При решении задач, как сугубо теоретических, так и практических, возникает необходимость в работе с данными. Но помимо самих данных, необходимо иметь инструментарий, который позволит их обработать в соответствие со спецификой задачи, в связи с чем возникает потребность в хранении данных внутри памяти ЭВМ. Введем ряд определений.

Определение 2.1. *Ячейкой памяти* будем называть структурную единицу ЭВМ, которая позволяет хранить некоторый фиксированный объем информации, выражающийся целым числом *байт*.

В силу того, что данные могут быть различны по своей природе, введем понятие *тип данных* (определение 2.2).

Определение 2.2. *Типом данных* будем называть некоторое конечное множество значений, каждое из которых требует для своего хранения фиксированное число ячеек памяти.

Типы данных принято разделять на *примитивные* и *составные*. К примитивным типам данных относятся такие, как булевский тип, числовые типы данных, символьные, указатели и другие. Составной тип данных представляет собой логическое объединение конечного числа как примитивных типов данных, так и составных, в том числе и самого себя. Составной тип данных обычно принято называть *структурой* или *записью*. Базируясь на введенных выше определениях, которые можно считать атомарными и присущими любой ЭВМ, дадим следующее определение *структуры данных* (определение 2.3).

Определение 2.3. *Структурой данных* будем называть программную единицу ЭВМ, описывающую способ организации ячеек памяти, хранящих данные одного и того же типа.

Таким образом, структуры данных позволяют хранить необходимые наборы данных в памяти ЭВМ. Неотъемлемой частью любой такой структуры является набор операций, который позволяет эффективно управлять данными, хранящимися в структуре. Таковыми обычно являются операции *чтения* и *записи*, предоставляющие доступ к хранимым данным. Следует отметить, что обе операции обычно реализуются на уровне ЭВМ, поэтому в дальнейшем будем предполагать, что их трудоемкость составляет $O(1)$ и не зависит от того, каким типом данных оперирует рассматриваемая структура.

2.1. Элементарные структуры данных

Под *элементарными структурами данных* будем понимать структуры данных, которые либо предоставляются непосредственно ЭВМ, либо моделируются средствами ЭВМ с использованием типов данных, доступных на уровне ЭВМ. Таким образом, элементарной структурой данных можно назвать любую базовую структуру данных ЭВМ. Ниже будут рассмотрены такие, как *массив* и *связный список*. Необходимо отметить, что их можно считать атомарными структурами данных, так как не предполагается их моделирование путем использования других элементарных структур данных. Хотя такая возможность и существует с точки зрения ЭВМ, построенная реализация не даст ощутимый

выигрыш перед системной структурой данных, поскольку она заведомо ведет к снижению эффективности некоторых из базовых операций, таких, как вставка и удаление элемента.

2.1.1. Массивы

Массив – линейная, однородная, элементарная структура данных, представляющая собой непрерывную упорядоченную последовательность ячеек памяти и предназначенная для хранения данных одного и того же типа.

Доступ к элементам массива осуществляется по *индексу*, либо по *набору индексов*, где под индексом будем понимать некоторое целочисленное значение. В общем виде массив представляет собой многомерный параллелепипед, в котором величины измерений могут быть различны. Массив A с m измерениями будем называть *m-мерным* и обозначать через $A[n_1, n_2, \dots, n_m]$, где n_i – число элементов, которое может быть записано на уровне i $(1 \le i \le m)$.

Таким образом, массив A с m измерениями позволяет хранить внутри себя $k = n_1 \cdot n_2 \cdot \dots \cdot n_m$ элементов с затратами памяти $k \cdot s = O(k)$, где s – количество ячеек памяти, требуемое для хранения одного элемента.

Как было отмечено ранее, любой массив – это непрерывный участок памяти, поэтому многомерный массив технически представляет собой одномерный массив, доступ к элементам которого осуществляется по следующей формуле:

$$(i_1, i_2, \dots, i_m) \Leftrightarrow \\ ((\dots(i_1 \cdot n_2 + \dots) \cdot n_{m-2} + i_{m-2}) \cdot n_{m-1} + i_{m-1}) \cdot n_m + i_m, \quad (2.1)$$

где i_j – позиция элемента по i-ому измерению, $0 \le i_j < n_j, 1 \le j \le m$.

В дальнейшем изложении будем предполагать, что индексация элементов массива начинается с единицы, если не оговорено иное. Процедура вычисления соответствующего индекса согласно приведенному выражению представлена ниже (процедура 2.1.1):

Процедура 2.1.1. Get Index

```
01:  Get Index Impl(i1, i2, ..., im)
02:         index = 0
03:         for j = 1 to m – 1 do
04:               index = (index + ij) * nj+1
05:         index = index + im
06:         return index
```

Рассмотрим основные операции, которые позволяют манипулировать данными, хранящимися в массиве, при этом не ограничивая общности, будем считать, что задан одномерный массив $A[n]$, состоящий из n элементов:

1) чтение / запись элемента требуют $O(1)$ времени, так как массив является структурой данных с произвольным доступом;

2) трудоемкость поиска элементов в массиве зависит от того, упорядочен он или нет: в случае упорядоченного массива поиск необходимых данных мож-

но осуществить за $O(\log n)$ операций, иначе потребуется $O(n)$ операций в худшем случае;

3) вставка / удаление элементов в массив требуют $O(n)$ операций в худшем случае, так как изменение структуры массива требует перемещения элементов. Если же вставка и удаление элементов осуществляются только на конце массива, то трудоемкость операций составит $O(1)$. Алгоритм вставки элемента в массив приведен ниже (процедура 2.1.2).

Процедура 2.1.2. Array: Insert

```
01:  Insert Impl(A, index, element)
02:        if index > 0 then
03:              for i = n downto index do
04:                    A[i + 1] = A[i]
05:              A[index] = element
```

В заключение еще раз подчеркнем, что многомерный массив можно неявно рассматривать как массив массивов одинакового размера. В то же самое время существует понятие *рваного массива*[*], использование которого оправдано в том случае, если его элементами являются массивы произвольной длины и следует явно обозначить, что он состоит из массивов. В этом случае рваный массив будем обозначать как $A[\ \ ][\ \ ]\ldots[\ \ ]$. Пример рваного массива приведен ниже (рис. 2.1):

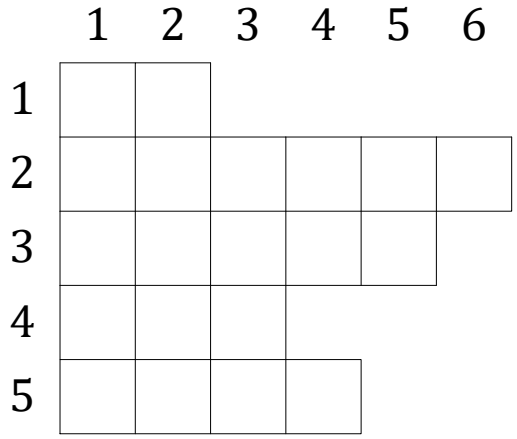

Рис. 2.1. Пример рваного массива $A[5][\ \ ]$

Следует отметить, что большинство современных императивных языков программирования имеют возможности для работы с массивами. Некоторые из языков предоставляют возможности для создания рваных массивов, как одного из типов данных.

Как было отмечено выше, операции вставки и удаления элемента в массив осуществляются в худшем случае за $O(n)$ операций. Поэтому возникает необходимость в разработке структур данных, которые позволят избавиться от данного недостатка и, как следствие, предоставят эффективные методы по включению/исключению элементов. К достоинствам же массива можно отнести экономное расходование памяти, требуемое для хранения данных, а также

[*] В англоязычной литературе используется термин *jagged array*.

быстрое время доступа к элементам массива. Следовательно, применение массивов оправдано в том случае, если обращение к данным происходит по целочисленному индексу и в ряде случаев, не предполагающих модификацию набора данных путем увеличения / уменьшения числа его элементов.

2.1.2. Связные списки

Связным списком будем называть линейную, однородную, элементарную структуру данных, предназначенную для хранения данных одного и того же типа, в которой порядок объектов определяется указателями. В отличие от массива, связный список, как следует из данного определения, хранит элементы не обязательно в последовательных ячейках памяти. Данным свойством и обосновано название «Связный список» рассматриваемой структуры данных, поскольку при помощи указателей ячейки памяти связываются в единое целое.

Рассматривая связный список, будем предполагать, что его элемент описывается составным типом данных, который хранит в себе пользовательские данные некоторого типа и указатель на следующий элемент списка. У связного списка принято различать концевые элементы: *головой* называют начальный элемент списка, *хвостом* – конечный. Таким образом, рекурсивное описание связного списка выглядит следующим образом (процедура 2.1.3):

Процедура 2.1.3. Linked List Type

```
01:  linkedListElement<T> {
02:        item: T
03:        link: linkedList
04:  }
05:  linkedList<T> {
06:        head: linkedListElement<T>
07:        tail: linkedListElement<T>
08:  }
09:  emptyLinkedList<T> = linkedList<T> {
10:        head = null
11:        tail = null
12:  }
```

Помимо данных пользователя и указателей на соседние элементы, элемент списка может хранить и некоторую служебную информацию, продиктованную спецификой задачи (например, время создания элемента списка, адрес элемента в памяти, информацию о владельце и т. д.). На практике же чаще всего используются те либо иные вариации связных списков, которые в большинстве своем обусловлены особенностями итерирования по списку. Основными из них являются односвязные, двусвязные и кольцевые списки, которые будут рассмотрены ниже вместе с базовым набором операций.

Если это не будет приводить к противоречиям, то пустой список далее будем обозначать через символ ∅.

Односвязные списки

Односвязным списком будем называть связный список, в котором каждый из элементов списка содержит указатель непосредственно на следующий элемент того же списка.

Односвязный список является простейшей разновидностью связного списка, поскольку каждый его элемент предполагает хранение минимальной информации для связи элементов списка в единую структуру. Можно видеть, что данное определение является идентичным определению связного списка, поэтому и описание их будет похожим. Детали, касающиеся односвязного списка, приведены ниже (процедура 2.1.4):

Процедура 2.1.4. Single Linked List Type

```
01:  singleLinkedListElement<T> {
02:        item: T
03:        next: singleLinkedList
04:  }
05:  singleLinkedList<T> {
06:        head: singleLinkedListElement<T>
07:        tail: singleLinkedListElement<T>
08:  }
```

Рассмотрим такие основные операции, как вставка элемента в список, удаление элемента из списка и поиск элемента в списке. Перед непосредственным рассмотрением необходимо сказать о ряде вспомогательных процедур: 1) процедура, которая проверяет, является ли список пустым; 2) процедура получения значения первого элемента списка; 3) процедура получения списка, на который указывает начальный элемент текущего списка (процедуры 2.1.5, 2.1.6):

Процедура 2.1.5. Single Linked List: Is Empty

```
01:  Is Empty Impl(L)
02:        if L.head = L.tail = null then
03:              return true
04:        return false
```

Процедура 2.1.6. Single Linked List: First, Next

```
01:  First Impl(L)
02:        if not Is Empty(L) then
03:              return L.head.item
04:        return ∅
05:
06:  Next Impl(L)
07:        if not Is Empty(L) then
08:              return L.head.next
09:        return ∅
```

Рассмотрим также процедуры разделения списка по заданному элементу (2.1.7) и процедуру соединения двух списков в единое целое (2.1.8). Процедура разделения возвращает два списка, а процедура соединения по двум спискам

формирует один. Отметим, что операция разделения будет работать только для непустых списков.

Процедура 2.1.7. Single Linked List: Split

```
01:  Split Impl(L, listElement)
02:        Left  = L
03:        Right = listElement.next
04:        listElement.next = ∅
05:        return Left, Right
```

Процедура 2.1.8. Single Linked List: Join

```
01:  Join Impl(Left, Right)
02:        if Is Empty(Left)  then return Right
03:        if Is Empty(Right) then return Left
04:        with Left do
05:              tail.next = Right
06:              tail = Right.tail
07:        return Left
```

В зависимости от того, на какую позицию помещается вставляемый элемент, процедура его вставки в список может претерпевать некоторые изменения. В связи с этим, как отдельные случаи следует рассмотреть вставку элемента в пустой список (процедура 2.1.9) и вставку элемента в начало, конец либо середину списка (процедура 2.1.10).

Процедура 2.1.9. Single Linked List: Insert Empty

```
01:  Insert Empty Impl(L, dataItem)
02:        if Is Empty(L) then
03:              temp = singleLinkedListElement<T> {
04:                    item = dataItem
05:                    next = ∅
06:              }
07:              L = singleLinkedList<T> {
08:                    head = temp
09:                    tail = temp
10:              }
11:        return L
```

Процедура 2.1.10. Single Linked List: Insert Front/Insert Back, Insert At

```
01:  Insert Front Impl(L, dataItem)
02:        return Join(Insert Empty(∅, dataItem), L)
03:
04:  Insert Back Impl(L, dataItem)
05:        return Join(L, Insert Empty(∅, dataItem))
06:
07:  Insert At Impl(L, listElement, dataItem)
08:        Left, Right = Split(L, listElement)
09:        return Join(Insert Back(Left, dataItem), Right)
```

Операция удаления элемента списка, по сравнению с операцией вставки, представляет некоторые сложности, так как для ее корректного выполнения требуется ссылка на элемент списка, расположенный непосредственно перед

удаляемым элементом. В качестве альтернативы можно воспользоваться операцией удаления элемента, который следует непосредственно после элемента, поступающего на вход процедуре. Таким образом, процедура удаления элемента списка, следующего непосредственно после заданного элемента, приведена в процедуре 2.1.11:

Процедура 2.1.11. Single Linked List: Delete

```
01:  Delete Impl(L, listElement)
02:        Left, Right = Split(L, listElement)
03:        if Is Empty(Right) then
04:              return Left
05:        return Join(Left, Next(Right))
```

Отметим, что все операции, рассмотренные выше, имеют трудоемкость $O(1)$ в худшем случае.

