CHAPTER

THREE

BASIC DATA STRUCTURES

3.1 Objectives

- To understand the abstract data types stack, queue, deque, and list.
- To be able to implement the ADTs stack, queue, and deque using Python lists.
- To understand the performance of the implementations of basic linear data structures.
- To understand prefix, infix, and postfix expression formats.
- To use stacks to evaluate postfix expressions.
- To use stacks to convert expressions from infix to postfix.
- To use queues for basic timing simulations.
- To be able to recognize problem properties where stacks, queues, and deques are appropriate data structures.
- To be able to implement the abstract data type list as a linked list using the node and reference pattern.
- To be able to compare the performance of our linked list implementation with Python's list implementation.

3.2 What Are Linear Structures?

We will begin our study of data structures by considering four simple but very powerful concepts. Stacks, queues, deques, and lists are examples of data collections whose items are ordered depending on how they are added or removed. Once an item is added, it stays in that position relative to the other elements that came before and came after it. Collections such as these are often referred to as linear data structures.

Linear structures can be thought of as having two ends. Sometimes these ends are referred to as the "left" and the "right" or in some cases the "front" and the "rear." You could also call them the "top" and the "bottom." The names given to the ends are not significant. What distinguishes one linear structure from another is the way in which items are added and removed, in particular the location where these additions and removals occur. For example, a structure might allow

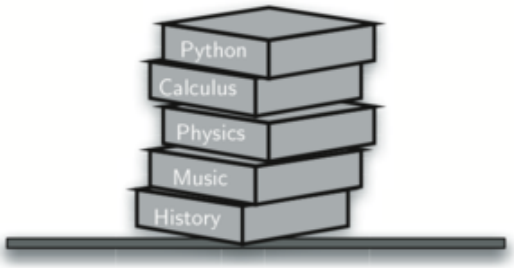

Figure 3.1: A Stack of Books

new items to be added at only one end. Some structures might allow items to be removed from either end.

These variations give rise to some of the most useful data structures in computer science. They appear in many algorithms and can be used to solve a variety of important problems.

3.3 Stacks

3.3.1 What is a Stack?

A **stack** (sometimes called a "push-down stack") is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the "top." The end opposite the top is known as the "base."

The base of the stack is significant since items stored in the stack that are closer to the base represent those that have been in the stack the longest. The most recently added item is the one that is in position to be removed first. This ordering principle is sometimes called **LIFO, last-in first-out**. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base.

Many examples of stacks occur in everyday situations. Almost any cafeteria has a stack of trays or plates where you take the one at the top, uncovering a new tray or plate for the next customer in line. Imagine a stack of books on a desk (Figure 3.1). The only book whose cover is visible is the one on top. To access others in the stack, we need to remove the ones that are sitting on top of them. Figure 3.2 shows another stack. This one contains a number of primitive Python data objects.

One of the most useful ideas related to stacks comes from the simple observation of items as they are added and then removed. Assume you start out with a clean desktop. Now place books one at a time on top of each other. You are constructing a stack. Consider what happens when you begin removing books. The order that they are removed is exactly the reverse of the order that they were placed. Stacks are fundamentally important, as they can be used to reverse the order of items. The order of insertion is the reverse of the order of removal. Figure 3.3 shows the Python data object stack as it was created and then again as items are removed. Note the order of the objects.

Considering this reversal property, you can perhaps think of examples of stacks that occur as

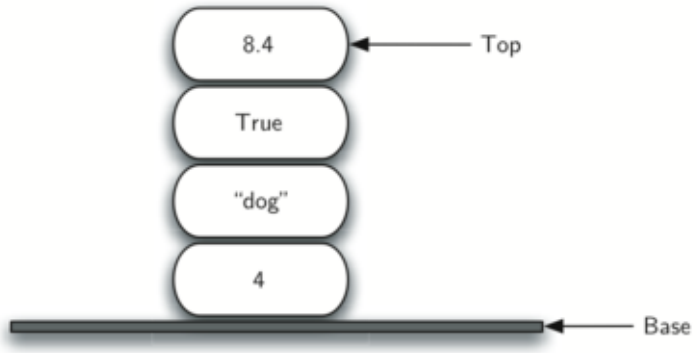

Figure 3.2: A Stack of Primitive Python Objects

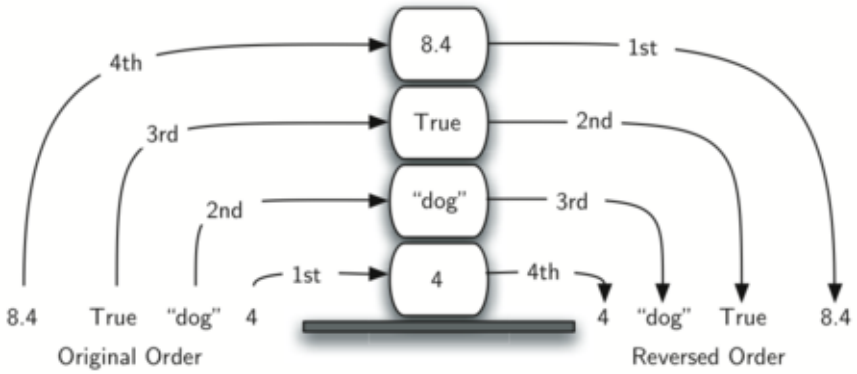

Figure 3.3: The Reversal Property of Stacks

Stack Operation	Stack Contents	Return Value
`s.is_empty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.is_empty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

Table 3.1: Sample Stack Operations

you use your computer. For example, every web browser has a Back button. As you navigate from web page to web page, those pages are placed on a stack (actually it is the URLs that are going on the stack). The current page that you are viewing is on the top and the first page you looked at is at the base. If you click on the Back button, you begin to move in reverse order through the pages.

3.4 The Stack Abstract Data Type

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top." Stacks are ordered LIFO. The stack operations are given below.

- **`Stack()`** creates a new stack that is empty. It needs no parameters and returns an empty stack.
- **`push(item)`** adds a new item to the top of the stack. It needs the item and returns nothing.
- **`pop()`** removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- **`peek()`** returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- **`is_empty()`** tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- **`size()`** returns the number of items on the stack. It needs no parameters and returns an integer.

For example, if s is a stack that has been created and starts out empty, then Table 3.1 shows the results of a sequence of stack operations. Under stack contents, the top item is listed at the far right.

3.4.1 Implementing A Stack in Python

Now that we have clearly defined the stack as an abstract data type we will turn our attention to using Python to implement the stack. Recall that when we give an abstract data type a physical implementation we refer to the implementation as a data structure.

As we described in Chapter 1, in Python, as in any object-oriented programming language, the implementation of choice for an abstract data type such as a stack is the creation of a new class. The stack operations are implemented as methods. Further, to implement a stack, which is a collection of elements, it makes sense to utilize the power and simplicity of the primitive collections provided by Python. We will use a list.

Recall that the list class in Python provides an ordered collection mechanism and a set of methods. For example, if we have the list $[2, 5, 3, 6, 7, 4]$, we need only to decide which end of the list will be considered the top of the stack and which will be the base. Once that decision is made, the operations can be implemented using the list methods such as append and pop.

The following stack implementation assumes that the end of the list will hold the top element of the stack. As the stack grows (as push operations occur), new items will be added on the end of the list. pop operations will manipulate that same end.

```
# Completed implementation of a stack ADT
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

Remember that nothing happens when we click the run button other than the definition of the class. We must create a Stack object and then use it. shows the Stack class in action as we perform the sequence of operations from Table 3.1.

```
s = Stack()

print(s.is_empty())
s.push(4)
s.push('dog')
```

```
print(s.peek())
s.push(True)
print(s.size())
print(s.is_empty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

It is important to note that we could have chosen to implement the stack using a list where the top is at the beginning instead of at the end. In this case, the previous pop and append methods would no longer work and we would have to index position 0 (the first item in the list) explicitly using pop and insert. The implementation is shown below.

```
class Stack:
  def __init__(self):
    self.items = []

  def is_empty(self):
    return self.items == []

  def push(self, item):
    self.items.insert(0, item)

  def pop(self):
    return self.items.pop(0)

  def peek(self):
    return self.items[0]

  def size(self):
    return len(self.items)

s = Stack()
s.push('hello')
s.push('true')
print(s.pop())
```

This ability to change the physical implementation of an abstract data type while maintaining the logical characteristics is an example of abstraction at work. However, even though the stack will work either way, if we consider the performance of the two implementations, there is definitely a difference. Recall that the append and pop() operations were both $O(1)$. This means that the first implementation will perform push and pop in constant time no matter how many items are on the stack. The performance of the second implementation suffers in that the insert(0) and pop(0) operations will both require $O(n)$ for a stack of size n. Clearly, even though the implementations are logically equivalent, they would have very different timings when performing benchmark testing.

3.4.2 Self Check

Given the following sequence of stack operations, what is the top item on the stack when the sequence is complete?

```
m = Stack()
m.push('x')
m.push('y')
m.pop()
m.push('z')
m.peek()
```

1. 'x'
2. 'y'
3. 'z'
4. The stack is empty

Given the following sequence of stack operations, what is the top item on the stack when the sequence is complete?

```
m = Stack()
m.push('x')
m.push('y')
m.push('z')
while not m.is_empty():
  m.pop()
  m.pop()
```

1. 'x'
2. the stack is empty
3. an error will occur
4. 'z'

Write a function rev_string(my_str) that uses a stack to reverse the characters in a string.

3.4.3 Simple Balance Parentheses

We now turn our attention to using stacks to solve real computer science problems. You have no doubt written arithmetic expressions such as

$(5+6)*(7+8)/(4+3)$

where parentheses are used to order the performance of operations. You may also have some experience programming in a language such as Lisp with constructs like

```
(defun square(n)
    (* n n))
```

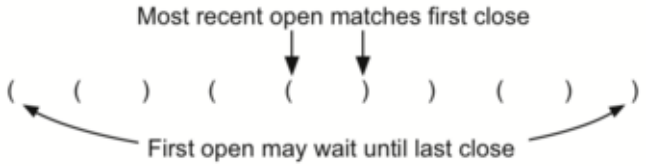

Figure 3.4: Matching Parentheses

This defines a function called square that will return the square of its argument n. Lisp is notorious for using lots and lots of parentheses.

In both of these examples, parentheses must appear in a balanced fashion. **Balanced parentheses** means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. Consider the following correctly balanced strings of parentheses:

```
(()()()())

(((())))

(()((())()))
```

Compare those with the following, which are not balanced:

```
((((((())

())))

(()()(()
```

The ability to differentiate between parentheses that are correctly balanced and those that are unbalanced is an important part of recognizing many programming language structures.

The challenge then is to write an algorithm that will read a string of parentheses from left to right and decide whether the symbols are balanced. To solve this problem we need to make an important observation. As you process symbols from left to right, the most recent opening parenthesis must match the next closing symbol (see Figure 3.4). Also, the first opening symbol processed may have to wait until the very last symbol for its match. Closing symbols match opening symbols in the reverse order of their appearance; they match from the inside out. This is a clue that stacks can be used to solve the problem.

Once you agree that a stack is the appropriate data structure for keeping the parentheses, the statement of the algorithm is straightforward. Starting with an empty stack, process the parenthesis strings from left to right. If a symbol is an opening parenthesis, push it on the stack as a signal that a corresponding closing symbol needs to appear later. If, on the other hand, a symbol is a closing parenthesis, pop the stack. As long as it is possible to pop the stack to match every closing symbol, the parentheses remain balanced. If at any time there is no opening symbol on the stack to match a closing symbol, the string is not balanced prop-

erly. At the end of the string, when all symbols have been processed, the stack should be empty.

```
1  import Stack #import the Stack class as previously defined
2
3  def par_checker(symbol_string):
4     s = Stack()
5     balanced = True
6     index = 0
7     while index < len(symbol_string) and balanced:
8        symbol = symbol_string[index]
9        if symbol == "(":
10          s.push(symbol)
11       else:
12          if s.is_empty():
13             balanced = False
14          else:
15             s.pop()
16
17       index = index + 1
18
19    if balanced and s.is_empty():
20       return True
21    else:
22       return False
23
24 print(par_checker('((()))'))
25 print(par_checker('(()'))
```

This function, par_checker, assumes that a Stack class is available and returns a boolean result as to whether the string of parentheses is balanced. Note that the boolean variable balanced is initialized to True as there is no reason to assume otherwise at the start. If the current symbol is (, then it is pushed on the stack (lines 9–10). Note also in line 15 that pop simply removes a symbol from the stack. The returned value is not used since we know it must be an opening symbol seen earlier. At the end (lines 19–22), as long as the expression is balanced and the stack has been completely cleaned off, the string represents a correctly balanced sequence of parentheses.

