Глава 1

Структуры данных

Для эффективной работы с данными используются различные способы организации их хранения в памяти — *структуры данных*. В данной главе мы поговорим про самые базовые структуры данных, а так же научимся оценивать амортизированную сложность *запросов* к ним[1].

1.1. Массивы

Наиболее простой структурой данных для RAM-машины является массив. Массив представляется собой *непрерывную* область памяти, разбитую на ячейки одинакового размера. Зная адрес начала массива можно легко вычислить адрес произвольной ячейки. Языки программирования обычно уже имеют готовые функции для работы с массивами, но нам было бы полезно разобраться, как они устроены.

Если нам изначально известно, сколько элементов будет хранится в массиве, или же у нас есть некоторая оценка сверху на количество элементов, то мы можем сразу же *выделить* массив достаточного размера (т.е. договориться, что некоторая непрерывная область памяти будет использоваться для хранения элементов массива). Для добавления элемента в конец заранее выделенного массива достаточно просто скопировать его в ещё незаполненную ячейку и увеличить счётчик, который хранит количество заполненных ячеек. Таким образом, сложность запроса на добавление элемента — $O(1)$ операций. Запрос на удаление последнего элемента реализовать ещё проще, т.к. нам достаточно просто уменьшить счётчик числа заполненных ячеек.

Замечание 1.1.1. Здесь и далее при оценке сложности запросов к структурам данных мы всегда будем предполагать, что размер элементов структуры данных фиксирован и не зависит от размера входа. Другими словами, мы будем предполагать, что сложность копирования любого элемента структуры данных константна, т.е. стоит нам $O(1)$ операций. Если это не так, то в большинстве случаев полученные оценки сложности легко скорректировать, домножив их на сложность копирования одного элемента. Например, если в массиве хранятся строки длины $\sqrt{n}$, то оценки сложности всех запросов, которые приводят к копированию элементов, нужно умножить на $\sqrt{n}$: добавление элемента в конец будет иметь сложность $O(\sqrt{n})$, а удаление последнего элемента — $O(1)$, т.к. не требует копирования элементов.

[1] Для того, чтобы избежать путаницы мы будем говорить об *операциях* процессора и *запросах* к структурам данных.

Если мы захотим удалить или добавить элемент в начале или где-то в середине массива, то нам потребуется “сдвинуть” все последующие элементы, поэтому эти запросы будут иметь сложность $O(n)$, где n — это количество элементов в массиве.

Однако, мы далеко не всегда заранее знаем, сколько элементов будет хранится в массиве. В этом случае можно начать с небольшого массива и расширять его постепенно, добавляя новые ячейки тогда, когда они потребуются. Предположим, что все ячейки массива заполнены, а нам нужно добавить к массиву ещё один элемент? Вполне возможно, что ячейки памяти, которые расположены сразу же после конца массива, свободны. Тогда мы могли бы расширить массив за счёт присоединения к нему необходимого количество ячеек памяти (например, это может сделать функция `realloc` в языке C). Но что делать, если ячейки памяти после конца массива уже заняты или наш язык программирования не поддерживает расширение массивов? В этом случае вместо расширения существующего массива нам придётся создать новый массив большего размера и скопировать туда все данные из исходного массива (массив обязательно должен лежать в непрерывной области памяти, поэтому мы не можем расположить дополнительные ячейки где-то отдельно). Давайте рассмотрим две различные реализации этой идеи.

1.1.1. Расширяемый массив с аддитивной схемой выделения

Предположим, что нам нужно добавить один новый элемент в конец массива размера s, который уже полностью заполнен, т.е. хранит s элементов. В этом случае мы можем выделить новый массив размера $s + c$, где c — некоторая целая положительная константа, скопировать туда все элементы из исходного массива и добавить новый элемент в первую из добавленных пустых ячеек (см. рис. 1.1.1). Это называют *расширяемым массивом с аддитивной схемой выделения*. Какую сложность будет иметь запрос на добавление элемента в при такой реализации расширения? Пусть в массиве n элементов. Если свободных ячеек нет, то при добавлении нового элемента нам потребуется скопировать n элементов из исходного массива и один новый элемент. Таким образом запрос на добавление элемента будет иметь сложность $\Theta(n)$.

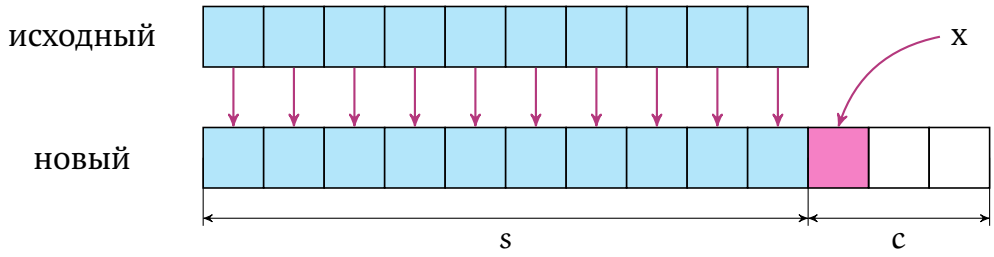

Рис. 1.1.1. Добавление элемента x в заполненный расширяемый массив с аддитивной схемой выделения при $c = 3$.

