БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

СБОРНИК ЗАДАЧ

ПО ТЕОРИИ АЛГОРИТМОВ.

Структуры данных.
Часть. Простейшие структуры данных.

МИНСК
2020

УДК 510.51(075.8)
ББК 22.12я73-1
С23

А в т о р ы :
С. А. Соболь, К. Ю. Вильчевский, В. М. Котов, Е. П. Соболевская

Р е ц е н з е н т ы :
кафедра информатики и методики преподавания информатики физико-математического факультета Белорусского государственного педагогического университета им. М. Танка
(заведующий кафедрой, кандидат педагогических наук, доцент *С. В. Вабищевич*);
профессор кафедры информационных технологий в культуре Белорусского государственного университета культуры и искусства, кандидат физико-математических наук, доцент *П. В. Гляков*

Часть 1

ПРОСТЕЙШИЕ СТРУКТУРЫ ДАННЫХ

Структура данных представляет собой набор некоторым образом сгруппированных данных. Для каждой структуры определяется, каким образом данные хранятся в памяти компьютера, какие базовые операции можно выполнять над этими данными и за какое время.

Для того чтобы разработать эффективный алгоритм решения поставленной задачи, нередко необходимо проанализировать несколько различных структур данных и выбрать наиболее подходящую. Может оказаться, что каждая из структур данных позволяет нам выполнять одну из операций легко, а другие — с большим трудом, поэтому часто выбирают ту структуру, которая лучше всего подходит для решения всей задачи в целом (не делает максимально лёгким выполнение ни одной операции, но позволяет выполнить всю работу лучше, чем при любом очевидном подходе). Поэтому разработка эффективного алгоритма напрямую связана с выбором хорошей структуры данных.

В данном разделе рассмотрим такие простейшие структуры данных, как массив фиксированного размера, динамический массив и связный список [3].

В дальнейшем, если не оговорено иное, то будем предполагать, что в рассматриваемой структуре данных хранится n элементов.

1.1. МАССИВ

Наиболее известной и распространённой структурой данных является массив.

Массив (англ. *array*) — это структура данных с *произвольным доступом* (англ. *random access*) к элементу, т. е. доступ к любому элементу по индексу осуществляется за время O(1) вне зависимости от того, где в массиве располагается элемент (в отличие от *последовательного доступа*, когда время доступа к элементу зависит от места его расположения в структуре).

Эта структура однородна, так как все компоненты имеют один и тот же тип. Под массив в памяти компьютера выделяется непрерывный блок памяти. Элементы массива в памяти располагаются один за другим и являются равнодоступными. *Индексами* являются последовательные целые числа (рис. 1.1). Математическим аналогом массива является вектор или матрица:

$$A = \begin{pmatrix} a_1 & a_2 & \ldots & a_n \end{pmatrix}.$$

При написании программ на псевдокоде будем использовать для i-го элемента массива A обозначение `a[i]`, в формулах будем писать a_i.

Поддержка массивов (свой синтаксис объявления, функции для работы с элементами и т. д.) есть в большинстве высокоуровневых языков программирования.

Как правило, нумерация элементов начинается с нуля (0-индексация). Это удобно, так как адрес расположения i-го элемента в памяти можно быстро вычислить, прибавив к адресу начала массива размер одного элемента, умноженный на i.

Рис. 1.1. Устройство массива

Кроме одномерных (линейных) массивов, нередко в задачах удобно использовать многомерные массивы, в которых обращение к элементу идёт по нескольким индексам. Доступ к произвольному элементу такого массива тоже выполняется за константное время.

В массивах трудно осуществлять поиск элемента (для этого в общем случае необходим проход по всему массиву), трудно производить включение или исключения отдельных элементов (необходимо выполнять сдвиг других элементов вправо или влево).

1.2. ДИНАМИЧЕСКИЙ МАССИВ

Размер массива в простейшем случае фиксирован и должен быть известен заранее. Однако на практике часто удобно использовать динамический массив, который можно расширять по мере надобности.

Под *динамическим массивом* (англ. *dynamic array*) понимается структура данных, которая обеспечивает произвольный доступ и позволяет добавлять или удалять элементы.

Рассмотрим классический сценарий использования динамического массива: пусть изначально массив пуст, затем в него последовательно добавляют n элементов, при этом каждый раз новый элемент добавляется в конец. Как можно организовать динамический массив на базе статического?