3.4.4 Balanced Symbols (A General Case)

he balanced parentheses problem shown above is a specific case of a more general situation that arises in many programming languages. The general problem of balancing and nesting different kinds of opening and closing symbols occurs frequently. For example, in Python square brackets, `[` and `]`, are used for lists; curly braces, `{` and `}`, are used for dictionaries; and parentheses, `(` and `)`, are used for tuples and arithmetic expressions. It is possible to mix symbols as long as each maintains its own open and close relationship. Strings of symbols such as

```
{ { ( [ ] [ ] ) } ( ) }
```

```
[ [ { { ( ( ) ) } } ] ]

[ ] [ ] [ ] ( ) { }
```

are properly balanced in that not only does each opening symbol have a corresponding closing symbol, but the types of symbols match as well.

Compare those with the following strings that are not balanced:

```
( [ ) ]

( ( ( ) ] ) )

[ { ( ) ]
```

The simple parentheses checker from the previous section can easily be extended to handle these new types of symbols. Recall that each opening symbol is simply pushed on the stack to wait for the matching closing symbol to appear later in the sequence. When a closing symbol does appear, the only difference is that we must check to be sure that it correctly matches the type of the opening symbol on top of the stack. If the two symbols do not match, the string is not balanced. Once again, if the entire string is processed and nothing is left on the stack, the string is correctly balanced.

The Python program to implement this is shown below The only change appears in line 16 where we call a helper function, matches, to assist with symbol-matching. Each symbol that is removed from the stack must be checked to see that it matches the current closing symbol. If a mismatch occurs, the boolean variable balanced is set to False.

```
1  import Stack # As previously defined
2
3  # Completed extended par_checker for: [,{,(,),},]
4
5  def par_checker(symbol_string):
6     s = Stack()
7     balanced = True
8     index = 0
9     while index < len(symbol_string) and balanced:
10       symbol = symbol_string[index]
11       if symbol in "([{":
12          s.push(symbol)
13       else:
14          if s.is_empty():
15             balanced = False
16          else:
17             top = s.pop()
18             if not matches(top, symbol):
19                   balanced = False
20       index = index + 1
```

```
21      if balanced and s.is_empty():
22          return True
23      else:
24          return False
25
26  def matches(open, close):
27      opens = "([{"
28      closes = ")]}"
29      return opens.index(open) == closes.index(close)
30
31
32  print(par_checker('{{([][])}()}'))
33  print(par_checker('[{()]'))
```

These two examples show that stacks are very important data structures for the processing of language constructs in computer science. Almost any notation you can think of has some type of nested symbol that must be matched in a balanced order. There are a number of other important uses for stacks in computer science. We will continue to explore them in the next sections.

3.4.5 Converting Decimal Numbers to Binary Numbers

In your study of computer science, you have probably been exposed in one way or another to the idea of a binary number. Binary representation is important in computer science since all values stored within a computer exist as a string of binary digits, a string of 0s and 1s. Without the ability to convert back and forth between common representations and binary numbers, we would need to interact with computers in very awkward ways.

Integer values are common data items. They are used in computer programs and computation all the time. We learn about them in math class and of course represent them using the decimal number system, or base 10. The decimal number 233_10 and its corresponding binary equivalent 11101001_2 are interpreted respectively as $2 * 10^2 + 3 * 10^1 + 3 * 10^0$ and $1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$

But how can we easily convert integer values into binary numbers? The answer is an algorithm called "Divide by 2" that uses a stack to keep track of the digits for the binary result.

The Divide by 2 algorithm assumes that we start with an integer greater than 0. A simple iteration then continually divides the decimal number by 2 and keeps track of the remainder. The first division by 2 gives information as to whether the value is even or odd. An even value will have a remainder of 0. It will have the digit 0 in the ones place. An odd value will have a remainder of 1 and will have the digit 1 in the ones place. We think about building our binary number as a sequence of digits; the first remainder we compute will actually be the last digit in the sequence. As shown in Figure 3.5, we again see the reversal property that signals that a stack is likely to be the appropriate data structure for solving the problem.

The Python code in below implements the Divide by 2 algorithm. The function divide_by_2 takes an argument that is a decimal number and repeatedly divides it by 2. Line 7 uses the built-in modulo operator, %, to extract the remainder and line 8 then pushes it on the stack. After the division process reaches 0, a binary string is constructed in lines 11–13. Line

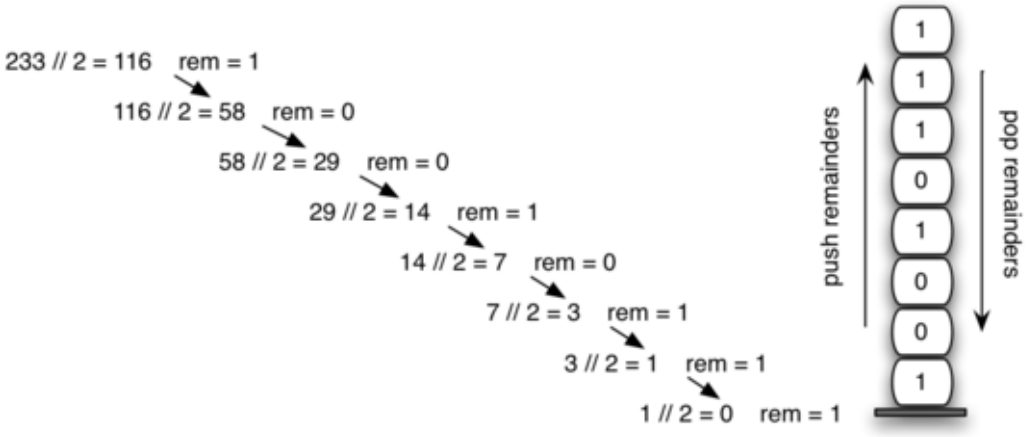

Figure 3.5: Decimal-To-Binary Conversion)

11 creates an empty string. The binary digits are popped from the stack one at a time and appended to the right-hand end of the string. The binary string is then returned.

```
import Stack # As previously defined

def divide_by_2(dec_number):
    rem_stack = Stack()

    while dec_number > 0:
        rem = dec_number % 2
        rem_stack.push(rem)
        dec_number = dec_number // 2

    bin_string = ""
    while not rem_stack.is_empty():
        bin_string = bin_string + str(rem_stack.pop())

    return bin_string

print(divide_by_2(42))
```

he algorithm for binary conversion can easily be extended to perform the conversion for any base. In computer science it is common to use a number of different encodings. The most common of these are binary, octal (base 8), and hexadecimal (base 16).

The decimal number 233 and its corresponding octal and hexadecimal equivalents 351_8 and $E9_16$ are interpreted as $3 * 8^2 + 5 * 8^1 + 1 * 8^0$ and $14 * 16^1 + 9 * 16^0$

The function divide_by_2 can be modified to accept not only a decimal value but also a base for the intended conversion. The "Divide by 2" idea is simply replaced with a more general "Divide by base." A new function called base_converter, shown below, takes a decimal number and any base between 2 and 16 as parameters. The remainders are still pushed onto the stack until the value being converted becomes 0. The same left-to-right string construction technique can be used with one slight change. Base 2 through base 10

numbers need a maximum of 10 digits, so the typical digit characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 work fine. The problem comes when we go beyond base 10. We can no longer simply use the remainders, as they are themselves represented as two-digit decimal numbers. Instead we need to create a set of digits that can be used to represent those remainders beyond 9.

```
import Stack # As previously defined

def base_converter(dec_number, base):
    digits = "0123456789ABCDEF"

    rem_stack = Stack()

    while dec_number > 0:
        rem = dec_number % base
        rem_stack.push(rem)
        dec_number = dec_number // base

    new_string = ""
    while not rem_stack.is_empty():
        new_string = new_string + digits[rem_stack.pop()]

    return new_string

print(base_converter(25, 2))
print(base_converter(25, 16))
```

A solution to this problem is to extend the digit set to include some alphabet characters. For example, hexadecimal uses the ten decimal digits along with the first six alphabet characters for the 16 digits. To implement this, a digit string is created that stores the digits in their corresponding positions. 0 is at position 0, 1 is at position 1, A is at position 10, B is at position 11, and so on. When a remainder is removed from the stack, it can be used to index into the digit string and the correct resulting digit can be appended to the answer. For example, if the remainder 13 is removed from the stack, the digit D is appended to the resulting string.

3.4.6 Self Check

1. What is the value of 25 expressed as an octal number?
2. What is the value of 256 expressed as a hexidecimal number?
3. What is the value of 26 expressed in base 26?

3.4.7 Infix, Prefix, and Postfix Expressions

When you write an arithmetic expression such as $B*C$, the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator $*$ appears between them

in the expression. This type of notation is referred to as infix since the operator is in between the two operands that it is working on.

Consider another infix example, $A + B * C$. The operators $+$ and $*$ still appear between the operands, but there is a problem. Which operands do they work on? Does the $+$ work on A and B or does the $*$ take B and C? The expression seems ambiguous.

In fact, you have been reading and writing these types of expressions for a long time and they do not cause you any problem. The reason for this is that you know something about the operators $+$ and $*$. Each operator has a **precedence** level. Operators of higher precedence are used before operators of lower precedence. The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

Let's interpret the troublesome expression $A + B * C$ using operator precedence. B and C are multiplied first, and A is then added to that result. $(A + B) * C$ would force the addition of A and B to be done first before the multiplication. In expression $A + B + C$, by precedence (via associativity), the leftmost $+$ would be done first.

Although all this may be obvious to you, remember that computers need to know exactly what operators to perform and in what order. One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a fully parenthesized expression. This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations; there is no ambiguity. There is also no need to remember any precedence rules.

The expression $A + B * C + D$ can be rewritten as $((A + (B * C)) + D)$ to show that the multiplication happens first, followed by the leftmost addition. $A + B + C + D$ can be written as $(((A + B) + C) + D)$ since the addition operations associate from left to right.

There are two other very important expression formats that may not seem obvious to you at first. Consider the infix expression $A + B$. What would happen if we moved the operator before the two operands? The resulting expression would be $+AB$. Likewise, we could move the operator to the end. We would get $AB+$. These look a bit strange.

These changes to the position of the operator with respect to the operands create two new expression formats, prefix and postfix. Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands. A few more examples should help to make this a bit clearer (see Table 3.2).

$A + B * C$ would be written as $+A * BC$ in prefix. The multiplication operator comes immediately before the operands B and C, denoting that $*$ has precedence over $+$. The addition operator then appears before the A and the result of the multiplication.

In postfix, the expression would be $ABC * +$. Again, the order of operations is preserved since the $*$ appears immediately after the B and the C, denoting that $*$ has precedence, with $+$ coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

Now consider the infix expression $(A + B) * C$. Recall that in this case, infix requires the parentheses to force the performance of the addition before the multiplication. However, when

Infix Expression	Prefix Expression	Postfix Expression
$A + B$	$+AB$	$AB+$
$A + B * C$	$+A * BC$	$ABC * +$

Table 3.2: Examples of Infix, Prefix, and Postfix

Infix Expression	Prefix Expression	Postfix Expression
$(A + B) * C$	$* + ABC$	$AB + C*$

Table 3.3: An Expression with Parentheses

$A+B$ was written in prefix, the addition operator was simply moved before the operands, $+AB$. The result of this operation becomes the first operand for the multiplication. The multiplication operator is moved in front of the entire expression, giving us $* + ABC$. Likewise, in postfix $AB+$ forces the addition to happen first. The multiplication can be done to that result and the remaining operand C. The proper postfix expression is then $AB + C*$.

