11
Три примера использования наборов

Случайный выбор и карточные игры

Проблема пьяного таракана

Обход бинарных деревьев

Бинарное дерево слов

В этой главе приводятся три примера описания класса. В каждом примере используются классы чисел и наборов, содержащиеся в системе Smalltalk-80. Каждый из примеров иллюстрирует различные способы расширения функциональных возможностей системы.

Карточные игры можно создать с помощью механизма случайного выбора из набора, представляющего колоду карт. Например, экземпляр класса Card (Карта) представляет карту из колоды определенной масти (suit) и достоинства (rank). Класс CardDeck (КолодаКарт) представляет набор таких карт; а CardHand (КартыИгрока) — набор карт у одного играющего. Выбор карты из любого такого набора выполняется на примере классов, которые предоставляют выборку с заменой (класс SampleSpaceWithReplacement (ВыборкаСЗаменой)) или без замены (класс SampleSpaceWithoutReplacement (ВыборкаБезЗамены)). Второй пример — решение известной в программировании задачи о “пьяном таракане”, состоящей в подсчете числа шагов, которые должен сделать таракан, чтобы передвигаясь случайным образом наступить на все плитки пола в комнате. Решение задачи, приведенное в этой главе, представляет каждую плитку пола как экземпляр класса Tile (Плитка), а путешествующее насекомое — как экземпляр класса DrunkenCockroach (ПьяныйТаракан). В третьем примере главы определяется древовидная структура данных, представленная классами Tree (Дерево) и Node (Узел); класс WordNode (УзелСлов) иллюстрирует способ применения структуры дерева для хранения слов, представленных экземплярами класса String, представляющих слова строк.

Случайный выбор и карточные игры

В главе 8 описывался класс системы Smalltalk-80 Random, экземпляры которого действуют как генераторы случайных чисел от 0 до 1. Он дает основу для реализации случайного выбора из множества возможных событий, называемого выборочным пространством. Простую форму дискретной случайной выборки можно получить, если случайное число использовать для выбора элемента из последовательности, например из экземпляра класса SequenceableCollection. Если выбранный элемент остается в наборе, то это выборка “с заменой”, то есть каждый элемент из набора доступен всякий раз, когда происходит выборка. В качестве альтернативы, выбранный элемент может удаляться из набора после того, как выборка сделана; это будет выборка “без замены”.

Класс SampleSpaceWithReplacement представляет случайный выбор с заменой из упорядоченного набора объектов. Экземпляр класса создается определением набора объектов, из которого будет делаться случайный выбор. Это инициализационное сообщение имеет имя data:. Затем мы выбираем элемент из набора, посылая экземпляру класса сообщение next. Если надо выбрать несколько случайных чисел, то экземпляру класса посылается сообщение next: anInteger.

Предположим, мы хотим сделать выборку из массива, содержащего имена людей.

```
people ← SampleSpaceWithReplacement
                    data: #(sally sam sue sarah steve)
```

Каждый раз, когда мы хотим выбрать случайным образом имя из массива, мы должны выполнить выражение

```
people next
```

Ответом будет одно из имен: sally, sam, sue, sarah или steve. Выполняя выражение

```
people next: 5
```

мы получим упорядоченный набор, состоящий из пяти выбранных имен. Экземпляры класса SampleSpaceWithReplacement отвечают на сообщения isEmpty и size, проверяя, существуют ли элементы в выборочном пространстве и сколько элементов там находится. Поэтому, ответом на

```
people isEmpty
```

будет false, а ответом на

```
people size
```

будет 5.

Приведем реализацию класса SampleSpaceWithRepiacement. В каждом методе даются заключенные в кавычки комментарии о назначении метода.

```
class name                  SampleSpaceWithReplacement
superclass                  Object
instance variable names     data
                            rand

class methods

instance creation

    data: aSequenceableCollection
        "Создает экземпляр класса SampleSpaceWithReplacement с аргументом
        aSequenceableCollection в качестве выборочного пространства"
        ↑self new setData: aSequenceableCollection

instance methods

accessing

    next
        "Случайным образом выбирается элемент из набора данных.
        Индекс элемента в наборе определяется выбором случайного числа
        между 0 и 1 с последующей нормализацией, чтобы получить его
        в пределах размера набора данных."
        self isEmpty
            ifTrue: [self error: 'no values exist in the sample space'].
        ↑data at:  (rand next * data size) truncated + 1

    next: anInteger
        "Возвращает упорядоченный набор, содержащий заданное
        в anInteger число случайных выборок из набора данных."
        | aCollection |
        aCollection ← OrderedCollection new: anInteger.
        anInteger timesRepeat: [aCollection addLast: self next].
        ↑aCollection

testing

    isEmpty
        "Проверяет есть ли элементы в выборочном пространстве."
        ↑self size = 0

    size
        "Возвращает число элементов в выборочном пространстве."
        ↑data size

private

    setData: aSequenceableCollection
        "Создает генератор случайных чисел для выборки из пространства,
        заданного аргументом aSequenceableCollection."
        data ← aSequenceableCollection.
        rand ← Random new
```

