Part II

Data Structures
Chapter 4

Basic Techniques

Algorithms must manipulate data, often in large quantities. Therefore, the way in which an algorithm stores and accesses data can have a significant impact on performance. The principles developed in the first three chapters apply to data structures, but in a somewhat different way than to algorithms. In this chapter, we will examine the ways in which top-down design, correctness proofs, and performance analysis can be applied to data structures. We will use rather simple data structures as examples. In succeeding chapters, we will apply these techniques to more involved structures.

4.1 Stacks

One of the strengths of both top-down design and object-oriented design is their use of abstraction to express high-level solutions to problems. In fact, we can apply abstraction to the problems themselves to obtain high-level solutions to many similar problems. Such high-level solutions are known as design patterns. For example, consider the “undo” operation in a word processor. We have some object that is undergoing a series of modifications. An application of the “undo” operation restores the object to its state prior to the last modification. Subsequent applications of “undo” restore the object to successively earlier states in its history.

We have captured the essence of the “undo” operation without specifying any of the details of the object being modified or the functionality of the document formatter in which it will appear. In fact, our description is general enough that it can apply to other applications, such as a spreadsheet or the search tree viewer on this book’s web site. We have therefore specified a design pattern for one aspect of functionality of an application.
Figure 4.1 The Stack ADT

**Precondition:** true.

**Postcondition:** Constructs an empty Stack.

\texttt{Stack()}

**Precondition:** true.

**Postcondition:** $a$ is added to the end of the represented sequence.

\texttt{Stack.Push(a)}

**Precondition:** The stack is nonempty.

**Postcondition:** The last element of the represented sequence is removed and returned.

\texttt{Stack.Pop()}

**Precondition:** true.

**Postcondition:** Returns true iff the represented sequence is empty.

\texttt{Stack.IsEmpty()}

We may apply the top-down approach to designing an implementation of the undo operation specified above. We first observe that we need to store a history of the edits applied to the edited object. These edits must be represented in such a way that we can undo or invert them. Furthermore, we only need last-in-first-out (LIFO) access to the history of edits. This suggests that the history of edits should be stored in a stack. We can then implement undo by popping the top edit from the stack, and applying its inverse to the edited object. If the stack is empty, then no undo is possible. Likewise, when edits are performed, each edit must be pushed onto the stack.

Let us now make these ideas a bit more formal. In order to be able to reason about this solution, we must have a formal definition of a stack and its operations. We therefore define a stack as a finite sequence of elements $\langle a_1, \ldots, a_n \rangle$, together with the operations specified in Figure 4.1. A mathematical structure, together with a specification of operations on that structure, is known as an abstract data type (ADT). The operations specified are the interface of the ADT. If we have a stack $S$, we refer to its Pop operation by $S.Pop()$. We refer to the Pop operation of the Stack ADT by \texttt{Stack.Pop()}. The \texttt{Stack()} operation is a specification of a constructor,
meaning that it is possible to construct an instance of any implementation without supplying any arguments.

In practice, we might want additional operations such as an operation that returns the top element without changing the sequence; however, the above operations are sufficient for now. Later, we will add to this set.

Note that the definition of an ADT is purely mathematical; i.e., it defines a stack as a sequence, which is a mathematical object. It says nothing about how a stack is to be implemented. Such a mathematical definition is sufficient for proving the correctness of \texttt{Undo}, shown in Figure 4.3, together with the specification of the \texttt{Editable} ADT in Figure 4.2. Such a proof is, in fact, trivial. Note that by specifying in the precondition that the stack is nonempty, we place on the caller the responsibility of checking this condition prior to calling \texttt{Undo}.

Continuing with the top-down approach, we need to design an implementation for the \texttt{Stack ADT}. In what follows, we will first consider a simple
approach that does not quite meet the specification. We will then consider
two full implementations of the Stack ADT.

4.2 A Simple Stack Implementation

The first step in designing an implementation for a data structure is to decide
upon a representation of the data. Perhaps the simplest representation of a
stack is an array storing the sequence of elements. Because we will want to
push additional elements onto the stack, we should use an array larger than
the number of elements. We therefore need a way to find the last element
of the sequence, or the top element of the stack. We can accomplish this by
keeping track of the number of elements in the stack.

Such a representation is a bit too simple, because the size of the array
limits the size of the stack — i.e., once we have constructed the array,
we have limited the size of the sequence we can represent. In order to
accommodate this shortcoming, we will associate with each of these stacks a
capacity, which gives the maximum number of elements it can hold. Later,
we will consider how this restriction might be removed. As a result of this
restriction, we must modify our specification of the Push operation so that
if \( n \) is strictly less than the stack’s capacity, then Push\((a)\) adds \( a \) to the end
of the sequence. In order to be able to check this condition, we will replace
the IsEmpty operation with a more general Size operation that returns the
number of elements in the stack.

Our representation, therefore, consists of:

- an Array \( \text{elements}[1..M] \) for some \( M \geq 0 \); and
- a Nat \( \text{size} \).

The value \( M \) above is not an explicit part of the representation, but we
assume that we can obtain its value by using the function SizeOf, specified
in Figure 1.20 on page 23.

In order to make sense of a given representation, we need an interpretation,
which relates a given set of values for the variables in the representation to a specific instance of the formal definition. The variables
used by the interpretation may include not only the representation variables, but also variables that may be accessed using the representation variables. In our present example, \( \text{elements}[1..\text{size}] \) describes the stack
\( \langle \text{elements}[1], \ldots, \text{elements}[\text{size}] \rangle \), and SizeOf\((\text{elements})\) gives the capacity
of the stack.
The above interpretation is problematic when size is outside the bounds of the array. We therefore need a mechanism to ensure that the values of a given representation are valid. To this end we use a structural invariant. This invariant is a statement about the values of the representation variables, and perhaps other variables that can be accessed using the representation variables. It should be true at virtually all times. The only exception is that we allow it to be temporarily violated while an operation is modifying the structure, provided that it is true by the time the operation completes. The structural invariant for our present example will be:

\[ 0 \leq \text{size} \leq \text{SIZEOF(elements)}. \]  

The values of the representation variables, together with all values used by the interpretation and the structural invariant, comprise the state of the data structure. Thus, the state of our stack implementation consists of the value of size, the array elements, and the values stored in elements[1..size]. (We will clarify shortly the distinction between the array and the values stored in the array.)

We can now complete our implementation by giving algorithms for the SimpleStack constructor and operations. These algorithms are shown in Figure 4.4.

Note that the preconditions and postconditions for the constructor and operations are stated in terms of the definition of a stack, not in terms of our chosen representation. For example, the precondition for the Push operation could have been stated as,

\[ \text{size} \leq \text{SIZEOF(elements)}. \]

However, preconditions and postconditions for operations on data structures should specify the externally observable behavior of the operation, and hence should not refer to the representation of the structure. Thus, the preconditions and postconditions still make sense even if we change the representation.

The performance of these operations can be analyzed using the techniques given in Chapter 3. It is easily seen that each of the operations Push, Pop, and Size operate in $\Theta(1)$ time. Analysis of the constructor is more problematic because we must include the time for constructing an array. This time depends on such factors as how the operating system allocates memory and whether the elements are initialized to some default value. For the sake of simplicity, we will assume that the memory can be allocated in constant time, and that the array will not be initialized. Thus,
Figure 4.4 The data type SIMPLESTACK, which does not quite implement the STACK ADT

**Structural Invariant:** $0 \leq \text{size} \leq \text{SIZEOF(elements)}$.

**Precondition:** \( \text{cap} \) is a NAT.

**Postcondition:** The constructed stack is empty, and its capacity is \( \text{cap} \).

\[
\text{SIMPLESTACK}(\text{cap})
\begin{align*}
\text{size} & \leftarrow 0; \text{elements} \leftarrow \text{new ARRAY}[1..\text{cap}] \\
\end{align*}
\]

**Precondition:** The number of elements in the represented sequence is strictly less than the capacity.

**Postcondition:** \( a \) is added to the end of the represented sequence.

\[
\text{SIMPLESTACK}.\text{Push}(a)
\begin{align*}
\text{if } \text{size} < \text{SIZEOF(elements)} & \text{ then}
\text{size} \leftarrow \text{size} + 1; \text{elements}[\text{size}] \leftarrow a \\
\text{else} & \text{ error}
\end{align*}
\]

**Precondition:** The represented sequence is nonempty.

**Postcondition:** The last element of the represented sequence is removed and returned.

\[
\text{SIMPLESTACK}.\text{Pop}()
\begin{align*}
\text{if } \text{size} > 0 & \text{ then}
\text{size} \leftarrow \text{size} - 1; \text{return elements}[\text{size} + 1] \\
\text{else} & \text{ error}
\end{align*}
\]

**Precondition:** true.

**Postcondition:** Returns the length of the represented sequence.

\[
\text{SIMPLESTACK}.\text{Size}()
\begin{align*}
\text{return } \text{size}
\end{align*}
\]
the time to construct a new array is in \( \Theta(1) \), and the constructor operates in \( \Theta(1) \) time.