Consider these three expressions again (see Table 3.3). Something very important has happened. Where did the parentheses go? Why don't we need them in prefix and postfix? The answer is that the operators are no longer ambiguous with respect to the operands that they work on. Only infix notation requires the additional symbols. The order of operations within prefix and postfix expressions is completely determined by the position of the operator and nothing else. In many ways, this makes infix the least desirable notation to use.

Table 3.4 shows some additional examples of infix expressions and the equivalent prefix and postfix expressions. Be sure that you understand how they are equivalent in terms of the order of the operations being performed.

3.4.8 Conversion of Infix Expressions to Prefix and Postfix

So far, we have used ad hoc methods to convert between infix expressions and the equivalent prefix and postfix expression notations. As you might expect, there are algorithmic ways to perform the conversion that allow any expression of any complexity to be correctly transformed.

The first technique that we will consider uses the notion of a fully parenthesized expression that was discussed earlier. Recall that $A + B * C$ can be written as $(A + (B * C))$ to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

Look at the right parenthesis in the subexpression $(B * C)$ above. If we were to move the

Infix Expression	Prefix Expression	Postfix Expression
$A + B * C + D$	$+ + A * BCD$	$ABC * +D+$
$(A + B) * (C + D)$	$* + AB + CD$	$AB + CD + *$
$A * B + C * D$	$+ * AB * CD$	$AB * CD * +$
$A + B + C + D$	$+ + +ABCD$	$AB + C + D+$

Table 3.4: Additional Examples of Infix, Prefix, and Postfix

Figure 3.6: Moving Operators to the Right for Postfix Notation)

Figure 3.7: Moving Operators to the Left for Prefix Notation)

multiplication symbol to that position and remove the matching left parenthesis, giving us $BC*$, we would in effect have converted the subexpression to postfix notation. If the addition operator were also moved to its corresponding right parenthesis position and the matching left parenthesis were removed, the complete postfix expression would result (See figure 3.6)

If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation (see Figure 3.7). The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.

So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix notation.

Here is a more complex expression: $(A + B) * C - (D - E) * (F + G)$. Figure 3.8 shows the conversion to postfix and prefix notations.

3.4.9 General Infix-to-Postfix Conversion

We need to develop an algorithm to convert any infix expression to a postfix expression. To do this we will look closer at the conversion process.

Consider once again the expression $A + B * C$. As shown above, $ABC * +$ is the postfix equivalent. We have already noted that the operands A, B, and C stay in their relative positions. It is only the operators that change position. Let us look again at the operators in the infix expression. The first operator that appears from left to right is $+$. However, in the postfix expression, $+$ is at the end since the next operator, $*$, has precedence over addition. The order

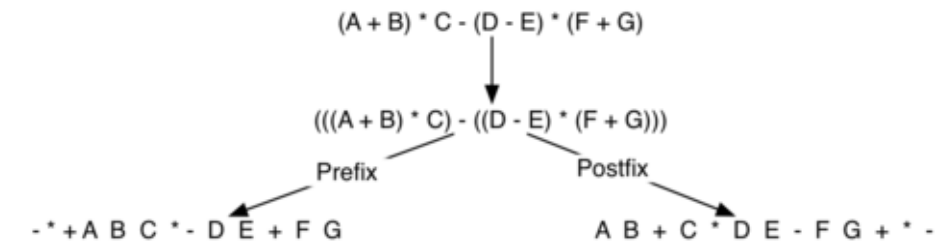

Figure 3.8: Converting a Complex Expression to Prefix and Postfix Notations)

of the operators in the original expression is reversed in the resulting postfix expression.

As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their precedence. This is the case with the addition and the multiplication in this example. Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used. Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

What about $(A + B) * C$? Recall that $AB + C*$ is the postfix equivalent. Again, processing this infix expression from left to right, we see $+$ first. In this case, when we see $*$, $+$ has already been placed in the result expression because it has precedence over $*$ by virtue of the parentheses. We can now start to see how the conversion algorithm will work. When we see a left parenthesis, we will save it to denote that another operator of high precedence will be coming. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique). When that right parenthesis does appear, the operator can be popped from the stack.

As we scan the infix expression from left to right, we will use a stack to keep the operators. This will provide the reversal that we noted in the first example. The top of the stack will always be the most recently saved operator. Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are $*$, $/$, $+$, and $-$, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on. The following steps will produce a string of tokens in postfix order.

1. Create an empty stack called op_stack for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method split.
3. Scan the token list from left to right.
    - If the token is an operand, append it to the end of the output list.
    - If the token is a left parenthesis, push it on the op_stack.
    - If the token is a right parenthesis, pop the op_stack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
    - If the token is an operator, $*$, $/$, $+$, or $-$, push it on the op_stack. However, first remove any operators already on the op_stack that have higher or equal precedence and append them to the output list.

    When the input expression has been completely processed, check the op_stack. Any operators still on the stack can be removed and appended to the end of the output list.

Figure 3.9 shows the conversion algorithm working on the expression $A * B + C * D$. Note that the first $*$ operator is removed upon seeing the $+$ operator. Also, $+$ stays on the stack when the second $*$ occurs, since multiplication has precedence over addition. At the end of the

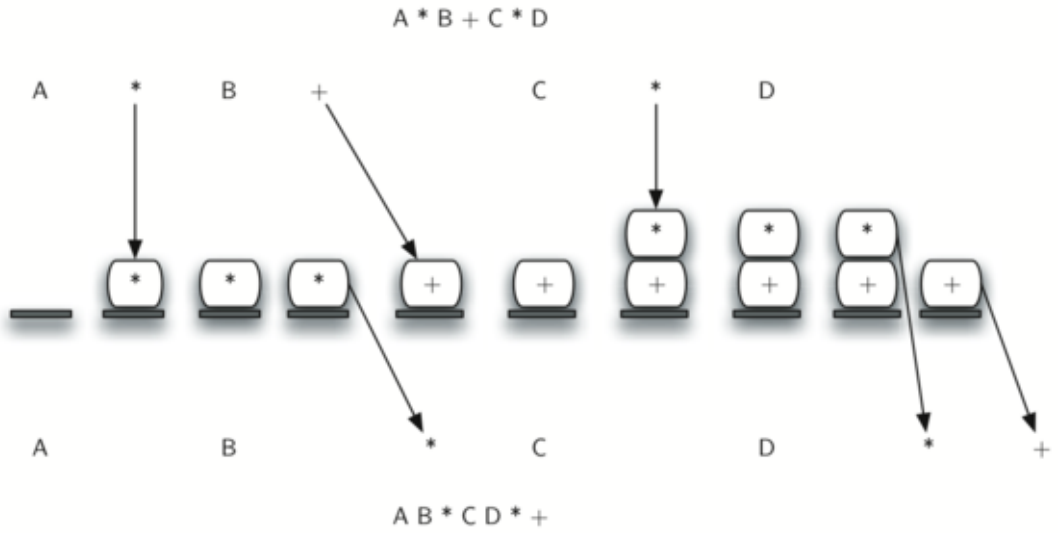

Figure 3.9: Converting $A * B + C * D$ to Postfix Notation)

infix expression the stack is popped twice, removing both operators and placing + as the last operator in the postfix expression.

In order to code the algorithm in Python, we will use a dictionary called prec to hold the precedence values for the operators. This dictionary will map each operator to an integer that can be compared against the precedence levels of other operators (we have arbitrarily used the integers 3, 2, and 1). The left parenthesis will receive the lowest value possible. This way any operator that is compared against it will have higher precedence and will be placed on top of it. Line 15 defines the operands to be any upper-case character or digit. The complete conversion function is shown below.

```
1  import Stack  # As previously defined
2
3  def infix_to_postfix(infix_expr):
4     prec = {}
5     prec["*"] = 3
6     prec["/"] = 3
7     prec["+"] = 2
8     prec["-"] = 2
9     prec["("] = 1
10    op_stack = Stack()
11    postfix_list = []
12    token_list = infix_expr.split()
13
14    for token in token_list:
15       if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in
          "0123456789":
16          postfix_list.append(token)
17       elif token == '(':
18          op_stack.push(token)
19       elif token == ')':
20          top_token = op_stack.pop()
```

```
21          while top_token != '(':
22             postfix_list.append(top_token)
23             top_token = op_stack.pop()
24       else:
25          while (not op_stack.is_empty()) and \
26             (prec[op_stack.peek()] >= prec[token]):
27               postfix_list.append(op_stack.pop())
28          op_stack.push(token)
29
30    while not op_stack.is_empty():
31       postfix_list.append(op_stack.pop())
32    return " ".join(postfix_list)
33
34 print(infix_to_postfix("A * B + C * D"))
35 print(infix_to_postfix("( A + B ) * C - ( D - E ) * ( F + G )"))
36 ...
```

A few more examples of execution in the Python shell are shown below.

```
>>> infix_to_postfix("( A + B ) * ( C + D )")
'A B + C D + *'
>>> infix_to_postfix("( A + B ) * C")
'A B + C *'
>>> infix_to_postfix("A + B * C")
'A B C * +'
>>>
```

3.4.10 Postfix Evaluation

As a final stack example, we will consider the evaluation of an expression that is already in postfix notation. In this case, a stack is again the data structure of choice. However, as you scan the postfix expression, it is the operands that must wait, not the operators as in the conversion algorithm above. Another way to think about the solution is that whenever an operator is seen on the input, the two most recent operands will be used in the evaluation.

To see this in more detail, consider the postfix expression $456 * +$. As you scan the expression from left to right, you first encounter the operands 4 and 5. At this point, you are still unsure what to do with them until you see the next symbol. Placing each on the stack ensures that they are available if an operator comes next.

In this case, the next symbol is another operand. So, as before, push it and check the next symbol. Now we see an operator, $*$. This means that the two most recent operands need to be used in a multiplication operation. By popping the stack twice, we can get the proper operands and then perform the multiplication (in this case getting the result 30).

We can now handle this result by placing it back on the stack so that it can be used as an operand for the later operators in the expression. When the final operator is processed, there will be only one value left on the stack. Pop and return it as the result of the expression. Figure 3.10 shows the stack contents as this entire example expression is being processed.

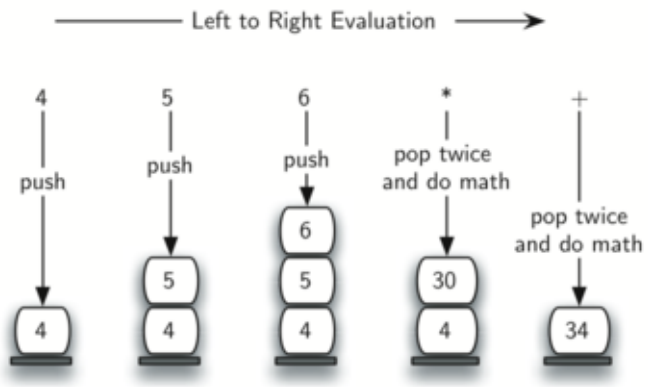

Figure 3.10: Sack Contents During Evaluation)

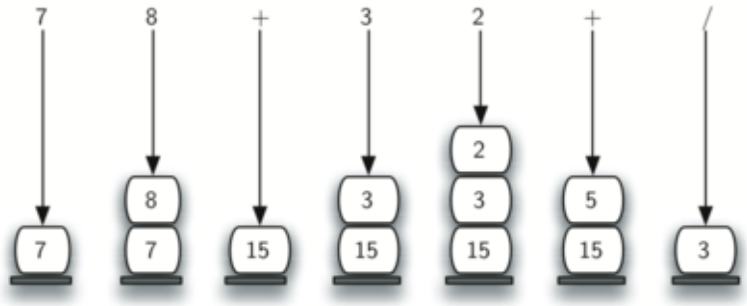

Figure 3.11: A More Complex Example of Evaluation)

Figure 3.11 shows a slightly more complex example, $78+32+/$. There are two things to note in this example. First, the stack size grows, shrinks, and then grows again as the subexpressions are evaluated. Second, the division operation needs to be handled carefully. Recall that the operands in the postfix expression are in their original order since postfix changes only the placement of operators. When the operands for the division are popped from the stack, they are reversed. Since division is *not* a commutative operator, in other words $15/5$ is not the same as $5/15$, we must be sure that the order of the operands is not switched.