Операция поиска заданного значения в списке требует просмотра всех элементов списка и поэтому имеет трудоемкость $O(n)$ (процедура 2.1.12):

Процедура 2.1.12. Single Linked List: Contains

```
01:  Contains Impl(L, dataItem)
02:        if not Is Empty(L) then
03:              if (First(L) = dataItem) then
04:                    return true
05:              return Contains Impl(Next(L), dataItem)
06:        return false
```

В заключение вышеизложенного, приведем реализацию операций вставки и удаления, которые не будут изменять существующий список, а будут только создавать его копию, если это необходимо. Наличие таких операций позволяет сделать список неизменяемым (процедура 2.1.13):

Процедура 2.1.13. Single Linked List: Immutable Insert and Delete Ops

```
01:  Insert Front Impl(L, dataItem)
02:        if Is Empty(L) then
03:              return Insert Empty(L, dataItem)
04:        return singleLinkedList<T> {
05:              head = singleLinkedListElement<T> {
06:                    item = dataItem
07:                    next = L
08:              }
09:              tail = L.tail
10:        }
11:
12:  Insert Back Impl(L, dataItem)
13:        return Insert At(L, L.tail, dataItem)
14:
15:  Insert At Impl(L, listElement, dataItem)
16:        return IRec(L)
```

```
17:         IRec(L)
18:               if L.head = listElement then
19:                     return Insert Front(
20:                               Insert Front(Next(L), dataItem), First(L))
21:               return Insert Front(IRec(Next(L)), First(L))
22:
23:   Delete Impl(L, listElement)
24:         return DRec(L)
25:
26:         DRec(L)
27:               if L.head = listElement then
28:                     return Next(L)
29:               return Insert Front(DRec(Next(L)), First(L))
```

Отметим, что неизменяемые операции вставки элемента в конец списка, а также удаления последнего элемента из списка, в худшем случае имеют трудоемкость $O(n)$. Несмотря на кажущуюся неэффективность приведенных операций, они находят широкое применение при реализации неизменяемых *абстрактных типов данных*, которые более детально будут рассмотрены ниже.

Двусвязные списки

Двусвязным списком будем называть связный список, в котором каждый из элементов содержит указатель как на следующий элемент списка, так и на предыдущий.

Рекурсивное описание двусвязного списка приведено в процедуре 2.1.14:

Процедура 2.1.14. Double Linked List Type

```
01:   doubleLinkedListElement<T> {
02:         item: T
03:         next: doubleLinkedList<T>
04:         prev: doubleLinkedList<T>
05:   }
06:   doubleLinkedList {
07:         head: doubleLinkedListElement<T>
08:         tail: doubleLinkedListElement<T>
09:   }
```

На практике двусвязные списки находят широкое применение, поскольку они позволяют итерироваться по элементам списка в обоих направлениях. Что же касается операций над двусвязным списком, то они будут претерпевать незначительные изменения по сравнению с операциями, приведенными для односвязного списка. Трудоемкость операций вставки, удаления, поиска составит $O(1)$ в случае изменяемой структуры данных. Для неизменяемой либо персистентной структуры данных «Двусвязный список», выполнение любых n операций над списком будет требовать времени порядка $O(n^2)$, из расчета, что работа начинается с пустого списка. В этом случае можно говорить об амортизированном времени выполнения каждой из операций $O(n)$.

Кольцевые списки

Кольцевым списком будем называть связный список, в котором последний элемент ссылается на первый.

Обычно кольцевые списки реализуются на основе либо односвязных списков, либо двусвязных. В силу того, что список замкнут в кольцо, требуется всего лишь один указатель, который будет определять точку входа в список. Реализации базовых операций для кольцевого списка не приводятся по причине большого сходства с соответствующими операциями для односвязного списка. К основным достоинствам кольцевых списков следует отнести возможность просмотра всех элементов списка, начиная с произвольного, а также быстрый доступ к первому и последнему элементам в случае, если список реализован на базе двусвязного списка.

2.2. Абстрактные типы данных. Общая идеология

Под *абстрактным типом данных (АТД)* будем понимать математическую модель с определенным на ней набором операций и правил поведения, которые определяют ожидаемый результат каждой из операций.

Суть каждой из операций состоит в изменении внутреннего состояния АТД. Поведение же определяет инвариант, который должен сохраняться на протяжении всего жизненного цикла АТД вне зависимости от того, какие операции были выполнены в тот либо иной момент. В некоторых случаях поведение можно рассматривать как политику, которой должна придерживаться каждая из операций, определенная на АТД. В зависимости от правил, которые диктует поведение, абстрактные типы данных принято разделять на *изменяемые* и *неизменяемые*, а также на *персистентные* и *неперсистентные*[*]. Говоря об неизменяемых и персистентных АТД, будем подразумевать, что каждую из операций, определенную на АТД, можно отнести к одному из трех типов – создания, обновления, чтения. Операции создания используются для создания конкретного экземпляра АТД с последующей инициализацией его внутреннего состояния. Под операциями обновления (или изменения) будем понимать такие, выполнение которых ведет к изменению внутреннего состояния АТД. Под операциями чтения – все иные операции.

Под *неизменяемыми* АТД будем понимать такие, внутреннее состояние которых не изменяется на протяжении их жизненного цикла. Таким образом, неизменяемые АТД предполагают инициализацию внутреннего состояния операциями создания, а последующие операции изменения ведут к созданию нового экземпляра того же АТД.

Под *персистентными* АТД будем понимать такие, которые сохраняют себя после каждой операции изменения с последующей возможностью доступа к каждому из сохраненных экземпляров. Различают *частично-персистентные*

[*] В англоязычной литературе соответственно используются термины *immutable* и *persistent*.

и *полностью персистентные* АТД. В частично-персистентных операции обновления разрешены только для последней версии экземпляра АТД, в то время как полностью персистентные предполагают модификацию каждой из сохраненных версий. В большинстве случаев персистентные структуры данных являются неизменяемыми либо частично неизменяемыми.

В противоположность неизменяемым и персистентным, *изменяемые* АТД не преследуют своей целью сохранение внутреннего состояния после каждой операции изменения. Как следствие, изменяемые АТД обладают более высокой производительностью, а также характеризуются более экономным расходованием памяти, требуемой для хранения данных. В то же самое время следует отметить, что неизменяемые и персистентные структуры данных находят широкое применение на практике, в частности в функциональных языках программирования, при работе с параллельными вычислениями, а также в других прикладных отраслях, связанных с информационными технологиями.

Одним из основных понятий абстрактных типов данных является понятие *интерфейса* АТД, определяющего набор операций, доступный для манипулирования внутренним состоянием АТД. Следует понимать, что интерфейс предоставляет ограниченный набор операций для работы с экземплярами АТД, в то время как весь набор операций, определенный на АТД, может быть значительно шире. Основной целью, преследуемой интерфейсом, является предоставление возможности описания внутренних деталей АТД с использованием различных структур данных, а также других АТД. Таким образом, интерфейс определяет набор операций и поведение, в то время как конкретные реализации одного и того же АТД могут отличаться как по времени выполнения операций, так и по затратам памяти.

Рассматривая абстрактные типы данных, следует уделить внимание концепции *контейнера*. Под *контейнером* будем понимать АТД, предназначенный для хранения однотипных наборов данных и предоставляющий методы для манипулирования этими данными. Здесь следует отметить, что существует понятие *коллекции* данных, которое во многом пересекается с понятием контейнера. Чтобы избежать двусмыслицы, в дальнейшем будем придерживаться следующего соглашения: если речь идет о конкретном АТД, то будем говорить о контейнере, если же речь идет о наборе однотипных элементов, то будем говорить о коллекции данных. В этом смысле контейнер, как было отмечено выше, предоставляет возможности по сохранению коллекций данных. Таким образом, большинство АТД, которые будут рассмотрены ниже, суть контейнера, основным предназначением которых является эффективное хранение и манипулирование коллекциями данных.

Контейнеры классифицируют по следующим способам:

1) доступа к данным: то, каким образом контейнер предоставляет доступ к элементам коллекции данных, – является ли он контейнером с последовательным доступом либо с произвольным. Например, в случае обычного массива это может быть индекс, а в случае стека –вершина стека;

2) хранения данных: то, каким образом элементы коллекции данных сохранены в контейнер, – в виде массива, связного списка либо древовидной структуры данных;

3) итерирования: то, каким образом осуществляется просмотр элементов коллекции данных, – в прямом, обратном или ином порядке.

Контейнеры принято разделять на *последовательные* и *ассоциативные*. Под *последовательным* контейнером будем понимать контейнер, чьи элементы хранятся в строго линейном порядке. Под *ассоциативным* – тот, который предназначен для хранения коллекций, состоящих из пар «ключ – значение ключа», где каждый из возможных ключей встречается не более одного раза.

Неотъемлемым компонентом контейнера является *перечислитель* (или *итератор*), основное предназначение которого заключается в предоставлении набора операций, позволяющих осуществить просмотр содержимого контейнера. В зависимости от того, какая структура данных выбрана для сохранения коллекции данных, итераторы могут отличаться друг от друга для одного и того же АТД.

Следует отметить, что АТД, которые по своей природе являются контейнерами, обычно расширяют базовый набор операций контейнера путем добавления функций, специфических для данного АТД. К базовым же операциям следует отнести такие, как определение количества элементов в контейнере; проверка того, является ли контейнер пустым; преобразование контейнера в массив; создание контейнера из массива и др. Дальнейшее изложение не будет акцентировать внимание на упомянутых операциях в силу их простоты как с точки зрения понимания, так и реализации.

В общем случае все абстрактные типы данных можно условно разделить на *линейные* и *нелинейные*. Под *линейными* АТД будем понимать те, которые используют массивы либо связные списки для хранения коллекции данных. Под *нелинейными* АТД будем понимать все остальные (в частности те, которые используют иерархические структуры данных). Следует отметить, что как линейные, так и нелинейные АТД могут быть отнесены к классу либо последовательных, либо ассоциативных контейнеров, в зависимости от того, с какими наборами данных работает АТД.

Ниже будут рассмотрены АТД, получившие наиболее широкое распространение. Общей особенностью, касающейся всех рассматриваемых ниже АТД, является их возможность реализации как посредством массивов, так и посредством связных списков. Способ той либо иной организации данных выбирается исходя из специфики задачи.

2.2.1. Список

Под *списком* будем понимать линейный АТД, предназначенный для хранения упорядоченных наборов данных одного и того же типа. Альтернативным определением списка является определение через последовательный контейнер с произвольным доступом к элементам коллекции данных.

С точки зрения математики список представляет собой *конечную последовательность* однотипных элементов. Основными операциями над списком являются создание списка (как пустого, так и из массива элементов), вставка элемента в список, удаление элемента из списка и поиск элемента в списке. В зависимости от того, какое поведение накладывается на операции над списком, в качестве внутренней структуры может быть выбран массив либо связный список. Так, если требуется, чтобы экземпляр списка был неизменяемым или персистентным, то наиболее удачным будет выбор связного списка. То же самое касается и случаев, если над списком планируется проводить большое число операций изменения. Массивы же для представления списков чаще всего используются в тех случаях, когда заранее известно максимальное количество элементов, которое будет помещено в список (так называемая *емкость* списка), а также в случаях, когда элементы только вставляются в список, и не имеет принципиальной важности позиция вставки элемента.

Ниже рассмотрим реализацию абстрактного типа данных «Список» через массив (процедура 2.2.1). Реализация списка через структуру данных «Связный список» (будь то односвязный либо двусвязный) рассматриваться не будет, поскольку АТД «Список» по сути своей адаптирует методы связных списков для своих целей.

Процедура 2.2.1. ADT List

```
01:  list<T> {
02:        item: T[max items count]
03:        next: integral[max items count]
04:        itemsCount: integral
05:  }
```

К основной процедуре над рассмотренным списком относится вставка элемента в список, описание которой приведено в процедуре 2.2.2:

Процедура 2.2.2. ADT List: Insert

```
01:  Insert Impl(L, dataItem)
02:        with L do
03:              item[itemsCount + 1] = dataItem
04:              next[itemsCount + 1] = itemsCount
05:              itemsCount = itemsCount + 1
```

Процедуру проверки принадлежности заданного значения списку можно реализовать, например, так (процедура 2.2.3):

Процедура 2.2.3. ADT List: Contains

```
01:  Contains Impl(L, dataItem)
02:        with L do
03:              current = itemsCount
04:              while current ≠ 0 do
05:                    if item[current] = dataItem then
06:                          return true
07:                    current = next[current]
08:        return false
```

Еще раз отметим, что приведенная реализация списка через массив лучшим образом подходит для тех случаев, в которых не предполагается осу-

ществлять удаление элементов. Если же операция удаления необходима, то лучше прибегнуть к реализации АТД «Список» через связный список.

В качестве примера реализации списка через массив, рассмотрим следующую задачу. Пусть требуется построить список L, состоящий из n списков, – $L_1, L_2, \dots, L_n$. Будем предполагать, что списки $L_i (1 \le i \le n)$ хранят данные одного и того же типа и что суммарная длина списков L_i не превышает некоторого целого числа C. Также будем предполагать, что в распоряжении имеется $O(C)$ ячеек памяти, т. е. достаточное для того, чтобы сохранить данные списков L_i. Тогда описание списка L будет иметь следующий вид (процедура 2.2.4):

Процедура 2.2.4. ADT List of Lists

```
01:  listOfLists<T> {
02:        item: T[C]
03:        head: integral[C]
04:        next: integral[C]
05:        itemsCount: integral
06:  }
```

В сравнение со списком, описание которого приведено в процедуре 2.2.1, в описание списка L добавлен дополнительный массив, предназначенный для хранения позиций-начал каждого из списков L_i. Если же таковое требуется, то можно хранить и позиции-окончания каждого из списков L_i.