Описание класса определяет, что каждый экземпляр имеет две переменных с именами data и rand. Сообщение инициализации data: посылает новому экземпляру сообщение setData:, в котором переменная data связывается с набором aSequenceableCollection (значением

аргумента в инициализационном сообщении), а переменная rand — с новым экземпляром класса Random.

Класс SampleSpaceWithReplacement не подкласс класса Collection, поскольку поддерживает малое число тех сообщений, которые доступны классу Collection. В ответ на сообщения next и size, посылаемые экземплярам класса SampleSpaceWithReplacement, посылаются сообщения at: и size переменной data этого экземпляра. Последнее означает, что любой набор, который отвечает на сообщения at: и size, может служить выборочным пространством. Все экземпляры подклассов класса SequenceableCollection отвечают на эти два сообщения. Так, например, кроме массива, как показано в предыдущем примере, выборочное пространство может быть интервалом. Пусть нам нужно смоделировать бросок игральной кости. Тогда элементами выборочного пространства будут положительные целые числа от 1 до 6.

```
die ← SampleSpaceWithReplacement data: (1 to: 6)
```

Бросить игральную кость — выполнить выражение

```
die next
```

Ответом будет одно из чисел 1, 2, 3, 4, 5 или 6.

Подобным же образом мы могли бы выбирать карту из колоды, если бы набор, связанный с экземпляром класса SampleSpaceWithReplacement, состоял из объектов, представляющих игральные карты. Однако, чтобы играть в карты, мы должны иметь возможность сдавать карты игроку и удалять эти карты из колоды. То есть, мы должны уметь выполнять случайный выбор карт без замены.

Чтобы реализовать случайный выбор без замены, определим ответ на сообщение next таким образом, чтобы происходило удаление выбранного элемента. Так как не все экземпляры подклассов класса SequenceableCollection отвечают на сообщение remove: (например, интервалы), мы должны в сообщении инициализации или проверять аргумент или преобразовывать его в приемлемый тип набора. Так как упорядоченные наборы отвечают на оба сообщения и все наборы могут быть преобразованы в упорядоченные, мы можем использовать преобразование. Для того чтобы достичь цели, метод, связанный с именем setData:, посылает своему аргументу сообщение asOrderedCollection.

Класс SampleSpaceWithoutReplacement определяется как подкласс класса SampleSpaceWithReplacement. Методы, связанные с сообщениями next и setData:, переопределены; остальные сообщения наследуются без изменений.

```
class name                  SampleSpaceWithoutReplacement
superclass                  SampleSpaceWithReplacement

instance methods

accessing
    next
        ↑data remove: super next

private
    setData: aCollection
        data ← aCollection asOrderedCollection.
        rand ← Random new
```

Обратите внимание на то, что метод next зависит от метода, реализованного в суперклассе (super next). Метод суперкласса проверяет выборочное пространство на непустоту и, когда оно не пусто, случайным образом выбирает элемент. После чего метод подкласса удаляет выбранный элемент из набора data. Так как сообщение remove: anObject возвращает в качестве результата свой аргумент, результатом посылки сообщения next экземпляру класса SampleSpaceWithoutReplacement будет выбранный элемент.

Теперь реализуем простую карточную игру. Предположим, что элементы выборочного пространства data для карточной игры — экземпляры класса Card, причем каждая карта обладает мастью и достоинством. Экземпляр класса Card будем создавать, посылая классу

сообщение suit:rank:, которое задает двумя своими аргументами масть карты (черви — heart, трефы — club, пики — spade, бубны — diamond) и ее достоинство (1, 2,..., 13). Карта должна отвечать на сообщения suit и rank, возвращая значение соответствующей переменной.

```
class name                  Card
superclass                  Object
instance variable names     suit
                            rank

class methods

instance creation
    suit: aSymbol rank: anInteger
        "Создает карту масти aSymbol с достоинством anInteger."
        ↑self new setSuit: aSymbol rank: anInteger

instance methods

accessing
    suit
        "Возвращает масть карты-получателя."
        ↑suit

    rank
        "Возвращает достоинство карты-получателя."
        ↑rank

private
    setSuit: aSymbol rank: anInteger
        suit ← aSymbol.
        rank ← anInteger
```

