Т. А. РОМАНЕНКО

ПРОГРАММНЫЕ КОЛЛЕКЦИИ ДАННЫХ ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ

Учебник

САНКТ-ПЕТЕРБУРГ
МОСКВА
КРАСНОДАР
2021

УДК 004
ББК 32.97я73

Р 69 **Романенко Т. А.** Программные коллекции данных. Проектирование и реализация : учебник для вузов / Т. А. Романенко. — Санкт-Петербург : Лань, 2021. — 152 с. : ил. — Текст : непосредственный.

ISBN 978-5-8114-8206-1

Настоящий учебник предназначен для изучения основных типов программных коллекций, хранящих множества данных, фундаментальных структур данных и алгоритмов управления ими. Также в книге предлагается к применению технология проектирования и программирования коллекций, основывающаяся на объектно-ориентированном подходе в программировании.

Учебник предназначен для студентов всех форм обучения направлений подготовки «Информатика и вычислительная техника», «Программная инженерия».

УДК 004
ББК 32.97я73

Обложка
П. И. ПОЛЯКОВА

Предисловие

Организация и представление данных — это естественный процесс, сопровождающий разработку любой программы. Причём представление данных в программе и её алгоритм находятся в неразрывной связи. Иногда выбранный метод решения задачи определяет необходимое представление данных, а иногда первичен выбор структуры данных, который задаёт алгоритм программы.

В теории и практике программирования широко известны фундаментальные структуры данных и алгоритмы (списки, очереди, стеки, двоичные деревья поиска, кучи, очереди с приоритетами, хеш-таблицы, графы, алгоритмы сортировки, поиска). Различные инструментальные среды программирования (IDE) предлагают соответствующие программные средства для этих алгоритмов и структур данных. В качестве примера можно привести библиотеку STL (Standard Template Library), спецификация которой является частью стандартизованного языка ANSI/ISO Standard C++. Библиотека реализована и используется на различных IDE (Windows — C++ Builder и Visual C++, Unix — C++). В частности, для наиболее известных структур данных библиотека предлагает набор шаблонов контейнерных классов, воплощающих такие структуры данных, как векторы, списки, стеки, очереди, очереди с приоритетами кучи, ассоциативные множества с доступом по ключевым значениям данных. Эти контейнеры повсеместно используются при программировании различных приложений, являются базовыми компонентами более сложных форм данных с комбинированной, иерархической структурой.

Данный учебник адресован студентам, изучающим формы организации данных в программах и связанные с ними алгоритмы в рамках дисциплины «Алгоритмы и структуры данных». Учебник содержит теоретические сведения о структурах данных и алгоритмах управления ими, описание методики проектирования и реализации коллекций данных (аналогов классов-контейнеров). Практические задания предназначены для получения навыков проектирования, реализации и использования коллекций данных в виде библиотечных контейнерных классов.

1. Технология разработки коллекций данных

Разработка коллекции данных подчиняется модели жизненного цикла программного обеспечения и состоит из нескольких этапов:

- постановка задачи,
- проектирование,
- программирование,
- отладка и тестирование,
- сопровождение.

1.1. Постановка задачи

На этом этапе анализируются и детализируются все аспекты задания на проектирование коллекции данных. Перед тем как приступить к реализации коллекции, необходимо получить ее описание. Цель постановки задачи — разработать ясное представление того, *что* надо сделать. При этом пока не рассматриваются любые указания того, как будет программироваться коллекция.

Для проектирования коллекции рекомендуется следовать обобщенному принципу абстракции данных [5]. Коллекция представляется как абстрактный тип данных (АТД) — множество данных и множество операций над данными. АТД оговаривает только тип связующей структуры в основе множества данных, режим доступа к множеству (стек, очередь, прямой доступ, доступ по приоритету, доступ по ключу), набор основных операций. Описание операций должно быть строгим и однозначным для того, чтобы точно указать их воздействие на элементы множества, на изменения в состоянии множества, выходные результаты операций. Для этого используется формат АТД [3, 9], являющийся основой для проектирования, а затем и описания *программного интерфейса* между коллекцией и клиентской программой.

Формат АТД должен включать:

- общую характеристику АТД,
- раздел данных,

- раздел операций.

Общая характеристика АТД должна формировать внешнее, абстрактное представление о типе множества, его основных свойствах.

В разделе данных указываются основные параметры множества и вид программной структуры для хранения множества.

В разделе операций задаётся набор операций, выполняемых над множеством или отдельными его элементами. Учитывая общее назначение будущей коллекции, набор её операций должен быть функционально полным, непротиворечивым и неизбыточным. Обычно в набор операций входят операции опроса и установки общих параметров множества, операции доступа, добавления и удаления отдельных элементов, средства для последовательного доступа ко всем элементам множества (итераторы).

Операции в АТД представлены в виде спецификаций. В спецификации операции указываются:

- процесс — краткая формулировка основного действия операции без описания деталей реализации;
- количество и назначение входных параметров;
- ограничения, накладываемые операцией на входные параметры и исходное состояние коллекции (***предусловия***);
- результат выполнения операции, в том числе реакция операции при нарушении предусловий;
- ***постусловия***, отражающие изменения в коллекции, произошедшие в результате выполнения операции.

Предусловия и постусловия составляют контракт между коллекцией и использующей её клиентской программой: ***если программа перед вызовом операции обеспечит соблюдение её предусловий, то коллекция гарантирует, что операция будет успешно завершена и что при этом будут достигнуты ожидаемые результаты операции — постусловия.***

В качестве примера ниже приведён фрагмент АТД вектора, подобного контейнеру *vector*, предлагаемого библиотекой STL [5].

АТД «ВЕКТОР»

Общая характеристика

Вектор — объект, управляющий последовательностью элементов переменной длины. Вектор реализуется как динамический массив, размер которого увеличивается автоматически или по требованию. Элементы массива относятся к одному типу ***T***.

Размерность динамического массива, выделенного для вектора, называется ***ёмкостью*** вектора (***capacity***).

Количество элементов, вставленных в последовательность, называется ***размером*** вектора (***size***).

Набор операций позволяет управлять вектором как последовательностью: вставлять и удалять элементы по указанным позициям, давать прямой доступ по чтению и записи к элементам, в том числе с помощью операции индексирования.

Вектор снабжён итератором произвольного доступа к элементам последовательности, хранящейся в нём.

Данные:

Параметры:

T — тип элементов, хранящихся в векторе.

Capacity — ёмкость вектора.

Size — размер последовательности в векторе.

Структура хранения коллекции:

динамический массив элементов типа ***T — array[capacity].***

Операции:

Конструктор vector ()

Вход: нет.

Предусловия: нет.

Процесс: создание пустого вектора.

Выход: нет.

Постусловия: создан пустой вектор с числом элементов ***size*** = 0 и с емкостью ***capacity*** = 0.

Конструктор vector (count, val)

Вход: ***count*** — емкость вектора, ***val*** — значение элементов.

Предусловия: нет.

Процесс: создание в векторе массива ***array*** ёмкостью ***count,*** заполненного значениями, равными ***val.***

Выход: нет.

Постусловия: создан вектор с последовательностью одинаковых значений ***val, capacity = size = count.***

Конструктор копирования vector (Right)

Вход: *Right* — ссылка на копируемый вектор.

Предусловия: нет.

Процесс: создание копии вектора ***Right.***

Выход: нет.

Постусловия: создана копия вектора ***Right*** с числом элементов ***size*** = = *Right. size ()* в массиве с ёмкостью ***capacity*** = *Right Capacity().*

Опрос емкости вектора capacity ()

Вход: нет.

Предусловия: нет.

Процесс: возврат текущей емкости вектора ***capacity.***

Выход: значение ***capacity*** до того, как вектору потребуется выделить больше памяти.

Постусловия: нет.

Опрос размера последовательности size ()

Вход: нет.

Предусловия: нет.

Процесс: возврат текущего количества значений — ***size***, хранящихся в векторе.

Выход: размер вектора ***size.***

Постусловия: нет.

Резервирование емкости вектора reserve (count)

Вход: минимальная емкость вектора ***count.***

Предусловия: нет.

Процесс: проверка текущей емкости вектора ***capacity*** и перестройка массива ***array***, если ***count > capacity***.

Выход: нет.

Постусловия: ***array[capacity], capacity ≥ count.***

Уменьшение емкости вектора shrink_to_fit ()

Предусловия: нет.

Процесс: уменьшение размера массива ***array*** за счёт освобождения неиспользованной памяти.

Выход: нет.

Постусловия: ***array[capacity], capacity = size.***

Оператор индексирования [pos]

Вход: позиция элемента ***pos.***

Предусловия: нет.

Процесс: возврат ссылки на элемент ***array*** [***pos***].

Контроль выхода за границы массива ***array*** не осуществляется.

Выход: ссылка на элемент массива ***array***[***pos***]. Выход не определен при нарушении предусловия.

Постусловия: нет.

Доступ к элементу at(pos)

Предусловия: ***0 ≤ pos < size.***

Процесс: возврат ссылки на элемент ***array*** [***pos***].

Выход: ссылка на элемент массива ***array***[***pos***] или генерация исключения ***out_of_range*** при невыполнении предусловия.

Постусловия: нет.

Операция добавления в конец push_back (val)

Вход: ***val*** — значение элемента типа T.

Предусловия: нет.

Процесс: добавление элемента ***val*** в конец вектора.

Выход: нет.

