Chapter 11 - Stacks, queues, linked lists, trees, and graphs

Overview

Although the collection classes presented in previous chapters are sufficient for most tasks, several other structures for holding collections of objects are also commonly used. The most important of them are stacks, queues, linked lists, trees, and graphs. This chapter is an introduction to these structures with emphasis on intuitive rather than most efficient implementations. For a more advanced treatment, we recommend one of the many books on data structures.

A stack is a collection whose elements can be accessed only at one end called the top of the stack. The operation adding an element on the top of the stack is called push, the operation removing the top element from the stack is called pop. Implementing stacks in Smalltalk does not require a new class because stack behavior is subsumed by OrderedCollection. Our coverage will thus be limited to several examples of the uses of the stack.

A queue is a collection in which elements are added at one end and retrieved at the other. Its familiar real-life example is a line in a bank. Queues do not require a new class because their behavior is also a part of the behavior of OrderedCollection and our presentation will thus be limited to examples.

A linked list is a linearly arranged collection of elements that allows insertion and deletion at any place in the sequence. This behavior is necessary in many applications and not easily achieved in the collections presented so far. Smalltalk's class LinkedList implements a basic linked list.

A tree is a structure whose graphical representation looks like a family tree: It starts with a root at the top, and branches downward. Typical uses of trees are the representation of the class hierarchy, storing data for fast access, and translation of program code. Computer applications use many kinds of trees but Smalltalk does not contain a general-purpose tree class. We will develop a class implementing the simplest kind of tree - the binary tree.

Graphs can be used to represent concepts such as road maps, house plumbing diagrams, and telephone networks. They consist of nodes and connections between them. Graphs have many different applications but Smalltalk does not have any built-in graph classes because the system does not need them. We will design and implement a graph class and demonstrate a few typical graph operations.

11.1 Stack - an access-at-top-only collection

A stack is usually defined with reference to a stack of cafeteria trays. New objects are added on the top by the push operation, and existing elements can only be removed by the pop operation which removes the top element (Figure 11.1). For obvious reasons, a stack is also called a last-in first-out (LIFO) collection. Stacks are very important in several areas of theoretical Computer Science and in the process of computing.

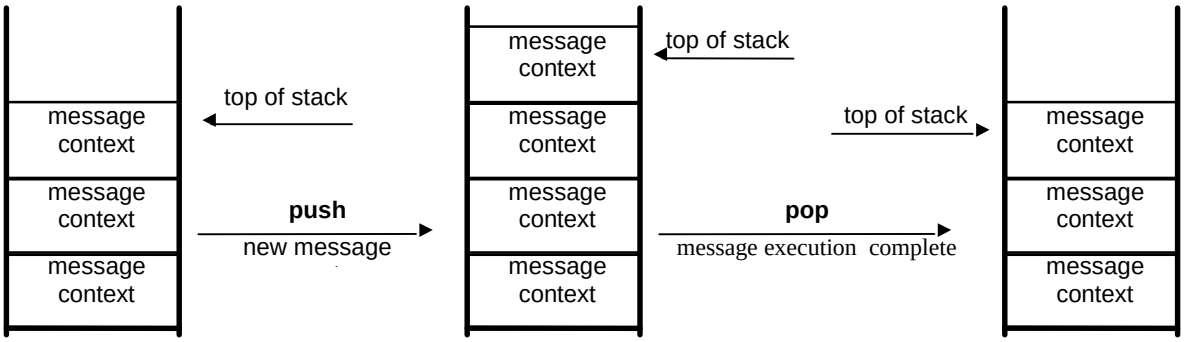


Figure 11.1. Execution of Smalltalk messages is based on a stack of 'contexts'.

In essence, a stack is an ordered collection whose elements can only be accessed at one end. If we treat the start of the collection as the top of the stack, addFirst: performs push and removeFirst performs pop. Alternatively, we can use the end of the OrderedCollection as the top of the stack with addLast: for push and removeLast for pop. If we need a stack and if we want to restrict ourselves to the stack-like behavior of OrderedCollection, there is thus no need to define a new Stack class and this is the approach taken in the built-in VisualWorks library. From a strict OO point of view, however, this approach is not appropriate because it leaves the simulated stack object open to all behaviors of oo instead of restricting it to the very small behavior of stacks.

In the following, we will restrict our coverage to two examples from the Smalltalk environment and leave an implementation of a Stack class as an assignment. Our first example is a behavior that resembles stacks but is not really a stack, the second is a very important use of a stack at the core of Smalltalk implementation.

Example 1: The stack-like behavior of the *paste* operation

The *paste* command in the text editor pop up menu can paste any of the recently cut or copied strings. To do this, press <shift> when selecting *paste*. This opens a dialog (Figure 11.2) displaying the most recent *copy* or *cut* string at the top and the oldest *copy* or *cut* string at the bottom.

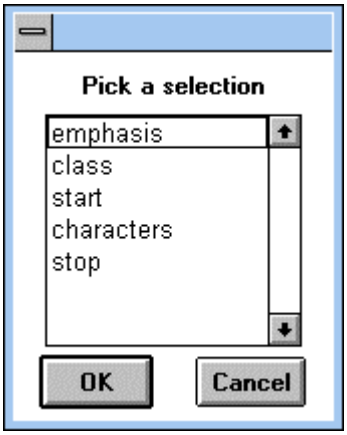


Figure 11.2. <shift> *paste* displays the latest copy/cut strings and allows selection.

Although this behavior is based on the stack principle and demonstrates its main purpose - keeping recent information accessible in the last-in first-out style - the structure is not strictly a stack: For one thing, the definition of the stack in class ParagraphEditor restricts its depth to five strings. To implement this restriction, updates of the OrderedCollection discard the element at the bottom when the size reaches five elements. Also, before adding a new string, the definition first checks whether the string already is on the top of the stack and if it is, it does not duplicate it. The main difference between the copy buffer and a true stack is that the user can select any string in the buffer and not only the top one.

It is interesting to note that the string buffer is held in a *class* variable of ParagraphEditor, making it available to any instance of ParagraphEditor. As a result, any of the last five strings copied from *any* text editor can be pasted into *any* text editor.

Although the paste buffer is not a pure implementation of a stack, its behavior is a nice illustration of the usefulness of the concept.

Example 2: Stack as the basis of message execution

When a program sends a message, the context associated with it - the code and the objects that the message needs to execute - are pushed on the top of the context stack. During execution, the Smalltalk *object engine* (the program responsible for managing program execution; also called the *virtual machine*) accesses this information to obtain the objects that it needs and the messages to send. When sending a message, the virtual machine creates a new context for this message, puts it on the stack, and starts its execution. When the message is finished, its context is popped from the stack and execution returns to the sending message (Figure 11.1).

The principle of message execution does not require further elaboration and we will thus dedicate the rest of this section to a brief discussion of contexts, the objects stored on context stacks. The reason for including this subject is its essential importance for understanding Smalltalk operation, and the fact that it illustrates the importance of stacks.

Context and related concepts

To start this discussion, we will first put message execution into the context of transformation of the source code into an executing program.

Before we can execute a method, its source code must be compiled, for example by clicking *accept*. In response to this, the compiler produces an internal representation of the code whose essential part is an instance of CompiledMethod (to be discussed shortly). We will now demonstrate how the compiling process works and what it produces.
Consider a class called Test and its instance method

```
test1: anInteger
        | temp |
        temp := anInteger factorial.
        ^temp
```

defined in protocol test. To see how this method is compiled and added to protocol test of class Test (with the same result as if you clicked *accept*) execute

```
self halt.
Test compile: 'test: anInteger |temp| temp := anInteger factorial. ^temp'
        classified: 'test'
```

When you observe the operation in the Debugger, you will find that execution of this expression consists of two main steps: In the first step, the code is compiled, producing a CompiledMethod object which is inserted into the method dictionary of class Test. In the second step, this object is complemented by the source code and other information. To see the result, *inspect*

```
Test compiledMethodAt: #test1:
```

The returned CompiledMethod object has several instance variables and the most interesting ones contain the source code of the method and the *bytecodes* stored in instance variable bytes. Under byte codes, you will find the following:

```
short CompiledMethod numArgs=1 numTemps=1 frameSize=12

literals: (#factorial )

1 <10> push local 0
2 <70> send factorial
3 <4D> store local 1; pop
```

```
4 <11> push local 1
5 <65> return
```

As you can see, a CompiledMethod contains information about the number of arguments of the method, the number of its temporary variables, a list of literals - the messages sent by this method (only #factorial in this case), and a sequence of bytecodes - an internal representation of the source code ready for execution by the stack-based virtual machine[1]. Let us now examine the bytecodes in more detail:

```
1 <10> push local 0
2 <70> send factorial
3 <4D> store local 1; pop
4 <11> push local 1
5 <65> return
```

The codes in brackets in the second column are expressed in hexadecimal notation, a shorthand for internal binary representation. They are the translation of the original source code and represent 'opcodes' of a fictitious Smalltalk CPU. This CPU does not in reality exist but is emulated by the virtual machine which interprets[2] the codes and produces the effect described next to the byte code. As an example, hexadecimal code 10 has the effect of pushing the value of the first argument of the message on the stack of intermediate results. We will now illustrate how the "interpreter" executes the byte codes representing the method, assuming the following situation:

```
test
"Some method. containing test1."
        ...
        Test new test1: 5
        ...
```

When the virtual machine encounters, for example, the message test1: 20 (its CompiledMethod equivalent produced by the compiler), it puts its context on the context stack (more on this later) and creates an *evaluation stack* to hold the intermediate results of execution (Figure 11.3). It then starts executing the byte codes of the test1: method one after another starting with the code in position 1:

1. Code 10: Push the value of argument 20 ('local object 0') on the evaluation stack.
2. Code 70: Send message factorial ('literal 0') to the object on the top of the evaluation stack. This finds and executes the CompiledMethod with the byte codes of factorial (not shown), and leaves the result (SmallInteger 720) on the top of evaluation the stack, replacing the original object (SmallInteger 20). Control returns to the test1: method. (This factorial message send is executed in the same way as the test1: message that we are now tracing.)
3. Code 4D: Stores the value on the top of the evaluation stack (the result of 20 factorial) in temp ('local 1') and pops the stack, removing the 20 factorial value. This step is equivalent to the assignment part of the assignment statement in the source code.
4. Code 11: Push the temp object ('local 1') on the evaluation stack.
5. Code 65: Return to the message that sent test: (in this case message test), pushing the value of temp - the value that test1 was supposed to return - on the top of its evaluation stack.

---

[1] For more on the virtual machine, see Appendix 8.

[2] The statement about 'interpretation' was strictly true for earlier implementations of Smalltalk but modern implementations translate bytecodes into the machine code of the CPU running the program when the method is first invoked during execution. This process, called *dynamic compilation* or *just in time (JIT) compilation* makes execution more efficient. Once compiled, the machine code is stored in a code cache so that it does not have to be retranslated.

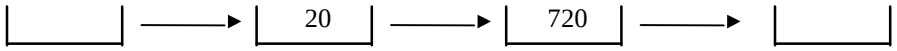


Figure 11.3. Effect of execution of test1: on its evaluation stack. In this case, the execution stack never contains more than one element but other methods may require a deeper stack.

Let us now analyze what information the virtual machine needs to have, to be able to do what we have just described. To execute a message, the virtual machine must know

- the *bytecodes* of the message with information about its arguments and temporary variables
- the *sender* of the message (to be able to transfer the result back to it)
- the *receiver* of the message (required by self, super, and for access to instance variables)
- an *evaluation stack* for calculations required by byte codes
- the *current position* in the execution of the byte code sequence; this object is referred to as the *program counter* or the PC of the virtual machine.

The object containing all this information is called a *context* and it is an instance of class MethodContext.
For a concrete illustration of these concepts, create the following method

```
test2
| context temp |
        context := thisContext.     "thisContext is a special variable like self and super. It returns the
                                     currently active context."
        temp := 3 + 7 * 5 factorial.
        ^temp
```

in class Test and execute

```
self halt.
Test new test2
```

In the Debugger, execute the message step-by-step and observe the changing value of PC in the context variable. Observe also the stack and the stack pointer.

Now that you understand the use of a context in execution, let's look at an example of execution in a broader setting. First, create the following three methods in class Test:

```
test3
| context |
        context := thisContext.
        self test4.
        ^self

test4
| context |
        context := thisContext.
        self test5.
        ^self

test5
| context |
        context := thisContext.
        ^self
```

Next, execute the following expression and observe how control passes from one method to another, how their contexts are pushed on top of one another in the context stack, and how they are popped when execution of the method ends:

```
self halt. Test new test3
```

These examples illustrate the critical importance of the stack in Smalltalk: At all times, Smalltalk operation directly depends on two stacks - one for contexts (representing message code), the other for intermediate results of the currently executing message. Whereas the context stack is shared by all contexts, each invoked message has its own working stack associated with its context.

Main lessons learned:

- A stack is a last-in first-out (LIFO) structure. Elements are added on the top using the push operation and removed using the pop operation.
- The behavior of a stack is included in the behavior of OrderedCollection and there is no need for a Stack class.
- Execution of Smalltalk messages depends on a stack of context objects, each of them carrying all information about a message, its receiver and sender, its arguments and local variables, and current state of execution.
- Translation of the source code of a method into a form executable on the virtual machine is expressed in bytecodes.
- A part of the context of each executing method is an evaluation stack holding intermediate results.

Exercises

1. Examine the nature of the copy buffer; in particular, check whether its elements must be strings. If not, can you think of some additional uses of the copy buffer?
2. When implementing a stack, is it better to use the end or the start of an OrderedCollection as the top?
3. Implement class Stack with only the essential stack behavior.
4. While debugging , you might want to print the names of selected messages whenever they are sent. You could, of course, use the show: message with the name of the selector explicitly spelled out as in Transcript show: 'Executing with:with:with:' but this is somewhat awkward. A neater solution is to extract the name of the method from the context. Implement this approach via a new method called printMethodName:.
5. Browse context-related classes and write a short description.

11.2 Context Stack and Exceptions

As another illustration of the use of stacks, we will now implement *Tester*, a tool to help automate the testing of classes. Testing is a very serious concern in software development because all code must be carefully tested before it is delivered to customers. Because testing generally requires verifying program behavior with many test conditions, the process may be very time consuming and expensive. To minimize the effort and time required for this part of program development, software tools have been developed to automate the process as much as possible. In Smalltalk, these test programs often assume that a class under test contains special testing methods in a special protocol, search these methods out, and execute them. We will develop a very simple version of such a program.

Our test tool (to be implemented by class Tester) will allow the user to select the class to be tested, locate all the test methods defined in the class, execute them, and write a report to the Transcript. The user interface will be as in Figure 11.4. For each of the executed methods, Tester will print the name of the method followed by the result (*success* or *failure*) in the Transcript. If a test method fails, Tester also prints

a message to the Transcript. All test methods of a class are assumed to be in class protocol testing, and each test must return true if it succeeds and false if it fails.