Полную колоду карт с именем cardDeck создадим, выполняя следующие выражения.

```
cardDeck ← OrderedCollection new: 52.
#(heart club spade diamond) do:
    [:eachSuit |
      1 to: 13 do: [:n | cardDeck add: (Card suit: eachSuit rank: n)]]
```

Первое выражение создает упорядоченный набор для 52 элементов. Второе выражение — перебор достоинств карт от 1 до 13 для всех четырех мастей (heart, club, spade, diamond). После этого каждый элемент cardDeck становится уникальной картой колоды некоторой масти с некоторым достоинством.

Способность делать выборку из этой колоды карт теперь обеспечивается созданием экземпляра класса SampleSpaceWithoutReplacement с набором cardDeck в роли выборочного пространства:

```
cards ← SampleSpaceWithoutReplacement data: cardDeck
```

Чтобы выбрать карту, достаточно выполнить выражение

```
cards next
```

Ответом на это сообщение будет экземпляр класса Card.

Другой способ создать колоду карт для игры иллюстрируется ниже описанием класса CardDeck. Основная идея состоит в том, чтобы хранить колоду как линейный список карт, а сообщение next понимается как поставщик первой карты списка. Карта может возвращаться в колоду либо в конец списка, либо в некоторую позицию списка, определяемую случайным образом. Линейный список карт создается путем *тасования* колоды (shuffle), то есть случайным упорядочением карт.

В реализации класса CardDeck мы сохраняем первоначальный вариант колоды карт в переменной класса. Она создается с использованием приведенных выше выражений. Всякий раз, когда мы будем создавать новый экземпляр колоды карт, мы будем делать копию этой переменной; перед каждой сдачей она будет перетасовываться. Каждая последующая перетасовка колоды использует текущее состояние переменной экземпляра, а не переменной класса. Конечно, процесс тасования колоды довольно однообразен, так как он базируется на использовании экземпляра класса SampleSpaceWithoutReplacement. Моделирование реального тасования включает первоначальное разбиение колоды карт приблизительно пополам и последующее перемежение этих двух частей. Перемежение состоит из выбора части карт из одной половины, а затем из другой. Действительно, такая модель тасования колоды карт может быть более случайной, чем тасование реального человека; человек, тасуя карты, может и подсмотреть, и передернуть.

При создании карточных игр необходимы такие сообщения экземплярам класса CardDeck, как return:, next и shuffle.

class name	CardDeck
superclass	Object
instance variable names	cards
class variable names	InitialCardDeck

class methods

initialization

initialize

```
"Создает колоду из 52 игральных карт"
InitiailCardDeck ← OrderedCollection new: 52.
#(heart club spade diamond) do:
   [:aSuit |
        1 to: 13
        do: [ :n | InitialCardDeck add: (Card suit: aSuit rank: n)]]
```

instance creation

new

```
"Создает экземпляр класса CardDeck с начальной колодой
из 52 игральных карт."
↑super new cards: InitialCardDeck copy
```

instance methods

accessing

next

```
"Выдает ("сдает") следующую карту."
↑cards removeFirst
```

return: aCard

```
"Помещает карту-аргумент aCard в конец колоды."
cards addLast: aCard
```

shuffle

```
| sample tempDeck |
sample ← SampleSpaceWithoutReplacement data: cards.
tempDeck ← OrderedCollection new: cards size.
cards size timesRepeat: [tempDeck addLast: sample next].
self cards: tempDeck
```

```
testing
    isEmpty
        "Определяет есть ли карты в колоде."
        ↑cards isEmpty

private
    cards: aCollection
        cards ← aCollection
```

Класс CardDeck должен быть инициализирован посредством выполнения выражения

```
CardDeck initialize
```

В реализации класса CardDeck, cards — переменная экземпляра и, следовательно, колода карт для игры. Чтобы начать игру, надо создать экземпляр класса CardDeck:

```
CardDeck new
```

Затем очередная карта сдается посылкой новому экземпляру сообщение next. Когда карту надо поместить обратно в колоду, экземпляру класса CardDeck посылают сообщение return:. Те карты, которые в данное время находятся в колоде cards всегда тасуются. Если после того, как очередной розыгрыш завершился, должна вновь использоваться полная колода карт, все сданные ранее карты должны быть возвращены в колоду.