Постусловия: элемент со значением ***val*** добавлен в конец вектора, размер вектора ***size = size* + 1**. Если ***size > capacity***, то ***capacity = 2×capacity***.

Операция добавления элементов в указанную позицию insert(val, pos)

Вход: ***val*** — значения элемента(ов), ***pos*** — позиция в векторе для вставки.

Предусловия: ***0 ≤ pos ≤ size***.

Процесс: добавление элемента ***val*** в позицию ***pos***, ***size = size + 1***.

Если ***size > capacity***, то удвоение емкости массива ***array***.

Выход: нет.

Постусловия: элемент со значением ***val*** добавлен в позицию, ***size = size + + 1***. Если ***size > capacity***, то ***capacity = 2×capacity***.

Операция удаления элемента из указанной позиции erase (pos)

Вход: ***pos*** — позиция в векторе для удаления.

Предусловия: ***0 ≤ pos < size***.

Процесс: удаление элемента из позиции ***pos***, ***size = size – 1***.

Выход: нет.

Постусловия: элемент в позиции ***pos*** удалён, ***size = size – 1***.

Запрос итератора произвольного доступа begin()

Вход: нет.

Предусловия: размер списка *size()* ≠ 0.

Процесс: формирование итератора произвольного доступа, установленного на первый элемент последовательности в векторе.

Выход: итератор произвольного доступа, установленный на первый элемент ***begin()***, или «неустановленный» итератор ***end()*** при невыполнении предусловия.

Постусловия: нет.

Запрос «неустановленного» итератора end()

Вход: нет.

Предусловия: нет.

Процесс: формирование «неустановленного» итератора, указывающего на позицию, следующую за последним элементом последовательности в векторе.

Выход: «неустановленный» итератор произвольного доступа ***end()***.

Постусловия: нет.

Запрос итератора произвольного доступа rbegin()

Вход: нет.

Предусловия: размер списка *size()* ≠ 0.

Процесс: формирование итератора произвольного доступа, установленного на последний элемент последовательности в векторе ***end()***.

Выход: итератор произвольного доступа к элементам ***rbegin()*** или «неустановленный» итератор ***rend()*** при невыполнении предусловия.

Постусловия: нет.

Запрос «неустановленного» итератора rend()

Вход: нет.

Предусловия: нет.

Процесс: формирование «неустановленного» итератора произвольного доступа, указывающего на позицию, предыдущую первому элементу последовательности в векторе ***rend()***.

Выход: «неустановленный» итератор произвольного доступа.

Постусловия: нет.

-
- другие запросы и операции **АТД ВЕКТОР**
-

КОНЕЦ АТД «ВЕКТОР»

Особо необходимо отметить использование для коллекции дополнительных объектов — итераторов. Необходимость итераторов связана с особенностями программирования коллекций. Поскольку данные хранятся в скрытой структуре коллекции, доступ к ним для клиентской программы невозможен. Можно использовать операции доступа к отдельным элементам по номеру позиции в последовательности. Но использование таких операций не всегда целесообразно. Например, последовательный перебор в цикле всех элементов по номерам зачастую приводит к большим затратам трудоёмкости на повторные пробеги операций от начала последовательности к меняющимся позициям.

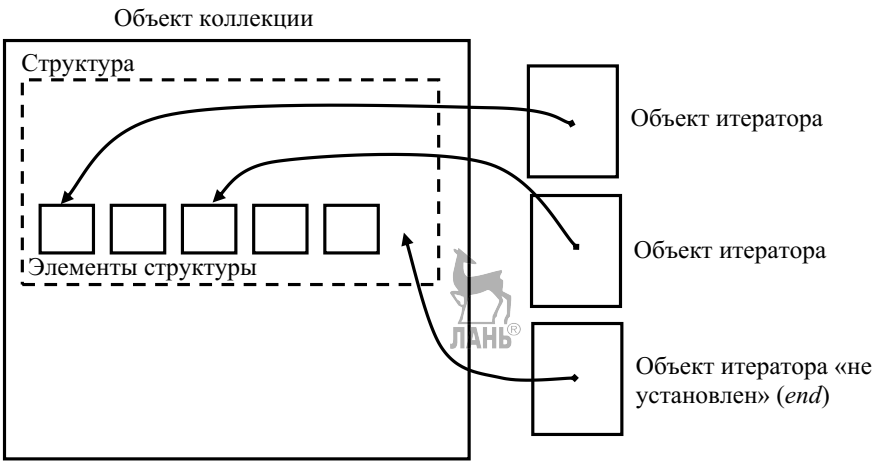


Рис. 1. Взаимосвязь итераторов и коллекции

Итератор — это объект, который дает возможность клиентской программе осуществлять последовательное продвижение от элемента к элементу коллекции и получать доступ к значениям элементов по чтению и/или записи. По свойствам и механизмам итератор похож на адресный указатель. Его можно перемещать по элементам как адресный указатель по ячейкам памяти, выполнять доступ по чтению и записи к элементу подобно операции разыменования (*) адресного указателя. Поэтому итератор часто определяют как ***обобщённый указатель***.

В большинстве контейнеров применяются различные итераторы в зависимости от структуры коллекции. Это, прежде всего, четыре типа итераторов:

– прямой итератор (тип **iterator**) даёт доступ по чтению и записи к данным и позволяет при увеличении перемещаться по элементам от первого к последнему;

– обратный итератор (тип **reverse_iterator**) даёт доступ по чтению и записи к данным и позволяет при увеличении перемещаться по элементам от последнего к первому;

– константный прямой итератор (тип **const_iterator**) даёт доступ по чтению к данным и позволяет при увеличении перемещаться по элементам от первого к последнему;

– константный обратный итератор (тип **const_reverse_iterator**) даёт доступ по чтению к данным и позволяет при увеличении перемещаться по элементам от последнего к первому.

Обратный итератор применяется так же, как прямой. Разница состоит в реализации операторов перехода к следующему и предыдущему элементам. Для прямого итератора операция инкремента ++ даёт доступ к следующему элементу контейнера, тогда как для обратного — к предыдущему.

Контейнеры STL предоставляют и другие типы итераторов:

– итераторы ввода/вывода позволяют многократно производить чтение или запись в любой позиции;

– итераторы произвольного доступа позволяют многократно производить чтение или запись в любой позиции, переходить к элементам в прямом или обратном направлении, на одну позицию или на произвольное расстояние за один шаг.

Для получения итераторов, установленных в исходные состояния, контейнеры обладают такими функциями, как **begin()** и **end().** Функция **begin()** возвращает итератор, который указывает на первый элемент контейнера (при наличии в контейнере элементов). Функция **end()** возвращает итератор,

который указывает на следующую позицию после последнего элемента. Соответственно функция **rbegin()** возвращает итератор, который указывает на последний элемент контейнера (при наличии в контейнере элементов). Функция **rend()** возвращает итератор, который указывает на предыдущую позицию для первого элемента. Если контейнер пуст, то итераторы, возвращаемые функциями **begin(), end(), rbegin()** и **rend(),** совпадают с итератором **end()**.

С итераторами можно проводить следующие операции:

– *iter: получение ссылки на элемент, на который указывает итератор;

– ++iter: перемещение прямого итератора к следующему элементу, а обратного итератора — к предыдущему;

– **--**iter: перемещение прямого итератора к предыдущему элементу (итераторы для односвязных структур не поддерживают операцию декремента), а обратного итератора — к следующему;

– iter1 == iter2: два итератора равны, если они связаны с одним и тем же контейнером и указывают на один и тот же элемент;

– iter1 **!=** iter2: два итератора не равны, если они связаны с разными контейнерами или указывают на разные элементы.

Все перечисленные операции выполняются корректно, когда итератор указывает на какой-либо элемент коллекции (итератор «установлен»). Исходную установку итераторов выполняют функции **begin()** или **rbegin()**.

Если итератор равен итератору **end()** или **rend(),** то он находится в состоянии «не установлен». Это состояние возникает при выдвижении итератора за пределы коллекции в результате операций инкремента или декремента или при пустой коллекции.

Управление и использование итераторов выполняет клиентская программа с помощью операций итератора.

Пример:

```
#include <vector>
#include <iostream>
```

```
using namespace std;
void main()
{vector<int> v = {1, 2, 3, 4, 5};   //v контейнер «Вектор»
 vector<int>::iterator iter = v.begin();//iter - итератор
 while(iter!=v.end())  // пока iter не равен итератору end
 {
*iter = (*iter) * (*iter); // возводим число в квадрат
++iter;          //передвинем итератор к следующему элементу
 }
for(iter = v.begin(); iter!=v.end();++iter) //вывод элементов
 cout<< *iter << endl;
}
```

Для итераторов коллекции также должны быть разработаны форматы АТД, задающие программный интерфейс итератора и клиентской программы. Пример формата АТД итератора произвольного доступа для контейнера vector библиотеки STL:

АТД «ИТЕРАТОР произвольного доступа» для контейнера «ВЕКТОР»

Двунаправленный итератор произвольного доступа — объект, который по отношению к ВЕКТОРУ играет роль указателя. Итератор позволяет многократно производить чтение или запись в любой позиции, переходить к элементам в прямом или обратном направлении, на одну позицию или на произвольное расстояние за один шаг. Итератор находится в одном из состояний: «установлен» — указывает на элемент последовательности, хранящейся в векторе; «не установлен» — итератор совпадает с неустановленным итератором **end()** или **rend().**

Параметры:

Указатель на вектор ***V***.