Proving correctness of operations on a data structure is similar to proving correctness of ordinary algorithms. There are five parts:

1. **Initialization:** If the precondition holds at the beginning of a constructor invocation, then the postcondition holds upon completion of the constructor. If the constructor terminates normally, then the structural invariant holds after the data structure has been constructed, regardless of the truth of the precondition. (If the constructor terminates abnormally, i.e., with an error condition, then we assume the structure has not been constructed.)

2. **Maintenance:** If the structural invariant holds prior to the beginning of an operation, then it holds following completion of that operation.

3. **Security:** If the structural invariant holds, then the state can only be modified by invoking one of this structure’s operations.

4. **Termination:** Each operation and constructor terminates.

5. **Correctness:** If the structural invariant and the precondition hold prior to the beginning of an operation, then the postcondition holds following the completion of that operation.

We have already seen four of these five parts in proofs of algorithm correctness. Security is needed not only to make sure that malicious or untrusted code cannot violate the intended purpose of the data structure, but also to guarantee that the structural invariant is maintained between operations. In order to guarantee security, we need a mechanism for restricting access to the representation variables. Before we can discuss this mechanism, however, we first need to provide some details about the computational model we are assuming. We will tackle all of this shortly; however, let us first give an example of the other four parts via a correctness proof for SimpleStack. We will first state security as a lemma to be proved later, then we will show that SimpleStack meets its specification.

**Lemma 4.1** Security holds for SimpleStack.

**Theorem 4.2** SimpleStack meets its specification.
CHAPTER 4. BASIC TECHNIQUES

Proof: We must show initialization, maintenance, security, termination, and correctness.

Initialization: First, suppose the precondition to the constructor is met; i.e., that $cap$ is a Nat. Then the constructor will terminate normally with $size = 0$, which we interpret as meaning the represented sequence is empty. Because we interpret $\text{SizeOf}(\text{elements})$, which is $cap$, to be the capacity of the stack, the postcondition is therefore met. Furthermore, regardless of the truth of the precondition, if the constructor terminates without error, $size$ will be 0 and $\text{elements}$ will refer to an array. Because an array cannot have a negative number of elements, its size is at least 0. Therefore, the structural invariant holds ($0 \leq size \leq \text{SizeOf}(\text{elements})$).

Maintenance: Suppose the structural invariant holds prior to the execution of an operation. We will only consider the operation $\text{Push}(a)$; proofs for the other two operations are left as an exercise.

The size of $\text{elements}$ is not changed by this operation. The value of $size$ is only changed if it is strictly less than the size of $\text{elements}$. In this case, because it is incremented by 1, the value of $size$ will remain nonnegative, but will not exceed the size of $\text{elements}$. The structural invariant therefore holds after this operation completes.

Security: Follows from Lemma 4.1.

Termination: Because there are no loops or recursive calls, all constructors and operations terminate.

Correctness: Suppose the structural invariant holds prior to the execution of an operation. We will only consider the operation $\text{Push}(a)$; proofs for the other two operations are left as an exercise.

Suppose the precondition holds, i.e., that the number of elements in the stack is strictly less than the stack’s capacity. Because we interpret the size of $\text{elements}$ to be the stack’s capacity and the value of $size$ to be the number of elements in the stack, it follows that $size < \text{SizeOf}(\text{elements})$. The if condition is therefore true. $size$ is therefore incremented by 1, which we interpret as increasing the number of elements on the stack by 1. $a$ is then assigned to $\text{elements}[size]$, which we interpret as the last element of the represented sequence. The postcondition is therefore met. \[\square\]

As the above proof illustrates, initialization, maintenance, termination,
and correctness can be shown using the techniques introduced in Chapter 2, although the specific statements to be shown are somewhat different. Proving security, on the other hand, not only uses different techniques, it also requires a more detailed specification of the underlying computational model that we are assuming. We have chosen a model that is reasonably simple and consistent, and which may be implemented easily in a variety of programming languages. In what follows, we will give its details.

One characteristic of our model involves the way in which data items are associated with variables. With each data item, there is a reference that uniquely identifies it (i.e., two distinct references may not identify the same data item). Furthermore, a reference may not itself be a data item; i.e., a reference may not refer to another reference. It is the reference, not the data item itself, that will be stored in a variable. We do not specify anything else regarding the reference, but often a reference will be implemented as the address in memory at which the data item resides. However, we will assume that a constant like the integer 3 also has a reference. In this case the reference may simply be the binary encoding of 3.

Such a distinction between a data constant and its reference may seem artificial, but it allows for a uniform treatment of variables and data. Thus, when variable assignments are made, the reference to the assigned data item is stored in the modified variable. Likewise, when formal parameters take their values from actual parameters, the references are copied from the actual parameters to the formal parameters.

Given the distinction between a data item and its reference, we can now define more precisely the state of a SimpleStack. It must first include the values of the reference variables, namely size (a reference to an integer) and elements (a reference to an array). Because the interpretation uses the values of elements[1..size], these values, which are references to arbitrary data items, are also part of the state. However, the values of the data items to which the references in elements[1..size] refer are not included in the state. Thus, if elements[size] contains a reference to an array A, that reference is a part of the state of the stack, but the contents of A are not. In particular, if the value stored in elements[size] changes, then the stack contents change, but if the contents of A change, the stack contents do not change — A is still the item at the top of the stack.

Our model uses a simple hierarchical type system. Each implementation has a unique type. This type may be a subtype of one or more interfaces (i.e., ADTs) that it implements. Thus, if an implementation A implements ADTs B and C, then any instance of type A also belongs to types B and C. We do not allow implementations to be subtypes of other implementations, so our
CHAPTER 4. BASIC TECHNIQUES

model includes no inheritance. Our algorithms will not always specify the type of a data item if its type is irrelevant to the essence of the algorithm. For example, we have not specified the type of the parameter \( a \) for the \texttt{STACK.Push} operation in Figure 4.4 because we do not care what kind of data is stored in the stack.

When the data type of a parameter is important, we can specify it in the precondition, as we have done for the constructor in Figure 4.4. Unless we explicitly state otherwise, when we state in a precondition that a variable refers to an item of some particular type, we mean that this variable must be non-nil. Note, however, that a precondition does not affect the execution of the code. When it is important that the type actually be checked (e.g., for maintaining a structural invariant), we will attach a type declaration in the parameter list, as in the two-argument constructor in Figure 4.10 (page 127). A type declaration applies to a single parameter only, so that in this example, \( L \) is of type \texttt{ConsList}, but \( a \) is untyped. We interpret a type declaration as generating an error if the value passed to that parameter is not \texttt{nil} and does not refer to an instance of the declared type.

As we have already suggested, the elements of a particular data type may have operations associated with them. Thus, each instance of the \texttt{STACK} type has a \texttt{Push} operation and a \texttt{Pop} operation. For the sake of consistency, we will consider that when a constructor is invoked, it belongs to the data item that it is constructing. In addition, the elements of a data type may have \textit{internal functions} associated with them. Internal functions are just like operations, but with restricted access, as described below.

In order to control the way in which a data structure can be changed, we place the following restrictions on how representation variables and internal functions can be accessed:

- Write access to a representation variable of an instance of data type \( A \) is given only to the operations, constructors, and internal functions of that instance.

- Read access to a representation variable of an instance of data type \( A \) is given only to operations, constructors, and internal functions of instances of type \( A \).

- Access to an internal function of an instance of a data type \( A \) is given only to operations, constructors, and internal functions of that instance.

These restrictions are severe enough that we will often need to relax them. In order to relax either of the first two restrictions, we can provide
**accessor operations.** Because we frequently need to do this, we will adopt some conventions that allow us to avoid cluttering our algorithms with trivial code.

- If we want to provide read access to a variable \( \text{var} \) in the representation of type \( \text{A} \), we define the operation \( \text{A.VAR()} \), which simply returns \( \text{var} \). Using this convention, we could have omitted operation \( \text{SimpleStack.Size()} \) from Figure 4.4.

- If we want to provide write access to \( \text{var} \), we define the operation \( \text{A.SETVAR}(x) \), which assigns the value of \( x \) to \( \text{var} \).

