBuffer` class to include the following file I/O operations for saving and loading the buffer:

- `save( filename )`: Saves the text in the buffer to the text file `filename`.
- `load( filename )`: Loads the text file named `filename` into the buffer. If the buffer is not empty, the original contents are deleted before the file is loaded. The given filename must refer to an existing and valid text file.

9.5 Create a line editor application that uses the `EditBuffer` class and allows a user to edit a text document in the old "line editing" style.

9.6 Define and implement a MultiChain ADT for storing and accessing student records as described in Section 9.3.1.

9.7 Implement the Sparse Matrix ADT using the multi-linked lists as described in Section 9.3.2.

9.8 Consider the Vector ADT from Programming Project 2.1:

(a) Provide a new implementation that uses a doubly linked list and a probe reference for locating a specific element.

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

(c) What are the advantages and disadvantages of using a doubly linked list to implement the Vector ADT? Compare this implementation to that of using a singly linked list.

9.9 Consider the Map ADT from Section 3.2:

(a) Provide a new implementation that uses a sorted doubly linked list and includes a probe reference for the search operations.

(b) Modify your `Map` class to include an iterator for use with a `for` loop.

9.10 A MultiMap ADT is similar to the Map ADT but it uses two keys that map to a single data item instead of a single key as used with the Map ADT. Define and implement a MultiMap ADT.