Указатель на текущий элемент в векторе ***cur***.

Операции:

Оператор доступа по чтению/записи к значению текущего элемента*

Вход: нет.

Предусловия: итератор находится в состоянии «*установлен*».

Процесс: формирование ссылки на значение данных текущего элемента ***cur***.

Выход: ссылка на данные текущего элемента или сообщение об ошибке при невыполненном предусловии.

Постусловия: нет.

Оператор инкремента ++

Вход: нет.

Предусловия: итератор находится в состоянии *«установлен»*.

Процесс: перемещение указателя итератора на следующий в последовательности элемент.

Выход: итератор, установленный на следующий элемент, или итератор в состоянии «не установлен» **end()**.

Постусловия: итератор указывает на следующий элемент последовательности или равен итератору **end()**.

Оператор декремента - -

Вход: нет.

Предусловия: итератор находится в состоянии *«установлен»*.

Процесс: перемещение указателя итератора на предыдущий элемент.

Выход: итератор, установленный на предыдущий элемент, или итератор в состоянии «не установлен» **rend()**.

Постусловия: итератор указывает на предыдущий элемент последовательности или равен итератору **rend()**.

Оператор установки итератора на N элементов вперед — X+N

Вход: смещение позиции **N**.

Предусловия: итератор находится в состоянии *«установлен» и* $N > 0$.

Процесс: перемещение указателя ***cur*** на ***N*** позиций вперёд к концу последовательности.

Выход: итератор, установленный на позицию ***cur + N,*** или итератор в состоянии «не установлен» **end()**, если ***cur + N*** > размера вектора ***size()*** или итератор в неизменённой позиции при невыполненном предусловии.

Постусловия: итератор установлен на позицию ***cur + N***, или итератор в состоянии «не установлен» **end()**.

***Оператор проверки равенства итераторов* ==**

Вход: итератор ***X*,** итератор ***Y***.

Предусловия: нет.

Процесс: проверка для ***X*** и ***Y*** параметров ***V*** и ***cur*** на совпадение.

Выход: ***true*** — итераторы ***X*** и ***Y*** равны***, false* —** итераторы ***X*** и ***Y*** не равны.

Постусловия: нет.

-
- другие запросы и операции **АТД ИТЕРАТОР**
-

КОНЕЦ АТД «ИТЕРАТОР произвольного доступа» для **АТД «ВЕКТОР»**

1.2. Проектирование структуры класса для коллекции

После составления формата АТД, определяющего для коллекции внешние свойства и семантику операций, можно приступать к её программной реализации. При программировании объекта, представленного в виде АТД, наиболее удачным является объектно-ориентированный метод программирования. Сильной стороной этой методики является сокрытие реализации за счёт действия принципа инкапсуляции. Коллекция определяется в виде класса — контейнера, объединяющего в единое целое данные, конструкторы и операции, итераторы.

В **раздел данных** класса вводятся именованные поля для общих параметров коллекции (емкость коллекции, текущий размер коллекции и т. п.) и структура данных, хранящая данные. Впоследствии, по мере разработки методов класса, для них в раздел данных могут добавляться вспомогательные поля и структуры данных (адрес массива или адрес первого элемента связной структуры, индекс или адрес элемента, следующего за последним элементом,

указатель текущего элемента, структура для вспомогательной очереди или стека и т. п.).

Для структуры, содержащей данные коллекции, применяются две альтернативные формы хранения — массив или связная структура на основе адресных указателей. Учитывая произвольное количество данных, включаемых в коллекцию, как правило, массив или связная структура размещаются в динамической памяти.

Массив предпочтителен для статических множеств (словарей), редко меняющих состав и количество элементов. В словарях на базе массива преобладают операции поиска по значению, прямого доступа к позиции через быстродействующую операцию индексирования [] в массиве. Другим преимуществом массива является размещение новых данных в элементах массива без запросов дополнительной динамической памяти.

Недостатком использования массива является его постоянная ёмкость, которая ограничивает размер растущего множества. В этом случае можно использовать массив, размещенный в динамической памяти, с организацией увеличения его ёмкости по мере заполнения новыми данными.

Альтернативой массиву является связная структура, использующая адресные указатели. В коллекцию вводится вспомогательный объект для хранения отдельного элемента структуры (элемент списка, узел дерева, вершина графа и т. п.). По мере пополнения коллекции новыми данными в динамической памяти создаются объекты — элементы структуры — и встраиваются в связную структуру с помощью адресных указателей.

Также коллекция дополняется итераторами — объектами, посредством которых клиентская программа получает доступ по чтению и/или записи к хранящимся в коллекции данным. Итераторы определяются в виде внутренних классов в открытой секции класса коллекции. Благодаря этому интерфейс итераторов доступен для клиентских программ. Для коллекции можно создавать произвольное количество связанных с ней объектов — итераторов, и использовать их в клиентской программе независимо и одновременно.

Далее в пособии рассматривается реализация для коллекции только прямого и обратного итераторов.

На этапе реализации коллекции выполняется разработка программного интерфейса коллекции. В **раздел методов** класса вводятся операции, с помощью которых клиентская программа может управлять коллекцией в целом и манипулировать её отдельными данными. Прежде всего, включаются операции, заданные в формате АТД, который задаёт точную спецификацию операций: их назначение, входные параметры и результаты, ограничения для входных параметров. На основе этой спецификации задаются программные прототипы будущих методов с указанием типов и имен входных переменных, результатов.

Все интерфейсные методы помещаются в секцию объекта коллекции, открытую для клиентских программ.

1.3. Трудоёмкость операций коллекции

Трудоёмкость операций для коллекций, работающих с большими наборами данных, становится очень важной характеристикой. Она напрямую зависит от количества объектов в коллекции ***n*** = *size* () — размера коллекции. Способ измерения ***n*** зависит от конкретной структуры данных, на базе которой организована коллекция. Значение ***n*** может отражать количество значений в массиве, число элементов связного списка, число узлов дерева поиска, число вершин и рёбер графа и т. п.

При выборе предпочтительными являются алгоритмы, которые выполняют минимально возможное число действий, т. е. имеют минимальную трудоёмкость.

Для предварительной оценки трудоёмкости алгоритмов на этапе проектирования используется нотация «**большое О**» (***Big-O***) [3, 6, 9]. Основная идея применения нотации ***Big-O*** заключается в том, что необходимо определить, как будет вести себя алгоритм при росте размера коллекции ***n***. Нотация ***Big-O*** даёт аппроксимированную верхнюю границу трудоёмкости алгоритма при неограниченном росте ***n***.

Технику получения аналитической оценки ***Big-O*** можно продемонстрировать на примере алгоритма сортировки массива методом выбора, содержащего ***n*** значений (табл. 1).

Таблица 1

Selection_Sort (A, n**)**	**Стоимость**	**Число повторений**
1. ▷ A — массив		
2. ▷ n — число значений в массиве		
3. **for** i←1 **to** n	*C1*	*n*
4. **do** imin←i	*C2*	*n – 1*
5. **for** j←i + 1 **to** n	*C3*	$\sum_{i=2}^{n} n-i$
6. **do if** A[j]<A[imin] **then** imin←j	*C4*	$\sum_{i=2}^{n} n-i$
7. temp←A[imin]	*C5*	*n – 1*
8. A[imin] ←A[i]	*C6*	*n – 1*
9. A[i] ←temp	*C7*	*n – 1*

Сложив вклады всех строк, получим

$$C1\times n + C2\times (n-1) + C3\times \sum_{i=1}^{n} n-i + C4\times \sum_{i=1}^{n} n-i + C5\times (n-1) + C6\times (n-1) +$$

$$+ C7\times (n-1) =$$

$$= C1\times n + (C2 + C5 + C6 + C7)\times (n-1) + (C3 + C4)\times \sum_{i=2}^{n} n-i =$$

$$= C1\times n + (C2 + C5 + C6 + C7)\times (n-1) + (C3 + C4)/2\times (n-1)\times (n-2) =$$

$$= C1\times n + (C2 + C5 + C6 + C7)\times (n-1) + (C3 + C4)/2\times n^2 - 3\times (C3 + C4)/2\times n -$$

$$- (C3+C4) =$$

$$= (C3 + C4)/2\times n^2 + (C1 + C2 - 3/2\times (C3 + C4) + C5 + C6 + C7)\times n - (C2 + C3 +$$

$$+ C4 + C5 + C6 + C7).$$

То есть можно утверждать, что время выполнения алгоритма прямо пропорционально функции $f(n) = a_2 \times n^2 + a_1 \times n + a_0$, зависящей от размера задачи ***n***.

Вообще, если $f(n)$ есть полином вида $a_k \times n^k + a_{k-1} \times n^{k-1} + ... + a_1 \times n + a_0$, то можно доказать, что $f(n) < C \times n^k$ при $n \geq n_0$.

В частности, полином $a_k \times n^k + a_{k-1} \times n^{k-1} + ... + a_1 \times n + a_0 < C \times n^k$ при $n_0 = 1$ и $C = |a_k| + |a_{k-1}| + ... + |a_1| + |a_0|$.

Приведённая к старшей степени оценка трудоёмкости $g(n) = C \times n^k$ называется порядком роста трудоёмкости алгоритма и обозначается как **O(g*(n)*)** или **O(n^k)**.