Assume the postfix expression is a string of tokens delimited by spaces. The operators are $*$, $/$, $+$, and $-$, and the operands are assumed to be single-digit integer values. The output will be an integer result.

1. Create an empty stack called operand_stack.
2. Convert the string to a list by using the string method split.
3. Scan the token list from left to right.
    - If the token is an operand, convert it from a string to an integer and push the value onto the operand_stack.
    - If the token is an operator, $*$, $/$, $+$, or $-$, it will need two operands. Pop the operand_stack twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the operand_stack.

4. When the input expression has been completely processed, the result is on the stack. Pop the operand_stack and return the value.

The complete function for the evaluation of postfix expressions is shown below. To assist with the arithmetic, a helper function do_math is defined that will take two operands and an operator and then perform the proper arithmetic operation.

```
import Stack  # As previously defined

def postfix_eval(postfix_expr):
    operand_stack = Stack()
    token_list = postfix_expr.split()

    for token in token_list:
        if token in "0123456789":
            operand_stack.push(int(token))
        else:
            operand2 = operand_stack.pop()
            operand1 = operand_stack.pop()
            result = do_math(token, operand1, operand2)
            operand_stack.push(result)
    return operand_stack.pop()

def do_math(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2

print(postfix_eval('7 8 + 3 2 + /'))
```

It is important to note that in both the postfix conversion and the postfix evaluation programs we assumed that there were no errors in the input expression. Using these programs as a starting point, you can easily see how error detection and reporting can be included. We leave this as an exercise at the end of the chapter.

Self Check

1. Convert the following expression to postfix $10 + 3 * 5/(16 - 4)$.
2. Modify the infix_to_postfix function so that it can convert the following expression: $5 * 3\hat{}(4 - 2)$

Figure 3.12: A Queue of Python Data Objects)

3.5 Queues

We now turn our attention to another linear data structure. This one is called **queue**. Like stacks, queues are relatively simple and yet can be used to solve a wide range of important problems.

3.5.1 What Is a Queue?

A queue is an ordered collection of items where the addition of new items happens at one end, called the "rear," and the removal of existing items occurs at the other end, commonly called the "front." As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called FIFO, first-in first-out. It is also known as "first-come first-served."

The simplest example of a queue is the typical line that we all participate in from time to time. We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line (so that we can pop the tray stack). Well-behaved lines, or queues, are very restrictive in that they have only one way in and only one way out. There is no jumping in the middle and no leaving before you have waited the necessary amount of time to get to the front. Figure 3.12 shows a simple queue of Python data objects.

Computer science also has common examples of queues. Our computer laboratory has 30 computers networked with a single printer. When students want to print, their print tasks "get in line" with all the other printing tasks that are waiting. The first task in is the next to be completed. If you are last in line, you must wait for all the other tasks to print ahead of you. We will explore this interesting example in more detail later.

In addition to printing queues, operating systems use a number of different queues to control processes within a computer. The scheduling of what gets done next is typically based on a queuing algorithm that tries to execute programs as quickly as possible and serve as many users as it can. Also, as we type, sometimes keystrokes get ahead of the characters that appear on the screen. This is due to the computer doing other work at that moment. The keystrokes are being placed in a queue-like buffer so that they can eventually be displayed on the screen in the proper order.

Queue Operation	Queue Contents	Return Value
`q.is_empty()`	`[]`	`True`
`q.enqueue(4)`	`[4]`	
`q.enqueue('dog')`	`['dog',4]`	
`q.enqueue(True)`	`[True,'dog',4]`	
`q.size()`	`[True,'dog',4]`	`3`
`q.is_empty()`	`[True,'dog',4]`	`False`
`q.enqueue(8.4)`	`[8.4,True,'dog',4]`	
`q.dequeue()`	`[8.4,True,'dog']`	`4`
`q.dequeue()`	`[8.4,True]`	`'dog'`
`q.size()`	`[8.4,True]`	`2`

Table 3.5: Example Queue Operations

3.5.2 The Queue Abstract Data Type

The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
- enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
- dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- is_empty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the queue. It needs no parameters and returns an integer.

As an example, if we assume that q is a queue that has been created and is currently empty, then Table 3.5 shows the results of a sequence of queue operations. The queue contents are shown such that the front is on the right. 4 was the first item enqueued so it is the first item returned by dequeue.

3.5.3 Implementing A Queue in Python

It is again appropriate to create a new class for the implementation of the abstract data type queue. As before, we will use the power and simplicity of the list collection to build the internal representation of the queue.

We need to decide which end of the list to use as the rear and which to use as the front. The implementation shown below assumes that the rear is at position 0 in the list. This allows us to use the insert function on lists to add new elements to the rear of the queue. The pop operation

can be used to remove the front element (the last element of the list). Recall that this also means that enqueue will be $O(n)$ and dequeue will be $O(1)$.

```
# Completed implementation of a queue ADT
class Queue:
   def __init__(self):
      self.items = []

   def is_empty(self):
      return self.items == []

   def enqueue(self, item):
      self.items.insert(0,item)

   def dequeue(self):
      return self.items.pop()

   def size(self):
      return len(self.items)
```

Self Check

Suppose you have the following series of queue operations:

```
q = Queue()
q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
q.dequeue()
```

What items are left in the queue?

1. 'hello', 'dog'
2. 'dog', 3
3. 'hello', 3
4. 'hello', 'dog', 3

3.5.4 Simulation: Hot Potato

One of the typical applications for showing a queue in action is to simulate a real situation that requires data to be managed in a FIFO manner. To begin, let's consider the children's game Hot Potato. In this game (see Figure 3.13) children line up in a circle and pass an item from neighbour to neighbour as fast as they can. At a certain point in the game, the action is stopped and the child who has the item (the potato) is removed from the circle. Play continues until only one child is left.

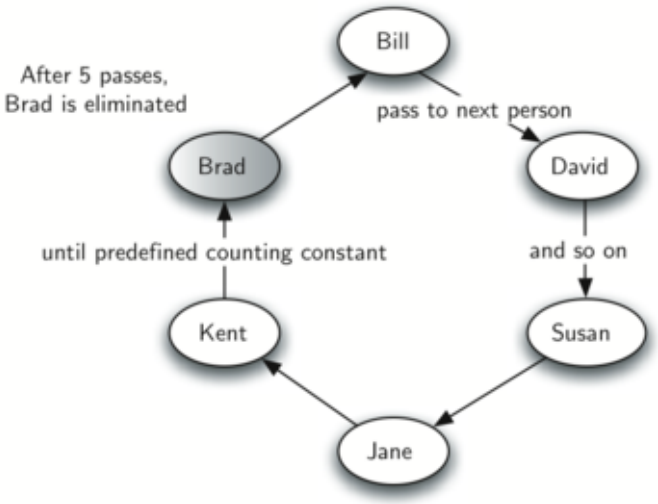

Figure 3.13: A Six Person Game of Hot Potato)

This game is a modern-day equivalent of the famous Josephus problem. Based on a legend about the famous first-century historian Flavius Josephus, the story is told that in the Jewish revolt against Rome, Josephus and 39 of his comrades held out against the Romans in a cave. With defeat imminent, they decided that they would rather die than be slaves to the Romans. They arranged themselves in a circle. One man was designated as number one, and proceeding clockwise they killed every seventh man. Josephus, according to the legend, was among other things an accomplished mathematician. He instantly figured out where he ought to sit in order to be the last to go. When the time came, instead of killing himself, he joined the Roman side. You can find many different versions of this story. Some count every third man and some allow the last man to escape on a horse. In any case, the idea is the same.

We will implement a general simulation of Hot Potato. Our program will input a list of names and a constant, call it "num" to be used for counting. It will return the name of the last person remaining after repetitive counting by num. What happens at that point is up to you.

To simulate the circle, we will use a queue (see Figure 3.14). Assume that the child holding the potato will be at the front of the queue. Upon passing the potato, the simulation will simply dequeue and then immediately enqueue that child, putting her at the end of the line. She will then wait until all the others have been at the front before it will be her turn again. After num dequeue/enqueue operations, the child at the front will be removed permanently and another cycle will begin. This process will continue until only one name remains (the size of the queue is 1).

A call to the hot_potato function using 7 as the counting constant returns Susan.

```
import Queue  # As previously defined

def hot_potato(name_list, num):
    sim_queue = Queue()
    for name in name_list:
        sim_queue.enqueue(name)
```

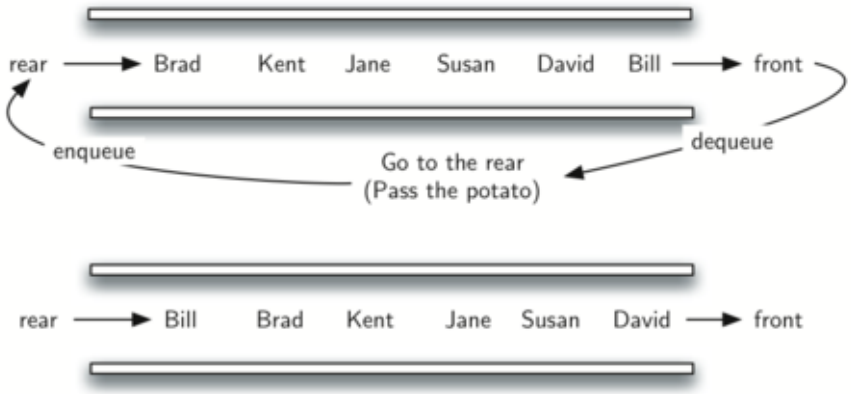

Figure 3.14: A Queue Implementation of Hot Potato)

```
    while sim_queue.size() > 1:
        for i in range(num):
            sim_queue.enqueue(sim_queue.dequeue())

        sim_queue.dequeue()

    return sim_queue.dequeue()

print(hot_potato(["Bill", "David", "Susan", "Jane", "Kent",
    "Brad"], 7))
```

Note that in this example the value of the counting constant is greater than the number of names in the list. This is not a problem since the queue acts like a circle and counting continues back at the beginning until the value is reached. Also, notice that the list is loaded into the queue such that the first name on the list will be at the front of the queue. Bill in this case is the first item in the list and therefore moves to the front of the queue. A variation of this implementation, described in the exercises, allows for a random counter.

3.5.5 Simulation: Printing Tasks

A more interesting simulation allows us to study the behavior of the printing queue described earlier in this section. Recall that as students send printing tasks to the shared printer, the tasks are placed in a queue to be processed in a first-come first-served manner. Many questions arise with this configuration. The most important of these might be whether the printer is capable of handling a certain amount of work. If it cannot, students will be waiting too long for printing and may miss their next class.

Consider the following situation in a computer science laboratory. On any average day about 10 students are working in the lab at any given hour. These students typically print up to twice during that time, and the length of these tasks ranges from 1 to 20 pages. The printer in the lab is older, capable of processing 10 pages per minute of draft quality. The printer could be switched to give better quality, but then it would produce only five pages per minute. The slower printing speed could make students wait too long. What page rate should be used?