1.2.1. Наивная реализация

Каждый раз при необходимости изменения размера будем делать *реаллокацию* (англ. *reallocation*), т. е. выделять новый массив и перемещать все элементы из старого массива в новый.

Подсчитаем общее число «лишних» операций по перемещению данных. Так, для добавления i-го по счёту ($1 \leqslant i \leqslant n$) элемента потребуется перенести все $i-1$ элементов на новое место и записать в конец один новый элемент. Всего на выполнение n перекладываний уйдёт

$$T(n) = \sum_{i=1}^{n}(i-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

операций. Это слишком медленно для использования на практике.

Попробуем уменьшить число реаллокаций. Очевидная идея — расширять массив «с запасом», оставляя пустые ячейки, которые можно будет использовать на следующих шагах. Число реально занятых ячеек памяти будем называть *логическим размером* (*size*) динамического массива. Общее число зарезервированных ячеек будем называть *ёмкостью* (*capacity*) (рис. 1.2).

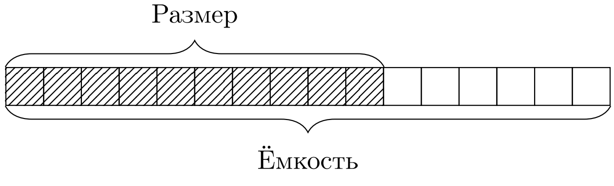

Рис. 1.2. Размер и ёмкость

1.2.2. Линейный рост

Будем каждый раз расширять массив не на один элемент, а сразу на Δ элементов ($\Delta \geqslant 1$). Последовательность изменения ёмкости будет иметь вид $0, \Delta, 2\Delta, 3\Delta, \ldots$.

Так, при добавлении первого элемента сразу будет выделен массив ёмкости Δ, в его начало будет сохранён первый элемент, а остальные $\Delta - 1$ ячеек пока останутся пустыми.

Следующие $\Delta - 1$ добавлений будут выполнены легко и быстро, так как перевыделение памяти не требуется. На каждом шаге будет увеличиваться логический размер, а ёмкость изменяться не будет.

Как только поступит значение под номером $\Delta + 1$, потребуется создать новый массив ёмкости 2Δ и перенести все данные в него, затем уже сохранить новый элемент (рис. 1.3).

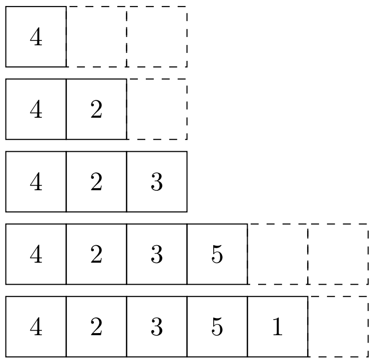

Рис. 1.3. Этапы линейного расширения массива при $\Delta = 3$

Нетрудно заметить, что в ходе добавления n элементов реаллокация выполняется $k = \lceil n/\Delta \rceil$ раз, каждый раз перемещается $0, \Delta, 2\Delta, \ldots$ значений, поэтому общее число «лишних» операций вычисляется по формуле

$$T(n) = \sum_{i=1}^{k} (i-1)\Delta = \Delta \cdot \sum_{i=0}^{k-1} i = \Delta \cdot \frac{k(k-1)}{2}.$$

Пользуясь свойством $\lceil x \rceil \geqslant x$ функции «потолок», получим оценку снизу:

$$T(n) \geqslant \Delta \cdot \frac{1}{2} \cdot \frac{n}{\Delta} \cdot \left(\frac{n}{\Delta} - 1\right) \geqslant \frac{n^2}{2\Delta} = \Omega(n^2). \tag{1.1}$$

Получается, что алгоритм по-прежнему квадратичный. Нетрудно построить аналогичным образом оценку сверху и доказать, что в формуле (1.1) букву Ω можно смело заменить на Θ.

Казалось бы, выбрав в качестве Δ число порядка n (скажем, $n/10$), можно добиться линейного времени работы. Но так рассуждать нельзя: число n заранее неизвестно, а константа Δ должна быть зафиксирована и не может зависеть от входных данных.

Практическое преимущество подхода «рост с шагом Δ» по сравнению с простейшим — он работает быстрее в константное число (в Δ) раз. Недостаток — перерасход памяти: в каждый момент времени некоторое количество (до $\Delta - 1$) ячеек в массиве реально не используется.

