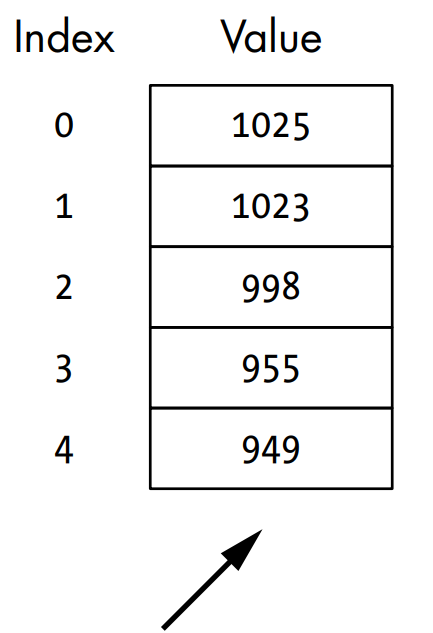
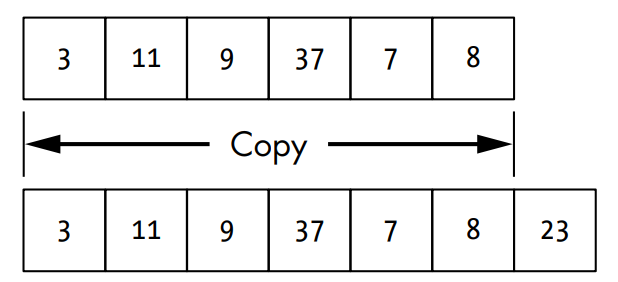
**3 D Y N A M I C D A T A S T R U C T U R E S**  
  
This chapter introduces dynamic data structures, which alter their structure as the data changes. These structural adaptations may include growing the size of the data structure on demand, creating dynamic, mutable linkings between different values, and more. Dynamic data structures lie at the heart of almost every computer program in the world and are the foundation of some of the most exciting, interesting, and powerful algorithms in computer science.

The basic data structures introduced in the previous chapters are like parking lots—they give us a place to store information, but don’t provide much in the way of adaptation. Sure, we can sort the values in an array (or cars in our parking lot) and use that structure to make binary search efficient. But we’re just changing the ordering of the data within the array. The data structure itself is neither changing nor responding to changes in the data. If we later change the data in a sorted array, say by modifying the value of an element, we need to re-sort the array. Worse yet, when we need to change the data structure itself—by growing or shrinking the array, for example—simple static data structures don’t provide any help.  
  
This chapter compares the static data structure introduced in Chapter 2, the array, with a simple dynamic data structure, the linked list, to demonstrate the advantages of the latter. In some respects, these two data structures are similar: they both allow programmers to store and access multiple values through a single reference, either the array or the head of the linked list. However, arrays have a structure fixed at time of creation, like rows of parking spaces. In contrast, linked lists can grow throughout the program’s memory. They behave more like a lengthening or shrinking line of people, allowing for additions and removals. Understanding these differences provides a foundation for understanding the more advanced data structures that we will visit in the rest of this book.  
  
**The Limitations of Arrays**  
  
While arrays are excellent data structures for storing multiple values, they suffer from one important limitation: their size and layout in memory are fixed at the time of creation. If we want to store more values than can fit in our array, we need to create a new, larger array and copy over the data from the older array. This fixed-size memory is acceptable for when we have an unmoving upper bound on the number of items we need to store. If we have sufficient bins to fit our data, we can set individual entries all day long without worrying about the array’s static layout in memory. However, many applications require dynamic data structures that can grow and change with our program.  
  
To meet this need for dynamic data structures, many modern programming languages offer dynamic “arrays” that grow and shrink as you add elements. However, these are actually wrappers around static arrays or other data structures that hide the complexities and costs associated with their dynamic nature. While this is convenient for the programmer, it can lead to hidden inefficiencies. When we add elements past the end of the array, the program still needs to increase the memory used. It just does so behind the scenes. To understand why dynamic data structures are so important, we need to discuss their limitations. In this book, we’ll use the term array to refer to a simple static array.  
  
To illustrate the array’s restrictions, imagine that you spend an entire week mastering the latest retro video game phenomenon, Space Frogger 2000. You smile with glee every time the main screen displays your five top scores. These monumental achievements represent hours of sweat, tears, shouting, and more tears. However, the very next day, your (soon to be former) best friend visits and goes on to beat your highest score five times in a row. Once you kick the traitorous ex-friend out of the house, you return to your game and gaze at the new top scores, shown in Figure 3-1, and cry out, “Why couldn’t the game store more scores? Would it really be so hard to keep a top ten list, or at least add one more to the very end?”  
  
  
*Your best score would go here.*  
Figure 3-1: A five-element array holding high scores for a video game. None, alas, are yours.

This is one of the fundamental limitations of any fixed-size data structure and its fixed layout in memory—it can’t grow with the data. As we see below, this restriction makes some common operations expensive. More practically, imagine the limitations of a word processor with space for only a fixed number of characters, a spreadsheet with a fixed number of rows, a photo storage program that can store a limited number of pictures, or a coffee journal limited to only a thousand entries.  
  
Since the size of an array is fixed at the time of creation, if we want to extend the array to store more data, we must create a new, larger block of memory. Consider the simplest case of adding a single element to the end of an array. Because an array is a single, fixed-size block of memory, we can’t just shove another value into the end. There might be another variable already occupying that space in the memory. Rather than risk overwriting that variable’s value, we have to allocate a new (bigger) block of memory, copy all the values of the original array into the new block, and write the new value at the end. That’s a lot of overhead for a single addition, as illustrated in Figure 3-2.  
  
  
  
Figure 3-2: Adding an element to the end of a full array requires allocating new space and copying over the elements.  
Think of an array as one of those heated hotel buffet counters with a fixed number of slots. It’s easy to pop out the empty tray of scrambled eggs and add a new one in its place. But you can’t just stick a new tray onto the end. There’s no room for it. If the chef decides to add pancakes to the menu, something else has to go.  
  
If you know you’ll need to insert a lot of new values, you might spread the cost out over multiple updates, amortizing the cost. You might adopt a strategy like array doubling, in which the size of an array doubles whenever it is expanded. For example, if we try to add a 129th element to our array of size 128, we first allocate a new array of size 256 and copy over the original 128 elements. This allows us to continue growing the array for a while before we next need to worry about allocating new space. However, the cost is potentially wasted space. If we only need 129 elements total, we have overallocated by 127.  
  
Array doubling provides a reasonable balance between expensive array copies and wasted memory. As the array grows, the doublings become less and less frequent. At the same time, by doubling the array when it is full, we are guaranteed to waste less than half the space. However, even with this balanced approach, we can clearly see the cost of using a fixed-size array in terms of both copying cost and memory usage.

-------------------------------------------------------------------------------

ArrayDouble(Array: old\_array):

Integer: length = length of old\_array

Array: new\_array = empty array of size length \* 2

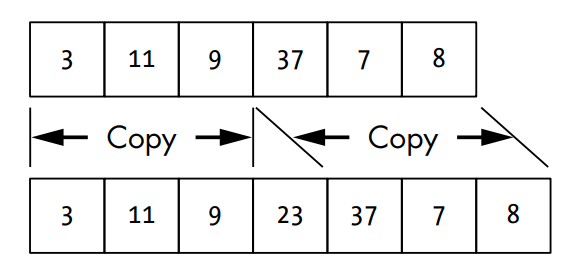
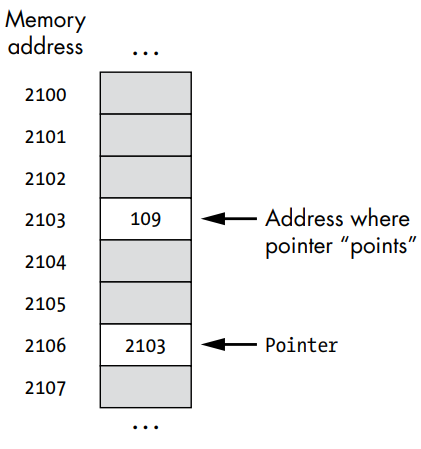
Integer: j = 0

WHILE j < length:

new\_array[j] = old\_array[j]

j = j + 1

return new\_arra

-------------------------------------------------------------------------------  
  
The code for array doubling starts by allocating a new array twice the size of the current array. A single WHILE loop iterates over the elements in the current array, copying their values into the new array. The new array is returned.  
  
Imagine applying this strategy to shelf space. We establish a bookstore, Data Structures and More, in a location and install a humble five shelves. Opening day sees surprising demand and requests for more variety: we need to expand our inventory. Panicked, we move to a new location with 10 shelves and migrate the books. The demand has temporarily been met. Since the lack of a comprehensive data structure store is a clear gap in the retail books market, our store is a runaway success, and demand continues to grow and grow. We might upgrade the store a few more times to locations with 20, 40, then 80 shelves. Each time we pay a cost to secure a new location and migrate the books.  
  
The fixed location of the array’s values in memory provides another limitation. We cannot easily insert additional items in the middle of an array. Even if there are enough empty spaces at the end of our original array to accommodate a new element, and therefore we don’t need to move the whole array to a new memory block, we still need to shift each existing element over one by one to make a space for the new value in the middle. Unlike a shelf of books, we can’t just shove all the elements over at once with a single good Data Structures the Fun Way (Early Access) © 2022 by Jeremy Kubica Dynamic Data Structures    29 push. If we had 10,000 elements and wanted to add something in the second position, we’d need to move 9,999 elements over. That’s a lot of effort to insert a single element.  
  
The problems compound when we try to insert new values into the middle of an array that is already full. Not only do we have to allocate a new block and copy the old values, but we need to shift the values after the new value down one position to clear a space for our new value. For example, suppose we wanted to insert the value 23 as the fourth element of an existing array of six elements, as illustrated in Figure 3-3.  
  
  
  
*Figure 3-3: Adding an element to the middle of a full array requires allocating new space, copying the elements before the insertion, and copying the values after the insertion to shifted locations.*  
  
To address the shortfalls of arrays, we need to move to more flexible data structures that can grow as new data is added: dynamic data structures. Before we jump into the details, let’s introduce pointers, the variable type that’s key to reconfiguring and growing data structures.  
  
**Pointers and References**  
  
One variable type stands above its peers in terms of both its sheer power and its ability to confuse new programmers: pointers. A pointer is a variable that stores only the addresses in the computer’s memory. The pointer therefore points to a second location in memory where the actual data is stored, as shown in Figure 3-4.  
  
  
  
*Figure 3-4: A pointer indicating an address in the computer’s memory*

The astute reader may ask, “What is the purpose of a variable that simply points to another location in memory? I thought the variable’s name already served this function. Why not store your data in the variable like a normal person? Why do you always have to make things so complicated?” Don’t listen to the skeptics. Pointers are the essential ingredient in dynamic data structures, as we’ll see shortly.  
Suppose we are working on a major architectural project at the office and have assembled a folder of example drawings to share with our team. Soon the project folder contains numerous floorplans, cost estimates, and artistic renderings. Rather than make a copy of the hefty file and leave it out in the open, we leave a note telling our collaborators to find the file in the third-floor records room, filling cabinet #3, the second drawer down, fifth folder. This note plays the role of a pointer. It doesn’t detail all the information that’s in the file, but rather allows our colleagues to find and retrieve the information. More importantly, we can share this single “address” with each of our coworkers without making a full copy of the file for them. They can each use this information to look up and modify the folder when needed. We could even leave an individual sticky note on each team member’s desk, providing 10 variables pointing to the same information.  
In addition to storing the location of a block of memory, pointers can take on a null value (denoted as None, Nil, or 0 in some programming languages). The null value simply denotes that the pointer isn’t currently pointing to a valid memory location. In other words, it indicates that the pointer doesn’t actually point to anything yet.  
Different programming languages provide different mechanisms to accomplish the task of pointers, and not all of them provide the raw memory address to the programmer. Lower-level languages like C and C++ give you raw pointers and allow you to directly access the memory location they store. Other programming languages, such as Python, use references, which use syntax like that of a normal variable while still allowing you to reference another variable. These different variations come with different behaviors and usages (dereferencing, pointer math, the form of null values, and so forth). For the sake of simplicity, throughout this book we will use the term pointer to cover all variables implemented by pointers, references, or indices into preallocated blocks of memory. We won’t worry about the complicated syntax needed to access the blocks of memory (which has caused more than a few programming enthusiasts to break down in tears). We will also use the final data’s type (instead of the more generic type pointer) when defining a pointer variable in pseudocode. The key concept for our purposes is that pointers provide a mechanism for linking to a block of memory as featured in our first dynamic data structure: the linked list.  
  