Explicitly allowing write access does not technically violate security, because any changes are made by invoking operations of the data structure. What can be problematic is allowing read access. For example, suppose we were to allow read access to the variable \( \text{elements} \) in the representation of a stack. Using this reference, a user’s code could change the contents of that array. Because this array’s contents belong to the state of the data structure, security would then be violated. We must therefore check for the following conditions, each of which might compromise security:

- An operation returns a reference to a portion of the state of the structure. This condition can include an operation that gives explicit read access to a representation variable. This condition will violate security if the reference refers to a data item whose value can change.

- An operation causes the data item to which one of its parameters refers to be a part of the state of the structure. Under this condition, the code that invokes the operation has a copy of the parameter, and hence has access to the state of the structure. If the data item in question can be changed, security is violated.

- A reference to a portion of the state is copied to the state of another instance of the same type. For example, if \( S \) and \( T \) are of type \( \text{SimpleStack} \), and their \( \text{elements} \) variables have the same value, then the operation \( S.\text{Push}(x) \) could change the contents of \( T \). Thus, if a shared data item can be changed, security is violated.

We can now illustrate the technique of proving security by proving Lemma 4.1.
Proof of Lemma 4.1: Read access is explicitly given to \textit{size}. However, \textit{size} refers to a \texttt{Nat}, which cannot be changed. The only other values returned are references to elements stored in the stack, and the values of these elements are not part of the state of the stack. Likewise, the only parameters are the capacity and the parameter to \texttt{Push}, neither of which refers to a data item that becomes a part of the state of the stack. Finally, no operations copy any part of the state of a \texttt{SimpleStack} to the state of another \texttt{SimpleStack}. We therefore can conclude that \texttt{SimpleStack} is secure. □

Designing secure data structures is sometimes rather challenging. In fact, there are occasions when security becomes too much of a burden — for example, when we are designing a data structure to be used only locally within some algorithm. In such a case, it may be easier to prove that our algorithm doesn’t violate the security of the data structure, rather than to prove that such a violation is impossible. If we define a data structure for which we can prove initialization, maintenance, termination, and correctness, we say that this structure is \textit{insecure}, but otherwise satisfies its specification.

Together, initialization, maintenance, and security are almost sufficient to prove that the structural invariant holds between execution of any operations on the structure. The only caveat is similar to the difficulty associated with mutual recursion, as was discussed in Section 2.5. Suppose that during execution of some operation \texttt{x.Op1}, a function call is made while the structural invariant is false. Suppose that through some sequence of nested calls, some operation \texttt{x.Op2} is then called (see Figure 4.5). At this point, the structural invariant is false, and the operation’s correctness cannot be guaranteed. This scenario is known as a \textit{callback}. Note that it does not matter how long the sequence of nested calls is. In particular, the function call made by \texttt{x.Op1} may be a direct call to \texttt{x.Op2}, or it may be a recursive call.

Though callbacks are common in software systems, they are more problematic than beneficial in the design of data structures. Furthermore, as is the case for mutual recursion, callbacks may be impossible to detect when we are designing a single data structure, as we may need to call a function whose implementation we don’t have. For these reasons, we will assume that if a callback is attempted, a runtime error results. Our correctness proofs will then rest on the assumption that data structures and algorithms are not combined in such a way as to result in a callback.

Given this assumption, once initialization, maintenance, and security are shown, it can be shown by induction that the structural invariant holds
Figure 4.5 Illustration of a callback — when $x$.Op2() is called, the structural invariant of $x$ is false.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{callback_diagram.png}
\caption{Illustration of a callback — when $x$.Op2() is called, the structural invariant of $x$ is false.}
\end{figure}
between execution of any operations that construct or alter an instance of the data structure. Note that the structural invariant will hold after the structure is constructed, regardless of whether preconditions to operations are met. Thus, we can be convinced that a structural invariant holds even if operations are not invoked properly. Because we can know that the structural invariant holds, we can then use it in addition to the precondition in proving the correctness of an individual operation.

4.3 Expandable Arrays

SimpleStack does not quite implement the Stack ADT because each SimpleStack must be constructed with a fixed capacity. We will now present a true implementation Stack.

We can use the basic idea from SimpleStack if we have a way to perform a Push when the array is full. Our solution will be to construct a new, larger array, and to copy the elements in the stack to this new array. This new array then replaces the original array. We now have room to add the new element.

Though this idea is simple, there are some performance issues to consider. In particular, \( \Theta(n) \) time is required to copy \( n \) elements from one array to another. We might be willing to pay this performance price occasionally, but if we are not careful, the overall performance of manipulating a stack may be significantly degraded. Suppose, for example, that the new array contains only one more location than the original array. Consider what happens if we push \( n \) elements onto such a stack, where \( n \) is much larger than the size of the original array. After the original array is filled, each Push requires \( \Theta(i) \) time, where \( i \) is number of elements currently in the stack. It is not hard to see that the total time for pushing all \( n \) elements is in \( \Theta(n^2) \), assuming the size of the original array is a fixed constant.

In order to avoid this bad performance, when we need to allocate a new array, we should make sure that it is significantly larger than the array we are replacing. As we will see shortly, we can achieve this goal by doubling the size of the array. The ExpandableArrayStack implementation shown in Figure 4.6 implements this idea. In order for this to work, however, the size of the array must always be nonzero; hence, we will need to include this restriction in the structural invariant. Note that we have added a constructor that was not specified in the interface (Figure 4.1). The no-argument constructor simply invokes this new constructor with a default value for its argument.
Figure 4.6 ExpandableArrayStack implementation of Stack

Structural Invariant: $0 \leq \text{size} \leq \text{SIZEOF}(\text{elements})$, and $\text{SIZEOF}(\text{elements}) > 0$.

ExpandableArrayStack()
ExpandableArrayStack(10)

Precondition: $\text{cap}$ is a strictly positive Nat.
Postcondition: An empty stack is constructed.

ExpandableArrayStack(cap)
-- if $\text{cap} > 0$
    $\text{size} \leftarrow 0$; $\text{elements} \leftarrow \text{new Array}[1..\text{cap}]$
-- else
    error

ExpandableArrayStack.Push($a$)
-- if $\text{size} = \text{SIZEOF}(\text{elements})$
    $\text{el} \leftarrow \text{new Array}[1..2 \cdot \text{size}]$
    for $i \leftarrow 1$ to $\text{size}$
        $\text{el}[i] \leftarrow \text{elements}[i]$
    $\text{elements} \leftarrow \text{el}$
    $\text{size} \leftarrow \text{size} + 1$; $\text{elements}[\text{size}] \leftarrow a$

ExpandableArrayStack.Pop()
-- if $\text{size} > 0$
    $\text{size} \leftarrow \text{size} - 1$
    return $\text{elements}[\text{size} + 1]$
-- else
    error

ExpandableArrayStack.IsEmpty()
return $\text{size} = 0$
At this point it is tempting to apply the top-down design principle by defining an ADT for an expandable array. However, when an idea is simple enough, designing an ADT often becomes more cumbersome than it is worth. To attain the full functionality of an expandable array, we would need operations to perform each of the following tasks:

- reading from and writing to arbitrary locations;
- obtaining the current size of the array; and
- explicitly expanding the array (we might want to do this before the array is full) to a size we choose.

Furthermore, we might wish to redistribute the data in the larger array when we expand it. It therefore seems best to characterize the expandable array design pattern as the practice of moving data from one array to a new one of at least twice the size whenever the current array becomes too full.

Clearly, the worst-case running time of the \texttt{Push} operation shown in Figure 4.6 is in $O(n)$, where $n$ is the number of elements in the stack. Furthermore, for any $n > 0$, if we construct a stack with the constructor call \texttt{ExpandableArrayStack($n$)}, then when the $(n + 1)$-st element is pushed onto the stack, $\Omega(n)$ time is required. Therefore, the worst-case running time of the \texttt{Push} operation is in $\Theta(n)$. All other operations clearly require $\Theta(1)$ time.

The above analysis seems inadequate because in any actual use of a stack, the $\Theta(n)$ behavior will occur for only a few operations. If an \texttt{ExpandableArrayStack} is used by some algorithm, the slow operations may be few enough that they do not significantly impact the algorithm’s overall performance. In such a case, it makes sense to consider the worst-case performance of an entire sequence of operations, rather than a single operation. This idea is the basis for \textit{amortized analysis}.