Обратите внимание на реализацию сообщения shuffle. Сначала, используя текущую колоду карт cards, создается выборочное пространство без замены с именем sample. Затем для хранения случайно выбранных карт создается новый упорядоченный набор с именем tempDeck. В него одна за другой помещаются карты, случайным образом выбранные из sample. Когда все доступные карты перемещены в tempDeck, он сохраняется как текущее состояние игровой колоды карт cards.

Предположим, что мы создаем простую карточную игру, в который принимают участие четыре игрока и банкомет. Банкомет сдает карты каждому игроку. Если по крайней мере один из игроков набирает от 18 до 21 очков, игра завершается с “призом”, разделяемым среди всех таких игроков. Очки игрока считаются суммированием значений карт, полученных им при сдаче. Игрок, набравший более 21 очка, не получает новых карт[1].

Каждого игрока представим экземпляром класса CardHand, который определяется сданными ему картами. Игрок знает, какие карты получены им при сдаче, и может определить общее количество очков в ответ на сообщение points.

```
class name                  CardHand
superclass                  Object
instance variable names     cards

class methods

instance creation
    new
        ↑super new setCards

instance methods

accessing
    take: aCard
        "Аргумент aCard добавляется к получателю
        (игрок берет карту)."
        cards add: aCard

    returnAllCardsTo: cardDeck
        "Возвращает все карты получателя в колоду-аргумент cardDeck
        и удаляет эти карты из рук получателя."
```

[1] Упрощенный вариант игры в “очко” — Прим. ред. перев.

```
        cards do: [ :eachCard | cardDeck return: eachCard].
        self setCards

inquiries
    points
        "Возвращает сумму значений всех карт получателя."
        ↑cards inject: 0 into: [:value :nextCard | value + nextCard rank]

private
    setCards
        cards ← OrderedCollection new
```

Теперь создадим множество из четырех игроков — множество с именем players. Каждый игрок — экземпляр класса CardHand. Сдаваемая колода карт — экземпляр класса CardDeck с именем gameCards. Банкомет (здесь это программист) тасует колоду; победителей еще нет. По условиям игры победителей может быть больше одного, поэтому все они будут перечисляться в множестве с именем winners.

```
players ← Set new.
4 timesRepeat: [players add: CardHand new].
gameCards ← CardDeck new.
gameCards shuffle
```

Пока нет победителей, каждому игроку, у которого количество очков меньше чем 21, сдают карту из колоды gameCards. Перед сдачей по очередной карте каждому из игроков, которые могут продолжать игру, банкомет проверяет, не появились ли победители, просматривая число очков у каждого игрока.

```
[winners ← players select: [:each | each points between: 18 and: 21].
 winners isEmpty and: [gameCards isEmpty not]]
    whileTrue:
        [players do:
            [:each |
                each points < 21 ifTrue: [each take: gameCards next]]
```

Условие для продолжения игры — блок с двумя выражениями. Первое выражение определяет победителей, если они есть. Второе выражение проверяет, появились ли победители (winners isEmpty) и, если нет, проверяет остались ли в колоде карты для сдачи (gameCards isEmpty not). Если нет победителей и еще есть карты, игра продолжается. Игра состоит из перебора всех игроков и сдаче каждому игроку новой карты (each take: gameCards next), если только число очков у него меньше чем 21 (each points < 21). Чтобы начать игру сначала, все сданные карты должны быть возвращены в колоду, которая вновь тасуется.

```
players do: [ :each | each returnAllCardsTo: gameCards].
gameCards shuffle.
```

Игроки и банкомет готовы начать игру сначала.

Проблема пьяного таракана

Мы можем применить некоторые классы наборов для решения известной в программировании задачи. Перед нами стоит задача подсчитать, какое число “шагов” сделает “пьяный таракан” пока не коснется всех квадратных плиток пола, имеющего N плиток в ширину и M плиток в длину. Немного конкретизируем задачу. Сделать “шаг” таракан может с равной степенью вероятности на любую из девяти плиток, а именно: на ту, где он находится (остается на месте) и на те, которые ее непосредственно окружают.

Конечно же, перемещение таракана будет ограничиваться стенами комнаты. Задача состоит в подсчете числа шагов, сделанных тараканом по полу, до тех пор, пока он не посетит *каждую* плитку хотя бы один раз.

Один из наиболее простых алгоритмов решения задачи мог бы начинаться с определения пустого множества и установленного в 0 счетчика шагов. После каждого шага, мы добавляем к множеству ту плитку, на которую наступил таракан и увеличиваем на 1 счетчик шагов. Так как дублирование элементов в множестве не дозволяется, все будет закончено, когда число элементов в нашем множестве достигнет N*M. Решением будет значение счетчика шагов.