Таким образом, для рассмотренного алгоритма сортировки оценка трудоёмкости имеет вид **O(n^2).**

Анализ трудоёмкости вновь разрабатываемого алгоритма подчас становится сложной математической задачей. В результате теоретического анализа наиболее известных фундаментальных алгоритмов получены оценки их трудоёмкости в виде нотации ***Big-O*** [3, 6, 9]. Если для реализации выбран один из известных алгоритмов, то можно воспользоваться этими оценками.

На практике для быстрого получения оценки ***Big-O*** можно воспользоваться более грубым приёмом анализа внешних признаков программной структуры алгоритма**.** Если в алгоритме отсутствует зависимость между его действиями и количеством обрабатываемых данных — ***n***, то его трудоёмкость не зависит от ***n*** и имеет оценку **O(*1*).** Если в алгоритме встречается цикл, число повторений которого пропорционально ***n***, то можно сказать, что его трудоёмкость имеет вид **O(*n*).** Если в алгоритме используются два цикла, число повторений которых пропорционально ***n***, причем один цикл вложен в другой, то трудоёмкость алгоритма имеет вид **O(n^2).** Тройная вложенность таких циклов свидетельствует о трудоёмкости **O(n^3),** и так далее. Алгоритмы, выполняющие разбиение задачи на две подзадачи и выбор для решения одной из подзадач (алгоритм двоичного поиска значения в упорядоченном массиве, алгоритмы операций для двоичного дерева поиска и т. п.), имеют оценку трудоёмкости

***O*(*log₂n*).** Алгоритмы, выполняющие разбиение задачи на подзадачи и комбинирующие решения подзадач в одно общее решение, имеют оценку ***O*(*n* × *log₂n*)** (алгоритмы сортировки методом разделения, методом слияния). Алгоритмы, выполняющие разбиение задачи и перебор всех возможных комбинаций решений подзадач, имеют трудоёмкость ***O*(2ⁿ)** и называются экспоненциальными. На практике такие алгоритмы очень редко используются.

Графики, приведенные на рисунке 2, наглядно иллюстрируют, насколько отличаются трудоёмкости алгоритмов с различным порядком роста.

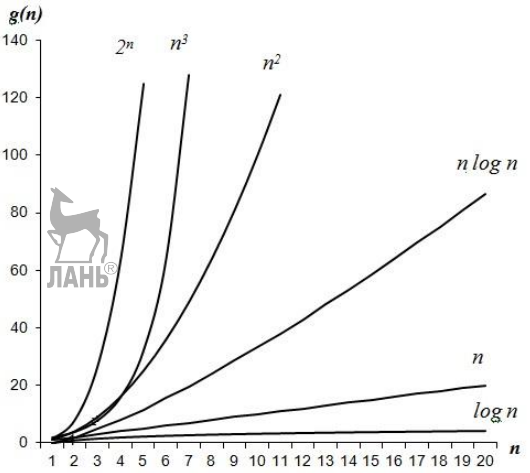


Рис. 2. Сравнение различных порядков роста трудоёмкости алгоритмов

Необходимо отметить, что трудоёмкость алгоритмов для коллекции в значительной мере определяется спецификой структуры данных, на которой базируется коллекция. Например, если в коллекции используется связный список элементов, то большинство алгоритмов будут иметь нотацию трудоёмкости ***O*(*n*)**. Если основой коллекции является двоичное дерево поиска, то трудоёмкость алгоритмов оценивается нотацией ***O*(*log₂n*)**. Эти оценки должны служить ориентиром при выборе и разработке алгоритмов для коллекции.

Для некоторых алгоритмов и структур данных при оценке трудоёмкости приходится рассматривать **худший и средний варианты работы**. Работа таких алгоритмов и структур зависит от порядка элементов в множестве данных. Например, алгоритмы сортировки методом вставки, обменной сортировки

выполняют максимальное число сравнений и перестановок, если значения в сортируемом массиве имеют обратную упорядоченность. Двоичное дерево поиска вырождается в линейный список, если оно формируется последовательными вставками возрастающих по значению ключа данных. Трудоёмкость в таком дереве возрастает от оценки $O(log_2 n)$ до $O(n)$.

В таком случае проводится анализ трудоёмкости для худшего и среднего режимов работы алгоритмов. Оценка трудоёмкости для худшего случая определяет максимальное время выполнения алгоритма. Эта оценка важна, если реализованный алгоритм будет встроен в систему реального времени, для которой оговаривается допустимое время ответа на входящий запрос. Анализ трудоёмкости для среднего случая позволяет оценить среднюю производительность алгоритма. Как правило, анализ среднего варианта носит вероятностный характер, и выполнить его намного сложнее, чем для худшего случая.

Предлагается следующая методика использования оценки трудоёмкости ***Big-O*** для выбора и реализации алгоритмов операций для коллекции. Для алгоритма или структуры задаётся ожидаемая оценка трудоёмкости. Все решения при выборе и разработке алгоритмов операций должны быть направлены на то, чтобы трудоёмкость реализованного алгоритма соответствовала этой оценке. В дальнейшем на этапе тестирования проводится программное измерение трудоёмкости операций коллекции и сопоставление экспериментальных оценок с ожидаемой оценкой. Если для операции оценки значительно отличаются, то необходимо либо обосновать более трудоёмкий алгоритм, либо внести в него изменения для уменьшения трудоёмкости, либо выбрать другой алгоритм с меньшей трудоёмкостью.

1.4. Программирование коллекции

В данном пособии рассматривается программная реализация коллекций на языке программирования C++. Для намеченных на стадии проектирования

объектов разрабатываются определения соответствующих классов, аналогичных контейнерным классам библиотеки STL [5].

В основе программирования контейнеров лежит понятие **шаблонного класса**. Шаблон позволяет разрабатывать класс без уточнения типов данных, хранящихся в контейнере. При объявлении в клиентской программе объекта-коллекции как параметр шаблона указывается конкретный тип данных.

Описание шаблонного класса и его реализацию (определение методов класса) следует оформить в едином тексте и файле. Связано это с особенностями технологии компиляции шаблонного класса в момент создания объекта в клиентской программе. Происходит генерация нового объявления класса и его методов для конкретного типа данных, указанного в качестве параметра шаблона. Для дальнейшей обработки компилятором они должны находиться в едином тексте.

Ниже дано обобщённое определение класса, которое может быть использовано для программирования коллекции.

```
//заголовочный файл “Collection.h”
template <class T>
class Collection        //шаблонный класс для объекта коллекции
{//скрытая секция объекта
    protected:
    class Node          //класс для элемента связной структуры
       {public:
        T item;    //значение данных
        Node *p1;//указатели на соседние элементы структуры
       Node *p2;
        Node(T item); // конструктор элемента
       };               //конец класса Node
    //скрытые поля и структуры Collection
         int size;      //размер коллекции (число элементов с данными)
         int capacity;      //ёмкость коллекции
```

```
T* data;          //или адрес массива
Node* first;      //или заголовки для связного списка
Node* last;       //или заголовки для связного списка
Node* root;       //или корень связного дерева
```

//прототипы внутренних методов Collection

-
-
-

//открытая секция объекта коллекции (интерфейс объекта)

public:

```
Collection ( );  //конструктор по умолчанию
Collection (…);  //конструктор с параметрами
Collection (const Collection<T> &); //конструктор копирования
~Collection ( ); //деструктор
```

//прототипы методов для операций, заданных форматом АТД Collection

-
-
-

//класс(ы) итератора(ов) для объекта Collection

```
class Iterator     //прямой итератор
{Collection* p;    //указатель на объект коллекции
 int cur;          //или индекс текущего элемента
 Node* cur;        //или адрес текущего элемента
 public:
 //конструкторы для методов begin(), end() класса Collection
 Iterator ( )          //конструкторы итератора
 Iterator(Node*)  //или конструктор с установкой на адрес элемента
 Iterator(int)    //или конструктор с установкой на индекс элемента
 //интерфейс итератора
```

//прототипы операций, заданных форматом АТД «Прямой итератор»

-
-
-

}; **//конец класса Iterator**

```
class reverse_Iterator      //обратный итератор
{Collection* p;   //указатель на объект коллекции
 int cur;         //или индекс текущего элемента
 Node* cur;       //или адрес текущего элемента
 public:
```

//конструкторы для методов *rbegin(), rend()* класса Collection

```
 reverse_Iterator ( )            //конструктор без параметра
 reverse_Iterator (Node*) //или конструктор с установкой на адрес элемента
 reverse_Iterator (int)   //или конструктор с установкой на индекс элемента
```

//интерфейс итератора

//прототипы операций АТД «Обратный итератор»

-
-
-

}; **//конец класса reverse_Iterator**

```
 friend class Iterator;
 friend class reverse_Iterator;
 iterator begin( );          //запрос итератора begin( )
 iterator end( );            //запрос итератора end( )
 reverse_Iterator rbegin( ); //запрос итератора rbegin( )
 reverse_Iterator rend( );   //запрос итератора rend( )
```

}; **//конец класса Collection**

//реализации методов классов Collection, Iterator

-
-
-

//конец заголовочного файла “Collection.h”

Объявление класса обычно содержит только прототипы методов за исключением небольших, подставляемых методов. Реализация методов обычно размещается после определения класса и включает тексты программ конструкторов, деструктора, интерфейсных и вспомогательных методов класса коллекции и классов итераторов.

Некоторые методы шаблонного класса используют операции сравнения данных, хранящихся в коллекции. Если заданный в шаблоне тип данных — T — нестандартный и определён в клиентской программе, то для этого типа необходимо переопределение операций отношения (>, <, == и !=). **Конкретные операции отношений определяются при программировании методов и вносятся в формат АТД как предусловия соответствующих операций**.