Полезно также оценить суммарные накладные расходы на расширение массива. Пусть мы начали с пустого массива, и в процессе работы он увеличился до размера n. Сколько дополнительных операций потребовалось? Для простоты будем учитывать только операции копирования элементов. При первом нетривиальном расширении массива было скопировано c элементов, при втором — $2c$ элементов, и так далее. В сумме получается, что на все расширения массива по-

требовалось

$$c + 2c + 3c + \cdots + (n - c) = \frac{n(n/c - 1)}{2} = \Theta(n^2)$$

дополнительных копирований (см. лемму A.1.1). Получается, что при такой реализации расширяемого массива любой алгоритм, который использует массив размера n, имеет сложность $\Omega(n^2)$.

1.1.2. Расширяемый массив с мультипликативной схемой выделения.

Вместо увеличения размера массива на константу, можно увеличивать размер массива *в константу раз*. Эта идея называется *расширяемым массивом с мультипликативной схемой выделения*. Выберем некоторое $\alpha > 1$ (наиболее типичным значением является $\alpha = 2$). Если в текущем массиве размера s все ячейки заполнены, то выделим новый массив размера $\lceil \alpha s \rceil$ (см. рис. 1.1.2).

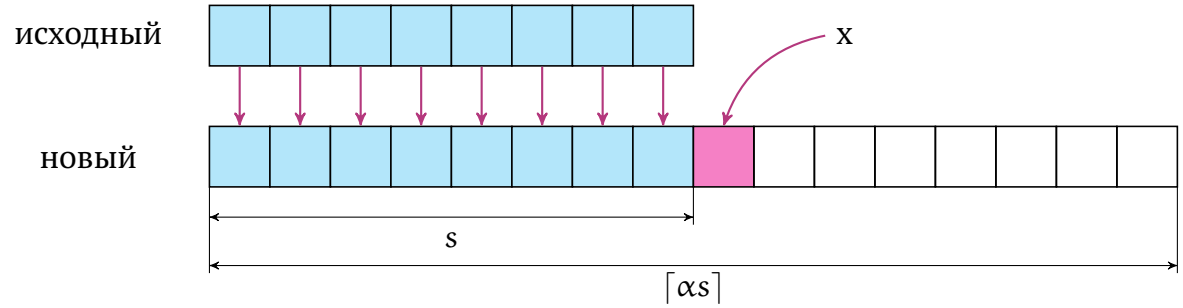

Рис. 1.1.2. Добавление элемента x в заполненный расширяемый массив с мультипликативной схемой выделения при $\alpha = 2$.

Давайте оценим также суммарные накладные расходы на расширение массива. Пусть мы начали с массива некоторого фиксированного размера, и в процессе работы размер увеличился до n. При последнем расширении массива было скопировано $\lfloor n/\alpha \rfloor$ элементов, при предпоследнем — $\lfloor n/\alpha^2 \rfloor$, и так далее. В сумме на расширения массива потребовалось $\lfloor n/\alpha \rfloor + \lfloor n/\alpha^2 \rfloor + \cdots + c$ дополнительных копирований, где c — начальный размер массива. Эту сумму можно оценить сверху суммой бесконечной убывающей геометрической прогрессии (см. лемму A.1.3)

$$\lfloor n/\alpha \rfloor + \lfloor n/\alpha^2 \rfloor + \cdots + c \leqslant \sum_{k=1}^{\infty} \lfloor n/\alpha^k \rfloor < \sum_{k=1}^{\infty} n/\alpha^k = n \cdot \frac{1}{1 - 1/\alpha} = \Theta(n).$$

Таким образом при мультипликативной схемой выделения накладные расходы линейные. Это можно переформулировать следующим образом: в среднем на каждый элемент массива требуется $O(1)$ дополнительных операций на расширение массива. Стоит отметить, что не смотря на это, запрос на добавление в массив будет иметь сложность $\Theta(n)$ в худшем (сложность в худшем оценивает количество операций в самом “плохом” случае).

Упражнение 1.1.1. Как реализовать запрос на удаление элемента из массива так, чтобы в среднем на каждый элемента массива дополнительно требовалось $O(1)$ операций, и при этом свободное место использовалось бы экономно (т.е., например, если из большого массива удалить почти все элементы, то зарезервированное свободное место должно пропорционально уменьшиться)?

1.2. Списки

Другая базовая структура данных — это список. В отличие от массива, где все данные хранятся в непрерывной области памяти, элементы списка могут быть разбросаны по памяти произвольным образом. Выделяют два основных вида списков: *односвязный* и *двусвязный*.

1.2.1. Односвязный список

Каждый элемент *односвязного списка* хранится в отдельной структуре с двумя полями — *узле списка*. Кроме самого элемента каждый узел хранит *указатель* на следующий узел — номер ячейки памяти, где находятся данные следующего узла списка (см. рис. 1.2.1). Односвязный список задаётся парой указателей: на первый узел списка, *голову списка*, и на последний узел списка, *хвост списка*.