Процедура добавления элемента в список L_i будет выглядеть следующим образом (процедура 2.2.5):

Процедура 2.2.5. ADT List of Lists: Insert

```
01:  Insert Impl(L, i, dataItem)
02:        with L do
03:              item[itemsCount + 1] = dataItem
04:              next[itemsCount + 1] = head[i]
05:              head[i] = itemsCount + 1
06:              itemsCount = itemsCount + 1
```

Если же требуется обработать элементы списка L_i, то это можно сделать при помощи процедуры 2.2.6:

Процедура 2.2.6. ADT List of Lists: Process

```
01:  Process Impl(L, i)
02:        with L do
03:              current = head[i]
04:              while current ≠ 0 do
05:                    {* process current item *}
06:                    current = next[current]
```

В заключение отметим, что списки принято разделять на *динамические* и *статические*. Статические списки обычно не предполагают исключения элементов, поэтому их реализация обычно базируется на массивах. Наиболее же гибкая и эффективная реализация данного АТД базируется на основе двусвязного списка, поскольку все операции, кроме операции поиска, требуют в худшем случае $O(1)$ операций. Если же речь идет о неизменяемом списке, то в худшем случае вставка и удаление элементов будут требовать $O(n)$ операций.

2.2.2. Стек

Стек (или *магазин*) – линейный абстрактный тип данных, доступ к элементам которого организован по принципу LIFO (от англ. last-in-first-out).

В соответствие с принципом LIFO для стека должна быть определена *точка входа* (или *вершина стека*), через которую будет осуществляться вставка и удаление элементов. В частном случае стек можно рассматривать как список, работа с элементами которого происходит только на одном из концов списка.

Эффективные реализации стека базируются как на массивах, так и на связных списках. Рекомендуется использовать реализацию через массив в тех случаях, когда требуется добиться высокой производительности и экономного расходования памяти. В общем же случае наиболее предпочтительным оказывается подход, базирующийся на идее односвязного списка (можно сказать, что структура односвязного списка в этом случае используется неявно). Данный подход представляется наиболее интересным как с точки зрения расширения функционала, поддерживаемого стеком, так и с точки зрения преобразования данного АТД в неизменяемый тип данных. Детали реализации АТД «Стек» приведены в процедуре 2.2.7:

Процедура 2.2.7. ADT Stack

```
01:  stackEntry<T> {
02:        item: T
03:        next: stack
04:  }
05:  stack<T> {
06:        entry: stackEntry<T>
07:  }
```

Процедуры вставки, удаления и получения значения на вершине стека приведены ниже (процедура 2.2.8):

Процедура 2.2.8. ADT Stack: Push, Pop, Top

```
01:  Push Impl(S, dataItem)
02:        return stack<T> {
03:              entry = stackEntry<T> {
04:                    item = dataItem
05:                    next = S
06:              }
07:        }
08:
09:  Pop Impl(S)
10:        if not Is Empty(S) then
11:              return S.entry.next
12:        return ∅
13:
14:  Top Impl(S)
15:        if not Is Empty(S) then
16:              return S.entry.item
```

Можно видеть, что поведение операций, определенных на стеке данным образом, задается следующим набором инвариантов:

$$\mathrm{Top}\big(\mathrm{Push}(S, e)\big) = e,$$
$$\mathrm{Pop}\big(\mathrm{Push}(S, e)\big) = S.$$

Следует отметить, что для стека не определена операция поиска значения. Если же такая операция необходима, то реализовать ее можно с трудоемкостью $O(n)$ следующим образом (процедура 2.2.9):

Процедура 2.2.9. ADT Stack: Contains

```
01:  Contains Impl(S, dataItem)
02:        if not Is Empty(S) then
03:              if Top(S) = dataItem then
04:                    return true
05:              return Contains Impl(Pop(S), dataItem)
06:        return false
```

2.2.3. Очередь

Очередь – линейный абстрактный тип данных, доступ к элементам которого организован по принципу FIFO (от англ. first-in-first-out).

В соответствие с принципом FIFO для очереди должны быть определены две точки входа: *начало* очереди, откуда происходит удаление элементов, и *конец* очереди, куда происходит добавление элементов. В частном случае очередь можно рассматривать как список, вставка элементов в который происходит на одном конце, а удаление элементов – на другом.

Эффективные реализации очереди базируются как на массивах, так и на связных списках. Рекомендуется использовать реализацию через массив тогда, когда требуется добиться высокой производительности и экономного расходования памяти, а также в случаях, когда заранее известно, что операции добавления и удаления элементов встречаются равновероятно (тогда достаточно завести два счетчика, каждый из которых будет перемещаться в начало массива в тот момент, когда достигнута последняя ячейка выделенного участка памяти). Как и для АТД «Стек», здесь наиболее предпочтительным оказывается подход, базирующийся на идее односвязного списка. В силу простоты реализации данный подход рассматриваться не будет. Более интересным представляется моделирование очереди через два стека, как с точки зрения реализации интерфейса одного АТД через другой АТД, так и с точки зрения возможностей, предлагаемых данным подходом. Детали описания очереди через АТД «Стек» приведены в процедуре 2.2.10:

Процедура 2.2.10. ADT Queue

```
01:  queue<T> {
02:        inStack:  stack<T>
03:        outStack: stack<T>
04:  }
```

Процедуры вставки, удаления и получения значения первого элемента очереди представлены в процедуре 2.2.11:

Процедура 2.2.11. ADT Queue: Enqueue, Dequeue, Peek

```
01:  Enqueue Impl(Q, dataItem)
02:        return queue<T> {
03:              inStack  = Push(Q.inStack, dataItem)
04:              outStack = Q.outStack
05:        }
06:
07:  Dequeue Impl(Q)
08:        if not Is Empty(Q) then
09:              Balance(Q)
10:              return queue<T> {
11:                    inStack  = Q.inStack
12:                    outStack = Pop(Q.outStack)
13:              }
14:        return ∅
15:
16:  Peek Impl(Q)
17:        if not Is Empty(Q) then
18:              Balance(Q)
19:              return Top(Q.outStack)
```

Отдельного внимания заслуживает операция балансировки стеков, суть которой заключается в том, чтобы переместить данные из входного стека в выходной (процедура 2.2.12):

Процедура 2.2.12. ADT Queue: Balance

```
01:  Balance Impl(Q)
02:        with Q do
03:              if Is Empty(outStack) then
04:                    while not Is Empty(inStack) do
05:                          outStack = Push(outStack, Top(inStack))
06:                          inStack  = Pop(inStack)
```

В силу того, что элементы, добавленные в очередь, помещаются в каждый из двух стеков ровно один раз, то можно считать, что время, затраченное на выполнения E операций добавления и D операций удаления будет пропорционально $O(E + D)$. Если же очередь реализована через массив либо связный список, то учетное время выполнения каждой из операций составит $O(1)$. Как и для стека, для АТД «Очередь» не определена операция поиска значения. При необходимости такую операцию можно реализовать с трудоемкостью $O(n)$ следующим образом (процедура 2.2.13):

Процедура 2.2.13. ADT Queue: Contains

```
01:  Contains Impl(Q, dataItem)
02:        with Q do
03:              return Contains(inStack, dataItem) or
04:                        Contains(outStack, dataItem)
```

2.2.4. Дек

Дек – линейный абстрактный тип данных, доступ к элементам которого организован по принципу двусторонней очереди[*].

Наиболее тривиальным определением дека является определение через список, в который запрещена вставка элементов в середину либо определение через очередь, где оба конца открыты для полного манипулирования данными. Дек является наиболее гибкой и, как следствие, наиболее сложной структурой данных, предоставляющей широкий доступ к формированию последовательности, в которой будут храниться необходимые наборы данных. В соответствие с определением дека, наиболее подходящей структурой данных для его реализации является двусвязный список, позволяющий производить все операции на деке (кроме операции поиска) с трудоемкостью $O(1)$. Если же требуется эффективная реализация неизменяемого дека, то структура неизменяемого двусвязного списка не подойдет по причине неэффективности операций, выполняемых над ним (напомним, что в этом случае каждая из операций вставки и удаления будет выполняться за $O(n)$). Следовательно, требуется использовать другой подход, отличный от использования связных списков.

Рассмотрим структуру данных, которая позволит эффективно реализовать неизменяемый АТД «Дек». Идея, которая лежит в основе рассматриваемого подхода заключается в том, что любой дек можно представить в виде трех последовательностей элементов: левый элемент, средняя часть дека и правый элемент (процедура 2.2.14):

Процедура 2.2.14. ADT Deque

```
01:  deque<T> {
02:        left:   T
03:        right:  T
04:        middle: deque<T>
05:  }
06:  singleDeque<T>: {
07:        item: T
08:  }
```

Отметим, что как одиночный дек, так и пустой дек подчиняются общему интерфейсу дека, т. е. над этими типами данных допустимы операции, определенные над АТД «Дек». Так как дек является двусторонней очередью, то отдельно стоит рассмотреть операции вставки в начало / конец дека и удаления из начала / конца дека (процедура 2.2.15)[**]:

Процедура 2.2.15. ADT Deque: Enqueue, Dequeue, Peek

```
01:  Empty Deque
02:        Enqueue Left Impl(D, dataItem)
03:              return singleDeque<T> {
04:                    item = dataItem
05:              }
```

[*] Термин «дек» происходит от английского *deque – double-ended queue*.

[**] Процедуры вставки, удаления и извлечения элементов приведены только для левого конца дека. Процедуры для правого конца дека будут аналогичны.

```
06:  Single Deque
07:        Enqueue Left Impl(D, dataItem)
08:              return deque<T> {
09:                    left   = dataItem
10:                    right  = Peek Left(D)
11:                    middle = ∅
12:              }
13:  Deque
14:        Enqueue Left Impl(D, dataItem)
15:              return deque<T> {
16:                    left   = dataItem
17:                    right  = Peek Right(D)
18:                    middle = Enqueue Left(middle, Peek Left(D))
19:              }
20:
21:        Dequeue Left Impl(D)
22:              with D do
23:                    if Is Empty(middle) then
24:                          return singleDeque<T> {
25:                                item = Peek Right(D)
26:                          }
27:                    return deque<T> {
28:                          left   = Peek Left(middle)
29:                          right  = Peek Right(D)
30:                          middle = Dequeue Left(middle)
31:                    }
32:
33:        Peek Left Impl(D)
34:              with D do
35:                    return left
```

Недостатком приведенной реализации дека является наличие рекурсивных вызовов в операциях вставки и удаления элементов. Таким образом, если требуется вставить n элементов в дек, то затраченное время составит $O(n^2)$. Существуют модификации данного подхода, позволяющие улучшить время работы процедур вставки и удаления до $O(\log n)$ и даже до $O(1)$. За деталями можно обратиться к источникам [1, 2, 3].

2.2.5. Дерево. Терминология и варианты реализации

Выше были рассмотрены линейные структуры данных, предназначенные для хранения последовательностей элементов. Так как последовательности задаются отношением линейного порядка, такие структуры данных, как массив и связный список, наиболее подходят для их хранения.

Зачастую возникают задачи, в которых данные связаны более сложными отношениями, поэтому есть необходимость в разработке структур данных, учитывающих особенности связей между рассматриваемыми элементами. Одним из отношений, описывающим порядок, отличный от линейного, является общее отношение частичного порядка.

Будем говорить, что набор данных *M* образует *иерархию*, если на нем задано отношение частичного порядка, которое не допускает циклических зависимостей (таким образом, последовательность является частным случаем иерархии). Условно иерархию можно представить в виде уровней, на каждом из которых находятся элементы, попарно между собой не принадлежащие заданному отношению и следующие непосредственно за элементами предыдущего уровня, с которыми они образуют отношение. Говоря об иерархии, обычно выделяют элемент, у которого нет предшественников. В соответствие с данным определением описание иерархии может выглядеть следующим образом (процедура 2.2.16):

Процедура 2.2.16. Hierarchy Type

```
01:  hierarchyEntry<T> {
02:        item: T
03:        next: hierarchy<T>[]
04:  }
05:  hierarchy<T> {
06:        entry: hierarchyEntry<T>
07:  }
08:  emptyHierarchy<T> = hierarchy<T>
```

Наиболее удачной математической моделью для представления иерархий является дерево. Неформально *дерево* можно определить как совокупность элементов, называемых *узлами* (или *вершинами*), и отношений, образующих иерархическую структуру узлов. Формально же дерево можно определить несколькими способами.

Определение 2.4. *Дерево* – это связный граф без циклов. Под *корневым деревом* будем понимать ориентированный граф со следующими свойствами:

1) существует в точности одна вершина, входящая степень которой равняется нулю; такую вершину принято называть *корнем дерева*;

2) входящая степень каждой из вершин дерева, отличных от корневой, равняется единице;

3) существует единственный путь от корня к каждой из вершин дерева.

Альтернативным определением является следующее рекурсивное определение дерева.

Определение 2.5. *Дерево* – это математический объект, определенный рекурсивно следующим образом:

1) один узел, называемый *корнем дерева*, является деревом;

2) пусть n – это узел, а $T_1, \dots, T_k$ – непересекающиеся между собой деревья с корнями $n_1, \dots, n_k$ соответственно. Тогда, сделав узел n родителем узлов n_i, получим новое дерево T с корнем n и поддеревьями $T_1, \dots, T_k$. В этом случае, узлы $n_1, \dots, n_k$ называются *дочерними узлами* (или *детьми*) узла n.

Будем говорить, что дерево T является *упорядоченным деревом*, если на сыновьях каждого из узлов дерева заданы отношения порядка. В противном случае дерево T является *неупорядоченным*.

Пусть T – некоторое упорядоченное дерево. Рассмотрим узел n дерева T с детьми $n_1, \dots, n_k$. *Левым сыном* узла n назовем такой узел n_i, для которого не существует узла n_j такого, что $n_j \preccurlyeq n_i$. *Правым братом* узла n_i назовем такой узел n_j, для которого не существует узла n_k такого, что $n_i \preccurlyeq n_k \preccurlyeq n_j$.

Листом назовем узел, у которого нет детей. Альтернативным может служить определение, согласно которому *лист* это узел, детьми которого являются пустые деревья. *Уровнем узла* назовем число, на единицу большее уровня родительского узла при условии, что уровень корневого узла есть 0. Тогда *высотой дерева* назовем наибольший из уровней узлов дерева. Очевидно, что высоту дерева определяют листья данного дерева.