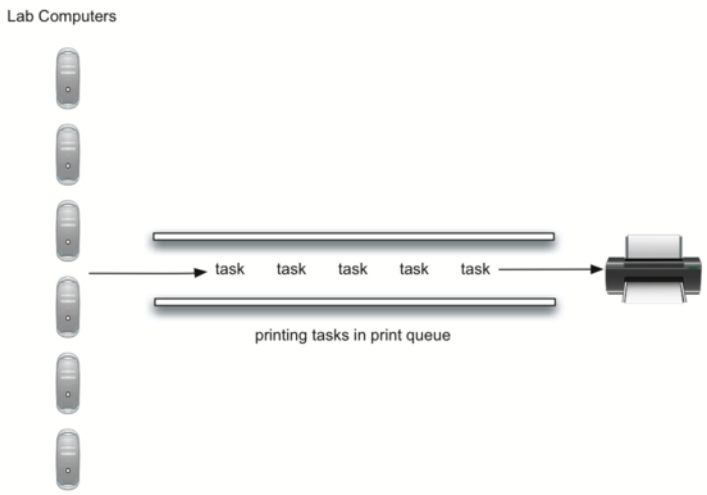

Figure 3.15: Computer Science Laboratory Printing Queue)

We could decide by building a simulation that models the laboratory. We will need to construct representations for students, printing tasks, and the printer (Figure 3.15). As students submit printing tasks, we will add them to a waiting list, a queue of print tasks attached to the printer. When the printer completes a task, it will look at the queue to see if there are any remaining tasks to process. Of interest for us is the average amount of time students will wait for their papers to be printed. This is equal to the average amount of time a task waits in the queue.

To model this situation we need to use some probabilities. For example, students may print a paper from 1 to 20 pages in length. If each length from 1 to 20 is equally likely, the actual length for a print task can be simulated by using a random number between 1 and 20 inclusive. This means that there is equal chance of any length from 1 to 20 appearing.

If there are 10 students in the lab and each prints twice, then there are 20 print tasks per hour on average. What is the chance that at any given second, a print task is going to be created? The way to answer this is to consider the ratio of tasks to time. Twenty tasks per hour means that on average there will be one task every 180 seconds:

$$\frac{20 \text{ tasks}}{1 \text{ hour}} \times \frac{1 \text{ hour}}{60 \text{ minutes}} \times \frac{1 \text{ minute}}{60 \text{ seconds}} = \frac{1 \text{ task}}{180 \text{ seconds}}$$

For every second we can simulate the chance that a print task occurs by generating a random number between 1 and 180 inclusive. If the number is 180, we say a task has been created. Note that it is possible that many tasks could be created in a row or we may wait quite a while for a task to appear. That is the nature of simulation. You want to simulate the real situation as closely as possible given that you know general parameters.

3.5.6 Main Simulation Steps

Here is the main simulation.

1. Create a queue of print tasks. Each task will be given a timestamp upon its arrival. The queue is empty to start.

2. For each second (current_second):
    - Does a new print task get created? If so, add it to the queue with the current_second as the timestamp.
    - If the printer is not busy and if a task is waiting,
        - Remove the next task from the print queue and assign it to the printer.
        - Subtract the timestamp from the current_second to compute the waiting time for that task.
        - Append the waiting time for that task to a list for later processing.
        - Based on the number of pages in the print task, figure out how much time will be required.
    - The printer now does one second of printing if necessary. It also subtracts one second from the time required for that task.
    - If the task has been completed, in other words the time required has reached zero, the printer is no longer busy.
3. After the simulation is complete, compute the average waiting time from the list of waiting times generated.

3.5.7 Python Implementation

To design this simulation we will create classes for the three real-world objects described above: Printer, Task, and PrintQueue.

The Printer class will need to track whether it has a current task. If it does, then it is busy and the amount of time needed can be computed from the number of pages in the task. The constructor will also allow the pages-per-minute setting to be initialized. The tick method decrements the internal timer and sets the printer to idle if the task is completed.

```
class Printer:
   def __init__(self, ppm):
      self.page_rate = ppm
      self.current_task = None
      self.time_remaining = 0

   def tick(self):
      if self.current_task != None:
         self.time_remaining = self.time_remaining - 1
         if self.time_remaining <= 0:
            self.current_task = None

   def busy(self):
      if self.current_task != None:
         return True
      else:
         return False
```

```
    def start_next(self,new_task):
        self.current_task = new_task
        self.time_remaining = new_task.get_pages() * 60 /
            self.page_rate
```

The Task class will represent a single printing task. When the task is created, a random number generator will provide a length from 1 to 20 pages. We have chosen to use the randrange function from the random module.

```
>>> import random
>>> random.randrange(1, 21)
18
>>> random.randrange(1, 21)
8
>>>
```

Each task will also need to keep a timestamp to be used for computing waiting time. This timestamp will represent the time that the task was created and placed in the printer queue. The wait_time method can then be used to retrieve the amount of time spent in the queue before printing begins.

```
import random

class Task:
    def __init__(self, time):
        self.timestamp = time
        self.pages = random.randrange(1, 21)

    def get_stamp(self):
        return self.timestamp

    def get_pages(self):
        return self.pages

    def wait_time(self, current_time):
        return current_time - self.timestamp
```

The main simulation implements the algorithm described above. The print_queue object is an instance of our existing queue ADT. A boolean helper function, new_print_task, decides whether a new printing task has been created. We have again chosen to use the randrange function from the random module to return a random integer between 1 and 180. Print tasks arrive once every 180 seconds. By arbitrarily choosing 180 from the range of random integers, we can simulate this random event. The simulation function allows us to set the total time and the pages per minute for the printer.

```
import Queue  # As previously defined
import Printer  # As previously defined
```

```
import Task  # As previously defined

import random

def simulation(num_seconds, pages_per_minute):

    lab_printer = Printer(pages_per_minute)
    print_queue = Queue()
    waiting_times = []

    for current_second in range(num_seconds):

      if new_print_task():
         task = Task(current_second)
         print_queue.enqueue(task)

      if (not lab_printer.busy()) and (not print_queue.is_empty()):
        next_task = print_queue.dequeue()
        waiting_times.append(next_task.wait_time(current_second))
        lab_printer.start_next(next_task)

      lab_printer.tick()

    average_wait = sum(waiting_times) / len(waiting_times)
    print("Average Wait %6.2f secs %3d tasks remaining."
       %(average_wait, print_queue.size()))

def new_print_task():
    num = random.randrange(1, 181)
    if num == 180:
       return True
    else:
       return False

for i in range(10):
    simulation(3600, 5)
```

When we run the simulation, we should not be concerned that the results are different each time. This is due to the probabilistic nature of the random numbers. We are interested in the trends that may be occurring as the parameters to the simulation are adjusted. Here are some results.

First, we will run the simulation for a period of 60 minutes (3,600 seconds) using a page rate of five pages per minute. In addition, we will run 10 independent trials. Remember that because the simulation works with random numbers each run will return different results.

```
>>>for i in range(10):
     simulation(3600, 5)

Average Wait 165.38 secs 2 tasks remaining.
Average Wait 95.07 secs 1 tasks remaining.
```

```
Average Wait 65.05 secs 2 tasks remaining.
Average Wait 99.74 secs 1 tasks remaining.
Average Wait 17.27 secs 0 tasks remaining.
Average Wait 239.61 secs 5 tasks remaining.
Average Wait 75.11 secs 1 tasks remaining.
Average Wait 48.33 secs 0 tasks remaining.
Average Wait 39.31 secs 3 tasks remaining.
Average Wait 376.05 secs 1 tasks remaining.
```

After running our 10 trials we can see that the mean average wait time is 122.155 seconds. You can also see that there is a large variation in the average weight time with a minimum average of 17.27 seconds and a maximum of 239.61 seconds. You may also notice that in only two of the cases were all the tasks completed.

Now, we will adjust the page rate to 10 pages per minute, and run the 10 trials again, with a faster page rate our hope would be that more tasks would be completed in the one hour time frame.

```
>>>for i in range(10):
     simulation(3600, 10)

Average Wait 1.29 secs 0 tasks remaining.
Average Wait 7.00 secs 0 tasks remaining.
Average Wait 28.96 secs 1 tasks remaining.
Average Wait 13.55 secs 0 tasks remaining.
Average Wait 12.67 secs 0 tasks remaining.
Average Wait 6.46 secs 0 tasks remaining.
Average Wait 22.33 secs 0 tasks remaining.
Average Wait 12.39 secs 0 tasks remaining.
Average Wait 7.27 secs 0 tasks remaining.
Average Wait 18.17 secs 0 tasks remaining.
```

The code to run the simulation is as follows:

```
import Queue  # As previously defined

import random

# Completed program for the printer simulation

class Printer:
   def __init__(self, ppm):
      self.page_rate = ppm
      self.current_task = None
      self.time_remaining = 0

   def tick(self):
      if self.current_task != None:
         self.time_remaining = self.time_remaining - 1
         if self.time_remaining <= 0:
```

```
            self.current_task = None

    def busy(self):
        if self.current_task != None:
            return True
        else:
            return False

    def start_next(self, new_task):
        self.current_task = new_task
        self.time_remaining = new_task.get_pages() * 60/self.page_rate

class Task:
    def __init__(self, time):
        self.timestamp = time
        self.pages = random.randrange(1, 21)

    def get_stamp(self):
        return self.timestamp

    def get_pages(self):
        return self.pages

    def wait_time(self, current_time):
        return current_time - self.timestamp


def simulation(num_seconds, pages_per_minute):

    lab_printer = Printer(pages_per_minute)
    print_queue = Queue()
    waiting_times = []

    for current_second in range(num_seconds):

      if new_print_task():
         task = Task(current_second)
         print_queue.enqueue(task)

      if (not lab_printer.busy()) and (not print_queue.is_empty()):
        next_task = print_queue.dequeue()
        waiting_times.append(next_task.wait_time(current_second))
        lab_printer.startNext(next_task)

      lab_printer.tick()

    average_wait = sum(waiting_times) / len(waiting_times)
    print("Average Wait %6.2f secs %3d tasks remaining."
        %(average_wait, print_queue.size()))

def new_print_task():
```

```
    num = random.randrange(1, 181)
    if num == 180:
        return True
    else:
        return False

for i in range(10):
    simulation(3600, 5)
```

3.5.8 Discussion

We were trying to answer a question about whether the current printer could handle the task load if it were set to print with a better quality but slower page rate. The approach we took was to write a simulation that modeled the printing tasks as random events of various lengths and arrival times.

The output above shows that with 5 pages per minute printing, the average waiting time varied from a low of 17 seconds to a high of 376 seconds (about 6 minutes). With a faster printing rate, the low value was 1 second with a high of only 28. In addition, in 8 out of 10 runs at 5 pages per minute there were print tasks still waiting in the queue at the end of the hour.

Therefore, we are perhaps persuaded that slowing the printer down to get better quality may not be a good idea. Students cannot afford to wait that long for their papers, especially when they need to be getting on to their next class. A six-minute wait would simply be too long.

This type of simulation analysis allows us to answer many questions, commonly known as "what if" questions. All we need to do is vary the parameters used by the simulation and we can simulate any number of interesting behaviors. For example,

- What if enrolment goes up and the average number of students increases by 20?
- What if it is Saturday and students are not needing to get to class? Can they afford to wait?
- What if the size of the average print task decreases since Python is such a powerful language and programs tend to be much shorter?

These questions could all be answered by modifying the above simulation. However, it is important to remember that the simulation is only as good as the assumptions that are used to build it. Real data about the number of print tasks per hour and the number of students per hour was necessary to construct a robust simulation.

Self Check

How would you modify the printer simulation to reflect a larger number of students? Suppose that the number of students was doubled. You make need to make some reasonable assumptions about how this simulation was put together but what would you change? Modify the code. Also suppose that the length of the average print task was cut in half. Change the code to reflect that change. Finally How would you parameterize the number of students, rather than changing the code we would like to make the number of students a parameter of the simulation.

