В. А. МАТЬЯШ
С. А. РОГАЧЕВ

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ

В. А. Матьяш, С. А. Рогачев

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие

ГУАП
Санкт-Петербург
2021

УДК 004.421.6(075)
ББК 32.973я73
М35

Рецензенты:
кандидат технических наук, доцент *В. И. Исаков*;
кандидат технических наук *С. А. Потрясаев*

Утверждено
редакционно-издательским советом университета
в качестве учебного пособия
Протокол № 6 от 1 ноября 2020 г.

Матьяш, В. А.

М35 Алгоритмы и структуры данных: учеб. пособие / В. А. Матьяш, С. А. Рогачев. – СПб.: ГУАП, 2021. – 71 с.

ISBN 978-5-8088-1553-7

Рассматриваются понятие структур данных, анализа сложности алгоритмов, линейные и циклические списки, стеки и очереди, хеширование данных, сортировка данных, алгоритмы на деревьях и алгоритмы поиска слова в тексте. Приведены задания и требования, а также даны пояснения к порядку выполнения курсового проекта, что позволяет студенту подготовиться к курсовому проектированию, при необходимости изучить или актуализировать в памяти соответствующий раздел дисциплины.

Предназначено для выполнения курсового проекта по дисциплине «Структуры и алгоритмы и обработки данных» студентами различных форм обучения, проходящих подготовку по направлениям бакалавриата 01.03.02 (Прикладная математика и информатика); 02.03.03 (Математическое обеспечение и администрирование информационных систем); 09.03.04 (Программная инженерия).

УДК 004.421.6(075)
ББК 32.973я73

ISBN 978-5-8088-1553-7

ПРЕДИСЛОВИЕ

В пособии описаны алгоритмы и структуры данных, которые базово используются в наше время при разработке программного обеспечения.

Понимание описанного в учебном пособии материала позволит осуществлять выбор наиболее оптимальных путей решения прикладных задач, возникающих при разработке и проектировании программного обеспечения различного назначения.

Учебное пособие состоит из девяти глав. В первой рассматриваются основные понятия алгоритмов и структур данных, а также подходы к анализу их сложности. Во второй рассматривается алгоритм хеширования данных. Третья посвящена алгоритмам внутренней сортировки. В четвертой рассматриваются такие структуры данных, как списки. В пятой даются пояснения, что из себя представляют структуры данных «стек» и «очередь». Шестая содержит информацию о деревьях поиска как частный случай орграфов. Седьмая посвящена алгоритмам поиска в тексте. Восьмая содержит задание на курсовое проектирование, порядок его выполнения, содержание отчетных материалов и общие рекомендации по выполнению. В девятой приведен и описан перечень предметных областей, которые могут быть выбраны по заданию.

Материал учебного пособия базируется на следующих дисциплинах: «Информатика», «Основы программирования», «Дискретная математика».

ВВЕДЕНИЕ

В настоящее время электронным вычислительным машинам (ЭВМ) приходится не только считывать и выполнять определенные алгоритмы, но и хранить значительные объемы информации. При этом к хранимой информации нужно иметь быстрый доступ. Она в некотором смысле представляет собой абстракцию того или иного фрагмента реального мира и состоит из определенного множества данных, относящихся к какому-либо объекту или предметной области.

Современная ЭВМ хранит и обрабатывает только один вид данных (двоичные цифры) и может работать с ними только в соответствии с алгоритмами, которые определяются системой команд центрального процессора. С помощью компьютера решаются задачи, которые крайне редко выражаются в виде битов и байтов. Зачастую данные имеют форму чисел, текстов, символов и более сложных структур типа множеств, массивов, списков и деревьев [1].

В то время как физическое представление данных имеет достаточно простую организацию, которая хорошо понятна ЭВМ, для человека взаимодействовать с таким представлением не очень удобно. Физическая структура данных отражает представление данных в памяти ЭВМ.

Рассмотрение структур данных без учета представления их в памяти ЭВМ обычно называют абстрактным, как и сами структуры. Оперирование абстрактными структурами данных позволяет программисту изменять детали логики работы программы независимо от подробностей того, как эти структуры будут реализованы.

Именно об абстрактных структурах данных и алгоритмах работы с ними пойдет речь в этом пособии.

1. АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

1.1. Структуры данных

Структура данных – это в общем понимании некий набор элементов и множество связей между ними. Давая такое определение, можно охватить все существующие подходы к структуризации данных, но в каждой отдельной задаче могут использоваться различные подходы. Поэтому имеет смысл ввести дополнительную классификацию структур данных, которая соответствует различным аспектам их рассмотрения (рис. 1.1) [1–3].

Структура данных может быть абстрактной или логической, это значит, что она рассматривается без учета ее конкретного представления в машинной памяти. Понятие «физическая структура данных» будет в свою очередь неразрывно связано с физическим представлением данных в памяти машины. Существуют процедуры, которые осуществляют отображение логической структуры в физическую, и наоборот. Это связано с различиями между логическим и физическим представлениями структур в вычислительной системе [1].

В качестве примера можно привести доступ к элементу двумерного массива, который на логическом уровне реализуется указанием номеров строки и столбца в таблице, на пересечении координат которых расположен соответствующий элемент. Но на физическом уровне доступ к элементу массива осуществляется с помощью функции адресации, которая преобразует номера строки и столбца в адрес элемента массива (при известном начальном адресе массива в машинной памяти). Таким образом, для каждой структуры данных существует ее логическое (абстрактное) и физическое представление, и, конечно, определена некая совокупностью операций на этих двух уровнях представления структуры.

Как правило, когда упоминается та или иная структура данных, речь идет о ее логическом представлении. Это происходит потому, что физическое представление обычно скрыто от программиста и в зависимости от типа ЭВМ, может сильно отличаться. Объем машинной памяти всегда ограничен, поэтому способ реализации физической структуры должен учитывать проблемы распределения и управления памятью [4].

Структуры данных, которые могут применяться в алгоритмах, могут быть как простыми, так и чрезвычайно сложными. Поэтому выбор оптимального способа представления данных является важнейшей задачей при разработке программных и может сильно

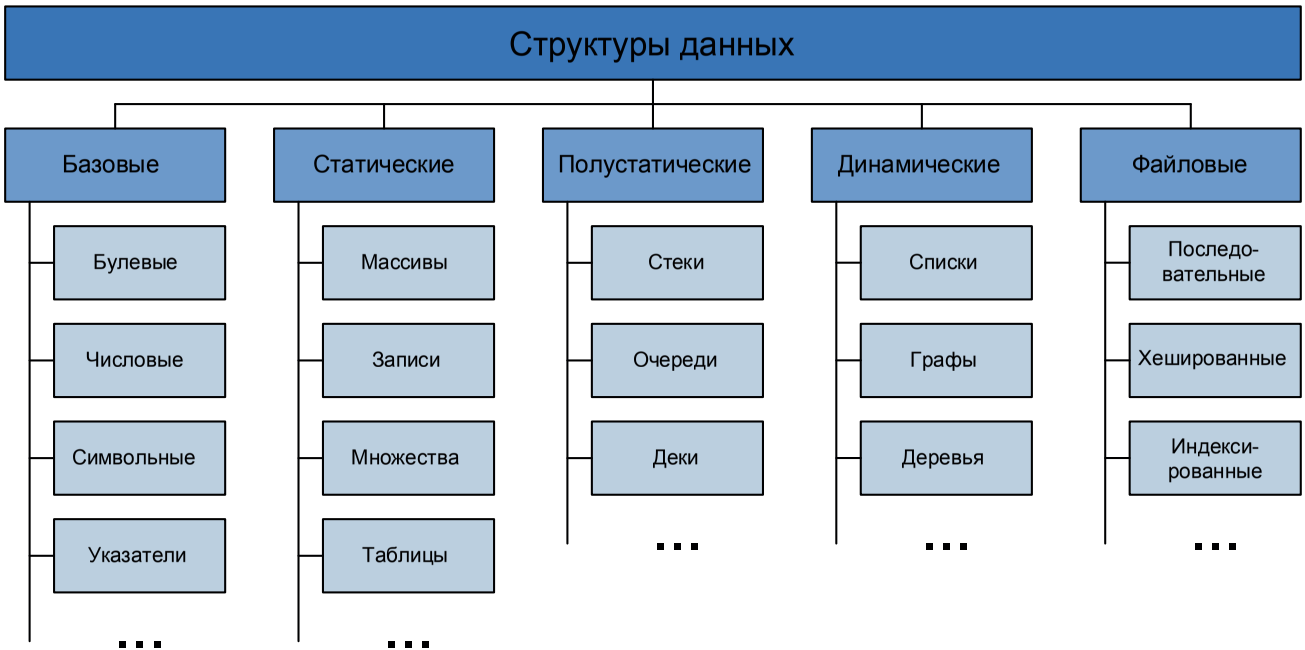


Рис. 1.1. Классификация данных по признаку изменчивости

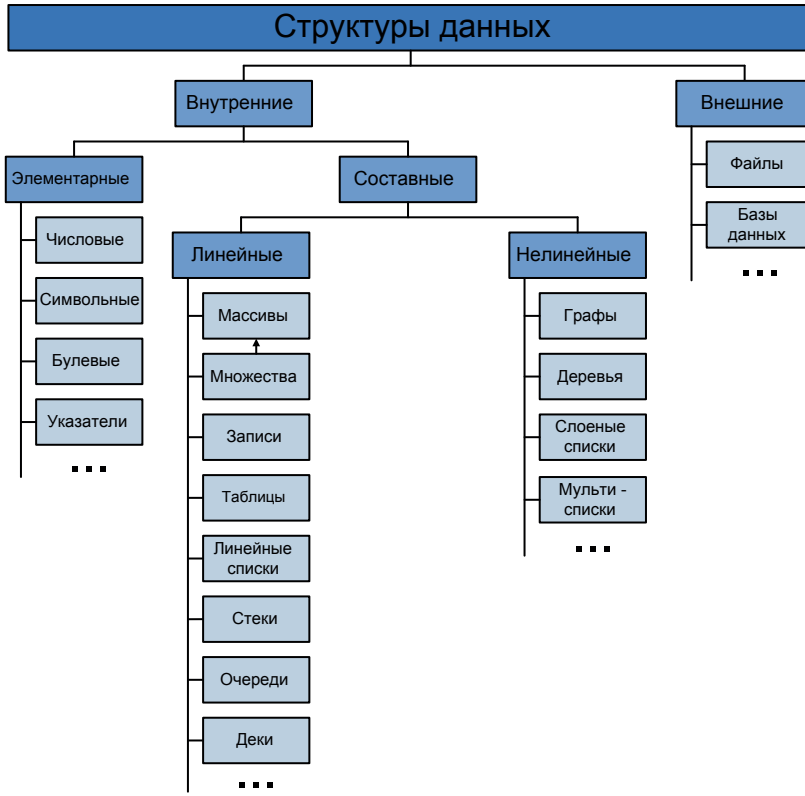


Рис. 1.2. Классификация структур данных на основе их размещения в памяти

влиять на производительность программы. На рис. 1.2 приведена классификация структур данных в зависимости от типа размещения в памяти [1].

1.2. Анализ сложности алгоритмов

Алгоритм – это точная конечная последовательность действий, которая определяет вычислительный процесс, ведущий от варьируемых начальных данных к конечному результату [1].

Изначально при решении определенной задачи выбор алгоритма вызывает прогнозируемые трудности, поскольку любой алгоритм

должен удовлетворять двум иногда противоречивым требованиям, он должен:

1) быть простым для понимания человека и написания программного кода, а также дальнейшей отладки;

2) максимально эффективно использовать вычислительные ресурсы ЭВМ и по возможности быстро выполняться.

Первое требование является особенно важным, когда программа, реализующая определенный алгоритм, должна выполняться некоторое (небольшое) количество раз. В таких случаях написанная программа оптимизируется с точки зрения трудозатрат по ее написанию, а не по выполнению. Некоторые задачи при решении требуют достаточно больших вычислительных ресурсов, поэтому в таких задачах оптимизация производится с точки зрения времени выполнения, а трудозатраты на разработку отходят на второй план. В этих случаях более сложный и запутанный алгоритм может стать лучшей альтернативой, так как всегда есть надежда, что итоговая программа будет выполняться существенно быстрее. В итоге программист (разработчик), прежде чем принимать решение об использовании того или иного алгоритма, должен квалифицированно оценить его сложность и эффективность [1, 5–7].

Под сложностью алгоритма понимается величина, которая отражает порядок объема необходимого ресурса (времени или памяти) в зависимости от размерности конкретной задачи.

При расчете сложности алгоритмов и структур данных обычно оперируют двумя метриками: количеством операций, необходимых для получения конечного результата (временная сложность), и объемом вычислительных ресурсов (памяти), который будет использоваться при реализации алгоритма (пространственная сложность). Рассмотрение оценок сложности алгоритмов будет происходить на примере временной сложности, так как пространственная сложность оценивается аналогично [6, 8].

Чаще всего, когда анализируют сложность алгоритма, говорят об анализе времени, которое необходимо программе для обработки очень большого объема информации (входных данных). Такой анализ называют асимптотическим [6, 9].

Важным значением при асимптотическом анализе является порядок роста. Он описывает то, как будет расти сложность алгоритма при увеличении размера входных данных. Чаще всего он представлен в виде O-символики (от немецкого «*Ordnung*» – порядок): $O(f(x))$, где $f(x)$ – формула, которая выражает сложность алгоритма. Как правило, будет присутствовать переменная n, которой обычно

обозначают размер входных данных. Ниже будут рассмотрены наиболее часто встречающиеся порядки роста.

1.2.1. Константный порядок роста

Когда сложность алгоритма обозначается *O(1)*, это значит, что теоретическая сложность алгоритма не зависит от размера входных данных. Единица не означает, что в алгоритм работает быстро и для его выполнения потребуется совершить только одну операцию. Такое обозначение говорит только об одном: быстродействие алгоритма не зависит от объема входных данных [1, 6, 8].

1.2.2. Линейный порядок роста

Порядок роста, обозначающийся *O(n),* означает, что сложность алгоритма будет линейно расти в зависимости от увеличения объема входных данных. Например, если алгоритм, имеющий линейный порядок роста, обрабатывает один элемент за пять миллисекунд, то разработчик может предполагать, что тысячу элементов он обработает за пять секунд.

Подобные алгоритмы легко узнать по наличию цикла, количество итераций которого зависит от элементов входного массива данных [1, 6, 8].

1.2.3. Логарифмический порядок роста

Логарифмический порядок роста *O(log n)* показывает логарифмическую зависимость времени выполнения алгоритма от размера входных данных. Примечание: в анализе алгоритмов по умолчанию используется логарифм по основанию два. Такую сложность, как правило, имеют алгоритмы, которые работают по принципу «деления пополам» [1, 6, 8].

1.2.4. Линеарифмический порядок роста

Линеарифметический (еще иногда его называют линейно-логарифмический) алгоритм имеет порядок роста *O(n*log n)*. По своей сути такие алгоритмы объединяют в себе порядки роста линейных и логарифмических алгоритмов. Большинство алгоритмов, работающих по принципу «разделяй и властвуй», попадают в эту категорию. Пример такого алгоритма – быстрая сортировка (Хоара), которая будет рассмотрена далее [1, 6, 8].

1.2.5. Квадратичный порядок роста

Время работы алгоритма с квадратичным порядком роста $O(n^2)$ зависит от квадрата размера входного массива. Если алгоритм имеет квадратичную сложность, это может быть поводом, чтобы пересмотреть выбранные пути решения поставленной задачи. Главная проблема таких алгоритмов – это их плохая масштабируемость. Например, если одна операция требует миллисекунду для выполнения, при использовании алгоритма с квадратичным порядком роста входной массив из тысячи элементов будет обрабатываться почти полчаса.

Классическим примером алгоритма с квадратичной сложностью может служить пузырьковая сортировка [1, 6, 8].