Согласно определению 2.5 для представления иерархического набора данных в памяти ЭВМ можно выбрать следующую структуру данных, схожую со связными списками (процедура 2.2.17):

Процедура 2.2.17. Tree Data Type

```
01:  treeNode<T> {
02:        item: T
03:        children: container<tree<T>>
04:  }
05:  tree<T> {
06:        root: treeNode<T>
07:  }
```

В качестве контейнера для хранения детей данной вершины обычно выбирается последовательный контейнер (например, список) либо одна из линейных структур данных (массив, односвязный список и т. д.).

Предположив, что узлы дерева занумерованы натуральными числами от 1 до n, получим следующий способ хранения информации о дереве в памяти ЭВМ. Рассмотрим массив P, состоящий из n ячеек, изначально заполненный нулями. Тогда, записав в ячейку $P[i]$ родителя узла с номером i, получим весьма компактный способ представления дерева. Массив P принято называть *массивом предков* данного корневого дерева.

Следует отметить, что рассмотренный способ хранения дерева обладает рядом недостатков, одним из которых является отсутствие эффективного способа просмотра узлов дерева, начиная с корня. Для того чтобы избавиться от данного недостатка, дерево можно хранить в виде списков смежности узлов, где под *списком смежности* узла v будем понимать линейный список узлов, непосредственно связанных с узлом v. Тогда процедура добавления связи между узлами u и v может выглядеть следующим образом (процедура 2.2.18):

Процедура 2.2.18. Tree Data Type: Adjacency List, Insert

```
01:  Insert Impl(T, u, v)
02:        Insert Edge Impl(T, u, v)
03:        Insert Edge Impl(T, v, u)
04:  Insert Edge Impl(T, u, v)
05:        with T do
06:              Insert(adjacencyLists[u], v)
```

Исходя из этого, процедура просмотра узлов дерева из заданной вершины (необязательно корня) не представляет сложностей (процедура 2.2.19):

Процедура 2.2.19. Tree Data Type: Adjacency List, Process

```
01:  Process(T, root)
02:        Process Impl(T, root, ∅)
03:
04:  Process Impl(T, currNode, prevNode)
05:        {* process current node *}
06:        with T do
07:              foreach nextNode in adjacencyLists[currNode] do
08:                    if nextNode ≠ prevNode then
09:                          Process Impl(T, nextNode, currNode)
```

Описанный выше подход для хранения дерева обладает рядом достоинств, основным из которых является эффективный способ доступа к узлам, непосредственно связанным с заданным. Таким образом, трудоемкость процедуры (2.2.19) составляет $O(n)$, где n – число вершин в дереве. В то же самое время списки смежности не позволяют эффективно определять, связаны ли две вершины непосредственно между собой (для этого в худшем случае может потребоваться $O(n)$ операций). Для эффективного выполнения такого рода запросов можно воспользоваться *матрицами смежности* размерностью $n \times n$, где в ячейке (i, j) матрицы будет записана 1, если вершины i и j связаны между собой и 0 в противном случае. Тогда запрос о наличии связи между вершинами можно выполнять за $O(1)$. Недостатком данного подхода является большое число операций, пропорциональное $O(n^2)$, требуемое для обхода дерева.

Бинарным деревом назовем упорядоченное дерево, у которого каждый из узлов содержит не более двух детей. Таким образом, у бинарного дерева обычно принято выделять левое поддерево и правое поддерево или, с точки зрения узлов, левого сына и правого сына. Согласно данному определению, описание бинарного дерева может выглядеть следующим образом (процедура 2.2.20):

Процедура 2.2.20. Binary Tree Data Type

```
01:  binaryTreeNode<T> {
02:        item:  T
03:        left:  binaryTree<T>
04:        right: binaryTree<T>
05:  }
06:  binaryTree<T> {
07:        root: binaryTreeNode<T>
08:  }
```

Как и для общих деревьев, массивы весьма удачно подходят для хранения бинарных деревьев с фиксированным числом вершин. Пусть, как и ранее, узлы дерева занумерованы натуральными числами от 1 до n. Рассмотрим массив B, состоящий из n ячеек. Информацию о корне дерева запишем в ячейку $B[1]$. Пусть информация о некотором узле k записана в ячейку $B[i]$. Тогда информация о левом ребенке узла k будет записана в ячейку $B[2 * i]$, а информация о

правом ребенке – в ячейку $B[2 * i + 1]$. В этом случае родителем узла, хранящегося в ячейке $B[i]$, будет узел $B[\lfloor i/2 \rfloor]$. Тогда процедура обработки узлов дерева может выглядеть следующим образом (процедура 2.2.21):

Процедура 2.2.21. Binary Tree Data Type: Process

```
01:  Process Impl(B)
02:        return Process(B, 1)
03:
04:        Process Impl(B, root)
05:              {* process root *}
06:              for i = 0 to 1 do
07:                    if B[2 * root + i] ≠ ∅ then
08:                          Process(B, 2 * root + i)
```

Одним из свойств бинарных деревьев, которое позволило получить им широкое распространение, является свойство, связывающее высоту бинарного дерева h с количеством узлов дерева n:

$$\log_2(n+1) - 1 \leq h \leq n - 1.$$

Будем говорить, что бинарное дерево является *сбалансированным*, если его высота h есть $O(\log_2 n)$, уровни от 0 до $h - 1$ заполнены полностью, а уровень h заполнен элементами слева направо.

Определение 2.6. *M-арным* назовем дерево, у которого каждый из узлов, кроме листьев, содержит не более M детей. Если же это количество равно M, то *M-арное* дерево будем называть полным.

Специальным случаем M-арного дерева является бинарное дерево, у которого $M = 2$. Несложно получить формулу для количества узлов полного M-арного дерева высотой h:

$$n = \frac{M^{h+1} - 1}{M - 1}.$$

Обход дерева

Существует несколько способов обхода упорядоченного дерева, т. е. способов просмотра его узлов. К основным из них относятся *прямой*, *обратный* и *симметричный* обход. Для того чтобы описать правила, которых придерживается каждый из способов просмотра узлов дерева, будем считать, что задано дерево T с корнем n и поддеревьями $T_1, \dots, T_k$. Тогда:

1) при *прямом обходе* сначала посещается корень n дерева T, далее посещаются в прямом порядке поддеревья $T_1, \dots, T_k$;

2) при *обратном обходе* сначала посещаются в обратном порядке поддеревья $T_1, \dots, T_k$, последним посещается корень n дерева T;

3) при *симметричном обходе* сначала посещается в симметричном порядке поддерево T_1, затем посещается корень n дерева T, далее посещаются в симметричном порядке узлы поддеревьев $T_2, \dots, T_k$.

Процедуры, моделирующие каждый из описанных способов обхода дерева, приведены ниже (процедура 2.2.22):

Процедура 2.2.22. Tree Data Type: PreOrder, PostOrder, InOrder

```
01:  PreOrder Impl(T)
02:        if not Is Empty(T) then
03:              process(Root(T))
04:              for i = 1 to children count do
05:                    PreOrder(T_i)
06:
07:  PostOrder Impl(T)
08:        if not Is Empty(T) then
09:              for i = 1 to children count do
10:                    PostOrder(T_i)
11:              process(Root(T))
12:
13:  InOrder Impl(T)
14:        if not Is Empty(T) then
15:              InOrder(T_1)
16:              process(Root(T))
17:              for i = 2 to children count do
18:                    InOrder(T_i)
```

Следует отметить, что каждый из приведенных обходов дерева может быть реализован в виде нерекурсивной процедуры путем использования АТД «Стек». Что же касается использования этих процедур, то каждая из них получила свое конкретное применение в областях информатики (в частности, в теории игр, динамическом программировании, в теории автоматов и языков и так далее). Трудоемкость каждой из процедур есть $O(|T|)$.

Одним из обходов дерева, который отличается от описанных выше, является обход *в ширину*. В данном случае узлы дерева просматриваются по уровням. Для эффективной реализации обхода в ширину используется АТД «Очередь» (процедура 2.2.23):

Процедура 2.2.23. Tree Data Type: Breadth First Search

```
01:  BFS(T)
02:        queue q = Enqueue(∅, Root(T))
03:        while not Is Empty(q) do
04:              node = Peek(q), q = Dequeue(q)
05:              process(node)
06:              for i = 1 to children count do
07:                    q = Enqueue(q, Root(T_i))
```

Так как каждый из узлов дерева помещается в очередь один раз (в силу свойств дерева), то трудоемкость приведенной процедуры составит $O(|T|)$, где $|T|$ – число узлов в дереве T.

Помимо обходов, которые были рассмотрены выше, существует ряд других алгоритмов, которые позволяют эффективно просматривать узлы дерева, причем не только деревьев, чья структура известна заранее, а также и деревьев, которые динамически расширяются в процессе обхода. К одной из таких техник относится техника итеративного погружения, получившая широкое распространение в области искусственного интеллекта.

Преобразование произвольного дерева в бинарное дерево

Зачастую работа с произвольными деревьями затруднительна по причине того, что требуется хранить список детей для каждого из узлов дерева. В то же самое время существует широкий класс задач на бинарных деревьях, для которых разработаны эффективные алгоритмы их решения. Эти алгоритмы не всегда могут быть приспособлены для решения обобщенных версий задач на произвольных деревьях. В связи с этим возникает потребность в разработке алгоритмов, которые позволят свести задачу на произвольных деревьях к задачам на бинарных деревьях. Одним из них является алгоритм преобразования произвольного дерева в соответствующее ему бинарное.

Между произвольными упорядоченными деревьями и бинарными деревьями существует взаимно-однозначное соответствие. Для того чтобы построить такое соответствие достаточно произвольное дерево T рассмотреть с точки зрения «левый сын – правый брат».

Определение 2.7. Пусть T – произвольное дерево с узлами $\{n_1, \dots, n_k\}$, а B – некоторое бинарное дерево с узлами $\{n'_1, \dots, n'_k\}$. Тогда дерево B является взаимно-однозначным *соответствием* дерева T, если:

1) узел n_i дерева T соответствует узлу n'_i дерева B, причем метки узлов n_i и n'_i являются одинаковыми;

2) узел n_i есть корень дерева T, то узел n'_i – корень дерева B;

3) узел n_j есть самый левый сын узла n_i, то n'_j – левый сын узла n'_i; если у узла n_i нет детей, то у узла n'_i нет левого сына;

4) узел n_j есть правый брат узла n_i, то n'_j – правый сын узла n'_i.

Пример произвольного дерева и соответствующего ему бинарного дерева приведены на рис. 2.2.

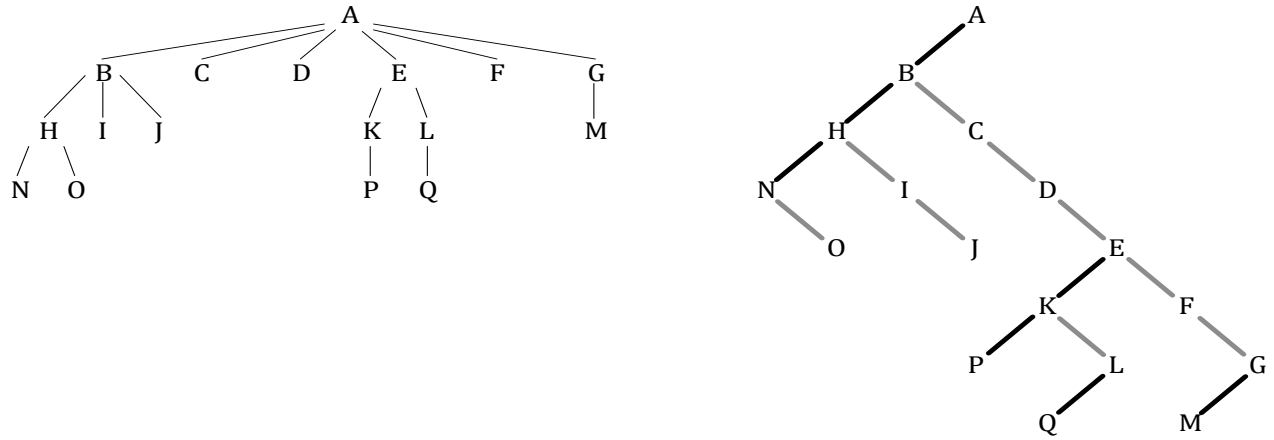

Рис. 2.2. Пример преобразования произвольного дерева в бинарное дерево

Если предположить, что некоторое произвольное дерево T описано при помощи типа данных (2.2.17), то алгоритм преобразования дерева T в бинарное дерево может выглядеть следующим образом (процедура 2.2.24):

```
Процедура 2.2.24. Tree Data Type: Build Binary Tree by Arbitrary Tree
01:  Build Binary Tree By(T)
02:         return Build Impl(T, ∅)
```

```
03:  Build Impl(T, rightTree)
04:        leftTree = ∅
05:        for i = children count downto 1 do
06:              leftTree = Build Impl(children[i], leftTree)
07:        return binaryTree {
08:              root = binaryTreeNode {
09:                    item  = Item(Root(T))
10:                    left  = leftTree
11:                    right = rightTree
12:              }
13:        }
```

Следует отметить, что существуют и другие алгоритмы преобразования произвольных деревьев в бинарные деревья, большинство из которых базируется на определение 2.7. Изменения же обычно продиктованы спецификой задачи (например, очень часто бывает полезным введение дополнительных узлов с тем, чтобы избежать связей между братьями).

Наименьший общий предок двух вершин в дереве

Одной из наиболее часто возникающих задач на деревьях является задача определения наименьшего общего предка для двух заданных вершин.

Определение 2.8. *Общим предком* узлов u и v в корневом дереве T будем называть такой узел p, который лежит на пути от корня дерева T как в узел u, так и в узел v. *Наименьшим общим предком* узлов u и v в корневом дереве T будем называть такого общего предка p узлов u и v, который наиболее удален от корня дерева T[*].