With amortized analysis, we consider an arbitrary sequence of operations performed on an initially empty data structure. We then do a form of worst-case analysis of this sequence of operations. Clearly, longer sequences will have longer running times. In order to remove the dependence upon the length of the sequence, we \textit{amortize} the total running time of the sequence over the individual operations in the sequence. For the time being, this amortization will simply be to compute the average running time for an individual operation in the sequence; later in this chapter we will generalize this definition. The analysis is still worst-case because the sequence is arbitrary — we are finding the worst-case amortized time for the operations on the data structure.
For example, consider any sequence of $n$ operations on an initially empty stack constructed with ExpandableArrayStack($k$). We first analyze the worst-case running time of such a sequence. We can use the techniques presented in Chapter 3, but the analysis is easier if we apply a new technique. We first analyze the running time ignoring all iterations of the for loop in the Push operation and any loop overhead that results in the execution of an iteration. Having ignored this code, it is easily seen that each operation requires $\Theta(1)$ time, so that the entire sequence requires $\Theta(n)$ time.

We now analyze the running time of all iterations of the for loop throughout the entire sequence of operations. In order to accomplish this, we must compute the total number of iterations. The array will be expanded to size $2k$ the first time the stack reaches size $k + 1$. For this first expansion, the for loop iterates $k$ times. The array will be expanded to size $4k$ the first time the stack reaches size $2k + 1$. For this expansion, the loop iterates $2k$ times. In general, the array will be expanded to size $2^{i+1}k$ the first time the stack reaches size $2^i k + 1$, and the loop will iterate $2^i k$ times during this expansion. Because the sequence contains $n$ operations, the stack can never exceed size $n$. Therefore, in order to compute an upper bound on the total number of iterations, we must sum $2^i k$ for all $i \geq 0$ such that

$$
2^i k + 1 \leq n \\
2^i \leq (n - 1)/k \\
i \leq \lg(n - 1) - \lg k.
$$

The total number of iterations is therefore at most

$$
\sum_{i=0}^{\lfloor \lg(n-1)-\lg k \rfloor} 2^i k = k \sum_{i=0}^{\lfloor \lg(n-1)-\lg k \rfloor} 2^i
$$

$$
= k(2^{\lfloor \lg(n-1)-\lg k \rfloor + 1} - 1) \quad \text{by (2.2)}
$$

$$
\leq k 2^{\lg(n-1) - \lg k + 1}
$$

$$
= k 2^{\lg(n-1)+1}
$$

$$
\leq \frac{2^{\lg k + 1}}{2^{\lg k}}
$$

$$
= 2(n - 1).
$$

Because each loop iteration requires $\Theta(1)$ time, the time required for all loop iterations is in $O(n)$. Combining this result with the earlier analysis that ignored the loop iterations, we see that the entire sequence runs in $\Theta(n)$ time.
Now to complete the amortized analysis, we must average the total running time over the \( n \) operations in the sequence. By Exercise 3.6 on page 97, if \( f(n) \in \Theta(n) \), then \( f(n)/n \in \Theta(1) \). Therefore, the worst-case amortized time for the stack operations is in \( \Theta(1) \). We conclude that, although an individual \texttt{Push} operation may be expensive, the expandable array yields a stack that performs well on any sequence of operations starting from an initially empty stack.

### 4.4 The ConsList ADT

On this textbook’s web site is a search tree viewer that allows users to insert and remove strings from various kinds of search trees and to view the results. Included are “Back” and “Forward” buttons that allow the user to step through the history of the trees created. Two stacks are used, one to store the history, and one to store the “future” after the user has stepped back into the history. Also included is a “Clone” button, which causes an identical window to be created, with identical history and future. This new window can be manipulated independently from the first. In order to accomplish this independence, the two stacks must be cloned.

Let us consider how we might clone an \texttt{ExpandableArrayStack}. In order to simplify the discussion, we will restrict our attention to \textit{shallow cloning}. Shallow cloning consists of cloning only the state of the structure, and not the items contained in the structure. Thus, if a data item is stored in a stack which is then cloned, any subsequent changes to that data item will be reflected in both stacks. However, changes to one of the stacks will not affect the other. In order to perform a shallow clone of an \texttt{ExpandableArrayStack}, the array must clearly be copied, so that the two stacks can be manipulated independently. Copying one array to another requires \( \Theta(n) \) time, where \( n \) is the number of elements copied.

We might be able to improve on this running time if we can use a data structure that facilitates \textit{nondestructive updates}. An update is said to be \textit{nondestructive} if it does not change any of the existing structure, but instead builds a new structure, perhaps using some or all of the existing structure. If all updates are nondestructive (i.e., the structure is \textit{immutable}), it is possible for different structures to share substructures that are common to both. This sharing can sometimes lead to improved efficiency; for example, to clone an immutable structure all that we need to copy is the reference to it.

In order to apply this idea to stacks, it is helpful to think of a finite
sequence as nested ordered pairs. In particular, a sequence of length \( n > 0 \)
is an ordered pair consisting of a sequence of length \( n - 1 \) followed by an
element. As a special case, the sequence of length 0 is denoted \( () \). Thus,
the sequence \( \langle a_1, a_2, a_3 \rangle \) can be thought of as the pair \( (((()), a_1), a_2), a_3) \). If
we think of this sequence as a \textit{Stack} \( S \), then we can think of \( S \text{push}(a_4) \)
as a function returning a new sequence \( (((()), a_1), a_2), a_3, a_4) \). Note that
this new sequence can be constructed simply by pairing \( S \) with \( a_4 \), leaving
\( S \) unchanged.

Nested pairs form the basic data structure in the programming language
Lisp and its derivatives. The Lisp function to build an ordered pair is called
\textit{cons}. Based on this background is the ADT known as a \textit{ConsList}. It is
useful to think of a nonempty \textit{ConsList} as a pair \( \langle \text{head}, \text{tail} \rangle \), where \text{head}
is an element and \text{tail} is a \textit{ConsList}. (Note that the two components of
the pair are in the reverse order of that described in the above paragraph.)

More formally, we define a \textit{ConsList} to be a finite sequence \( \langle a_1, \ldots, a_n \rangle \),
together with the operations specified in Figure 4.7. Note that none of these
operations changes the \textit{ConsList}. We therefore say that a \textit{ConsList} is an
immutable structure, meaning that though the elements in the sequence may
change their state, the sequence itself will not change.

In what follows, we will show how to implement \textit{Stack} using a \textit{ConsList}. We will have thus applied top-down design to the task of implementing
\textit{Stack}, as we will have reduced this problem to the problem of implementing
\textit{ConsList}. The resulting \textit{Stack} implementation will support constant-time
\textit{push}, \textit{pop}, and shallow cloning, which we will support via an additional
constructor. We will then complete the design by showing how to implement
\textit{ConsList}.

Our \textit{Stack} representation will be a \textit{ConsList} \textit{elements}. We interpret
\textit{elements} as storing the stack in reverse order; i.e., the head of \textit{elements} is
the top element on the stack. The structural invariant will be that \textit{elements}
refers to a \textit{ConsList}. The implementation is shown in Figure 4.8. The
one-argument constructor is used to construct a shallow clone.

Again, all constructors and operations can easily be seen to run in \( \Theta(1) \)
time, and proving initialization, maintenance, termination, and correctness
is straightforward. Regarding security, we note that \textit{elements} is shared when
the one-argument constructor is used; however, because a \textit{ConsList} is im-
mutable, this sharing cannot violate security. The full correctness proof is
left as an exercise.

We will now complete the implementation of \textit{ConsListStack} by im-
plementing \textit{ConsList}. Our representation consists of the following:
Figure 4.7 The ConsList ADT

**Precondition:** true

**Postcondition:** Constructs a ConsList representing an empty sequence.

ConsList()

**Precondition:** L is a ConsList \(a_1, \ldots, a_n\).

**Postcondition:** Constructs a ConsList representing the sequence \(a, a_1, \ldots, a_n\).

ConsList\(a, L\)

**Precondition:** true.

**Postcondition:** Returns a Bool that is true iff the represented sequence is empty.

ConsList.IsEmpty()

**Precondition:** The represented sequence is nonempty.

**Postcondition:** Returns the first element of the sequence.

ConsList.Head()

**Precondition:** The represented sequence \(a_1, \ldots, a_n\) is nonempty.

**Postcondition:** Returns a ConsList representing the sequence \(a_2, \ldots, a_n\).

ConsList.Tail()

- a readable Bool `isEmpty`;
- a readable element `head`; and
- a readable ConsList `tail`.

If `isEmpty` is true, then we interpret the ConsList to represent an empty sequence. Otherwise, we interpret `head` as the first element of the sequence, and `tail` as the remainder of the sequence. As our structural invariant, we require a ConsList to represent a finite sequence according to the above interpretation. The representation of the ConsList \(a_1, a_2, a_3, a_4\) is illustrated in Figure 4.9.