**Linked Lists**   
  
Linked lists are the simplest example of a dynamic data structure and are a close cousin to arrays. Like arrays, they are a data structure for storing multiple values. Unlike arrays, linked lists are composed of a chain of nodes linked together by pointers. A basic node in a linked list is a composite data structure containing two parts: a value (of any type) and a pointer to the next node in the list:   
  
  
LinkedListNode {

Type: value

LinkedListNode: next

}  
  
 We can picture a linked list as a series of linked bins, as in Figure 3-5. Each bin stores a single value and contains a pointer to the next bin in the series.

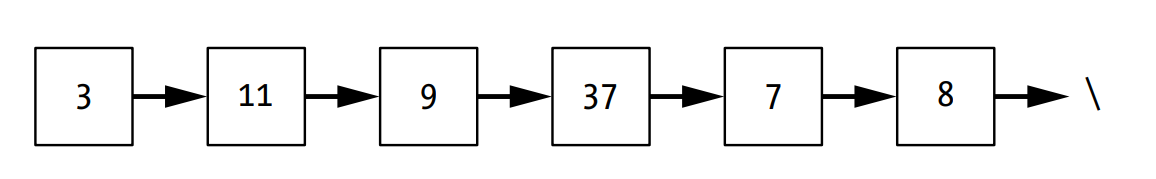


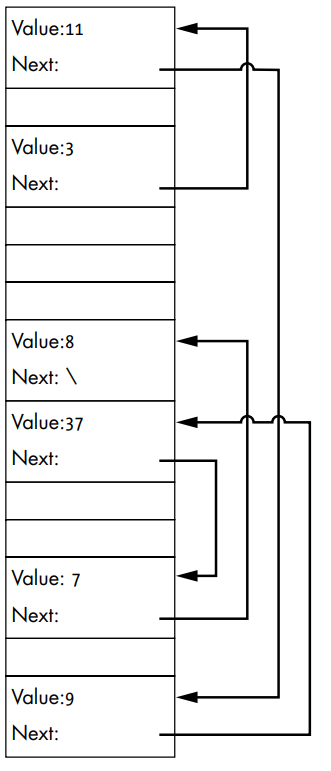
Figure 3-5: A linked list represented as a series of nodes that are linked by pointers  
  
The slash at the end of the list represents a null value and indicates the end of the list. Effectively we are saying that the last node’s next pointer does not point to a valid node. A linked list is like a long line of people waiting at our favorite coffee shop. People rarely know their absolute position in the line—“I’m on the fifty-third floor tile back from the counter.” They pay attention to their relative order, namely the single person before them, which we store in a pointer. Even if the line winds throughout the store (and its parking lot) in complex loops, we can still reconstruct the order by asking each person who is immediately in front of them. We can traverse the line toward the counter by asking each person who is before them.   
  
Because they include pointers as well as values, linked lists require more memory than arrays to store the same items. If we have an array of size K, storing values of N bytes each, we only need K × N bytes. In contrast, if each pointer requires another M bytes, our data structure now has a cost of K × (M + N) bytes. Unless the size of the pointers is much smaller than the size of our values, the overhead is significant. However, the increased memory usage is often worth it for the increased flexibility the pointers provide.  
  
While textbooks often represent linked lists as neat, orderly structures (as shown in Figure 3-5 or implied in our line-of-humans example), our list can actually be scattered throughout the program’s memory. As illustrated in Figure 3-6, the list’s nodes are linked only via their pointers. This is the real power of pointers and dynamic data structures. We aren’t constrained to keep the entire list in a single contiguous block of memory. We’re free to grab space for new nodes wherever space happens to exist.  
  


Figure 3-6: A linked list in the computer’s memory. Nodes are not necessarily adjacent to each other.

Programs typically store linked lists by keeping a single pointer to the front, or head, of the linked list. The program can then access any element in the list by starting at the head and iterating through the nodes via the pointers:

LinkedListLookUpLinkedListNode: head, Integer: element\_number):(

1 LinkedListNode: current = head   
Integer: count = 0   
  
2 WHILE count < element\_number AND current != null:

current = current.next   
count = count + 1

return current

The code starts at the head of the list 1. We maintain a second variable count to track the index of the current node. The WHILE loop then iterates through each node in the list until it has found the correct number, count == element\_number, or run off the end of the list, current == null 2. In either case, the code can return current. If the loop terminates due to running off the edge of the list, then the index is not in the list and the code returns null.  
  
For example, if we wanted to access the fourth element of a linked list, the program would access first the head, then the second, third, and fourth elements in order to find the correct memory location. Figure 3-7 shows this process, where the node with value 3 is the head of the list.

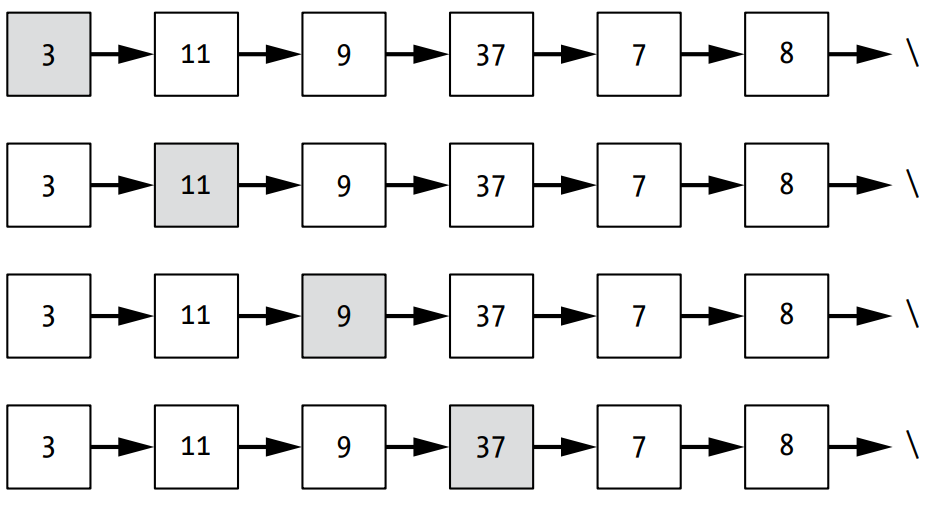


Figure 3-7: Traversing a linked list requires moving from one node to the next along the chain of pointers.

It’s worth noting, however, that there’s a tradeoff: linked lists have a higher computing overhead than arrays. When accessing an element in an array, we just compute a single offset and look up the correct address of memory. The array access only takes one mathematical computation and one memory lookup regardless of which index we choose. Linked lists require us to iterate from the beginning of the list until we get to the element of interest. For longer lists, the lack of direct access can add significant overhead.

At first glance, this restricted access pattern is a strike against the linked list. We’ve dramatically increased the cost of looking up an arbitrary element! Consider what this means for binary search. A single lookup requires iterating over many of the elements, removing the advantage of a sorted list.

Yet despite these costs, linked lists can become real assets in practical programs. Data structures almost always involve tradeoffs among complexity, efficiency, and usage patterns. The very behaviors that disqualify a data structure for one use might make it the perfect choice to support other algorithms. Understanding these tradeoffs is the key to effectively combining algorithms and data structures. In the case of linked lists, the tradeoff for increased overhead in accessing elements is a significant increase in the flexibility of the overall data structure, as we will see in the next section.  
  
**Operations on Linked Lists**

While some lament the chaotic sprawl of the linked list compared to the aesthetic beauty of the compact array, it is exactly this ability to link across different blocks of memory that makes the data structure so powerful, allowing us to dynamically rearrange the data structure. Let’s compare inserting a new value to an array with adding a value to a linked list.

**Inserting into a Linked List**

As we’ve seen, inserting a new element into an array may require us to allocate a new (bigger) block of memory and copy all the values of the original array into the new block. Further, the insertion itself may require us to traverse the array and shift elements over.

The linked list, on the other hand, doesn’t need to stay in a single contiguous block—it probably isn’t in a single block to begin with. We only need to know the location of the new node, update the previous node’s next pointer to point to our new node, and point the new node’s next pointer at the correct node. If we want to add a node with value 23 to the front of the linked list in Figure 3-5, we simply set the new node’s next pointer to the previous start of the list (value = 3). This procedure is shown in Figure 3-8. Any variables previously pointing to the start of the list (the first node) also need to be updated to point to the new first node.

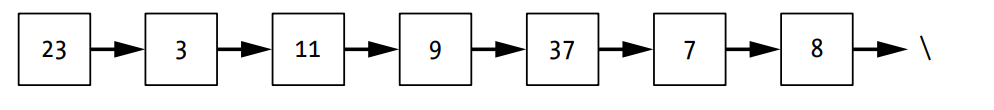


Figure 3-8: A linked list that has been extended by adding a new node to the front

Similarly, we can add a node to the end of the list, as shown in Figure 3-9, by traversing the list to the end, updating the next pointer from the final node (value = 8) to point to the new node, and setting the new node’s next pointer to null. Done naively, this approach requires traversing the entire array to reach the end, but, as we will see in the next chapter, there are ways to avoid this additional cost.

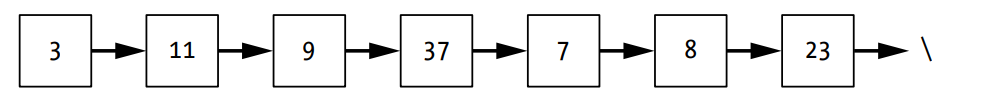


Figure 3-9: A linked list that has been extended by appending an additional node to the end

If we want to insert a value in the middle, we update two pointers: the previous node and the inserted node. For example, to add node N between nodes X and Y, we have two steps:

1. Set N’s next pointer to point at Y (the same place X’s next pointer currently points).
2. Set X’s next pointer to point at N.

The order of these two steps is important. Pointers, like all other variables, can hold only a single value—in this case a single address in memory. If we set X’s next pointer first, we would lose the data on where Y resides.   
Once we’ve finished, X points to N and N points to Y. Figure 3-10 illustrates this process.