2. ХЕШИРОВАНИЕ ДАННЫХ

В таблицах для ускорения доступа к данным часто используют предварительное упорядочивание по значению ключей [1].

Чтобы найти данные, которые упорядочены по ключам, можно использовать, например, метод половинного деления, который достаточно сильно сократит время выполнения алгоритма по сравнению с последовательным поиском. При этом добавление новых данных в упорядоченную таблицу потребует больших временных затрат, так как будет необходимо заново упорядочить таблицу. Как уже говорилось ранее, не всегда время, которое потребуется на повторное упорядочивание таблицы, компенсирует выигрыш от быстрого поиска элемента.

Хеширование – это метод случайного упорядочивания элементов, который обеспечивает сокращение времени доступа к данным. При таком алгоритме данные также организуются в виде таблицы (рис. 2.1), но место хранения данных в таблице «вычисляется» при помощи хеш-функции h по значению ключа [1, 10, 11].

Идеальной хеш-функцией будет такая функция, которая для любых двух разных ключей выдает различные адреса в хеш-таблице.

Эта функция может существовать только в том случае, если пространство потенциальных адресов в хеш-таблице больше, чем количество различных ключей. Такая организация данных обычно называется «совершенным хешированием». На практике размер хеш-таблицы во много раз меньше, чем количество возможных ключей, поэтому чаще всего используют хеш-функции, которые не являются идеальными.

Ситуации, когда для двух различных ключей функция вычисляет один и тот же адрес, называются «коллизиями», а ключи – «синонимами» [1, 10, 11].

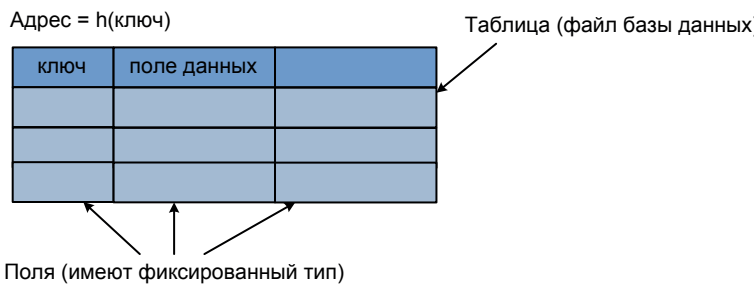


Рис. 2.1. Хеш-таблица

2.1. Методы разрешения коллизий

При возникновении коллизий используют различные алгоритмы, которые называются методами решения коллизий. Как правило, такие алгоритмы сводятся к методу «цепочек» и методам «повторного хеширования» (рис. 2.2).

Метод цепочек – это специфичный способ организации хеш-таблицы, когда в ячейках таблицы хранятся указатели на списки, которые называются «цепочками переполнения». При заполнении хеш-таблицы, когда возникает коллизия в список, указатель на который хранится по полученному адресу, добавляется еще один элемент (рис. 2.3) [11].

Для поиска элемента в хеш-таблице, организованной с использованием цепочек переполнения, нужно вначале вычислить адрес по значению ключа, а потом выполнить последовательный поиск в списке, указатель на который хранится по вычисленному адресу хеш-таблицы.

Рис. 2.2. Разновидности методов разрешений коллизий

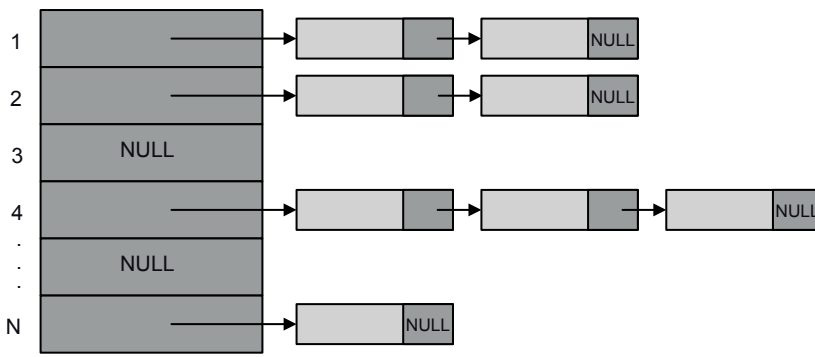


Рис. 2.3. Разрешение коллизий при добавлении элементов методом цепочек в открытом хешировании

Удаление элемента из хеш-таблицы, коллизии в которой обработаны методом цепочек, происходит с использованием двух алгоритмов: поиска элемента в хеш-таблице и удаления элемента из списка.

В отличие от открытого хеширования, при закрытом в хеш-таблице хранятся непосредственно сами элементы. Поэтому в каждой ячейке (сегменте) таблицы может храниться только один элемент. При использовании закрытого хеширования для решения коллизий используются алгоритмы повторного хеширования. При повторном хешировании, если возникает ситуация, когда сегмент с номером $h(x)$ (где x – элемент, который необходимо записать) занят, то в соответствии с выбранным методом повторного хеширования выбирается другой номер сегмента, куда можно поместить элемент x. Сегменты хеш-таблицы последовательно проверяются до тех пор, пока не будет найден свободный. Если таблица заполнена и свободных сегментов нет, то элемент x добавить нельзя. Ситуации, в которых адрес следующего сегмента может выйти за пределы пространства адресов хеш-таблицы, рассмотрены в пункте 2.2.

Существует несколько методов повторного хеширования, то есть определения местоположений $h(x)$, $h1(x)$, $h2(x)$, ... :

– линейное опробование (рис. 2.4);

– квадратичное опробование (рис. 2.5);

– двойное хеширование (рис. 2.6).

Под методом линейного опробования понимается последовательный перебор адресов сегментов таблицы с фиксированным шагом. Величина шага принципиального значения не имеет [10, 11].

В методе квадратичного опробования шаг перебора сегментов таблицы будет нелинейно зависеть от номера попытки записать элемент. За счет нелинейности изменения шага уменьшается число

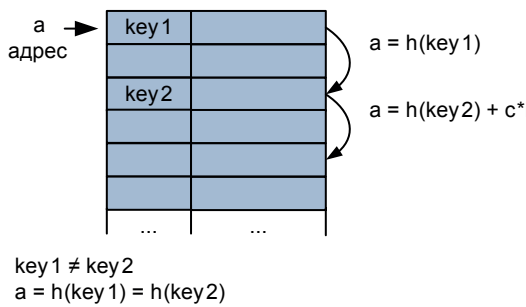


Рис. 2.4. Разрешение коллизий при добавлении элементов методом линейного опробования

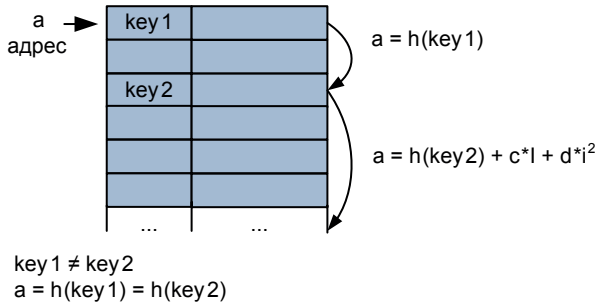


Рис. 2.5. Разрешение коллизий при добавлении элементов методом квадратичного опробования

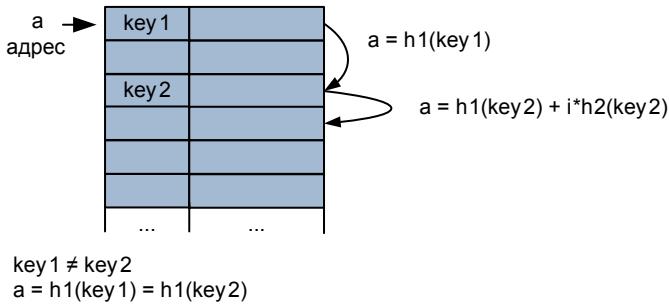


Рис. 2.6 Разрешение коллизий при добавлении элементов методом двойного хеширования

попыток при большом количестве ключей-синонимов. Минусом данного метода является то, что из-за квадратичной зависимости изменения величины шага очередная попытка может выйти за адресное пространство таблицы (если таблица небольшого размера, то таких попыток может потребоваться совсем немного) [10, 11].

Метод двойного хеширования основан на том, что в зависимости от значения, которое мы хотим записать в таблицу, высчитывается и величина шага, который будет использован при разрешении коллизий. Величина шага рассчитывается с помощью второй хеш-функции (отличной от основной) [10, 11].

Алгоритм вставки (добавления) и поиска элемента в хеш-таблице Table для метода линейного опробования

Вставка элемента:

1. $i = 0$
2. $a = h(key) + i*c$

3. Если *Table(a)* свободно, то *Table(a) = key*, записать элемент, «стоп элемент добавлен»

4. *i = i + 1*, переход к шагу 2.

Поиск элемента:

1. *i = 0*

2. *a = h(key) + i*c*

3. Если *Table(a) = key*, то «стоп элемент найден»

4. Если *Table(a)* свободно, то «стоп элемент не найден»

5. *i = i + 1*, переход к шагу 2.

По аналогии описываются алгоритмы поиска и добавления элементов при использовании методов квадратичного опробования и двойного хеширования. По сути, методы решения коллизий в закрытом хешировании отличаются только формулой вычисления следующего адреса [10].

Удаление элемента из хеш-таблицы не является обратной процедурой добавлению, поэтому ее стоит рассмотреть отдельно.

Сложность состоит в том, что изначально каждый сегмент хеш-таблицы может быть либо свободным, либо занятым. При удалении элемента, по логике, нужно объявить, что сегмент стал свободен. Но после такого удаления алгоритм поиска может начать работать некорректно. Дело в том, что удаляемый ключ удаляемого элемента мог иметь ключи-синонимы. В таком случае при поиске элемента, вызывавшего коллизию с удаленным, всегда будет получаться отрицательный результат (ключ не найден), поскольку алгоритм поиска останавливается при обнаружении пустой ячейки.

Самый простой способ скорректировать эту ситуацию – осуществлять поиск ключа до тех пор, пока не закончится пространство адресов хеш-таблицы. Однако тогда не будет никакого выигрыша от ускорения доступа к данным [1].

Также можно при удалении элемента находить все ключи-синонимы и перераспределять их в хеш-таблице. Минусом данного подхода будут являться большие временные затраты при перераспределении ключей (особенно в том случае, если имеется большое количество ключей-синонимов).

Есть подход, который лишен вышеперечисленных недостатков. Его суть заключается в том, что в список возможных состояний сегментов таблицы вводится дополнительное состояние «удалено». Данное состояние при поиске будет интерпретироваться как «занято», а при добавлении записи как «свободно» [1].

Алгоритмы вставки (добавления), поиска и удаления элемента из хеш-таблицы, в которой три состояния

Вставка элемента:

1. $i = 0$
2. $a = h(key) + i*c$
3. Если *Table(a)* свободно или *Table(a)* удалено, то $Table(a) = key$, записать элемент, «стоп элемент добавлен»
4. $i = i + 1$, переход к шагу 2.

Удаление элемента:

1. $i = 0$
2. $a = h(key) + i*c$
3. Если $Table(a) = key$, то *Table(a)* = удалено, «стоп элемент удален»
4. Если *Table(a)* свободно, то «стоп элемент не найден»
5. $i = i + 1$, переход к шагу 2.

Поиск элемента:

1. $i = 0$
2. $a = h(key) + i*c$
3. Если $Table(a) = key$, то «стоп элемент найден»
4. Если *Table(a)* свободно, то «стоп элемент не найден»
5. $i = i + 1$, переход к шагу 2.

При наличии трех возможных состояний хеш-таблицы алгоритм поиска практически не будет отличаться от описанного выше. Разница лишь в том, что нужно отличать свободные и удаленные элементы. Реализовать это возможно, например, выделив дополнительный бит для ключевого поля. В качестве альтернативы можно предложить введение дополнительного поля, фиксации состояния сегмента.

2.2. Переполнение таблицы и рехеширование

По мере заполнения хеш-таблицы почти всегда будут происходить коллизии. Рано или поздно адрес следующего сегмента выйдет за пределы пространства адресов хеш-таблицы. Один из вариантов избавления от такой ситуации – увеличение размера хеш-таблицы (при этом нужно, чтобы ее размер был больше диапазона адресов, которые выдает хеш-функция) [1, 11, 12].

Такой подход приведет к сокращению числа коллизий, но в то же время он влечет избыточное расходование памяти. Даже если таблица будет увеличена в разы по сравнению с диапазоном значений хеш-функции, это не означает, что при очередной попытке записи

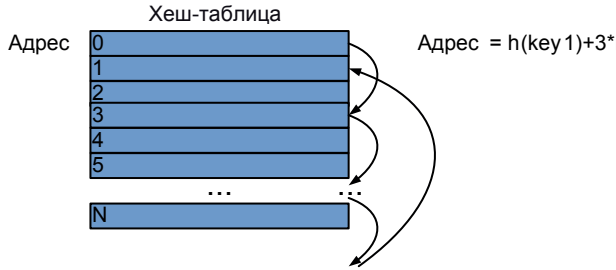


Рис. 2.7. Циклический переход к началу таблицы

мы не выйдем за пространство адресов. Для решения этой проблемы используют циклический переход (рис. 2.7) [1, 12].

Идея заключается в том, что при вычислении адреса очередного элемента диапазон адресов, выдаваемый хеш-функцией, ограничивается размером таблицы (реализуется это как остаток от целочисленного деления адреса на длину таблицы *N)*. Алгоритм будет выглядеть следующим образом:

Вставка элемента:

1. $i = 0$

2. $a = (h(key) + c*i) \bmod N$

3. Если *Table(a)* свободно или *Table(a)* удалено, то *Table(a) = key*, записать элемент, «стоп элемент добавлен»

4. $i = i + 1$, переход к шагу 2.

При рассуждениях на тему выхода за пределы адресного пространства нужно учитывать также два фактора: степень заполненности таблицы и удачный выбор хеш-функции. В первом случае коллизии возникают достаточно часто из-за большого количества занятых сегментов, как следствие, выполняется большое количество циклических переходов. Во втором случае происходит практически тоже самое, но связано это с тем что хеш-функция часто выдает в качестве результатов одни и те же адреса. При полном заполнении хеш-таблицы возможно зацикливание программы, и такие ситуации нужно предусматривать и обрабатывать. Когда используют хеш-таблицы, стараются отслеживать степень заполненности и избегают плотного заполнения (часто длину таблицы выбирают в два раза больше, чем количество потенциальных записей).

При решении практических задач заранее предсказать количество записей не всегда возможно, поэтому при использовании хеш-таблиц стоит определять процедуру рехеширования. Рехеши-

рование предполагает, что изменится размер таблицы, сама хеш-функция, и все данные буду перезаписаны с использованием новой хеш-функции [1, 12].

Для отслеживания плотности заполнения таблицы обычно используют косвенную оценку – количество возникших коллизий при добавлении элемента. Можно установить некий порог количества коллизий, после которого выполняется процедура рехеширования. Одновременно такой подход обрабатывает ситуации с возможным зацикливанием алгоритма в случае повторного просмотра элементов таблицы [1, 10].

Алгоритм вставки элемента, который реализует вышеописанный подход

Вставка элемента:

1. $i = 0$

2. $a = (h(key) + c*i) \bmod N$

3. Если *Table(a)* свободно или *Table(a)* удалено, то $Table(a) = key$, записать элемент, «стоп элемент добавлен»

4. Если $i > ReHash$, то «стоп требуется рехеширование»

5. $i = i + 1$, переход к шагу 2.

В данном описании *ReHash* – пороговое значение, если количество итераций превысит это значение, выполняется рехеширование.

2.3. Оценка качества хеш-функции

Выше уже упоминалось, что на скорость работы алгоритма хеширования сильно влияет выбор хеш-функции. Хорошая хеш-функция обычно характеризуется следующими свойствами:

– обеспечивает равномерное распределение ключей по пространству адресов;

– порождает как можно меньше коллизий для заданного пространства адресов;

– явно не отображает связь между значениями ключей и адресов;

– должна быть быстрой и простой для вычисления [1].

