3
DYNAMIC DATA STRUCTURES

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 1, 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 the limitations of static data structures. 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?"

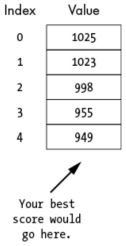

Figure 3-1: A five-element array holding high scores for a video game. None, alas, are yours.

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*.

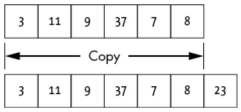

Figure 3-2: Adding an element to the end of a full array.

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_array
```

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 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*.

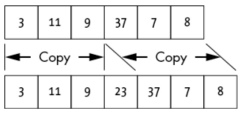

Figure 3-3: Adding an element to the middle of a full array

To address the shortfalls of arrays, we need to move to more flexible data struc-

tures 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*.

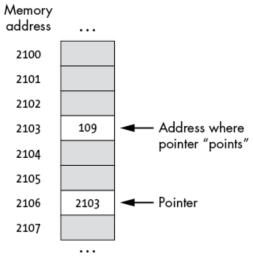

Figure 3-4: A pointer indicating an address in the computer's memory

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, filing 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 shown as a series of nodes 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 $K \times N$ bytes. In contrast, if each pointer requires another M bytes, our data structure now has a cost of $K \times (M + N)$ 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.

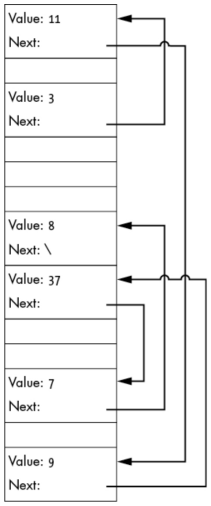

Figure 3-6: A linked list in the computer's memory. Nodes are not necessarily adjacent to each other.

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:

```
LinkedListLookUp(LinkedListNode: head, Integer: element_number)
  ❶ LinkedListNode: current = head
    Integer: count = 0

  ❷ WHILE count < element_number AND current != null:
        current = current.next
        count = count + 1
    return current
```

The code starts at the head of the list ❶. 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` ❷. 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.

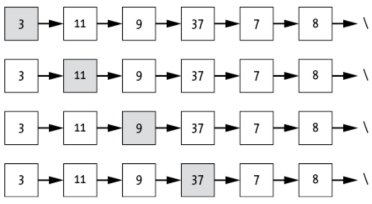

Figure 3-7: Traversing a linked list requires moving from one node to the next along the chain of pointers.

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 ac-

compute a single offset and look up the correct address in 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: Extending a linked list 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: Extending a linked list 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.

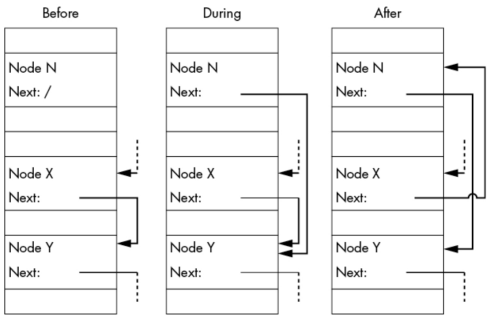

Figure 3-10: The process of inserting a new node N into a linked list between nodes X and Y

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*.

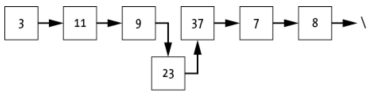

Figure 3-11: Inserting the node 23 into a linked list requires updating pointers from the previous node (9) and to the following node (37).

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 to 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 choose, 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.
 ❶ IF index == 0:
        LinkedListNode: new_head = LinkedListNode(value)
        new_head.next = head
        return new_head

    LinkedListNode: current = head
    LinkedListNode: previous = null
    Integer: count = 0
 ❷ 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.
 ❸ IF count < index:
        Produce an invalid index error.

 ❹ LinkedListNode: new_node = LinkedListNode(value)
    new_node.next = previous.next
    previous.next = new_node

 ❺ return head
```

The code for insertion starts with the special case of inserting a new node at `index = 0`, the beginning of the list ❶. 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 ❷. 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 ❸. 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 ❹. 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 ❾, 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*.

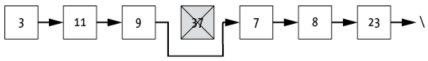

Figure 3-12: Removing a node (37) from a linked list requires updating the pointer in the previous node (9) to skip ahead to the following node (7).

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):
  ❶ IF head == null:
        return null

  ❷ IF index == 0:
        new_head = head.next
        head.next = null
        return new_head

    LinkedListNode: current = head
    LinkedListNode: previous = null
    Integer: count = 0
  ❸ WHILE count < index AND current != null:
        previous = current
        current = current.next
        count = count + 1

  ❹ IF current != null:
      ❺ previous.next = current.next
      ❻ current.next = null
    ELSE:
        Produce an invalid index error.
  ❼ return head
```

This code follows the same approach as insertion. This time we start with an additional check ❶. 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 ❷ 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 ❸. If the code finds a node at the correct index ❹, it splices out the node to be removed by setting `previous.next` to point at one node past the current node ❺. 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 ❻. The function completes by returning the address of the list's head node ❼.

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 ❸ 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 node given a pointer to it, 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*.

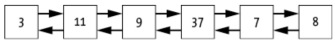

Figure 3-13: A doubly linked list contains pointers to both the next and previous entries.

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.

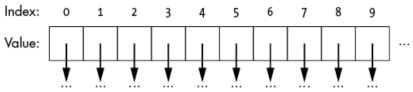

Figure 3-14: Arrays can store a series of pointers, allowing them to link to larger data structures.

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 re-

quirements, efficiency of operations, and complexity. In the next chapter, we will show how we can build on these fundamental concepts to create two data structures, stacks and queues, that enable different behavior.