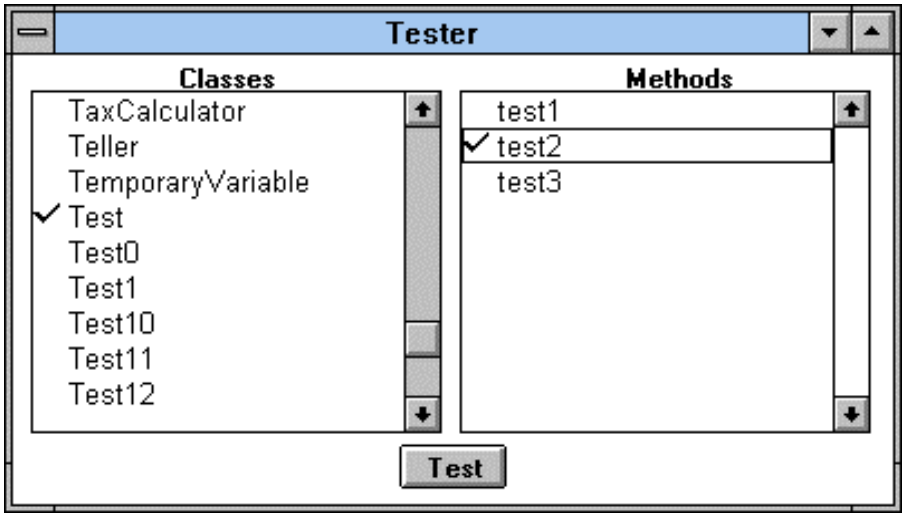


Figure 11.4. Desired user interface for Tester.

The required functionality is described by the following scenarios:

Scenario 1: User selects a class and test methods and successfully runs the test
Conversation:
1. *User* clicks class to be tested.
2. *System* displays the list of all methods defined in its class protocol testing.
3. *User* selects methods to be executed and clicks *Test*.
4. *System* executes the selected methods and prints report in Transcript.

Scenario 2: User selects a class and test methods, one of the methods fails during execution
Conversation:
1. *User* clicks class.
2. *System* displays list of corresponding test methods.
3. *User* selects methods to be executed and clicks *Test*.
4. *System* starts executing the selected methods and one of them fails as 'not understood'.
5. *System* displays appropriate note in the Transcript and executes the remaining methods.

Solution: Scenario 1 can be implemented rather easily but Scenario 2 introduces a problem that we have not yet encountered: The Tester must be capable of completing execution even if a method fails to execute - technically speaking, even when an exception occurs. In the following, we will show how to deal with this problem without explaining how the mechanism works. (We will also skip the user interface of Tester and leave it as an exercise.) In the next section, we will explain the principle of exception handling in VisualWorks and show that it relies on the context stack.
Preliminary considerations. To implement the given scenarios, Tester must know how to

1. obtain a list of class names
2. obtain a list of methods in class protocol testing
3. execute a method and recover if it raises an exception.

As we know, class names are returned by Smalltalk classNames. To obtain information about classes, we already learned to use category Kernel - Classes and this is indeed where we find an answer to the second question (see also Appendix 3): Instance variable organization in ClassDescription contains an instance of ClassOrganizer that holds information about a class's categories and method names. Sending listAtCategoryNamed: aSymbol to this object returns the names of all methods in the specified protocol. As an example, the following expression returns the names of the selectors of all *instance* methods defined in protocol #update in class Object:

```
Object organization listAtCategoryNamed: #updating
```

and

```
Test class organization listAtCategoryNamed: #testing
```

returns all *class* methods under protocol #testing in class Test.

Finally, the third requirement. To execute a message 'safely' and recover when an exception occurs, we must use the handle:do: message with appropriate arguments. We will explain the details later but for now, suffice it to say that Smalltalk contains a number of predefined 'signal' objects that correspond to various error conditions such as 'division by zero', 'index out of bounds', and 'message not understood', and when such a condition occurs, the signal object can be used to trigger exception-recovery behavior. As an example, the first of the following statements will attempt to execute 3/0, intercept the attempt to divide by zero, write 'Division by zero' to the Transcript, and continue with the next statement instead of opening an Exception window:

```
ArithmeticValue divisionByZeroSignal
        handle: [:exception| Transcript cr; show: 'Division by zero'.
                exception return]
        do: [ 3 / 0].
Transcript cr; show: 'Look ma, no exception window'
```

As you can see, the block argument of the do: part of the handle:do: message contains the operation that we would like to execute, and the block argument of the handle: keyword specifies what to do if the do: block fails. The handle: block has one argument - the Exception object. In our block, we sent message return to this argument to request that if the exception occurs, execution should return to the original code (our test program) and continue. Class Exception provides various other behaviors explained in the User Guide.

Let's now return to our problem. Assuming that the only possible cause of failure is 'message not understood', the following method will execute each test method, print *success* or *failure* if the method executes (and returns true or false), and print a message to the Transcript if the 'message not understood' exception occurs:

```
testClass: classToTest methods: methodsToRun
"Execute specified test methods and print report to Transcript."
        methodsToRun isEmpty ifTrue: [Dialog warn: 'No method selected'].
        Transcript clear; show: 'Results of test on class ' , classToTest name; cr.
        methodsToRun
                do: [:method |
                        Transcript cr; show: method asString; tab.
                        Object messageNotUnderstoodSignal
                                handle: [:exception |
                                        Transcript show: 'Message not understood'.
                                        exception return]
                                do: [Transcript show: ((classToTest perform: method)
                                                ifTrue: ['success']
                                                ifFalse: ['failure'])]]
```

We assume that our test methods don't have any parameters but this restriction could be easily removed

To test whether the method works, we created a class called Test and defined the following three methods in its class protocol testing:

```
test1
"Should cause an exception."
        3 open.
        ^true

test2
"Should execute and return true."
        ^3 factorial = 6

test3
"Should execute and return false."
        ^3 squared = 10
```

Since we don't have the proper user interface, we tested our code by executing

```
Tester new testClass: Test methods:  #(#test1 #test2 #test3)
```

which produced the following expected result:

```
Results of test on class Test

test1    Message not understood
test2    success
test3    failure
```

Watch out and make sure that you put your Test test methods in the correct class protocol, otherwise you will get only 'message not understood' reports. We leave the full implementation of Tester with the prescribed user interface as an exercise.

Main lessons learned:

- VisualWorks Smalltalk has a built-in mechanism for dealing with exceptional situations. It allows the programmer to anticipate exceptional behaviors and deal with them programatically, preventing the program from raising an exception.
- Exception handling depends on instances of class Signal.

Exercises

1. Implement and test the Tester.
2. Generalize Tester to handle test methods with any number of arguments.

11.3 More about exceptions

To explain the internal operation of exceptions, we will now take our example from the previous section and trace its execution. The sequence is lengthy and we suggest that you read and execute it, read the summary, and reread the trace one more time. Our test code is as follows:

```
self halt.
ArithmeticValue divisionByZeroSignal
        handle: [:exception| Transcript cr; show: 'Division by zero'.
                exception return]
        do: [ 3 / 0].
Transcript cr; show: 'Look ma, no exception window'
```

and the main events that occur in its execution are as follows (Figure 11.5):

Expression ArithmeticValue divisionByZeroSignal returns the DivisionByZeroSignal object. This Signal object then executes handle:do: which is defined as follows:

```
handle: handlerBlock do: doBlock
"Execute doBlock. If an exception occurs whose Signal is included in the receiver, execute handlerBlock."
        <exception: #handle>
        ^doBlock value
```

The message first evaluates the doBlock which in our example invokes the division message 3/0. As this message executes, it eventually reaches the following method in class Fraction:

```
reducedNumerator: numInteger denominator: denInteger
"Answer a new Fraction numInteger/denInteger."
        | gcd denominator numerator |
        denominator := denInteger truncated abs.
        denominator isZero
                ifTrue: [^self raise: #divisionByZeroSignal
                                receiver: numInteger
                                selector: #/
                                arg: denInteger
                                errorString: 'Can"t create a Fraction with a zero denominator'].
        etc.
```

If the denominator argument is 0, the method sends raise:receiver:selector:arg:errorString: defined in the ArithmeticValue superclass of Fraction as follows:

```
raise: signalName receiver: anObject selector: aSymbol arg: anArg errorString: aMessage
        ^(self perform: signalName) raiseRequestWith: (MessageSend receiver: anObject
                selector: aSymbol argument: anArg) errorString: aMessage
```

This message essentially asks the appropriate Signal to ‘raise an exception request’. The part

```
self perform: signalName
```

returns DivisionByZeroSignal, and

```
MessageSend receiver: anObject selector: aSymbol argument: anArg
```

returns a MessageSend. This is an interesting object and we will digress briefly to explain it. Its *print it* execution in the debugger produces

a MessageSend with receiver: 3, selector: #/ and arguments: #(0)

which shows that this object knows the receiver, the selector, and the arguments of a message, and is an instance of class MessageSend, a subclass of Message. The comment of Message is as follows:

Class Message represents a selector and its argument values. Generally, the system does not use instances of Message. However, when a message is not understood by its receiver, the interpreter will make up a Message (to capture the information involved in an actual message transmission) and send it as an argument with the message doesNotUnderstand:.

In other words, instances of Message are not used to execute messages (messages are compiled from byte codes into machine code and executed by the CPU). But when a message is not understood, the system creates a Message, passes it up the inheritance chain from the original receiver to Object to produce an appropriate doesNotUnderstand: message. This explains the secret of doesNotUnderstand:.

Class MessageSend, which is what we are dealing with here, adds several new behaviors and information about the sender. Its class comment is as follows:

A MessageSend represents a specific invocation of a Message. It is essentially a message send represented as an object, and supports protocol for *evaluating* that send.

After this digression, let's return to our original problem. All arguments of raiseRequestWith:errorString: are now available and Signal executes it. This method is defined as follows:

```
raiseRequestWith: parameter errorString: aString
"Raise the receiver, that is, create an exception on the receiver and have it search the execution stack for a
handler that accepts the receiver, then evaluate the handler's exception block. The exception block may
choose to proceed if this message is sent. The exception will answer the first argument when asked for its
parameter, and will use aString as its error string"
        ^self newException
                parameter: parameter;
                errorString: aString;
                originator: thisContext sender homeReceiver;
                raiseRequest
```

The first thing that happens is self newException according to

```
newException
"Answer an appropriate new Exception object.Subclasses may wish to override this."
        ^Exception new signal: self
```

This returns a new Exception object whose main function is to know which kind of Signal is associated with it. In our case, this is the DivisionByZeroSignal signal. As the next step, raiseRequestWith:errorString: obtains some additional information and raiseRequest raises the exception, triggering a sequence of events to find the handler code. To do this, raiseRequest searches the context stack from the top down until it finds the context in which everything started, in our case the unboundMethod

```
self halt.
ArithmeticValue divisionByZeroSignal
        handle: [:exception| Transcript cr; show: 'Division by zero'.
                exception return]
        do: [ 3 / 0].
Transcript cr; show: 'Look ma, no exception window'
```

The highlighted handle: block is now evaluated, displays 'Division by zero' in the Transcript, and proceeds to exception return. This message 'unwinds' the context stack (the whole context stack is still there, the exception only accessed and executed the handler block), removing all contexts down to the

originating context and terminating their execution. Our unboundMethod context is now on the top of the stack and execution continues at the point where we left off in our test program.

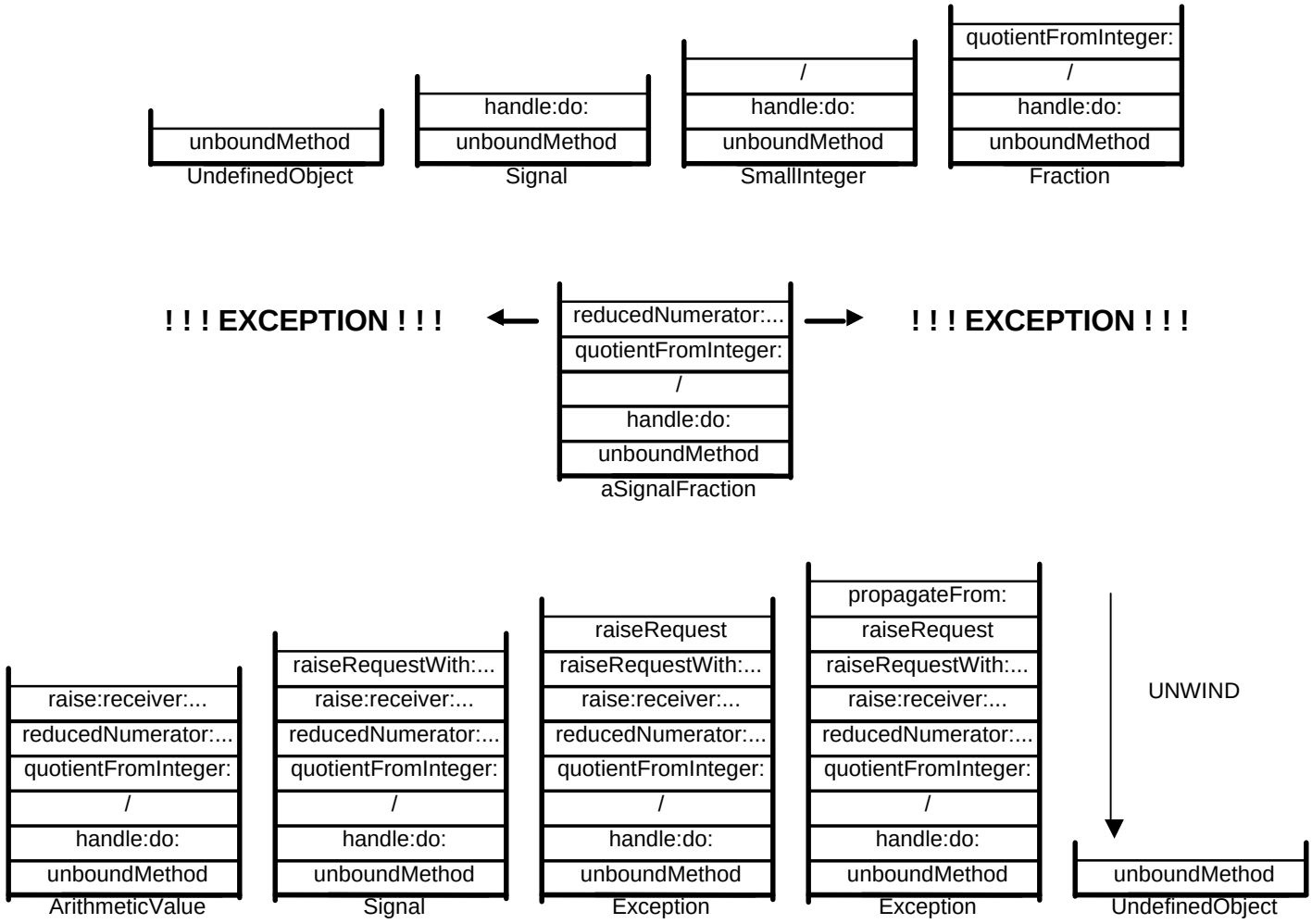


Figure 11.5. Behavior of the context stack in the execution of the test example. The class in which the currently active message is defined is shown below the context stack. Occurrence of exception is indicated.