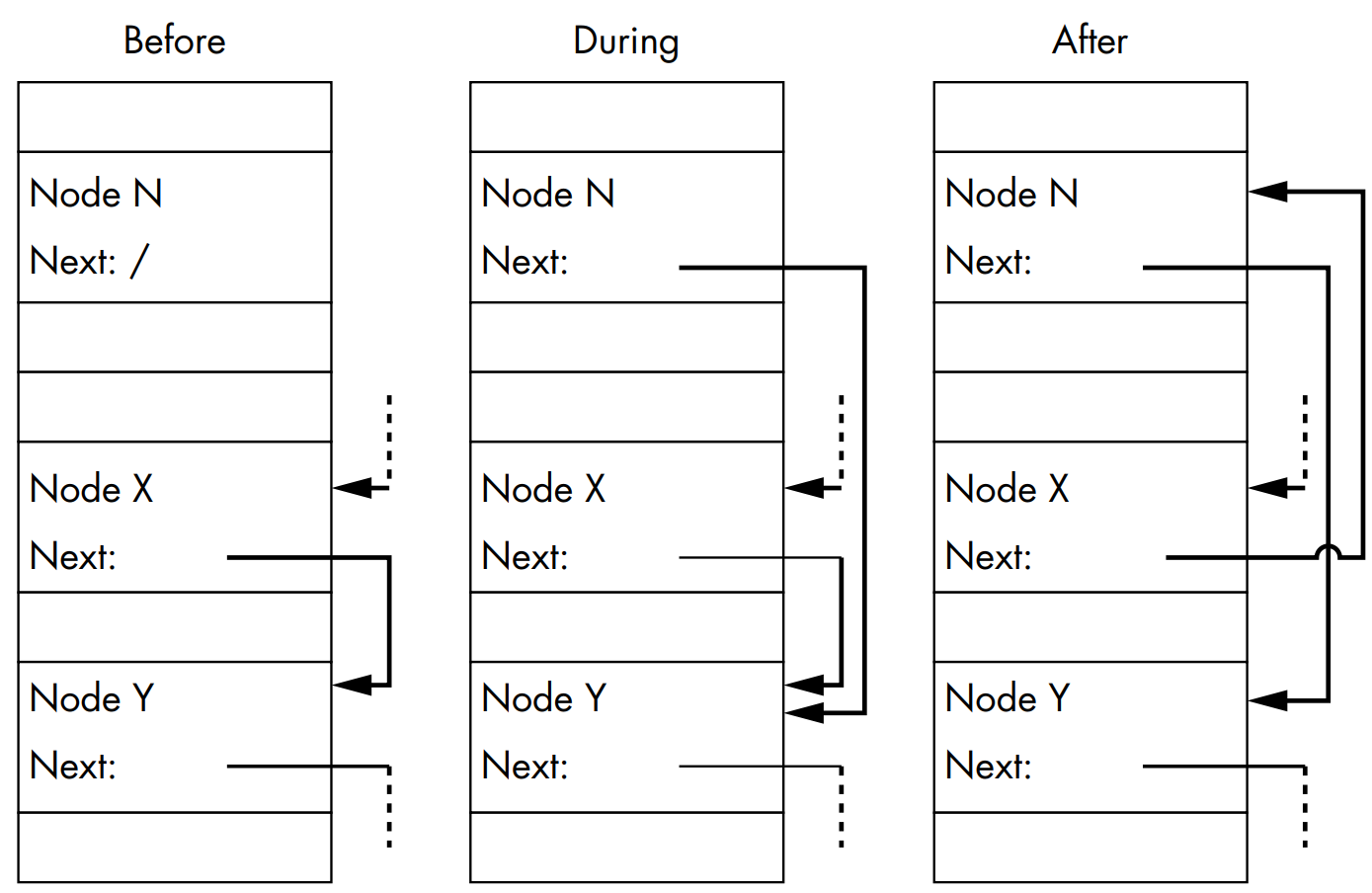


Figure 3-10: The process of inserting a new node N into a linked list between nodes X and Y

Despite the shuffling of pointers, the code for this kind of operation is relatively simple:

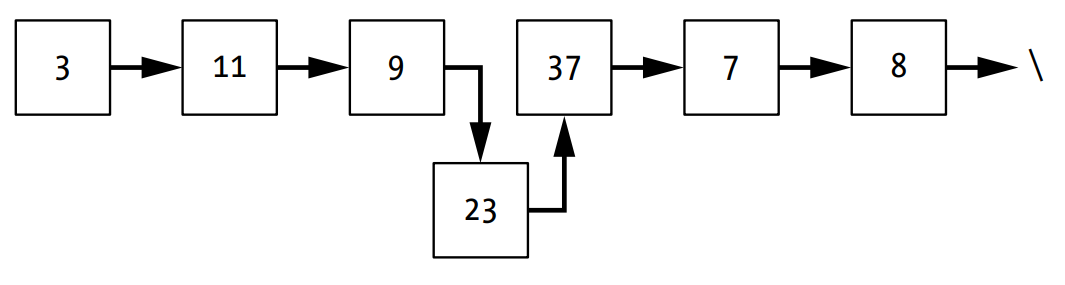
LinkedListInsertAfter(LinkedListNode: previous,

LinkedListNode: new\_node):

new\_node.next = previous.next

previous.next = new\_node

Say we instead wanted to insert a node with value 23 between the nodes 9 and 37 in our current linked list. The resulting chain of pointers would appear as shown in Figure 3-11.

Figure 3-11: Inserting the node 23 into a linked list requires updating pointers from the previous node (9) and to the following node (37).  
  
Likewise, when a customer lets their friend step in front of them to join the middle of the line, two pointers change. Recall that in this analogy, each person “points” to, or keeps track of, the person in front of them. The overly generous customer now points to their line-jumping friend who stands right in front of them. Meanwhile the happy line jumper points to the person who was previously in front of their enabling friend. Everyone behind them in line gives dirty looks and mumbles unkind things.   
Again, the diagrams and the café line analogy hide the insertion process’s true messiness. While we aren’t inserting the new node in a memory location adjacent to the last node, we are logically inserting it next in line. The node itself could be on the other end of the computer’s memory next to the variable counting our spelling errors or daily cups of coffee. As long as we keep the list’s pointers up-to-date, we can treat them and the nodes to which they point as a single list.   
Of course, we must take extra care when inserting a node in front of the head node (index == 0) or at an index past the end of the list. If we are inserting a node before the head node, we need to update the head pointer itself; otherwise, it will continue point to the old front of the list, and we will lose the ability to access the new first element. If we are trying to insert a node into an index past the end of the list, there is no valid previous node at index - 1. In this case, we could fail the insertion, return an error, or append the element to the end of the list (at a smaller index). Whichever approach you chose, it is critical that you clearly document your code. We can bundle this extra logic into a helper function that combines our linear lookup code to insert a new node at a given position:

LinkedListInsert(LinkedListNode: head, Integer: index,

Type: value):

# Special case inserting a new head node.

1 IF index == 0:

LinkedListNode: new\_head = LinkedListNode(value)

new\_head.next = head

return new\_head

LinkedListNode: current = head

LinkedListNode: previous = null

Integer: count = 0

2 WHILE count < index AND current != null:

previous = current

current = current.next  
 count = count + 1

# Check if we've run off the end of the list before

# getting to the necessary index.

3 IF count < index:

Produce an invalid index error.

4 LinkedListNode: new\_node = LinkedListNode(value)

new\_node.next = previous.next

previous.next = new\_node

5 return head  
  
The code for insertion starts with the special case of inserting a new node at index = 0, the beginning of the list 1. It creates a new head node, sets the new head node’s next pointer to the previous head of the list, and returns the new head of the list. Since there isn’t a node before the new head node, we do not need to update a previous node’s next pointer in this case.  
  
For elements in the middle of the list, the code needs to traverse the list to find the correct location 2. This is similar to the LinkedListLookUp search: the code follows each node’s next pointer, while tracking the current node and the count seen, until it hits the end of the list or the correct location. The code also tracks an additional piece of information, previous, a pointer to the node before the current node. Tracking previous allows us to update the pointer into the inserted node.  
The code then checks whether it has arrived at the desired index of insertion 3. By making the check count < index, we still allow insertion at the very end of the list. We only fail with an error in cases where we try to insert at least one additional spot past the end of the list.

If the code has found the correct location to insert the node, it splices it in between previous and current. The code performs the insertion by creating a new node, setting that node’s next pointer to the address indicated by previous.next, and then setting previous.next to point to the new node 4. This logic also works for the case where we are appending the new node immediately after the last node in the list. Since previous.next == null in that case, the new node’s next pointer is assigned to null and correctly indicates the new end of the list.

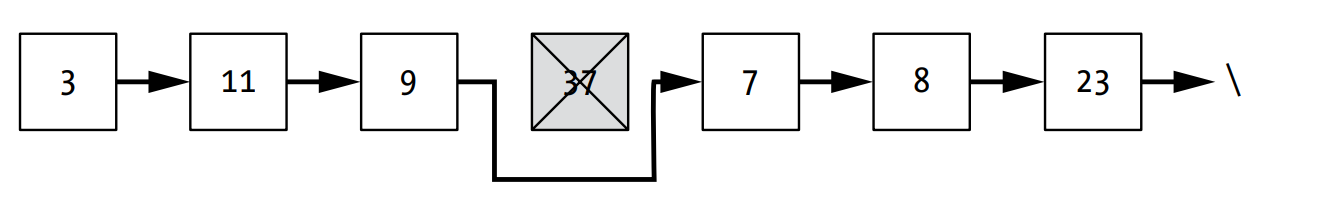
By returning the head of the list 5, we can account for insertions before the head node. Alternatively, we could wrap the head node in a LinkedList composite data structure and operate on that directly. We will use this alternate approach later in the book to handle binary search trees.  
  
**Deleting from a Linked List**  
  
To delete an element anywhere in a linked list, all we need to do is delete that node and adjust the previous node’s pointer, as shown in Figure 3-12.  
  


Figure 3-12: Removing a node (37) from a linked list requires updating the pointer in the previous node (9) to skip ahead to the following node (7).

This corresponds to someone making the questionable decision that coffee isn’t worth the wait in line. They look at their watch, mutter something about having instant at home, and leave. As long as the person behind the newly departed customer knows who they are now behind, the line’s integrity is maintained.

In the case of an array, we would have to pay a significantly higher cost to delete an element, shifting everything following the node containing 37 by one bin toward the front of the array in order to close up the gap. This could require us to walk the entire array.

Again, we must take special care when deleting the first element in a linked list or deleting past the end of the list. When deleting the first node, we update the list’s head pointer to the address of the new head node, effectively making that node the new head of the list. When deleting past the end of the list, we have options similar to those for insertion: we can skip the deletion or return an error. The following code does the latter:

LinkedListDelete(LinkedListNode: head, Integer: index):

1 IF head == null:

return null

2 IF index == 0:

new\_head = head.next

head.next = null return new\_head

LinkedListNode: current = head

LinkedListNode: previous = null

Integer: count = 0

3 WHILE count < index AND current != null:

previous = current

current = current.next

count = count + 1

4 IF current != null:

5 previous.next = current.next

6 current.next = null

ELSE:

Produce an invalid index error.

7 return head

This code follows the same approach as insertion. This time we start with an additional check 1. If the list is empty, there is nothing to delete, and we can return the value null to indicate the list is still empty. Otherwise, we check whether we are deleting the first node 2 and, if so, remove the previous first node from the list and return the address of the new head node.  
To remove any later nodes (index > 0), the code must travel to the correct location in the list. Using the same logic as for insertion, the code tracks current, count, and previous while iterating through the nodes until it either finds the correct location or hits the end of the list 3. If the code finds a node at the correct index 4, it splices out the node to be removed by setting previous.next to point at one node past the current node 5. However, if the while loop ran off the end of the list and current is null, there is nothing to delete, so the code throws an error. The function also sets the removed node’s next pointer to null both to ensure consistency (it no longer has a next node in the list) and to allow programming languages with memory management to correctly free memory that is no longer used 6. The function completes by returning the address of the list’s head node 7.  
We can adapt this code to use information other than the node’s index for deletion. If we have the value of the node to delete, we could update the loop conditions 3 to remove the first node with that value:  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
WHILE current != null AND current.value != value:  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

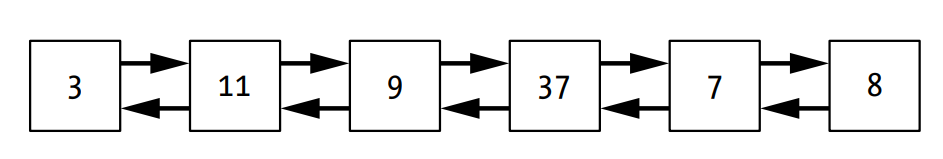
In this case, we need to reverse the order of comparison and check if current is null before accessing its value. Similarly, if we need to delete a pointer to the node, we could compare the address stored in that pointer to the address of the current node.   
The strength of linked lists is that they allow us to insert or remove elements without shifting those elements around in the computer’s memory. We can leave the nodes where they are and just update the pointers to indicate their movement.  
  