Because our representation fits so closely with the definition of a ConsList, the implementation is trivial — the operations are simply the acces-
Figure 4.8 ConsListStack implementation of Stack

**Structural Invariant:** elements refers to a ConsList.

ConsListStack()
   elements ← new ConsList()

**Precondition:** S refers to a ConsListStack.

**Postcondition:** The constructed stack is a shallow clone of the stack S.

ConsListStack(S)
   elements ← S.elements

ConsListStack.Push(a)
   elements ← new ConsList(a, elements)

ConsListStack.Pop()
   if elements.IsEmpty()
      error
   else
      top ← elements.Head(); elements ← elements.Tail()
      return top

ConsListStack.IsEmpty()
   return elements.IsEmpty()

Figure 4.9 An illustration of the representation of a ConsList

```
  a1 → a2 → a3 → a4
```
Figure 4.10 Implementation of CONSList

**Structural Invariant:** The CONSList represents a finite sequence.

CONSList()

\[ \text{isEmpty} \leftarrow \text{true} \]

CONSList(a, L : CONSList)

\[ \begin{align*}
\text{if } L &= \text{nil} \\
\text{error}
\text{else}
\text{isEmpty} &\leftarrow \text{false}; \text{head} \leftarrow a; \text{tail} \leftarrow L
\end{align*} \]

sors for the three representation variables. Note that our specification says nothing about the contents of head and tail when isEmpty is true; hence, if these accessors are called for an empty list, arbitrary values may be returned. The implementation is shown in Figure 4.10. Because we will only present a single implementation of CONSList, we use the same name for the implementation as for the interface.

It is easily seen that each constructor and operation runs in \( \Theta(1) \) time. We will now prove that the implementation meets its specification.

**Theorem 4.3** The CONSList implementation meets its specification.

**Proof:**

**Initialization:** We consider the two constructors as separate cases.

**Case 1:** CONSList(). Because isEmpty is set to true, we interpret the constructed CONSList as representing an empty sequence. The structural invariant and postcondition therefore hold.

**Case 2:** CONSList(a, L : CONSList). If this constructor terminates normally, then L refers to a CONSList. Let us therefore assume that this is the case. Let L represent the sequence \( \langle a_1, \ldots, a_n \rangle \). isEmpty is set to false, so the constructed instance is interpreted to be the nonempty sequence \( \langle a, a_1, \ldots, a_n \rangle \). Because this is a finite sequence, the structural invariant
and postcondition both hold.

**Maintenance:** Because no operations change any representation variables, maintenance holds trivially.

**Security:** Read access is explicitly given to the three representation variables. However, is\textit{Empty} and \textit{tail} are immutable, so this read access cannot result in changes to either of them. Because \textit{head} refers to a data item that is not a part of the state, changes that may result from reading this reference do not affect the security of the ConsList. Finally, although the parameter \textit{L} to the two-argument constructor is copied to a representation variable, because it refers to an immutable data item, security is not violated.

**Termination:** Because there are no loops or recursion, all constructors and operations terminate.

**Correctness:** The only operations simply provide read access, and so are trivially correct.

Example 4.4 Suppose we construct an empty ConsListStack \textit{S}, then push data items \textit{a}_1, \textit{a}_2, and \textit{a}_3 in sequence onto \textit{S}. Figure 4.11(a) illustrates the result of these operations. Suppose we then construct \textit{T} using ConsListStack(\textit{S}). At this point \textit{T}.\textit{elements} is equal to \textit{S}.\textit{elements}. If we then execute \textit{T}.\textit{Pop}() twice, \textit{T}.\textit{elements} is assigned \textit{T}.\textit{elements}.\textit{Tail}().\textit{Tail}(), as shown in Figure 4.11(b). Notice that this does not affect the contents of \textit{S}. If we then push \textit{a}_4 onto \textit{T}, we obtain the result shown in Figure 4.11(c). Again, the contents of \textit{S} are unchanged.

We conclude our discussion of ConsLists by noting that there are some disadvantages to this implementation of Stack. The running time of the \textit{Push} operations is likely to be slower than either of the other implementations because new memory is always allocated. Furthermore, this memory is never explicitly released, so this implementation should only be coded in a language that provides automatic garbage collection.

The idea behind a ConsList can be modified to form a mutable structure if we allow the value of \textit{tail} to be modified. It is difficult to define a general-purpose ADT based on this idea other than to allow write access to \textit{tail}. If we do this, then there is little security; i.e., the user can construct complicated linked structures that share data or perhaps form loops. Never-
theless, if we are careful, we can use this idea as a building block for several more advanced data structures. We will therefore refer to this idea as the linked list design pattern.

4.5 Amortized Analysis Using Potential Functions

In Section 4.3, we introduced the technique of amortized analysis. The actual analysis was rather straightforward, mainly because the worst case is easily identifiable. For many data structures, amortized analysis is not so straightforward. Furthermore, we would like to be able to amortize in a more general way than simply averaging over all operations in a sequence. Specifically, we would like to be able to amortize in such a way that operations on small structures receive smaller amortized cost than operations on large structures. For example, if $n$ represents the size of a structure, we would like to be able to speak about an amortized running time in $O(lg \, n)$. In this section, we introduce a more general notion of amortized cost and present a corresponding technique for performing amortized analysis.

In order to motivate this technique, it is helpful to think of amortized analysis using an analogy. Suppose we have a daily budget for gasoline for
a car. We want to track our gasoline purchases to ensure that we don’t exceed this budget. Further suppose that we begin tracking these expenses when the tank is full. We may then have several days in which we spend no money on gasoline. At some point, we will refill the tank, thus incurring a large expense which may be greater than our daily budget. However, if we amortize this cost over all of the days since we last filled the tank, we will hopefully find that we have remained within our daily budget for each of these days.

One way to monitor our budget more closely is to consider the potential cost of filling the tank at the end of each day. Specifically, suppose that we have a very precise gas gauge on our car. In order to keep the analogy simple, we will also suppose that the cost of gasoline remains constant. At the end of each day, we could then measure the amount of gasoline in the tank and compute the cost of potentially filling the tank at that point.

For example, suppose gasoline costs $3 per gallon. Further suppose that on consecutive days, we find that our 10-gallon tank contains 8 gallons and 6.5 gallons, respectively. On the first day, the potential cost of filling the tank was $6, as the tank would hold 2 additional gallons (see Figure 4.12). On the second day, the potential cost was $10.50, as the tank would hold 3.5 additional gallons. Assuming that no gasoline was added to the tank that day, the cost of the gasoline used that day was then $4.50 — the difference in the two potential costs.

On days in which we fill the tank, the computation is only slightly more complicated. Note that on these days, the potential cost is likely to decrease. For example, suppose that the previous day’s level (day 4 in Figure 4.12) was 3 gallons, today’s level is 9 gallons, and that we spent $24 to purchase 8 gallons of gasoline. The potential cost of filling the tank has decreased from $21 to $3; hence, the change in potential cost is negative $18. However, we should include the cost of the gasoline we actually purchased, resulting in an amortized cost of $6 for that day. In general, we compute the amortized cost by adding the actual cost to the change in potential cost.

Note that this amortization process is somewhat pessimistic, as we are assessing costs before we actually incur them; however, it is a safe way of verifying our budget. Specifically, suppose we sum up the amortized costs for any sequence of days, beginning with a day in which the tank is full. The sum of changes in potential costs will be the net change in potential cost. Because the tank is initially full, the initial potential cost is 0; hence the net change in potential cost is the final potential cost. The remainder of the sum of amortized costs is the sum of actual costs of gasoline purchases. Thus, the sum of amortized costs is the sum of actual costs plus the final
potential cost. Because the potential cost can never be negative (the tank can’t be “overfull”), the sum of the amortized costs will be at least the sum of the actual costs.

Let us now consider how we might apply this technique to the amortized analysis of a data structure such as an ExpandableArrayStack. The potential gasoline cost is essentially a measure of how “bad” the state of the gas tank is. In a similar way, we could measure how “bad” the state of an ExpandableArrayStack is by considering how full the array is — the closer the array is to being filled, the closer we are to an expensive operation. We can formalize this measure by defining a potential function $\Phi$, which maps states of a data structure into the nonnegative real numbers, much like the potential gasoline cost maps “states” of the gas tank into
nonnegative real numbers.