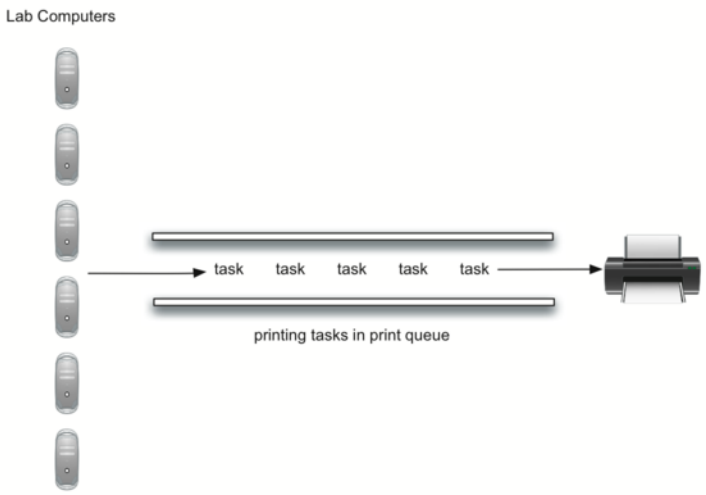

Figure 3.16: A Deque of Python Data Objects Queue)

3.6 Deques

We will conclude this introduction to basic data structures by looking at another variation on the theme of linear collections. However, unlike stack and queue, the deque (pronounced "deck") has very few restrictions. Also, be careful that you do not confuse the spelling of "deque" with the queue removal operation "dequeue."

3.6.1 What Is a Deque?

A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure. Figure 3.16 shows a deque of Python data objects.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.

3.6.2 The Deque Abstract Data Type

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

- Deque() creates a new deque that is empty. It needs no parameters and returns an empty deque.
- add_front(item) adds a new item to the front of the deque. It needs the item and returns nothing.

Deque Operation	Deque Contents	Return value
d.is_empty()	[]	True
d.add_rear(4)	[4]	
d.add_rear('dog')	['dog',4,]	
d.add_front('cat')	['dog',4,'cat']	
d.add_front(True)	['dog',4,'cat',True]	
d.size()	['dog',4,'cat',True]	4
d.is_empty()	['dog',4,'cat',True]	False
d.add_rear(8.4)	[8.4,'dog',4,'cat',True]	
d.remove_rear()	['dog',4,'cat',True]	8.4
d.remove_front()	['dog',4,'cat']	True

Table 3.6: Examples of Deque Operations

- add_rear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.
- remove_front() removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- remove_rear() removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- is_empty() tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the deque. It needs no parameters and returns an integer.

As an example, if we assume that d is a deque that has been created and is currently empty, then Table 3.6 shows the results of a sequence of deque operations. Note that the contents in front are listed on the right. It is very important to keep track of the front and the rear as you move items in and out of the collection as things can get a bit confusing.

3.6.3 Implementing a Deque in Python

As we have done in previous sections, we will create a new class for the implementation of the abstract data type deque. Again, the Python list will provide a very nice set of methods upon which to build the details of the deque. Our implementation will assume that the rear of the deque is at position 0 in the list.

```
# Completed implementation of a deque ADT
class Deque:
   def __init__(self):
      self.items = []

   def is_empty(self):
      return self.items == []

   def add_front(self, item):
```

```
        self.items.append(item)

    def add_rear(self, item):
        self.items.insert(0,item)

    def remove_front(self):
        return self.items.pop()

    def remove_rear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

In remove_front we use the pop method to remove the last element from the list. However, in remove_rear, the pop(0) method must remove the first element of the list. Likewise, we need to use the insert method in add_rear since the append method assumes the addition of a new element to the end of the list.

You can see many similarities to Python code already described for stacks and queues. You are also likely to observe that in this implementation adding and removing items from the front is $O(1)$ whereas adding and removing from the rear is $O(n)$. This is to be expected given the common operations that appear for adding and removing items. Again, the important thing is to be certain that we know where the front and rear are assigned in the implementation.

3.6.4 Palindrome Checker

An interesting problem that can be easily solved using the deque data structure is the classic palindrome problem. A **palindrome** is a string that reads the same forward and backward, for example, *radar*, *toot*, and *madam*. We would like to construct an algorithm to input a string of characters and check whether it is a palindrome.

The solution to this problem will use a deque to store the characters of the string. We will process the string from left to right and add each character to the rear of the deque. At this point, the deque will be acting very much like an ordinary queue. However, we can now make use of the dual functionality of the deque. The front of the deque will hold the first character of the string and the rear of the deque will hold the last character (see Figure 3.17)

Since we can remove both of them directly, we can compare them and continue only if they match. If we can keep matching first and the last items, we will eventually either run out of characters or be left with a deque of size 1 depending on whether the length of the original string was even or odd. In either case, the string must be a palindrome.

```
import Deque  # As previously defined

def pal_checker(a_string):
    char_deque = Deque()

    for ch in a_string:
```

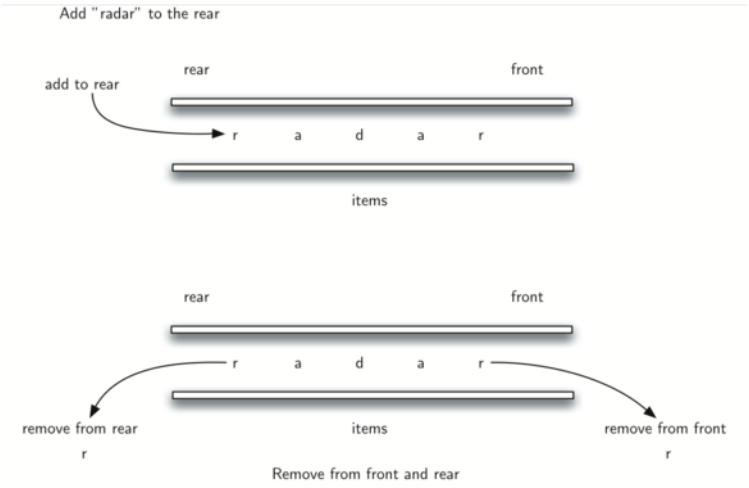

Figure 3.17: A Deque)

```
        char_deque.add_rear(ch)

    still_equal = True

    while char_deque.size() > 1 and still_equal:
        first = char_deque.remove_front()
        last = char_deque.remove_rear()
        if first != last:
            still_equal = False

    return still_equal

print(pal_checker("lsdkjfskf"))
print(pal_checker("radar"))
```

3.7 Lists

Throughout the discussion of basic data structures, we have used Python lists to implement the abstract data types presented. The list is a powerful, yet simple, collection mechanism that provides the programmer with a wide variety of operations. However, not all programming languages include a list collection. In these cases, the notion of a list must be implemented by the programmer.

A list is a collection of items where each item holds a relative position with respect to the others. More specifically, we will refer to this type of list as an unordered list. We can consider the list as having a first item, a second item, a third item, and so on. We can also refer to the beginning of the list (the first item) or the end of the list (the last item). For simplicity we will assume that lists cannot contain duplicate items.

For example, the collection of integers $54, 26, 93, 17, 77$, and 31 might represent a simple unordered list of exam scores. Note that we have written them as comma-delimited values,

a common way of showing the list structure. Of course, Python would show this list as $[54, 26, 93, 17, 77, 31]$.

3.8 The Unordered List Abstract Data Type

The structure of an unordered list, as described above, is a collection of items where each item holds a relative position with respect to the others. Some possible unordered list operations are given below.

- List() creates a new list that is empty. It needs no parameters and returns an empty list.
- add(item) adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- remove(item) removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- search(item) searches for the item in the list. It needs the item and returns a boolean value.
- is_empty() tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the list. It needs no parameters and returns an integer.
- append(item) adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.
- index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- insert(pos,item) adds a new item to the list at position pos. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position pos.
- pop() removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- pop(pos) removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

3.9 Implementing an Unordered List: Linked Lists

In order to implement an unordered list, we will construct what is commonly known as a **linked list**. Recall that we need to be sure that we can maintain the relative positioning of the items. However, there is no requirement that we maintain that positioning in contiguous memory. For example, consider the collection of items shown in Figure 3.18. It appears that these values have been placed randomly. If we can maintain some explicit information in each item, namely

Figure 3.18: Items Not Constrained in Their Physical Placement

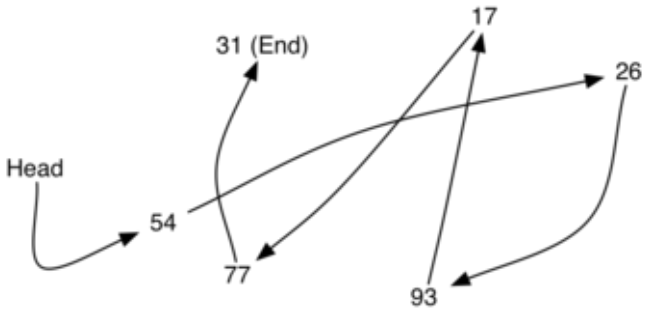

Figure 3.19: Relative Positions Maintained by Explicit Links.

the location of the next item (see Figure 3.19, then the relative position of each item can be expressed by simply following the link from one item to the next.

It is important to note that the location of the first item of the list must be explicitly specified. Once we know where the first item is, the first item can tell us where the second is, and so on. The external reference is often referred to as the head of the list. Similarly, the last item needs to know that there is no next item.

3.9.1 The Node Class

The basic building block for the linked list implementation is the **node**. Each node object must hold at least two pieces of information. First, the node must contain the list item itself. We will call this the **data field** of the node. In addition, each node must hold a reference to the next node. To construct a node, you need to supply the initial data value for the node. Evaluating the assignment statement below will yield a node object containing the value 93 (see Figure 3.20). You should note that we will typically represent a node object as shown in Figure 3.21. The Node class also includes the usual methods to access and modify the data and the next reference.

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

   def get_data(self):
      return self.data
```

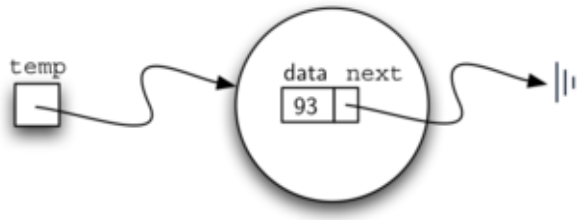

Figure 3.20: A Node Object Contains the Item and a Reference to the Next Node

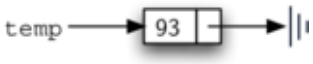

Figure 3.21: A Typical Representation for a Node.

```
def get_next(self):
    return self.next

def set_data(self, new_data):
    self.data = newdata

def set_next(self, new_next):
    self.next = new_next
```

We create Node objects in the usual way.

```
>>> temp = Node(93)
>>> temp.get_data()
93
```

The special Python reference value None will play an important role in the Node class and later in the linked list itself. A reference to None will denote the fact that there is no next node. Note in the constructor that a node is initially created with next set to None. Since this is sometimes referred to as "grounding the node," we will use the standard ground symbol to denote a reference that is referring to None. It is always a good idea to explicitly assign None to your initial next reference values.

3.9.2 The Unordered List Class

As we suggested above, the unordered list will be built from a collection of nodes, each linked to the next by explicit references. As long as we know where to find the first node (containing the first item), each item after that can be found by successively following the next links. With this in mind, the UnorderedList class must maintain a reference to the first node. The following code shows the constructor. Note that each list object will maintain a single reference to the head of the list.

```
class UnorderedList:
```

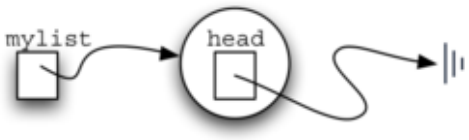

Figure 3.22: An Empty List

Figure 3.23: Linked List of Integers

```
   def __init__(self):
      self.head = None
```

Initially when we construct a list, there are no items. The assignment statement

```
>>> mylist = UnorderedList()
```

creates the linked list representation shown in Figure 3.22. As we discussed in the Node class, the special reference None will again be used to state that the head of the list does not refer to anything. Eventually, the example list given earlier will be represented by a linked list as shown in Figure 3.23. The head of the list refers to the first node which contains the first item of the list. In turn, that node holds a reference to the next node (the next item) and so on. It is very important to note that the list class itself does not contain any node objects. Instead it contains a single reference to only the first node in the linked structure.