**Doubly Linked Lists**  
  
There are many additional ways we can add structure with pointers, many of which we’ll examine in later chapters. For now, we’ll discuss just one simple extension of the linked list: the doubly linked list, which includes backward as well as forward pointers, as shown in Figure 3-13.  


Figure 3-13: A doubly linked list contains pointers to both the next and previous entries  
  
For algorithms that need to iterate lists in both directions, or just for adventurous programmers looking to expand the number of pointers in their data structures, it is easy to adapt a linked list to a doubly linked one:  
---------------------------------------------------------------------------------------------------

DoublyLinkedListNode {

Type: Value

DoublyLinkedListNode: next

DoublyLinkedListNode: previous

}  
---------------------------------------------------------------------------------------------------

The code for operating on doubly linked lists is similar to the code for singly linked lists. Lookups, insertions, and deletions often require traversing the list to find the correct element. Updating the appropriate nodes’ previous pointers in addition to their next pointers requires additional logic. Yet this small amount of additional information can enable shortcuts to some of the operations. Given the pointer to any node in a doubly linked list, we can also access the node before it without having to traverse the entire list from the beginning, as we would have to do for a singly linked list.  
  
**Arrays and Linked Lists of Items**  
  
So far, we have primarily used arrays to store individual (numeric) values. We might be storing a list of top scores, a list of reminder times for a smart alarm clock, or a log of how much coffee we consume each day. This is useful in a variety of applications but is only the most basic way to use an array. We can use the concept of pointers to store more complex and differently sized items.

Suppose you’re planning a party. We will make the generous assumption that, unlike many parties thrown by the author, your gathering is popular enough to require an RSVP list. As you begin to receive responses to your invitations, you write a new program using an array to keep track of the guests. You’d like to store at least a single string in each entry, indicating the name of the person who has responded. However, you immediately run into the problem that strings might not be fixed size, so you can’t guarantee they will fit in the array’s fixed-size bin. You could expand the bin size to fit all possible strings. But how much is enough? Can you reliably say all your invitees will have fewer than 1,000 characters in their name? And if we allow for 1,000 characters, what about the waste? If we are reserving space for 1,000 characters per invitee, then entries for “John Smith” are using only a tiny fraction of their bins. What if we want to include even more dynamic data with each record, such as a list of each guest’s music preferences or nicknames?

The natural solution is to combine arrays and pointers, as shown in Figure 3-14. Each bin in the array stores a single pointer to the data of interest. In this case, each bin stores a pointer to a string located somewhere else in memory. This allows the data for each entry to vary in size. We can allocate as much space as we need for each string and point to those strings from the array. We could even create a detailed composite data structure for our RSVP records and link those from the array.

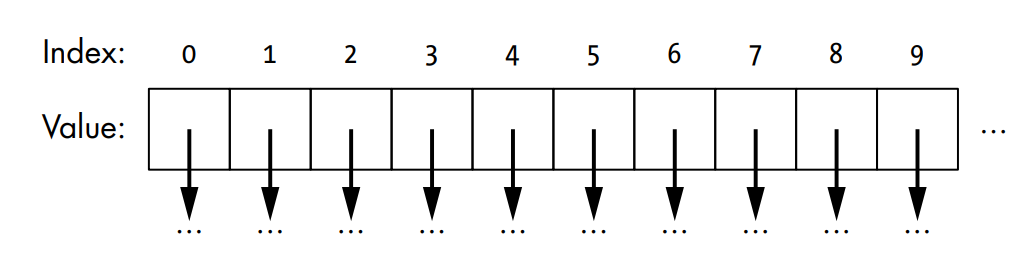


Figure 3-14: Arrays can store a series of pointers, allowing them to link to larger data structures.  
  
The RSVP records don’t need to fit into the array bins, because their data lives somewhere else in memory. The array bins only hold (fixed-size) pointers. Similarly, a linked list’s nodes can contain pointers to other data. Unlike the next pointers in a linked list, which are pointing to other nodes, these pointers can point to arbitrary other blocks of data.   
The rest of the book includes many cases where individual “values” are actually pointers to complex and even dynamic data structures.

**Why This Matters**

Linked lists and arrays are only the simplest example of how we can trade off among complexity, efficiency, and flexibility in our data structures. By using a pointer, a variable that stores addresses in memory, we can link across blocks of memory. A single fixed-size array bin can point to complex data records or strings of different lengths. Further, we can use pointers to create dynamically linked structures through the computer’s memory. By changing a pointer’s value to point to a new address, we can change this structure as needed at any time.   
Over the remaining chapters, we will see numerous examples of how dynamic data structures can be used to both improve organization of the data and make certain computations more efficient. However, it is important to keep the relative tradeoffs in mind. As we saw with arrays and linked lists, each data structure comes with its own advantages and disadvantages in terms of flexibility, space requirements, efficiency of operations, and complexity. In the next chapter, we will show how can build on these fundamental concepts to create two data structures, stacks and queues, that enable different behavior.

**3 ESTRUCTURAS DE DATOS DINÁMICAS**

Este capítulo introduce las estructuras de datos dinámicas, que alteran su estructura a medida que los datos cambian. Estas adaptaciones estructurales pueden incluir el aumento del tamaño de la estructura de datos según la demanda, la creación de enlaces dinámicos y mutables entre diferentes valores, y más. Las estructuras de datos dinámicas están en el corazón de casi todos los programas informáticos del mundo y son la base de algunos de los algoritmos más emocionantes, interesantes y poderosos en la ciencia de la computación. Las estructuras de datos básicas introducidas en los capítulos anteriores son como estacionamientos: nos dan un lugar para almacenar información, pero no ofrecen mucho en cuanto a adaptación. Claro, podemos ordenar los valores en un arreglo (o los coches en nuestro estacionamiento) y usar esa estructura para hacer que la búsqueda binaria sea eficiente. Pero solo estamos cambiando el orden de los datos dentro del arreglo. La estructura de datos en sí misma no está ni cambiando ni respondiendo a los cambios en los datos. Si más tarde cambiamos los datos en un arreglo ordenado, por ejemplo, modificando el valor de un elemento, necesitamos reordenar el arreglo. Peor aún, cuando necesitamos cambiar la estructura de datos en sí misma —ya sea ampliando o reduciendo el arreglo, por ejemplo— las estructuras de datos estáticas simples no proporcionan ninguna ayuda.

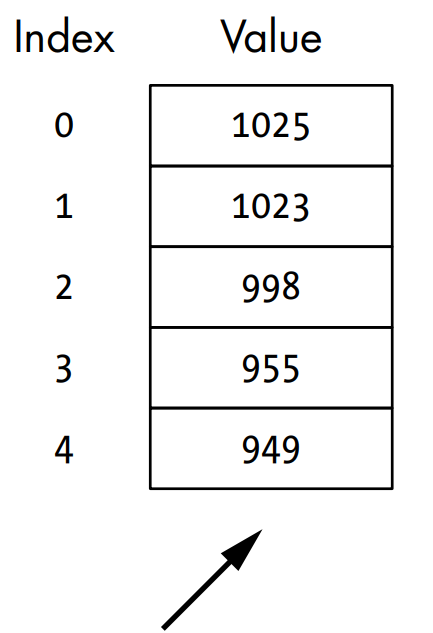
Este capítulo compara la estructura de datos estática introducida en el Capítulo 2, el arreglo, con una estructura de datos dinámica simple, la lista enlazada, para demostrar las ventajas de esta última. En algunos aspectos, estas dos estructuras de datos son similares: ambas permiten a los programadores almacenar y acceder a múltiples valores a través de una única referencia, ya sea el arreglo o la cabeza de la lista enlazada. Sin embargo, los arreglos tienen una estructura fija en el momento de su creación, como filas de espacios de estacionamiento. En contraste, las listas enlazadas pueden crecer a lo largo de la memoria del programa. Se comportan más como una línea de personas que se alarga o se acorta, permitiendo adiciones y eliminaciones. Comprender estas diferencias proporciona una base para entender las estructuras de datos más avanzadas que visitaremos en el resto de este libro.

**Las limitaciones de los arreglos**

Si bien los arreglos son excelentes estructuras de datos para almacenar múltiples valores, tienen una limitación importante: su tamaño y disposición en la memoria se fijan en el momento de la creación. Si queremos almacenar más valores de los que pueden caber en nuestro arreglo, necesitamos crear un nuevo arreglo más grande y copiar los datos del arreglo anterior. Esta memoria de tamaño fijo es aceptable cuando tenemos un límite superior inamovible en la cantidad de elementos que necesitamos almacenar. Si tenemos suficientes compartimentos para nuestros datos, podemos establecer entradas individuales todo el día sin preocuparnos por la disposición estática del arreglo en la memoria. Sin embargo, muchas aplicaciones requieren estructuras de datos dinámicas que puedan crecer y cambiar con nuestro programa.

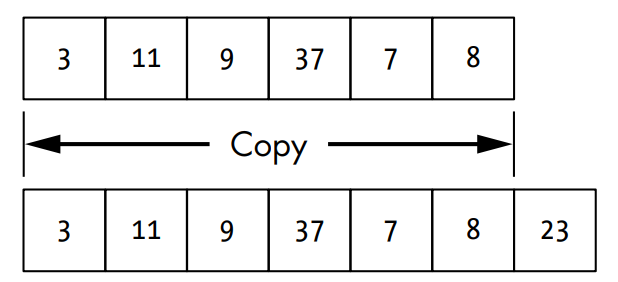
Para satisfacer esta necesidad de estructuras de datos dinámicas, muchos lenguajes de programación modernos ofrecen "arreglos" dinámicos que crecen y se reducen a medida que se agregan elementos. Sin embargo, en realidad, estos son envoltorios alrededor de arreglos estáticos u otras estructuras de datos que ocultan las complejidades y costos asociados con su naturaleza dinámica. Si bien esto es conveniente para el programador, puede llevar a ineficiencias ocultas. Cuando agregamos elementos más allá del final del arreglo, el programa todavía necesita aumentar la memoria utilizada. Simplemente lo hace detrás de escena. Para entender por qué las estructuras de datos dinámicas son tan importantes, necesitamos discutir sus limitaciones. En este libro, utilizaremos el término arreglo para referirnos a un arreglo estático simple.

Para ilustrar las restricciones del arreglo, imagina que pasas toda una semana dominando el último fenómeno de los videojuegos retro, Space Frogger 2000. Sonríes de alegría cada vez que la pantalla principal muestra tus cinco mejores puntuaciones. Estos logros monumentales representan horas de sudor, lágrimas, gritos y más lágrimas. Sin embargo, al día siguiente, tu (pronto ex-mejor) amigo te visita y supera tu puntuación más alta cinco veces seguidas. Después de echar al traicionero ex-amigo de la casa, regresas a tu juego y miras las nuevas mejores puntuaciones, mostradas en la Figura 3-1, y gritas: "¿Por qué el juego no puede almacenar más puntuaciones? ¿Realmente sería tan difícil mantener una lista de los diez mejores, o al menos agregar una más al final?".