As we assumed our potential gasoline cost to be initially 0, so also we require that $\Phi$ maps the initial state (usually an empty data structure) to 0. Each operation, by changing the state of the data structure, also changes the value of $\Phi$. An increase in $\Phi$ corresponds to using gasoline, whereas a decrease in $\Phi$ corresponds to adding gasoline to the tank (though not necessarily filling it, as $\Phi$ might not reach 0). Thus, for ExpandableArrayStack, we would want the potential function to increase when a Push that does not expand the array is performed, but to decrease when either a Pop or a Push that expands the array is performed.

Let $\sigma$ denote the state of a data structure prior to some operation, and let $\sigma'$ denote the state of that structure following the operation. We then define the change in $\Phi$ to be $\Phi(\sigma') - \Phi(\sigma)$. We further define the amortized cost of the operation relative to $\Phi$ to be the actual cost plus the change in $\Phi$.

The above defines what a potential function is and suggests how it might be used to perform amortized analysis. It does not, however, tell us precisely how we can obtain a potential function. We will address this issue in detail shortly; for now however, we will show that an amortized analysis using any valid potential function will give a true upper bound on amortized cost. The proof is essentially the same argument that we gave justifying our use of the potential gasoline cost for amortized analysis.

**Theorem 4.5** Let $\Phi$ be a valid potential function for some data structure; i.e., if $\sigma_0$ is the initial state of the structure, then $\Phi(\sigma_0) = 0$, and if $\sigma$ is any state of the structure, then $\Phi(\sigma) \geq 0$. Then for any sequence of operations from the initial state $\sigma_0$, the sum of the amortized costs of the operations relative to $\Phi$ is at least the sum of the actual costs of the operations.

**Proof:** Let $o_1, \ldots, o_m$ be a sequence of operations from $\sigma_0$, and let $\sigma_i$ be the state of the data structure after operation $o_i$ has been performed. Also,
let $c_i$ be the actual cost of $o_i$ applied to state $\sigma_{i-1}$. Then

$$
\sum_{i=1}^{m} (c_i + \Phi(\sigma_i) - \Phi(\sigma_{i-1})) = \sum_{i=1}^{m} c_i + \sum_{i=1}^{m} \Phi(\sigma_i) - \sum_{i=0}^{m-1} \Phi(\sigma_i)
$$

$$
= \sum_{i=1}^{m} c_i + \Phi(\sigma_m) - \Phi(\sigma_0)
$$

$$
\geq \sum_{i=1}^{m} c_i
$$

because $\Phi(\sigma_m) \geq 0$ and $\Phi(\sigma_0) = 0$. \qed

This notion of amortized cost is therefore meaningful and more general than the one introduced in Section 4.3. Specifically, because an amortized cost is defined for each operation, we can analyze this cost much like we would analyze the actual running time of an operation. Note, however, that this notion of amortization only provides an upper bound. For this reason, we only use $O$-notation (not $\Omega$ or $\Theta$) when we perform this type of analysis.

This technique can now be used to analyze the amortized performance of ExpandableArrayStack. However, finding an appropriate potential function for this analysis turns out to be a bit tricky. As Theorem 4.5 implies, we can perform an amortized analysis using any valid potential function; however, a poor choice of potential function may result in a poor upper bound on the amortized cost. For example, we could choose as our potential function the constant function 0 — i.e., for each state, the potential is 0. This function meets the requirements of a potential function; however, because the change in potential will always be 0, the amortized cost relative to this potential function is the same as the actual cost. Finding a potential function that yields constant amortized cost for the ExpandableArrayStack operations requires a bit of insight.

For this reason, before we give a potential-function analysis for ExpandableArrayStack, we will begin with a simpler example. In particular, consider the ADT BinaryCounter, specified in Figure 4.13. A BinaryCounter maintains a value that is initially 1. The Increment operation can be used to increment the value by 1, and the Value operation can be used to retrieve the value as a ConsList of 1s and 0s, least significant bit first. It is unlikely that this ADT has any useful purpose; however, the implementation shown in Figure 4.14 yields an amortized analysis that is simple enough to illustrate clearly the potential-function technique.
Precondition: true.
Postcondition: Constructs a counter with value 1.

**BinaryCounter()**

Precondition: true.
Postcondition: Increases the counter value by 1.

**BinaryCounter.Increment()**

Precondition: true.
Postcondition: Returns a ConsList of 0s and 1s ending with a 1 and giving the binary representation of the counter value, with the least significant bit first; i.e. if the sequence is \(\langle a_0, \ldots, a_n\rangle\), the value represented is

\[
\sum_{i=0}^{n} a_i 2^i.
\]

**BinaryCounter.Value()**

This implementation uses a single readable representation variable, `value`. The structural invariant states that `value` refers to the ConsList specified by the Value operation. We leave it as an exercise to show that this implementation satisfies its specification.

Let us now analyze the worst-case running time of Increment. In the worst case, the first loop can iterate \(n\) times, where \(n\) is the length of `value`. This case occurs when `value` consists entirely of 1s; however, when `value` begins with a 0, this loop will not iterate at all. It is easily seen that the second loop always iterates the same number of times as the first loop; hence, in the worst case, the **Increment** operation runs in \(\Theta(n)\) time, or in \(\Theta(\lg v)\) time, where \(v\) is the value represented.

We wish to show that the amortized costs of the **IterBinCounter** operations are in \(O(1)\). We first need to identify the actual costs. Observe that the Value operation runs in \(\Theta(1)\) time and does not change the structure; hence, we can ignore this operation (we will be ignoring only some constant time for each operation). Because the two loops in **Increment** iterate the same number of times, the running time of **Increment** is proportional to
Figure 4.14 IterBinCounter implementation of BinaryCounter, specified in Figure 4.13

**Structural Invariant:** value is a ConsList of 0s and 1s ending with a 1. The sequence represented by this ConsList gives the current value of the BinaryCounter in binary, with the least significant bit first.

IterBinCounter()

\[ value \leftarrow \text{new } \text{ConsList}(1, \text{new } \text{ConsList}) \]

IterBinCounter.Increment()

\[ k \leftarrow 0; c \leftarrow value \]

// **Invariant:** value contains \( k \) 1s, followed by \( c \).

\[ \text{while not } c.\text{isEmpty}() \text{ and } c.\text{head}() = 1 \]

\[ k \leftarrow k + 1; c \leftarrow c.\text{tail}() \]

if \( c.\text{isEmpty}() \)

\[ c \leftarrow \text{new } \text{ConsList}(1, c) \]

else

\[ c \leftarrow \text{new } \text{ConsList}(1, c.\text{tail}()) \]

// **Invariant:** \( c \) contains \( i - 1 \) 0s and a 1, followed by the ConsList // obtained by removing all initial 1s and the first 0 (if any) from value.

\[ \text{for } i \leftarrow 1 \text{ to } k \]

\[ c \leftarrow \text{new } \text{ConsList}(0, c) \]

\[ value \leftarrow c \]

the number of iterations of the \textbf{while} loop, plus some constant. We can therefore use the number of iterations of the \textbf{while} loop as the actual cost of this operation. Note that the actual cost varies from 0 to \( n \), depending on the current value represented.

The next step is to define an appropriate potential function. This step is usually the most challenging part of this technique. While finding a suitable potential function requires some creativity, there are several guidelines we can apply.

First, we can categorize operations of a data structure according to two criteria relevant to amortized analysis:

- the actual cost of the operation; and
• how much it degrades or improves future performance of the data structure.

Using the above criteria, we can divide operations into four categories:

1. Operations that cost little and improve future performance. IterBinCounter contains no such operation; however, we needn’t be too concerned with operations of this type because they cause no problems for our analysis.

2. Operations that cost little but degrade future performance. The Increment operation when the head of value is 0 is an example of this type. It performs no loop iterations, but causes value to have at least one leading 1, so that the next Increment will perform at least one iteration.

3. Operations that cost much but improve future performance. The Increment operation when value has many leading 1s is an example of this type. It performs a loop iteration for each leading 1, but replaces these leading 1s with 0s. Thus, the next Increment will not perform any iterations. In fact, a number of Increment operations will be required before we encounter another expensive one.

4. Operations that cost much and degrade future performance. Our IterBinCounter includes no operations of this type. In fact, operations of this type usually make amortized analysis futile.

The key to finding an appropriate potential function is in striking a good balance between operations of types 2 and 3 above. Consider an operation of type 3. The potential function needs to decrease enough to cancel out the high cost of the operation. On the other hand, it cannot increase too much on an operation of type 2, or this operation’s amortized cost will be too high. We are trying to show that the Increment operation has a constant amortized cost. Therefore, an operation of type 2 must increase the potential function by at most a constant value. Furthermore, an operation of type 3 requires k iterations, so our potential function must have a decrease of roughly k for such an operation. In addition, any potential function must be 0 initially and always nonnegative.