Если хеш-функция выбрана удачно, то хеш-таблица будет заполняться более равномерно, количество коллизий не будет велико, соответственно, операции поиска, добавления и удаления элементов будут выполняться за меньшее время. Можно провести имитационное моделирование для предварительной оценки выбранной хеш-функции. Для этого создается массив, длина которого соответствует длине хеш-таблицы. Далее можно сгенерировать большое количество случайных ключей (обычно их количество превышает раз-

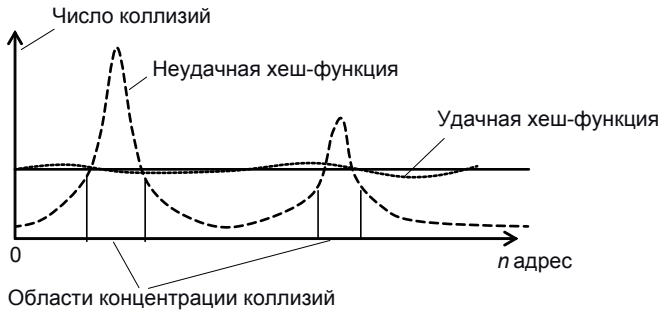


Рис. 2.8 Распределение коллизий в адресном пространстве таблицы

мер таблицы в несколько раз). В ячейках массива ведется запись, какое количество раз данный адрес был вычислен хеш-функцией.

В результате на основе полученного заполненного массива можно построить график распределения значений хеш-функции (рис. 2.8) [1, 12].

Если размер хеш-таблицы достаточно большой, то можно объединить адреса в группы (например, вся таблица разбивается на 100 частей, отслеживается попадание адреса в каждую часть). Возникновение серьезных неравномерностей говорит о высокой вероятности коллизий в отдельных местах таблицы. Конечно, такой подход имеет некоторое допущения и недостатки (например, странности при генерации случайных ключей), однако позволяет предварительно оценить качество хеш-функции и избежать грубых ошибок при ее построении [1, 12].

Описанная оценка будет более точной, если генерируемые ключи будут максимально соответствовать реальным значениям, которые будут храниться в хеш-таблице. Для ключей, состоящих из символов, важно добиться соответствия генерируемых кодов символов тем кодам символов, которые имеются в реальном ключе. Для этого стоит отследить, какие конкретно символы могут быть использованы в ключе. Если ключ представляет собой фамилию на русском языке, то при генерации будет использоваться только код символов, соответствующих русским буквам. При этом часто имеет смысл отслеживать регистр символов (фамилия, имя, отчество начинаются с большой буквы). Резюмируя, перед тем как составлять хеш-функцию, следует всегда оценить свойства ключей, которые она будет обрабатывать.

3. АЛГОРИТМЫ СОРТИРОВКИ

Алгоритм сортировки (далее – сортировка) – это алгоритм упорядочивания элементов некоторого множества по какому-либо признаку. Как правило, для этого множества определены отношения: «меньше», «больше», «равно» и подобные. Чаще всего под использованием сортировки понимают упорядочение элементов по возрастанию или убыванию. Поскольку данные могут храниться в различных структурах данных, с которыми определены различные виды допустимых операций, то и алгоритмы для каждой структуры могут иметь отличия. Если в сортируемой структуре данных имеются элементы с одинаковыми значениями, то в результате сортировки эти элементы будут находиться рядом, но в любом порядке. Однако в некоторых случаях требуется сохранить изначальный порядок элементов [1].

Как правило, элементы структуры данных имеют несколько полей, которые хранят значения, поэтому в случае, когда требуется отсортировать такую структуру, выбирают какое-то одно поле, называемое ключом сортировки, которое будет служить критерием порядка [2, 12].

К основным характеристикам алгоритмов сортировки обычно относят следующие [1]:

1) временная сложность (вычислительная сложность) – основной параметр любого алгоритма, характеризующий его быстродействие;

2) пространственная сложность алгоритма определяет количество памяти, которое потребуется занять для его работы, в зависимости от объема входящих данных. При оценке характеристики не учитывается место, которое занимает исходная структура данных. Если алгоритм сортировки не потребляет дополнительной памяти, он относят к сортировкам на месте;

3) устойчивость (стабильность) – характеристика, отражающая сохранение взаимного расположения исходных элементов, что иногда бывает необходимым. Если сортировка является устойчивой, то она не меняет взаимного расположения элементов с одинаковыми ключами;

4) естественность поведения – характеристика «поведения» алгоритма при обработке уже отсортированного массива [13].

Различают внутреннюю и внешнюю сортировку. При внутренней сортировке исходная структура данных будет целиком помещаться в оперативную память с произвольным доступом к любой

ячейке. Данные обычно упорядочиваются на том же месте без дополнительных затрат памяти, и основной целью алгоритма будет оптимизация числа выполняемых операций программы. Внешняя сортировка предполагает, что данные хранятся на внешнем устройстве с медленным доступом (диск, лента и т. д.), и, прежде всего, надо снизить число обращений к этому устройству [1, 2, 7, 11].

Далее будут рассмотрены алгоритмы на примере сортировки массива.

3.1. Сортировка подсчетом

В этой сортировке потребуется вспомогательный массив B, который в итоге станет результирующим. Суть этого алгоритма заключается в том, что на каждой итерации подсчитывается, на какую позицию результирующего массива B надо записать текущий элемент исходного массива A. Если некоторый элемент $A[i]$ будет помещаться в результирующий массив в позицию $k+1$, то слева от $B[k+1]$ должны стоять элементы меньшие или равные $B[k+1]$. Получается, что число k складывается из количества элементов меньших $A[i]$ и, возможно, некоторого числа элементов равных $A[i]$. На рис. 3.1 приведена иллюстрация работы алгоритма сортировки подсчетом [1].

3.2. Сортировка включением

Изначально массив делится на отсортированную и неотсортированную часть. На каждом шаге алгоритма будет выбираться один из элементов неотсортированной части и вставляться на нужную позицию в уже отсортированной части массива до тех пор, пока неотсортированная часть массива данных не будет

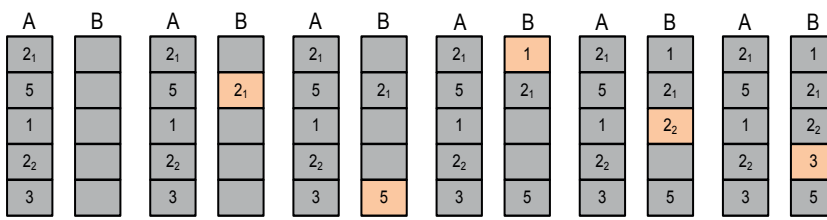


Рис. 3.1. Сортировка подсчетом

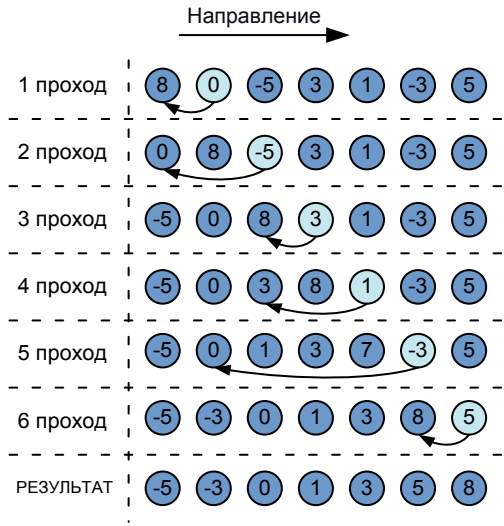


Рис. 3.2. Сортировка включением

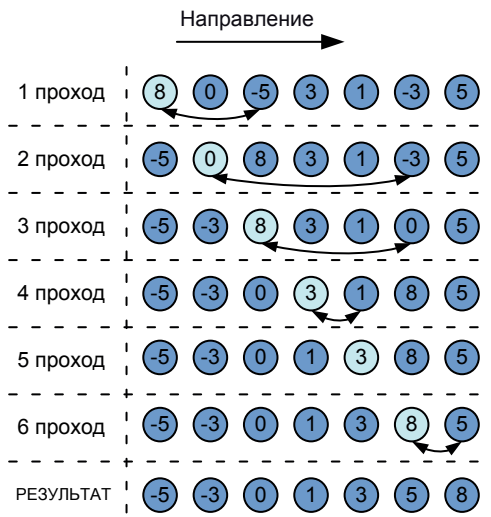


Рис. 3.3. Сортировка извлечением

исчерпана (рис. 3.2). Метод выбора очередного элемента из исходного массива произволен. Изначально отсортированная часть массива равна одному элементу [1, 7].

3.3. Сортировка извлечением

Изначально массив делится на отсортированную и неотсортированную части. В данном виде сортировки на первом шаге из неотсортированной части исходного массива данных выбирается элемент с наименьшим значением и меняется местами с наибольшим элементом отсортированной части. Далее аналогичная операция производится на следующем шаге и повторяется до тех пор, пока размер неотсортированной части массива не будет равен нулю (рис. 3.3) [1, 7].

3.4. Шейкерная сортировка

Процесс сортировки на первой итерации схож с сортировкой «пузырьком». Выполняется проход массива из n элементов слева направо и попарно сравниваются элементы. Если левый элемент больше правого, они меняются местами. По окончании итерации самый правый элемент будет максимальным. Затем происходит обход в обратную сторону, не затрагивающий последний отсортированный правый элемент. В результате обратного обхода самый левый элемент будет минимальным. Далее процедура повторяется, при каждом обходе слева направо происходит сдвиг начала операций сравнений на *+1*, в обходе же справа налево на *–1*. Выполнение алгоритма (рис. 3.4) прекращается, когда произведена последняя перестановка в центре массива [12].

Итерации	Элементы					
Начало	5	7	9	1	3	8
1 (право)	5	7	1	3	8	9
2 (лево)	1	5	7	3	8	9
3 (право)	1	5	3	7	8	9
4 (лево)	1	3	5	7	8	9
5 (право) - конец	1	3	5	7	8	9

Рис. 3.4. Шейкерная сортировка

3.5. Быстрая сортировка (Хоара)

Это один из самых распространенных алгоритмов сортировки. Он заключается в том, что на каждом шаге выбирается «опорный» элемент. Далее все остальные элементы переставляются относительно него, таким образом, что слева будут находиться элементы меньше «опорного» и, возможно, равные ему, а справа больше и, возможно, равные ему (в зависимости от принципа, по которому сортируются элементы массива, порядок элементов относительно «опорного» может отличаться от описанного варианта). На следующем шаге каждая из получившихся ранее частей будет сортироваться аналогичным образом (рис. 3.5) [1, 2, 7].

3.6. Сортировка слиянием

Процедуру сортировки слиянием можно описать следующим образом [1, 14]:

1) если в рассматриваемом массиве один элемент, то он уже отсортирован – алгоритм завершает работу;

2) исходный массив разбивается на k частей;

3) сортируем каждую часть массива;

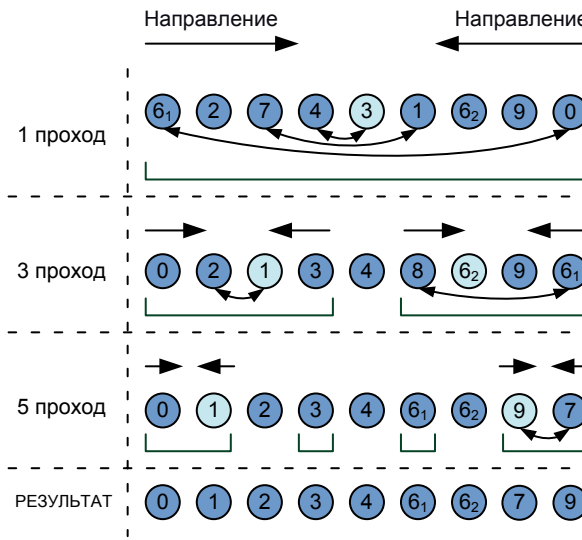


Рис. 3.5. Быстрая сортировка

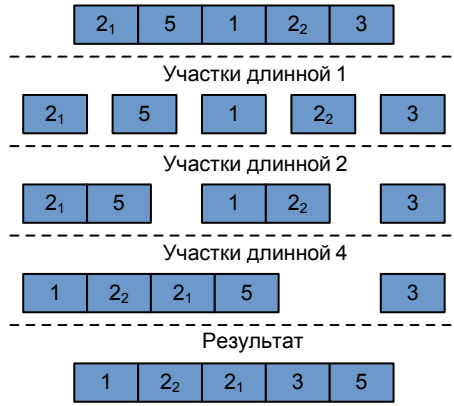


Рис. 3.6. Сортировка слиянием

4) «сливаем» попарно части массива;

5) проделываем пункты 3 и 4 до тех пор, пока все части массива не будут объединены.

На рис. 3.6 приведена иллюстрация работы алгоритма сортировки слиянием.

3.7. Сортировка распределением

В этой сортировке имеется исходный массив чисел A длины n. Также необходим вспомогательный массив B с индексами от 0 до k (максимальный элемент исходного массива), изначально заполняе-

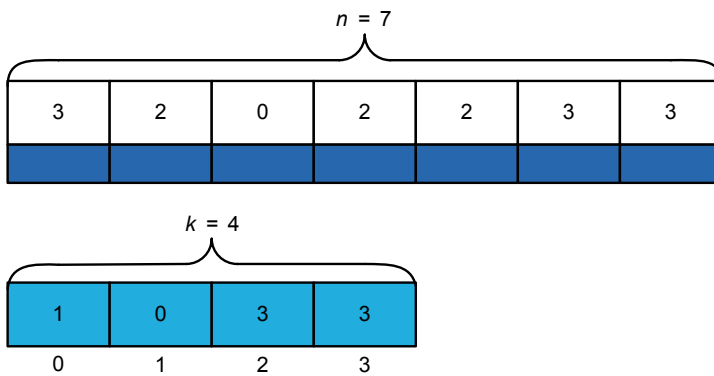


Рис. 3.7. Сортировка распределением

мый нулями. Алгоритм сортировки можно описать следующим образом [1]:

1) последовательно пройдем по массиву *A* и запишем в *B[i]* количество чисел равных *i*;

2) для получения отсортированного массива необходимо просто пройти по элементам массива *B* и для каждого *i* в массив *A* последовательно записать число *B[i]* раз.

На рис. 3.7 приведен пример исходного и вспомогательного массивов.

Стоит обратить особое внимание на то, что данную сортировку имеет смысл использовать только в том случае, если длина исходного массива больше или сопоставима с хранящимися значениями. В противном случае может возникнуть ситуация, при которой объем памяти, выделяемый под дополнительный массив, будет в десятки раз превосходить исходный.

3.8. Сравнение алгоритмов сортировки

Выше были рассмотрены некоторые алгоритмы внутренней сортировки. Нужно отметить, что это только небольшая часть существующих алгоритмов. Может возникнуть логичный вопрос, касающийся необходимости определения одного наиболее эффективного алгоритма, который можно использовать в любых прикладных задачах. К сожалению, при решении реальных задач всегда имеются определенные ограничения, определяемые как условиями решаемой задачи, так и свойствами используемой программной среды. Таким образом, выбор конкретного алгоритма с учетом внешний факторов и условий остается за разработчиком программного обеспечения. Теоретические временные и пространственные сложности рассмотренных методов сортировки показаны на рис. 3.8 [1].

Метод сортировки	Характеристики			
	Tmax	Tmid	Tmin	Vmax
Подсчетом	O(n^2)			O(n)
Простым включением	O(n^2)		O(n)	O(1)
Простым извлечением	O(n^2)			O(1)
Шейкерная	O(n^2)		O(n)	O(1)
Быстрая (Хоара)	O(n^2)	O(n*log n)		O(log n)
Слиянием	O(n*log n)			O(n)
Распределением	O(n)			O(n)

Рис. 3.8. Сводная таблица показателей алгоритмов сортировки

4. СПИСКИ

Список – это абстрактная структура данных, элементы которой содержат пользовательские данные и указатели на следующий (иногда и на предыдущий) элемент списка [1].