  
"Tu mejor puntuación iría aquí."  
  
"Figura 3-1: Un arreglo de cinco elementos que contiene las puntuaciones más altas de un videojuego. Ninguna, por desgracia, es tuya."

Esta es una de las limitaciones fundamentales de cualquier estructura de datos de tamaño fijo y su disposición fija en la memoria: no puede crecer con los datos. Como veremos a continuación, esta restricción hace que algunas operaciones comunes sean costosas. Más prácticamente, imagina las limitaciones de un procesador de texto con espacio para solo un número fijo de caracteres, una hoja de cálculo con un número fijo de filas, un programa de almacenamiento de fotos que puede almacenar un número limitado de imágenes, o un diario de café limitado a solo mil entradas.

Dado que el tamaño de un arreglo está fijado en el momento de su creación, si queremos extender el arreglo para almacenar más datos, debemos crear un nuevo bloque de memoria más grande. Considera el caso más simple de agregar un solo elemento al final de un arreglo. Debido a que un arreglo es un solo bloque de memoria de tamaño fijo, no podemos simplemente insertar otro valor al final. Podría haber otra variable que ya ocupe ese espacio en la memoria. En lugar de arriesgarse a sobrescribir el valor de esa variable, tenemos que asignar un nuevo bloque de memoria (más grande), copiar todos los valores del arreglo original en el nuevo bloque y escribir el nuevo valor al final. Eso implica mucho trabajo extra para una sola adición, como se ilustra en la Figura 3-2.

  
  
"Figura 3-2: Agregar un elemento al final de un arreglo completo requiere asignar un nuevo espacio y copiar los elementos."  
  
Imagina un arreglo como uno de esos bufés de hotel con calefacción y un número fijo de ranuras. Es fácil sacar la bandeja vacía de huevos revueltos y colocar una nueva en su lugar. Pero no puedes simplemente agregar una bandeja nueva al final. No hay espacio para ello. Si el chef decide agregar panqueques al menú, algo más tiene que salir.

Si sabes que necesitarás insertar muchos valores nuevos, puedes distribuir el costo a lo largo de múltiples actualizaciones, amortizando el costo. Podrías adoptar una estrategia como el "duplicado de arreglo", en la que el tamaño del arreglo se duplica cada vez que se expande. Por ejemplo, si intentamos agregar un elemento número 129 a nuestro arreglo de tamaño 128, primero asignamos un nuevo arreglo de tamaño 256 y copiamos los 128 elementos originales. Esto nos permite seguir aumentando el arreglo durante un tiempo antes de que necesitemos preocuparnos por asignar nuevo espacio. Sin embargo, el costo es el espacio potencialmente desperdiciado. Si solo necesitamos 129 elementos en total, hemos sobredimensionado en 127.

El duplicado de arreglo proporciona un equilibrio razonable entre costosas copias de arreglo y memoria desperdiciada. A medida que el arreglo crece, las duplicaciones se vuelven cada vez menos frecuentes. Al mismo tiempo, al duplicar el arreglo cuando está lleno, garantizamos desperdiciar menos de la mitad del espacio. Sin embargo, incluso con este enfoque equilibrado, podemos ver claramente el costo de usar un arreglo de tamaño fijo en términos de costo de copia y uso de memoria.

-------------------------------------------------------------------------------

ArrayDouble(Array: old\_array):

Integer: length = length of old\_array

Array: new\_array = empty array of size length \* 2

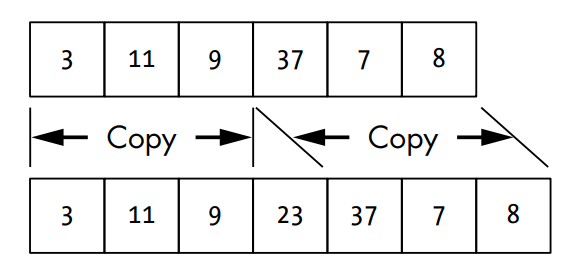
Integer: j = 0

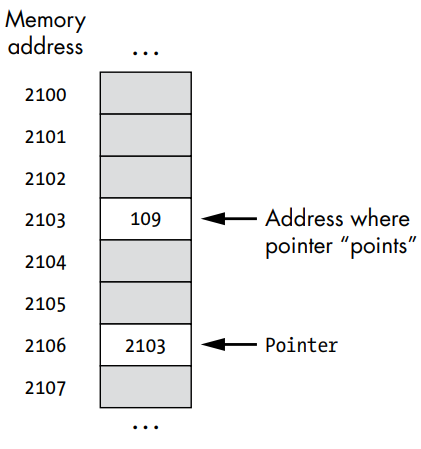
WHILE j < length:

new\_array[j] = old\_array[j]

j = j + 1

return new\_array

-------------------------------------------------------------------------------  
  
El código para duplicar un arreglo comienza asignando un nuevo arreglo con el doble de tamaño del arreglo actual. Un solo bucle WHILE itera sobre los elementos del arreglo actual, copiando sus valores en el nuevo arreglo. El nuevo arreglo es devuelto al final del proceso.  
  
Imagina aplicar esta estrategia al espacio en estanterías. Establecemos una librería, "Estructuras de Datos y Más", en una ubicación e instalamos humildemente cinco estantes. El día de apertura ve una sorprendente demanda y solicitudes de más variedad: necesitamos expandir nuestro inventario. En pánico, nos mudamos a una nueva ubicación con 10 estantes y migramos los libros. La demanda ha sido temporalmente satisfecha.  
  
Dado que la falta de una tienda integral de estructuras de datos es una clara brecha en el mercado de libros al por menor, nuestra tienda se convierte en un éxito rotundo y la demanda sigue creciendo y creciendo. Podríamos actualizar la tienda varias veces a ubicaciones con 20, 40, luego 80 estantes. Cada vez pagamos un costo para asegurar una nueva ubicación y migrar los libros.  
  
La ubicación fija de los valores del arreglo en la memoria proporciona otra limitación. No podemos insertar fácilmente elementos adicionales en el medio de un arreglo. Incluso si hay suficientes espacios vacíos al final de nuestro arreglo original para acomodar un nuevo elemento, y por lo tanto no necesitamos mover todo el arreglo a un nuevo bloque de memoria, aún necesitamos desplazar cada elemento existente uno por uno para hacer espacio para el nuevo valor en el medio.  
  
A diferencia de una estantería de libros, no podemos simplemente mover todos los elementos a la vez con un solo empujón. Si tuviéramos 10,000 elementos y quisiéramos agregar algo en la segunda posición, necesitaríamos mover 9,999 elementos. Eso requiere mucho esfuerzo para insertar un solo elemento.  
  
Los problemas se agravan cuando intentamos insertar nuevos valores en el medio de un arreglo que ya está lleno. No solo tenemos que asignar un nuevo bloque y copiar los valores antiguos, sino que también necesitamos desplazar los valores después del nuevo valor una posición hacia abajo para hacer espacio para nuestro nuevo valor. Por ejemplo, supongamos que queremos insertar el valor 23 como el cuarto elemento de un arreglo existente de seis elementos, como se ilustra en la Figura 3-3.  
  
  
***Figura 3-3:*** *Agregar un elemento en el medio de un arreglo completo requiere asignar un nuevo espacio, copiar los elementos antes de la inserción y copiar los valores después de la inserción a ubicaciones desplazadas.*  
  
Para abordar las deficiencias de los arreglos, necesitamos pasar a estructuras de datos más flexibles que puedan crecer a medida que se agregan nuevos datos: las estructuras de datos dinámicas. Antes de adentrarnos en los detalles, presentemos los punteros, el tipo de variable que es clave para reconfigurar y hacer crecer las estructuras de datos.  
  
**Punteros y Referencias**  
  
Un tipo de variable destaca entre sus pares tanto por su gran poder como por su capacidad para confundir a los nuevos programadores: los punteros. Un puntero es una variable que almacena solo las direcciones en la memoria de la computadora. Por lo tanto, el puntero apunta a una segunda ubicación en la memoria donde se almacenan los datos reales, como se muestra en la Figura 3-4.

  
"Figura 3-4: Un puntero que indica una dirección en la memoria de la computadora

El lector perspicaz puede preguntar: '¿Cuál es el propósito de una variable que simplemente apunta a otra ubicación en la memoria? Pensé que el nombre de la variable ya cumplía esa función. ¿Por qué no almacenar tus datos en la variable como una persona normal? ¿Por qué siempre tienes que complicar las cosas?' No escuches a los escépticos. Los punteros son el ingrediente esencial en las estructuras de datos dinámicas, como veremos en breve. Supongamos que estamos trabajando en un importante proyecto arquitectónico en la oficina y hemos reunido una carpeta con dibujos de ejemplo para compartir con nuestro equipo. Pronto, la carpeta del proyecto contiene numerosos planos, estimaciones de costos y representaciones artísticas. En lugar de hacer una copia del archivo voluminoso y dejarlo a la vista, dejamos una nota indicando a nuestros colaboradores que encuentren el archivo en la sala de registros del tercer piso, en el archivador #3, el segundo cajón hacia abajo, la quinta carpeta. Esta nota cumple el papel de un puntero. No detalla toda la información que está en el archivo, sino que permite a nuestros colegas encontrar y recuperar la información. Más importante aún, podemos compartir esta única 'dirección' con cada uno de nuestros compañeros de trabajo sin hacer una copia completa del archivo para ellos. Cada uno puede usar esta información para buscar y modificar la carpeta cuando sea necesario. Incluso podríamos dejar una nota adhesiva individual en el escritorio de cada miembro del equipo, proporcionando 10 variables que apuntan a la misma información. Además de almacenar la ubicación de un bloque de memoria, los punteros pueden tener un valor nulo (denotado como None, Nil o 0 en algunos lenguajes de programación). El valor nulo simplemente indica que el puntero no está apuntando actualmente a una ubicación válida en la memoria. En otras palabras, indica que el puntero aún no apunta a nada. Diferentes lenguajes de programación proporcionan diferentes mecanismos para lograr la tarea de los punteros, y no todos proporcionan la dirección de memoria en bruto al programador. Los lenguajes de bajo nivel como C y C++ te dan punteros en bruto y te permiten acceder directamente a la ubicación de memoria que almacenanOtros lenguajes de programación, como Python, utilizan referencias, que emplean una sintaxis similar a la de una variable normal, pero que permiten hacer referencia a otra variable. Estas diferentes variaciones tienen diferentes comportamientos y usos (desreferenciación, matemáticas con punteros, la forma de valores nulos, etc.). Para simplificar, a lo largo de este libro utilizaremos el término puntero para referirnos a todas las variables implementadas por punteros, referencias o índices en bloques de memoria preasignados. No nos preocuparemos por la sintaxis complicada necesaria para acceder a los bloques de memoria (lo que ha causado que más de unos pocos entusiastas de la programación se derrumben en lágrimas). También utilizaremos el tipo de datos final (en lugar del tipo más genérico puntero) al definir una variable de puntero en pseudocódigo. El concepto clave para nuestros propósitos es que los punteros proporcionan un mecanismo para vincularse a un bloque de memoria, como se presenta en nuestra primera estructura de datos dinámica: la lista enlazada.  
  