Наиболее простым алгоритмом нахождения LCA двух вершин в дереве является алгоритм с трудоемкостью $\langle O(n), O(h)\rangle$. В этом случае алгоритму предварительно требуется обойти дерево за $O(n)$ и собрать информацию о глубине узлов (т. е. об уровнях, на которых узлы размещены) и об их родителях. Непосредственно фаза запроса LCA будет иметь трудоемкость $O(h)$, где h – высота дерева (процедура 2.2.25):

Процедура 2.2.25: Tree: LCA

```
01:  LCA Query Impl(v1, v2)
02:        h1 = Depth(v1)
03:        h2 = Depth(v2)
04:        while h1 ≠ h2 do
05:              if h1 > h2 then
06:                    v1 = Parent(v1)
07:                    h1 = h1 - 1
08:              else
09:                    v2 = Parent(v2)
10:                    h2 = h2 - 1
11:        while v1 ≠ v2 do
12:              v1 = Parent(v1)
13:              v2 = Parent(v2)
14:        return v1
```

[*] В англоязычной литературе используется термин LCA от *least common ancestor*.

Внимательно проанализировав приведенный алгоритм, заметим, что подъем из вершин происходит с шагом 1. Если попытаться на каждом шаге подниматься на большее число уровней, то можно улучшить общее время выполнения алгоритма.

Рассмотрим корневое дерево T, вершины которого занумерованы числами от 1 до n. Обозначим через $P[i,k]$ $(1 \le i \le n, 0 \le k \le \lfloor \log_2 n \rfloor)$ предка вершины i, который удален от нее на 2^k уровней (в случае, если уровень вершины i меньше, чем 2^k, то соответствующее значение $P[i,k]$ будет равно корню дерева T). Тогда построение массива P можно выполнить за $O(n \log n)$ шагов, используя следующее рекуррентное соотношение:

$$\begin{cases} P[i,k] = parent(i), \ k = 0, \\ P[i,k] = P[P[i,k-1], k-1], \ k > 0. \end{cases} \tag{2.2}$$

Графическое пояснение приведенного рекуррентного соотношения приведено на рис. 2.3:

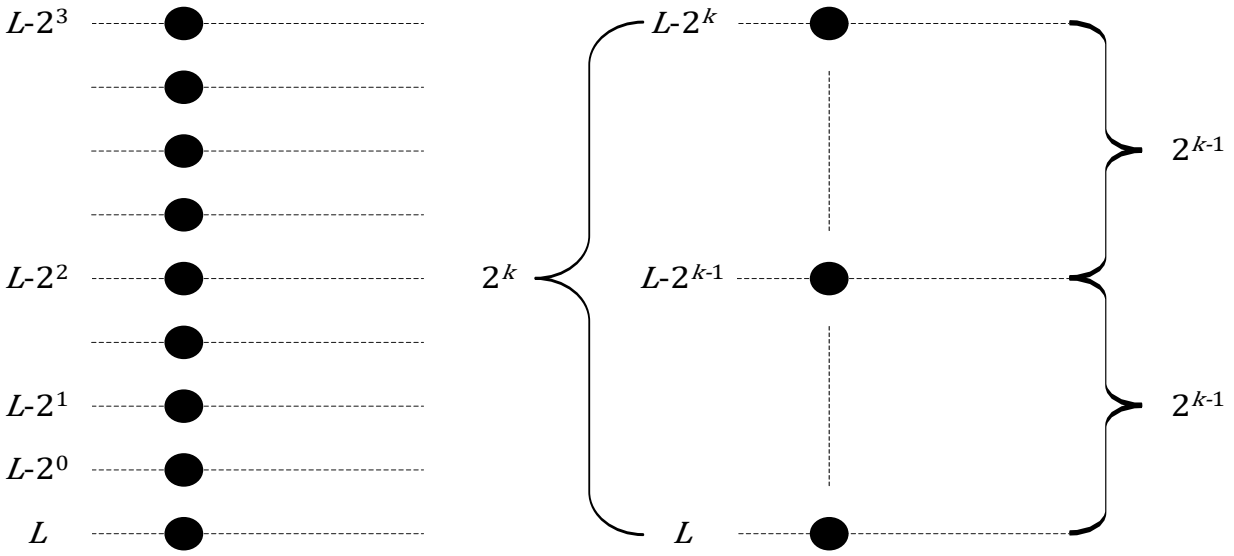

Рис. 2.3. К рекуррентному соотношению 2.1

Процедура заполнения массива P требуемыми значениями приведена ниже (2.2.26):

Процедура 2.2.26: Tree: LCA, PreProcess

```
01:  LCA PreProcess Impl(T)
02:        for i = 1 to n do
03:              P[i, 0] = Parent(i)
04:        for k = 1 to ⌊log2 n⌋ do
05:              for i = 1 to n do
06:                    P[i, k] = P[P[i, k – 1], k – 1]
```

Для того чтобы воспользоваться информацией, хранящейся в массиве P, необходимо иметь процедуру, которая будет для двух вершин проверять, является ли одна из них предком (необязательно непосредственным) другой. Существуют методы, базирующиеся на поиске в глубину, которые позволяют осуще-

ствить требуемую проверку за $O(1)$ [4]. Тогда запрос на LCA двух вершин можно реализовать за $O(\log n)$ шагов следующим образом (процедура 2.2.27):

Процедура 2.2.27: Tree: LCA

```
01:  LCA Query Impl(v1, v2)
02:        if Is Ancestor(v1, v2) then return v1
03:        if Is Ancestor(v2, v1) then return v2
04:        for k = ⌊log2 n⌋ downto 0 do
05:              if not Is Ancestor(P[v1, k], v2) then
06:                    v1 = P[v1, k]
07:        return Parent(v1)
```

Таким образом, в начале алгоритм находит наименее удаленный от корня узел, который не является предком вершины v_2, а затем возвращает родителя найденного узла в качестве LCA узлов v_1 и v_2.Трудоемкость рассмотренного выше алгоритма составляет $\langle O(n \log n), O(\log n) \rangle$. Рассмотренный выше способ нахождения LCA двух вершин носит название *техники двоичного подъема*. С другой стороны, данную технику можно рассматривать с точки зрения более общей *техники масштабирования*. Следует отметить, что существуют и другие техники нахождения LCA двух узлов в заданном корневом дереве, основные из которых рассмотрены в источнике [5].

АТД «Дерево»

Выше были рассмотрены базовые структуры данных для представления деревьев как произвольных, так и бинарных, в памяти ЭВМ. Помимо структур данных следует выделить понятие абстрактного типа данных «Дерево», на котором определен ряд операций, основными из которых являются добавление и удаление узла в дерево, поиск заданного значения в дереве, определение отношений между узлами (например, являются ли узлы братьями, является ли узел предком либо потомком другого узла и т. д.), а также операции по созданию дерева по заданной коллекции данных. Для работы с АТД «Дерево» может быть выбрана одна из структур данных, рассмотренных выше. Нередко подходы могут быть скомбинированы (процедура 2.2.28):

Процедура 2.2.28: ADT Tree

```
01:  treeDataType {
02:        parent: treeNode<T>[max nodes count]
03:        tree: tree<T>
04:  }
```

Отдельно стоит выделить АТД «Бинарное дерево» как наиболее распространенный среди всех остальных типов деревьев. Одной из областей применения АТД «Бинарное дерево» является область его использования при реализации бинарных деревьев поиска (будут рассмотрены в разд. 4.2), а также других более сложных АТД, таких, как «Множество».

Независимо от того, какой из типов деревьев используется для хранения коллекции данных, будем предполагать, что АТД «Дерево» принадлежит к классу нелинейных контейнеров. При этом хранимые элементы могут быть

упорядочены и записаны в линейный контейнер путем использования операции обхода дерева. Если же АТД «Дерево» используется для хранения пар «ключ – значение ключа», то будем предполагать, что данный АТД принадлежит к классу ассоциативных контейнеров.

2.3. Множества

Под *множеством* будем понимать некоторую совокупность *элементов*, каждый из которых является либо множеством, либо примитивным элементом, называемым *атомом*. В этом случае принято говорить, что атомы множества – это его неделимые элементы, в то время как элементы, отличные от атомарных, могут быть представлены как совокупность некоторого числа атомов (возможно, из другого множества). В дальнейшем рассмотрению будут подлежать конечные множества, т. е. те, число элементов в которых выражается натуральным числом либо нулем. Также будем предполагать, что элементы множества являются атомарными, например, множество чисел, строк, массивов, списков и т. д. (если не оговорено другое).

Будем говорить, что некоторое множество S является *линейно упорядоченным* (или просто *упорядоченным*), если на нем задано отношение линейного порядка. Основными операциями над множествами являются операции создания множества, вставки / удаления элементов, поиска, объединения и пересечения и др. Будем говорить, что некоторый АТД является «Множеством», если на нем поддерживаются операции, определенные на математической модели множества.

В зависимости от структуры элементов множества выбирается та либо иная структура данных для хранения его элементов в памяти ЭВМ. Так, например, цепочки символов (строки, последовательности бит и т. д.) обычно принято хранить в нагруженном дереве[*], в то время как числовые множества могут быть эффективно представлены как с помощью списковых АТД, так и с помощью деревьев. Следует отметить, что на выбор структуры данных, посредством которой будет реализован АТД «Множество», влияет не только природа обрабатываемых элементов, но также и то, является множеством статическим или динамическим. Ниже будут рассмотрены основные линейные способы представления множеств, а также специальный способ хранения тех, элементами которых являются множества.

2.3.1. Линейные способы представления множеств

К линейным способам представления множеств относятся массивы, линейные структуры данных, линейные АТД и битовые маски. Выбор того либо иного способа представления информации зависит от специфики задачи и от того, задано ли на множестве элементов отношение порядка. Так, массивы и линейные АТД, вместе с соответствующими структурами данных, обычно ис-

[*] Помимо термина *нагруженное дерево* используется термин *бор*. В англоязычной литературе используется термин *trie*.

пользуются для хранения упорядоченных множеств, в то время как битовые маски применяются в тех случаях, когда число элементов достаточно невелико (скажем, не превышает 20). Так как использование массивов для хранения множеств является достаточно очевидным, рассмотрим ниже способы, основанные на линейных списках и битовых масках.

Линейные списки

Будем считать, что на некотором множестве A задано отношение линейного порядка, в соответствие с которым элементы из A могут быть упорядочены. Тогда для представления подмножеств множества A можно воспользоваться как связными списками, так и линейными АТД, в которых элементы будут храниться в порядке $a_1, a_2, \ldots, a_n$, причем $a_1 \preccurlyeq a_2 \preccurlyeq \cdots \preccurlyeq a_n$. Тогда операции объединения и пересечения двух множеств можно реализовать следующим образом (процедура 2.3.1):

Процедура 2.3.1: ADT Set: Union, Intersect

```
01:  Union Impl(L₁, L₂)
02:        L = ∅
03:        while not Is Empty(L₁) and not Is Empty(L₂) do
04:              if First(L₁) < First(L₂) then
05:                    L₁ = Move(L₁)
06:              else
07:                    L₂ = Move(L₂)
08:        while not Is Empty(L₁) do L₁ = Move(L₁)
09:        while not Is Empty(L₂) do L₂ = Move(L₂)
10:        return L
11:
12:        Move Impl(S)
13:              L = Insert(L, First(S))
14:              return Next(S)
15:
16:  Intersect Impl(L₁, L₂)
17:        L = ∅
18:        while not Is Empty(L₁) and not Is Empty(L₂) do
19:              if First(L₁) = First(L₂) then
20:                    L = Insert(L, First(L₁))
21:                    L₁ = Next(L₁)
22:                    L₂ = Next(L₂)
23:              if First(L₁) < First(L₂) then
24:                    L₁ = Next(L₁)
25:              if First(L₂) < First(L₁) then
26:                    L₂ = Next(L₂)
27:        return L
```

Трудоемкость приведенных выше операций есть $O(n_1 + n_2)$, где n_1 и n_2 – число элементов в множествах, которые подлежат обработке. Процедура же поиска во множестве, реализованном посредством линейного списка, будет линейна относительно числа его элементов, а вставка элемента в требуемую позицию будет иметь трудоемкость $O(1)$. Если же воспользоваться упорядочен-

ными массивами для представления множеств, то поиск элементов можно осуществлять за $O(\log n)$ операций, в то время как операция вставки будет иметь трудоемкость $O(n)$.

Битовые маски

Рассмотрим множество A, объекты которого могут быть занумерованы целыми числами от 0 до $n-1$. Если число элементов n во множестве A сопоставимо с размером машинного слова, то можно предложить следующий способ хранения подмножеств множества A в памяти ЭВМ. Для этого достаточно зарезервировать переменную целочисленного типа, в которой бит i будет соответствовать элементу a_i из A. Отождествив элементы множества A с их номерами, можно предложить следующую реализацию операций по созданию множеств (процедура 2.3.2):

Процедура 2.3.2: ADT Set: Make Set

```
01:  Make Set Impl(a)
02:        return 1 shl a
03:
04:  Make Set Impl(A)
05:        set = 0
06:        foreach a in A do
07:              set = Include(set, Make Set(a))
08:        return set
09:
10:  Make Universe Impl(n)
11:        return (1 shl n) - 1
```

Помимо операции создания наиболее востребованными являются операции включения / исключения, подсчета числа элементов, получения всех подмножеств данного множества и др. (процедура 2.3.3):

Процедура 2.3.3: ADT Set: Contains, Cardinality, Enumerate

```
01:  Contains Impl(set, subset)
02:        if set and subset = subset then
03:              return true
04:        return false
05:
06:  Next Impl(set, subset)
07:        return (subset – 1) and set
08:
09:  Cardinality Impl(set) {
10:        if set = 0 then
11:              return 0
12:        return 1 + Cardinality(Next(set, set))
13:
14:  Enumerate Impl(set)
15:        subset = set
16:        while subset > 0 do
17:              yield return subset
18:              subset = Next(set, subset)
19:        yield return 0
```

Следует отметить, что описанные выше операции являются достаточно эффективными, поскольку большинство их них выполняется за $O(1)$, кроме операции подсчета числа элементов и перечисления подмножеств. Обе операции имеют трудоемкость $O(b)$ и $O(2^b)$ соответственно, где b – число установленных бит во множестве.