В то время как предложенный алгоритм решает самую простую версию задачи, мы могли бы востребовать и некоторую дополнительную информацию, например, сколько раз каждая плитка пола посещалась тараканом? Чтобы сохранить эту информацию, мы можем использовать вместо множества простой набор. Размер этого набора будет общим числом сделанных шагов; размер множества, полученного преобразованием из этого набора, определяет общее число различных плиток, на которые наступил таракан. Когда последний размер достигает N*M, задача будет решена. Число вхождений каждой плитки в простой набор будет равно числу посещений тараканом этой плитки.

Каждая плитка пола может быть определена ее строкой и столбцом. Объекты, представляющие плитки в решаемой задаче, мы представим экземплярами класса Tile (Плитка). Реализация класса Tile с комментариями может быть такой.

class name Tile
superclass Object
instance variable names location
floorArea

instance methods

accessing

location

```
"Возвращает координаты получателя."
↑location
```

location: aPoint

```
"Устанавливает координаты получателя
равными аргументу aPoint"
location <- aPoint
```

floorArea: aRectangle

```
"Устанавливает область пола равной прямоугольной
области aRectangle"
floorArea <- aRectangle
```

moving

neighborAt: deltaPoint

```
"Создает и возвращает новую плитку, расположенную в точке,
координаты которой представляют координаты расположения
получателя, измененные на аргумент deltaPoint.
Обеспечивает координаты новой плитки newTile внутри
границ области пола."

| newTile |
newTile ← Tile new floorArea: floorArea.
newTile location: ((location + deltaPoint max: floorArea origin)
        min: floorArea corner).
↑newTile
```

```
comparing
    = aTile
        "Возвращает true, когда получатель равен аргументу aTile."
        ↑(aTile isKindOf: Tile) and: [location = aTile location]

    hash
        ↑location hash
```

Положение плитки определяется номерами ее строки и столбца, которые должны быть не меньше 1 и не больше ширины или длины пола. Поэтому в дополнение к тому, что плитка запоминает свое положение, она должна помнить максимальное пространство пола, на котором она размещается. Плитке можно послать сообщение neighborAt: aPoint для того, чтобы определить новую плитку в одной из возможных позиций. Эта новая плитка должна быть расположена в пределах границ пола.

Способ, с помощью которого мы будем моделировать движение таракана, должен выбирать направление движения в терминах изменения координат (x, y) его местоположения. Для каждого местоположения таракана (плитки с координатами x, y), существуют 9 плиток, на которые насекомое может передвинуться, если только плитка не лежит на границе площади пола. Мы будем хранить возможные изменения x и y в упорядоченном наборе, который используем для случайного выбора направления дальнейшего движения. Упорядоченный набор содержит элементами точки, которые представляют векторы всех возможных перемещений таракана. Мы создадим этот набор, выполняя выражения

```
Directions ← OrderedCollection new: 9.
(–1 to: 1) do: [ :x | (–1 to: 1) do: [ :y | Directions add: x @ y]].
```

Тогда набор Directions будет совокупностью элементов

```
–1 @ –1, –1 @ 0, –1 @ 1, 0 @ –1, 0 @ 0, 0 @ 1, 1 @ –1, 1 @ 0, 1 @ 1.
```

Как составную часть модели движения таракана, мы будем генерировать случайное число для выбора элемента из набора Directions. В другом случае, вместо непосредственного использования генератора случайных чисел мы могли бы использовать экземпляр класса SampleSpaceWithReplacement из предыдущего примера с набором Directions в качестве выборочного пространства.

Предположим, таракан начинает движение с плитки пола, расположенной в точке 1 @ 1. Каждый раз, когда таракан предполагает сделать шаг, мы случайным образом получаем элемент из набора Directions. Этот элемент используем как аргумент сообщения neighborAt:, посылаемого плитке, на которой находится таракан. Далее будем полагать, что Rand — экземпляр класса Random.

```
tile neighborAt:
    (Directions at: ((Rand next * Directions size) truncated + 1))
```

В результате будет возвращен экземпляр класса Tile — новая плитка пола, на которую наступил таракан.

Каждую плитку пути нужно запоминать, чтобы иметь возможность определить, все ли плитки прошел таракан и сколько всего им сделано шагов. Сохраняя каждую такую плитку в простом наборе, мы сохраняем информацию о числе посещений тараканом каждой плитки пола. Поэтому при каждом шаге создается новая плитка, которая является копией предшествующей плитки. Новая плитка определяется по случайно выбранному направлению движения и добавляется в простой набор. Когда число уникальных элементов этого набора сравняется с общим числом плиток пола, путь таракана завершается.