1.2.3. Экспоненциальный рост

Следующей идеей будет использовать другую стратегию изменения ёмкости: не «расширяем на Δ единиц», а «расширяем в α раз» ($\alpha > 1$) (рис. 1.4). Если начать с массива ёмкости 1, то получим последовательность ёмкостей $1, \lfloor\alpha\rfloor, \lfloor\alpha^2\rfloor, \lfloor\alpha^3\rfloor, \ldots$. Отметим, что в общем случае число α может быть дробным (например, можно увеличивать размер на каждом шаге в полтора раза). Ёмкость массива — всегда целое число, поэтому в формулах появляется операция округления, которая их заметно усложняет.

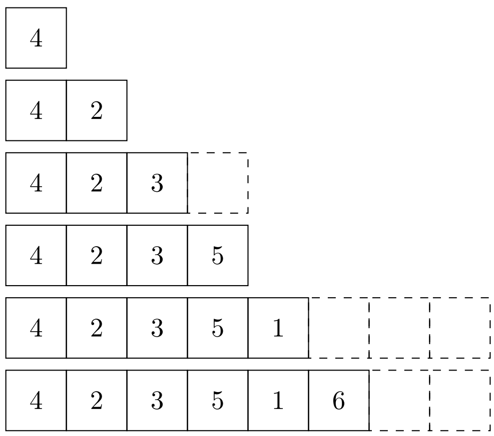

Рис. 1.4. Этапы экспоненциального расширения массива при $\alpha = 2$

Определим число k — количество шагов увеличения ёмкости массива, необходимое для сохранения n элементов. Нетрудно увидеть, что раз выполнить $k - 1$ шаг ещё недостаточно, а k шагов — достаточно, то

$$\lfloor\alpha^{k-1}\rfloor < n \leqslant \lfloor\alpha^k\rfloor.$$

Рассмотрим левую часть неравенства. Исходя из свойства $x - 1 < \lfloor x\rfloor$ операции «пол», имеем

$$\alpha^{k-1} - 1 < \lfloor\alpha^{k-1}\rfloor < n,$$

отсюда получим оценку:

$$\alpha^{k-1} < n + 1.$$

Всего при выполнении этих k реаллокаций нужно будет выполнить

$$T(n) = \sum_{i=1}^{k} \lfloor \alpha^{i-1} \rfloor$$

элементарных операций переноса значений из старого массива в новый. Оценивая слагаемые сверху и пользуясь формулой суммы геометрической прогрессии, имеем:

$$T(n) \leqslant \sum_{i=1}^{k} \alpha^{i-1} = \frac{\alpha^k - 1}{\alpha - 1} \leqslant \frac{\alpha(n+1) - 1}{\alpha - 1} = 1 + \frac{\alpha}{\alpha - 1} \cdot n,$$

или

$$T(n) = \mathrm{O}(n).$$

Таким образом, можно сделать вывод, что при фиксированной константе $\alpha > 1$ общее число операций по перемещению данных в памяти, которые выполняются при последовательном добавлении n элементов, растёт линейно с ростом n.

Пример 1.1. Часто на практике используется значение $\alpha = 2$. Это значит, что ёмкость динамического массива изменяется следующим образом: 1, 2, 4, 8, 16, … .

1.2.4. Амортизированная константа

Итак, если следовать стратегии удвоения размера, на добавление в динамический массив n элементов требуется затратить время $\mathrm{O}(n)$. Значит, каждая вставка выполняется *в среднем* за время $\mathrm{O}(1)$.

На самом деле конкретная операция вставки каждого элемента осуществляется или за константное время (когда в массиве есть свободная ёмкость), или за линейное (когда свободного места нет, выполняется реаллокация). Но усреднённо время вставки одного элемента получается константным. В этом случае говорят, что $\mathrm{O}(1)$ — *амортизированная* оценка для операции вставки.

На практике в системах реального времени такие непредсказуемые задержки при выполнении отдельной операции могут представлять проблему. Если известно примерное число элементов, которые будут в итоге добавлены в массив, можно заранее *зарезервировать* (англ. *reserve*) нужную ёмкость массива и уменьшить число реаллокаций.

1.2.5. Пример реализации

Опишем на псевдокоде класс, который организует динамический массив на основе статического с использованием стратегии удвоения.

```
class DynamicArray:
    def __init__(self):
        self.data = array(1)
        self.size = 0
        self.capacity = 1

    def append(self, x):
        if self.size == self.capacity:
            new_capacity = self.capacity * 2
            new_data = array(new_capacity)

            for i in range(self.capacity):
                new_data[i] = self.data[i]

            self.data = new_data
            self.capacity = new_capacity

        self.data[self.size] = x
        self.size += 1
```

С целью оптимизации память для пустого динамического массива (нулевого размера) можно не выделять, но потребуется этот случай рассмотреть отдельно.