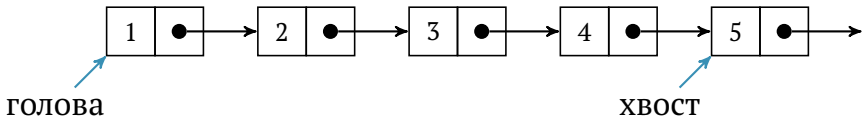

Рис. 1.2.1. Односвязный список.

Такая организация хранения данных позволяет перебирать элементы от головы к хвосту списка, причём переход к следующему элементу выполняется за $O(1)$, удалять и добавлять элементы, находящиеся в начале списка, за $O(1)$, добавлять элементы в конец списка за $O(1)$, а так же удалять и добавлять элементы в произвольном месте списка за $O(1)$ по ссылке на предыдущий элемент см. рис. 1.2.2. При этом обращение по индексу и удаление из конца списка будут работать за $\Theta(n)$ (для удаления последнего элемента нужно найти предпоследний узел, а для этого придётся пройти весь список от головы до хвоста).

Кроме того, списки позволяют эффективно осуществлять некоторые более сложные операции. Например, можно «вырезать» часть списка и выделить её в отдельный список или наоборот «вклеить» один список внутрь другого, причём обе операции будут работать за $O(1)$, т.к. они требуют изменения всего лишь пары указателей.

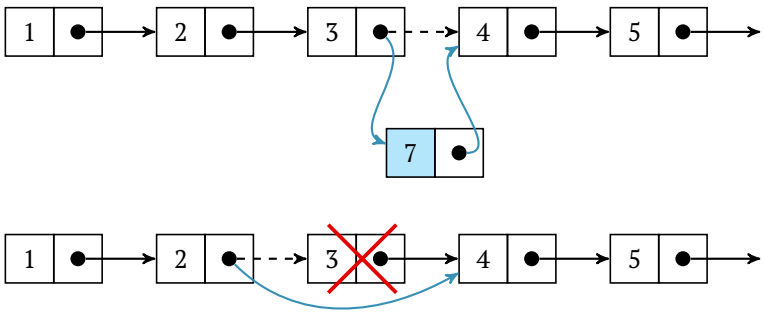

Рис. 1.2.2. Добавление и удаление элемента односвязного списка.

1.2.2. Двусвязный список

Если есть необходимость перебирать элементы списка в обоих направлениях, то применяется *двусвязный список*. Элементы двусвязного списка хранятся в узлах вместе с указателями на следующий и предыдущий узлы списка (см. рис. 1.2.3).

За счёт хранения двух указателей двусвязный список позволяет добавлять и удалять элементы в произвольном месте списка за $O(1)$. При этом доступ к элементу по индексу так же осуществляется за $\Theta(n)$.

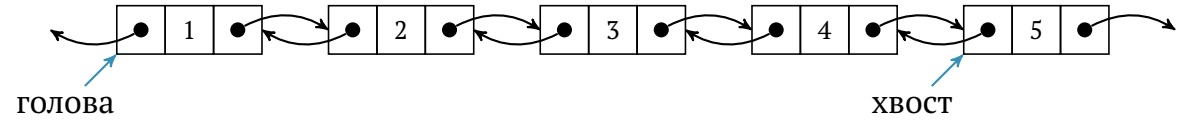

Рис. 1.2.3. Двусвязный список.

По функциональности двусвязный список ни в чём не уступает односвязному, в т.ч. так же позволяет эффективно реализовать операции «вырезания» и «вклеивания». Тем не менее, в тех приложениях, где память очень ограничена (например, внутри кода драйверов или во встроенных системах), предпочитают использовать односвязные списки, если их возможностей достаточно для решения поставленной задачи.

Упражнение 1.2.1. Придумайте, как можно хранить двусвязный список так, чтобы в каждом узле хранился только один указатель, но при этом мы могли бы перебирать элементы списка в обоих направлениях. (Подсказка: нужно использовать битовые операции с целыми числами.)

1.2.3. Сравнение с массивами

В следующей таблице представлены сложности основных запросов к массивам и спискам.

Запрос	**Массив**	**Односвязный список**	**Двусвязный список**
Обращение по номеру	$O(1)$	$O(n)$	$O(n)$
Добавление в начало	$O(n)$	$O(1)$	$O(1)$
Удаление из начала	$O(n)$	$O(1)$	$O(1)$
Добавление в конец	$O(n)$	$O(1)$	$O(1)$
Удаление из конца	$O(1)$	$O(n)$	$O(1)$
Добавление в середину	$O(n)$	$O(1)$	$O(1)$
Удаление из середины	$O(n)$	$O(1)$	$O(1)$

Таблица 1.1. Сравнение сложности запросов к массивам и спискам.

1.3. Абстрактные типы данных

Определение 1.3.1. *Абстрактный тип данных (АТД)* — это математическое описание структуры данных в терминах реализуемых ей запросов.