При решении большинства реальных прикладных задач невозможно на этапе проектирования и разработки программного обеспечения определить диапазон изменения значений переменной. Для решения подобных проблем имеет смысл использовать динамические структуры данных.

Динамической структурой данных может быть любая структура данных, для которой объем памяти не является фиксированным. Объем такой структуры данных, по сути, ограничен только имеющейся в наличии оперативной памятью. За счет способности к своего рода «адаптации» доступ к элементам таких структур, как правило, дольше и сложнее [1, 2, 14].

Динамические структуры данных, в отличие от статических структур, характеризуются двумя основными свойствами:

1) так как динамические структуры потенциально могут занимать очень большой объем памяти, для них зачастую неудобно (невозможно) выделить единый блок памяти, поэтому приходится предусматривать тот или иной метод для управления динамической памятью;

2) поскольку каждый элемент «связан» и хранит информацию о других элементах структуры, в заголовке не получится хранить информацию о всей структуре данных.

Для доступа к элементам динамических данных применяют указатели (в языке *C++*), под которыми понимаются переменные, инициализированные адресом, где хранится переменная (элемент структуры). Создание динамических данных должно обеспечивать самой разработанной программой непосредственно во время работы.

Одна из самых простых видов структур данных, содержащая некоторое количество элементов, – это линейный список. Он организует элементы в цепочку, которая связывается системой указателей [1–4].

4.1. Линейный однонаправленный список

В таких списках каждый элемент содержит в дополнение к полям данных одно поле указателя, который указывает на следующий элемент списка. Если элемент является последним, то указатель на следующий элемент является пустым (рис. 4.1) [1, 7, 11, 14].

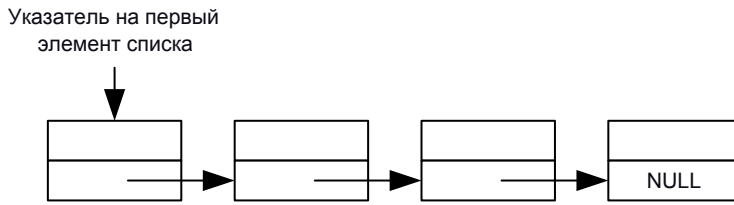


Рис. 4.1. Линейный однонаправленный список

С линейными однонаправленными списками определены следующие операции:

1) вставка (добавление) элемента;

2) просмотр (проход) списка;

3) поиск элемента;

4) удаление элемента.

Следует обратить особое внимание на то, что при выполнении любых операций с линейным однонаправленным списком необходимо всегда обеспечивать хранение указателя на первый элемент. Если указатель на первый элемент будет утерян (недоступен), то это приведет к потере данных всего списка [11].

Наличие только одного указателя в элементах линейного однонаправленного списка позволяет уменьшить расход дополнительной памяти на организацию структуры. В качестве недостатка такой организации нужно упомянуть, что осуществлять переходы между элементами списка можно только в одном направлении. Это увеличивает время, затрачиваемое на работу со списком. Например, если задача заключается в доступе к предыдущему элементу, нужно осуществить просмотр списка с первого элемента до элемента, указатель которого установлен на текущий элемент.

4.2. Линейный двунаправленный список

В этом виде линейных списков все элементы содержат два указателя. Один по аналогии с однонаправленным списком указывает на следующий элемент (у последнего элемента этот указатель будет пустым). Второй будет указывать на предыдущий элемент (у первого элемента этот указатель будет пустым) (рис. 4.2) [1, 7, 11, 14].

С линейным двунаправленным списком определены следующие операции:

1) вставка (добавление) элемента;

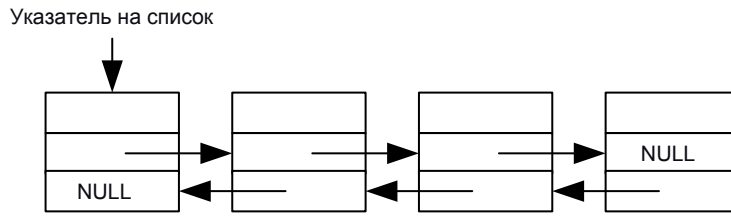


Рис. 4.2 Линейный двунаправленный список

2) просмотр (проход) списка;
3) поиск элемента;
4) удаление элемента.

В отличие от однонаправленного списка, в двунаправленном нет необходимости хранить указатель именно на первый элемент, так как благодаря наличию дополнительного указателя на предыдущий элемент всегда можно получить доступ к любому элементу списка из любого другого элемента. Но несмотря на описанную выше возможность, для удобства и быстродействия рекомендуется хранить указатель на первый элемент (голову списка).

Наличие второго указателя облегчает навигацию по элементам списка, однако дополнительный объем памяти под второй указатель, операции вставки (добавления) и удаления элементов становятся сложнее (требуется оперирование дополнительным указателем).

4.3. Циклический однонаправленный список

Идея циклического списка заключается в том, что указатель последнего элемента не является пустым, а указывает на первый элемент. В подобной структуре не получится выделить «первый» и «последний» элементы. В циклическом однонаправленном списке фактически все элементы являются «средними». В отличие от линейного однонаправленного списка, где удаление «первого» и «последнего» элементов несколько отличается от удаления «среднего», в циклическом однонаправленном списке все элементы могут быть удалены с помощью одного и того же подхода [10].

В циклическом однонаправленном списке понятие «первого» элемента весьма условно. Для полного обхода такого списка достаточно иметь указатель на произвольный элемент (рис. 4.3). Хранить указатель на «первый» элемент необязательно, но крайне желательно отслеживать какой-либо указатель на элемент списка, например,

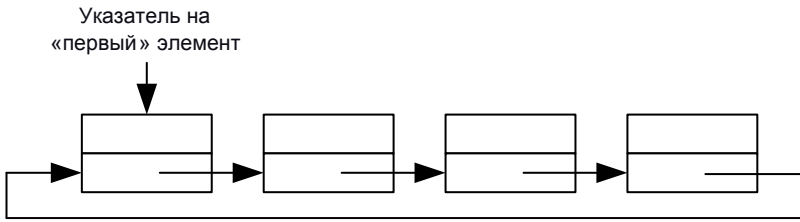


Рис. 4.3. Циклический однонаправленный список

это полезно при предотвращении «зацикливания» при поиске элемента [1].

С циклическим однонаправленным списком можно определить следующие операции:

1) вставка (добавление) элемента;

2) просмотр (проход) списка;

3) поиск элемента;

4) удаление элемента.

Наличие только одного указателя позволяет экономить объем памяти для хранения такой структуры данных. Но по аналогии с линейным однонаправленным списком проход списка возможен только в одном направлении. Из плюсов также можно отметить, что операции вставки (добавления) и удаления элементов достаточно упрощены.

Использовать циклический список имеет смысл, когда количество «проходов» по списку заранее неизвестно.

4.4. Циклический двунаправленный список

В таком виде циклических списков по аналогии с линейными списками каждый элемент имеет два указателя (один на следующий, второй на предыдущий элемент) (рис. 4.4) [1, 10].

С циклическим двунаправленным списком могут быть определены следующие операции:

1) вставка (добавление) элемента;

2) просмотр (проход) списка;

3) поиск элемента;

4) удаление элемента.

За счет использования двух указателей навигация и передвижение по элементам циклического двунаправленного списка значительно упрощены. Элементы списка из-за дополнительного поля

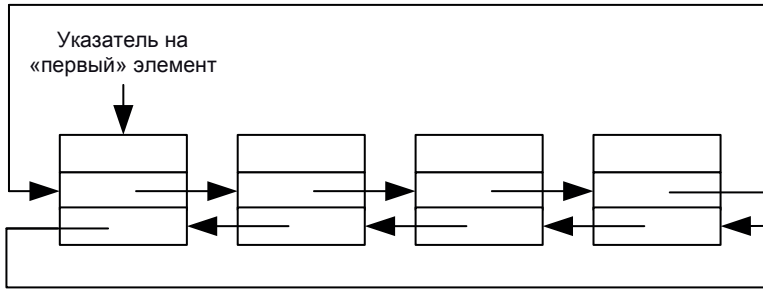


Рис. 4.4. Циклический двусвязный список

занимают больший объем памяти. Поскольку в этом виде списков нет «первого» и «последнего» элементов, операции вставки (добавления) и удаления элементов проще, чем в линейном двунаправленном списке. Однако за счет наличия второго указателя перечисленные операции сложнее, чем в циклическом однонаправленном списке (требуется оперирование большим числом указателей). В циклических двунаправленных списках рекомендуется хранить указатель на условно «первый» элемент для избегания возможного «зацикливания».

4.5. Слоеный список

Слоеные, или разделенные, списки – динамическая нелинейная структура данных (связные списки), которые позволяют «перескакивать» через определенное количество элементов. За счет возможности «пропустить» (от англ. – *skip*) некоторое количество элементов можно обойти ограничения последовательного поиска. Вставка (добавление) и удаление элементов остаются сравнительно эффективными в слоеных списках [1, 2].

Основная идея слоеных списков хорошо иллюстрируется с помощью такого примера, как записная книжка, в которой чтобы найти человека с определенной фамилией, сразу переходят на страницу, где начинается фамилия, а уже потом просматривают все имеющиеся там записи. Одновременно при необходимости (желании) можно последовательно просматривать все страницы.

Каждый элемент слоеного списка имеет дополнительный указатель. Все элементы списка будут группироваться по определенному признаку, и первый элемент каждой группы содержит указатель на первый элемент следующей группы (рис. 4.5). Дополнительный

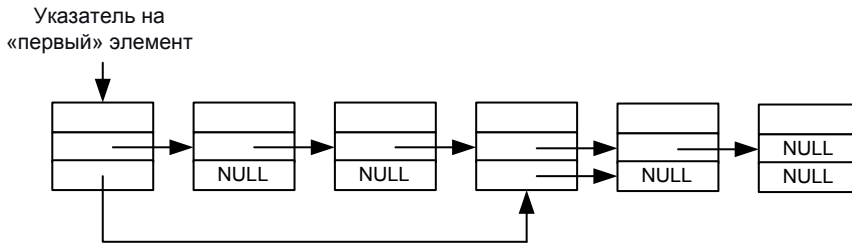


Рис. 4.5. Слоеный список

указатель принимает значение NULL в том случае, если следующая группа отсутствует или элемент не является первым в группе.

Идею этой структуры данных можно модифицировать путем добавления необходимого числа указателей, позволяющих группировать элементы по различным признакам.

Если реализован только один уровень (один указатель), то это фактически будет обычный линейный список, а время поиска элемента пропорционально $O(n)$. Однако, если организовать достаточное количество уровней (количество дополнительных указателей), то слоеный список можно считать деревом с корнем на высшем уровне, а для дерева время поиска будет пропорционально $O(\log n)$ [1].

5. СТЕК И ОЧЕРЕДЬ

Списки обеспечивают достаточно простое хранение данных: по сути, это некая абстракция над массивом [1, 14]. В этом разделе будут рассмотрены структуры хранения и доступа к данным с несколько специфичным поведением.

5.1. Стек

Стеком называется структура данных, в которой элементы могут добавляться и извлекаться только с одного конца (вершины стека) (рис. 5.1). Принцип работы этой структуры данных обычно называют «последним пришел – первым вышел», в англоязычной литературе «*last input – first output*» (LIFO) [1, 10].

Стеки отличаются от списков и массивов. Их нельзя индексировать напрямую, объекты добавляются и удаляются разными методами, и их содержимое недоступно для просмотра, в отличие от списков и массивов.

Одна из наиболее распространенных аналогий стека – стопка тарелок в ресторане [8].

Стек может быть реализован и как статическая, и как динамическая структура данных.

Как статическую структуру стек можно реализовать с помощью массива. Для реализации стека как динамической структуры данных обычно используют линейный список. Надо обратить особое внимание, что теоретически для реализации статического стека можно

Рис. 5.1. Стек

использовать тот же принцип, что и в списках (элементы связаны указателями), но не определять операции выделения дополнительной памяти под новый элемент. Однако такой подход не совсем корректен, так как линейный список, для которого не определены операции вставки (добавления) нового элемента, по своей сути полноценным списком не является.

Как уже говорилось ранее, для реализации статического списка нужно использовать массив, размер которого будет соответствовать максимальной глубине стека. Такая реализация влечет за собой неэффективное использование памяти, но вместе с тем взаимодействовать со стеком, реализованным таким образом, удобней и проще. При реализации стека с помощью массива его дно может располагаться в первом элементе массива, а каждый новый элемент будет добавляться в стек в сторону возрастания индексов массива. При добавлении или извлечении элемента необходимо будет осуществлять поэлементный сдвиг всех хранящихся в стеке элементов. Такой подход влечет за собой значительные вычислительные затраты. Если же добавлять элементы в стек в сторону увеличения индексов массива, то одновременно необходимо отдельно хранить значение индекса элемента массива, отслеживать и отдельно хранить индекс элемента, который является вершиной стека [1, 10].

Как динамическая структура данных стек легко реализовывается с помощью обычного линейного. Поскольку работа со стеком всегда идет через обращение к его вершине, а процедура просмотра всех элементов не требуется, то для экономии ресурсов (памяти) имеет смысл использовать линейный однонаправленный список. Для него

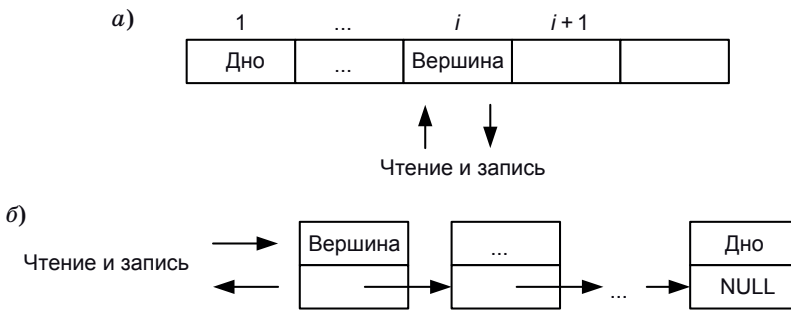


Рис. 5.2. Варианты реализации стека:
а – статистическая реализация (массив);
б – динамическая реализация (линейный список)

и так всегда хранится указатель на его голову, а в данной реализации этот указатель и будет указателем на вершину стека [2, 14].

На рис. 5.2 проиллюстрированы статическая и динамическая реализации стека.

Основные операции, производимые со стеком:

1) записать (*Push( )*);

2) прочитать (*Pop( )*);

3) посмотреть вершину (*Top( )*);

4) проверка пустоты стека (*Empty( )*).

Нужно понимать, что концепция структуры данных стека не предполагает просмотр всех данных, лежащих в нем, однако в последних версиях стандарта *C++* была добавлена функция *peek()*, позволяющая обратиться к любому элементу стека.

5.2. Очередь

Очередью называется структура данных, в которой элементы добавляются в один конец, а могут извлекаться только из другого конца. Таким образом, очередь будет представляться последовательностью, содержащей элементы в порядке их поступления (рис. 5.3). Для этой структуры данных работает принцип «первым пришел – первым вышел» [1, 10].

Очереди обычно используются в приложениях, чтобы обеспечить буфер добавления элементов для будущей обработки или для обеспечения упорядоченного доступа к общему ресурсу. Например,

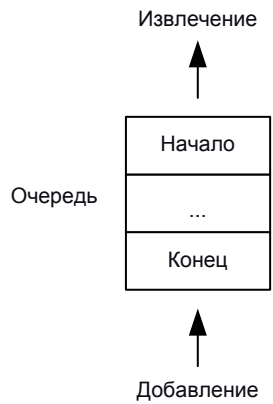


Рис. 5.3. Очередь

если база данных способна обрабатывать только одно соединение, очередь может использоваться, чтобы позволить потокам ожидать своего времени для доступа к базе данных [8].