**Listas Enlazadas**  
  
Las listas enlazadas son el ejemplo más simple de una estructura de datos dinámica y son parientes cercanos de los arreglos. Al igual que los arreglos, son una estructura de datos para almacenar múltiples valores. A diferencia de los arreglos, las listas enlazadas están compuestas por una cadena de nodos enlazados entre sí mediante punteros. Un nodo básico en una lista enlazada es una estructura de datos compuesta que contiene dos partes: un valor (de cualquier tipo) y un puntero al siguiente nodo en la lista:

LinkedListNode {

Type: value

LinkedListNode: next

}  
  
Podemos imaginar una lista enlazada como una serie de contenedores enlazados, como en la Figura 3-5. Cada contenedor almacena un solo valor y contiene un puntero al siguiente contenedor en la serie.

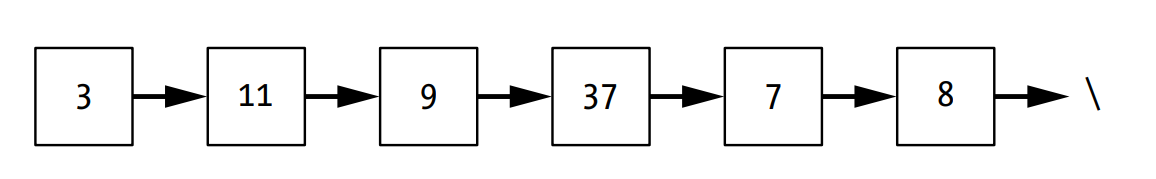
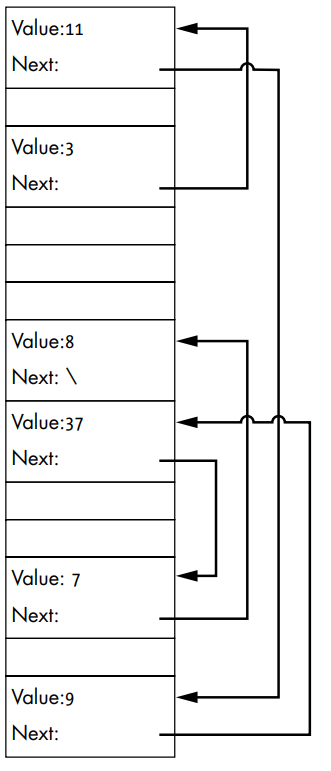


Figura 3-5: Una lista enlazada representada como una serie de nodos enlazados por punteros.

La barra al final de la lista representa un valor nulo e indica el final de la lista. En efecto, estamos diciendo que el siguiente puntero del último nodo no apunta a un nodo válido. Una lista enlazada es como una larga fila de personas esperando en nuestra cafetería favorita. Las personas rara vez conocen su posición absoluta en la fila—'Estoy en el quincuagésimo tercer azulejo desde el mostrador.' Prestan atención a su orden relativo, es decir, la única persona que está delante de ellos, que almacenamos en un puntero. Incluso si la fila se enrosca por toda la tienda (y su estacionamiento) en bucles complejos, aún podemos reconstruir el orden preguntando a cada persona quién está inmediatamente delante de ellos. Podemos recorrer la fila hacia el mostrador preguntando a cada persona quién está delante de ellos.

Porque incluyen punteros además de valores, las listas enlazadas requieren más memoria que los arreglos para almacenar los mismos elementos. Si tenemos un arreglo de tamaño K, que almacena valores de N bytes cada uno, solo necesitamos K × N bytes. En contraste, si cada puntero requiere otros M bytes, nuestra estructura de datos ahora tiene un costo de K × (M + N) bytes. A menos que el tamaño de los punteros sea mucho menor que el tamaño de nuestros valores, la sobrecarga es significativa. Sin embargo, el aumento en el uso de memoria a menudo vale la pena por la mayor flexibilidad que proporcionan los punteros.  
  
Aunque los libros de texto a menudo representan las listas enlazadas como estructuras ordenadas y organizadas (como se muestra en la Figura 3-5 o se sugiere en nuestro ejemplo de la fila de personas), nuestra lista en realidad puede estar dispersa por toda la memoria del programa. Como se ilustra en la Figura 3-6, los nodos de la lista solo están conectados a través de sus punteros. Este es el verdadero poder de los punteros y las estructuras de datos dinámicas. No estamos limitados a mantener toda la lista en un único bloque contiguo de memoria. Somos libres de asignar espacio para nuevos nodos en cualquier lugar donde exista espacio disponible.  
  


**Figura 3-6:** Una lista enlazada en la memoria de la computadora. Los nodos no están necesariamente adyacentes entre sí.

Los programas suelen almacenar listas enlazadas manteniendo un único puntero al frente, o cabeza, de la lista enlazada. Luego, el programa puede acceder a cualquier elemento de la lista comenzando en la cabeza e iterando a través de los nodos mediante los punteros:

LinkedListLookUpLinkedListNode: head, Integer: element\_number):(

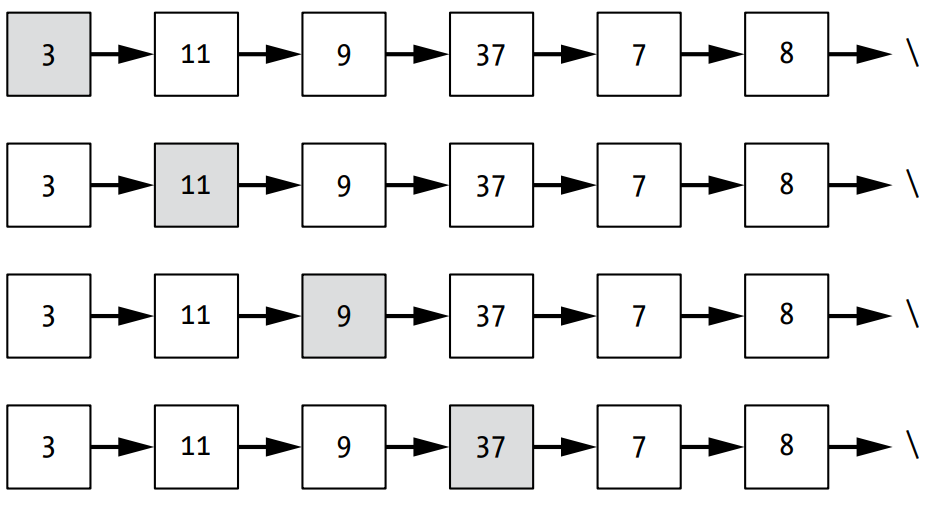
1 LinkedListNode: current = head   
Integer: count = 0   
  
2 WHILE count < element\_number AND current != null:

current = current.next   
count = count + 1

return current

El código comienza en la cabeza (head) de la lista **1.** Mantenemos una segunda variable, count, para rastrear el índice del nodo actual. El bucle WHILE itera a través de cada nodo en la lista hasta que encuentra el número correcto, count == element\_number, o hasta que llega al final de la lista, current == null **2.** En cualquier caso, el código puede devolver current. Si el bucle termina porque se ha salido del final de la lista, entonces el índice no está en la lista y el código devuelve null.

Por ejemplo, si quisiéramos acceder al cuarto elemento de una lista enlazada, el programa accedería primero a la cabeza, luego al segundo, tercer y cuarto elemento, en ese orden, para encontrar la ubicación de memoria correcta. La figura 3-7 muestra este proceso, donde el nodo con el valor 3 es la cabeza de la lista.



***Figura 3-7:****Recorrer una lista enlazada requiere moverse de un nodo al siguiente a lo largo de la cadena de punteros.*Sin embargo, vale la pena señalar que hay una compensación: las listas enlazadas tienen un mayor costo computacional en comparación con los arreglos. Cuando accedemos a un elemento en un arreglo, simplemente calculamos un único desplazamiento y buscamos la dirección de memoria correcta. El acceso a un arreglo solo requiere un cálculo matemático y una búsqueda en memoria, sin importar qué índice elijamos. En cambio, las listas enlazadas requieren que iteremos desde el principio de la lista hasta llegar al elemento de interés. Para listas más largas, la falta de acceso directo puede agregar un costo significativo.

A primera vista, este patrón de acceso restringido es una desventaja para las listas enlazadas. ¡Hemos aumentado drásticamente el costo de buscar un elemento arbitrario! Considera lo que esto significa para la búsqueda binaria. Una sola búsqueda requiere iterar sobre muchos de los elementos, eliminando la ventaja de tener una lista ordenada.

A pesar de estos costos, las listas enlazadas pueden convertirse en activos valiosos en programas prácticos. Las estructuras de datos casi siempre implican compensaciones entre complejidad, eficiencia y patrones de uso. Los mismos comportamientos que descalifican una estructura de datos para un uso podrían hacerla la elección perfecta para apoyar otros algoritmos. Entender estas compensaciones es la clave para combinar de manera efectiva algoritmos y estructuras de datos. En el caso de las listas enlazadas, la compensación por el mayor costo en el acceso a los elementos es un aumento significativo en la flexibilidad de la estructura de datos en general, como veremos en la siguiente sección.

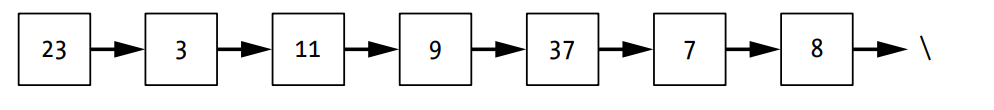
**Operaciones en listas enlazadas**

Mientras que algunos lamentan el desorden caótico de las listas enlazadas en comparación con la belleza estética de un arreglo compacto, es precisamente esta capacidad de enlazar a través de diferentes bloques de memoria lo que hace que esta estructura de datos sea tan poderosa, permitiéndonos reorganizar dinámicamente la estructura de datos. Comparemos la inserción de un nuevo valor en un arreglo con la adición de un valor a una lista enlazada.

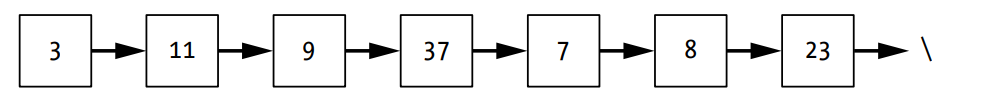
**Inserción en una lista enlazada**

Como hemos visto, insertar un nuevo elemento en un arreglo puede requerir que asignemos un nuevo bloque de memoria (más grande) y copiemos todos los valores del arreglo original en el nuevo bloque. Además, la inserción en sí puede requerir que recorramos el arreglo y desplacemos los elementos.

Por otro lado, la lista enlazada no necesita permanecer en un solo bloque contiguo—probablemente no esté en un solo bloque para empezar. Solo necesitamos conocer la ubicación del nuevo nodo, actualizar el puntero next del nodo anterior para que apunte al nuevo nodo y hacer que el puntero next del nuevo nodo apunte al nodo correcto. Si queremos agregar un nodo con el valor 23 al frente de la lista enlazada en la Figura 3-5, simplemente establecemos el puntero next del nuevo nodo para que apunte al inicio anterior de la lista (valor = 3). Este procedimiento se muestra en la Figura 3-8. Cualquier variable que apuntara previamente al inicio de la lista (el primer nodo) también debe actualizarse para apuntar al nuevo primer nodo.



***Figura 3-8:****Una lista enlazada que ha sido extendida al agregar un nuevo nodo al frente.*

De manera similar, podemos agregar un nodo al final de la lista, como se muestra en la Figura 3-9, recorriendo la lista hasta el final, actualizando el puntero next del último nodo (valor = 8) para que apunte al nuevo nodo y estableciendo el puntero next del nuevo nodo en null. De manera ingenua, este enfoque requiere recorrer toda la lista para llegar al final, pero, como veremos en el próximo capítulo, hay formas de evitar este costo adicional.