При необходимости можно описать и другие операции: удаление последнего элемента, установка размера массива в фиксированное значение, предварительное резервирование заданной ёмкости и др.

1.2.6. Применение на практике

Динамические массивы очень удобны и широко используются на практике в прикладных задачах. С точки зрения скорости доступа к элементам они эквивалентны статическим массивам. Готовые реализации динамических массивов предоставляются стандартными библиотеками всех основных современных языков программирования.

В языке C++ динамический массив реализован в классе `std::vector`. Значение множителя роста не зафиксировано стандартом языка и различается в зависимости от конкретной реализации (так, в libc++ традиционно применяют число 2, в версии от Microsoft — число 1,5).

При создании программ на Java широко используется класс `ArrayList` (множитель роста 1,5).

В языке программирования Python применяется тип `list`. Если обратиться к реализации CPython 3.7, то можно увидеть, что при расширении действует оригинальная стратегия: старый размер умножается на 1,125, затем к нему прибавляется константа 3 или 6. Получается такая последовательность ёмкостей: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, … .

1.3. СВЯЗНЫЙ СПИСОК

Связный список (англ. *linked list*) — некоторая последовательность элементов, которые связаны друг с другом логически. Логический порядок прохождения элементов определяется с помощью ссылок, при этом он может не совпадать с физическим порядком размещения элементов в памяти компьютера. Доступ к элементам списка осуществляется *последовательно*, т. е. чем дальше в структуре расположен элемент, тем дольше к нему по времени будет осуществляться доступ.

Список состоит из *узлов* (англ. *nodes*). Каждый узел включает две части: информационную (непосредственные данные, принадлежащие элементу) и ссылочную (указатель/ссылка на следующий и/или предыдущий узел).

В *односвязном*, или *однонаправленном связном*, списке (англ. *singly linked list*) каждый узел содержит ссылку на следующий узел (рис. 1.5). Для последнего узла эта ссылка обычно является нулевой. По односвязному списку можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

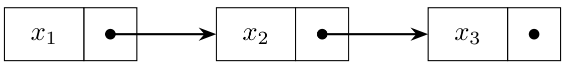

Рис. 1.5. Односвязный список

В *двусвязном*, или *двунаправленном связном*, списке (англ. *doubly linked list*) ссылки в каждом узле указывают на предыдущий и на последующий узел (рис. 1.6). Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом даёт возможность перемещения в обе стороны. В таком списке проще производить удаление и перестановку элементов, так как легко получить доступ ко всем элементам списка, ссылки которых направлены на изменяемый элемент.

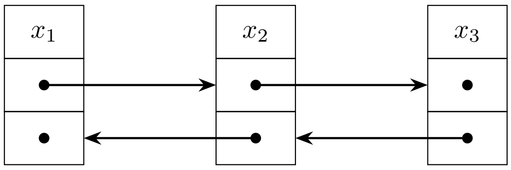

Рис. 1.6. Двусвязный список

При работе со списком вводятся дополнительные ссылки на первый и последний элемент списка. Будем называть их `head` («голова») и `tail` («хвост»).

1.3.1. Введение буферного элемента

Часто для списка вводят *буферный элемент* (англ. *sentinel*), т. е. такой вспомогательный элемент, на который ссылается физически последний элемент списка. На рис. 1.7 показан пример такого списка.

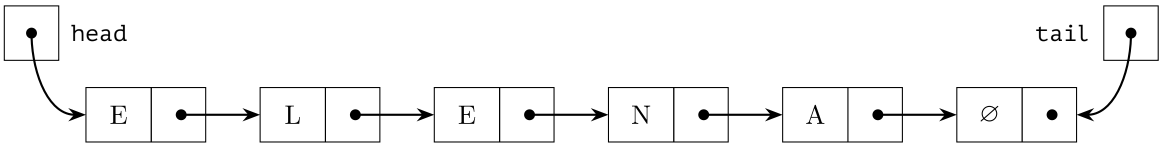

Рис. 1.7. Односвязный список с буферным элементом

Этот приём даёт возможность упростить реализацию структуры данных. Так, теперь ссылки `head` и `tail` всегда ненулевые, для пустого списка они ссылаются на буферный элемент (рис. 1.8).