Let's now summarize what we found: To execute a block of statements safely, let a Signal execute AS IN

aSignal handle: blockThatHandlesException do: blockToBeExecutedWhereExceptionCouldOccur

The sequence of events is as follows:

aSignal evaluates the do: block. If the block evaluates normally, execution continues to the next message. If a message in the do: block raises an exception addressed to aSignal, aSignal executes the following exception handling mechanism:

a. aSignal creates an Exception object.
b. The Exception object searches the context stack looking from top downward for the context of the Signal that raised the exception and executes its handler block – the argument of handle:.

c. In our case, the handler sends return and in response to this, the Exception object unwinds all contexts from the top of the context stack down to and including the handle:do: context and execution continues.

We have only scratched the surface of exception handling and we encourage you to explore it further. Exception handling is very important because many situations require catching illegal or special situations and handling them. Some of the examples are attempts to execute a message that the receiver does not understand, sending a mathematical method with inappropriate arguments, accessing an array with an index out of bounds, and failed file access. The principle of implementation of exceptions again shows the critical importance of stacks in Smalltalk's operation.

In closing, note that numerous signals and other ways of handling exceptions are predefined. The following classes contain the most useful signals: ArithmeticValue, BinaryStorage, ByteCodeStream, ByteEncodedStream, ClassBuilder, CodeStream, ColorValue, CompiledCode, Context, GraphicalContext, KeyboardEvent, Metaclass, Object, ObjectMemory, OSErrorHandler, Palette, ParagraphEditor, and Process. The programmer can also define new signals to intercept any desired conditions.

Main lessons learned:

- Signal delegates the handling of exceptions to an instance of Exception.
- Exception handling depends on the context stack.
- A number of Signal objects are built into the library and users can define their own as well.

Exercises

1. Assume that two other exceptions that might occur in our test methods are division by zero and subscript out of bounds. Modify Tester to intercept these exceptions. (Hint: Use class HandlerList.)
2. Modify Tester to catch *any* error condition. (Hint: Examine the Object protocol.)
3. Explain the role of the MessageSend object in exception handling.
4. MessageSend objects can be evaluated as in (MessageSend receiver: 3 selector: #factorial) value. Execute 3 + 4 and 3 between: 5 and: 17 using this technique.
5. What happens when you remove exception return from the handle: block?
6. What other messages can be sent to the Exception object in the handle: block and what is their effect?
7. Trace the operation of doesNotUndderstand: by executing $s factorial. Write a short description.

11.4. Queues

A queue is a collection of linearly ordered elements in which elements are added at one end and retrieved at the other end (Figure 11.6). As in a queue in a bank, the first item entering the queue is also the first to be retrieved and removed from the queue and this is why a queue is also called a first-in-first-out (FIFO) structure.

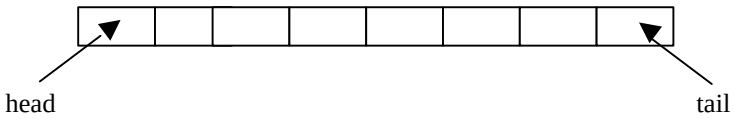


Figure 11.6. In a queue, elements are added at the tail and removed at the head.

In spite of its simplicity, the queue is a very important concept with many applications in simulation of real life events such as lines of customers at a cash register or cars waiting at an intersection, and in programming (such as printer jobs waiting to be processed. Many Smalltalk applications use a queue but instead of implementing it as a new class, they use an OrderedCollection because it performs all the required functions[3]. Since the concept is so simple, we will limit ourselves to an illustration of its use on two examples, one in this section and one in the next.

Simulating a bank queue

At the beginning of the design of a new bank outlet, the designer needs to know how many tellers to provide to satisfy the expected number of customers, and the management will need how to staff the tellers to guarantee satisfactory but economical handling of customers. In our example, the management wants to simulate the following situation in order to evaluate the need for tellers, the cost of operation, and the number of customers that can be handled.

Problem: A bank has a certain fixed number of teller stations (Figure 11.7). Customers arrive at unpredictable random times and queue up for service. There is a single queue and when a teller becomes available, the customer at the head of the queue goes to this teller. Each customer has an unpredictable number of transactions and takes an unpredictable amount of time to process. Our task is to develop and test a program to simulate this situation - but not to use it to make managerial decisions.

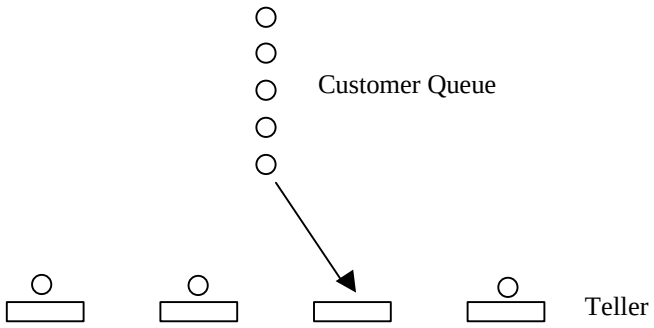


Figure 11.7. Bank layout.

Preliminary considerations: The task has several components. We must

- decide how to model the random arrivals of customers and their random servicing times,
- identify the external parameters that characterize the problem,
- identify expected results
- identify, design, and implement the classes involved in the simulation,
- decide on the user interface.

[3] Just as with stacks, a cleaner implementation would be a Queue class with behaviors limited to those required by a queue.

Modeling randomness

We will assume that the unpredictable parts of the behavior of the problem can be described by a probabilistic model. In other words, we will assume that we can formulate a mathematical function describing the probability that a new customer will join the queue within the next n minutes, and we will use a random number generator to generate customers according to this formula. We will use the same principle to determine how much time each individual customer spends at the teller.

Practitioners of simulation use a variety of probabilistic models in their simulations, all of them based on specific mathematical assumptions. Matching the situation at hand with the proper set of assumptions requires some knowledge of probability and an understanding of the domain that is being simulated and we will restrict ourselves to the simplest model – we will assume that the distribution of arrival times is *uniform*. In other words, we will assume that there is a certain minimum and maximum inter-arrival time, and that any time between these two limits is equally likely to occur. The advantages of this model are that it is simple and that generation of random numbers is already implemented by class Random, its disadvantage is that it does not describe a bank queue well. We will leave a more realistic model as an exercise.

External parameters

In order to make simulation possible, we must identify the parameters that must be specified to start a new simulation. From the specification of the problem and from our discussion of modeling of randomness, it is clear that we need the following parameters:

- Total number of tellers.
- Minimum inter-arrival time.
- Maximum inter-arrival time.
- Minimum expected time required to service a customer.
- Maximum expected time required to service a customer.
- Desired duration of simulation in terms of time or number of customers.

All time-related parameters are expressed in fictitious time units.

Expected results

We will restrict ourselves to creating a log listing customer arrival times, customer departure times, and the average length of the queue calculated over the whole simulation..

Desired user interface

The user interface must make it possible to enter any combination of input parameters and run a simulation. The input of data will be as in Figure 11.8 and the results will be printed in the Transcript ordered along the fictitious time axis.

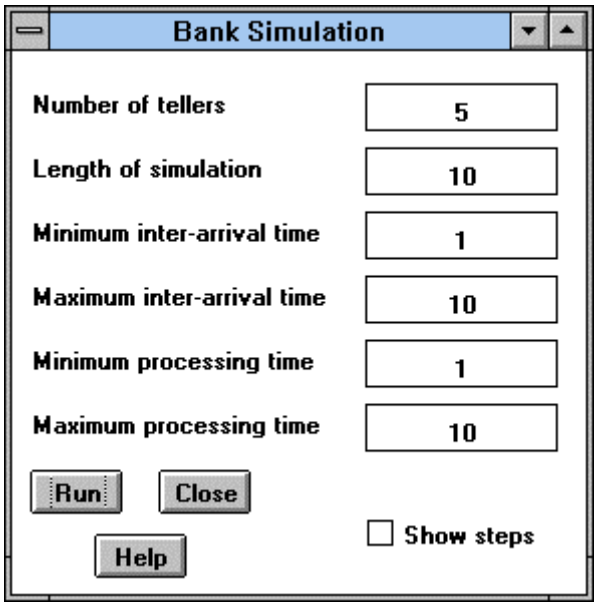


Figure 11.8. Desired user interface. The *Show steps* check box determines whether all customer transfers are output to the Transcript or not.

Class exploration

The objects immediately identifiable from the specification are customer objects (class Customer), tellers (class Teller), and the queue (class Queue). We also need an object to generate new customers and add them to the queue (class CustomerProducer). With this basic set of objects, let us explore how the simulation might proceed:

- The CustomerProducer generates a new customer and adds it to the queue. It then creates a random inter-arrival time t and another customer who will enter the queue at time t.
- The customer in the queue is allocated to the first available teller. (This is not quite fair because the first teller will be getting more work than others and we should allocate customers to tellers randomly. In our problem, such considerations are irrelevant because they don't change anything on the outcome.)
- The teller starts processing the customer and releases him or her after the amount of time required for processing by this particular customer object. This parameter is generated by CustomerProducer.
- From this point, execution continues along the following lines until the total length of simulation is completed:
    - If one or more tellers are available and a customer is waiting, the customer is sent to the first available teller.
    - When the inter-arrival time expires, the CustomerProducer generates a new customer object and adds it to the end of the queue. It also generates a new inter-arrival time to be used to generate a new customer.
    - Processing of customers by tellers is as explained above.

This algorithm suggests that the simulation is driven by time – the ticks of a fictitious clock determine when new customers are produced and when they are released by teller. We thus need a time managing object (class SimulationManager) whose responsibility will be to notify the CustomerProducer, Teller, and Queue objects when a unit of time expired. These objects will be responsible for taking an appropriate action. We also need to take care of the results. We will have the Queue and Teller objects report the appropriate information and the Queue object will be responsible for calculating the average

queue length. The output will, of course, be left to the application model (class BankSimulation) to display. This way, if we want to change the user interface (output of results) or make our simulation a part of a larger scheme, the domain objects can stay the same and only the application model must be changed.

Preliminary design of classes and their responsibilities

We have, so far, identified need for the following classes:

- BankSimulation. In charge of user interface: Input of parameters, start of simulation, output of results.
- Customer. Knows when it entered the queue and how much processing time it will require from a teller.
- CustomerProducer. Generates Customer objects with random values of processing time, keeps information about inter-arrival time and uses it to produce another customer when the time arrives.
- Queue. Knows its customers, can accept new Customer objects, knows how to check whether a Teller is available, knows how to send a Customer to a Teller. Calculates queue statistics and notifies BankSimulation when a customer is added.
- SimulationManager. Starts, runs, and ends simulation. Collects results at the end and notifies BankSimulation. Issues timing information to CustomerProducer, Queue, and Teller objects.
- Teller. Knows how to accept and process a customer, knows whether it has a Customer. Notifies BankSimulation when a customer arrives or is released.

Is this set of classes complete? To find out, we will re-execute our informal scenario dividing it into three phases: simulation start up, body (repeated over and over until the end of simulation), and end of simulation. The following is a rather detailed description with comments on the feasibility of individual steps within the existing class descriptions:

Phase	Description of step.	Comment
Start up	BankSimulation gets valid parameters and starts SimulationManager.	OK
	SimulationManager initializes time and asks CustomerProducer, Queue, and Teller to initialize themselves.	OK
	CustomerProducer generates a Customer with *random processing time* and adds it to the Queue.	*How?*
	Queue tells BankSimulation that it received a Customer.	OK
	BankSimulation outputs the event to Transcript.	OK
	Queue sends Customer to the first available Teller.	OK
	Teller tells BankSimulation that it received a Customer.	OK
	BankSimulation outputs the event to Transcript.	OK
	CustomerProducer generates another Customer with *random processing time* and assigns it a *random waiting time* (customer 'waits in fron of the bank'). It will now wait to release the customer to Queue.	*How?*
Body	SimulationManager increments time and checks for end of simulation. If not end, it informs CustomerProducer, Queue, and Teller about time change.	OK
	CustomerProducer checks whether to release a Customer. If so, it sends Customer to Queue which updates and notifies BankSimulation. BankSimulation outputs the event to Transcript. CustomerProducer creates new Customer and inter-arrival (waiting) time.	OK
	Each Teller checks whether to release its Customer. If so, it releases Customer and notifies BankSimulation which outputs the event to Transcript.	OK
	Queue checks whether it has a waiting Customer. If so, it checks whether a Teller is available.	OK

	If so, it sends Customer to the first available Teller, Teller calculates time to release this Customer, notifies BankSimulation of arrival, BankSimulation outputs the event to Transcript. Repeated until either no more customers in queue or all tellers busy.	
End of simulation	SimulationManager calculates average wait time and sends this information and total number of customers to BankSimulation.	OK
	BankSimulation outputs the result to Transcript.	OK

Our conclusions from this analysis are as follows:

- We need a new class to generate random numbers uniformly distributed between two positive integers (we will call this class RandomInteger).
- We have reassigned the responsibility for reporting events and results to the objects that are involved in the events. This has the following consequences:
    - Queue needs to know about BankSimulation so that it can communicate results.
    - Teller needs to know about BankSimulation so that it can communicate results.
    - Since Queue is created by SimulationManager, SimulationManager must know about BankSimulation.

The last point is worth closer examination. In our problem, we have two choices to implement the generation of the report. One is to have BankSimulation poll all objects that have something to report, the other is to leave it to the components to notify BankSimulation (Figure 11.9). The first solution has several disadvantages: One is that whenever we change the simulation by adding new types of objects, we must modify the corresponding part of BankSimulation. Another disadvantage is that as the number of objects that have something to report gets larger, the complexity of methods in BankSimulation that perform the gathering of reports also increases. Eventually, BankSimulation will become much too large and its role in the application too predominant. Finally, polling is based on the assumption that the polled object may or may not have something to report. If it does not, polling wastes time. The second approach - leaving the responsibility to report an event to the object that experienced the event (*event-driven* design) can be more efficient. We selected the event-driven approach and leave the polling approach as an exercise.

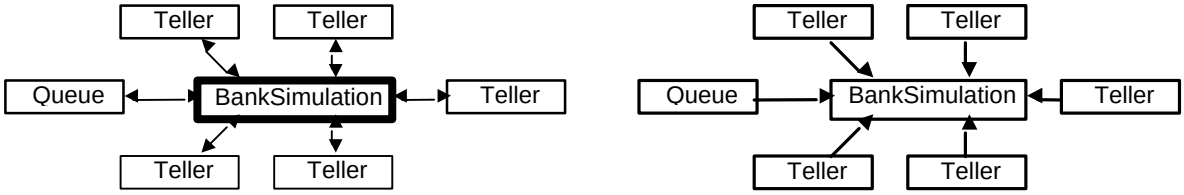


Figure 11.9. Centralized control (left) results in unbalanced distribution of intelligence. Distributed intelligence (right) is generally preferred. The 'intelligence' of an object is indicated by the thickness of the line around it.