Понятия абстрактного типа данных и структуры данных соотносятся так же, как в объектно-ориентированном программировании соотносятся понятия интерфейса и класса. Когда мы определяем структуру данных, то мы подробно описываем то, как эта структура устроена, и какие запросы к этой структуре можно реализовать эффективно. При определении абстрактного типа данных мы описываем только те запросы, которые этот тип данных должен реализовывать, но ничего не говорим про его устройство в памяти и реализацию запросов. Таким

образом, у одного абстрактного типа данных может быть несколько реализаций, основанных на разных структурах данных. В этом разделе мы рассмотрим два базовых абстрактных типа данных и поговорим о том, как их можно реализовать на основе массивов и списков.

1.3.1. Стек

Абстрактный тип данных *стек* определяется следующими запросами:

- `push(x)` — добавить элемент в стек,
- `pop()` — вытолкнуть элемент стека.

Порядок, в котором элементы выталкиваются из стека, является обратным порядку добавления, т.е. первым выталкивается тот элемент, который был добавлен последним (см. рис. 1.3.1). Про стек говорят, что он реализует концепцию «последним пришёл — первым ушёл» («last in, first out», LIFO). В обычной жизни такой порядок можно встретить, например, если положить несколько книг в стопку на стол: сверху будет лежать та книга, которую мы положили последней. В соответствии с этой аналогией про элемент, который добавлен последним, говорят, что он находится *на вершине* стека, а про тот, что был добавлен первым — что он находится *на дне* стека.

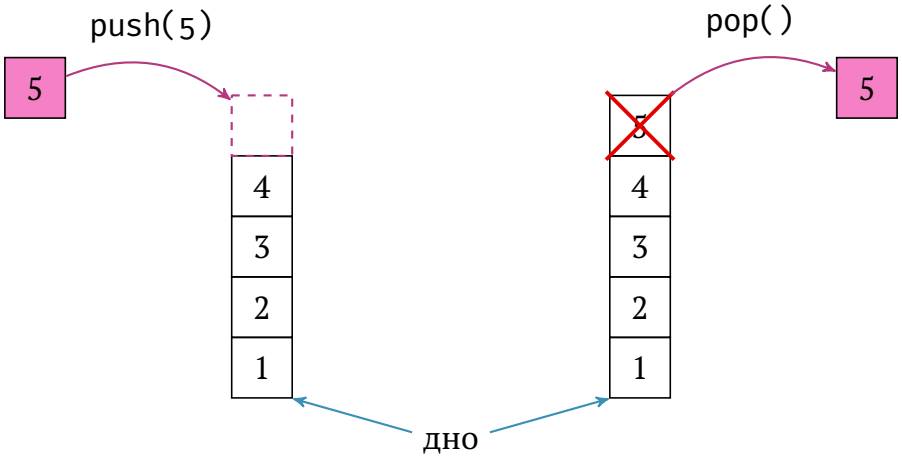

Рис. 1.3.1. Операции со стеком: добавление и выталкивание элемента.

Реализация на списках. Стек эффективно реализуется на односвязном списке, а следовательно и на двухсвязном: элементы добавляются и выталкиваются из начала списка.

Реализация на массиве. При эффективной реализации стека на массиве элементы добавляются в конец массива и выталкиваются из конца массива. Если массив заполняется полностью, то происходит его расширение по мультипликативной схеме.

1.3.2. Очередь

Абстрактный тип данных *очередь* определяется следующими запросами:

- `enqueue(x)` — добавить элемент `x` в очередь,
- `dequeue()` — извлечь элемент из очереди.

Как и в обычной жизни, порядок, в котором элементы извлекаются из очереди, должен соответствовать порядку, в котором они очередь добавляются. Про очередь говорят, что она реализует концепцию «первым пришёл — первым ушёл» («first in, first out», FIFO).

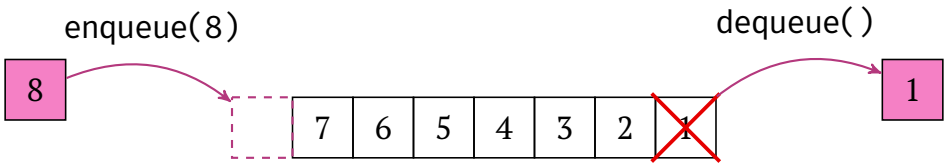

Рис. 1.3.2. Операции с очередью.

Реализация на списках. Очередь можно эффективно реализовать на двусвязном списке, т.к. он позволяет добавлять элементы в начало и удалять из конца за $O(1)$. Если немного подумать, то становится ясно, что очередь можно эффективно реализовать и на односвязном списке — для этого нужно добавлять элементы в конец списка, а удалять из начала.

Реализация на массиве. Для эффективной реализации очереди на массиве требуется использовать идею *кольцевого буфера*. Для этого в массиве поддерживаются два указателя: на начало очереди и на конец очереди (см. рис. 1.3.3). Элементы добавляются по указателю на конец очереди, выталкиваются по указателю на начало. При этом, если указатель достигает конца массива, то он «перескакивает по циклу» на начало массива. Если после добавления элемента в очередь указатели на начало и конец встретились, т.е. весь массив заполнен элементами очереди, то массив расширяется по мультипликативной схеме, и элементы очереди копируются в новый массив от начала до конца.