Помимо данных операций для работы с битовыми масками, т. е. с множествами, закодированными посредством бит, на практике часто возникает потребность в обработке всех множеств из заданного универсального множества вместе с их подмножествами (процедура 2.3.4):

Процедура 2.3.4: ADT Set: Process Universe

```
01:  Process Universe Impl(n)
02:        universe = Make Universe(n)
03:        for set = 0 to universe do
04:              subset = set
05:              while subset > 0 do
06:                    {* process set and subset *}
07:                    subset = Next(set, subset)
```

Для того чтобы оценить время работы процедуры 2.3.4, достаточно заметить, что подмножества из k бит могут быть выбраны C_n^k способами и у каждого из выбранных множеств будет 2^k подмножеств. Следовательно, полное число подмножеств, которое будет обработано, равно $\sum_{k=0}^{n} C_n^k \cdot 2^k = (1+2)^n$. Отсюда можно заключить, что трудоемкость приведенной процедуры есть $O(3^n)$.

Если же число элементов множества A значительно превосходит размер машинного слова, а специфика задачи требует компактно представлять подмножества A, то можно воспользоваться целочисленным массивом, каждый элемент которого будет представлять собой некоторое подмножество. Тогда, для того, чтобы обработать элемент a_i необходимо и достаточно узнать: а) сегмент – соответствующую ячейку массива, в которой элемент a_i будет храниться; б) смещение – номер соответствующего бита, который кодирует элемент с номером i (процедура 2.3.5):

Процедура 2.3.5: ADT Set: Segment, Offset

```
01:  Segment Impl(i)
02:        return i div 32
03:
04:  Offset Impl(i)
05:        return i mod 32
```

Вышеприведенные процедуры исходят из того факта, что размер машинного слова равен 32 бита. Если же размер машинного слова отличен от 32, то соответствующая константа должна быть изменена (например, 64). В заключение, авторы рекомендуют обратиться к литературному источнику [6] как к незаменимому справочнику по манипулированию с битами.

2.3.2. Системы непересекающихся множеств

Рассмотрим совокупность множеств $S = \{A_1, \dots, A_n\}$. Будем говорить, что совокупность множеств S образует *систему непересекающихся множеств (СНМ)*, если $A_i \cap A_j = \emptyset$ для всех $1 \le i, j \le n, i \ne j$.[*]

В ряде случаев удобно считать, что множества A_i разбивают множество S на n классов эквивалентности, т. е. $S = A_1 \cup \dots \cup A_n$. У каждого из классов A_i обычно выделяется представитель $a_i \in A_i$, который идентифицирует данный класс. Представителем множества A_i может быть любой элемент, так как все элементы, образующие A_i, считаются эквивалентными с точки зрения множества S. Тогда под АТД «Система непересекающихся множеств» будем понимать нелинейный АТД, предназначенный для хранения СНМ и поддерживающий следующий набор операций:

1) создание одноэлементного множества;

2) объединение множеств;

3) поиск множества, которому принадлежит заданный элемент.

В дальнейшем изложении условимся, что множество S состоит из n объектов, занумерованных от 1 до n, а каждое из множеств A_i – подмножество S. Будем считать, что операции объединения и поиска работают не с самими объектами, а с отождествленными на них номерами. При этом операция поиска возвращает номер представителя множества, которому принадлежит заданный объект.

Основными структурами данных, которые используются для хранения информации о системе непересекающихся множества, являются массивы, линейные списки и деревья. Для того чтобы оценить эффективность структуры данных, используемой для хранения СНМ, будем предполагать, что на АТД выполняется набор из m операций создания, поиска и объединения.

Реализация на основе массива

Реализация СНМ через массив заключается в том, чтобы для каждого объекта сохранить в массив номер представителя множества, которому этот объект принадлежит (процедура 2.3.6):

Процедура 2.3.6: ADT DSS: Array

```
01:  Make Set Impl(D, a)
02:        with D do
03:              leader[a] = a
04:
05:  Find Set Impl(D, a)
06:        with D do
07:              return leader[a]
08:  Union Impl(D, a, b)
09:        a = Find(D, a)
10:        b = Find(D, b)
11:        if a ≠ b then
12:              Link(D, a, b)
```

[*] В англоязычной литературе используется термин *disjoint sets*.

```
13:         Link Impl(D, a, b)
14:               with D do
15:                     for i = 1 to n do
16:                           if leader[i] = b then
18:                                 leader[i] = a
```

В полученной реализации СНМ операция поиска работает за $O(1)$, в то время как операция объединения множеств имеет трудоемкость $O(n)$. В силу того, что операцию объединения можно выполнить не более $n-1$ раз, можно заключить, что любая последовательность из m операций будет выполняться за время, не превосходящее $O(n^2+m)$. Следовательно, реализация СНМ через массив крайне неэффективна в случае достаточно большого числа объектов. Поэтому данный подход применяется при работе с множествами, число элементов в которых невелико.

Реализация на основе списка

Оценку $O(n^2+m)$ можно значительно улучшить, если вместо массива использовать линейные списки для хранения элементов каждого из подмножеств (процедура 2.3.7):

Процедура 2.3.7: ADT DSS: List

```
01:  Make Set Impl(D, a)
02:         with D do
03:               leader[a] = a
04:               Insert(La, a)
05:
06:  Find Set Impl(D, a)
07:         with D do
08:               return leader[a]
09:
10:  Union Impl(D, a, b)
11:         a = Find(D, a)
12:         b = Find(D, b)
13:         if a ≠ b then
14:               Link(D, a, b)
15:
16:         Link Impl(D, a, b)
17:               with D do
18:                     if |La| ≥ |Lb| then
19:                           foreach x in Lb do
20:                                 leader[x] = a
21:                           Insert(La, Lb)
22:                     else Link(D, b, a)
```

В полученной реализации СНМ операция поиска работает за $O(1)$, в то время как операция объединения множеств имеет трудоемкость $O(\min\{|A_i|,|A_j|\})$, где A_i и A_j – объединяемые множества. В худшем случае операция объединения имеет трудоемкость $O(n)$. Несмотря на кажущуюся неэффективность операции объединения, трудоемкость выполнения любых m

операций не будет превосходить $O(n \log n + m)$. Действительно, как можно видеть из реализации операции объединения, ключевой идеей является присоединение меньшего множества к большему. Следовательно, каждый элемент после операции объединения переходит во множество, в котором будет по крайней мере в два раза больше элементов. Отсюда можно сделать вывод о том, что каждый из элементов сделает не более $O(\log n)$ переходов. Суммарное же число переходов для n элементов составит $O(n \log n)$, что приводит к оценке $O(n \log n + m)$ для m операций.

Полученная реализация данного АТД является достаточно эффективной как с точки зрения выполнения операций поиска и объединения, так и с точки зрения возможности перечисления элементов, которые составляют каждое из подмножеств. Для того чтобы перечислить элементы подмножества A_i достаточно совершить $O(|A_i|)$ операций, что лучше чем $O(n)$.

Реализация на основе леса

Наиболее эффективная реализация СНМ получается в том случае, если хранить элементы каждого множества в виде дерева. Тогда вся система непересекающихся множеств будет представлять собой набор деревьев – *лес*. Для хранения леса будем использовать массив предков P. Представителем множества в данной реализации СНМ будем считать корень дерева, которое содержит элементы этого множества.

Для реализации операции поиска необходимо подняться от объекта до корня дерева, в котором он находится, и возвратить найденное значение. Для реализации операции объединения необходимо провести ребро от корня одного из объединяемых деревьев к корню другого (процедура 2.3.8):

Процедура 2.3.8: ADT DSS: Forest Base Impl

```
01:  Make Set Impl(D, a)
02:        with D do
03:              parent[a] = a
04:
05:  Find Set Impl(D, a)
06:        with D do
07:              if a ≠ parent[a] then
08:                    return Find Set(D, parent[a])
09:              return a
10:
11:  Union Impl(D, a, b)
12:        a = Find(D, a)
13:        b = Find(D, b)
14:        if a ≠ b then
15:              Link(D, a, b)
16:
17:        Link Impl(D, a, b)
18:              with D do
19:                    parent[b] = a
```

Если объединять деревья случайным образом, как это сделано в процедуре 2.3.6, то пользы от рассматриваемой реализации СНМ не будет, – деревья могут выродиться в цепочки и операция поиска будет работать за линейное (относительно числа элементов во множестве) время. Для получения результата следует минимизировать высоту получаемого при объединении дерева. Для этого применяется следующая эвристика, которая носит название *объединение по рангам*. Рассмотрим массив R, состоящий из n элементов. Значение в ячейке $R[i]$ будем интерпретировать как оценку сверху для высоты дерева с корнем в узле i. Тогда операцию объединения можно проводить следующими способами: 1) если ранги объединяемых деревьев не равны, то проводится ребро от дерева с меньшим рангом к дереву с большим рангом; 2) если ранги равны, то проводится ребро произвольным образом, и при этом увеличивается на единицу ранг корня дерева, к которому было проведено ребро. Таким образом, на данном этапе реализации $R[i]$ есть высота дерева с корнем в вершине i.

Объединение по рангам приводит к таким же оценкам производительности, как и рассмотренная выше реализация СНМ через списки. Для того чтобы добиться еще большего выигрыша, рассмотрим эвристику, которая получила название *сжатие путей*. Идея эвристики сжатия путей заключается в том, чтобы у всех узлов, которые были рассмотрены на пути от объекта до корня дерева, содержащего этот объект, установить родителем корень дерева. Это позволит ускорить работу последующих вызовов операции поиска для этих объектов. Иллюстрация эвристики сжатия путей приведена на рис. 2.4. Реализация операций над системой непересекающихся множеств, с учетом рассмотренных эвристик, приведена в процедуре 2.3.9:

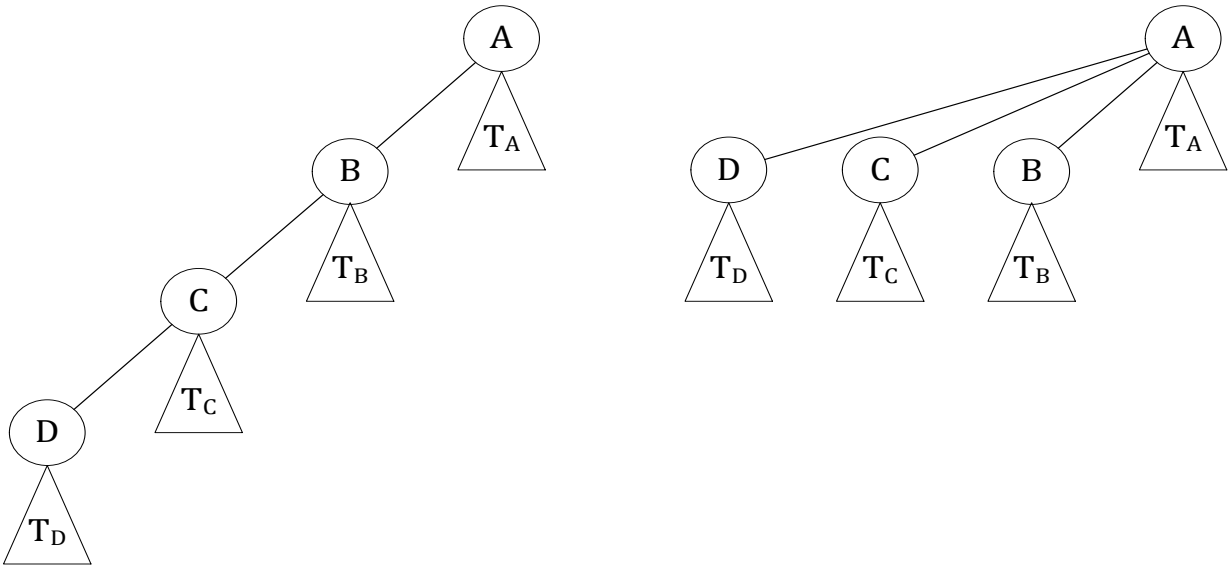

Рис. 2.4. Сжатие путей

```
Процедура 2.3.9: ADT DSS: Forest
01:  Make Set Impl(D, a)
02:        with D do
03:              parent[a] = a
04:              rank[a] = 1
```

```
05:  Find Set Impl(D, a)
06:        with D do
07:              if a ≠ parent[a] then
08:                    parent[a] = Find Set(D, parent[a])
09:              return parent[a]
10:
11:  Union Impl(D, a, b)
12:        a = Find(D, a)
13:        b = Find(D, b)
14:        if a ≠ b then
15:              Link(D, a, b)
16:
17:        Link Impl(D, a, b)
18:              with D do
19:                    if rank[a] < rank[b] then
20:                          parent[a] = b
21:                    else
22:                          parent[b] = a
23:                          if rank[a] = rank[b] then
24:                                rank[a] = rank[a] + 1
```

В полученной реализации СНМ трудоемкость операции объединения есть $O(1)$. Что касается трудоемкости операции поиска, то в худшем случае она может оказаться равной $O(n)$. В случае последовательности из m операций, суммарное время работы не будет превосходить $O(m\log^* n)$, где $\log^* n$ – это минимальное количество раз взятия двоичного логарифма от числа n для получения числа, которое меньше единицы. Следует отметить, что $\log^* n$ – очень медленно растущая величина. В частности, $\log^* n \leq 5$ для всех $n < 2^{65536}$. Так что можно считать, что операция поиска в данной реализации СНМ выполняется в среднем за время, равное константе. Оценка $O(m\log^* n)$ не является точной и может быть улучшена. Доказано, что описанная реализация СНМ является самой эффективной из всех возможных [7, 8].

В заключение отметим, что альтернативной эвристикой объединения по рангам может служить *весовая эвристика*, идея которой базируется на том, чтобы дерево с меньшим числом узлов подвешивать к корню дерева с большим числом узлов.