Final design

We are now ready to write detailed descriptions of all required classes. We will use the term *reporter* to refer to the BankSimulation object because is responsible for reporting results:

BankSimulation: In charge of user interface - input of parameters, start of simulation, output of results.
Attributes: simulationManager, simulation parameters (number of tellers, minimum and maximum customer processing and arrival times), other aspect variables.
Responsibilities and collaborators:

- Output - implemented by
  - displayEvent: aString. Writes aString to Transcript followed by cr.
- User interface action buttons
  - run - starts simulation
  - close - close application
  - help - display help window

Customer: Represents customer with timing information.
Attributes: processingTime (time to spend at teller station), timeToQueue (time at which Customer entered Queue - for calculation of time spent in bank).
Responsibilities and collaborators:

- Creation - implemented by
  - newWithProcessingTime: anInteger. Creates new Customer. Collaborates with Customer.

CustomerProducer: Generates Customer objects with random values of processing time, keeps information about inter-arrival time and uses it to send Customer to Queue when the time expires and produces another Customer.
Attributes: customer (customer waiting to be released to the queue), releaseTime (when to release current Customer), gapGenerator (random number generator calculating inter-arrival times), processingGenerator (random number generator calculating Customer processing times).
Responsibilities and collaborators:

- Creation - implemented by
  - newWithGapGenerator: aRandomInteger withGapGenerator: gapGenerator withProcessingGenerator: processingGenerator . Collaborates with RandomInteger.
- Updating - implemented by
  - updateTime. Updates time, sends Customer to Queue and creates a new one if appropriate. Collaborates with Queue.

Queue: Knows its customers, can accept new Customer objects, knows how to check whether a Teller is available, knows how to send a Customer to a Teller. Calculates queue statistics and notifies BankSimulation when a customer is added.
Attributes: customers, reporter (reference to BankSimulation), tellers, time (fictitious simulation time)
Responsibilities and collaborators:

- Creation - implemented by
  - numberOfTellers: anInteger reporter: aBankSimulation. Also creates the required number of Teller objects. Collaborates with Teller.
- Processing - implemented by
  - updateTime. Checks whether it has Customer; if so, checks if there is an available Teller; if so, sends Customer to it. Repeated until Queue is empty or no more tellers available. Collaborates with RandomInteger, Teller.
  - addCustomer: aCustomer. Add Customer at end of queue and report it to BankSimulator. Collaborates with BankSimulator.

RandomInteger: Generates random integer numbers within prescribed range.
Attributes: randomGenerator (reference to Random), lowerLimit, upperLimit
Responsibilities and collaborators:

- Creation - implemented by

    - lowerLimit: anInteger upperLimit: anInteger.
- Accessing - implemented by
    - next. Returns random integer.

SimulationManager: Starts, runs, and ends simulation. Collects results at the end and notifies BankSimulation. Issues timing information to CustomerProducer, Queue, and Teller objects.
Attributes: customer, lengthOfSimulation, producer, queue, reporter, totalCustomers (total number of customers sent to Queue), totalTimeInBank (sum of times spent in bank by all customers), simulation parameters.
Responsibilities and collaborators:
- Creation - implemented by
    - tellers: anInteger reporter: aBankSimulator simulationLength: anInteger, minProcessing: anInteger maxProcessing: anInteger minArrival: anInteger maxArrival: anInteger. Collaborates with Queue.
- Simulation - implemented by
    - run. Starts time, creates CustomerProducer, creates Queue, creates one object and puts it in Queue, asks CustomerProducer to produce another Customer and hold it until release. Repeats time update notifying Queue and CustomerProducer, moving Customer to queue when released by CustomerProducer. Tells reporter about number of processed customers and average time in bank. Collaborates with Queue, CustomerProducer.

Teller: Knows how to accept and process a customer, knows whether it has a Customer. Notifies BankSimulation when a Customer arrives or is released.
Attributes: customer, customerReleaseTime, reporter, time, number (number of teller used in reporting).
Responsibilities and collaborators:
- Creation - implemented by
    - number: anInteger reporter: aBankSimulation.
- Updating - implemented by
    - addCustomer: aCustomer. Adds Customer and calculates release time. Collaborates with Customer.
    - updateTime. Checks whether to release Customer, reports when Customer released. Collaborates with BankSimulator.

Implementation

The implementation is simple and we will limit ourselves to a few methods. The rest is left as an exercise.

Output of customer transfers to the Transcript

A typical simulation will run for many units of time and generate a lot of output to the Transcript if the *Show steps* check box is on. This can be relatively time consuming because individual show: messages consume long time. In cases like these, it is better to accumulate the output in the Transcript object (a TextCollector) using nextPut: aCharacter and nextPutAll: aString messages, and flush all accumulated output at the end. We will illustrate this technique on the example of customer dispatch to tellers by Queue.

When Queue contains customers and there is an available teller, it sends a customer using the following message. As a part of this message it informs reporter, the active BankSimulation object:

```
customer: aCustomer
"A customer has arrived.
        customer := aCustomer.
        customerReleaseTime := customer processingTime + time.
        reporter displayEvent: 'Time: ' , time printString , '. Teller ' , number printString , '
                        new customer with processing time ' , customer processingTime printString
```

steps: true

The message to reporter uses method displayEvent: aString steps: aBoolean as general purpose event notification message. The method is defined in BankSimulation as follows:

```
displayEvent: aString steps: aBoolean
"Send aString to Transcript unless this is one of the detailed steps and display of steps is not  desired."
        (showSteps value or: [aBoolean not])
                ifTrue: [Transcript nextPut: Character cr; nextPutAll: aString]
```

Since the message to Transcript is not show:, the event notification string is not displayed at this point but only stored for future input in Transcript. At the end of simulation, the SimulationManager sends message flush to BankSimulation which then sends the flush message to the Transcript. This causes all accumulated output to be output in Transcript in one chunk, a much more efficient operation than a sequence of show: messages.

Main lessons learned:

- A queue is a first-in first-out structure (FIFO). Elements are added at one end (tail) and removed from the other end (head).
- Queue behavior is subsumed by OrderedCollection and there is no need for class Queue.

Exercises

1. Implement class Queue with only the essential stack behavior.
2. Implement bank simulation as described.
3. Reimplement bank simulation using polling instead of event-driven design.
4. Modify bank simulation by adding output to file.
5. Repeat our simulation using Poisson's distribution for all random events in the simulation. This distribution is based on the assumption that there is no limit on inter-arrival time.
6. Implement cash register simulation where customers line up in front of individual registers.
7. Explain why queues are also called First-In-First-Out (or FIFO) objects while stacks are called First-In-Last-Out objects.

11.5 Text filter - a new implementation

In this section, we will show how the text filter from the previous chapter can be implemented in a more sophisticated way using a queue.

Probelm: The behavior of our previous solution was unsatisfactory and we will thus try to find a better specification and a better solution. Let's try this: Class TextFilter takes an original String object and replaces occurrences of match substrings with corresponding replacement substrings, compressing the original as much as possible.

Scenario 1: Filtering 'abcdeab' using match/replacement pairs pair1 = 'ab'->'xx', pair2 = 'eab'->'yy'
Assume original string = 'abcdeab' and match/replacement pairs pair1 = 'ab'->'xx', pair2 = 'eab'->'yy'. This is the scenario that failed in our previous implementation. We will now show how the scenario is executed assuming that we keep track of the current position in the original and in each match string via a position counter, and that the resulting string is held in result.

1. Initialize result to empty string; initialize position counters to 1.
2. Compare position 1 in pair1 ($a) with first character of string ; match found, increment pair1 pointer.
3. Compare position 1 in pair2 ($e) with first character of string key; no match.

4. No complete match, copy character from string to result, increment position in string and result.
5. Compare position 2 in pair1 ($b) with first character of string - match.
6. Compare position 1 in pair2 ($a) with first character of string key; no match.
7. We have a complete match. Select the longest replacement possible, changing result to 'xx'.
8. Continue in this way until the last character in string. At this point, there are two matches and two possible replacements. The longer one is selected giving result = 'xxcdyy' as expected.

This scenario works as expected. But wait - the behavior in the following scenario does not quite meet our expectations.

Scenario 2: Filtering string = 'abcd' with match/replacement pairs pair1 = 'bc'->'xx', pair2 = 'abcd'->'yy'
Assume string = 'abcd' and match/replacement pairs pair1 = 'bc'->'xx', pair2 = 'abcd'->'yy'. We would assume to obtain the most compressed result = 'yy' but our position counter-based implementation will produce 'axxcd' (left as an exercise).

The reason for the unsatisfactory behavior is that 'bc' is finished matching before 'abcd', and 'abcd' thus never gets a chance to complete matching and be replaced. We thus need a new approach and the best place to start is to analyze why our current approach fails. Our approach does not work because as soon as a substring is matched, it is immediately replaced. However, we cannot make a replacement unless we know that there is no matching in progress that started earlier than the matching that just succeeded, and that is not yet finished.

To deal with this situation, we will use a *queue* of all matching processes in progress such that the matching or matchings that started first are at the head of the queue. The idea is as follows: Match the original character by character. For each character, put all (match string -> replacement string) associations that succeed in matching their first character into a collection and add the collection to the end of the queue. (The queue thus consists of collection objects.) When an association fails to match, remove it from the queue. When a complete match occurs, check whether the matching association is in the collection at the head of the queue. If it is, there is no earlier matching in progress and we can thus safely make the replacement in result. At this point, we will empty the whole queue and start a new matching sequence. If the matching association is not in the collection at the head of the queue, mark it as ready for replacement and proceed; don't do any replacement yet - wait until the association reaches the beginning of the queue. When it does, use it to make the appropriate replacement.

Before we formulate the algorithm more carefully, we will take a look at the objects needed to support it. In addition to streams and position counters to keep track of the original string and the result, we also need

- A queue whose elements are collections of partially matched associations. We will call it MatchesInWaiting and implement it as a class variable.
- A collection of all associations that are fully matched and ready for replacement. This will be a class variable called ReadyForReplacement.

With this background, we can fully describe the matching procedure as follows:

1. Create a ReadStream over the original string and a WriteStream over the string being constructed.

Create an empty MatchesInWaiting queue.

Create a MatchDictionary and initialize it to *match string -> two-element array* associations; each association has a match string for key. The first element of its value array is the replacement string, the second element of the array is the position counter initialized to 0.

2. Repeat for each position of the input stream beginning from the start:
   a. Add a new collection to the end of the MatchesInWaiting queue.
   b. For each element of the dictionary do:

      Increment position counter of this dictionary item.

Compare original character and match character.
If no match, reset current position counter of dictionary item to 0. If this item is in MatchesInWaiting, remove it.
If match, check if this is match on first character of match string.
If so,
add this association to the collection at the end of the MatchesInWaiting queue.
Check if this is the last character to match.
If so (match succeeded), check if ReadyReplacement is empty.
If so, store this association in ReadyReplacement.
If this is not the last character (match incomplete), increment position counter of this association.

c. If ReadyReplacement contains an association and if this association is at the head of the queue, use the association to make a replacement in OutputStream.
Empty MatchesInWaiting queue, reset ReadyReplacement and ReplacementPosition to nil.

We leave it to you to check whether this algorithm works, correct it if it does not, and implement it.

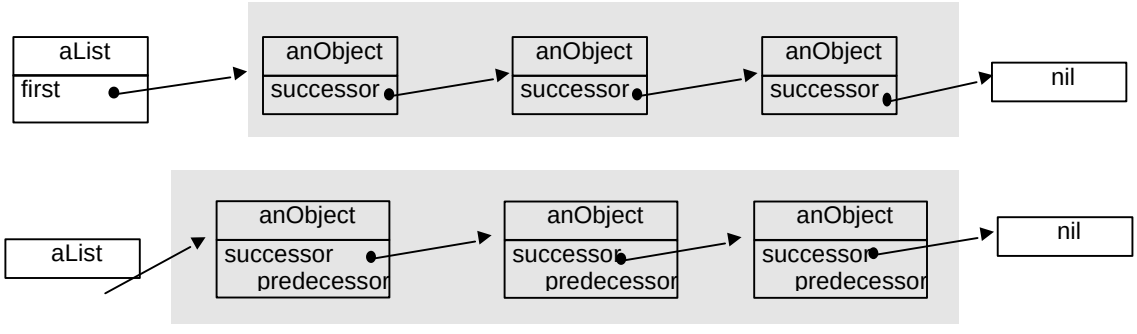
Exercises

1. Complete the design of the text filter and implement and test it.
2. Our algorithm will work if all the replacement strings are equally long. But what if they are not?
3. Extend the filter to allow replacement calculated by blocks.
4. Our various formulations of the string replacement problem were not incorrect but merely different. Are there any other possible formulations of the string replacement problem? If so, outline the appropriate solutions.
5. Since there are several possible formulations of text filtering that each require a different solution, the problem seems amenable to an abstract class with concrete subclasses implementing different specifications. Design such a hierarchy and comment on the advantages of this approach - if any.

11.5 Linked Lists

None of the sequenceable collections covered so far are specifically designed for insertion of new elements at arbitrary locations. Arrays allow replacement but not insertion, and ordered collections allow insertion but the internal operation is complex and inefficient.

A linked list is a collection of objects linked to one another like a string of pearls. As a minimum, a linked list knows about its first element, and each element knows about its successor. A *doubly* linked list is an extension of a singly linked list whose elements also know about their predecessor (Figure 11.10). Both provide a way insert a new element at any location. Another advantage of linked lists is that they occupy only the amount of space required to hold their elements whereas ordered collections may occupy more space because they are normally only partially filled. However, elements of linked lists (their *nodes*) must be packaged in more complex objects.



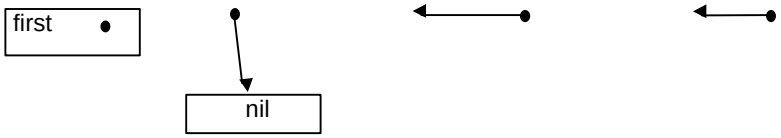


Figure 11.10. Linked list (top), doubly linked list (bottom). List elements are shaded.

As our illustration shows, the implementation of linked lists requires two kinds of objects - list elements and a linked list object itself. A minimal linked list must know at least about its first node.
In the Smalltalk library, linked lists are implemented by class LinkedList with the following comment:

```
The class LinkedList implements ordered collections using a chain of elements. Each element of a
LinkedList must be an instance of class Link or of one of its subclasses.
A new instance of LinkedList can be initialized using
        LinkedList with: Link new
Instance Variables:
        firstLink       <Link>
        lastLink        <Link>
```

Class LinkedList is a subclass of SequenceableCollection and inherits its protocol including creation, enumeration, and access by index (normally unused), and redefines many of them.