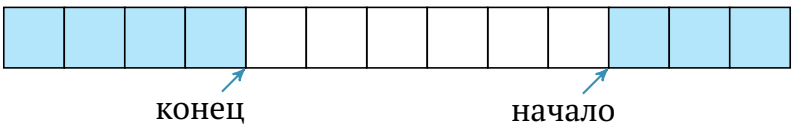

Рис. 1.3.3. Реализация очереди на массиве с использованием кольцевого буфера.

Реализация на двух стеках. В некоторых случаях можно описать реализацию одного абстрактного типа данных через другой абстрактный тип данных. В частности в некоторых алгоритмах используется следующий способ реализации очереди на двух стеках. Пусть далее s_1 и s_2 — это два стека, на основе которых мы должны реализовать запросы `enqueue` и `dequeue`. Будем предполагать, что стеки реализуют запросы `push`, `pop` и также запрос `empty`, который возвращает `True`, если стек пуст. Реализация запроса `enqueue` будет затрагивать только первый стек.

```
def enqueue(x):
    s1.push(x)
```

При запросе dequeue проверяется, есть ли элементы во втором стеке. Если второй стек пуст, то мы перекладываем элементы первого стека во второй, и после этого выталкиваем верхний элемент из второго стека (см. рис. 1.3.4).

```
def dequeue(x):
    if s2.empty():
        while not s1.empty():
            s2.push(s1.pop())

    return s2.pop()
```

При такой реализации очереди запрос dequeue будет иметь сложность $\Theta(n)$ в худшем (при условии, что стек реализован эффективно). Однако, можно показать, что суммарные накладные расходы на все запросы dequeue так же будут линейными. Для этого нужно заметить, что для каждого элемента очереди будет сделано не более двух запросов push и не более двух запросов pop. Следовательно, сложность любой последовательности запросов к такой очереди для n элементов ограничена $O(n)$.

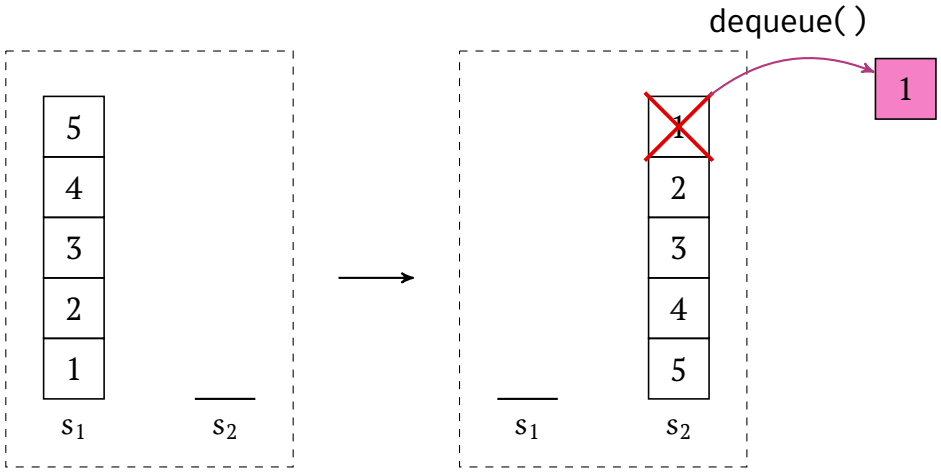

Рис. 1.3.4. Если при запросе dequeue( ) стек s_2 пуст, то содержимое стека s_1 поэлементно перекладывается в стек s_2.

Упражнение 1.3.1. Объясните, почему не получится реализовать стек при помощи нескольких (константного количества) очередей.

1.4. Двухсторонняя очередь

Двухсторонняя очередь позволяет добавлять и выталкивать элементы с обеих сторон, т.е. определяется следующими запросами:

- enqueue_back(x) — добавляет элемент x в конец очереди,
- enqueue_front(x) — добавляет элемент x в начало очереди,
- dequeue_front( ) — вытаскивает элемент из начала очереди.
- dequeue_back( ) — вытаскивает элемент из конца очереди.

Реализация на списках. Двухстороннюю очередь можно эффективно реализовать на двусвязном списке, но не на односвязном, т.к. он не позволяет эффективно удалять из конца списка.

Реализация на массиве. Эффективная реализация двухсторонней очереди на массиве использует ту же идею кольцевого буфера, только теперь указатели могут двигаться в обоих направлениях. Соответственно, если указатель на начало массива нужно сдвинуть влево, то он «перепрыгивает по циклу» на последний элемент массива.

1.5. Амортизационный анализ

Амортизационный анализ (англ. amortized analysis) — это способ подсчёта средней сложности запроса к структуре данных. Он позволяет оценить *амортизированную стоимость*[2] *запроса* — средняя сложность (в худшем случае), где усреднение берётся по всем запросам к структуре данных. Мы рассмотрим два метода получения таких оценок.