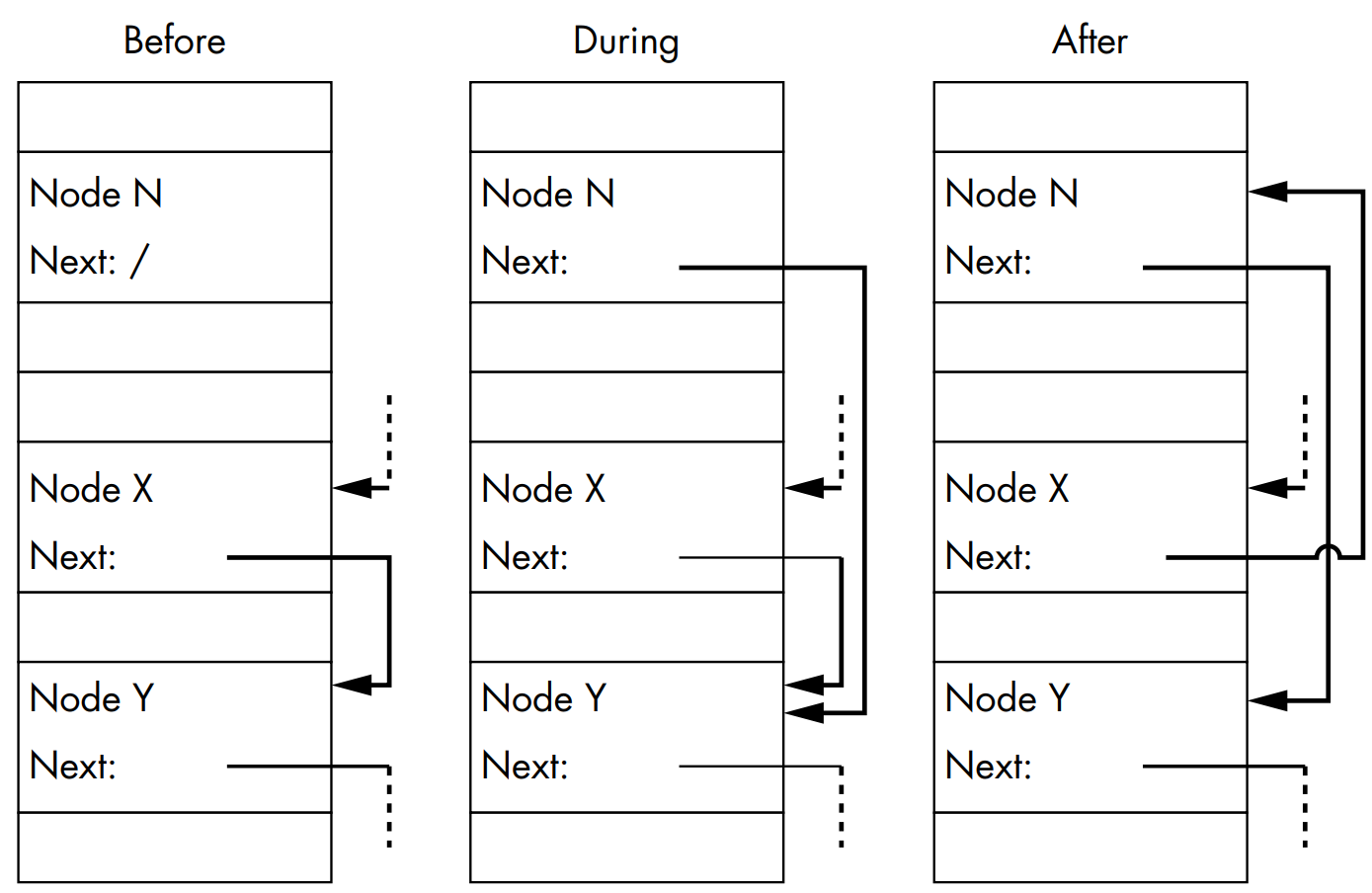
**Figura 3-9:** Una lista enlazada que ha sido extendida al agregar un nodo adicional al final.

Si queremos insertar un valor en el medio, actualizamos dos punteros: el nodo anterior y el nodo insertado. Por ejemplo, para agregar un nodo N entre los nodos X e Y, seguimos dos pasos:

1. Establecer el puntero next de N para que apunte a Y (el mismo lugar al que apunta actualmente el puntero next de X).
2. Establecer el puntero next de X para que apunte a N.

El orden de estos dos pasos es importante. Los punteros, como todas las demás variables, solo pueden contener un único valor: en este caso, una única dirección de memoria. Si establecemos primero el puntero next de X, perderíamos la información sobre dónde se encuentra Y.

Una vez que hayamos terminado, X apuntará a N y N apuntará a Y. La Figura 3-10 ilustra este proceso.



**Figura 3-10:** El proceso de inserción de un nuevo nodo N en una lista enlazada entre los nodos X e Y.

A pesar del reordenamiento de punteros, el código para este tipo de operación es relativamente simple:

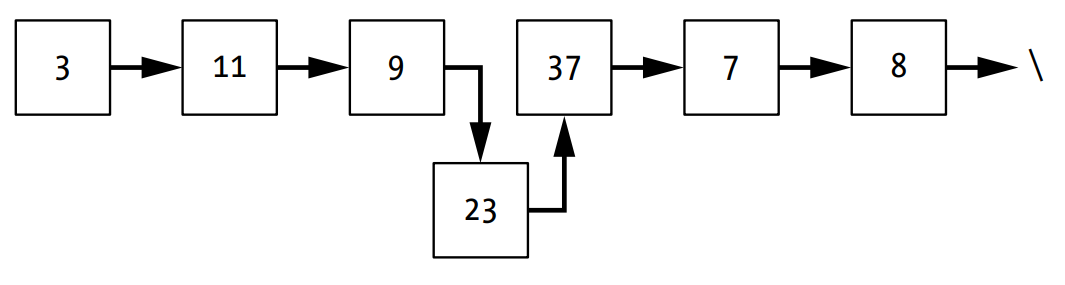
LinkedListInsertAfter(LinkedListNode: previous,

LinkedListNode: new\_node):

new\_node.next = previous.next

previous.next = new\_node

Supongamos que, en lugar de eso, queremos insertar un nodo con el valor 23 entre los nodos 9 y 37 en nuestra lista enlazada actual. La cadena de punteros resultante aparecería como se muestra en la Figura 3-11.

 **Figura 3-11:** Insertar el nodo 23 en una lista enlazada requiere actualizar los punteros del nodo anterior (9) y del nodo siguiente (37).

De manera similar, cuando un cliente permite que su amigo se coloque delante de él para unirse al medio de la fila, dos punteros cambian. Recordemos que, en esta analogía, cada persona "apunta" a, o lleva un seguimiento de, la persona que está delante de ellos. El cliente demasiado generoso ahora apunta a su amigo que se ha colado, quien se coloca justo delante de él. Mientras tanto, el afortunado amigo que se ha colado apunta a la persona que estaba previamente delante de su amigo generoso. Todos los que están detrás de ellos en la fila les lanzan miradas sucias y murmuran cosas desagradables.

Una vez más, los diagramas y la analogía de la fila del café ocultan el verdadero desorden del proceso de inserción. Aunque no estamos insertando el nuevo nodo en una ubicación de memoria adyacente al último nodo, lo estamos insertando lógicamente en la siguiente posición de la fila. El nodo en sí podría estar en el otro extremo de la memoria de la computadora, junto a la variable que cuenta nuestros errores ortográficos o las tazas de café diarias. Mientras mantengamos actualizados los punteros de la lista, podemos tratarlos y a los nodos a los que apuntan como una sola lista.

Por supuesto, debemos tener especial cuidado al insertar un nodo delante del nodo cabeza (índice == 0) o en un índice más allá del final de la lista. Si estamos insertando un nodo antes del nodo cabeza, necesitamos actualizar el puntero de la cabeza; de lo contrario, seguirá apuntando al antiguo frente de la lista y perderemos la capacidad de acceder al nuevo primer elemento. Si intentamos insertar un nodo en un índice más allá del final de la lista, no habrá un nodo anterior válido en el índice - 1. En este caso, podríamos fallar la inserción, devolver un error o agregar el elemento al final de la lista (en un índice menor). Cualquiera que sea el enfoque que elijas, es fundamental que documentes claramente tu código. Podemos agrupar esta lógica adicional en una función auxiliar que combine nuestro código de búsqueda lineal para insertar un nuevo nodo en una posición dada:

LinkedListInsert(LinkedListNode: head, Integer: index,

Type: value):

# Caso especial: insertar un nuevo nodo cabeza.

1 IF index == 0:

LinkedListNode: new\_head = LinkedListNode(value)

new\_head.next = head

return new\_head

LinkedListNode: current = head

LinkedListNode: previous = null

Integer: count = 0

2 WHILE count < index AND current != null:

previous = current

current = current.next

count = count + 1

# Verificar si hemos llegado al final de la lista antes

# de alcanzar el índice necesario.

3 IF count < index:

Producir un error de índice inválido.

4 LinkedListNode: new\_node = LinkedListNode(value)

new\_node.next = previous.next

previous.next = new\_node

5 return head

El código para la inserción comienza con el caso especial de insertar un nuevo nodo en el índice = 0, el principio de la lista 1. Crea un nuevo nodo cabeza, establece el puntero next del nuevo nodo cabeza para que apunte a la cabeza anterior de la lista y devuelve la nueva cabeza de la lista. Como no hay un nodo antes del nuevo nodo cabeza, no necesitamos actualizar el puntero next de un nodo anterior en este caso.

Para los elementos en el medio de la lista, el código necesita recorrer la lista para encontrar la ubicación correcta 2. Esto es similar a la búsqueda de LinkedListLookUp: el código sigue el puntero next de cada nodo, mientras rastrea el nodo actual y el contador visto, hasta que llega al final de la lista o a la ubicación correcta. El código también rastrea una información adicional, previous, un puntero al nodo anterior al nodo actual. Rastrear previous nos permite actualizar el puntero al nodo insertado.

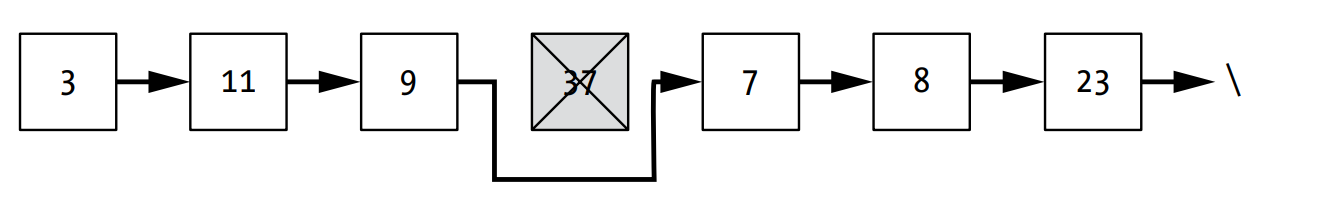
Luego, el código verifica si ha llegado al índice de inserción deseado 3. Al hacer la verificación count < index, todavía permitimos la inserción al final de la lista. Solo fallamos con un error en los casos en los que intentamos insertar al menos un lugar adicional más allá del final de la lista.

Si el código ha encontrado la ubicación correcta para insertar el nodo, lo empalma entre previous y current. El código realiza la inserción creando un nuevo nodo, estableciendo el puntero next de ese nodo en la dirección indicada por previous.next, y luego estableciendo previous.next para que apunte al nuevo nodo 4. Esta lógica también funciona para el caso en el que estamos agregando el nuevo nodo inmediatamente después del último nodo de la lista. Como previous.next == null en ese caso, el puntero next del nuevo nodo se asigna a null e indica correctamente el nuevo final de la lista.