Очередь, так же как и стек, может быть реализована и как статическая, и как динамическая структура данных.

При статической реализации очереди используют массив. Размер памяти под элементы массива необходимо резервировать, учитывая максимально возможную длину очереди. Как и в случае со стеком, такая реализация будет не эффективна с точки зрения выделения памяти. При реализации очереди с использованием массива ее началом является первый элемент массива, добавление новых элементов будет происходить в сторону увеличения индексов. При использовании организованной таким образом очереди придется всегда осуществлять поэлементный сдвиг хранящихся в очереди данных. Необходимость сдвига вызвана тем фактом, что элементы в очередь добавляются и извлекаются с разных концов, таким образом, постепенно будет происходить «миграция» элементов в конец массива. Рано или поздно, если не осуществлять сдвиг, может произойти исчерпание выделенной памяти, а, следовательно, невозможности добавления нового элемента в очередь. При работе с очередь, организованной с помощью массива, необходимо отслеживать и отдельно хранить значение индекса массива, который является концом очереди [1].

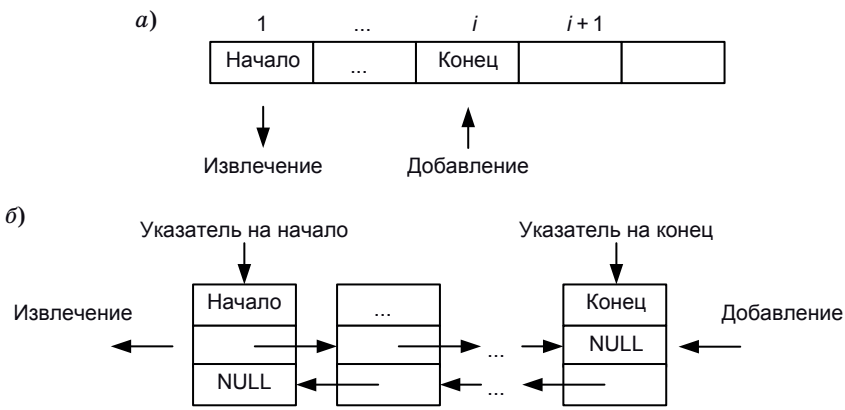


Рис. 5.4. Варианты реализации очереди.
а) – статистическая реализация (массив);
б) – динамическая реализация (линейный список)

Как динамическую структуру данных очередь обычно реализовывают с помощью линейного списка. Так как элементы в очередь добавляются и извлекаются с разных концов, то имеет смысл использовать двунаправленный список и отдельно хранить указатель на начало и на конец очереди.

На рис. 5.4 проиллюстрированы статическая и динамическая реализации стеков.

Основные операции, производимые с очередью:

1) добавление (*Push( )*);
2) извлечение (*Pop( )*);
3) посмотреть начало очереди (*Front( )*);
4) посмотреть конец очереди (*Back( )*)
5) проверка очереди на пустоту (*Empty( )*).

6. ДЕРЕВЬЯ ПОИСКА

6.1. Основные понятия теории графов

Поскольку деревья по своей сути являются частным случаем ориентированного графа, необходимо коротко привести основную информацию о таких структурах данных, как граф.

Графом G называется упорядоченная пара двух множеств (V, E), где V – непустое множество вершин; E – множество пар элементов множества V, называемое множеством ребер [1, 7, 15]. Нужно обратить внимание, что множество E может не содержать ни одного элемента. Такая ситуация возникает, например, когда в результате работы какого-либо алгоритма из графа удаляются все ребра, а остаются только вершины.

Дугой будут называться элементы множества E, если они упорядочены. Если все элементы являются дугами, то такой граф называется ориентированным (орграфом).

Ребра и в частном случае дуги имеют свойство инцидентности. Под инцидентностью понимают некий тип бинарного отношения между двумя объектами. Дуга инцидентна той вершине, в которую она входит, а вершина, из которой исходит дуга, называется смежной. Ребро инцидентно обеим вершинам, которые оно соединяет.

Любая последовательность элементов множества V такая, что одна вершина может следовать за другой, только если существует дуга, соединяющая первую вершину со второй, называется путем. Циклом в графе называют путь, который начинается и заканчивается в одной и той же вершине. Если граф не содержит циклов, то говорят, что граф ациклический.

Петлей называют дугу, которая соединяет вершину саму с собой. Петля не является циклом, ее определение не соответствует определению пути (путь определен для двух различных вершин).

«Теория графов является важной частью вычислительной математики. С помощью этой теории решаются большое количество задач из различных областей. Граф состоит из множества вершин и множества ребер, которые соединяют между собой вершины. С точки зрения теории графов не имеет значения, какой смысл вкладывается в вершины и ребра. Вершинами могут быть населенные пункты, а ребрами дороги, соединяющие их, или вершинами являться подпрограммы, а соединение вершин ребрами означает взаимодействие подпрограмм. Часто имеет значение направление дуги в графе» [1, 15].

6.2. Деревья

На основе ранее описанных определений из теории графов можно дать определение структуры данных, которая называется деревом.

Дерево является частным случаем орграфа, который удовлетворяет следующим условиям:

1) есть элемент дерева (узел), в который не входит ни одной дуги, такой элемент называется корнем дерева;

2) во все остальные (кроме корня) вершины орграфа входит только одна дуга.

С точки зрения наименований все вершины дерева можно разделить на следующие типы:

1) корнем называется вершина, в которую не входит ни одной дуги;

2) узлом называется вершина, в которую входит строго одна дуга, а выходить может некоторое множество дуг;

3) листьями называется вершина, в которую входит строго одна дуга, и из которой не выходят дуги [1].

Если из вершины *a* выходят дуги в вершины *b* и *c*, то говорят, что *b* и *c* являются потомками вершины *a*, а сама вершины *a* будет являться предком. В любом дереве только одна вершина не будет иметь предка – корень. В любом дереве листья не будут иметь своих потомков.

Чтобы легче было описывать иерархическую структуру дерева, обычно выделяют уровни. На первом уровне дерева существует только корень, на втором уровне могут существовать потомки корня, на третьем – потомки потомков корня и так далее [1–4, 10].

Деревья обычно графически представляют в «перевернутом» виде (корень будет сверху). На рис. 6.1 изображено дерево, в котором вершина *a* – корень; вершины *b, c, e* – узлы дерева; вершины *d, f, g, h, i, j* – листья дерева.

Максимальная из возможных длина последовательности потомков дерева называется высотой дерева. Если элементы дерева располагаются на *n* уровнях (рис. 6.1), то высота дерева также будет *n*. Если дерево является пустым, то высота такого дерева равняется нулю.

Узлы в дереве могут иметь различные степени. Под степенью вершины дерева понимают количество дуг, выходящих из нее (количество потомков вершины). Степень всего дерева определяется как максимальная степень вершины, которая входит в дерево. У листьев дерева степень всегда равняется нулю.

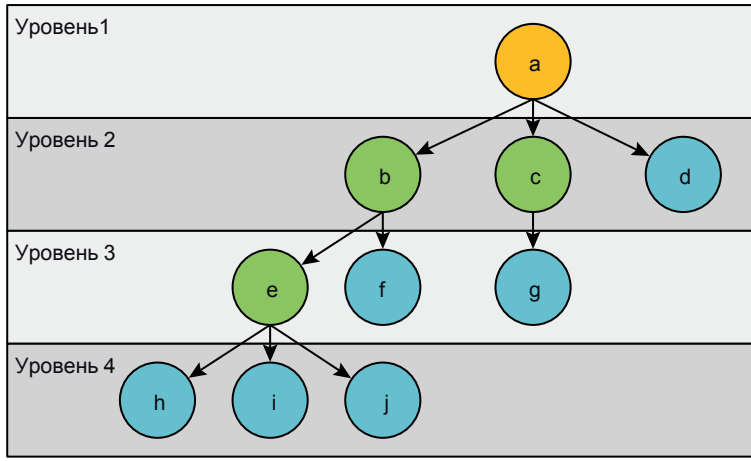


Рис. 6.1 Структура данных «Дерево»

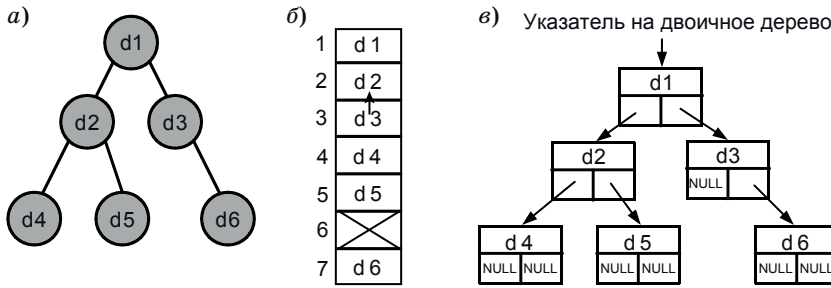


Рис. 6.2. Способы реализации двоичного дерева. а) – двоичное дерево; б) – статистическая реализация; в) – динамическая реализация

С точки зрения степеней деревья классифицируют на два типа:

1) двоичные (бинарные) – это деревья со степенью не больше двух;

2) сильноветвящиеся – это деревья со степенью больше двух.

В этой классификации нет деревьев со степенью один, потому что такие деревья по факту являются линейными списками.

С точки зрения реализации двоичного дерева есть несколько способов: статический (с помощью массива) и динамический (с использованием структур и указателей). Иллюстрация статической и динамической реализации дерева приведена на рис. 6.2 [1, 10].

Для работы с такой структурой данных, как дерево, необходимо просматривать все элементы. Из-за нелинейности этой структуры

данных есть несколько специфических способов обхода (просмотра) всех вершин дерева. Наиболее часто используют следующие виды обходов:

1) обход в прямом порядке;

2) обход в обратном порядке;

3) обход в симметричном (центрированном) порядке.

Для реализации алгоритмов обхода иногда используют стек и очередь. Но в данном пособии будут рассмотрены рекурсивные методы обхода.

6.3. Симметричный обход дерева

При прохождении дерева в симметричном порядке сначала рекурсивно посещаются вершины левого поддерева, далее корень, затем вершины правого поддерева. На рис. 6.3 приведена иллюстрация симметричного обхода дерева и код на языке *C++* [1, 10].

6.4. Обратный обход дерева

При прохождении дерева в обратном порядке сначала рекурсивно посещаются вершины левого поддерева, далее вершины правого поддерева, последним посещается корень. На рис. 6.4 приведена иллюстрация обратного обхода дерева и код на языке *C++* [1, 10].

6.5. Прямой обход дерева

При прохождении дерева в прямом порядке сначала рекурсивно посещается корень, далее вершины левого поддерева, затем вершины правого поддерева. На рис. 6.5 приведена иллюстрация обратного обхода дерева и код на языке *C++* [1, 10].

6.6. АВЛ – деревья поиска

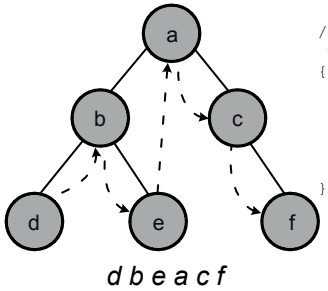
В рамках настоящего курсового проектирования нужно организовать не просто дерево, а именно дерево поиска. В общем случае оно должно быть:

1) бинарным;

2) упорядоченным;

3) сбалансированным.

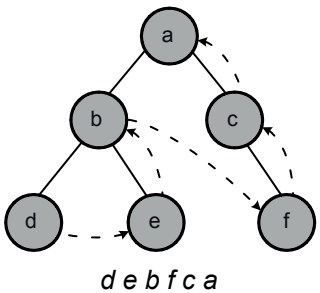
В нашем случае под сбалансированным будет пониматься дерево, сбалансированное по высоте. Балансировка дерева выполняется с целью избавиться от возможной вырожденности дерева при случайном добавлении элементов.




```
/*СИММЕТРИЧНЫЙ ОБХОД*/
 void Show(Node *&tree)
{
    if (tree == NULL) return; //Если дерева нет, выход

    Show(tree->left); //Обошли левое поддерево
    cout<<tree->data<<endl; //Посетили узел
    Show(tree->right); //Обошли правое поддерево
}
```

Рис. 6.3. Симметричный обход дерева

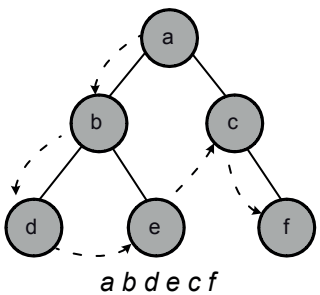



```
/*ОБРАТНЫЙ ОБХОД*/
 void Show(Node *&tree)
{
    if (tree == NULL) return; //Если дерева нет, выход

    Show(tree->left); //Обошли левое поддерево
    Show(tree->right); //Обошли правое поддерево

    cout<<tree->data<<endl; //Посетили узел
}
```

Рис. 6.4. Обратный обход дерева




```
/*ОБХОД В ПРЯМОМ ПОРЯДКЕ*/
 void Show(Node *&tree)
{
    if (tree == NULL) return; //Если дерева нет, выход

    cout<<tree->data<<endl; //Посетили узел
    Show(tree->left); //Обошли левое поддерево
    Show(tree->rifgt); //Обошли правое поддерево
}
```

Рис. 6.5. Прямой обход дерева

В 1962 году два советских математика Г. М. Адельсон-Вельский и Е. М. Ландис ввели определение сбалансированности дерева и доказали, что добавление и удаление элемента в такое древо будут иметь логарифмическую сложность [1, 10].

«Дерево считается сбалансированным по АВЛ (далее – «сбалансированным»), если для каждой вершины выполняется требование: высота левого и правого поддеревьев различаются не более, чем на единицу» [1].

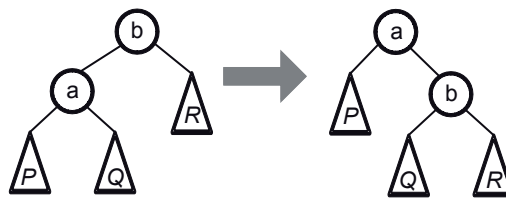
При добавлении и удалении может произойти нарушение структуры сбалансированного дерева. В таких ситуациях потребуются следующие преобразования:

1) малый левый поворот;
2) малый правый поворот;
3) большой левый поворот;
4) большой правый поворот.

Стоит отметить, что большие повороты по своей сути состоят из двух малых поворотов. Также для понимания и более простого объяснения поворотов введем понятие баланс фактора (BF).

Баланс фактор вершины – это разница между высотой левого и правого поддеревьев этой вершины. На рис. 6.6 приведено условие, при котором осуществляется малый левый поворот, схема поворота, код программы на языке *C++* и пример [1, 2].

Когда используется:
BF(b)=2 и BF[a]=1
или
BF(b)=2 и BF[a]=0




```
// левый поворот
node* rotateleft(node* b)
{
     node* a = b->left;
     b->left = a->right;
     a->right = b;
     return a;
}
```

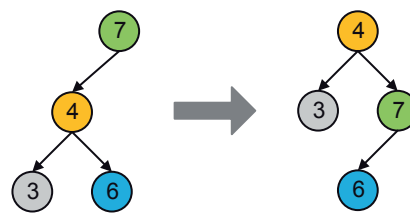
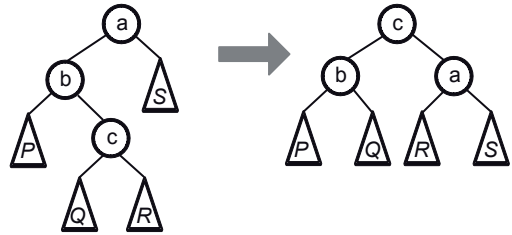


Рис. 6.6. Малый левый поворот

Когда используется:
BF(a)=2 и BF[b]=-1




```
// Большой левый поворот
node* bidrotateleft(node* a)
{
     rotateright(a->left);
     rotateleft(a);
}
```

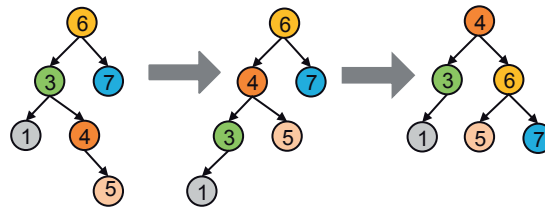


Рис. 6.7. Большой левый поворот

Левые и правые повороты абсолютно симметричны, поэтому пример правого поворота приводить не имеет смысла. Однако стоит обратить внимание, что в различных источниках левые повороты могут называться правыми, и наоборот. Принципиального значения название не имеет, но в данном пособии мы будем руководствоваться тем, из какого поддерева потомок встал на место первоначальной вершины (относительно которой совершался поворот) [1, 2].