1.5.1. Метод учётный стоимостей

Опишем технику оценки амортизированной стоимости запросов к структурам данных. Пусть к некоторой структуре данных, содержащей не более n элементов, выполняются m запросов, и пусть $c_1, \ldots, c_m$ — *истинные* стоимости этих запросов (т.е. реальное количество операций, потраченных на их выполнение). Если (индивидуальную) сложность каждого запроса нельзя оценить лучше, чем $O(f(n))$, то суммарная стоимость m таких запросов можно оценить так

$$\sum_{i=1}^{m} c_i = O(m \cdot f(n)).$$

Пусть $S = \{S_0, S_1, \ldots, S_m\}$ — это множество состояний структуры данных, где S_0 — это исходное состояние, а S_i — состояние после запроса с номером i. Для любой функции $\Phi : S \to \mathbb{R}_+$ можно определить *учётную стоимость запроса* i *относительно* Φ, обозначаемую $\tilde{c}_i$, следующим образом:

$$\tilde{c}_i = c_i + \Phi(S_i) - \Phi(S_{i-1}).$$

Для удобства введём обозначение $\Phi_i = \Phi(S_i)$. В таких обозначениях

$$\tilde{c}_i = c_i + \Phi_i - \Phi_{i-1}.$$

Просуммируем это соотношение по всем i:

$$\sum_{i=1}^{m} \tilde{c}_i = \sum_{i=1}^{m} c_i + \sum_{i=1}^{m} \Phi_i - \sum_{i=1}^{m} \Phi_{i-1} = \sum_{i=1}^{m} c_i + \underbrace{\Phi_m - \Phi_0}_{\Delta\Phi}.$$

[2] В амортизационном анализе принято использовать термин «стоимость» как синоним «сложности»

Основная идея метода. Пусть про Φ дополнительно известно, что $\Delta\Phi \geqslant 0$ (можно потребовать, например, чтобы $\Phi(S_0) = 0$). Тогда мы можем оценить сумму истинных стоимостей как

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} \tilde{c}_i - \Delta\Phi \leqslant \sum_{i=1}^{m} \tilde{c}_i.$$

Заметим, что у нас есть большая свобода в выборе функции Φ, т.к. от неё почти ничего не требуется. Если нам удастся подобрать Φ такую, что учётная стоимость[3] любого запроса будет ограничена $O(g(n))$, т.е. $\tilde{c}_i = O(g(n))$ для любого i, и при этом $g(n) = o(f(n))$, то в результате мы получим лучшую оценку на общее число операций

$$\sum_{i=1}^{m} c_i \leqslant \sum_{i=1}^{m} \tilde{c}_i = O(m \cdot g(n)) = o(m \cdot f(n)).$$

В таком случае мы будем говорить[4], что *амортизированная стоимость запроса* не более $O(g(n))$.

Пример: анализ расширяющегося массива. Применим метод учётных стоимостей для оценки сложности запроса добавления элемента в расширяющийся массив. Мы уже знаем, что вне зависимости от схемы выделения памяти в худшем случае сложность добавления нового элемента в массив $O(n)$. Однако мы также показали, что при мультипликативной схему выделения памяти *в среднем* на добавление каждого нового элемента мы потратим $O(1)$. Давайте докажем это при помощи метода учётных стоимостей.

Будем предполагать, что мы начинаем с пустого массива и последовательно добавляем в него n элементов, кроме того коэффициент расширения $\alpha = 2$. Для получения оценки нам достаточно оценить количество копирований как самой трудоёмкой части запроса.

Для доказательства нам нужно предъявить функцию потенциала, относительно которой учётная стоимость запросов будет константной. Чтобы подобрать такую функцию, давайте посмотрим, сколько копирований происходит при добавлении нового элемента в конец массива в зависимости от его номера. Если это первый элемент, то требуется одно копирование: выделяется массив единичного размера и в него копируется первый элемент. Для добавления второго элемента требуется выделить массив размера два и скопировать туда первый и второй элементы. И так далее. Заметим, что для добавления четвёртого элемента нам требуется только одно копирование, т.к. к этому моменту мы уже выделили массив размера 4, одна из ячеек которого свободна. Аналогично, только одно копирование требуется для пятого, шестого, седьмого и восьмого элементов (см. рис. 1.5.1).

Искомая функция потенциала в должна вести себя в некотором смысле противоположно тому, как ведут себя стоимости операций c_i: в те моменты, когда c_i большое, функция потенциала должна резко убывать, компенсируя таким образом c_i, а в тех случаях, когда для добавления требуется только одна операция,

[3] Учётная стоимость используется как удобная абстракция, которая не имеет “физического” смысла — в зависимости от разных Φ мы будем получать разные учётные стоимости, которые ничего не говорят про истинную стоимость запросов.

[4] Тут важно не забывать, что мы оценили амортизированную стоимость запроса относительно последовательности из m запросов. Сама по себе эта оценка ничего не говорит про сложность одного запроса.