При программировании методов классов следует придерживаться правил и рекомендаций к стилю программирования, обеспечивающих **модульность, модифицируемость, надежность, читабельность** разработанных для коллекции программ.

Модульность является основополагающим принципом структуры больших программ. Декомпозиция большой программы на небольшие взаимосвязанные программные модули резко снижает сложность программирования, способствует пониманию работы программы в целом и отдельных её модулей, исключает избыточность кода, обеспечивает локализацию ошибок в пределах модулей и облегчает отладку программы. Модульная структура облегчает и модификацию программы. Любое вносимое изменение касается, как правило, одного или нескольких модулей. Модульность изолирует модификации, поскольку изменения в одном модуле не затрагивают остальных.

Естественное выделение методов в АТД и в классе коллекции автоматически обеспечивает модульность при разработке программ для коллекции. Но даже при программировании отдельного метода необходимо помнить об этом принципе. Если алгоритм некоторого метода достаточно громоздкий и содержит в себе несколько вспомогательных алгоритмов, необходимо оформить части модуля в виде отдельных функций. Если в программе метода или нескольких методов встречается фрагмент с однотипными действиями, то его также желательно оформить в виде функции. Все вспомогательные функции фиксируются в определении класса коллекции как скрытые методы.

Надёжность работы методов коллекции обеспечивается за счёт защиты внутренних данных и структуры коллекции от произвольных манипуляций со стороны клиентской программы. Все данные и структура помещаются в скрытую секцию коллекции. Если клиентской программе позволены чтение и/или запись значения в какую-либо скрытую переменную, в интерфейс объекта вводятся интерфейсные методы для доступа к значению скрытых переменных коллекции.

Также должна обеспечиваться надёжность и устойчивость работы интерфейсных методов при вызовах с неверными входными параметрами. Прежде всего, по ходу работы метода выполняется проверка всех предусловий, указанных для соответствующей операции в формате АТД. **Обнаружив нарушенное предусловие, метод не должен прекращать работу клиентской программы или выводить на экран монитора диагностическое сообщение.** В этом случае возможны два сценария действий модуля. Во-первых, метод может прекратить работу, возвратив клиентской программе соответствующий признак ошибки. Возврат признака можно выполнить с помощью оператора **return** или записав его в специально предназначенный входной параметр — ссылку. Кроме того, метод может прекращать работу, используя механизм исключительной ситуации. Метод подаёт сигнал о возникшей ошибке, генерируя (throwing) исключительную ситуацию. Клиентская программа

должна реагировать на исключительную ситуацию, перехватывая её и выполняя обработку ошибок. Генерация исключений обязательно должна быть оговорена в формате АТД как особая форма выхода операции.

Чтобы избежать многих ошибок и получить надёжно функционирующие методы коллекции, рекомендуется придерживаться дополнительных правил в стиле программирования:

- методы коллекции не должны использовать переменные и константы, объявленные в клиентской программе;
- если методы специально не предназначены для ввода с клавиатуры или вывода данных на экран, то они не должны выполнять непосредственный ввод с клавиатуры и вывод на экран;
- следует избегать статических переменных, используемых несколькими объектами;
- следует избегать глобальных переменных, используемых различными методами объекта;
- рекомендуется использовать преимущественно передачу методам параметров по значению;
- рекомендуется использовать передачу методам параметров по ссылке, если значения параметров занимают большой объём памяти;
- рекомендуется использовать входные параметры-ссылки для передачи из метода нескольких значений-результатов;
- рекомендуется использовать перед входным параметром-ссылкой ключевое слово const, если он не должен меняться методом;
- рекомендуется использовать перед выходным параметром-ссылкой для результата ключевое слово const, если результат не должен меняться в клиентской программе;
- методам запрещено **создавать** в качестве результата переменные или объекты, размещённые в динамической памяти.

Для того чтобы алгоритмы методов можно было легко понимать, исходные тексты методов должны быть хорошо структурированы, содержать комментарии для основных переменных и фрагментов. Рекомендуется использовать в тексте программ межстрочные интервалы для отображения отдельных программных фрагментов и дрейф строк вправо для отображения вложенности. Имена идентификаторов переменных и методов должны отражать их смысл.

Соблюдение всех этих требований и рекомендаций значительно улучшает **читабельность** программ класса, ускоряет анализ программ в процессе их отладки и последующего сопровождения.

1.5. Отладка и тестирование

Целью отладки и тестирования является обнаружение ошибок в алгоритмах и программах и оценка эффективности методов. Реализовав класс коллекции, необходимо провести его двойное тестирование.

Сначала нужно проверить корректность всех методов классов для коллекции и для итераторов. Для этого создаётся небольшая программа, вызывающая через меню все методы и отображающая на экране монитора результаты их работы. Тестирование отдельных методов позволяет провести первоначальный этап отладки программ и выявить семантические ошибки, допущенные при написании программ.

Программа-меню, тестирующая корректность методов, создаёт объект коллекции, первоначально не содержащий данных. С помощью меню можно в произвольном порядке задать с клавиатуры любую операцию и любые значения её входных параметров. Тестирующая программа выполняет вызов метода коллекции для выбранной операции **без предварительной проверки правильности входных параметров и состояния коллекции**. После возврата из операции тестирующая программа должна выводить возвращённые результаты операции. При правильной работе операции её результаты и

состояние структуры должны соответствовать выходам и постусловиям, сформулированным для них в формате АТД.

Для отслеживания изменений в структуре данных в коллекции необходимо реализовать метод вывода структуры на экран и включить его вызов в меню. Метод должен отображать содержимое скрытой структуры коллекции в адекватной ей форме (последовательность значений для списка, древовидную структуру значений для дерева и т. п.).

В процессе тестирования отдельных методов необходимо проверить их устойчивость путём подбора образцов входных параметров, включающих граничные допустимые значения, неверные значения, нарушающие предусловия методов. Кроме того, если метод имеет худший случай работы (вырожденное дерево поиска, переполненная хеш-таблица и т. п.), то необходимо воспроизвести этот случай и проверить работу методов в этих условиях.

Дополнительно для измерения трудоёмкости операций вставки, удаления и поиска в соответствующие методы вводится подсчёт числа просмотренных элементов во время работы операции (число просмотренных элементов списка, число пройденных узлов дерева, число зондирований в хеш-таблице и т. п.). Для опроса трудоёмкости выполненной операции в меню вводится отдельный пункт.

После первого этапа проверки корректности отдельных методов работа класса тестируется для коллекций, содержащих большие объёмы данных. Целью тестирования является получение статистических оценок трудоёмкости ***Big-O*** для основных операций коллекции, а также проверка устойчивости работы коллекции, хранящей большой объём данных, в условиях потока вызовов различных методов. Прежде всего, необходимо получение статистических оценок для трёх основных операций коллекции — поиска, вставки и удаления данных.

Для статистического тестирования операций используется специально разработанная программа (драйвер тестирования эффективности),

генерирующая поток операций с заданными свойствами для коллекции заданного размера. Единицы измерения размера коллекции и трудоёмкости операций определяются спецификой коллекции.

Драйвер тестирования создаёт модель потока операций гипотетической клиентской программы, имеющую следующие характеристики:

- поток состоит из чередующихся равновероятных операций поиска, вставки и удаления данных;
- общее количество операций пропорционально размеру коллекции;
- операции задают равновероятные позиции и/или равновероятные значения ключей данных в коллекции;
- для операций задаётся одинаковая вероятность промаха. Для операций поиска или удаления промахом считается поиск или удаление значения, отсутствующего в коллекции. Если в коллекции запрещено хранение данных с одинаковыми значениями, то вставка нового значения, уже присутствующего в коллекции, ведёт к промаху операции вставки.

Драйвер тестирования должен обеспечивать измерение трудоёмкости для среднего и худшего режимов работы коллекции. Порядок действий программы-драйвера следующий. Перед тестированием программе задаётся режим работы коллекции (средний или худший случай), её размер, вероятность промахов. По заданным параметрам программа формирует исходное состояние коллекции, занося в неё выборку данных. Во время тестирования программа генерирует операции с заданными свойствами, фиксируя при этом количество операций каждого вида и суммарную трудоёмкость для каждого вида операций. Размер коллекции во время и в конце тестирования не должен отличаться от первоначального. По окончании тестирования программа выводит на экран усреднённую трудоёмкость для каждого вида тестируемых операций и размер коллекции после тестирования.

Полученные экспериментальные оценки должны соответствовать эталонным оценкам трудоёмкости ***Big-O***. Расхождение в оценках может возникнуть из-за ошибок, допущенных при проектировании структуры данных,

алгоритмов и программ соответствующих методов, либо из-за ошибочной методики тестирования коллекции. В этом случае должен быть проведён тщательный анализ и пересмотр решений, принятых при проектировании и разработке класса коллекции или тестирующей программы-драйвера.

1.6. Сопровождение

Разработанный класс для коллекции должен сопровождаться подробной документацией, чтобы его определение и реализацию можно было легко читать и использовать. Документация формируется на всех этапах проектирования коллекции, начиная с разработки АТД и заканчивая процессом отладки. В заключение разработки класса в качестве документации должны быть представлены формат АТД и справочное определение класса для клиентских программ.