Based upon the above discussion, it would appear that the number of leading 1s in value would be a good measure of the structure’s degradation. Let us therefore consider using as our potential function the number of leading 1s. Unfortunately, we immediately encounter a problem with this
function — it is not initially 0, because an IterBinCounter initially has one leading 1. This function is therefore not a valid potential function. We could make a small adjustment by subtracting 1 from the number of leading 1s; however, the resulting function will then go negative whenever there are no leading 1s.

Before we look for an alternative potential function, we should make one more observation regarding the number of leading 1s. Suppose value begins with a 0, which is followed by a large number of 1s. When an Increment is performed on this state, the leading 0 is replaced by a 1, thus causing the number of leading 1s to increase by a large amount. Hence, even if the number of leading 1s qualified as a valid potential function, it wouldn’t be an appropriate one — the amortized cost of this operation would be high due to the large increase in potential caused by a single operation.

This observation suggests that it is not just the leading 1s that degrade the structure, but that all of the 1s in value contribute to the degradation. We might therefore consider using the total number of 1s in value as our potential function. Again, this number is initially 1, not 0; however, from the structural invariant, value will always contain at least one 1. Therefore, if we subtract 1 from the number of 1s in value, we obtain a valid potential function.

Let us now analyze the amortized cost of Increment relative to this potential function. Suppose the actual cost is \( k \) (i.e., the loops both iterate \( k \) times). The change in potential is simply the number of 1s in value after the operation, minus the number of 1s in value before the operation (the two \(-1\s\) cancel each other when we subtract). The while loop removes \( k \) 1s from \( c \). The if statement adds a 1. The for loop does not change the number of 1s. The total change is therefore \( 1 - k \). The amortized cost is then the actual cost plus the change in potential, or

\[
k + (1 - k) = 1.
\]

We can therefore conclude that the amortized running time of the IterBinCounter operations is in \( O(1) \).

Let us now use this technique to analyze the amortized performance of ExpandableArrayStack. We first observe that operations which result in an error run in \( \Theta(1) \) time and do not change the state of the structure; hence, we can ignore these operations. As we did in Section 4.3, we will again amortize the number of loop iterations; i.e., the actual cost of an operation will be the number of loop iterations performed by that operation. An operation that does not require expanding the array performs no loop
iterations, and an operation that requires expanding the array performs \( n \) loop iterations, where \( n \) is the size of the stack prior to the operation.

We now need an appropriate potential function. We first note that the POP operation not only is cheap, but it also improves the state of the stack by making more array locations available. We therefore don't need to focus on this operation when looking for a potential function. Instead, we need to focus on the PUSH operation. A PUSH that does not expand the array is inexpensive, but degrades the future performance by reducing the number of available array locations. We therefore want the potential function to increase by at most a constant in this case. A PUSH that requires an array expansion is expensive — requiring \( n \) iterations — but improves the performance of the structure by creating additional array locations. We want the potential function to decrease by roughly \( n \) in this case.

We mentioned earlier that we wanted the potential function to be a measure of how full the array is. Perhaps the most natural measure is \( n/k \), where \( n \) is the number of elements in the stack and \( k \) is the size of the array. This function is 0 when \( n \) is 0 and is always nonnegative. Furthermore, because \( n \leq k \), \( n/k \) never exceeds 1; hence, no operation can increase this function by more than 1. However, this also means that no operation causes it to decrease by more than 1. Therefore, it does not fit the characteristics we need for a tight amortized analysis.

In order to overcome this problem, let us try multiplying \( n/k \) by some value in order to give it more of a range. Because we need for the function to decrease by about \( n \) when we expand the array, it will need to have grown by about \( n \) after we have done \( n \) PUSHes; hence, it needs to exhibit at least linear growth. \( n/k \) is bounded by a constant; hence, to cause it to be linear in \( n \), we would want to multiply it by a function that is linear in \( n \). This suggests that we might want to try some function of the form \( an^2/k \), where \( a \) is some positive real number to be determined later.

Using this potential function, consider the amortized cost of a PUSH operation that expands the array. Prior to the operation, \( n = k \). Therefore, the change in potential is

\[
\frac{a(n+1)^2}{2n} - \frac{an^2}{n} = \frac{an^2 + 2an + a - 2an^2}{2n}
\]

\[
= -\frac{an}{2} + a + \frac{a}{2n}
\]

\[
\leq -\frac{an}{2} + a + \frac{3a}{2}
\]

\[
= -\frac{an}{2} + \frac{3a}{2}.
\]
When we add the actual cost of \( n \), we need the result to be bounded by a fixed constant. We can accomplish this by setting \( a = 2 \). The potential function \( 2n^2/k \) therefore results in an amortized cost of no more than 3 in this case.

Now let us consider the amortized cost of a \texttt{Push} operation that does not expand the array. Because \( n \) increases by 1 and \( k \) does not change, the change in the potential function \( 2n^2/k \) is

\[
\frac{2(n+1)^2}{k} - \frac{2n^2}{k} = \frac{2n^2 + 4n + 2 - 2n^2}{k} = \frac{4n + 2}{k} = 4\left(\frac{n + \frac{1}{2}}{k}\right) < 4
\]

because \( k \) must be strictly larger than \( n \), and both are integers. Because no loop iterations are performed in this case, the actual cost is 0; hence, the amortized cost is less than 4.

In order to complete the analysis, we must consider the \texttt{Pop} operation. Because \( n \) is initially positive and decreases by 1, and because \( k \) remains the same, the change in potential is

\[
\frac{2(n-1)^2}{k} - \frac{2n^2}{k} = \frac{2n^2 - 4n + 2 - 2n^2}{k} = \frac{2 - 4n}{k} < 0.
\]

The actual cost is 0. The amortized cost is therefore less than 0.

In each case, the amortized cost is in \( O(1) \). Because the time for each loop iteration and the time required by each operation apart from the loop iterations are both in \( O(1) \), we conclude that the amortized running time of the stack operations is in \( O(1) \).

### 4.6 Summary

We have shown how the top-down design paradigm can be applied to the design of data structures. In many cases, we can reduce the implementation of an ADT to the implementation of simpler or lower-level ADTs. In other
cases, we can reduce the implementation to a common design pattern. The algorithms we have used for implementing the operations of ADTs have been quite simple. As we examine more advanced data structures in the following chapters, we will see that the algorithms in the implementations also use the top-down approach as presented in Chapter 1.

Applying the top-down approach yields clean techniques for proving that implementations of ADTs meet their specifications. The techniques are similar to those presented in Chapter 2, but additionally require proving security of the implementations. Borrowing some ideas from modular and object-oriented languages, we have supplied a computational model that facilitates security in a straightforward way. This model also facilitates the implementation of immutable structures, which in some cases yield performance benefits by eliminating the need to copy data. However, use of immutable structures tends to increase the amount of dynamic memory allocation and requires the presence of an automatic garbage collector.

The analysis techniques of Chapter 3 can be applied to data structures as well. In addition, amortized analysis is sometimes useful for analyzing structures for which operations are occasionally expensive. By amortizing the cost, we can see that sequences of operations may be less expensive than a simple worst-case analysis would suggest. Potential functions provide a general approach to amortized analysis.

4.7 Exercises

Exercise 4.1 Complete the proof of Theorem 4.2 by giving proofs of maintenance and correctness for the two missing cases.

Exercise 4.2 Prove that \textsc{ConsListStack}, shown in Figure 4.8 on page 126, meets its specification, given in Figure 4.1 on page 107.

* Exercise 4.3 Give an algorithm for \textsc{Append}, specified in Figure 4.15. Your algorithm should run in \(O(n)\) time, where \(n\) is the number of elements in \(x\).

Exercise 4.4 Prove that \textsc{IterBinCounter}, shown in Figure 4.14 (page 135), meets the specification shown in Figure 4.13 (page 134).

Exercise 4.5 Let \(f(n)\) denote the number of 1s in \textit{value} after \(n\) calls to \textsc{Increment} on a new \textsc{IterBinCounter}. Prove by induction on \(n\) that the
**Figure 4.15** Specification for **APPEND**

**Precondition:** $x$ and $y$ are ConsLists representing the sequences $\langle x_1, x_2, \ldots, x_n \rangle$ and $\langle y_1, y_2, \ldots, y_m \rangle$, respectively.

**Postcondition:** Returns a ConsList representing the sequence $\langle x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \rangle$.