В классе DrunkenCockroach нужны только два сообщения. Первое — команда таракану начать движение по заданной прямоугольной площади пола с заданной плитки пола и завершить его, когда каждая плитка будет пройдена хотя бы один раз. Это сообщение walkWithin:startingAt:. Второе — запрос относительно сделанного насекомым числа шагов; это сообщение numberOfSteps. Мы можем еще спросить о числе посещений тараканом заданной плитки пола, посылая экземпляру класса DrunkenCockroach сообщение

timesSteppedOn:. Набор векторов всевозможных перемещений (как это описано ранее) создается как переменная класса DrunkenCockroach; генератор случайных чисел Rand — также переменная класса DrunkenCockroach.

```
class name                  DrunkenCockroach
superclass                  Object
instance variable names     currentTile
                            tilesVisited
class variable names        Directions
                            Rand

class methods

class initialization
    initialize
        "Создает набор векторов перемещений и генератор случайных чисел"
        Directions ← OrderedCollection new: 9.
        (–1 to: 1) do: [ :x | (–1 to: 1) do: [ :y | Directions add: x @ y]].
        Rand ← Random new

instance creation
    new
        ↑super new setVariables

instance methods

simulation
    walkWithin: aRectangle startingAt: aPoint
        | numberTiles |
        tilesVisited ← Bag new.
        currentTile location: aPoint.
        currentTile floorArea: aRectangle.
        numberTiles ← (aRectangle width + 1) * (aRectangle height + 1).
        tilesVisited add: currentTile.
        [tilesVisited asSet size < numberTiles] whileTrue:
           [currentTile ← currentTile neighborAt:
                (Directions at: (Rand next * Directions size) truncated + 1).
            tilesVisited add: currentTile]

data
    numberOfSteps
        ↑tilesVisited size

    timesSteppedOn: aTile
        ↑tilesVisited occurrencesOf: aTile

private
    setVariables
        currentTile ← Tile new.
        tilesVisited ← Bag new
```

Чтобы поэкспериментировать с “пьяным тараканом”, прежде всего необходимо инициализировать класс DrunkenCockroach и создать экземпляр этого класса.

```
DrunkenCockroach initialize.
cockroach ← DrunkenCockroach new
```

Получить результаты 10 экспериментов на полу размером 5 на 5 мы сможем, выполняя выражения

```
result ← OrderedCollection new: 10.
10 timesRepeat:
   [cockroach walkWithin: (1 @ 1 corner: 5 @ 5) startingAt: (1 @ 1).
    result add: cockroach numberOfSteps]
```

Средний результат 10 экспериментов - это среднее число шагов, которые требуются насекомому, чтобы выполнить условия задачи:

```
(results inject: 0 into: [:sum :exp | sum + exp]) / 10
```

Обратите внимание, что в реализации сообщения с именем walkWithin:startingAt: условие завершения задачи — равенство размера набора tilesVisited, преобразованного в множество, числу N * M. Более быстрый способ сделать то же самое мог бы состоять в том, чтобы добавить сообщение uniqueElements в класс Bag и использовать его, а не преобразовывать каждый раз простой набор в множество.

Для читателей, желающих пробовать провести это изменение, отметим, что метод, добавляемый к классу Bag должен иметь вид

```
uniqueElements
    ↑contents size.
```

Тогда сообщение с именем walkWithin:startingAt: может быть изменено так, чтобы условием завершения метода стало выражение tilesVisited uniqueElements < numberTiles

---

Обход бинарных деревьев

Дерево — важная нелинейная структура данных, используемая во многих компьютерных алгоритмах. Структура дерева означает, что существует разветвление между элементами. Есть один элемент, обозначаемый как корень дерева. Если есть только один элемент, то это — корень. Если элементов более одного, то они разбиваются на непересекающиеся (под)деревья. Бинарное дерево — это либо только корень, либо корень с одним двоичным (под)деревом, либо корень с двумя двоичными (под)деревьями. Полное описание структуры дерева приведено в книге Д. Кнута *“Искусство программирования для ЭВМ”*, М.: Мир, 1976, т.1, стр. 382-500. Здесь мы предполагаем, что читатель знаком с основными идеями, и поэтому можем приступить к описанию этой структуры данных в виде класса языка Smalltalk-80.

Мы определим класс Tree (Дерево) подобно тому, как определяли класс LinkedList. Элементы дерева — это узлы (экземпляры класса Node), которые подобны экземплярам класса Link в классе LinkedList. Узел позволяет соединять элементы дерева между собой. Каждый экземпляр класса Tree будет ссылаться только на узел корня.