The is_empty method simply checks to see if the head of the list is a reference to None. The result of the boolean expression self.head==None will only be true if there are no nodes in the linked list. Since a new list is empty, the constructor and the check for empty must be consistent with one another. This shows the advantage to using the reference None to denote the "end" of the linked structure. In Python, None can be compared to any reference. Two references are equal if they both refer to the same object. We will use this often in our remaining methods.

```
def is_empty(self):
   return self.head == None
```

So, how do we get items into our list? We need to implement the add method. However, before we can do that, we need to address the important question of where in the linked list to place the new item. Since this list is unordered, the specific location of the new item with respect to the other items already in the list is not important. The new item can go anywhere. With that in mind, it makes sense to place the new item in the easiest location possible.

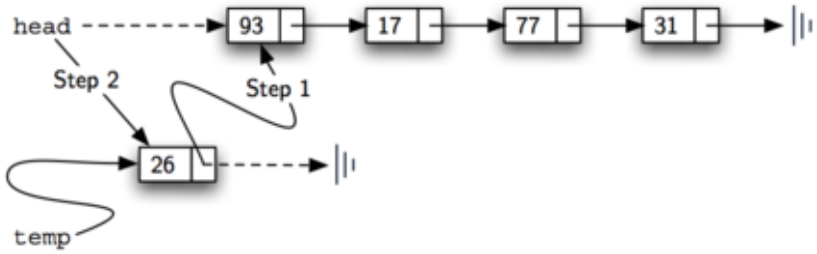

Figure 3.24: Adding a New Node is a Two-Step Process

Recall that the linked list structure provides us with only one entry point, the head of the list. All of the other nodes can only be reached by accessing the first node and then following next links. This means that the easiest place to add the new node is right at the head, or beginning, of the list. In other words, we will make the new item the first item of the list and the existing items will need to be linked to this new first item so that they follow.

The linked list shown in Figure 3.23 was built by calling the add method a number of times.

```
>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)
```

Note that since 31 is the first item added to the list, it will eventually be the last node on the linked list as every other item is added ahead of it. Also, since 54 is the last item added, it will become the data value in the first node of the linked list.

The add method is shown below. Each item of the list must reside in a node object. Line 2 creates a new node and places the item as its data. Now we must complete the process by linking the new node into the existing structure. This requires two steps as shown in Figure 3.24. Step 1 (line 3) changes the next reference of the new node to refer to the old first node of the list. Now that the rest of the list has been properly attached to the new node, we can modify the head of the list to refer to the new node. The assignment statement in line 4 sets the head of the list.

The order of the two steps described above is very important. What happens if the order of line 3 and line 4 is reversed? If the modification of the head of the list happens first, the result can be seen in Figure 3.25. Since the head was the only external reference to the list nodes, all of the original nodes are lost and can no longer be accessed.

```
def add(self, item):
    temp = Node(item)
    temp.set_next(self.head)
    self.head = temp
```

The next methods that we will implement-size, search, and remove-are all based on a technique known as linked list traversal. Traversal refers to the process of systematically visiting each

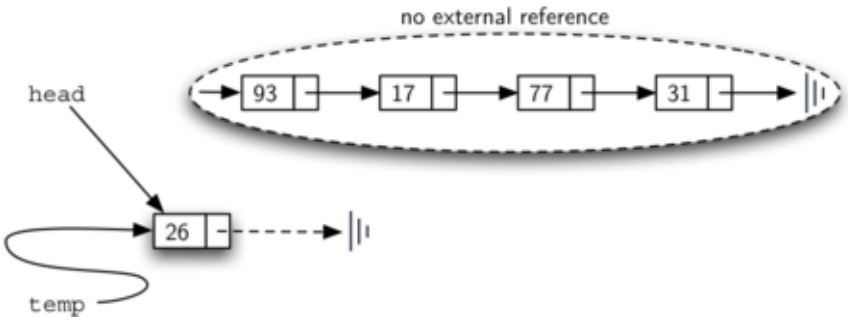

Figure 3.25: Result of Reversing the Order of the Two Steps.

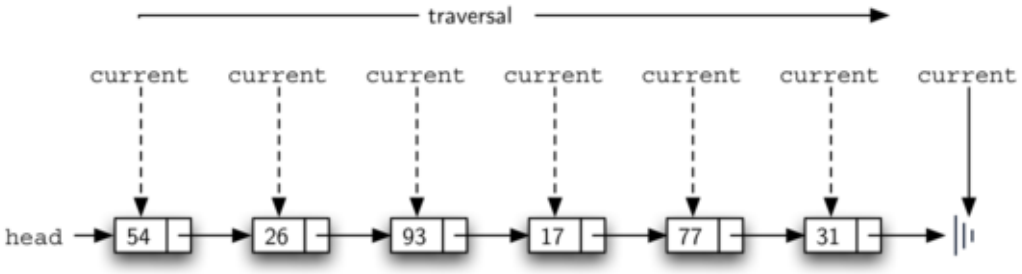

Figure 3.26: Traversing the Linked List from the Head to the End.

node. To do this we use an external reference that starts at the first node in the list. As we visit each node, we move the reference to the next node by "traversing" the next reference.

To implement the size method, we need to traverse the linked list and keep a count of the number of nodes that occurred. Below we show the Python code for counting the number of nodes in the list. The external reference is called current and is initialized to the head of the list in line 2. At the start of the process we have not seen any nodes so the count is set to 0. Lines 4–6 actually implement the traversal. As long as the current reference has not seen the end of the list (None), we move current along to the next node via the assignment statement in line 6. Again, the ability to compare a reference to None is very useful. Every time current moves to a new node, we add 1 to count. Finally, count gets returned after the iteration stops. Figure 3.26 shows this process as it proceeds down the list.

```
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.get_next()

    return count
```

Searching for a value in a linked list implementation of an unordered list also uses the traversal technique. As we visit each node in the linked list we will ask whether the data stored there matches the item we are looking for. In this case, however, we may not have to traverse all the way to the end of the list. In fact, if we do get to the end of the list, that means that the item we are looking for must not be present. Also, if we do find the item, there is no need to continue.

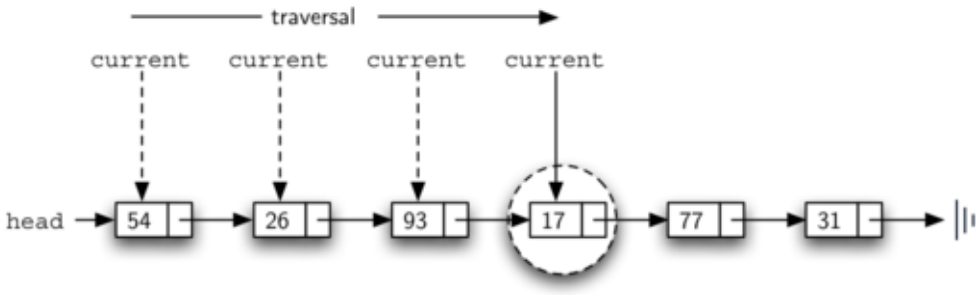

Figure 3.27: Successful Search for the Value 17

The code below shows the implementation for the search method. As in the size method, the traversal is initialized to start at the head of the list (line 2). We also use a boolean variable called found to remember whether we have located the item we are searching for. Since we have not found the item at the start of the traversal, found can be set to False (line 3). The iteration in line 4 takes into account both conditions discussed above. As long as there are more nodes to visit and we have not found the item we are looking for, we continue to check the next node. The question in line 5 asks whether the data item is present in the current node. If so, found can be set to True.

```
def search(self,item):
   current = self.head
   found = False
   while current != None and not found:
      if current.get_data() == item:
         found = True
      else:
         current = current.get_next()

   return found
```

As an example, consider invoking the search method looking for the item 17.

```
>>> mylist.search(17)
True
```

Since 17 is in the list, the traversal process needs to move only to the node containing 17. At that point, the variable found is set to True and the while condition will fail, leading to the return value seen above. This process can be seen in Figure 3.27

he remove method requires two logical steps. First, we need to traverse the list looking for the item we want to remove. Once we find the item (recall that we assume it is present), we must remove it. The first step is very similar to search. Starting with an external reference set to the head of the list, we traverse the links until we discover the item we are looking for. Since we assume that item is present, we know that the iteration will stop before current gets to None. This means that we can simply use the boolean found in the condition.

When found becomes True, current will be a reference to the node containing the item to be removed. But how do we remove it? One possibility would be to replace the value of the

item with some marker that suggests that the item is no longer present. The problem with this approach is the number of nodes will no longer match the number of items. It would be much better to remove the item by removing the entire node.

In order to remove the node containing the item, we need to modify the link in the previous node so that it refers to the node that comes after current. Unfortunately, there is no way to go backward in the linked list. Since current refers to the node ahead of the node where we would like to make the change, it is too late to make the necessary modification.

The solution to this dilemma is to use two external references as we traverse down the linked list. current will behave just as it did before, marking the current location of the traverse. The new reference, which we will call previous, will always travel one node behind current. That way, when current stops at the node to be removed, previous will be referring to the proper place in the linked list for the modification.

The code below shows the complete remove method. Lines 2–3 assign initial values to the two references. Note that current starts out at the list head as in the other traversal examples. previous, however, is assumed to always travel one node behind current. For this reason, previous starts out with a value of None since there is no node before the head (see Figure 3.28). The boolean variable found will again be used to control the iteration.

In lines 6–7 we ask whether the item stored in the current node is the item we wish to remove. If so, found can be set to True. If we do not find the item, previous and current must both be moved one node ahead. Again, the order of these two statements is crucial. previous must first be moved one node ahead to the location of current. At that point, current can be moved. This process is often referred to as "inch-worming" as previous must catch up to current before current moves ahead. Figure 3.29 shows the movement of previous and current as they progress down the list looking for the node containing the value 17.

```
1  def remove(self, item):
2      current = self.head
3      previous = None
4      found = False
5      while not found:
6          if current.get_data() == item:
7              found = True
8          else:
9              previous = current
10             current = current.get_next()
11
12     if previous == None:
13         self.head = current.get_next()
14     else:
15         previous.set_next(current.get_next())
```

Once the searching step of the remove has been completed, we need to remove the node from the linked list. Figure 3.30 shows the link that must be modified. However, there is a special case that needs to be addressed. If the item to be removed happens to be the first item in the list, then current will reference the first node in the linked list. This also means that previous will be None. We said earlier that previous would be referring to the node whose next reference