Class Link whose instances are nodes of LinkedList is very primitive and implements only linking with no provision to hold a value. Its comment is

```
Class Link represents a simple record of a pointer to another Link.
Instance Variables:
        nextLink        <Link> a pointer referencing the next Link in the chain
```

Link protocol includes nextLink (returns the next node in a linked list), nextLink: aLink (sets nextLink to specified Link object), and the creation message nextLink: that initializes the successor of the receiver. Being so primitive, Link is only used as a superclass of classes that attach a value to the link object, thus allowing the creation of LinkedList objects with non-trivial elements as in Figure 11.10.
Linked lists are simple but useful objects and we will illustrate their use on the following example:

Example. Reimplement sorted collections using linked lists

Problem: Use a linked list to implement class SortedLinkedList with the functionality of SortedCollection. Restrict the definition to creation and adding.
Solution: Being an extension of LinkedList, SortedLinkedList will be a subclass of LinkedList with a new instance variable called sortBlock. The nature of the sort block will be the same as that of SortedCollection and there will also be a default sort block stored in a class variable DefaultSortBlock. Its value will be [:x :y | x <= y] so that a SortedLinkedList with the default block will have its elements sorted in the order of ascending magnitude of its elements. The new message will create a SortedLinkedList with the default sort block, and an instance with a different sort block can be created by the class method sortBlock: aSortBlock.
The definition of the class is

```
LinkedList subclass: #SortedLinkedList
        instanceVariableNames: 'sortBlock '
        classVariableNames: 'DefaultSortBlock '
        poolDictionaries: ''
        category: 'Book'
```

The value of the default sort block will be assigned by the same *class* initialization method as in SortedCollection:

```
initialize
"Create a default sort block"
        DefaultSortBlock := [:x :y | x <= y]
```

Don't forget to execute the initialization method with

```
SortedLinkedList initialize
```

before using SortedLinkedList for the first time. We suggest using *inspect* to see the effect.
Next, the creation method. The method creates a new instance and then initializes the sort block by

```
new
        ^super new initialize
```

where the *instance* method initialize is again the same as in SortedCollection:

```
initialize
"Set the initial value of the receiver's sorting algorithm to a default."
        sortBlock := DefaultSortBlock
```

When we test the code developed so far with

```
SortedLinkedList new
```

we get the expected result – try it with *inspect*.

Before we can implement adding, we must be able to create node objects. Since the nodes in our list have a value, they cannot be simple Link objects. We will thus create a new class called LinkNode with instance variable value, extending Link with creation method nodeValue: anObject, and accessing methods value and value: anObject. We will leave these methods as an exercise and limit ourselves to comparison by <= which is required by the sort block:

```
<= aNode
"Compare the value of the receiver and the value of the argument."
        ^self value <= aNode value
```

We suggest that you test that LinkNode methods work.

Now that we have a class to implement nodes, we can return to SortedLinkedList. We will limit ourselves to method add: anObject for adding a new object to a linked list and leave other functionality as an exercise. The add: message will be based on careful consideration of all possible situations expressed in the following algorithm:

1. If the list is empty, add the new node using the inherited add: message and exit.
2. If the list is not empty, search it from the beginning looking for a node that returns false when compared with the new node using the sort block.
    a. If such a node is found, insert the new node before this element and exit.

b. If such a node is not found, add the new node at the end of the list

We will now examine Step 2 in more detail. Each evaluation of the sort block has two possible outcomes: If the block returns true, we must continue the search and proceed to the next node. If the block returns false, the current node is either at the beginning of the linked list or inside it (Figure 11.11). If the current node is the first node in the list, the new node must be inserted in front of it using the inherited addFirst: message. If the current node is not the first node, the new node must become the nextLink of the previous node, and its own nextLink must point to the 'current' node. This means that the method must keep track of the previous node.

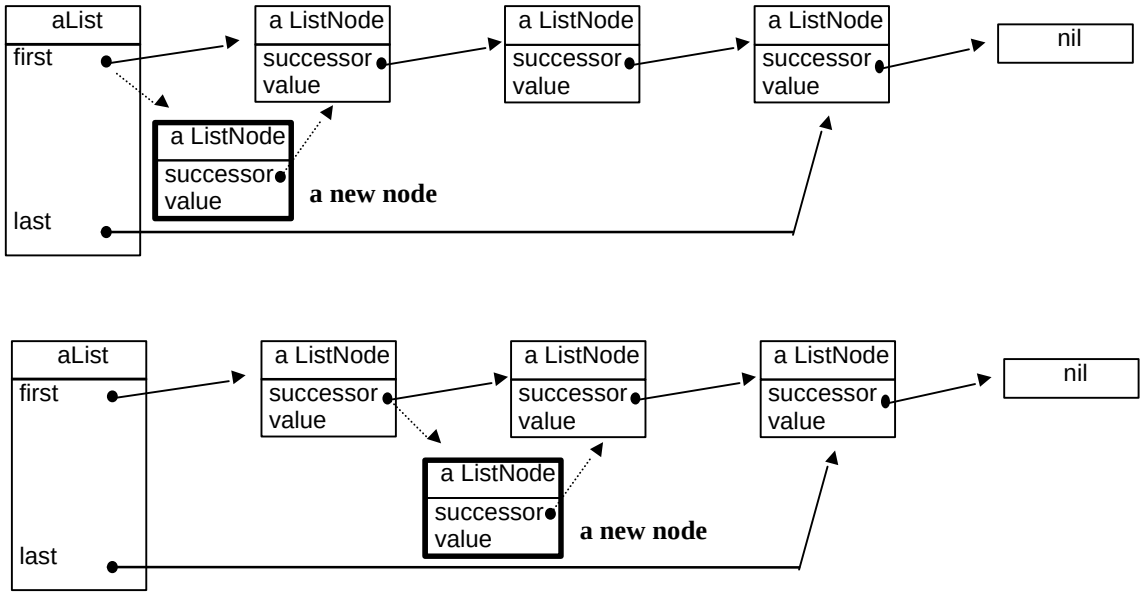


Figure 11.11. When the list is not empty, the new node will be inserted either at the start of the list (top), or into the list (bottom), or behind the last element (not shown).

With this background, the definition of add: can be written as follows:

```
add: anObject
"Add anObject to the list at position determined by sortBlock. Return anObject."
| current newNode previous |
        "Create a node link with value anObject."
        newNode := LinkNode nodeValue: anObject.
        "If the list is empty, simply put the node at the start of the list."
        self isEmpty ifTrue: [super add: newNode. ^anObject].
        "List is not empty, traverse it from the start."
        current := self first.
        [current isNil]       "End of list?"
                whileFalse: [
                        (sortBlock value: current value: newNode) "Is new node behind current node?"
                                ifTrue: "Sort block succeeded, go to next list node."
                                        [previous := current.
                                        current := current nextLink]
                                ifFalse: "Sort block failed, distinguish two possibilities."
                                        [current == firstLink
                                                ifTrue: "Insert new node at the start of the list."
                                                        [newNode nextLink: firstLink.
                                                        self addFirst: newNode]
```

```
                                                    ifFalse: "Insert between previous and current."
                                                            [previous nextLink: newNode.
                                                            newNode nextLink: current].
                                            ^anObject]].
        "Node does not fit inside he linked list, add it at the end."
        self addLast: newNode.
        previous nextLink: newNode.
        ^anObject
```

We are now ready for testing. We cannot test every possible situation but we must try to test every *logical* possibility. We will test add: in the following situations:

1. List is empty.
2. List is not empty and the new node becomes the first node in the list.
3. List is not empty and the new node is inserted between two existing nodes.
4. List is not empty and the new node is added at the end of the list

An example of a test program that checks all these alternatives is

```
| sll |
Transcript clear.
sll := SortedLinkedList new.
Transcript show: sll printString; cr.
sll add: 3.
Transcript show: sll printString; cr.
sll add: 33.
Transcript show: sll printString; cr.
sll add: 13.
Transcript show: sll printString; cr.
sll add:  0.
Transcript show: sll printString
```

Before we can execute this program, we must make sure that printString creates meaningful SortedLinkedList descriptions. Since SortedLinkedList is a subclass of Collection, we reuse the printing protocol in Collection which depends on do:. We must thus define do: as follows:

```
do: aBlock
"Evaluate aBlock with each of the receiver's elements as the argument."
        current := firstLink.                                   “Initialize for traversal.”
        [current isNil]                                         “Iterate up to and including the last node.”
                whileFalse:
                        [aBlock value: current.
                        current := current nextLink].           “Get the next node, if any.”
```

With this method, printing works but does not produce any meaningful information about nodes. We define node printing as

```
printOn: aStream
"Append to the argument, aStream, the elements of the Array enclosed by parentheses."
        aStream nextPutAll: self class name; nextPutAll: ' '; nextPutAll: self value printString
```

and with this, our test program produces the following correct output which confirms that our definitions work as expected:

```
SortedLinkedList ()
SortedLinkedList (LinkNode 3)
SortedLinkedList (LinkNode 3 LinkNode 33)
```

```
SortedLinkedList (LinkNode 3 LinkNode 13 LinkNode 33)
SortedLinkedList (LinkNode 0 LinkNode 3 LinkNode 13 LinkNode 33)
```

Finally, we must test that SortedLinkedList works even with non-default sort blocks and we thus define a new SortedLinkedList creation method called sortBlock: and modify our test program to create a new list with the same elements sorted in *descending* order:

```
| sll |
Transcript clear.
sll := SortedLinkedList sortBlock: [:x :y | x > y].
Transcript show: sll printString; cr.
sll add: 3.
Transcript show: sll printString; cr.
sll add: 33.
Transcript show: sll printString; cr.
sll add: 13.
Transcript show: sll printString; cr.
sll add:  0.
Transcript show: sll printString
```

The test produces

```
SortedLinkedList ()
SortedLinkedList (LinkNode LinkNode 3)
SortedLinkedList (LinkNode LinkNode 33 LinkNode LinkNode 3)
SortedLinkedList (LinkNode LinkNode 33 LinkNode LinkNode 13 LinkNode LinkNode 3)
SortedLinkedList (LinkNode LinkNode 33 LinkNode LinkNode 13 LinkNode LinkNode 3 LinkNode LinkNode
0)
```

in the Transcript and again confirms the correctness of our implementation.

Main lessons learned:

- A linked list is a linear collection of nodes with a pointer to its first element and links between consecutive nodes.
- In addition to forward links, linked nodes can also have backward links. Such lists are called doubly linked lists.
- A linked list allows insertion of elements at any point in the list.
- Smalltalk library contains a pair of general classes called LinkedList and Link implementing the basic linked list behavior. For concrete use, these classes must be subclassed.

Exercises

1. Complete the implementation of the example from this section and add node removal.
2. Find how an element can be inserted into an OrderedCollection.
3. Test which implementation of sorted collection is faster using the Speed Profiler included in the Advanced Tools of VisualWorks. Justify the result.
4. Compare the memory requirements of the two implementations of sorted collection using the Allocation Profiler included in the Advanced Tools of VisualWorks.
5. Add node removal to sorted linked list.
6. Add method asArray returning an Array containing the values of all elements of the list in the order in which they are stored.
7. Implement a doubly linked list.

11.6 Trees

A tree is a two-dimensional collection of objects called nodes. Three examples are a class hierarchy tree, a family tree, and a tree of student records as in Figure 11.12. The nodes of a tree can be classified as follows:

- The node at the top of the tree is called the *root.* This is the node through which the tree can be accessed. The node with ID 27 in our example is the root of the tree. A tree has exactly one root.
- A node that is not a root and has at least one child is an *internal* node. The records with IDs 13 and 32 in our example are internal nodes. Every internal node has exactly one parent node but it may have any number of children, unless the definition of the tree specifies otherwise.
- A node that does not have any children is called a *leaf.* The node with ID 11 in our example is a leaf node. Like an internal node, a leaf has exactly one parent node.

An interesting and useful property of trees is that any node in the tree can be treated as the root of a new tree consisting of all the underlying nodes. Such a tree is called a *subtree* of the original tree. As an example, the node with ID 32 in Figure 11.12 can be used as the root of a tree with children 29 and 45. Leaves, such as 29, can be thought of as degenerate trees that consist of a root and nothing else.

Trees are very important in computer applications and a variety of trees have been devised to provide the best possible performance for different uses. The differences between different kinds of trees are in the number of children that a node may have[4], and the way in which the tree is managed (how nodes are added and deleted). The subject of trees is not trivial and we will restrict our presentation to the example of a simple binary tree. In a binary tree, each node may have at most two children as in Figure 11.12.

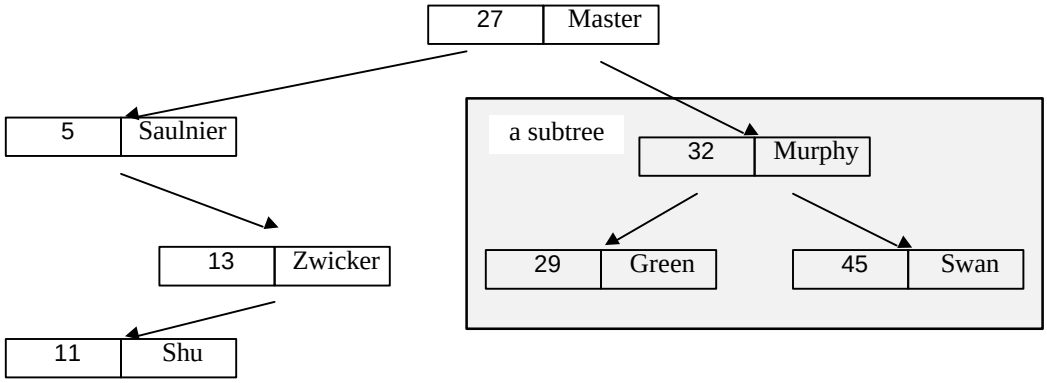


Figure 11.12. Binary tree of student records based on comparison of student IDs.

Problem: Implement a binary tree whose nodes are arranged on the basis of comparison of node values. The tree must be capable of adding a new object, deleting an existing node, and finding whether an object with a given value is in the tree. Provide enumeration and printing as well.
Solution: Our implementation will use two kinds of objects - nodes (class Node), and the tree itself (class BinaryTree). The BinaryTree object must know about its root and how to perform comparison (comparison block), the Node object must know its value and its children; knowing the parent may also be useful. We will start with Node because we cannot create BinaryTree without nodes. The *comment* of Node is as follows:

Class Node implements nodes for class BinaryTree. Each node consists of

[4] Each node with the exception of the root always has exactly one parent.

value	<Object>	- value of node, must understand the tree's comparison block
leftChild	<Node|nil>	- root of left sub-tree of this node, a Node or nil
rightChild	<Node|nil>	- root of right sub-tree of this node, a Node or nil
parent	<Node|nil>	- parent of this node, a Node or nil

The superclass of Node will be Object and we will start its implementation with a class method to create a Node with its value. The method is styled in the usual way:

```
newWithObject: anObject
        ^self new  value: anObject
```

and instance method value: sets value to anObject. We leave the remaining instance variables uninitialized which is as it should be because theyr will be assigned as new objects are added to the tree. The methods to add children and the parent are equally simple but we must remember that when a node is added as a child, the receiver node must become the child's parent as in

```
rightChild: aNode
        rightChild := aNode.
        aNode parent: self
```

After defining all accessing methods, we can now test our design by executing the following code with *inspect*:

```
|parentNode node nodeOnLeft nodeOnRight|
parentNode:= Node newWithObject: 'parent node'.
node := Node newWithObject: 'this node'.
nodeOnLeft := Node newWithObject: 'left sub-node'.
nodeOnRight := Node newWithObject: 'right sub-node'.
node parent: parentNode.
node leftChild: nodeOnLeft.
node rightChild: nodeOnRight.
node
```

We find that everything is OK.