Узел представляется в языке Smalltalk-80 объектом с двумя переменными экземпляра, одна переменная указывает на *левый* узел, а другая на *правый* узел. Мы определим, что упорядочивание узлов происходит “слева-направо”. Это означает, что сначала надо пройти левый подузел, затем корень и, наконец, правый подузел[2]. Если узел не имеет подузлов, то он называется *листом*. Введем понятие “размер узла”, который определим как 1 плюс размер подузлов, если они есть. Таким образом, лист — узел размера 1, а узел с двумя листами как подузлами имеет размер 3. Размер дерева — это размер корня дерева. Такое определение размера соответствует общему понятию размера для наборов.

Сообщения к экземпляру класса Node обеспечивают доступ к его левому узлу, правому узлу и к последнему узлу. Также есть возможность удалить подузел (remove:ifAbsent:) и корень (rest).

---

[2] Такой обход еще называют обходом в прямом порядке. О порядках прохождения узлов бинарного дерева можно прочитать в книге Д. Кнута, *“Искусство программирования для ЭВМ”*, М., “Мир”, 1976, т.1, стр. 394-410. — Прим. перев.

class name	Node
superclass	Object
instance variable names	leftNode rightNode

class methods

instance creation

left: lNode right: rNode

```
"Создает экземпляр класса Node с аргументами lNode и rNode,
как левым и правым подузлами, соответственно."
| newNode |
newNode ← self new.
newNode left: lNode.
newNode right: rNode.
↑newNode
```

instance methods

testing

isLeaf

```
"Возвращает true, если получатель - лист,
то есть узел без подузлов"
↑leftNode isNil & rightNode isNil
```

accessing

left

```
↑leftNode
```

left: aNode

```
leftNode ← aNode
```

right

```
↑rightNode
```

right: aNode

```
rightNode ← aNode
```

size

```
↑1 + (leftNode isNil ifTrue: [0] ifFalse: [leftNode size])
   + (rightNode isNil ifTrue: [0] ifFalse: [rightNode size])
```

end

```
| aNode |
aNode ← self.
[aNode right isNil] whileFalse: [aNode ← aNode right].
↑aNode
```

removing

remove: subnode ifAbsent: exceptionBlock

```
"Принимает за корень получатель, self,
который не может быть удален"
self isLeaf ifTrue: [↑exceptionBlock value].
 leftNode = subnode
    ifTrue: [leftNode <- leftNode rest. ↑subnode].
rightNode = subnode
    ifTrue: [rightNode <- rightNode rest. ↑subnode].
leftNode isNil
```

```
            ifTrue: [↑rightNode remove: subnode ifAbsent: exceptionBlock].
        ↑leftNode
            remove: subnode
            ifAbsent: [rightNode isNil
                        ifTrue: [exceptionBlock value]
                        ifFalse: [rightNode remove: subnode
                                        ifAbsent: exceptionBlock]]

    rest
        leftNode isNil
            ifTrue: [↑rightNode]
            ifFalse: [leftNode end right: rightNode.
                    ↑leftNode]

enumerating
    do: aBlock
        leftNode isNil ifFalse: [leftNode do: aBlock].
        aBlock value: self.
        rightNode isNil ifFalse: [rightNode do: aBlock]
```

Если узел — лист, это обозначается так: , где
тогда

левый узел правый узел

то есть

Перечисление бинарного дерева происходит “слева-направо”, поэтому сначала перечисляется левый подузел как значение аргумента блока, затем корень, и, наконец, правый подузел. Блок в этом случае должен определяться как блок с аргументом-узлом.

Определим класс Tree, как подкласс класса SequenceableCollection, чьи элементы — узлы. Экземпляр класса Tree имеет одну переменную экземпляра с именем root (корень), при этом root — либо nil, либо экземпляр класса Node. Как подкласс класса SequenceableCollection, класс Tree реализует сообщения add: anElement, remove: anElement ifAbsent: exceptionBlock и do: aBlock. В основном, методы связанные с каждым из этих сообщений выясняют, пусто ли дерево (root isNil) и, если нет, передают соответствующее сообщение корню. Проверка “на пустоту” наследуется из класса Collection. Цель в том, чтобы программист, использующий структуру дерева, получал доступ к элементам этой структуры только через экземпляр класса Tree.