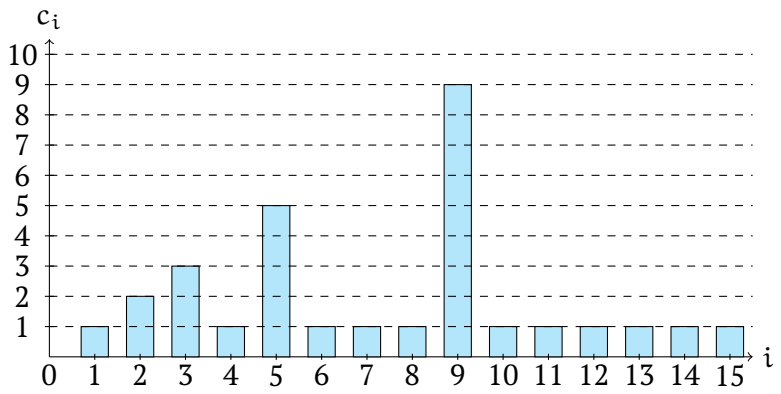

Рис. 1.5.1. Стоимость добавления элемента в расширяющийся массив.

то потенциал должен возрастать не сильно. Для того, чтобы определить такую функцию, нам нужно как-то описать состояние расширяющегося массива. Давайте за l обозначим размер массива, а за s — количество заполненных ячеек (см. рис. 1.5.2).

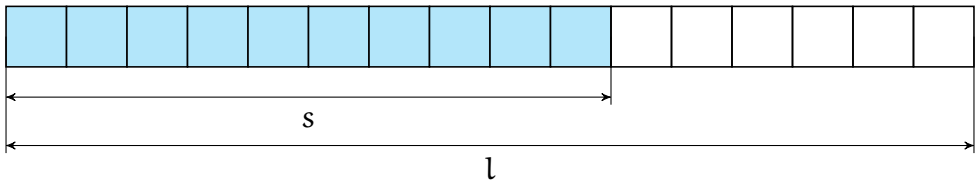

Рис. 1.5.2. Массив размера l с s элементами.

Заметим, что s с каждым добавленным элементом возрастает на 1, а l либо не изменяется, либо увеличивается в 2 раза, если массив был полностью заполнен, т.е., когда для добавления нового элемента требуются дополнительные копирования. На основе этих наблюдений можно догадаться, что нам может подойти линейная функция потенциала, которая положительно зависит от s и отрицательно от l — тогда эта функция будет медленно возрастать, пока в массиве есть свободные ячейки, и резко падать, если массив заполнился и произошло расширение. Осталось подобрать подходящие коэффициенты. Исходя из того, что при расширении массива уменьшение функции потенциала должно компенсировать дополнительные затраты на копирования, предлагается использовать функцию $\Phi(s, l) = 2s - l$ (см. рис. 1.5.3).

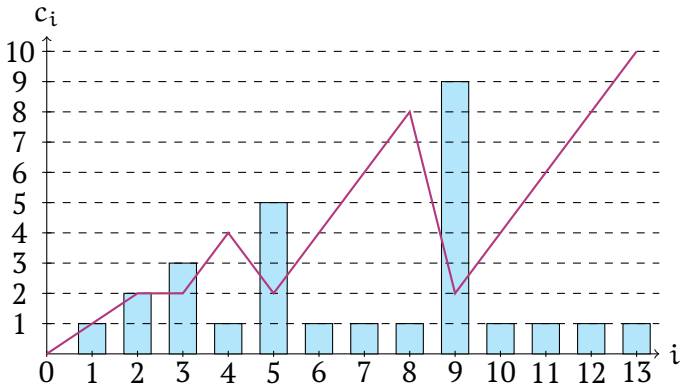

Рис. 1.5.3. Потенциал Φ и стоимость добавления элемента массива.

Нам осталось показать, что учётные стоимости запросов на добавление действительно ограничены константой относительно предложенной функции потенциала $\Phi(s, l) = 2s - l$. По определению учётная стоимость равна

$$\tilde{c}_i = c_i + \Phi_i - \Phi_{i-1}.$$

Пусть l и s обозначают соответственно размер массива и количество элементов в нём до добавления элемента с номером i. Нам нужно рассмотреть два случая.

1. В массиве есть пустые ячейки, т.е. $s < l$. Тогда для добавления нам потребуется только одно копирование, т.е. $c_i = 1$, следовательно
$$\tilde{c}_i = c_i + \Phi(s+1, l) - \Phi(s, l) = 1 + (2(s+1) - l) - (2s - l) = 3.$$
2. Массив полностью заполнен, т.е. $s = l$. В этом случае нужно скопировать все s элементов в новый массив и потом скопировать туда же новый элемент, т.е. $c_i = s + 1$. Поэтому
$$\tilde{c}_i = c_i + \Phi(s+1, 2l) - \Phi(s, l) = (s+1) + (2(s+1) - 2l) - (2s - l) = 3$$
(тут мы пользуемся тем, что $s = l$).

Мы показали, что относительно такой функции потенциала учётные стоимости запросов ограничены константой. Осталось заметить, что потенциал пустого массива $\Phi_0 = \Phi(0, 0) = 0$. Это позволяет нам заключить, что запрос на добавление элемента в конец массива при использовании мультипликативной схемы расширения имеет амортизированную оценку сложности $O(1)$.