По формату АТД коллекции пользователи будут знать тип множества и свойства множества, что делает та или иная операция, правила использования операций. АТД сосредотачивает внимание на предназначении операций, особенностях их вызова.

В дополнение к формату АТД формируется справочное определение класса для клиентской программы, содержащее **только открытые, интерфейсные переменные и методы, соответствующие операциям формата АТД**. Также клиентское определение класса должно быть снабжено комментариями, связывающими класс с форматом АТД (название операций, параметров из АТД).

Методы справочного определения класса задаются только прототипами. Комментарий для каждого метода содержит название соответствующей операции формата АТД. Со смыслом входных параметров и результатов операции, её предусловиями и постусловиями можно познакомиться в формате АТД.

Ниже приведен фрагмент справочного определения классов для АТД «ВЕКТОР» и АТД «ИТЕРАТОР ПРОИЗВОЛЬНОГО ДОСТУПА» (стр. 6 и стр. 14).

//Клиентское определение класса для АТД «ВЕКТОР»

```
template <class T>      // T — тип элементов данных
class vector            //шаблонный класс для объекта вектора
{public:
vector ( )              //конструктор вектора размером 0
vector( int n)          //конструктор вектора размером n
vector (const vector& _Right) //Конструктор копирования
int capacity( ) const;      //Опрос емкости вектора
int size( ) const;          //Опрос размера вектора
void reserve(int n);        //Резервирование емкости вектора
void shrink_to_fit ();      //Уменьшение емкости вектора
T& operator [ ] (int pos);//Оператор индексирования
T& at (int pos);            //Доступ к элементу
void push_back(T Val);      //Добавление в конец
void insert(T val, int pos);//Добавление в указанную позицию
void erase (int pos);       //Удаление из указанной позиции
```

//прототипы других методов класса vector

-
-
-

//класс итератора Iterator

```
class Iterator
{public:
T& opereator *( );          // доступ по чтению/записи
```

```
Iterator& operator ++( ); //оператор префиксного инкремента
Iterator& operator --( ); //оператор префиксного декремента
bool opereator ==(const Iterator &);   // проверка равенства
bool opereator !=(const Iterator &);   // проверка неравенства
```

//прототипы других операторов и операций итератора

-
-
-

}; **//конец класса Iterator**

```
friend class Iterator;
iterator begin( );          //запрос итератора begin( )
iterator end( );            // запрос итератора end( )
};
```
//конец класса vector

2. Практическая работа «Коллекция данных — вектор. Алгоритмы сортировки»

Цели работы: изучение и реализация методов сортировки; экспериментальное исследование эффективности методов сортировки.

2.1. Алгоритмы внутренней сортировки

Одной из наиболее распространенных компьютерных операций является сортировка, т. е. размещение элементов коллекции в определенном порядке. Программист должен уметь выбирать и использовать различные алгоритмы сортировки: от простых для небольших по размеру коллекций до высокоэффективных алгоритмов для коллекций большого объёма.

Алгоритмы сортировки делятся на два типа. При выполнении внутренней сортировки предполагается, что все данные находятся в оперативной памяти компьютера. При внешней сортировке данные могут храниться на вспомогательных запоминающих устройствах, например на жестком диске.

В лабораторной работе исследуются алгоритмы внутренней сортировки коллекции, содержащей массив с неупорядоченными элементами.

Существует множество методов внутренней сортировки, среди которых выделяются три элементарных метода, от которых произошли многие другие, более эффективные. Это **сортировка методом выбора, методом вставок и обменная сортировка** [2, 3, 5, 7]. Поскольку эти простейшие методы сортировки обладают низкой эффективностью и имеют трудоёмкость $\boldsymbol{O(n^2)}$, рекомендуется использовать их для коллекций небольшого размера.

От этих трех методов произошли многие усовершенствованные методы сортировки, достигающие большей эффективности для больших объёмов данных. Производными от сортировки методом выбора являются **сортировка с помощью дерева выбора** [1] и **пирамидальная сортировка** [3, 6, 8]. Эти методы используют выбор минимального элемента массива с помощью вспомогательной структуры — дерева выбора или частично упорядоченного дерева (пирамиды). На рисунке 3 приведены примеры дерева выбора и пирамиды для набора 14 34 1 32 7 56 2 10 5.

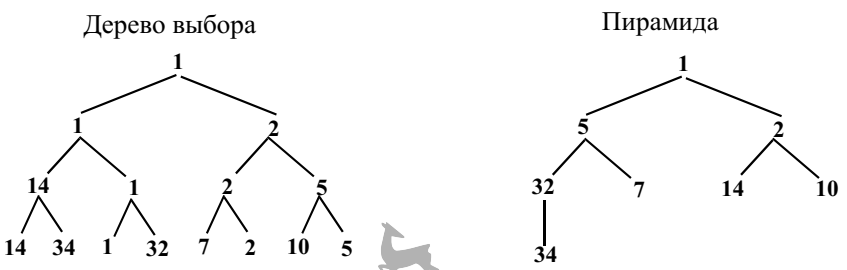


Рис. 3. Структура сортирующих деревьев

Особенностью дерева выбора и пирамиды является то, что в вершине дерева всегда находится минимальный элемент. Алгоритмы обоих методов сортировки предварительно строят соответствующую структуру дерева, а затем используют его для последовательного изъятия из вершины дерева минимальных элементов в отсортированный массив. Благодаря использованию

структуры дерева трудоёмкость алгоритмов сортировки имеет оценку $\boldsymbol{O(n \times log_2 n)}$.

Наиболее известным методом, производным от сортировки методом вставок, является **сортировка Шелла** [2, 5, 7]. Основная идея этого метода заключается в том, что упорядочение ускоряется за счет перемещения элементов при вставках на большие расстояния в массиве. Для этого в массиве выделяются группы элементов, отстоящих друг от друга с первоначально большим шагом $\boldsymbol{h}$. В каждой группе выполняется сортировка методом вставок. Затем шаг $\boldsymbol{h}$ уменьшается, выделяются новые группы элементов, отстоящих друг от друга с уменьшенным шагом, и вновь выполняется сортировка групп методом вставок. Уменьшение шага и выделение новых подгрупп продолжаются, пока шаг не уменьшится до единицы. На последнем этапе выполняется сортировка методом вставок всего массива. Поскольку на предыдущих этапах при сортировке подгрупп массив был значительно упорядочен, на последнем этапе число перестановок в массиве будет минимальным.

На рисунке 4 приведена схема сортировки массива методом Шелла.

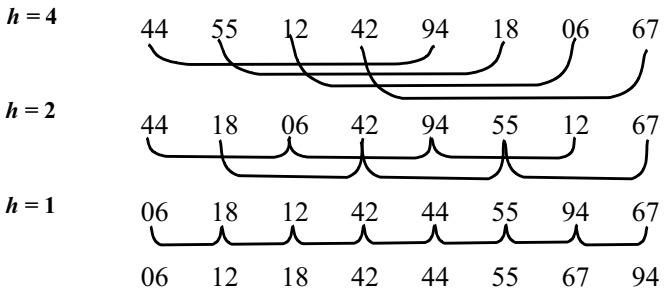


Рис. 4. Схема сортировки массива методом Шелла

Для увеличения эффективности сортировки Шелла предлагаются правила для уменьшения шага h. Широко известна последовательность Кнута [5], использование которой повышает эффективность сортировки Шелла до оценки $\boldsymbol{O(n^{3/2})}$. Она имеет вид 1, 4, 13, 40, 121, 364, 1093, 3280, 9841. Шаги находятся в соотношении $h_k = 3 \times h_{k+1} + 1$.

Есть другие последовательности шагов [8], дающих хорошую эффективность, например: 1, 8, 23, 77, 281, 1073, 4193, 16577, т. е. последовательность $4^{i+1} + 3 \times 2^i + 1$ для $i \geq 0$. Эта последовательность обеспечивает хорошую эффективность для самых трудных случаев сортировки — $O(n^{4/3})$. Приведенные последовательности эффективны в силу того, что шаги взаимно просты.

Сортировка разделением (быстрая сортировка) [1–3, 5–8] является улучшением обменной сортировки и в настоящее время признана самым эффективным методом сортировки для больших объёмов данных. Трудоёмкость быстрой сортировки имеет нотацию $O(n \log_2 n)$. Идея улучшения метода та же, что и в сортировке Шелла, — увеличение расстояния при перемене местами элементов. Благодаря этому элементы быстрее занимают правильные позиции в упорядочиваемом массиве. Метод быстрой сортировки действует по схеме разделения задач на подзадачи. Массив разделяется на две части относительно некоторого опорного элемента. Затем разделение выполняется для каждой из этих частей и т. д. до тех пор, пока размер каждой части не станет равным 1. По окончании этого процесса массив отсортирован. Схема разделения методом быстрой сортировки показана на рисунке 5.