На рис. 6.7 приведено условие, при котором осуществляется большой левый поворот, схема поворота, код программы на языке *C++* и пример.

Как уже упоминалось ранее, при большом повороте совершаются два малых поворота, в частности, при большом левом повороте в начале совершается малый правый поворот относительно правого потомка вершины (в которой нарушен баланс фактор), а затем совершается малый левый поворот уже относительно самой вершины.

При операциях добавления и удаления элементов из дерева всегда необходимо проверять, не нарушена ли сбалансированность дерева.

В сбалансированном дереве алгоритмы поиска, добавления и удаления элементов имеют логарифмическую сложность [1].

7. ПОИСК В ТЕКСТЕ

При решении практических задач, особенно связанных с обработкой больших массивов данных, приходится использовать алгоритмы поиска слова в тексте. Такие алгоритмы можно описать следующим образом. В заданном массив данных T (текст) размерностью n элементов, требуется найти последовательность элементов массива W (слово), размер которого не превышает n [1, 7, 10].

В результате работы подобных алгоритмов, как правило, обнаруживается первое совпадение слова (массив W) с фрагментом текста (массив T). Такая задача является очень распространенной в любых системах обработки данных, поэтому естественно, что существует определенное множество алгоритмов, способных решить поставленную задачу, с разной степенью эффективности. В данном пособии будут рассмотрены только два подобных алгоритма.

7.1. Алгоритм прямого поиска

В случае прямого поиска предполагается, что символы текста (массив T) и слова (массива W) будут сравниваться поэлементно.

На первом этапе работы алгоритма искомое слово сопоставляется с текстом так, что первый символ слова соотносится с первым символом текста. Далее происходит поэлементное сравнение первого символа слова с первым символом текста, второго символа слова со вторым символом текста и так далее. Если все символы совпадают, то результат работы алгоритма положительный (слово в тексте найдено). Если на каком-либо символе обнаружилось несоответствие, то происходит смещение слова на одну позицию так, что первый символ слова соответствует второму символу текста. Далее повторяется посимвольное сравнение. Смещение слова может повторяться до тех пор, пока не будет обнаружено полное совпадение символов либо пока не будет достигнут конец массива. Под достижением конца массива в данном случае понимается, что при попытке очередного смещения для последнего символа слова уже не будет соответствовать символ текста, так как текст закончился [1, 7, 10].

На рис. 7.1 приведена иллюстрация работы алгоритма прямого поиска слова в тексте.

Использование данного примитивного алгоритма достаточно эффективно в том случае, если для поиска строки потребуется небольшое количество смещений. К плюсам такого алгоритма можно отнести его понятность, он не требует предварительной подготовки

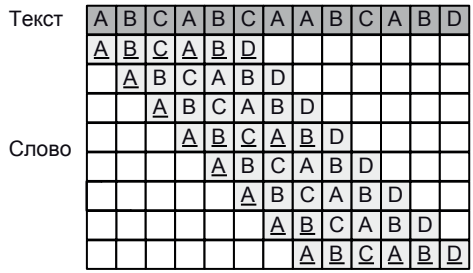


Рис. 7.1. Прямой поиск слова в тексте

данных. В языках высокого уровня данный алгоритм, подразумевающий сравнение участков памяти, достаточно хорошо оптимизирован. Также стоит отметить, что у этого алгоритма не наблюдается регрессия на «плохих» данных.

7.2. Алгоритм Боуера и Мура

В 1975 году Р. Боуер и Д. Мур предложили свой алгоритм поиска слова в тексте. Предполагается, что данный алгоритм улучшает обработку самого плохого случая.

Алгоритм Боуера и Мура (далее – БМ-поиск) предполагает, что символы будут сравниваться с конца слова. Перед началом сравнений необходима предобработка слова, которая преобразует его в некоторую таблицу. В ней помимо символов алфавита (символы, входящие и в слово, и в текст) хранится еще величина, характеризующая расстояние самого правого вхождения символа в слово до правого конца слова. Если символа в слове вообще нет, то в таблицу заносится величина равная длине слова. Алгоритм будет работать следующим образом. На первом шаге искомое слово сопоставляется с текстом так, что первый символ слова соотносится с первым символом текста. Далее начинается поэлементное сравнение символов с конца слова. Если все символы совпали, то слово в тексте найдено. Если обнаружено несовпадение символов, то слово можно сместить вправо относительно текста таким образом, чтобы самый правый символ слова оказался на той же позиции, что и последний несовпадающий символ текста. Если несовпадающего символа текста в слове вообще нет, то смещение происходит на длину всего слова относительно текущей позиции (рис. 7.2) [1, 7, 10].

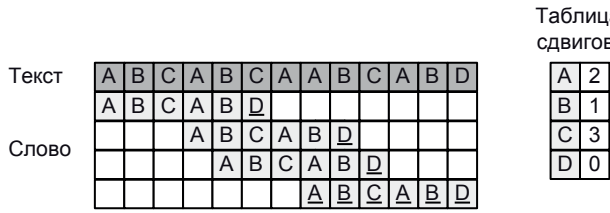


Рис. 7.2. Алгоритм поиска слова в тексте Боуера и Мура

К достоинствам данного алгоритма можно отнести то, что смещение слова относительно текста часто происходит сразу на несколько позиций. Данный алгоритм считается наиболее быстрым алгоритмом поиска слова в тексте общего назначения. Существует достаточно большое количество его оптимизаций. К минусам алгоритма стоит отнести то, что его эффективность сильно страдает при «плохих» данных.

8. ЗАДАНИЕ НА КУРСОВОЙ ПРОЕКТ

8.1. Цели и задачи проектирования

Цель курсового проектирования: изучение структур данных и алгоритмов их обработки, а также получение практических навыков их использования при разработке программ.

Задача курсового проекта: разработка информационной системы для заданной предметной области с использованием заданных структур данных и алгоритмов.

Тема курсового проекта: «Использование заданных структур данных и алгоритмов при разработке программного обеспечения информационной системы».

8.2. Варианты задания

Вариант задания на курсовой проект формируется из нескольких компонент:

– предметная область (табл. 8.1);
– метод хеширования (табл. 8.2);
– метод сортировки (табл. 8.3);
– вид списка (табл. 8.4);
– метод обхода дерева (табл. 8.5);
– алгоритм поиска слова в тексте (табл. 8.6).

Номер варианта для конкретного компонента определяется студентом как три последние цифры номера его студенческого билета, взятые по модулю количества вариантов для конкретного компонента, то есть

$$N_{вар} = nnn \bmod K,$$

где $N_{вар}$ – номер варианта; nnn – три последние цифры номера студенческого билета; K – количество вариантов заданий для конкретного компонента.

Для определения каждого компонента варианта задания на курсовой проект студенту необходимо воспользоваться таблицами, которые представлены ниже.

В табл. 8.1 представлен перечень возможных предметных областей. Подробное описание каждой предметной области приведено в разделе 9 настоящего учебного пособия.

Например, три последние цифры номера студенческого билета – «076». Тогда вариант задания на курсовой проект, который необходимо выполнить студенту, будет:

– предметная область – «Продажа авиабилетов» ($076 \bmod 6 = 4$);

Таблица 8.1

№ п/п	Предметная область
0	Обслуживание читателей в библиотеке (см. п. 9.1)
1	Обслуживание клиентов в бюро проката автомобилей (см. п. 9.2)
2	Регистрация постояльцев в гостинице (см. п. 9.3)
3	Регистрация больных в поликлинике (см. п. 9.4)
4	Продажа авиабилетов (см. п. 9.5)
5	Обслуживание клиентов оператора сотовой связи (см. п. 9.6)

Таблица 8.2

№ п/п	Метод хеширования
0	Открытое хеширование
1	Закрытое хеширование с линейным опробованием
2	Закрытое хеширование с квадратичным опробованием
3	Закрытое хеширование с двойным хешированием

Таблица 8.3

№ п/п	Метод сортировки
0	Подсчетом
1	Включением
2	Извлечением
3	Шейкерная
4	Быстрая (Хоара)
5	Слиянием
6	Распределением

Таблица 8.4

№ п/п	Вид списка
0	Линейный однонаправленный
1	Линейный двунаправленный
2	Циклический однонаправленный
3	Циклический двунаправленный
4	Слоеный

Таблица 8.5

№ п/п	Метод обхода дерева
0	Симметричный
1	Обратный
2	Прямой

Таблица 8.6

№ п/п	Алгоритм поиска слова в тексте
0	Боуера и Мура (БМ)
1	Прямой

– метод хеширования – открытое (*076 mod 4 = 0*);
– метод сортировки – распределением (*076 mod 7 = 6*);
– вид списка – линейный двунаправленный (*076 mod 5 = 1*);
– метод обхода дерева – обратный (*076 mod 3 = 1*);
– алгоритм поиска слова в тексте – Боуера и Мура (БМ) (*076 mod 2 = 0*).

Для защиты курсового проекта необходима пояснительная записка, содержание которой определяется соответствующим пунктом данного учебного пособия. Образец титульного листа можно получить на сайте ГУАП в разделе нормативной документации.

Система и критерии оценки выполненного курсового проекта доводятся до сведения студентов преподавателем на первом занятии учебного семестра.

8.3. Порядок выполнения работы

Примерный график выполнения курсового проекта в течение одного семестра приведен в табл. 8.7. Окончательный список этапов и график выполнения курсового проекта доводятся до студентов преподавателем на первом занятии в учебном семестре.

Таблица 8.7

Этап выполнения	Порядковый номер недели семестра
Получение задания, выбор варианта задания в соответствии с разделом 8 и согласование его с преподавателем	1–2
Разработка первой структуры данных, выбор и программирование алгоритмов обработки этой структуры данных	3–6

Окончание табл. 8.7

Этап выполнения	Порядковый номер недели семестра
Разработка второй структуры данных, выбор и программирование алгоритмов обработки этой структуры данных	7–8
Разработка третьей структуры данных, выбор и программирование алгоритмов обработки этой структуры данных	9–12
Компоновка программы, ее тестирование, отладка и демонстрация	13–14
Оформление пояснительной записки и защита проекта	15–17

8.4. Содержание пояснительной записки

Пояснительная записка к курсовому проекту должна содержать:

1) титульный лист;

2) содержание;

3) задание на курсовой проект (с указанием выбранного варианта);

4) введение (краткая характеристика решаемой задачи, обоснование необходимости решения этой и подобных задач);

5) алгоритмы и структуры данных (описание и анализ используемых в курсовом проекте алгоритмов и структур данных, описание их реализации в вычислительных машинах);

6) описание программы (краткое описание структуры программы, руководство по использованию программы, листинг программы с комментариями);

7) тестирование программы (исходные данные для тестовых прогонов программы, результаты тестирования);

8) заключение;

9) список использованной литературы.

Все приведенные разделы являются обязательными. Листинг программы допускается помещать в приложении к пояснительной записке.

8.5. Рекомендации по выполнению курсового проекта

Теоретический материал, используемый при курсовом проектировании, изложен в учебном пособии.

В качестве языка программирования, используемого для реализации заданных структур данных и алгоритмов, можно выбрать любой язык, изученный студентом ранее, по согласованию с преподавателем. Стоит обратить внимание, что функционал информационной системы, который необходимо разработать по заданию, должен быть реализован студентом самостоятельно. Рекомендуется обосновывать выбор языка программирования.

Особых требований к интерфейсу программы не предъявляется. Состав и форма отображаемой информации, а также способы управления программой и ввода данных должны быть достаточными для демонстрации всех функций и структур данных, которые определены в задании.

Первым этапом выполнения курсового проекта являются получение задания и выбор варианта задания в соответствии с разделом 8. Во избежание недоразумений целесообразно согласовать выбранный вариант с преподавателем.

Дальнейшее выполнение курсового проекта заключается в поочередной разработке заданных структур данных и реализации алгоритмов обработки этих структур.

При разработке структуры данных студент должен самостоятельно определять размерность некоторых элементов данных (например, длину строк), способ реализации этой структуры данных в памяти вычислительной машины, а также выбирать алгоритмы выполнения некоторых операций. Принятые решения должны быть отражены в пояснительной записке. Кроме того, должны быть приведены обоснования принятым решениям. При обосновании могут быть перечислены альтернативные решения, и указаны преимущества выбранного решения по сравнению с остальными.

Некоторые операции, которые необходимо реализовать в курсовом проекте, требуют наличия сразу нескольких структур данных. При поэтапной разработке обращения к еще не существующим структурам данных временно заменяются на так называемые заглушки. При окончательной компоновке программы вместо «заглушек» будут использованы реальные обращения.

Для демонстрации программы целесообразно подготовить тестовый набор данных. Объем этих данных должен быть достаточен для демонстрации основных свойств разработанных структур данных и выполнения всех заданных операций. В частности, в хеш-таблицу должны быть внесены несколько элементов, образующих коллизию, а АВЛ-дерево должно заполняться данными таким образом, чтобы продемонстрировать процесс его балансировки.

9. ПЕРЕЧЕНЬ ПРЕДМЕТНЫХ ОБЛАСТЕЙ

В соответствии со стандартом *ISO/IEC 2382:2015* под информационной системой понимается система, организующая обработку информации о предметной области и ее хранение [16]. В свою очередь под понятием предметная область можно понимать часть реального мира, которая рассматривается в пределах данного контекста. Под контекстом здесь может пониматься, например, область исследования или область, которая является объектом некоторой деятельности.

Таким образом, при разработке информационной систем ключевую роль будет играть предметная область. Именно она определяет, как и какие данные будут храниться в разрабатываемой системе.

9.1. Обслуживание читателей в библиотеке

9.1.1. Информационная система для предметной области «Обслуживание читателей в библиотеке» должна осуществлять ввод, хранение, обработку и вывод данных о:

– читателях;

– книгах;

– выдаче и приеме книг от читателей.

9.1.2. Данные о каждом читателе должны содержать:

– Номер читательского билета – строка формата «*ANNNN-YY*», где *A* – буква, обозначающая права доступа читателя (А – только абонемент, Ч – только читальный зал, В – читальный зал и абонемент); *NNNN* – порядковый номер регистрации (цифры); *YY* – последние две цифры номера года регистрации;

– ФИО – строка;

– Год рождения – целое;

– Адрес – строка;

– Место работы/учебы – строка.

Примечание: длина строк (кроме номера читательского билета) определяется студентом самостоятельно.

9.1.3. Данные о читателях должны быть организованны в виде хеш-таблицы, первичным ключом которой является «номер читательского билета». Метод хеширования определяется вариантом задания.

9.1.4. Данные о каждой книге должны содержать:

– Шифр – строка формата «*NNN.MMM*», где *NNN* – номер тематического раздела (цифры); *MMM* – порядковый номер книги в разделе (цифры);

– Автор(ы) – строка;
– Название – строка;
– Издательство – строка;
– Год издания – целое;
– Количество экземпляров всего – целое;
– Количество экземпляров в наличии – целое.

Примечание: длина строк (кроме Шифра) определяется студентом самостоятельно.

9.1.5. Данные о книгах должны быть организованны в виде АВЛ-дерева поиска, упорядоченного по «Шифру».