Упражнение 1.5.1. Подберите функцию потенциала для произвольного коэффициента расширения $\alpha > 1$.

Упражнение 1.5.2. Рассмотрим двоичный счётчик, который реализует запрос на увеличение значения на единицу. Если в счётчике хранится число n, то для увеличения его на единицу в худшем случае необходимо изменить значения $1 + \lfloor \log_2 n \rfloor$ битов, т.е. сложность этого запрос равна $O(\log_2 n)$. Покажите, что амортизированную стоимость запроса на увеличение не превосходит $O(1)$.

1.5.2. Метод предоплаты

Метод предоплаты значительно менее формален, чем метод учётных стоимостей, и за счёт этого позволяет упростить некоторые рассуждения. Метод основан на следующей идее: для того, чтобы показать, что суммарное количество операций невелико, можно каждому элементу структуры данных «выдать» некоторое количество монеток, которыми этот элемент сможет «оплачивать» операций с ним. Если показать, что каждому элементу хватит выданных монеток, и что в сумме на m запросов к структуре данных, хранящей n элементов, нам потребуется не более $O(m \cdot g(n))$ монет, то можно заключить, что амортизированную стоимость запросов ограничена $O(g(n))$.

Пример: анализ очереди на двух стеках. В разделе 1.3.2 мы рассматривали реализацию очереди на двух стеках. Воспользуемся методом предоплаты для того, чтобы показать, что запросы к такой очереди имеют амортизированную стоимость $O(1)$. Для этого каждому элементу, который попадает в очередь, выдаётся четыре монетки на оплату копирований. Легко увидеть, что четырёх монеток достаточно: каждый элемент должен оплатить две операции push и две операции pop. В сумме на оплату любых m запросов нам потребуется не более $4m$ монеток. Следовательно амортизированная стоимость запросов к такой очереди $O(1)$.

Приложение А

Математические факты

А.1. Суммирование последовательностей

Лемма А.1.1. *Сумма первых* n *членов* арифметической прогрессии

$$a_1 = a, \quad a_i = a_{i-1} + d = a + (i-1)d,$$

вычисляется по формуле

$$S_n = \frac{a_1 + a_n}{2} \cdot n = \frac{2a + (n-1)d}{2} \cdot n.$$

Доказательство.

$$2S_n = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} a_{n-i+1} = \sum_{i=1}^{n} (a_i + a_{n-i+1}) = \sum_{i=1}^{n} (2a + (n-1)d).$$

Отсюда получаем, что

$$S_n = \frac{1}{2} \sum_{i=1}^{n} (2a + (n-1)d) = \frac{n}{2} \cdot (2a + (n-1)d).$$

□

Лемма А.1.2. *Сумма первых* n *членов* геометрической прогрессии

$$a_1 = a, \quad a_i = a_{i-1} \cdot q = a \cdot q^{i-1},$$

вычисляется по формуле

$$S_n = \frac{a_{n+1} - a_1}{q - 1} = a \cdot \frac{q^n - 1}{q - 1}.$$

Доказательство.

$$S_n = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} a \cdot q^{i-1}.$$

$$qS_n = q \sum_{i=1}^{n} a \cdot q^{i-1} = \sum_{i=2}^{n+1} a \cdot q^{i-1}.$$

$$qS_n - S_n = \sum_{i=2}^{n+1} a \cdot q^{i-1} - \sum_{i=1}^{n} a \cdot q^{i-1} = a \cdot q^n - a.$$

Отсюда получаем, что

$$S_n = \frac{a \cdot q^n - a}{q - 1} = a \cdot \frac{q^n - 1}{q - 1}.$$

□

Лемма А.1.3. *Сумма бесконечной убывающей геометрической прогрессии с* $q \in (0, 1)$ *вычисляется по формуле* $S = \frac{1}{1-q}$.

Доказательство.

$$S = \lim_{n \to \infty} a \cdot \frac{q^n - 1}{q - 1} = a \cdot \frac{-1}{q - 1} = \frac{a}{1 - q}.$$

□

Лемма А.1.4. *Сумма первых* n *членов бесконечной убывающей последовательности* $a_i = i \cdot q^i$ *при* $q \in (0, 1)$ *вычисляется по формуле* $S_n = \frac{q}{(1-q)^2}$

Доказательство. Рассмотрим функцию $f(x) = \sum_{i=1}^{\infty} x^{i-1}$ и вычислим её вычислим производную.

$$f'(x) = \sum_{i=1}^{\infty} (i - 1)x^{i-2} = \sum_{i=1}^{\infty} i \cdot x^{i-1}.$$

С другой стороны по лемме А.1.3 $f(x) = \frac{1}{1-x}$ при $x \in (0, 1)$.

$$f'(x) = \left(\frac{1}{1 - x}\right)' = \frac{1}{(1 - x)^2}.$$

Осталось заметить, что

$$S_n = \sum_{i=1}^{\infty} i \cdot q^i = q \cdot f'(q),$$

а следовательно $S_n = \frac{q}{(1-q)^2}$. □