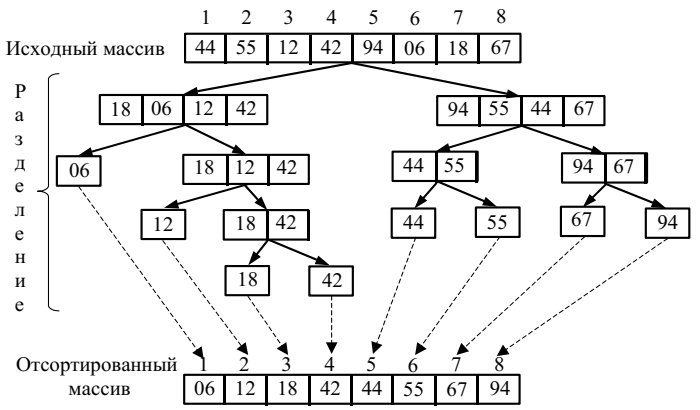


Рис. 5. Схема сортировки методом массива разделения (быстрой сортировки)

Сортировка методом слияния аналогична такому же методу, предназначенному для упорядочивания внешних файлов [2, 3, 8]. Как и быстрая

сортировка, метод действует по схеме нисходящего разделения массива на части. Первоначальный массив разделяется на две равные части, затем выполняется деление частей на две равные части и т. д., пока не будут получены части из одного элемента. Затем выполняется восходящее слияние смежных в массиве частей в упорядоченные части по два элемента, по четыре элемента и т. д., пока при слиянии не будет получена часть, соответствующая по размеру всему массиву. Схема сортировки методом слияния показана на рисунке 6.

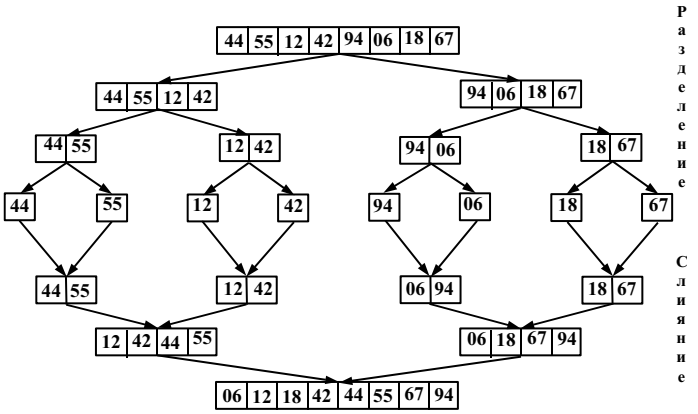


Рис. 6. Схема сортировки методом слияния

Выше была описана сортировка слиянием с нисходящим разбиением массива на части, поэтому она получила наименование «нисходящая сортировка слиянием». Существует альтернативная схема сортировки слиянием — «восходящая сортировка». Она исключает первоначальный этап нисходящего разделения массива на части и считает каждый элемент массива частью, готовой к слиянию с соседней частью, т. е. сортировка сразу начинается с восходящего слияния упорядоченных частей из одного, двух и т. д. элементов.

Независимо от схемы сортировки слиянием оба альтернативных метода имеют оценку трудоёмкости $O(n \log_2 n)$.

Поразрядная (карманная) сортировка [3, 8] основана на поэтапном распределении сортируемых значений с одинаковым значением текущего

разряда по отдельным «карманам». Для этого алгоритм поразрядной сортировки рассматривает значения в массиве как числа, представленные в позиционной системе счисления с основанием *R*, и работает на каждом этапе с отдельным разрядом — цифрой. Различают две схемы выборки текущей цифры из сортируемых значений. **MSD-сортировка** (Most Significant Digit radix sort) анализирует цифры значений слева направо, начиная с наиболее значащих цифр. **LSD-сортировка** (Last Significant Digit radix sort) анализирует цифры в значениях справа налево, начиная с младших цифр. Сортировка заканчивается, когда будет выполнено распределение значений по последней цифре.

Ход MSD-сортировки аналогичен быстрой сортировке с разделением массива на части и дальнейшим рекурсивным разделением частей. На первом этапе значения в массиве распределяются по «карманам» в соответствии со значением старшей цифры. В свою очередь, к «карманам» рекурсивно применяется распределение по следующей цифре и т. д., пока не будет выполнено распределение по младшей цифре. Пример MSD-сортировки приведен на рисунке 7, здесь не показаны пустые «карманы», в которые не попало ни одно значение. Общее количество «карманов» равно количеству возможных значений цифры, т. е. основанию системы счисления *R*.

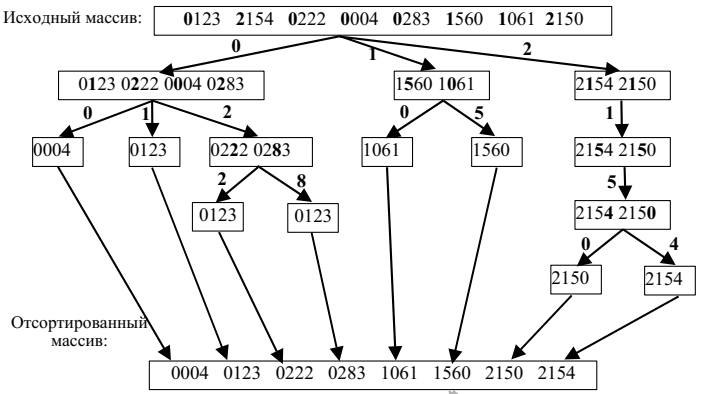


Рис. 7. Схема поразрядной десятичной MSD-сортировки

Трудоёмкость MSD-сортировки оценивается величиной $O(n \log_R n)$. При $R = 2$ трудоёмкость сопоставима с трудоёмкостью быстрой сортировки или

сортировки слиянием и равна $O(n \log_2 n)$. При больших R величина $\log_R n$ становится малой, и трудоёмкость становится линейной функцией $O(n)$.

LSD-сортировка выбирает цифры из сортируемых значений в направлении справа налево. В отличие от MSD-сортировки алгоритм LSD-сортировки не рекурсивный. На каждом этапе значения распределяются по «карманам» в соответствии со значением очередного разряда-цифры. Сложившееся распределение по «карманам» формирует новое содержимое сортируемого массива. На следующем этапе массив вновь распределяется по «карманам» по значению следующего разряда-цифры и т. д. Сортировка заканчивается, когда будет выполнено распределение по старшей цифре. Схема LSD-сортировки представлена на рисунке 8.

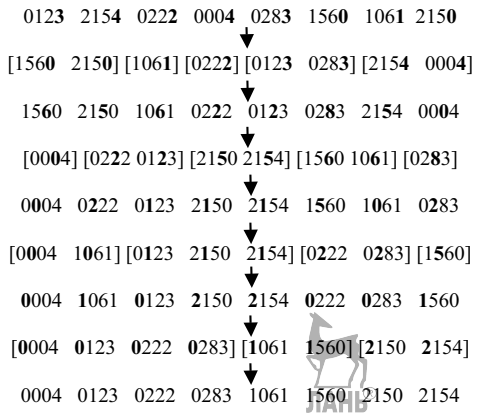


Рис. 8. Схема поразрядной десятичной LSD-сортировки

Трудоёмкость LSD-сортировки имеет нотацию $O(n \times w)$, где w — число разрядов в сортируемых значениях, т. е. трудоёмкость LSD-сортировки линейно зависит от объёма сортируемых данных.

2.2. Задание к практической работе

1. Спроектировать, реализовать и провести тестовые испытания АТД «Вектор» для коллекции, содержащей данные произвольного типа. Размер коллекции и тип данных задаются клиентской программой.

АТД «Вектор» представляет собой последовательность элементов, размещённых в одной сплошной области памяти. Доступ к элементам вектора осуществляется по индексу.

Интерфейс АТД «Вектор» включает следующие операции:

- опрос размера вектора;
- изменение размера вектора;
- формирование в векторе случайных значений;
- формирование в векторе упорядоченных значений;
- чтение/запись по индексу;
- элементарная сортировка (по варианту задания);
- эффективная сортировка (по варианту задания);
- запрос итератора произвольного доступа *begin()*;
- запрос «неустановленного» итератора произвольного доступа *end()*;
- запрос «неустановленного» итератора произвольного доступа *rend()*.

Для тестирования эффективности алгоритмов сортировки интерфейс АТД «Вектор» включает следующие дополнительные операции:

– опрос числа выполненных сравнений,

– опрос числа выполненных обменов.

Итератор произвольного доступа включает операции:

– операция доступа по чтению/записи к значению текущего элемента *;

– операция инкремента для перехода к следующему индексу в векторе ++;

– операция инкремента для перехода к предыдущему индексу в векторе --;

– операция установки итератора на N индексов вперед — X + N;

– операция установки итератора на N индексов назад — X – N;

– операция проверки равенства однотипных итераторов ==.

2. Выполнить отладку и тестирование отдельных операций АТД «Вектор» с помощью меню операций.

3. Выполнить сравнительное тестирование трудоёмкости алгоритмов сортировки для худшего и среднего случаев.

4. Провести анализ экспериментальных показателей трудоёмкости алгоритмов сортировки.

5. Составить отчет по лабораторной работе. Отчет должен содержать следующее:

1) титульный лист;

2) цель лабораторной работы;

3) общее задание (пп. 1–4) и вариант задания;

4) формат АТД вектора;

5) определение шаблонного класса для коллекции «Вектор», предназначенное для клиентской программы;

6) описание методики тестирования трудоёмкости алгоритмов сортировки;

7) таблицы и графики с полученными оценками трудоёмкости алгоритмов операций для худшего и среднего случаев функционирования АТД. Должны быть приведены следующие графики:

– число обменов, число сравнений для худшего и среднего случаев для элементарного алгоритма сортировки (графики совмещены в одной системе координат),

– число обменов, число сравнений для худшего и среднего случаев для эффективного алгоритма сортировки (графики совмещены),

– число сравнений алгоритмов элементарной и эффективной сортировки для среднего случая,

– число обменов алгоритмов элементарной и эффективной сортировки для среднего случая,

– сумма числа сравнений и обменов алгоритмов элементарной и эффективной сортировки для среднего случая;

8) сравнительный анализ теоретических и экспериментальных оценок эффективности алгоритмов АТД;

9) сравнительный анализ экспериментальных оценок трудоёмкости для различных алгоритмов сортировки;

10) выводы;

11) список использованной литературы;

12) приложение с текстами программ:

- полное определение класса и текстов методов класса,
- текст программ тестирования операций АТД.