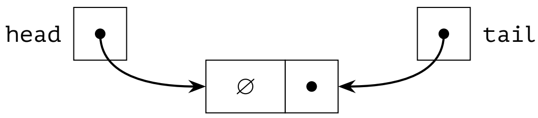

Рис. 1.8. Пустой односвязный список с буферным элементом

Для однонаправленного списка буферный элемент позволяет выполнять операцию удаления элемента за константное время, если удаляемый элемент не нужно искать, а сразу задана ссылка на него. Случай, когда удаляемый элемент стоит в списке последним, проиллюстрирован на рис. 1.9. При удалении узла мы берём узел, следующий за ним, и переносим всю информацию (данные и ссылку) в текущий узел, затем следующий узел удаляем. Затем при необходимости обновляем ссылку `tail`, которая всегда указывает на буферный элемент.

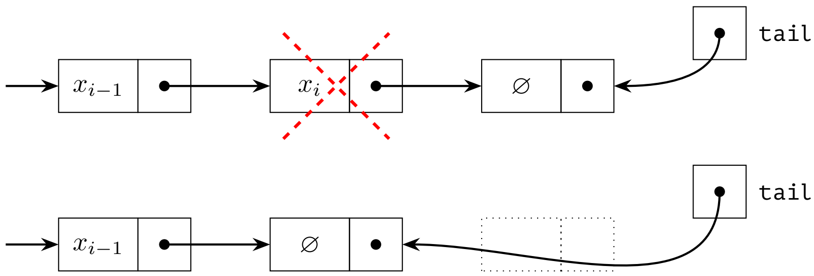

Рис. 1.9. Удаление элемента x_i из односвязного списка с буферным элементом

1.3.2. Реализация на массивах

Чаще всего узлы списка размещают в динамической памяти, при этом в качестве значений ссылок используются адреса узлов. Альтернативный способ — использовать для хранения информации обычные массивы, тогда в качестве значений ссылок будут выступать индексы (порядковые номера элементов массива). На рис. 1.10 приведём пример представления списка, представленного на рис. 1.7 (без буферного элемента), в виде двух массивов.

`i`	0	1	2	3	4
`list[i]`	A	E	L	N	E
`next[i]`	-1	2	4	0	3

Рис. 1.10. Представление связного списка на базе двух массивов

В данном представлении `list[i]` содержит значение элемента списка (латинскую букву), а `next[i]` определяет позицию (индекс) следующего за ним элемента. Индекс первого элемента списка `head = 1`, а индекс последнего элемента списка — `tail = 0`. Иногда для представления списка вместо двух отдельных массивов используют один массив, элементами которого являются структуры, состоящие из полей.

1.3.3. Сравнение связных списков и динамических массивов

Связные списки имеют несколько преимуществ перед динамическими массивами.

• **Быстрая вставка и удаление.** Операции вставки в конкретное место списка и удаления определённого элемента списка выполняются за $O(1)$ при условии, что на вход даётся ссылка на узел (идущий перед точкой вставки или предшествующий узлу, который будет удалён). Заметим, что если такая ссылка не предоставлена, то операции работают за $O(n)$. В то же время вставка в произвольное место динамического массива требует перемещения в среднем половины элементов, а в худшем случае — всех элементов. Хотя можно «удалить» элемент из массива за константное время, пометив его ячейку как «свободную», это вызовет фрагментацию, которая будет негативно влиять на скорость прохода по массиву.

- **Нет реаллокаций.** В связный список может быть вставлено произвольное количество элементов, ограниченное только доступной памятью. Ранее вставленные элементы никуда не перемещаются, их адреса в памяти не меняются. В динамических массивах при вставке иногда происходит реаллокация; это дорогостоящая операция, которая может оказаться невозможной при высокой фрагментированности памяти (не удастся найти непрерывный блок памяти нужного размера, хотя небольшие свободные блоки будут доступны в достаточном количестве).

С другой стороны, у списков есть и существенные недостатки.