Now that we have a working Node, we can start working on class BinaryTree. Each instance of BinaryTree needs to know the tree's root node and its comparison block which is used to decide whether a new node should be inserted in the left subtree or the right subtree. We will call these two instance variables root and isLeftBlock and comment the class as follows:

```
Class BinaryTree implements a binary tree. Nodes are added as leaves on the basis of a comparison block
applied to the value of the new object and values of existing nodes.

Instance variables

root            <Node|nil>          - root node of the tree, a Node or nil.
isLeftBlock     <BlockClosure>      - a two-argument block used to determine whether a new node should
                                      be added to the left of an existing node. The first argument of the block is the new node,
                                      the second argument is an existing node to which the new node is being compared.
                                      When the block returns true, the new node is attached as the left subnode; on false,
                                      the new node is attached on the right.
```

The superclass of BinaryTree is Object. The *creation* method will specify the root of the tree and the comparison block. It will be used as in

```
BinaryTree withRoot: 13 withBlock: [:new :old| new < old]
```

The creation method is as follows:

```
withRoot: anObject withBlock: comparisonBlock
        ^self new root: (Node newWithObject: anObject) block: comparisonBlock
```

where instance method root:block: initializes root and isLeftBlock. Implement these two methods and test everything by executing the above expression with *inspect*.

Now that we can create a tree with a root and a comparison block, we need a method to add an object to an existing tree. To add a new object, we start from the root and proceed downwards using the following principle: If the value of the new object satisfies the comparison block with the value of the current node, proceed to the left child, if any; otherwise proceed right. Upon reaching a node whose desired child is nil, further traversal is impossible; add the new object as a new leaf on the left or on the right on the basis of comparison of comparison.

Since adding works the same way for the root and for sub-nodes, the add: method can take any node as its argument and treat it as the root of a tree and re-send itself with this node as the root. A more accurate description of the algorithm is as follows:

1. If the tree is empty, make the new object the root of the tree and exit.
2. If the tree is not empty, proceed as follows:
   a. If the root and the new object satisfy the sort block, send add: to its left subtree.
   b. If evaluation of the sort block returns false, send add: to the right subtree.

You can check that the tree in Figure 11.12 is the result of this algorithm with nodes added in the following order of IDs: 27, 13, 8, 32, 29, 11, 45. Our description is easily translated into the following code:

```
add: newNode toNode: existingNode
"Attach newNode as leaf of the appropriate subtree of existingNode."
        existingNode isNil ifTrue: [root := newNode. ^newNode].
        (isLeftBlock value: newNode value value: existingNode value)
                ifTrue: [  "Add newNode to left sub-tree."
                        existingNode leftChild isNil
                                ifTrue: "Attach aNode as leftChild."
                                        [existingNode addLeftChild: newNode]
                                ifFalse: "Apply this method recursively."
                                        [self add: newNode toNode: existingNode leftChild]]
                ifFalse:   "Add newNode to right sub-tree."
                        [existingNode rightChild isNil
                                ifTrue: " Attach aNode as rightChild."
                                        [existingNode addRightChild: newNode]
                                ifFalse: "Apply this method recursively."
                                        [self add: newNode toNode: existingNode rightChild]].
                ^newNode
```

Note that we used value: value: to evaluate the comparison block because the block has two arguments.

This method can be used to add a node once we have a node to start from. First, however, we must get into the tree and get the starting node – the root. To do this, we will use the following method:

```
add: anObject
"Add node with anObject as a new leaf of the tree."
        self add: (Node newWithObject: anObject) toNode: root
```

We can now test our solution with the following program which should create the tree in Figure 11.13:

```
|tree |
tree := BinaryTree withRoot: 15 withBlock: [:new :old| new value < old value].
tree add: 8;
        add: 21;
        add: 14;
        add: 36;
        add: 17;
        add: 49;
        add: 32;
        yourself
```

Execute the program with *inspect* and check that the resulting BinaryTree is correct. Note that once you open the tree inspector, you can use it to traverse the whole tree by selecting first the root and then inspecting its children, proceeding recursively in this way as far as you wish.

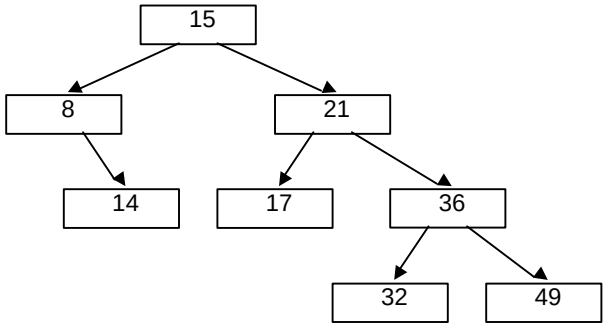


Figure 11.13. Binary tree created with the program in the text.

Unfortunately, our add: method ignores the possibility that the root is nil which will happen if we deleted all nodes. This can be corrected as follows:

```
add: anObject
"Attach node with anObject as a new leaf of the tree."
        | newNode |
        newNode := Node newWithObject: anObject.
        root isNil
                ifTrue: [root := newNode]
                ifFalse: [self add: newNode toNode: root]
```

The next protocol is finding. The find: anObject method will return the node whose value is anObject, or nil if such a node is not in the tree. We leave it as an exercise and suggest that you implement it via two methods as we did for adding.

To *delete* a node with a specified value, we must find the node, remove it from the tree, and rearrange the tree so that new node ordering still satisfies the sort block. The new tree may not look like the original tree but the relation between a node and its children must satisfy the comparison block.

The easiest (but not the most efficient) way to implement this specification is to remove the node and then add all its children and the remaining nodes to its parent using the *add* operation. By definition, this will preserve proper ordering. A detailed description of the deletion algorithm is as follows:

1. Find node node whose value is anObject. If there is no such node, execute an exception block and exit.
2. If node is root, then
    a. If its left child is nil then
        i) Replace root with left child.
        ii) Replace new root's parent with nil to cut the link to the removed root.
    b. Else (the left child is not nil)

i) Add root's right child to the root's left child.
ii) Make left child the new root.
iii) Replace new root's parent with nil to cut the link to the removed root.

3. If node is not root then
   a. Remove connection from node's parent to node.
   b. Add the left child of node to the parent – this adds all left subnodes because it leaves the child's subtrees unchanged.
   c. Add the right child of node to the parent– this adds all right subnodes because it leaves the child's subtrees unchanged.

This algorithm can be used to define delete as follows:

```
remove: anObject ifAbsent: aBlock
"Delete anObject from the tree and return anObject; evaluate aBlock if not present."
| parent theNode |
        theNode := self find: anObject.
        theNode isNil ifTrue: [^aBlock value].
        parent := theNode parent.
        parent isNil
                ifTrue: ["theNode must be the root."
                        root leftChild isNil
                                ifTrue: [root := root rightChild]
                                ifFalse: [self add: root rightChild toNode: root leftChild.
                                        root := root leftChild].
                        root parent: nil]
                ifFalse: [parent leftChild = theNode
                                ifTrue: [self removeLeftChild]
                                ifFalse: [self removeRightChild].
                        self add: theNode leftChild toNode: parent.
                        self add: theNode rightChild toNode: parent].
        ^anObject
```

where removeLeftChild and removeRightChild simply replace the child with nil. Finally, *enumeration* can be implemented by a combination of two methods:

```
do: aBlock
"Evaluate aBlock for each node of the tree rooted at the root."
        self do: aBlock startingFrom: root
```

and

```
do: aBlock startingFrom: aNode
"Evaluate aBlock for each node of the tree rooted at aNode."
        aNode isNil
                ifFalse: [aBlock value: aNode.
                        self do: aBlock startingFrom: aNode leftChild.
                        self do: aBlock startingFrom: aNode rightChild]
```

This very concise method is a typical example of the advantage of using recursion to implement a non-trivial operation on a complicated collection. The method can be tested, for example, by

```
|tree |
tree := BinaryTree withRoot: 15 withBlock: [:new :old| new value < old value].
        tree add: 8;
        add: 21;
        add: 14;
        add: 36;
```

```
        add: 17;
        add: 49;
        add: 32;
        do: [:node | Transcript show: node value printString; cr]
```

which produces

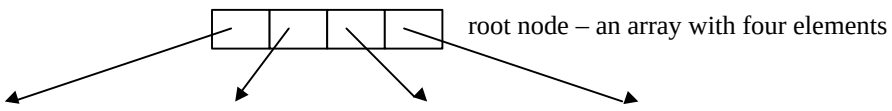
```
8
14
21
17
36
32
49
```

Main lessons learned:

- A tree is a branching structure of nodes and their children emanating from one node called the root.
- Nodes at the bottom of the tree (nodes with no children) are called leaves.
- Every node in a tree except the root has exactly one parent node. The root does not have a parent.
- Specialized trees may restrict the number of children of a node or the structure of the tree.

Exercises

1. List two different number sequences that will produce the tree in Figure 11.13.
2. Complete and test BinaryTree.
3. How could we improve our design of BinaryTree? One weakness of our approach is that when we traverse the tree, we check each node to determine whether it is empty or not because an empty node requires different treatment. This overhead is unnecessary if we use polymorphism an implement BinaryTree as an abstract class with two subclasses – one for an empty binary tree and one for a non-empty tree. Implement this idea (borrowed from [Pugh, Lalonde]) and compare the efficiency of the two approaches.
4. Add the following methods to the implementation of binary tee:
   a. A method to merge two trees together.
   b. A method to check whether a tree contains another tree as its subtree.
   c. A method to copy nodes from one tree to another tree.
5. Assume that in addition to binary trees, we also want to implement trees with up to three children, and trees with up to four children. Both trees use some further unspecified construction rules. Is it desirable to define an abstract superclass? If so describe its properties in detail. What other classes must be defined?
6. Explain in what order the do:startingFrom: method traverses the nodes of the tree. Can you describe other traversals? Implement them.
7. Add a method to find the depth of the tree - the maximum number of nodes encountered during the traversal from the root to a leaf.
8. Consider n = 255 objects sorted according to their value. What is the maximum number of steps required to find a given object if the objects are stored in a sorted collection? What is the answer if the objects are stored in a balanced tree whose branches have, as much as possible, equal depth? What is the answer for a general value of n?
9. A two-dimensional array can be implemented with a tree as in Figure 11.14. Describe how this principle could be used to implement an n-dimensional array.



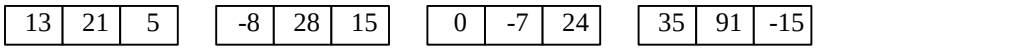


Figure 11.14. Representing a two-dimensional 4x3 matrix by a two-level tree. Leaves represent rows.

11.7 Use of trees in compilation

One of the most important uses of trees is in compilation. When you compile a method with *accept*, the following sequence of events occurs:

1. A message is sent to the Compiler object with the source code as one of its arguments.
2. The compiler sends a message to the Parser object to find the structure of the method.
3. The parser sends a message to the *lexical analyzer* which converts the source code string into *tokens* - symbols representing higher level language entities such as literal numbers or strings, and unary or binary selectors.
4. The parser uses the tokens and the definition of Smalltalk syntax to construct the *parse tree* representing the source code.
5. The *code generator*, reads the parse tree, and produces the translation - an instance of CompiledMethod.

The process is, of course, interrupted if the code contains illegal source code. We will now illustrate the operation on an example. Assume that you entered the following method in class Test using the Browser and clicked *accept*.

```
test
"A test to illustrate compilation."
3 + 3.
5 factorial
```

If you trace the execution of the ensuing compilation by inserting a breakpoint in front of the statement in Step 1 (below), you will find that the steps listed above take the following form:

1. The compiler is invoked with the message

```
Compiler new compile: 'test 3 + 3. 5 factorial'
        in: Test
        notifying: nil
        ifFail: []
```

2. The compiler invokes the parser with the message

```
Parser parse: aStream
    class: class
    noPattern: noPattern
    context: context
    notifying: handler
    builder: ProgramNodeBuilder new
    saveComments: mapFlag
    ifFail: [^failBlock value]
```

    where aStream contains the source code of the method, and the remaining arguments specify who will build the parse tree and provide additional information.

3. The parser asks the scanner to convert the source code into tokens. In our case, the scanner produces the following sequence of token-type/token-value pairs:

```
#word          'test'
#number   3
#binary        +
#number   3
#period   .
#number   5
#word          'factorial'
```

   In constructing this sequence, the scanner skipped all characters that don't have any execution effect such as spaces, tabs, and carriage returns. (In actual operation, the scanner transfers control to the parser after each token-type/token-value pair but the principle is the same as if all tokens were generated first which is what we will assume for ease of presentation.)

4. The parser converts the tokens into a parse tree. In terms of Smalltalk objects, the parse tree is one big nested object, the MethodNode object shown in Figures 11.15.