Figure 3.28: Initial Values for the previous and current references

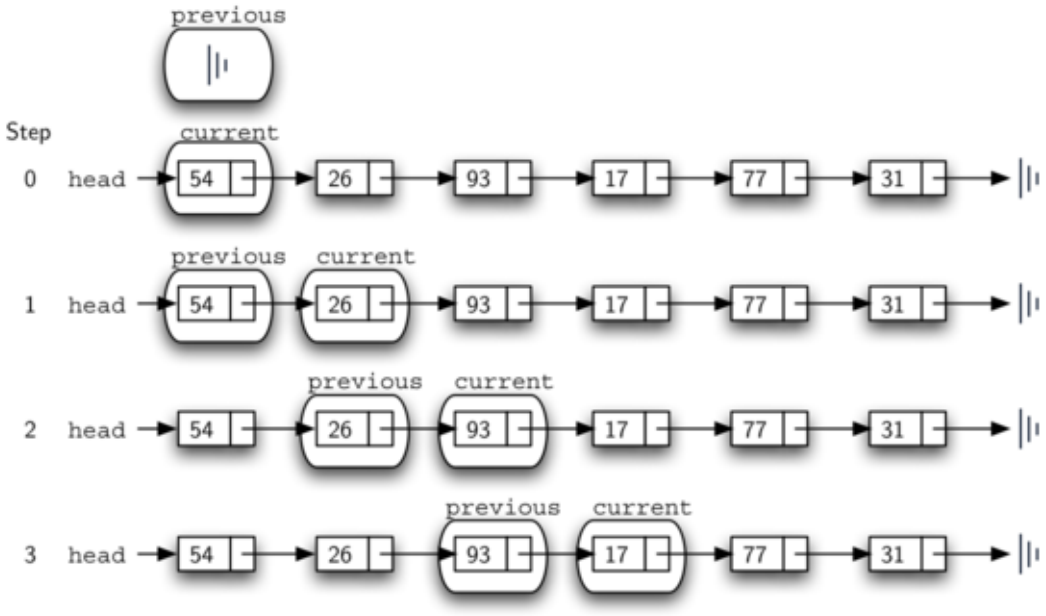

Figure 3.29: previous and current move down the list

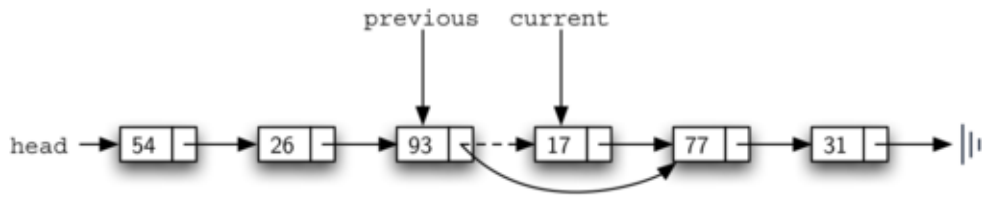

Figure 3.30: Removing an Item from the middle of the list

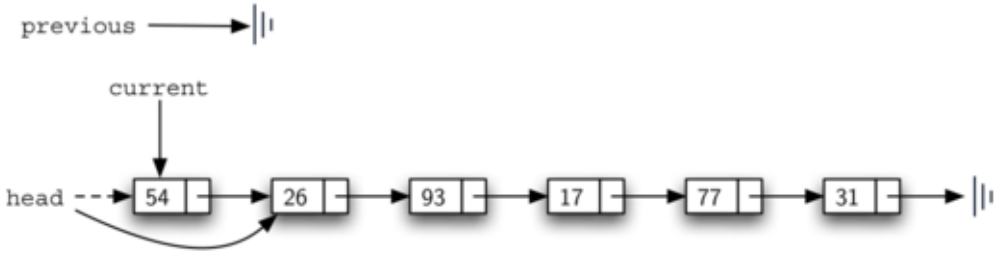

Figure 3.31: Removing the first node from the list

needs to be modified in order to complete the remove. In this case, it is not previous but rather the head of the list that needs to be changed (see Figure 3.31)

Line 12 allows us to check whether we are dealing with the special case described above. If previous did not move, it will still have the value None when the boolean found becomes True. In that case (line 13) the head of the list is modified to refer to the node after the current node, in effect removing the first node from the linked list. However, if previous is not None, the node to be removed is somewhere down the linked list structure. In this case the previous reference is providing us with the node whose next reference must be changed. Line 15 uses the set_next method from previous to accomplish the removal. Note that in both cases the destination of the reference change is current.get_next(). One question that often arises is whether the two cases shown here will also handle the situation where the item to be removed is in the last node of the linked list. We leave that for you to consider.

The remaining methods append, insert, index, and pop are left as exercises. Remember that each of these must take into account whether the change is taking place at the head of the list or someplace else. Also, insert, index, and pop require that we name the positions of the list. We will assume that position names are integers starting with 0.

Self Check

Implement the append method for UnorderedList. What is the time complexity of the method you created? It was most likely $O(n)$. If you add an instance variable to the UnorderedList class you can create an append method that is $O(1)$. Modify your append to be $O(1)$. Be careful! To really do this correctly you will need to consider a couple of special cases that may require you to make a modification to the add method as well.

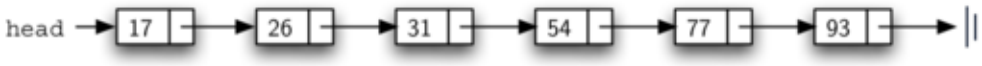

Figure 3.32: An Ordered Linked List

3.10 The Ordered List Abstract Data Type

We will now consider a type of list known as an ordered list. For example, if the list of integers shown above were an ordered list (ascending order), then it could be written as 17, 26, 31, 54, 77, and 93. Since 17 is the smallest item, it occupies the first position in the list. Likewise, since 93 is the largest, it occupies the last position.

The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item. The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined. Many of the ordered list operations are the same as those of the unordered list.

- OrderedList() creates a new ordered list that is empty. It needs no parameters and returns an empty list.
- add(item) adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
- remove(item) removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- search(item) searches for the item in the list. It needs the item and returns a boolean value.
- is_empty() tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the list. It needs no parameters and returns an integer.
- index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- pop() removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- pop(pos) removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

3.10.1 Implementing an Ordered List

In order to implement the ordered list, we must remember that the relative positions of the items are based on some underlying characteristic. The ordered list of integers given above (17, 26, 31, 54, 77, and 93) can be represented by a linked structure as shown in Figure 3.32. Again, the node and link structure is ideal for representing the relative positioning of the items.

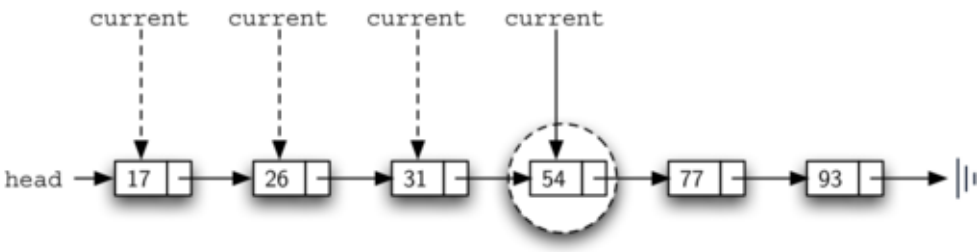

Figure 3.33: Searching an Ordered Linked List

To implement the OrderedList class, we will use the same technique as seen previously with unordered lists. Once again, an empty list will be denoted by a head reference to None.

```
class OrderedList:
   def __init__(self):
      self.head = None
```

As we consider the operations for the ordered list, we should note that the is_empty and size methods can be implemented the same as with unordered lists since they deal only with the number of nodes in the list without regard to the actual item values. Likewise, the remove method will work just fine since we still need to find the item and then link around the node to remove it. The two remaining methods, search and add, will require some modification.

The search of an unordered linked list required that we traverse the nodes one at a time until we either find the item we are looking for or run out of nodes (None). It turns out that the same approach would actually work with the ordered list and in fact in the case where we find the item it is exactly what we need. However, in the case where the item is not in the list, we can take advantage of the ordering to stop the search as soon as possible.

For example, Figure 3.33 shows the ordered linked list as a search is looking for the value 45. As we traverse, starting at the head of the list, we first compare against 17. Since 17 is not the item we are looking for, we move to the next node, in this case 26. Again, this is not what we want, so we move on to 31 and then on to 54. Now, at this point, something is different. Since 54 is not the item we are looking for, our former strategy would be to move forward. However, due to the fact that this is an ordered list, that will not be necessary. Once the value in the node becomes greater than the item we are searching for, the search can stop and return False. There is no way the item could exist further out in the linked list.

The following code shows the complete search method. It is easy to incorporate the new condition discussed above by adding another boolean variable, stop, and initializing it to False (line 4). While stop is False (not stop) we can continue to look forward in the list (line 5). If any node is ever discovered that contains data greater than the item we are looking for, we will set stop to True (lines 9 – –10). The remaining lines are identical to the unordered list search.

```
1  def search(self, item):
2     current = self.head
3     found = False
4     stop = False
5     while current != None and not found and not stop:
```

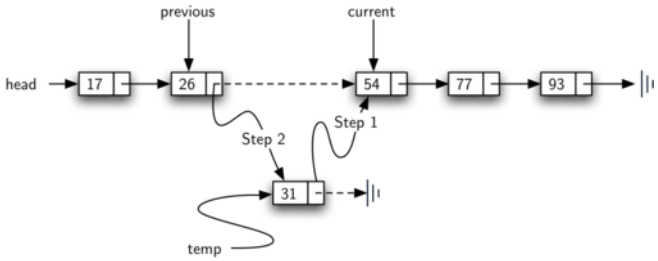

Figure 3.34: Adding an Item to an Ordered Linked List

```
6         if current.get_data() == item:
7             found = True
8         else:
9             if current.get_data() > item:
10                stop = True
11            else:
12                current = current.get_next()
13
14    return found
```

The most significant method modification will take place in add. Recall that for unordered lists, the add method could simply place a new node at the head of the list. It was the easiest point of access. Unfortunately, this will no longer work with ordered lists. It is now necessary that we discover the specific place where a new item belongs in the existing ordered list.

Assume we have the ordered list consisting of 17, 26, 54, 77, and 93 and we want to add the value 31. The add method must decide that the new item belongs between 26 and 54. Figure 3.34 shows the setup that we need. As we explained earlier, we need to traverse the linked list looking for the place where the new node will be added. We know we have found that place when either we run out of nodes (current becomes None) or the value of the current node becomes greater than the item we wish to add. In our example, seeing the value 54 causes us to stop.

As we saw with unordered lists, it is necessary to have an additional reference, again called previous, since current will not provide access to the node that must be modified. The below code shows the complete add method. Lines $2 - -3$ set up the two external references and lines $9 - -10$ again allow previous to follow one node behind current every time through the iteration. The condition (line 5) allows the iteration to continue as long as there are more nodes and the value in the current node is not larger than the item. In either case, when the iteration fails, we have found the location for the new node.

The remainder of the method completes the two-step process shown in Figure 3.34. Once a new node has been created for the item, the only remaining question is whether the new node will be added at the beginning of the linked list or some place in the middle. Again, previous == None (line 13) can be used to provide the answer.

```
1 def add(self, item):
2     current = self.head
3     previous = None
4     stop = False
```

```
    while current != None and not stop:
        if current.get_data() > item:
            stop = True
        else:
            previous = current
            current = current.get_next()

    temp = Node(item)
    if previous == None:
        temp.set_next(self.head)
        self.head = temp
    else:
        temp.set_next(current)
        previous.set_next(temp)
```

3.10.2 Analysis of Linked Lists

To analyze the complexity of the linked list operations, we need to consider whether they require traversal. Consider a linked list that has n nodes. The is_empty method is $O(1)$ since it requires one step to check the head reference for None. size, on the other hand, will always require n steps since there is no way to know how many nodes are in the linked list without traversing from head to end. Therefore, length is $O(n)$. Adding an item to an unordered list will always be $O(1)$ since we simply place the new node at the head of the linked list. However, search and remove, as well as add for an ordered list, all require the traversal process. Although on average they may need to traverse only half of the nodes, these methods are all $O(n)$ since in the worst case each will process every node in the list.

You may also have noticed that the performance of this implementation differs from the actual performance given earlier for Python lists. This suggests that linked lists are not the way Python lists are implemented. The actual implementation of a Python list is based on the notion of an array. We discuss this in more detail in another chapter.

3.11 Summary

- Linear data structures maintain their data in an ordered fashion.
- Stacks are simple data structures that maintain a LIFO, last-in first-out, ordering.
- The fundamental operations for a stack are push, pop, and is_empty.
- Queues are simple data structures that maintain a FIFO, first-in first-out, ordering.
- The fundamental operations for a queue are enqueue, dequeue, and is_empty.
- Prefix, infix, and postfix are all ways to write expressions.
- Stacks are very useful for designing algorithms to evaluate and translate expressions.
- Stacks can provide a reversal characteristic.

- Queues can assist in the construction of timing simulations.
- Simulations use random number generators to create a real-life situation and allow us to answer "what if" types of questions.
- Deques are data structures that allow hybrid behavior like that of stacks and queues.
- The fundamental operations for a deque are add_front, add_rear, remove_front, remove_rear, and is_empty.
- Lists are collections of items where each item holds a relative position.
- A linked list implementation maintains logical order without requiring physical storage requirements.
- Modification to the head of the linked list is a special case.

3.12 Key Terms

balanced parentheses
data field
deque
first-in first-out (FIFO)
fully parenthesized
head
infix
last-in first-out (LIFO)
linear data structure
linked list
linked list traversal
list
node
palindrome
postfix
precedence
prefix
queue
simulation
stack