Al devolver la cabeza de la lista 5, podemos tener en cuenta las inserciones antes del nodo cabeza. Alternativamente, podríamos envolver el nodo cabeza en una estructura de datos compuesta LinkedList y operar directamente sobre ella. Usaremos este enfoque alternativo más adelante en el libro para manejar árboles de búsqueda binaria.

**Eliminación de una lista enlazada**

Para eliminar un elemento en cualquier parte de una lista enlazada, todo lo que necesitamos hacer es eliminar ese nodo y ajustar el puntero del nodo anterior, como se muestra en la Figura 3-12.



**Figura 3-12:** Eliminar un nodo (37) de una lista enlazada requiere actualizar el puntero en el nodo anterior (9) para que salte al nodo siguiente (7).

Esto corresponde a alguien que toma la cuestionable decisión de que el café no vale la pena esperar en la fila. Miran su reloj, murmuran algo sobre tener café instantáneo en casa y se van. Mientras la persona detrás del cliente recién saliente sepa quién está ahora delante de ellos, la integridad de la fila se mantiene.

En el caso de un arreglo, tendríamos que pagar un costo significativamente mayor para eliminar un elemento, desplazando todo lo que sigue al nodo que contiene 37 en una posición hacia el frente del arreglo para cerrar la brecha. Esto podría requerir que recorramos todo el arreglo.

Nuevamente, debemos tener especial cuidado al eliminar el primer elemento de una lista enlazada o al eliminar más allá del final de la lista. Cuando eliminamos el primer nodo, actualizamos el puntero de la cabeza de la lista a la dirección del nuevo nodo cabeza, convirtiendo efectivamente ese nodo en la nueva cabeza de la lista. Cuando eliminamos más allá del final de la lista, tenemos opciones similares a las de la inserción: podemos omitir la eliminación o devolver un error. El siguiente código hace lo último:  
  
LinkedListDelete(LinkedListNode: head, Integer: index):

1 IF head == null:

return null

2 IF index == 0:

new\_head = head.next

head.next = null

return new\_head

LinkedListNode: current = head

LinkedListNode: previous = null

Integer: count = 0

3 WHILE count < index AND current != null:

previous = current

current = current.next

count = count + 1

4 IF current != null:

5 previous.next = current.next

6 current.next = null

ELSE:

Producir un error de índice inválido.

7 return head

Este código sigue el mismo enfoque que la inserción. Esta vez comenzamos con una verificación adicional 1. Si la lista está vacía, no hay nada que eliminar, y podemos devolver el valor null para indicar que la lista sigue vacía. De lo contrario, verificamos si estamos eliminando el primer nodo 2 y, de ser así, eliminamos el primer nodo anterior de la lista y devolvemos la dirección del nuevo nodo cabeza.

Para eliminar cualquier nodo posterior (índice > 0), el código debe llegar a la ubicación correcta en la lista. Usando la misma lógica que para la inserción, el código rastrea current, count y previous mientras itera a través de los nodos hasta que encuentra la ubicación correcta o llega al final de la lista 3. Si el código encuentra un nodo en el índice correcto 4, empalma el nodo que se va a eliminar estableciendo previous.next para que apunte a un nodo más allá del nodo actual 5. Sin embargo, si el bucle while se sale del final de la lista y current es null, no hay nada que eliminar, por lo que el código lanza un error. La función también establece el puntero next del nodo eliminado en null tanto para garantizar la consistencia (ya no tiene un nodo siguiente en la lista) como para permitir que los lenguajes de programación con gestión de memoria liberen correctamente la memoria que ya no se usa 6. La función finaliza devolviendo la dirección del nodo cabeza de la lista 7.

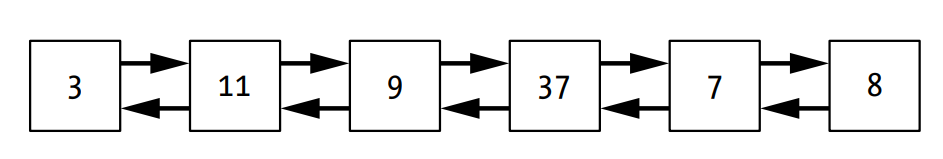
Podemos adaptar este código para usar información diferente al índice del nodo para la eliminación. Si tenemos el valor del nodo que queremos eliminar, podríamos actualizar las condiciones del bucle 3 para eliminar el primer nodo con ese valor:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
WHILE current != null AND current.value != value:  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

En este caso, necesitamos invertir el orden de comparación y verificar si current es null antes de acceder a su valor. De manera similar, si necesitamos eliminar un puntero al nodo, podríamos comparar la dirección almacenada en ese puntero con la dirección del nodo actual.

La fortaleza de las listas enlazadas es que nos permiten insertar o eliminar elementos sin necesidad de desplazar esos elementos en la memoria de la computadora. Podemos dejar los nodos donde están y simplemente actualizar los punteros para indicar su movimiento.

**Listas doblemente enlazadas**

Hay muchas formas adicionales en las que podemos agregar estructura con punteros, muchas de las cuales examinaremos en capítulos posteriores. Por ahora, discutiremos una simple extensión de la lista enlazada: la lista doblemente enlazada, que incluye punteros hacia atrás y hacia adelante, como se muestra en la Figura 3-13.  


**Figura 3-13:** Una lista doblemente enlazada contiene punteros tanto a la siguiente entrada como a la anterior.

Para algoritmos que necesitan iterar listas en ambas direcciones, o simplemente para programadores aventureros que buscan expandir el número de punteros en sus estructuras de datos, es fácil adaptar una lista enlazada a una doblemente enlazada:

---------------------------------------------------------------------------------------------------

DoublyLinkedListNode {

Type: Value

DoublyLinkedListNode: next

DoublyLinkedListNode: previous

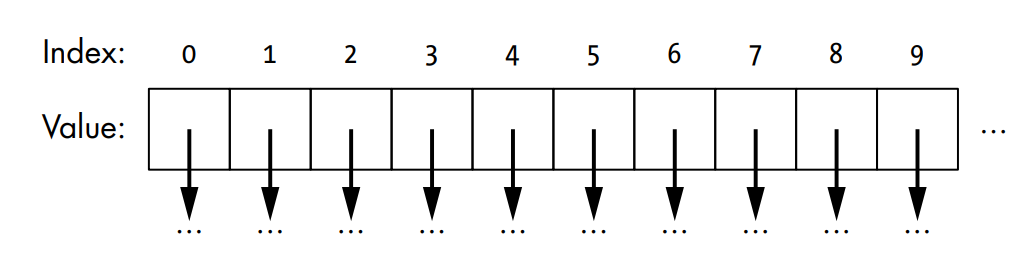
}  
---------------------------------------------------------------------------------------------------

El código para operar en listas doblemente enlazadas es similar al código para listas simplemente enlazadas. Las búsquedas, inserciones y eliminaciones a menudo requieren recorrer la lista para encontrar el elemento correcto. Actualizar los punteros previous de los nodos apropiados, además de sus punteros next, requiere lógica adicional. Sin embargo, esta pequeña cantidad de información adicional puede permitir atajos para algunas operaciones. Dado el puntero a cualquier nodo en una lista doblemente enlazada, también podemos acceder al nodo anterior sin tener que recorrer toda la lista desde el principio, como tendríamos que hacer en una lista simplemente enlazada.  
  
**Arreglos y listas enlazadas de elementos**

Hasta ahora, hemos utilizado principalmente arreglos para almacenar valores individuales (numéricos). Podríamos estar almacenando una lista de puntuaciones más altas, una lista de horas de recordatorio para un despertador inteligente o un registro de cuánto café consumimos cada día. Esto es útil en una variedad de aplicaciones, pero es solo la forma más básica de usar un arreglo. Podemos usar el concepto de punteros para almacenar elementos más complejos y de diferentes tamaños.

Supongamos que estás planeando una fiesta. Haremos el generoso supuesto de que, a diferencia de muchas fiestas organizadas por el autor, tu reunión es lo suficientemente popular como para requerir una lista de RSVP. A medida que comienzas a recibir respuestas a tus invitaciones, escribes un nuevo programa usando un arreglo para llevar un registro de los invitados. Te gustaría almacenar al menos una cadena en cada entrada, indicando el nombre de la persona que ha respondido. Sin embargo, inmediatamente te encuentras con el problema de que las cadenas pueden no tener un tamaño fijo, por lo que no puedes garantizar que encajarán en los contenedores de tamaño fijo del arreglo. Podrías expandir el tamaño del contenedor para que quepan todas las cadenas posibles. Pero, ¿cuánto es suficiente? ¿Puedes decir con certeza que todos tus invitados tendrán menos de 1,000 caracteres en sus nombres? Y si permitimos 1,000 caracteres, ¿qué pasa con el desperdicio? Si reservamos espacio para 1,000 caracteres por invitado, entonces las entradas para "John Smith" están usando solo una pequeña fracción de sus contenedores. ¿Qué pasa si queremos incluir datos aún más dinámicos con cada registro, como una lista de las preferencias musicales o apodos de cada invitado?

La solución natural es combinar arreglos y punteros, como se muestra en la Figura 3-14. Cada contenedor en el arreglo almacena un solo puntero a los datos de interés. En este caso, cada contenedor almacena un puntero a una cadena ubicada en otro lugar de la memoria. Esto permite que los datos de cada entrada varíen en tamaño. Podemos asignar tanto espacio como necesitemos para cada cadena y apuntar a esas cadenas desde el arreglo. Incluso podríamos crear una estructura de datos compuesta detallada para nuestros registros de RSVP y vincularlos desde el arreglo.



***Figura 3-14****: Las matrices pueden almacenar una serie de punteros, lo que les permite enlazarse a estructuras de datos más grandes.*

Los registros RSVP no necesitan ajustarse a los contenedores de la matriz, ya que sus datos residen en otro lugar de la memoria. Los contenedores de la matriz solo contienen punteros (de tamaño fijo). De manera similar, los nodos de una lista enlazada pueden contener punteros a otros datos. A diferencia de los siguientes punteros en una lista enlazada, que apuntan a otros nodos, estos punteros pueden apuntar a otros bloques de datos arbitrarios. El resto del libro incluye muchos casos en los que los "valores" individuales son en realidad punteros a estructuras de datos complejas e incluso dinámicas.

**Por qué esto importa**

Las listas enlazadas y los arreglos son solo el ejemplo más simple de cómo podemos equilibrar la complejidad, la eficiencia y la flexibilidad en nuestras estructuras de datos. Al usar un puntero, una variable que almacena direcciones en la memoria, podemos enlazar bloques de memoria. Un contenedor de arreglo de tamaño fijo puede apuntar a registros de datos complejos o cadenas de diferentes longitudes. Además, podemos usar punteros para crear estructuras dinámicamente enlazadas a través de la memoria de la computadora. Al cambiar el valor de un puntero para que apunte a una nueva dirección, podemos modificar esta estructura según sea necesario en cualquier momento.

A lo largo de los capítulos restantes, veremos numerosos ejemplos de cómo las estructuras de datos dinámicas pueden usarse tanto para mejorar la organización de los datos como para hacer que ciertos cálculos sean más eficientes. Sin embargo, es importante tener en cuenta las compensaciones relativas. Como vimos con los arreglos y las listas enlazadas, cada estructura de datos tiene sus propias ventajas y desventajas en términos de flexibilidad, requisitos de espacio, eficiencia de operaciones y complejidad. En el próximo capítulo, mostraremos cómo podemos construir sobre estos conceptos fundamentales para crear dos estructuras de datos, pilas y colas, que permiten comportamientos diferentes.