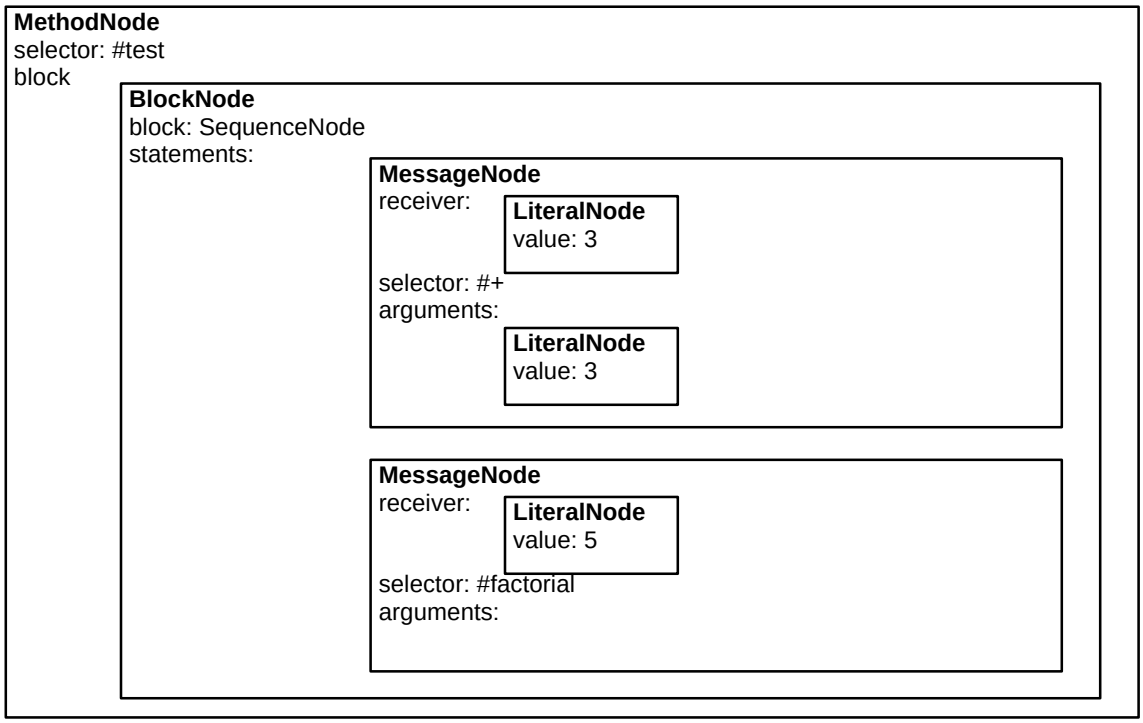


Figure 11.15. Nested MethodNode object produced for the example method in the text. Only the essential attributes are shown.

The methodNode object can be represented as the *parse tree* in Figure 11.16. Note that each node may be a different kind of object with different instance variables, that there is no prescribed number of children, and that the structure of the tree is determined by the structure of the source code and the syntax rules of Smalltalk rather than comparison of node values as in our previous example.

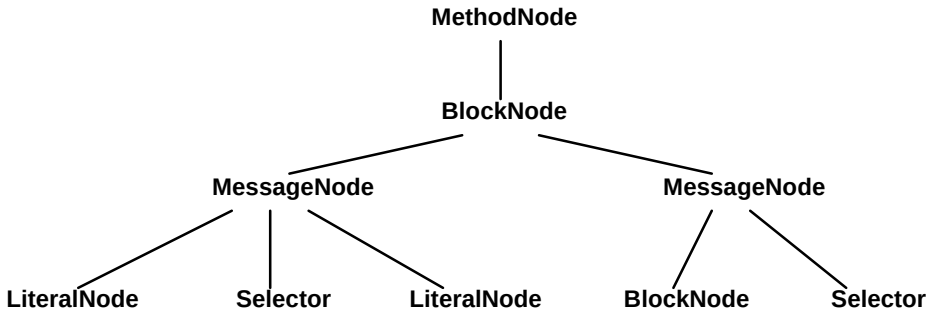


Figure 11.16. Parse tree corresponding to Figure 11.15.

5. Finally, the code generator uses the parse tree to calculate the following CompiledMethod:

normal CompiledMethod numArgs=0 numTemps=0 frameSize=12

literals: (#factorial)

```
1 <D8 03> push 3
3 <D8 03> push 3
5 <DF 00> no-check send +
7 <66> pop
8 <D8 05> push 5
10 <CC 00> no-check send factorial
12 <45> pop; push self
13 <65> return
```

In conclusion, let's note that both Parser and Scanner are in the library and you can subclasses them to create recognizer for objects defined by their syntax or to parse programs written in another programming language.

Main lessons learned:

- Compilation consists of lexical analysis (scanning), parsing, and code generation.
- Lexical analysis extracts program tokens from the source code, parsing recognizes the grammatical structure of the program, code generation produces a CompiledMethod with executable bytecodes.
- Parsing explicitly or implicitly creates a parse tree.

Exercises

1. Create a simple method, trace its execution, and describe the process.

11.8 Graphs

A graph is a mathematical abstraction of concepts such as the road map (Figure 11.17), the layout of house plumbing, the telephone network, the structure of course prerequisites, or the world-wide web. A graph can be completely described by listing its *nodes* (usually called *vertices* - the plural of *vertex*) and the *edges* (*arcs*) connecting them. When the edges are labeled with numbers representing, for example, distances or driving times between cities on a road map, the graph is called a *weighted graph* and edge weights must be included in the description of the graph. As an example, our road map can be described as follows:

```
V = {Chicago, New York, Toronto, Montreal, Vancouver, San Francisco, Denver, Dallas}   "Set of vertices."
E = {(VT,60), (TM,6), (VS,21), (VD,33), (TD,31), (TC,11), etc.}   "Set of weighted edges."
```

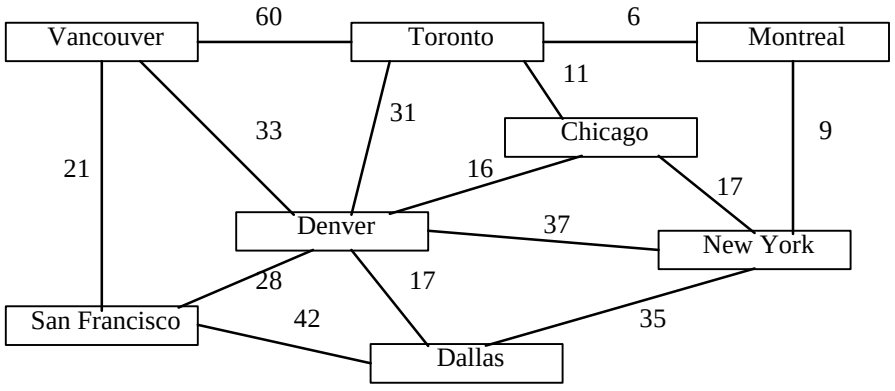


Figure 11.17. Road map with driving times along selected highways.

The graph in Figure 11.14 is an *undirected graph* because its edges don't restrict the direction in which they must be traversed. In other words, the roads between the cities are not one-way roads. A map showing airline connections (Figure 11.18), on the other hand, may have restrictions because the airline may not provide flights in both directions between all cities. Such a graph is called a *directed graph*.

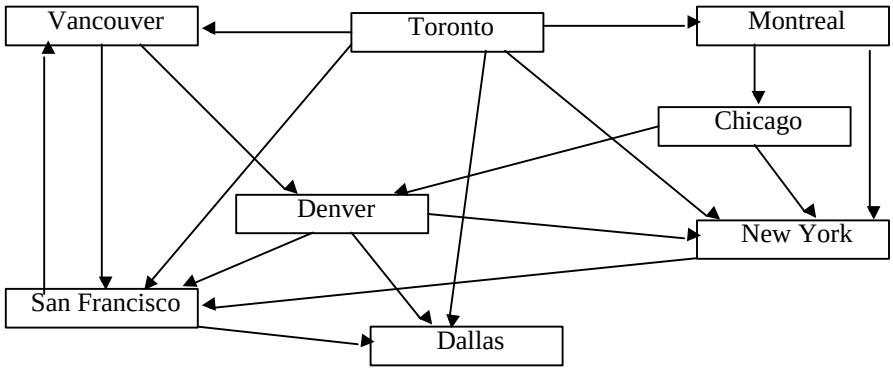


Figure 11.18. Directed graph of fictitious airline connections.

Many books have been written about graphs, their representation, the algorithms for performing various calculations on them, and their relative efficiency. In this section, we will only address the problem of finding the shortest path between two vertices in an undirected graph with weighted edges (the shortest driving time between two cities), finding the shortest path between two vertices in a directed graph with unweighted edges (the minimum number of flight changes), and testing connectivity (existence of a path from one city to another).

Class Graph

All graphs can be represented as directed weighted graphs: Undirected graphs are directed graphs in which all pairs of vertices have either edges in both directions or none, and unweighted graphs can be represented as graphs where all edges have some irrelevant weight, possibly nil. We can thus implement all graphs using just one class whose attributes consist of vertices and weighted edges. This representation is, of course, redundant but we will use it for our examples.

One way to think about a graph is as a set of vertices, each of which has a distinguished value (such as a city name), knows its neighbors (vertices that can be reached from it), and the cost of getting to each neighbor. This is easily implemented by a Dictionary whose keys are vertices and whose values are objects consisting of a connection to a neighbor (edge) and the weight of the connecting edge. We could implement these edge-weight objects as instances of a new class but they are so simple that we will implement them as two-element arrays. As an example, our unweighted airline map could be represented as a dictionary containing the following associations:

```
'Chicago' -> Set ( #('NewYork' nil) #('Denver' nil))
'Dallas' -> Set ()
'Denver' -> Set (#('Dallas' nil) #('NewYork' nil) #('SanFrancisco' nil))
etc.
```

It is tempting to make the new class Graph a subclass of Dictionary but this would be wrong because the two classes don't share any functionality. We will thus make Graph a subclass of Object and put the Dictionary describing its structure into an instance variable called structure. The following is the comment of the class:

```
I implement directed graphs consisting of vertices and weighted edges.
```

Instance variables

structure	<Dictionary>	association key is node, value is a set of two-element arrays consisting of vertex ID and the corresponding edge weight.

As usual, we will start implementation with the creation protocol. The creation method simply creates a new Graph and assigns a dictionary to structure as follows:

```
newWith: aDictionary
"Create a new graph whose structure is defined by aDictionary."
        ^self new dictionary: aDictionary
```

After creating a printOn: method and testing it (left as an exercise), we now define additional accessing methods. As an example, the following accessing method to simplifies access to neighbors:

```
edgesFor: anObject
"Return the set of vertex-weight objects of vertex anObject."
        ^structure at: anObject
```

We are now ready to deal with the graph problems listed at the beginning of this section.

Testing connectivity

Problem: Define a method to determine whether there is a path from one vertex to another, for example, a sequence of flights from Montreal to San Francisco.
Solution: The principle of the solution is simple (Figure 11.19): First, we find and mark all vertices that can be reached directly from to source vertex (Montreal). We then find and mark all vertices that can be reached directly from these cities. And we continue doing this until we either reach the destination vertex (San Francisco) or until there are no more unmarked reachable cities. In the first case there is a path from Montreal to San Francisco, in the second case there is no such path.

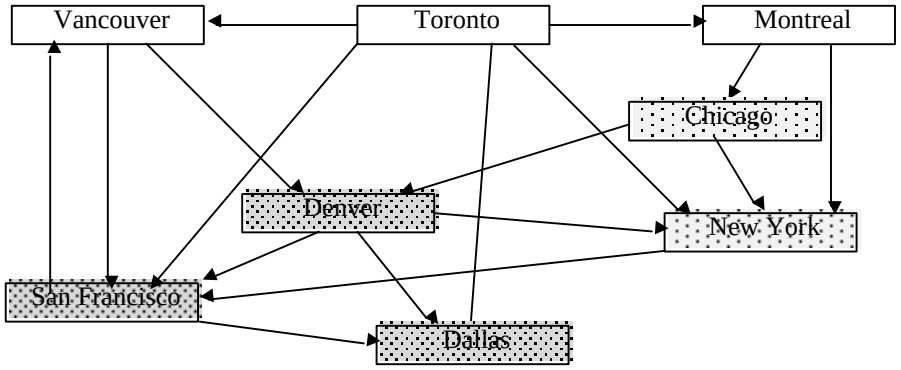


Figure 11.19. Testing connectivity. The source (Montreal) is a white rectangle, adjacent cities are shown with light gray, cities directly reachable from them are shown with darker gray, and so on until we reach San Francisco.

The method implementing this idea requires two arguments – an identification of the source vertex and the destination vertex - and returns true or false. We will call it connected:to: and use it as in

```
flights connected: 'San Francisco' to: 'Toronto'
```

where flights is presumably the graph describing the flight map.

Our definition will follow the informal description given above but refined down to the level of implementation. The informal description shows that we are always dealing with three kinds of vertices: Those that have already been processed, those that are now being used to find new reachable vertices, and those that have not yet been used at all. This can be implemented by keeping vertices of each kind in a collection. If we call these boxes done, active, and remaining, we can describe the algorithm as follows:

1. Put the starting vertex into active, initialize remaining to all other vertices, and done to empty.
2. Repeat the following until the destination vertex appears in active, or until no more vertices are accessible, in other words until active becomes empty:
   a. Enumerate over all vertices in active:
      For each vertex v
         add all its immediate neighbors in remaining into active
         move vertex v from active to done.

Using a queue for the collection leads to a *breadth-first algorithm* which first explores all vertices at the same 'level' from the starting situation. An algorithm that follows a path all the way down and then starts another is called a *depth-first* algorithm. We will use a queue for active as follows:

```
connected: object1ID to: object2ID
"Is the object whose ID is object1ID connected to the object whose ID is object2ID? Return true or false."
        | active done remaining startObject found |
        "Initialization."
        active := OrderedCollection new: structure size.          "Use OrderedCollection as a queue."
        active add: (startObject := structure associationAt: object1ID).
        done := Set new: structure size.
        remaining := structure - (Dictionary with: startObject).
        found := false.
        "Iteration."
        [active isEmpty or: [ found := active contains:
                [:vertex| (vertex value contains: [:array | (array at: 1)  = object2ID])]]] whileFalse:
                        [ |currentVertex neighbors |
                                done add: (currentVertex := active removeFirst).
                                neighbors := self unprocessedNeighborsOf: currentVertex key
                                                from: remaining.
                                active addAll: neighbors.
                                remaining := remaining - (Dictionary withAll: neighbors)].
        ^found
```

where neighborsOf: currentVertex searches collection remaining (unprocessed vertices) for the neighbors of the vertex currently being examined. To make its definition more readable, we expressed it using cities instead of general objects:

```
unprocessedNeighborsOf: aCity from: remainingVertices
"Return collection of associations of all neighbors of aCity that remain unprocessed."
| cities unprocessedNeighbors remainingCities neighboringCityArrays |
        "Gather names of unprocessed cities that are neighbors of aCity."
        remainingCities := remainingVertices keys. "Get names of all remaining city-vertices."
        neighboringCityArrays := (structure at: aCity) select: [ "Select those that neighbor aCity."
                        :setElement | remainingCities contains: [:cityEl | cityEl = (setElement at: 1)]].
        cities := neighboringCityArrays collect: [:array | array at: 1].      "Extract city names out of arrays."
        unprocessedNeighbors:= OrderedCollection new.
        "Now use cities to get associations – the complete vertex descriptions."
        remainingVertices keysAndValuesDo: [:key :value | (cities contains: [:city | city = key])
                        ifTrue: [unprocessedNeighbors add: (remainingVertices associationAt: key)]].
        ^ unprocessedNeighbors
```

Perhaps the trickiest part of these methods was to figure out what the individual messages return. As an example, we made the mistake of using includes: instead of contains: and it took a while before we realized that this was wrong.