class name	Tree
superclass	SequenceableCollection
instance variable names	root

instance methods

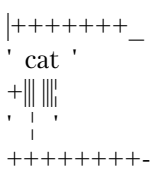
```
testing
    isEmpty
        ↑root isNil

accessing
    first
        | save |
        self emptyCheck.
        save ← root.
        [save left isNil] whileFalse: [save ← save left].
        ↑save

    last
        self emptyCheck.
        ↑root end

    size
        self isEmpty
            ifTrue: [↑0]
            ifFalse: [↑root size]

adding
    add: aNode
        ↑self addLast: aNode

    addFirst: aNode
        "Если набор пуст, тогда аргумент aNode - новый корень;
        иначе, это – левый узел текущего первого узла."
        self isEmpty
            ifTrue: [↑root ← aNode]
            ifFalse: [self first left: aNode].
        ↑aNode

    addLast: aNode
        "Если набор пуст, тогда аргумент aNode - новый корень;
        иначе это – последний элемент текущего корня. "
        self isEmpty
            ifTrue: [root ← aNode]
            ifFalse: [self last right: aNode].
        ↑aNode

removing
    remove: aNode ifAbsent: exceptionBlock
        "Сначала проверяет корень. Если нужное не найдено, перемещается
        вдоль дерева, проверяя каждый узел."
        self isEmpty ifTrue: [↑exceptionBlock value].
        root = aNode
            ifTrue: [root ← root rest. ↑aNode]
            ifFalse: [↑root remove: aNode ifAbsent: exceptionBlock]

    removeFirst
        self emptyCheck.
        ↑self remove: self first ifAbsent: []
```

```
    removeLast
        self emptyCheck.
        ↑self remove: self last ifAbsent: []

enumerating
    do: aBlock
        self isEmpty ifFalse: [root do: aBlock]

private
    emptyCheck
        self isEmpty
            ifTrue: [self error: 'the tree is empty']
```

Обратите внимание, что сообщения удаления не удаляют поддерево, начинающееся узлом, который требуется удалить, а удаляют только сам узел.

Бинарное дерево слов

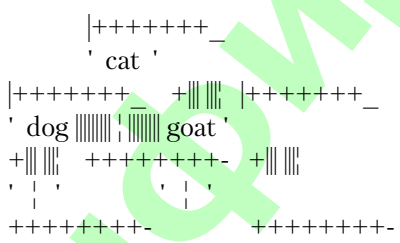
Определение класса Tree, подобно классу Link, является структурой без содержания. Мы наполним содержанием каждый узел дерева с помощью определения подкласса. Предположим, мы хотим использовать экземпляры класса Node, чтобы сохранять слова, представленные строками. Мы называем этот класс WordNode (УзелСлов). Узел слов будем создавать, посылая классу WordNode или сообщение for:, если не надо создавать подузлов, или сообщение for:left:right:, если одновременно надо создать два подузла. Так, узел слов, иллюстрируемый как



создается при выполнении выражения

```
WordNode for: 'cat'
```

Узел слов, который выглядит как



создается при выполнении выражения

```
WordNode for: 'cat'
        left: (WordNode for: 'dog' )
        right: (WordNode for: 'goat')
```

Реализация класса WordNode приведена ниже. Обратите внимание, что равенство (=) переопределяется, чтобы определять совпадение слов в узлах; это означает, что наследуемые сообщения удаления будут удалять подузел, если его слово такое же, как и слово-аргумент сообщения.

class name WordNode
superclass Node
instance variable names word

class methods

```
instance creation
    for: aString
        ↑self new word: aString

    for: aString left: lNode right: rNode
        | newNode |
        newNode ← super left: lNode right: rNode.
        newNode word: aString.
        ↑newNode
```

instance methods

```
accessing
    word
        ↑word

    word: aString
        word ← aString

comparing
    = aWordNode
        ↑(aWordNode isKindOf: WordNode) and: [word = aWordNode word]

    hash
        ↑word hash
```

Проиллюстрируем использование класса WordNode последовательностью выражений. Заметим, что в определении класса WordNode ничего не сделано для того, чтобы реализовать добавление новых элементов таким образом, чтобы при обходе дерева происходило алфавитное сравнение с каждым словом. Заинтересованный читатель может добавить в класс aWordNode сообщение вида insert: aWordNode, которое реализовывало бы сортировку слов в алфавитном порядке.

```
tree ← Tree new.

tree add: (WordNode for: 'cat')


tree addFirst: (WordNode for: 'frog')


tree addLast:
    (WordNode for: 'horse' left: (WordNode for: 'monkey') right: nil)


tree addFirst: (WordNode for: 'ape')


tree remove: (WordNode for: 'horse')
```

tree remove: (WordNode for: 'frog')