- **Нет произвольного доступа.** Динамические массивы обеспечивают произвольный доступ к любому элементу по индексу за константное время, в то время как связные списки допускают лишь последовательный доступ к элементам. По односвязному списку можно пройти только в одном направлении. Это делает связные списки непригодными для алгоритмов, в которых нужно быстро получать элемент по его индексу (например, к такому типу относятся многие алгоритмы сортировки).
- **Медленный последовательный доступ.** Линейный проход по элементам массива на реальных машинах выполняется гораздо быстрее, чем по элементам связного списка. Это связано с тем, что элементы массива хранятся в памяти один за одним, поэтому не требуется выполнять на каждом шаге переход по указателю. За счёт локальности хранения данных в массиве эффективно работает кеширование на уровне процессора.
- **Перерасход памяти.** На хранение ссылок в узлах связного списка расходуется дополнительная память. Эта проблема особенно актуальна, если полезные данные имеют небольшой размер. Накладные расходы на хранение ссылок могут превышать размер данных в восемь или более раз.

1.3.4. Применение на практике

В реальной практике прикладного программирования связные списки в чистом виде используются крайне редко. Динамические массивы обычно оказываются удобнее и эффективнее.

Так, если в задаче требуется хранить граф в виде списков смежности (для каждой вершины храним набор вершин, смежных с ней), эти абстрактные «списки смежности» не обязательно размещать именно в связных списках. С тем же успехом можно применить динамические массивы.

Однако есть ряд алгоритмов, при разработке которых не обойтись без классических связных списков (например, к ним относятся многие механизмы кеширования). Связные списки находят применение в системном программировании: в ядре операционной системы в связных списках хранятся активные процессы, потоки и другие динамические объекты, в менеджерах памяти (аллокаторах) в связных списках хранятся готовые к использованию блоки свободной памяти, и т. д.

Двусвязный список представлен в стандартной библиотеке языка C++ классом `std::list`, в библиотеке языка Java — классом `LinkedList`. В языке Python встроенной реализации нет.

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1. Алгоритмы: построение и анализ / Т. Кормен [и др.]. — М. : Вильямс, 2005. — 1296 с.
2. *Котов В. М.*, *Мельников О. И.* Информатика. Методы алгоритмизации : учеб. пособие для 10 – 11 кл. общеобразоват. шк. с углубл. изучением информатики. — Минск : Нар. асвета, 2000. — 221 с.
3. *Котов В. М.*, *Соболевская Е. П.*, *Толстиков А. А.* Алгоритмы и структуры данных : учеб. пособие. — Минск : БГУ, 2011. — 267 с. — (Классическое университетское издание).
4. Сборник задач по теории алгоритмов : учеб.-метод. пособие/ В. М. Котов [и др.]. — Минск : БГУ, 2017. — 183 с.
5. Теория алгоритмов : учеб. пособие / П. А. Иржавский [и др.]. — Минск : БГУ, 2013. — 159 с.
6. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Опыт использования образовательной платформы Insight Runner на факультете прикладной математики и информатики Белорусского государственного университета // Роль университетского образования и науки в современном обществе : материалы междунар. науч. конф., Минск, 26–27 февр. 2019 г. / Белорус. гос. ун-т ; редкол.: А. Д. Король (пред.) [и др.]. — Минск : БГУ, 2019. — С. 263-267.
7. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Методика преподавания дисциплин по теории алгоритмов с использованием образовательной платформы iRunner // Судьбы классического университета: национальный контекст и мировые тренды [Электронный ресурс] : материалы XIII Респ. междисциплинар. науч.-теорет. семинара «Инновационные стратегии в современной социальной философии» и междисциплинар. летней школы молодых ученых «Экология культуры», Минск, 9 апр. 2019 г. / Белорус. гос. ун-т ; сост.: В. В. Анохина, В. С. Сайганова ; редкол.: А. И. Зеленков (отв. ред.) [и др.] — С. 346–355.

СОДЕРЖАНИЕ

Учебное издание

Соболь Сергей Александрович
Вильчевский Константин Юрьевич
Котов Владимир Михайлович и др.

СБОРНИК ЗАДАЧ ПО ТЕОРИИ АЛГОРИТМОВ. СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие

Редактор *Х. Х. ХХХХХХХ*
Художник обложки *С. А. Соболь*
Технический редактор *Х. Х. ХХХХХХХ*
Компьютерная вёрстка *С. А. Соболя*
Корректор *Х. Х. ХХХХХХ*

Подписано в печать 29.02.2020. Формат 60×84/16. Бумага офсетная.
Печать офсетная. Усл. печ. л. 10,69. Уч.-изд. л. 9,6.
Тираж 150 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 1/270 от 03.04.2014.
Пр. Независимости, 4, 220030, Минск.

Республиканское унитарное предприятие
«Издательский центр Белорусского государственного университета».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 2/63 от 19.03.2014.
Ул. Красноармейская, 6, 220030, Минск.