APPEND($x$, $y$)

total number of iterations of the while loop in these $n$ calls is

$$n - f(n) + 1.$$  

**Exercise 4.6** Figure 4.16 gives an alternative implementation of the **INCREMENT** operation for the **BinaryCounter** ADT.

a. Prove that the implementation that uses this algorithm meets its specification.

b. It is easily seen that the running time of **INCREMENT** is proportional to the number of calls made (including recursive calls) to the internal function **Inc**. Using a potential function, show that the amortized number of calls to **Inc** is in $O(1)$.

**Exercise 4.7** Analyze the amortized cost of the **ExpandableArrayStack** operations using the number of iterations performed as the actual cost and

$$\frac{4n^3}{3k^2}$$

as the potential function, where $n$ is the number of elements in the stack and $k$ is the size of the array.

**Exercise 4.8** Let $c > 1$ be a fixed real number. Suppose we modify Figure 4.6 so that the new array is of size $\lceil c \cdot \text{size} \rceil$. Using the potential function approach, show that the amortized running time of the stack operations is in $O(1)$.

**Exercise 4.9** With **ExpandableArrayStack**, it is possible that the stack reaches a state in which it is using much more space than it requires. This
Figure 4.16 Implementation of the Increment operation for BinaryCounter, specified in Figure 4.13

RecBinCounter.Increment()
    value ← Inc(value)

— Internal Functions Follow —

Precondition: c is a ConsList of 0s and 1s not ending in 0.
Postcondition: Returns a ConsList representing c + 1, where the ConsLists are interpreted as the binary representation of natural numbers, least significant bit first.

RecBinCounter.Inc(c)
    if c.isEmpty()
        return new ConsList(1, c)
    else if c.head() = 0
        return new ConsList(1, c.tail())
    else
        return new ConsList(0, Inc(c.tail()))

can happen if a large number of elements are pushed onto the stack, then most are removed. One solution is to modify the Pop operation so that if the number of elements drops below half the size of the array, then we copy the elements to a new array of half the size. Give a convincing argument that this solution would not result in O(1) amortized running time.

Exercise 4.10 An alternative to the solution sketched in the above exercise is to reduce the size of the array by half whenever it becomes less than 1/4 full, but is still nonempty.

a. Give a modified Pop operation to implement this idea.

* b. Using the technique of Section 4.3, show that the stack operations have an amortized running time in O(1) when this scheme is used. You may assume that the array is initially of size 4.

** c. Repeat the above analysis using a potential function. [Hint: Your potential function will need to increase as the size of the array diverges
CHAPTER 4. BASIC TECHNIQUES

from 2\(n\), where \(n\) is the number of elements in the stack.]

**Exercise 4.11** A queue is similar to a stack, but it provides *first in first out* (FIFO) access to the data items. Instead of the operations Push and Pop, it has operations Enqueue and Dequeue — Enqueue adds an item to the end of the sequence, and Dequeue removes the item from the beginning of the sequence.

a. Give an ADT for a queue.

b. Using the linked list design pattern, give an implementation of your ADT for which all operations run in \(\Theta(1)\) time.

c. Prove that your implementation meets its specification.

**Exercise 4.12** A certain data structure contains operations that each consists of a sequence of zero or more Pops from a stack, followed by a single Push. The stack is initially empty, and no Pop is attempted when the stack is empty.

a. Prove that in any sequence of \(n\) operations on an initialized structure, there are at most 2\(n\) stack operations (i.e., Pushes and Pops).

b. Use a simple potential function to show that the amortized number of stack operations is bounded by a constant.

**Exercise 4.13** A **String**, as specified in Figure 4.17, represents a finite sequence of **Chars**, or characters.

a. Give an implementation of String for which
   - the constructor runs in \(O(n)\) amortized time;
   - Append runs in \(O(m)\) amortized time, where \(m\) is the length of \(x\);
   - Substring runs in \(O(len)\) amortized time; and
   - GetCharacter and Length run in \(\Theta(1)\) time in the worst case.

For the purpose of defining amortized running times, think of the constructor and the Substring operation as appending **Chars** to an empty string. Prove the above running times for your implementation.
Figure 4.17 The String ADT

**Precondition:** $A[0..n-1]$ is an array of Chars, and $n$ is a Nat.

**Postcondition:** Constructs a STRING representing the sequence of Chars in $A$.

$\text{STRING}(A[0..n-1])$

**Precondition:** $x$ is a STRING.

**Postcondition:** Adds the sequence of Chars represented by $x$ to the end of this STRING.

$\text{APPEND}(x)$

**Precondition:** $i$ and $\text{len}$ are Nats such that $i + \text{len}$ does not exceed the length of this STRING.

**Postcondition:** Returns the STRING representing the sequence $\langle a_i, a_{i+1}, \ldots, a_{i+\text{len}-1} \rangle$, where $\langle a_0, \ldots, a_{n-1} \rangle$ is the sequence represented by this STRING.

$\text{SUBSTRING}(i, \text{len})$

**Precondition:** $i$ is a Nat strictly less than the length of this STRING.

**Postcondition:** Returns the Char $a_i$, where $\langle a_0, \ldots, a_{n-1} \rangle$ is the sequence represented by this STRING.

$\text{GETCHARACTER}(i)$

**Precondition:** true.

**Postcondition:** Returns the length of the represented sequence.

$\text{LENGTH}()$
Exercise 4.14 Figure 4.18 gives an ADT for an immutable arbitrary-precision natural number. Such an ADT is useful for defining algorithms for operating on natural numbers which may not fit in a single machine word. We can implement this ADT using a single representation variable, $bits[0..n-1]$, which is an array of 0s and 1s. The structural invariant is that all elements of $bits$ are either 0 or 1, and that if $\text{SizeOf}(bits) \neq 0$,

$$bits[\text{SizeOf}(bits) - 1] = 1.$$  

If $n = \text{SizeOf}(bits)$, the represented number is then

$$\sum_{i=0}^{n-1} bits[i]2^i.$$

Note that the least significant bit has the lowest index; hence, it might be helpful to think of the array with index 0 at the far right, and indices increasing from right to left.

a. Complete this implementation of $\text{BigNum}$ such that

- $\text{NumBits}$ runs in $\Theta(1)$ time;
- $\text{Shift}$ and $\text{GetBits}$ run in $\Theta(n)$ time, where $n$ is the number of bits in the result; and
- the constructor and the remaining operations run in $\Theta(n)$ time, where $n$ is the number of bits in the largest number involved in the operation.

b. Prove that your implementation meets its specification.

4.8 Chapter Notes

The phenomenon that occurs when multiple copies are made of the same reference is known in the literature as aliasing. The problem is thoroughly discussed by, e.g., Aho, Sethi, and Ullman [3] and Muchnick [89].

Use of immutable structures has its roots in functional programming, though it has carried over to some degree to languages from other paradigms.
Figure 4.18 BigNum ADT.

**Precondition:** $A[0..n - 1]$ is an array whose values are all either 0 or 1.

**Postcondition:** Constructs a BigNum representing

$$\sum_{i=0}^{n-1} A[i]2^i.$$  

$\text{BigNum}(A[0..n - 1])$

**Precondition:** $v$ refers to a BigNum.

**Postcondition:** Returns 1 if the value of this BigNum is greater than $v$, 0 if it is equal to $v$, or $-1$ if it is less than $v$.

$\text{BigNum.CompareTo}(v)$

**Precondition:** $v$ refers to a BigNum.

**Postcondition:** Returns a BigNum representing the sum of the value of this BigNum and $v$.

$\text{BigNum.Add}(v)$

**Precondition:** $v$ refers to a BigNum no greater than the value of this BigNum.

**Postcondition:** Returns a BigNum representing the value of this BigNum minus $v$.

$\text{BigNum.Subtract}(v)$

**Precondition:** $i$ is an integer.

**Postcondition:** Returns the floor of the BigNum obtained by multiplying this BigNum by $2^i$.

$\text{BigNum.Shift}(i)$

**Precondition:** true.

**Postcondition:** Returns the number of bits in the binary representation of this BigNum with no leading zeros.

$\text{BigNum.NumBits}()$

**Precondition:** $start$ and $len$ are natural numbers.

**Postcondition:** Returns an array $A[0..len - 1]$ containing the values of bit positions $start$ through $start + len - 1$; zeros are assigned to the high-order positions if necessary.

$\text{GetBits}(start, len)$
Paulson [91] gives a nice introduction to functional programming using ML, where immutable data types are the norm.

The search tree viewer posted on this textbook’s web site contains complete Java implementations of \texttt{ConsList} and \texttt{ConsListStack}. Deep cloning is simulated in this code because only immutable items are placed on the stacks.

Exercise 4.12 is due to Tarjan [105], who gives an excellent survey of amortized analysis. He credits D. Sleator for the potential function method of amortized analysis.