2.3.3. Словари

Под *словарем* будем понимать специальный тип АТД «Множество», предназначенный для хранения множеств, состоящих из пар «ключ – значение ключа». Словари принято называть *отображениями* (множество ключей отображается на множество значений), либо *ассоциативными массивами* (множество значений ассоциируется с множеством ключей). К основным операциям над словарями относятся операции создания словаря, вставки / удаления элементов, поиска значения по ключу.

Основными структурами данных, которые используются для реализации АТД «Словарь» являются бинарные деревья поиска, слоеные списки и хеш-таблицы. В соответствие с определением словаря, массивы можно отнести к словарным структурам данных, которые в качестве ключей используют целые числа. Реализация словаря посредством бинарных деревьев поиска может выглядеть следующим образом (процедура 2.3.10):

Процедура 2.3.10: ADT Dictionary

```
dictionary<K, V> {
      container: binarySearchTree<K, V>
}

Insert Impl(D, key, value)
      with D do
            Insert(container, key, value)

Delete Impl(D, key)
      with D do
            Delete(container, key)

Look Up Impl(D, key)
      with D do
            return Search(container, key)
```

Таким образом, трудоемкость операций вставки, удаления и поиска непосредственно зависит от трудоемкости выполнения соответствующих операций над структурой данных, которая лежит в основе словаря. Отметим, что при использовании хеш-таблиц множество ключей не обязательно линейно упорядоченное множество (т. е. такое, когда элементы можно сравнить между собой), в то время как для большинства древовидных структур данных это условие является необходимым.

Словарные структуры данных находят широкое использование на практике. В качестве примера может служить применение словарных структур данных к так называемой технике *мемоизации*[*] в контексте задач, решаемых при помощи метода динамического программирования.

Рассмотрим некоторую абстрактную вычислительную задачу P, которая может быть решена методом динамического программирования, путем разбиения исходной задачи P на более мелкие подзадачи $P_1, P_2, \dots, P_{n_P}$, где n_P – число таких подзадач. Предположим, что каждой задаче P соответствует некоторое состояние S, которое описывает задачу P. Также выделим ряд *элементарных задач*, для которых решение известно. Состояния, соответствующие элементарным задачам, назовем *начальными*. Связав все состояния в некоторый ациклический граф G, можно предложить следующий способ нахождения решения задачи P, которой соответствует состояние S (процедура 2.3.11):

[*] Термин *мемоизация* происходит от английского *memoization*.

```
Процедура 2.3.11: Memoization Technique
01:  Calculate Impl(S)
02:        if not Contains(cache, S) then
03:              result = ∅
04:              foreach Sᵢ in Adjacency List(G, S) do
05:                    accumulate(result, Calculate(Sᵢ))
06:              cache[S] = result
07:        return cache[S]
```

Трудоемкость данного алгоритма есть $O(V + E)$, где V – число состояний, а E – число переходов в графе G, при условии, что функция накопления результата работает за константное время.

Как можно видеть из приведенного алгоритма, в качестве *кеша* должна быть использована такая словарная структура данных, которая позволит эффективно осуществлять вставку и поиск элемента. Если состояния можно занумеровать натуральными числами, то в качестве кеша может быть использован массив. Перед непосредственным вызовом процедуры вычисления в словарь должны быть помещены элементарные состояния вместе с вычисленными для них значениями. Для того чтобы ответить на вопрос, почему данная техника дает ощутимый выигрыш, рассмотрим задачу определения количества путей, ведущих из вершины 1 в вершину n, на следующем графе G (рис. 2.5):

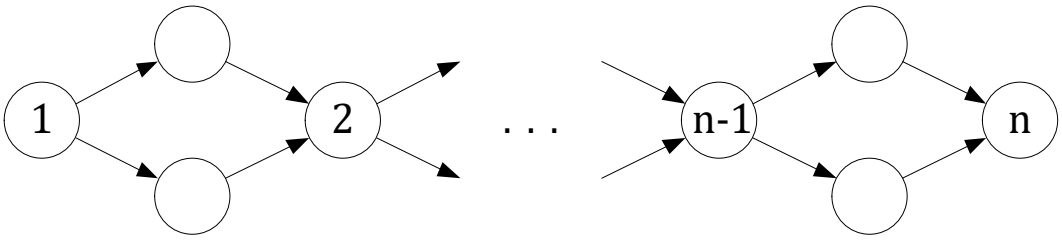

Рис. 2.5. Граф G

Если вычисление путей вести непосредственно, то количество шагов будет сравнимо с $O(2^n)$. Воспользовавшись техникой мемоизации, трудоемкость алгоритма можно значительно уменьшить. В этом случае достаточно запоминать количество путей для каждой из вершин на пути из 1 в n. Так как вычисления для каждой из вершин будут вестись ровно один раз, то трудоемкость алгоритма составит $O(n)$.

2.4. Очередь с приоритетами

Рассмотрим множества объектов A и B. Будем считать, что на множестве объектов B задано отношение частичного порядка «меньше либо равно», а элементы множества A занумерованы числами от 1 до n. *Очередь с приоритетом* для множества элементов из A и заданной функции $f: A \to B$ представляет собой АТД со следующими операциями:

1) добавление элемента;

2) удаление элемента с минимальным ключом.

Элементы множества B будем называть *приоритетами*. Соответственно, операция удаления элемента с минимальным ключом – это поиск и удаление

элемента, приоритет которого является наименьшим среди всех элементов, находящихся в очереди в данный момент.

Название «Очередь с приоритетами» обусловлено видом упорядочивания, которому подвергаются данные, хранящиеся посредством АТД. Термин «очередь» предполагает, что элементы ожидают некоторого обслуживания, а термин «приоритет» определяет последовательность обработки элементов. Необходимо отметить, что в отличие от обычных очередей, рассмотренных в разд. 2.2, АТД «Очередь с приоритетами» не придерживается принципа FIFO.

Реализация АТД «Очередь с приоритетами» может базироваться на основе массивов или списковых структур. Среди последних выбирают как связные списки, так и линейные АТД. Независимо от типа используемой линейной структуры данных, все реализации очереди с приоритетами можно условно разделить на два типа относительно того, хранятся элементы очереди в упорядоченном виде или нет. В случае упорядоченных последовательностей операцию поиска и удаления элемента с минимальным приоритетом можно выполнить за $O(1)$, а операцию вставки за $O(n)$. Если требуется удалить произвольный элемент из очереди, то на массиве потребуется $O(n)$ операций, в то время как на списковых структурах всего $O(1)$ при условии, что известна позиция, в которой находится удаляемый элемент. В случае неупорядоченных последовательностей операция вставки будет иметь трудоемкость $O(1)$, а операция поиска минимального элемента – $O(n)$. Трудоемкость выполнения операции удаления будет равна $O(n)$, если элементы хранятся в массиве, и $O(1)$, если в основе очереди лежит список.

Несложно видеть, что как линейные структуры данных, так и линейные АТД, не позволяют построить реализацию очереди с приоритетами, в которой бы все операции выполнялись быстрее, чем $O(n)$. В то же самое время такая реализация необходима, если число объектов, которые должны быть обработаны очередью, достаточно велико (скажем, $n \geq 1'000$). В таких случаях обычно применяются структуры данных, отличные от линейных, которые позволяют выполнять каждую из операций за $O(\log n)$. В большинстве своем эти структуры данных основаны на сбалансированных деревьях, из которых следует выделить бинарные кучи, биноминальные кучи, кучи Фибоначчи, деревья ван Эмде Боаса и др. Ниже будет рассмотрена реализация АТД «Очередь с приоритетами» на базе бинарной кучи, как наиболее распространенная среди остальных. Характеристика других типов деревьев выходит за рамки данного пособия [7].

Рассмотрим множество объектов A, на котором задано линейное отношение порядка, и бинарное дерево H, состоящее из $|A|$ узлов. Будем считать, что с каждым из узлов дерева H ассоциирован в точности один объект из множества A, который назовем *значением* узла. Тогда дерево H будем называть *бинарной кучей* (или *пирамидой*), если значение его узлов не превышает значений его потомков.

Бинарные кучи еще принято называть *частично-упорядоченными* деревьями, поскольку порядок задается только между родителями и детьми, в то время как порядок между братьями может быть выбран произвольно. Пример бинарной кучи для множества из 10 элементов {1,2,4,8,12,6,10,14,18,16} приведен на рис. 2.6:

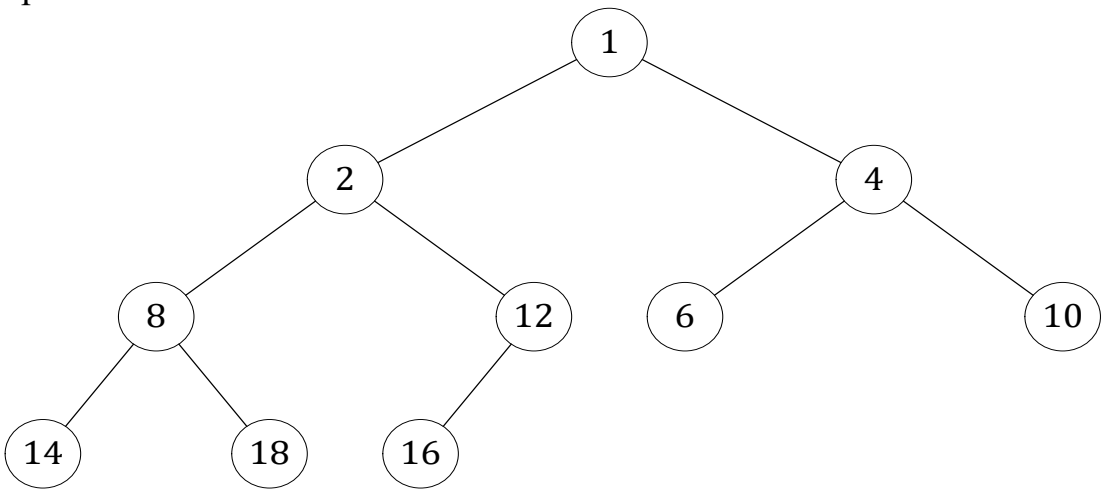

Рис. 2.6. Бинарная куча

Наиболее распространенным способом хранения пирамид в памяти ЭВМ является использование массивов. Согласно технике, описанной в разд. 2.2.5, применительно к бинарным деревьям можно предложить следующий способ хранения бинарной кучи (процедура 2.4.1):

Процедура 2.4.1: Heap

```
heapEntry<T> {
        item: T
        position: integral
}
heap<T> {
        data: heapEntry<T>[max elements count]
        elementsCount: integral
}
```

В дальнейшем изложении будем предполагать, что дерево H – это массив, для которого выполнено основное свойство бинарной кучи:

$$f(H[\lfloor i/2 \rfloor]) \leq f(H[i]),\ 1 < i \leq n, \tag{2.3}$$

где $f(H[i])$ – значение узла $H[i]$.

В частности, из условия (2.3) следует, что элемент с минимальным значением находится в корне дерева $H[1]$.

Процедуры нахождения детей и родителя данного узла могут выглядеть следующим образом (процедура 2.4.2):

Процедура 2.4.2: Heap: Parent, Left Child, Right Child

```
Parent Impl(H, entry)
        if 1 < entry.position ≤ |H| then
                return H[⌊entry.position / 2⌋]
        return ∅
```

```
05:  Left Child Impl(H, entry)
06:        if 2 * entry.position  ≤  |H| then
07:              return H[2 * entry.position]
08:        return ∅
09:  Right Child Impl(H, i)
10:        if 2 * entry.position + 1  ≤  |H| then
11:              return H[2 * entry.position + 1]
12:        return ∅
```

Основными операциями на бинарных кучах являются операция *проталкивания вверх* и операция *проталкивания вниз*. Суть каждой из операций заключается в том, чтобы поместить требуемый элемент на свое место в дереве с тем, чтобы сохранить основное свойство кучи. Таким образом, операция проталкивания вверх будет перемещать элемент до тех пор, пока не будет выполнено условие (2.3). Операция же проталкивания вниз будет перемещать элемент до тех пор, пока не будет выполнено условие

$$f(H[i]) \leq f(H[2 \cdot i]) \wedge f(H[i]) \leq f(H[2 \cdot i + 1]). \qquad (2.4)$$

В худшем случае каждой из операций может потребоваться пройти полный путь от самого глубокого листа до корня (в случае проталкивания вверх) либо наоборот (в случае проталкивания вниз), поэтому можно заключить, что каждая из них будет иметь трудоемкость $O(h)$, где h – высота дерева. Так как бинарная куча является сбалансированным деревом, то трудоемкость операций составит $O(\log n)$. Реализация операций проталкивания вверх и вниз приведена ниже (процедура 2.4.3):

Процедура 2.4.3: Heap: Swap, Push Up, Push Down

```
01:  Swap Impl(H, i, j)
02:        H[i].position = j
03:        H[j].position = i
04:        Swap(H[i], H[j])
05:
06:  Push Up Impl(H, entry)
07:        if entry.position > 1 then
08:              parent = Parent(H, entry)
09:              if f(entry) < f(parent) then
10:                    Swap(H, entry.position, parent.position)
11:                    Push Up(H, entry)
12:
13:  Push Down Impl(H, entry)
14:        leftChild  = Left Child (H, entry)
15:        rightChild = Right Child(H, entry)
16:        if leftChild ≠ ∅ then
17:              child = leftChild
18:              if rightChild ≠ ∅ then
19:                    if f(rightChild) < f(leftChild) then
20:                          child = rightChild
21:              if f(child) < f(entry) then
22:                    Swap(H, entry.position, child.position)
23:                    Push Down(H, entry)
```

На основе рассмотренных выше операций можно легко построить операции добавления и удаления элемента. Так, если требуется добавить элемент в кучу, то вначале он помещается на позицию самого нижнего правого листа, после чего происходит проталкивание вверх. В случае удаления элемента, он меняется местами с самым нижним правым листом, после чего происходит проталкивание вниз с позиции удаляемого элемента. Детали реализации приведены ниже (процедура 2.4.4):

Процедура 2.4.4: Heap: Push, Pop

```
01:  Push Impl(H, dataItem)
02:        if |H| + 1 ≤ max elements count then
03:              elementsCount = elementsCount + 1
04:              entry = heapEntry<T> {
05:                    item = dataItem
06:                    position = elementsCount
07:              }
08:              H[elementsCount] = entry
09:              Push Up(H, entry)
10:              return entry
11:        return ∅
12:
13:  Pop Impl(H, entry)
14:        if entry.position > 0 then
15:              rightist = H[elementsCount]
16:              Swap(H, entry.position, rightist.position)
17:              elementsCount = elementsCount - 1
18:              Push Up  (H, rightist)
19:              Push Down(H, rightist)
```

Помимо данного набора операций, которые можно выполнять над бинарными кучами, рассмотрим еще одну, суть которой заключается в том, чтобы преобразовать заданный массив объектов в пирамиду (процедура 2.4.5):

Процедура 2.4.5. Heap: Build

```
01:  Build Impl(A)
02:        H = Convert(A)
03:        for i = ⌊n / 2⌋ downto 1 do
04:              Push Down(H, i, n)
05:        return H
```

Для того чтобы оценить трудоемкость операции построения, заметим, что в худшем случае операции проталкивания вниз потребуется пройти весь путь от некоторого узла дерева до листа. Предположив, что высота дерева равняется h, и принимая во внимание тот факт, что на уровне i находится 2^i узлов, можно заключить, что полное число шагов S, которое совершит операция проталкивания вниз, не будет превышать следующей величины:

$$S \leq \sum_{i=0}^{h} 2^i \cdot (h-i) = h + \cdots + 2^{h-1}\big(h-(h-1)\big) = 2^{h+1} - h - 2. \quad (2.5)$$

Так как бинарная куча – это сбалансированное бинарное дерево, то можно заключить, что трудоемкость операции построения не будет превышать величины $O(n)$. Следовательно, она линейна относительно числа элементов, на которых строится пирамида.