9.1.6. Данные о выдаче или приеме книг от читателей должны содержать:

– Номер читательского билета – строка, формат которой соответствует аналогичной строке в данных о читателях;
– Шифр – строка, формат которой соответствует аналогичной строке в данных о книгах;
– Дата выдачи – строка;
– Дата возврата – строка.

Примечания:

1. Наличие в этих данных записи, содержащей в поле «Номер читательского билета» значение *X* и в поле «Шифр» значение *Y*, означает выдачу читателю с номером читательского билета *X* экземпляра книги с шифром *Y*. Отсутствие такой записи означает, что читателю с номером читательского билета *X* не выдавался ни один экземпляр книги с шифром *Y*.

2. Одному читателю может быть выдано несколько книг, и экземпляры одной книги могут быть выданы нескольким читателям. Таким образом, могут быть данные, имеющие повторяющиеся значения в своих полях.

9.1.7. Данные о выдаче или приеме книг от читателей должны быть организованны в виде списка, который упорядочен по первичному ключу – «Шифр». Вид списка и метод сортировки определяются вариантом задания.

9.1.8. Информационная система «Обслуживание читателей в библиотеке» должна осуществлять:

– регистрацию нового читателя;
– снятие с обслуживания читателя;
– просмотр всех зарегистрированных читателей;
– очистку данных о читателях;
– поиск читателя по номеру читательского билета. Результаты поиска – все сведения о найденном читателе и шифры книг, которые ему выданы;

– поиск читателя по ФИО. Результаты поиска – список найденных читателей с указанием номера читательского билета и ФИО;

– добавление новой книги;

– удаление сведений о книге;

– просмотр всех имеющихся книг;

– очистку данных о книгах;

– поиск книги по шифру. Результаты поиска – все сведения о найденной книге, а также номера читательских билетов и ФИО читателей, которым выданы экземпляры этой книги;

– поиск книги по фрагментам ФИО автора(ов) или названия. Результаты поиска – список найденных книг с указанием шифра, автора(ов), названия, издательства, года издания и количества экземпляров в наличии;

– регистрацию выдачи экземпляра книги читателю;

– регистрацию приема экземпляра книги от читателя.

9.1.9. Состав данных о читателе или книге, выдаваемых при просмотре всех зарегистрированных читателей или просмотре всех имеющихся книг, определяется студентом самостоятельно, но должен содержать не менее двух полей.

9.1.10. Метод поиска читателя по ФИО определяется студентом самостоятельно. Выбранный метод необходимо сравнить с альтернативными методами.

9.1.11. Поиск книги по фрагментам ФИО автора(ов) или названия должен осуществляться путем систематического обхода АВЛ-дерева поиска. Метод обхода определяется вариантом задания. При поиске книги по фрагментам ФИО автора(ов) или названия могут быть заданы как полное ФИО автора(ов) или названия, так и их части (например, ФИО одного из нескольких авторов, одно слово или часть слова из названия). Для обнаружения заданного фрагмента в полном ФИО автора(ов) или названии должен применяться алгоритм поиска слова в тексте, указанный в варианте задания.

9.1.12. Регистрация выдачи экземпляра книги читателю должна осуществляться только при наличии свободных экземпляров выдаваемой книги (значение поля «Количество экземпляров в наличии» для соответствующей книги больше нуля).

9.1.13. При регистрации выдачи экземпляра книги или приема экземпляра книги от читателя должно корректироваться значение поля «Количество экземпляров в наличии» для соответствующей книги.

9.1.14. При снятии с обслуживания читателя должны быть учтены и обработаны ситуации, когда у читателя имеются выданные книги. Аналогичным образом следует поступать и с удалением сведений о книгах.

9.2. Обслуживание клиентов в бюро проката автомобилей

9.2.1. Информационная система для предметной области «Обслуживание клиентов в бюро проката автомобилей» должна осуществлять ввод, хранение, обработку и вывод данных о:

– клиентах;

– автомобилях, принадлежащих бюро проката;

– выдаче на прокат и возврате автомобилей от клиентов.

9.2.2. Данные о каждом клиенте должны содержать:

– Номер водительского удостоверения – строка формата «*RR AA NNNNNN*», где *RR* – код региона (цифры); *AA* – серия (буквы из следующего множества: А, В, Е, К, М, Н, О, Р, С, Т, У, Х); *NNNNNN* – порядковый номер удостоверения (цифры). Код, серия и номер отделяются друг от друга пробелами;

– ФИО – строка;

– Паспортные данные – строка;

– Адрес – строка.

Примечание: длина строк (кроме номер водительского удостоверения) определяется студентом самостоятельно.

9.2.3. Данные о клиентах должны быть организованны в виде АВЛ-дерева поиска, упорядоченного по «номеру водительского удостоверения».

9.2.4. Данные о каждом автомобиле должны содержать:

– Государственный регистрационный номер – строка формата «*ANNNAA-NN*», где *N* –цифра; *A* – буква из следующего множества: А, В, Е, К, М, Н, О, Р, С, Т, У, Х;

– Марку – строка;

– Цвет – строка;

– Год выпуска – целое;

– Признак наличия – логическое.

Примечание: длина строк (кроме «Государственного регистрационного номера») определяется студентом самостоятельно.

9.2.5. Данные об автомобилях должны быть организованны в виде хеш-таблицы, первичным ключом которой является «Государственный регистрационный номер». Метод хеширования определяется вариантом задания.

9.2.6. Данные о выдаче на прокат или возврате автомобилей от клиентов должны содержать:

– строку, формат которой соответствует аналогичной строке в данных о клиентах;

– Государственный регистрационный номер – строка, формат которой соответствует аналогичной строке в данных об автомобилях;

– Дату выдачи – строка;

– Дату возврата – строка.

Примечание: наличие в этих данных записи, содержащей в поле «Номер водительского удостоверения» значение *X* и в поле «Государственный регистрационный номер» значение *Y*, означает выдачу клиенту с номером водительского удостоверения *X* автомобиля с государственным регистрационным номером *Y*. Отсутствие такой записи означает, что клиенту с номером водительского удостоверения *X* не выдавался автомобиль с номером *Y*.

9.2.7. Данные о выдаче на прокат или возврате автомобилей от клиентов должны быть организованны в виде списка, который упорядочен по первичному ключу – «Государственный регистрационный номер». Вид списка и метод сортировки определяются вариантом задания.

9.2.8. Информационная система «Обслуживание клиентов в бюро проката автомобилей» должна осуществлять следующие операции:

– регистрацию нового клиента;

– снятие с обслуживания клиента;

– просмотр всех зарегистрированных клиентов;

– очистку данных о клиентах;

– поиск клиента по «номер водительского удостоверения». Результаты поиска – все сведения о найденном клиенте и государственный регистрационный номер автомобиля, который ему выдан;

– поиск клиента по фрагментам ФИО или адреса. Результаты поиска – список найденных клиентов с указанием номера водительского удостоверения, ФИО и адреса;

– добавление нового автомобиля;

– удаление сведений об автомобиле;

– просмотр всех имеющихся автомобилей;

– очистку данных об автомобилях;

– поиск автомобиля по «Государственному регистрационному номеру». Результаты поиска – все сведения о найденном автомобиле, а также ФИО и номер водительского удостоверения клиента, которому выдан этот автомобиль;

– поиск автомобиля по названию марки автомобиля. Результаты поиска – список найденных автомобилей с указанием «Государственный регистрационный номер», марки, цвета, года выпуска;

– регистрацию отправки автомобиля в ремонт;
– регистрацию прибытия автомобиля из ремонта;
– регистрацию выдачи клиенту автомобиля на прокат;
– регистрацию возврата автомобиля от клиентов.

9.2.9. Состав данных о клиенте или автомобиле, выдаваемых при просмотре всех зарегистрированных клиентов или просмотре всех автомобилей, принадлежащих бюро проката, определяется студентом самостоятельно, но должен содержать не менее двух полей.

9.2.10. Метод поиска автомобиля по марке определяется студентом самостоятельно. Выбранный метод необходимо сравнить с альтернативными методами.

9.2.11. Поиск клиента по фрагментам ФИО или адреса должен осуществляться путем систематического обхода АВЛ-дерева поиска. Метод обхода определяется вариантом задания. При поиске клиента по фрагментам ФИО или адреса могут быть заданы как полное ФИО или адрес, так и их части (например, только фамилия клиента без имени и отчества, только название улицы из адреса). Для обнаружения заданного фрагмента в полном ФИО или адресе должен применяться алгоритм поиска слова в тексте, указанный в варианте задания.

9.2.12. Регистрация отправки автомобиля на ремонт должна осуществляться только при наличии этого автомобиля (значение поля «Признак наличия» для соответствующего автомобиля имеет значение «Истина»). При этом значение поля «Признак наличия» для соответствующего автомобиля изменяется на значение «Ложь».

9.2.13. При регистрации прибытия автомобиля из ремонта значение поля «Признак наличия» для соответствующего автомобиля изменяется на значение «Истина».

9.2.14. Регистрация выдачи автомобиля клиенту должна осуществляться только при наличии свободного выдаваемого автомобили (значение поля «Признак наличия» для соответствующего автомобиля имеет значение «Истина»).

9.2.15. При регистрации выдачи автомобиля клиенту или возврата автомобиля от клиента должно корректироваться значение поля «Признак наличия» для соответствующего автомобиля.

9.2.16. При снятии с обслуживания клиента должны быть учтены и обработаны ситуации, когда у клиента имеется выданный автомобиль. Аналогичным образом следует поступать и с удалением сведений об автомобилях.

9.3. Регистрация постояльцев в гостинице

9.3.1. Информационная система для предметной области «Регистрация постояльцев в гостинице» должна осуществлять ввод, хранение, обработку и вывод данных о:

– постояльцах;

– гостиничных номерах;

– вселении и выселении постояльцев.

9.3.2. Данные о каждом постояльце должны содержать:

– Номер паспорта – строка формата «*NNNN-NNNNNN*», где *N* – цифры;

– ФИО – строка;

– Год рождения – целое;

– Адрес – строка;

– Цель прибытия – строка.

Примечание: длина строк (кроме номера паспорта) определяется студентом самостоятельно.

9.3.3. Данные о постояльцах должны быть организованны в виде хеш-таблицы, первичным ключом которой является «номер паспорта» Метод хеширования определяется вариантом задания.

9.3.4. Данные о каждом гостиничном номере должны содержать:

– Номер гостиничного номера – строка формата «*ANNN*», где *A* – буква, обозначающая тип номера (Л – люкс, П – полулюкс, О – одноместный, М – многоместный); *NNN* – порядковый номер (цифры);

– Количество мест – целое;

– Количество комнат – целое;

– Наличие санузла – логическое;

– Оборудование – строка.

Примечание: длина строки «Оборудование», содержащая перечень оборудования номера (телевизор, холодильник и пр.), определяется студентом самостоятельно.

9.3.5. Данные о гостиничных номерах должны быть организованны в виде АВЛ-дерева поиска, упорядоченного по «Номеру гостиничного номера».

9.3.6. Данные о вселении или выселении постояльцев должны содержать:

– Номер паспорта – строка, формат которой соответствует аналогичной строке в данных о постояльцах;

– Номер гостиничного номера – строка, формат которой соответствует аналогичной строке в данных о гостиничных номерах;

– Дату заселения – строка;
– Дату выселения – строка.

Примечания:

1. Наличие в этих данных записи, содержащей в поле «номер паспорта» значение *X* и в поле «номер гостиничного номера» значение *Y*, означает заселение постояльца с номером паспорта *X* в гостиничный номер *Y*. Отсутствие такой записи означает, что постоялец с номером паспорта *X* не проживает в гостиничном номере *Y*.

2. В одном гостиничном номере (многоместном) могут проживать несколько постояльцев. Таким образом, могут быть данные, имеющие повторяющиеся значения в некоторых своих полях.

9.3.7. Данные о вселении или выселении постояльцев должны быть организованны в виде списка, который упорядочен по первичному ключу – «номер гостиничного номера». Вид списка и метод сортировки определяются вариантом задания.

9.3.8. Информационная система «Регистрация постояльцев в гостинице» должна осуществлять следующие операции:

– регистрацию нового постояльца;
– удаление данных о постояльце;
– просмотр всех зарегистрированных постояльцев;
– очистку данных о постояльцах;
– поиск постояльца по номеру паспорта. Результаты поиска – все сведения о найденном постояльце и номер гостиничного номера, в котором он проживает;
– поиск постояльца по ФИО. Результаты поиска – список найденных постояльцев с указанием номера паспорта и ФИО;
– добавление нового гостиничного номера;
– удаление сведений о гостиничном номере;
– просмотр всех имеющихся гостиничных номеров;
– очистку данных о гостиничных номерах;
– поиск гостиничного номера по «номеру гостиничного номера». Результаты поиска – все сведения о найденном гостиничном номере, а также ФИО и номера паспортов постояльцев, которые вселены в этот гостиничный номер;
– поиск гостиничного номера по фрагментам «Оборудования». Результаты поиска – список найденных гостиничных номеров с указанием номера гостиничного номера, количества мест, количества комнат, оборудования;
– регистрацию вселения постояльца;
– регистрацию выселения постояльца.

9.3.9. Состав данных о постояльцах или гостиничных номерах, выдаваемых при просмотре всех зарегистрированных постояльцев или просмотре всех имеющихся гостиничных номеров, определяется студентом самостоятельно, но должен содержать не менее двух полей.

9.3.10. Метод поиска постояльца по ФИО определяется студентом самостоятельно. Выбранный метод необходимо сравнить с альтернативными методами.

9.3.11. Поиск гостиничного номера по фрагментам «Оборудования» должен осуществляться путем систематического обхода АВЛ-дерева поиска. Метод обхода определяется вариантом задания. При поиске гостиничного номера по фрагментам «Оборудования» могут быть заданы как полный перечень оборудования гостиничного номера, так и его часть (например, указан только телевизор). Для обнаружения заданного фрагмента в полном перечне оборудования гостиничного номера должен применяться алгоритм поиска слова в тексте, указанный в варианте задания.

9.3.12. Регистрация вселения постояльца должна осуществляться только при наличии свободных мест в занимаемом гостиничном номере.

9.3.13. При удалении данных о постояльце должны быть учтены и обработаны ситуации, когда постоялец заселен в номер. Аналогичным образом следует поступать и с удалением сведений о гостиничном номере.

9.4. Регистрация больных в поликлинике

9.4.1. Информационная система для предметной области «Регистрация больных в поликлинике» должна осуществлять ввод, хранение, обработку и вывод данных о:

– больных;

– врачах;

– выдаче и возврате направлений к врачу.

9.4.2. Данные о каждом больном должны содержать:

– Регистрационный номер – строка формата «*MM-NNNNNN*», где *MM* – номер участка (цифры); *NNNNNN* – порядковый номер (цифры);

– ФИО – строка;

– Год рождения – целое;

– Адрес – строка;

– Место работы (учебы) – строка.

Примечание: длина строк (кроме «Регистрационного номера») определяется студентом самостоятельно.

9.4.3. Данные о больных должны быть организованны в виде хеш-таблицы, первичным ключом которой является «Регистрационный номер». Метод хеширования определяется вариантом задания.

9.4.4. Данные о каждом враче должны содержать:

– ФИО врача – строка длиной до 25 символов, содержащая фамилию врача и его инициалы;

– Должность – строка;

– Номер кабинета – целое;

– График приема – строка.

Примечание: длина строк (кроме «ФИО врача») определяется студентом самостоятельно.

9.4.5. Данные о врачах должны быть организованны в виде АВЛ-дерева поиска, упорядоченного по «ФИО врача».

9.4.6. Данные о выдаче или возврате направлений к врачу должны содержать:

– Регистрационный номер – строка, формат которой соответствует аналогичной строке в данных о больных;

– ФИО врача – строка, формат которой соответствует аналогичной строке в данных о врачах;

– Дату направления – строка;

– Время направления – строка.