To test our code, we used our map and checked whether Montreal is connected to San Francisco (should return true) and whether San Francisco is connected to Toronto (should return false):

```
| cities flights |
“Create data.”
cities := Dictionary new
		add: 'Montreal' -> (Set withAll: #(#('Chicago' nil) #('New York' nil)));
		add: 'New York' -> (Set withAll: #(#('San Francisco' nil)));
		add: 'Chicago' -> (Set withAll: #(#('Denver' nil) #('New York' nil)));
		add: 'Toronto' -> (Set withAll: #(#('Vancouver' nil) #('Montreal' nil) #('New York' nil)
('Dallas' nil) #('San Francisco' nil)));
		add: 'Denver' -> (Set withAll: #(#('New York' nil) #('San Francisco' nil) #('Dallas', nil)));
		add: 'Dallas' -> (Set new);
		add: 'San Francisco' -> (Set withAll: #(#('Dallas' nil) #('Vancouver' nil)));
		add: 'Vancouver' -> (Set withAll: #(#('San Francisco' nil) #('Denver' nil))); yourself.
flights := Graph newWith: cities.
“Do calculations and output results.”
Transcript clear
	show: 'Montreal is ',
		((flights connected: 'Montreal' to: 'San Francisco')
			ifTrue: [''] ifFalse: ['not']),
		' connected to San Francisco';
	cr;
	show: 'San Francisco is ',
		((flights connected: 'San Francisco' to: 'Toronto')
			ifTrue: [''] ifFalse: ['not']),
		' connected to Toronto'
```

The test returns the correct result

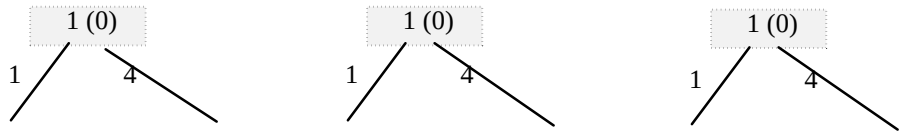
```
Montreal is  connected to San Francisco
San Francisco is not connected to Toronto
```

The shortest path in a weighted undirected graph

Consider our road map now and the problem of finding the shortest path from one city to another. Assuming that all weights are non-negative, this can again be done by successive iterations, constantly enlarging the set S of vertices whose shortest distance to the starting vertex v is already known.

At the beginning, we only have one vertex whose shortest distance to v is known - v itself - and its distance to v is 0. We will thus initialize set S to v. In each consecutive step, we examine all vertices that can be reached from any vertex already in S in exactly one step, select the one whose distance to v (calculated from vertices in S) is the shortest, and add it to S. Since this changes S, we recalculate the shortest distance of all vertices not in S that can be reached in one step from S. If the destination node is reachable from v, it will eventually become a member of S and the distance calculated at that point is its shortest distance from v.

This interesting strategy is called a *greedy algorithm* because it always grabs the most appealing choice. We will prove shortly that it indeed gives the desired result but first, let’s demonstrate how it works when finding the shortest distance from vertex 1 to vertex 6 in Figure 11.20.



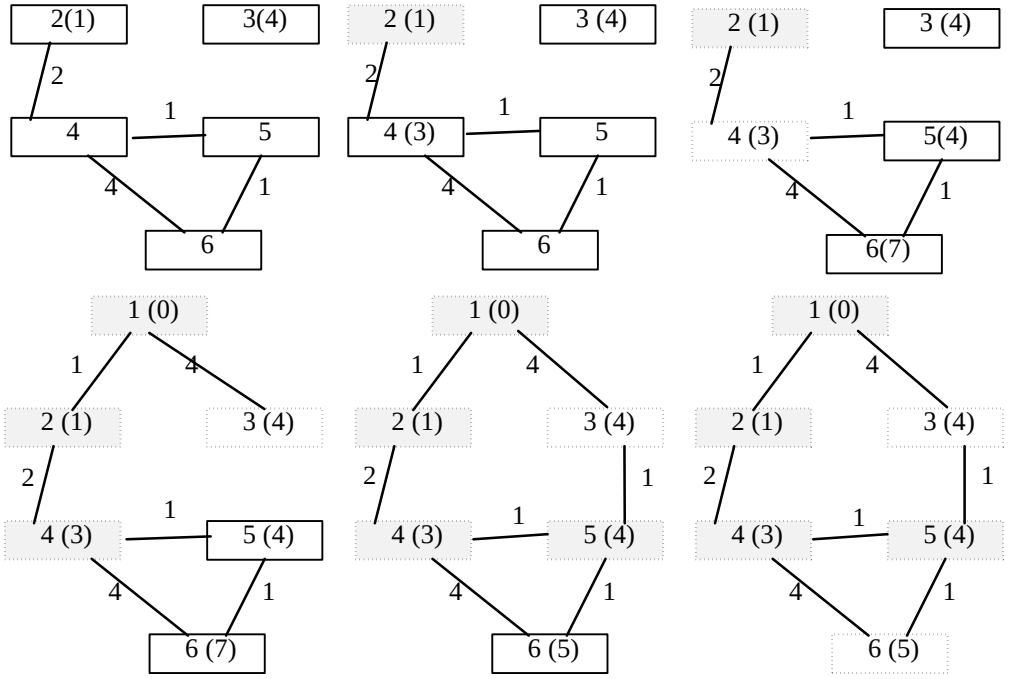


Figure 11.20. Left to right, top to bottom: Finding the shortest distance from vertex 1 to vertex 6.

1. Initialization. Initialize S to source vertex 1: S = {1}. This is indicated in the top leftmost diagram by showing vertex 1 with a dashed border.
2. Iteration 1. S = {1}.
   a. For each vertex reachable from S in one step, calculate the shortest distance from source vertex 1 to this vertex. In our case there are two such vertices - vertex 2 and vertex 3 - and we obtain the distances indicated in the diagram.
   b. Find the vertex that is not in S and whose calculated distance to vertex 1 is the shortest. In our case, this is vertex 2. Add this vertex to S so that S = {1 2}. We indicate that 2 is now in S by drawing its border dashed (second diagram from left).
3. Iteration 2. S = {1 2}.
   a. Recalculate shortest distances to 1 for all vertices not in S. In our case, this does not change existing distances.
   b. Find the vertex closest to v and not in S. In our case, this is vertex 4. Add this vertex to S (second diagram from left).
4. Iteration 3. S = {1 2 4}.
   a. Recalculate shortest distances for vertices not in S. No change in existing distances.
   b. Find the vertex closest to v and not in S. In our case, there are two candidates - vertex 3 and vertex 5, both with distance 4. We arbitrarily choose 3 (third diagram from left).
5. Iteration 4. S = {1 2 4 3}.
   a. Recalculate the shortest distances for vertices not in S. No change in existing distances.
   b. Find the vertex closest to v and not in S and add it to S. This will be vertex 5 3 (first diagram at left bottom).
6. Iteration 5. S = {1 2 3 4 5}.
   a. Recalculate the shortest distances for vertices not in S. This changes the shortest distance between vertex 1 and vertex 6.

b. Find the vertex closest to v and not in S and add it to S. This will be vertex 6 (second diagram from left).

7. There is only one vertex left that is not in S - vertex 6. Add it to S and stop (bottom right diagram). The shortest distance from 1 to 6 has now been found and its value is 5. Note that we found not only the shortest distance between 1 and 6 but also the shortest distance between 1 and all other vertices in the graph reachable from vertex 1.

After demonstrating how the algorithm works, we will now prove that it really produces the shortest distance. Our proof is based on induction and indirection.

Proof: Assume that the algorithm works for some intermediate value of S. (It obviously does when S consists of the source vertex only.) Assume that we just added vertex v to S using our algorithm. According to our claim, the distance from the source vertex to v is the shortest distance from the source to v.

Assume, for a momemnt, that the claim is *false*[5] and that there is a path giving a *shorter* distance from the source to v. Assume that the first vertex on this alternative path that is outside S is x (Figure 11.21). Let the distance from source to v found by our algorithm be dv, and let the distance from the source to v going through x be dx. If the distance through x is shorter, then dx < dv.
Since the distance from x to v is not negative,

dist (source→x) ≤ dist (source→x→v) = dx < dv

This implies dx < dv. However, if dx < dv, our algorithm would have added x to S rather than v because it always adds the closest reachable vertex. Since it added v, the assumption that the path through x is shorter is false. Consequently, the distance obtained for v by our algorithm *is* the shortest distance from the source and v.

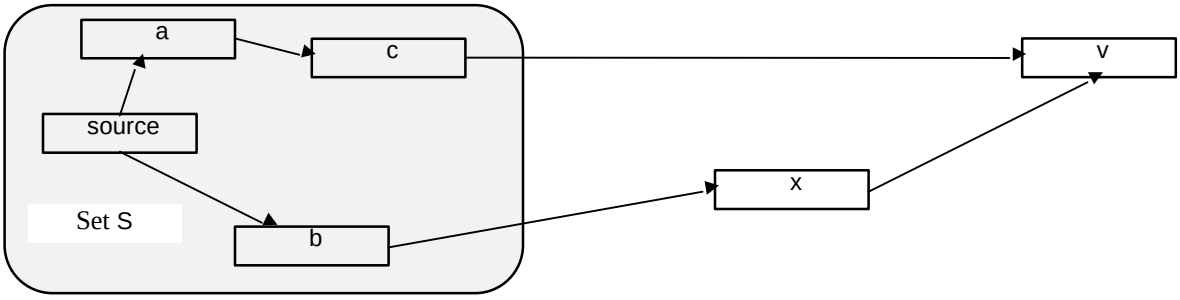


Figure 11.21. Illustration supporting the proof of the shortest path algorithm.

After proving that our algorithm indeed finds the shortest distance, let's formulate it in more detail. Using three collections (remaining, done, activeNeighbors), a more precise description is as follows:

1. Put all vertices except the source vertex s in remaining. Put vertex s in done. Initialize activeNeighbors to an empty collection. Initialize distance of vertex s to 0.
2. Repeat the following until remaining becomes empty or until done contains the destination vertex:
   a. Move all vertices in remaining reachable in one move from done into activeNeighbors
   b. For each vertex in activeNeighbors calculate the shortest distance to s via done.
   c. Move the activeNeighbors vertex whose distance to s is the shortest into done.
3. If done contains the destination vertex, return its distance to s. Otherwise return nil to indicate that there is no path from the source to the destination.

[5] Proving a claim by showing that its negation is false is called an *indirect proof*.

We leave it to you to implement the algorithm as an exercise.

Main lessons learned:

- A graph consists of nodes (vertices) connected by edges.
- Graphs and operations on them can be very complex and their efficient implementation is one of the major areas of research in Computer Science.

Exercises

1. Extend the flight connection checking method to calculate the smallest number of cities that must be traversed if there is a connection.
2. Prove that our connection checking algorithm indeed fulfills its purpose.
3. Design and implement methods that return the shortest weighted and unweighted path in addition to the weighted or unweighted distance.
4. One of the common applications of graphs is representation of activities that must occur in a certain order. Academic courses and their prerequisites and the sequence of activities in building a house are two typical examples. Sorting a directed graph on the basis of vertex precedences is called topological sort. Formulate, implement, and test an algorithm performing topological sort.
5. Examine our implementation of graph algorithms, find the most inefficient points, and improve the implementation.

Conclusion

This chapter introduced several specialized kinds of collections including stacks, queues, linked lists, trees, and graphs. Although most of them are not explicitly included in VisualWorks library, they are essential for Smalltalk operation and very important in Computer Science applications. In exploring these collections, we introduced several features of internal operation of Smalltalk.

A stack is a last-in first-out structure. Elements are added at the top using the push operation, and removed again from the top using the pop operation. Stack behavior is a part of the behavior of OrderedCollection and there is no need for a Stack class.

An important use of stacks is Smalltalk's execution of messages. Execution of Smalltalk messages depends on a stack of context objects, each of them carrying full information about a message including its receiver and sender, its arguments and local variables, and current state of execution. Each message context also has its evaluation stack for intermediate results. When a message is sent, its context is pushed on the top of the context stack and when finished, the context is popped off. A part of the context is a translation of the code into bytecodes.

Another important example of the use of stacks is exception handling. Smalltalk has a built-in mechanism for dealing with exceptional situations and since this process intervenes into message execution, it is very closely tied to the operation of the context stack. The existence of exception handling allows the programmer to anticipate possible exceptional behaviors and deal with them programatically, preventing the program from raising an exception. Exception handling is achieved by sending a message specifying the desired behavior and the exception handling code to an instance of Signal which then delegates the handling to an instance of Exception. A number of Signal objects for dealing with common exceptions are built into the library and users can define their own as well.

A queue is a first-in first-out structure where elements are added at one end and removed from the other. Queue behavior is subsumed by OrderedCollection and there is no need for class Queue. One of the most important applications of queues is in simulation but the Smalltalk run-time environment also uses queues for several operations. Some of these will be covered in Chapter 12.

A list is a linear collection in which each element knows about its successor (single linked list) or its successor and predecessor (doubly linked list). VisualWorks library contains a pair of general classes

called LinkedList and Link implementing the basic linked list behavior. For concrete use, these classes are usually subclassed. The advantage of links is that they allow easy insertion and deletion.

A tree is a branching structure of nodes and their children. The node at the top of a tree is called the root. Every node in a tree except the root has exactly one parent. The root does not have a parent. The bottom nodes in a tree - the nodes that don't have any children - are called leafs.

In general, a node in a tree may have any number of children but specialized trees may restrict the number of children a node is allowed to have. As an example, a node in a binary tree may have at most two children. A very important use of trees is in compilation but the Smalltalk compiler does not build a tree explicitly. Instead, it constructs a nested structure of node objects equivalent to a tree.

Graphs are the most complex type of collection. A graph consists of nodes (vertices) connected by edges. Edges may be directed or undirected, weighted or unweighted. Graphs and operations on them can be very complex and their efficient implementation is one of the major areas of research in Computer Science. Since the operation of Smalltalk does not require any graphs, graphs are not included in the library.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

CompiledMethod, **Exception**, **Signal**, LinkedList, Link.

Terms introduced in this chapter

binary tree - a tree allowing at most two children per node
breadth-first algorithm - an algorithm that deals with all children of a node before examining the children's children
context - information needed to execute a message including its code, sender, receiver, arguments, temporary variables, current state of execution, and working stack
context stack - stack of contexts of all currently active messages stored in order of execution
depth-first algorithm - an algorithm that follows a complete path to a leaf before dealing with sibling nodes on the same level
exception - abnormal behavior such as attempt to divide by zero or attempt to access an illegal index
exception handling - execution of predefined code when an exception occurs
graph - a collection of vertices connected by edges, possibly directed and weighted
leaf - a tree node with no children
lexical analysis - the process of converting textual source code into a collection of program components such as number, word, binary selector, or keyword; first step in compilation
linked list - linear collection held together by single links
parsing - the process of recognizing grammatical constructs such as statements and blocks during compilation; follows scanning and precedes code generation
pop - the act of removing an element from the top of a stack
push - the act of adding an element at the top of a stack
queue - linear collection where elements are added at one end and removed at the other
root - the top node of a tree; has no parent
scanning - synonym of lexical analysis
stack - linear collection where elements are added and removed at the same end
tree - collection of object nodes in which each node may have one or more children, and where each node except the root has exactly one parent