На основе бинарной кучи АТД «Очередь с приоритетами» может быть реализована следующим образом (процедура 2.4.6):

Процедура 2.4.6: ADT Priority Queue: Push, Peek, Pop

```
01:  priorityQueue<T> {
02:        H: Heap<T>(max elements count)
03:  }
04:
05:  Push Impl(Q, dataItem)
06:        with Q do
07:              Push(H, dataItem)
08:
09:  Peek Impl(Q, dataItem)
10:        with Q do
11:              if |H| > 0 then
12:                    return Item(H[1])
13:        return ∅
14:
15:  Pop Impl(Q, dataItem)
16:        with Q do
17:              if |H| > 0 then
18:                    Pop(H, H[1])
```

Помимо удовлетворительного времени работы каждой из операций (вставка и удаление – $O(\log n)$, нахождение минимума – $O(1)$) данный подход позволяет экономно расходовать память ЭВМ. Таким образом, если требуется обработать n объектов, то затраты по памяти составят $O(n)$ ячеек, что сопоставимо с реализацией очереди с приоритетами посредством массивов, других линейных структур данных и АТД.

В заключение ознакомимся с модификацией обычной очереди, которая позволяет находить элемент с минимальным ключом за $O(1)$. Такие модификации можно рассматривать с точки зрения очередей с приоритетами, на которых не определена операция удаления минимального элемента. Для того чтобы добиться оценки $O(1)$ для операции поиска минимального элемента, достаточно реализовать очередь посредством двух стеков, как это было сделано в разд. 2.2.3. Каждый из стеков, помимо хранения самих данных, дополнительно будет хранить в каждом элементе стека s минимальный элемент среди всех присутствующих, вершиной которого является элемент s. Тогда операцию нахождения минимума на стеке можно реализовать за $O(1)$. Для этого достаточно прочитать значение минимума, которое записано в вершине стека. Так как очередь будет реализована через два стека, то операция нахождения минимума в очереди будет также иметь трудоемкость $O(1)$.

Реализация очереди, описанная выше, находит широкое применение при работе с так называемыми *скользящими окнами*, которые используются для анализа потоков данных. Рассмотрим последовательность событий $e_1, e_2, \dots, e_n$, упорядоченных на временной оси. Предположим, что каждое из событий характеризуется некоторой числовой характеристикой. Скользящее окно предназначено для последовательной обработки событий e_i, причем максимальное число событий, которые могут одновременно находиться в рамках окна, задается некоторым числовым параметром t. Если же число событий, находящихся в рамках окна, превышает число t, то из окна удаляется событие, которое «попало» в окно ранее других. Задача состоит в том, чтобы достаточно быстро отвечать на ряд запросов, среди которых выделяется запрос «Чему равен минимальный элемент, находящийся в текущий момент времени в рамках окна?». Для ответа на такого рода запросы, можно применить либо очередь с приоритетами, либо описанную выше модификацию FIFO-очередей. Использование последней более предпочтительно, так как все операции будут иметь трудоемкость $O(1)$, в то время как использование очередей с приоритетами приведет к трудоемкости $O(\log n)$.*

2.5. Задачи для самостоятельного решения

1. Разработайте алгоритм, который для заданной позиции p в многомерном массиве $A[n_1, n_2, \dots, n_m]$ определит набор индексов $(i_1, i_2, \dots, i_m)$, соответствующий заданной позиции p. Считайте, что индексация по каждому из измерений начинается с нуля, т. е. $0 \le i_j < n_j,\ 1 \le j \le m$.

2. Задан массив A из n элементов. Необходимо циклически сдвинуть массив A на k позиций влево (вправо) используя $O(1)$ дополнительной памяти.

3. Задан неотсортированный массив A, содержащий n чисел. Определите множество P, в котором n элементов, где P_i является произведением всех чисел из A, за исключением a_i. Операция деления запрещена.

4. Задана матрица A размерностью $n \times m$, состоящая из чисел. Требуется найти подматрицу матрицы A, сумма элементов которой максимальна.

5. Задан массив A, в котором n чисел. Определите множество P, содержащее все числа, полученные путем операции сложения чисел из массива A. Предложите эффективный алгоритм, позволяющий преобразовать множество P во множество P_i, содержащее все числа, которые можно получиться из множества $A \backslash \{a_i\}$.

6. В заданном массиве A из n чисел требуется каждый элемент заменить на ближайший следующий за ним элемент, который больше него.

---

* В англоязычной литературе используются термины *sliding window* и *tumbling window*, различие между которыми определяется политикой удаления элементов. Помимо этого выделяется политика, определяющая условия, при которых срабатывает процедура обработки элементов окна. Детальное рассмотрение таких политик выходит за рамки данного пособия.

7. Рассмотрим n-мерный параллелепипед P размерами $n_1 \times n_2 \times ... \times n_m$. Разработайте алгоритм, который построит циклический обход заданного параллелепипеда, начиная с клетки с координатами $s = (s_1, s_2, ..., s_m)$. Каждая клетка параллелепипеда P должна быть посещена ровно один раз. За один ход разрешается перейти в соседнюю клетку по одной из координат. Сформулируйте условия, при которых требуемый обход параллелепипеда P не существует.

8. Задан односвязный список L, состоящий из n элементов. Предполагая, что известен указатель на голову списка L, разработайте алгоритм, имеющий трудоемкость $O(n)$ и требующий памяти $O(1)$, который определит, имеется ли в списке L цикл. Следует отметить, что по завершении работы Вашего алгоритма, список L должен остаться неизмененным. Модифицируйте разработанный алгоритм с целью нахождения количества элементов в цикле.

9. *Графом* будем называть пару (V, E), где V – непустое конечное множество, а E – бинарное отношение на V, т. е. подмножество множества $V \times V$. Множество V будем называть множеством *вершин* графа, а E – множеством *ребер* графа. Разработайте эффективную структуру данных для хранения информации о графе в памяти ЭВМ.

10. Рассмотрим форму тела, заданного матрицей A размерностью $n \times m$. Элемент $A[i, j]$ матрицы соответствует высоте вертикального столбика относительно нижнего основания площадки, расположенного горизонтально. Сечение всех столбиков есть квадрат размерностью 1×1. Требуется определить объем невытекшей воды, который останется внутри формы после того, как ее полностью погрузили в воду, а затем подняли.

11. Рассмотрим односвязный список L, у которого, помимо указателя на следующий элемент, имеется указатель на произвольный элемент того же списка L, отличный от нулевого (процедура 2.5.1):

Процедура 2.5.1. Custom Linked List Type

```
customLinkedListElement<T> {
        item: T
        next: customLinkedList<T>
        refr: customLinkedList<T>
}
customLinkedList<T> {
        head: customLinkedListElement<T>
}
```

Разработайте алгоритм, создающий глубокую копию списка L, т. е. такую копию L', структура которой будет полностью совпадать со структурой исходного списка L.

12. Рассмотрим ЭВМ, у которой есть в распоряжении N $(1 \leq N < 2^{32})$ последовательных ячеек памяти, пронумерованных от 1 до N. Разработайте менеджер памяти, который будет обрабатывать запросы на выделение и освобождение памяти.

Запрос на выделение памяти имеет один параметр K, означающий, что необходимо выделить K последовательных ячеек памяти. Если в распоряжении менеджера есть хотя бы один свободный блок из K последовательных ячеек, то он обязан в ответ на запрос выделить такой блок. При этом непосредственно перед самой первой ячейкой памяти выделяемого блока не должно располагаться свободной ячейки памяти. Если блока из K последовательных свободных ячеек памяти нет, то запрос отклоняется.

Запрос на освобождение памяти имеет один параметр T. Такой запрос означает, что менеджер должен освободить память, выделенную ранее при обработке запроса с порядковым номером T. Запросы нумеруются, начиная с единицы. Гарантируется, что запрос с номером T – запрос на выделение, причем к нему еще не применялось освобождение памяти. Если запрос с номером T был отклонен, то текущий запрос на освобождение памяти игнорируется.

13. Рассмотрим матрицу A размерностью $n \times n$, состоящую из натуральных чисел. *Путем* в матрице A будем называть последовательность из n элементов следующего вида: $p = (a_{1j_1}, a_{2j_2}, \dots, a_{nj_n})$, $1 \le j_k \le n$. *Весом* пути p назовем сумму его элементов, т. е. $w(p) = a_{1j_1} + a_{2j_2} + \cdots + a_{nj_n}$. Разработайте алгоритм, который найдет n путей в матрице A, имеющих наибольший вес.

14. Разработайте алгоритм, находящий радиус наибольшей окружности, которую можно вписать в заданный выпуклый многоугольник.

15. Рассмотрим матрицу $A[n \times m]$, состоящую из 0 и 1. В матрице A требуется найти подматрицу максимальной площади, состоящую из одних единиц.

16. Рассмотрим площадку, основание которой представляет собой прямоугольник, разделенный на $N \times M$ квадратов (т. е. прямоугольник, состоящий из N строк и M столбцов). Квадрат с координатами (i, j), $1 \le i \le N$, $1 \le j \le M$ находится на высоте H_{ij}. Разработайте алгоритм, который найдет участок площадки прямоугольного размера, удовлетворяющий следующим условиям:

- стороны участка должны быть параллельны сторонам площадки;
- площадь участка должна быть максимально возможной;
- ширина участка (то есть, количество столбцов) не должна превышать заданного числа W;
- разница между высотой самого высокого квадрата и самого низкого квадрата участка не должна превышать значения L.

17. Рассмотрим $n + 1$ пунктов, пронумерованных от 0 до n соответственно. Некоторые из пунктов соединены реками. По ним можно сплавлять лес, который вырубается в каждом из пунктов. Также известно, что в пункте 0 находится лесопилка, которая может переработать сплавленный лес, и что для каждого пункта существует единственный путь по рекам в пункт 0. Требуется построить k дополнительных лесопилок ($1 \le k \le n$) так, чтобы суммарная стоимость сплава леса была минимальной. Стоимость сплава одного кубометра леса на расстояние в один километр равняется одному центу.

18. Используя идею техники двоичного подъема, разработайте алгоритм, который позволит эффективно находить минимальное / максимальное число на заданном участке массива. Предложите алгоритм, трудоемкость которого составила бы $\langle O(n \log n), O(1) \rangle$.

19. На координатной прямой расположено n точек в позициях $x_1, \dots, x_n$. Для простоты будем предполагать, что $x_1 < \cdots < x_n$. В момент времени 0 каждая точка, имеющая координату x_i, начинает двигаться вправо с постоянной скоростью $v_i > 0$. В некоторые моменты времени одна из точек может «обогнать» другую. Будем считать, что никакие два «обгона» не происходят одновременно. Упорядочим все обгоны по возрастанию момента времени, в которые они происходят. Требуется вывести информацию о первых k произошедших обгонах. Предполагается, что число k не превосходит общего числа обгонов.

20. Рассмотрим n копилок, каждая из которых может быть либо разбита, либо открыта соответствующим ключом. Для каждого из n ключей известно, в какой копилке находится ключ. Требуется определить минимальное число копилок, разбив которые можно получить доступ ко всем из них.

21. Рассмотрим n городов, соединенных между собой дорогами таким образом, что для каждой пары городов существует единственный путь между ними. Требуется определить минимальное число дорог, которое должно быть разрушено, чтобы образовалась изолированная от остальных группа ровно из m городов, такая что из одного города этой группы в другой по-прежнему можно будет добраться по неразрушенным дорогам. Группа *изолирована* от остальных, если никакая неразрушенная дорога не соединяет город из этой группы с городом из другой группы.

22. Задан массив A из m различных элементов, принадлежащих множеству $\{1, .., n\}, m \leq n$. Требуется с дополнительной памятью $O(n - m)$ определить элементы, отсутствующие в массиве A.

Процедура 2.5.2. Process

```
IEnumerable<int> process(int n, int m, int[] a) {
      if (m < n) {
            a[m, n) = -1;
            for (int i = 0; i < m; ++i) {
                  while (a[i] ≥ 0 && a[a[i]] ≠ a[i]) {
                        swap(a[a[i]], a[i]);
                  }
            }
            for (int i = 0; i < n; ++i) {
                  if (a[i] ≠ i) {
                        yield return i;
                  }
            }
      }
}
```