Примечания:

1. Наличие в этих данных записи, содержащей в поле «Регистрационный номер» значение *X* и в поле «ФИО врача» значение *Y*, означает выдачу направления больному с регистрационным номером *X* к врачу с ФИО *Y*. Отсутствие такой записи означает, что больной с регистрационным номером *X* не имеет направления к врачу с ФИО *Y*.

2. К одному врачу могут направляться несколько больных в течение одного дня, но в разное время. Таким образом, могут быть данные, имеющие повторяющиеся значения в некоторых своих полях.

9.4.7. Данные о выдаче или возврате направлений к врачу должны быть организованны в виде списка, который упорядочен по первичному ключу – «ФИО врача». Вид списка и метод сортировки определяются вариантом задания.

9.4.8. Информационная система «Регистрация больных в поликлинике» должна осуществлять следующие операции:

– регистрацию нового больного;

– удаление данных о больном;

– просмотр всех зарегистрированных больных;

– очистку данных о больных;

– поиск больного по регистрационному номеру. Результаты поиска – все сведения о найденном больном и ФИО врача, к которому он имеет направление;

– поиск больного по его ФИО. Результаты поиска – список найденных больных с указанием регистрационного номера и ФИО;

– добавление нового врача;

– удаление сведений о враче;

– просмотр всех имеющихся врачей;

– очистку данных о врачах;

– поиск врача по «ФИО врача». Результаты поиска – все сведения о найденном враче, а также ФИО и регистрационные номера больных, которые имеют направление к этому врачу;

– поиск врача по фрагментам «Должность». Результаты поиска – список найденных врачей с указанием ФИО врача, должности, номера кабинета, графика приема;

– регистрацию выдачи больному направления к врачу;

– регистрацию возврата врачом или больным направления к врачу.

9.4.9. Состав данных о больных или врачах, выдаваемых при просмотре всех зарегистрированных больных или просмотре всех имеющихся врачей, определяется студентом самостоятельно, но должен содержать не менее двух полей.

9.4.10. Метод поиска больного по ФИО определяется студентом самостоятельно. Выбранный метод необходимо сравнить с альтернативными методами.

9.4.11. Поиск должности по фрагментам «Должности» должен осуществляться путем систематического обхода АВЛ-дерева поиска. Метод обхода определяется вариантом задания. При поиске врача по фрагментам «Должности» могут быть заданы как полное наименование должности врача, так и его часть. Для обнаружения заданного фрагмента в должности врача должен применяться алгоритм поиска слова в тексте, указанный в варианте задания.

9.4.12. Регистрация выдачи направления к врачу на определенную дату и время должна осуществляться только при отсутствии уже выданного направления к этому же врачу на те же дату и время.

9.4.13. При удалении сведений о враче, должны быть учтены и обработаны ситуации, когда к врачу уже есть записанные на прием больные. Аналогичным образом следует поступать и с удалением сведений больных.

9.5. Продажа авиабилетов

9.5.1. Информационная система для предметной области «Продажа авиабилетов» должна осуществлять ввод, хранение, обработку и вывод данных о:

– пассажирах;

– авиарейсах;

– продаже и возврате авиабилетов.

9.5.2. Данные о каждом пассажире должны содержать:

– Номер паспорта – строка формата «*NNNN-NNNNNN*», где *N* – цифры;

– Место и дата выдачи паспорта – строка;

– ФИО – строка;

– Дату рождения – строка.

Примечание: длина строк (кроме номера паспорта) определяется студентом самостоятельно.

9.5.3. Данные о пассажирах должны быть организованны в виде хеш-таблицы, первичным ключом которой является «номер паспорта». Метод хеширования определяется вариантом задания.

9.5.4. Данные о каждом авиарейсе должны содержать:

– Номер авиарейса – строка формата «*AAA-NNN*», где *AAA* – код авиакомпании (буквы латиницы); *NNN* – порядковый номер авиарейса (цифры);

– Название авиакомпании – строка;

– Аэропорт отправления – строка;

– Аэропорт прибытия – строка;

– Дату отправления – строка;

– Время отправления – строка;

– Количество мест всего – целое;

– Количество мест свободных – целое.

Примечание: длина строк (кроме «номер авиарейса») определяется студентом самостоятельно.

9.5.5. Данные об авиарейсах должны быть организованны в виде АВЛ-дерева поиска, упорядоченного по «номеру авиарейса».

9.5.6. Данные о выдаче или возврате авиабилета должны содержать:

– Номер паспорта – строка, формат которой соответствует аналогичной строке в данных о пассажирах;

– Номер авиарейса – строка, формат которой соответствует аналогичной строке в данных о авиарейсах;

– Номер авиабилета – строка из 9 цифр;

Примечания:

1. Наличие в этих данных записи, содержащей в поле «номер паспорта» значения *X* и в поле «номер авиарейса» значения *Y*, соответственно означает продажу авиабилета пассажиру с номером паспорта *X* на авиарейс с номером *Y*. Отсутствие такой записи означает, что пассажир с номером паспорта *X* не покупал билета на авиарейс с номером *Y*.

2. На один авиарейс может быть продано несколько билетов. Таким образом, могут быть данные, имеющие повторяющиеся значения в некоторых своих полях.

9.5.7. Данные о продаже или возврате авиабилетов должны быть организованны в виде списка, который упорядочен по первичному ключу – «номер авиабилета». Вид списка и метод сортировки определяются вариантом задания.

9.5.8. Информационная система «Продажа авиабилетов» должна осуществлять следующие операции:

– регистрацию нового пассажира;

– удаление данных о пассажире;

– просмотр всех зарегистрированных пассажиров;

– очистку данных о пассажирах;

– поиск пассажира по «номеру паспорта». Результаты поиска – все сведения о найденном пассажире и номерах авиарейсов, на который он купил билет;

– поиск пассажира по его ФИО. Результаты поиска – список найденных пассажиров с указанием номера паспорта и ФИО;

– добавление нового авиарейса;

– удаление сведений об авиарейсе;

– просмотр всех авиарейсов;

– очистку данных об авиарейсах;

– поиск авиарейса по «номеру авиарейса». Результаты поиска – все сведения о найденном авиарейсе, а также ФИО и номера паспортов пассажиров, которые купили билет на этот авиарейс;

– поиск авиарейса по фрагментам названия аэропорта прибытия. Результаты поиска – список найденных авиарейсов с указанием номера авиарейса, аэропорта прибытия, даты отправления, времени отправления;

– регистрацию продажи пассажиру авиабилета;

– регистрацию возврата пассажиром авиабилета.

9.5.9. Состав данных о пассажирах или авиарейсах, выдаваемых при просмотре всех зарегистрированных пассажиров или просмотре всех авиарейсов, определяется студентом самостоятельно, но должен содержать не менее двух полей.

9.5.10. Метод поиска пассажира по ФИО определяется студентом самостоятельно. Выбранный метод необходимо сравнить с альтернативными методами.

9.5.11. Поиск авиарейса по фрагментам названия аэропорта прибытия должен осуществляться путем систематического обхода АВЛ-дерева поиска. Метод обхода определяется вариантом задания. При поиске авиарейса по фрагментам «Аэропорт прибытия» могут быть заданы как полное наименование аэропорта, так и его часть. Для обнаружения заданного фрагмента в «Аэропорту прибытия» должен применяться алгоритм поиска слова в тексте, указанный в варианте задания.

9.5.12. Регистрация продажи авиабилета на определенный авиарейс должна осуществляться только при наличии свободных мест на этот авиарейс.

9.5.13. При удалении данных об авиарейсе должны быть учтены и обработаны ситуации, когда на него уже зарегистрированы пассажиры. Аналогичным образом следует поступать и с удалением данных о пассажирах.

9.6. Обслуживание клиентов оператора сотовой связи

9.6.1. Информационная система для предметной области «Обслуживание клиентов оператора сотовой связи» должна осуществлять ввод, хранение, обработку и вывод данных о:

– клиентах;

– SIM-картах, принадлежащих оператору сотовой связи;

– выдаче или возврате SIM-карт клиентами.

9.6.2. Данные о каждом клиенте должны содержать:

– Номер паспорта – строка формата «*NNNN-NNNNNN*», где *N* – цифры;

– Место и дату выдачи паспорта – строка;

– ФИО – строка;

– Год рождения – целое;

– Адрес – строка.

Примечание: длина строк (кроме номера паспорта) определяется студентом самостоятельно.

9.6.3. Данные о клиентах должны быть организованны в виде АВЛ-дерева поиска, упорядоченного по «номеру паспорта».

9.6.4. Данные о каждой SIM-карте должны содержать:

– Номер SIM-карты – строка формата «*NNN-NNNNNNN*», где *N* – цифра;

– Тариф – строка;

– Год выпуска – целое;

– Признак наличия – логическое.

Примечание: длина строк (кроме «номера SIM-карты») определяется студентом самостоятельно.

9.6.5. Данные об SIM-картах должны быть организованны в виде хеш-таблицы, первичным ключом которой является «номер SIM-карты» Метод хеширования определяется вариантом задания.

9.6.6. Данные о выдаче или возврате SIM-карт клиентами должны содержать:

– Номер паспорта – строка, формат которой соответствует аналогичной строке в данных о клиентах;

– Номер SIM-карты – строка, формат которой соответствует аналогичной строке в данных о SIM-картах;

– Дату выдачи – строка;

– Дату окончания действия – строка.

Примечания:

1. Наличие в этих данных записи, содержащей в поле «номер паспорта» значение *X* и в поле «номер SIM-карты» значение *Y*, соответственно означает выдачу клиенту с номером паспорта *X* SIM-карты с номером *Y*. Отсутствие такой записи означает, что клиенту с номером паспорта *X* не выдавалась SIM-карта с номером *Y*.

2. Одному клиенту может быть выдано несколько SIM-карт. Таким образом, могут быть данные, имеющие повторяющиеся значения в своих полях.

9.6.7. Данные о выдаче или возврате SIM-карты клиентов должны быть организованны в виде списка, который упорядочен по первичному ключу – «номер SIM-карты». Вид списка и метод сортировки определяются вариантом задания.

9.6.8. Информационная система «Обслуживание клиентов оператора сотовой связи» должна осуществлять следующие операции:

– регистрацию нового клиента;

– снятие с обслуживания клиента;

– просмотр всех зарегистрированных клиентов;

– очистку данных о клиентах;

– поиск клиента по «номер паспорта». Результаты поиска – все сведения о найденном клиенте и номера SIM-карт, которые ему выданы;

– поиск клиента по фрагментам ФИО или адреса. Результаты поиска – список найденных клиентов с указанием номера паспорта, ФИО и адреса;

– добавление новой SIM-карты;

– удаление сведений о SIM-карте;

– просмотр всех имеющихся SIM-карт;

– очистку данных о SIM-картах;

– поиск SIM-карты по «номеру SIM-карты». Результаты поиска – все сведения о найденной SIM-карте, а также ФИО и номер паспорта клиента, которому выдана эта SIM-карта;

– поиск SIM-карты по тарифу. Результаты поиска – список найденных SIM-карт с указанием «номера SIM-карты», тарифа, года выпуска;

– регистрацию выдачи клиенту SIM-карты;

– регистрацию возврата SIM-карты от клиента.

9.6.9. Состав данных о клиенте или SIM-карте, выдаваемых при просмотре всех зарегистрированных клиентов или просмотре всех SIM-карт, определяется студентом самостоятельно, но должен содержать не менее двух полей.

9.6.10. Метод поиска SIM-карты по тарифу определяется студентом самостоятельно. Выбранный метод необходимо сравнить с альтернативными методами.

9.6.11. Поиск клиента по фрагментам ФИО или адреса должен осуществляться путем систематического обхода АВЛ-дерева поиска. Метод обхода определяется вариантом задания. При поиске клиента по фрагментам ФИО или адреса могут быть заданы как полное ФИО или адрес, так и их части (например, только фамилия клиента без имени и отчества, только название улицы из адреса). Для обнаружения заданного фрагмента в полном ФИО или адресе должен применяться алгоритм поиска слова в тексте, указанный в варианте задания.

9.6.12. Регистрация выдачи SIM-карты клиенту должна осуществляться только при наличии SIM-карты у оператора сотовой связи (значение поля «Признак наличия» для соответствующей SIM-карты имеет значение «Истина»).

9.6.13. При регистрации выдачи SIM-карты клиенту или возврата SIM-карты клиентом должно корректироваться значение поля «Признак наличия» для соответствующей SIM-карты.

9.6.14. При удалении сведений о SIM-карте должны быть учтены и обработаны ситуации, когда эта SIM-карта уже выдана клиенту. Аналогичным образом следует поступать и с удалением данных о клиентах.

ЛИТЕРАТУРА

1. *[*]Ключарев А. А., Матьяш В. А., Щекин С. В.* Структуры и алгоритмы обработки данных: учеб. пособие. СПб.: ГУАП, 2004.

2. *[*]Вирт Н.* Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. Д. Б. Подшивалова. 2-е изд., испр. М.: ДМК Пресс, 2012. 272 с.

3. *Ахо А., Хопкрофт Д., Ульман Д.* Структуры данных и алгоритмы: монография / Пер. с англ. и ред. А. А. Минько. М.: Вильямс, 2016. 620 с.

4. *Лафорте Р.* Структуры данных и алгоритмы в JAVA / Пер. с англ. Е. Матвеева. 2-е изд. СПб.: Питер, 2017. 704 с.

5. *[*]Ахо А., Хопкрофт Д., Ульман Д.* Построение и анализ вычислительных алгоритмов: учеб. пособие / Пер. с англ. А. О. Слисенко. М.: Мир, 1979. 536 с.

6. *[*]Грин Д., Кнут Д.* Математические методы анализа алгоритмов. М.: Мир, 1987. 120 с.

7. *Макконнелл Д.* Анализ алгоритмов. Активный обучающий подход / Пер. с англ. С. Кулешова, С. Ландо. 2-е доп. изд. М.: Техносфера, 2009. 416 с.

8. *Horvick R.* Data Structures Succinctly Part 1 // Syncfusion Inc., 2013. 111 p.

9. *Скиена С.* Алгоритмы. Руководство по разработке. 2-е изд. / Пер. с англ. СПб.: БХВ-Петербург, 2017. 720 с.

10. *[*]Матьяш В. А., Путилов В. А., Фильчаков В. В.* Структуры и алгоритмы обработки данных. Апатиты, 2000. 80 с.

11. *[*]Кормен Т.* Алгоритмы: построение и анализ / Пер. И. В. Красикова, Н. А. Ореховой, В. Н. Романова. 2-е изд. М.: Вильямс, 2012. 1290 с.

12. *[*]Матьяш В. А., Рогачев С. А.* Структуры и алгоритмы обработки данных: учеб.-метод. пособие. СПб.: ГУАП, 2017.

13. *[*]Кнут Д.* Искусство программирования: в 3 т. Т. 1. Основные алгоритмы. М.: Вильямс, 2014. 720 с.

14. *[*]Седжвик Р.* Алгоритмы на C++: анализ структуры данных, сортировка, поиск, алгоритмы на графах. М.: Вильямс, 2014. 1056 с.

15. *[*]Харари Ф.* Теория графов / Пер.: В. П. Козырева. М.: Мир, 1973. 300 с.

16. ГОСТ 33707-2016 (ISO/IEC 2382:2015) Информационные технологии (ИТ). Словарь. Введ. 2017-09-01. М.: Стандартинформ, 2016. 549 с.

---

[*] Имеются в наличии в библиотеке ГУАП.

СОДЕРЖАНИЕ

Учебное издание

Матьяш Валерий Анатольевич,
Рогачев Сергей Александрович

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие

ISBN: 978-5-8088-1553-7

Редактор *Е. В. Торопова*
Компьютерная верстка *Н. Н. Караваевой*

Подписано к печати 11.01.21. Формат 60×84 1/16.
Усл. печ. л. 4,1. Уч.-изд. л. 4,4. Тираж 50 экз. Заказ № 7.

Редакционно-издательский центр ГУАП
190000, Санкт-Петербург, Б. Морская ул., 67