2.2.1. Варианты заданий

1. Алгоритм сортировки выбором, алгоритм сортировки с помощью дерева выбора.

2. Алгоритм сортировки выбором, алгоритм пирамидальной сортировки.

3. Алгоритм сортировки вставками, алгоритм сортировки Шелла.

4. Алгоритм обменной сортировки, рекурсивный алгоритм сортировки разделением.

5. Алгоритм обменной сортировки, итерационный алгоритм сортировки разделением.

6. Алгоритм нисходящей сортировки слиянием, алгоритм восходящей сортировки слиянием.

7. Алгоритм поразрядной MSD-сортировки.

8. Алгоритм поразрядной LSD-сортировки.

2.2.2. Методические указания к выполнению задания

1. Создание коллекции «Вектор» выполняется в соответствии с технологией реализации коллекций, изложенной в разделе 1. Для АТД «Вектор» разрабатываются формат АТД и шаблонный класс-контейнер.

2. Для тестирования разработанного класса-контейнера разрабатываются две программы: программа тестирования отдельных операций через меню и программа тестирования эффективности алгоритмов сортировки.

3. При реализации сортировок рекомендуется использовать алгоритмы, приведенные в приложении Б.

4. Тестирование отдельных операций через меню выполняется на коллекциях небольшого размера (до 20 элементов). Размер коллекции и тип данных, хранящихся в ней, задаются с клавиатуры перед началом тестирования. До и после выполнения операций сортировки необходимо вывести на экран содержимое коллекции.

5. Перед тестированием трудоёмкости сортировок задаются тип и размер коллекции, вид набора данных (упорядоченный или случайный). Размер коллекции может задаваться в пределах от 10 до 100 000 элементов. Полученная трудоёмкость (количество операций сравнения и операций обменов, выполненных в процессе сортировки) выводится на экран. При тестировании алгоритмов сортировки исследуются варианты худшего и среднего случаев работы алгоритмов. Сравнительное тестирование эффективности алгоритмов проводится на базе сортировки одного и того же набора данных.

2.3. Контрольные вопросы и упражнения

1. Что понимается под средним и худшим случаем работы алгоритма? Приведите примеры.

2. Дайте краткую характеристику алгоритмов сортировки, имеющих трудоёмкость $O(n^2)$.

3. Дайте краткую характеристику алгоритмов сортировки с трудоёмкостью $O(n \log_2 n)$.

4. Какие методы выбора опорного элемента используются в алгоритмах быстрой сортировки?

5. Постройте дерево выбора, размещенное в массиве, для значений 5 23 2 6 8 2 78 1 9 12.

6. Постройте пирамиду, размещенную в массиве, для значений 5 23 2 6 8 2 78 1 9 12.

7. Приведите схему сортировки Шелла для значений 5 23 2 6 8 2 78 1 9 12.

8. Приведите схему пирамидальной сортировки для значений 5 23 2 6 8 2 78 1 9 12.

9. Приведите схему быстрой сортировки для значений 5 23 2 6 8 2 78 1 9 12.

10. Приведите схему нисходящей сортировки слиянием для значений 5 23 2 6 8 2 78 1 9 12.

11. Приведите схему восходящей сортировки слиянием для значений 5 23 2 6 8 2 78 1 9 12.

12. Приведите схему десятичной MSD-сортировки для значений 3285 0023 1459 6346 8453 2000 1578 0341 0009 1245.

13. Приведите схему десятичной LSD-сортировки для значений 3285 0023 1459 6346 8453 2000 1578 0341 0009 1245.

3. Практическая работа «Коллекция данных — список»

Цель работы: освоение технологии реализации позиционных, линейных коллекций на примере АТД «Список».

Список представляет позиционно-ориентированную, линейную последовательность с доступом к элементам по номеру позиции или по значению. Элемент списка хранит значение объекта, включённого в список. Номера элементов в списке задаются в линейном порядке, начиная со значения 0 или 1. У списка есть первый и последний элементы. У всех остальных элементов есть предшествующий элемент (предшественник) и последующий элемент (преемник). Список имеет начало (голову), фиксирующую первый элемент, и конец (хвост), фиксирующий последний элемент.

Над списком и его элементами можно выполнять вставку, удаление, изменение значений элементов в произвольных позициях, поиск заданных значений, последовательный обход элементов списка.

Список имеет вспомогательный объект — прямой итератор для последовательного перемещения и доступа к данным, хранящимся в списке. Если структура списка позволяет перемещаться по элементам в обратном порядке, то список имеет также обратный итератор.

Список вместе с операциями над элементами и итераторами образует абстрактный тип данных.

3.1. Структуры списков

Для хранения списка используются либо массив элементов, либо связная структура элементов. Список на базе массива представляет собой последовательность значений, занимающую первые элементы массива (рис. 9).

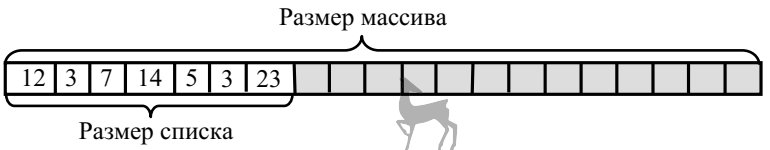


Рис. 9. Структура списка на базе массива

К достоинствам хранения списка в массиве можно отнести прямой доступ к элементам списка по номеру (индексу) для записи и чтения значений. Но при вставке и удалении данных приходится выполнять сдвиг последующих элементов в списке на одну позицию вправо или влево. Поэтому трудоёмкость операций вставки и удаления элементов имеет оценку ***O(n)***, остальные операции благодаря прямому доступу — оценку ***O(1).***

Длина последовательности и, следовательно, списка ограничена сверху размером массива. Чтобы снять это ограничение, можно организовать хранение списка на базе динамического массива, размер которого можно менять в зависимости от его заполнения. Возможны разные правила увеличения размера динамического массива по мере роста списка при включении новых данных. На рисунке 10 показаны перестройки массива при линейном и экспоненциальном росте размера массива (ёмкости списка).

Обе формы обеспечивают только последовательный доступ к элементам, вследствие чего все операции для элементов списка имеют трудоёмкость ***O(n)***.

Связные структуры на базе адресных указателей представляют собой множество элементов, ссылающихся друг на друга с помощью адресных указателей. Каждый элемент структуры размещён в динамической памяти и называется узлом списка. Узел структуры состоит их двух частей — значения и

указателей на соседние узлы структуры. Узел односвязной структуры содержит один указатель на следующий узел (на преемника в списке). Ввиду этого в односвязном списке возможен проход только в одном направлении — от начала списка к его концу. Узел двусвязной структуры содержит указатели на предыдущий и следующий узлы (на предшественника и преемника в списке), и в двусвязном списке возможен проход в обоих направлениях, от начала списка к концу и от конца к началу. Для односвязного списка вводится дополнительный указатель на первый элемент списка — голова списка (рис. 11). В двусвязном списке голова списка состоит из двух указателей на первый и последний элементы (рис. 12).

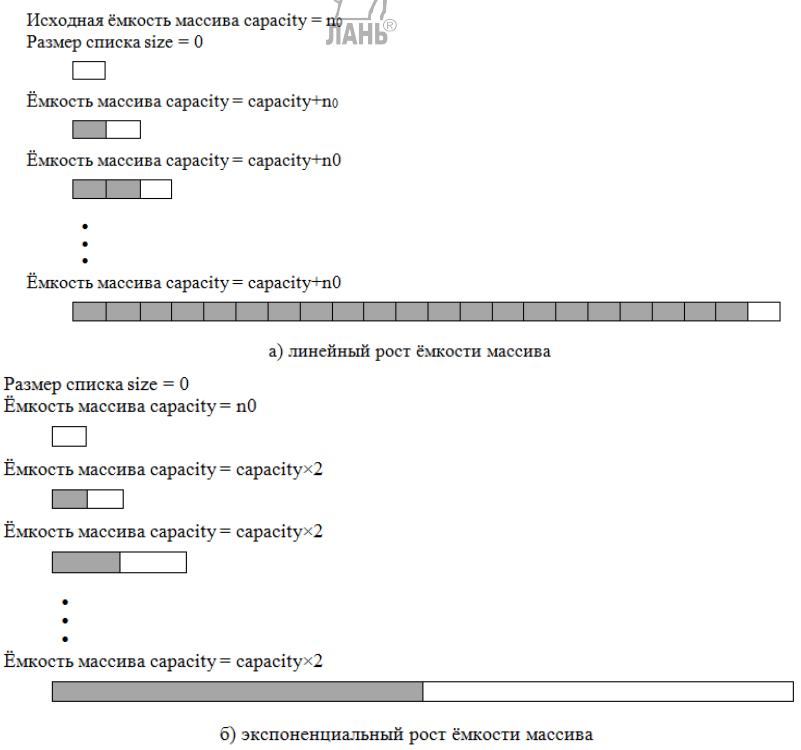


Рис. 10. Рост ёмкости списка

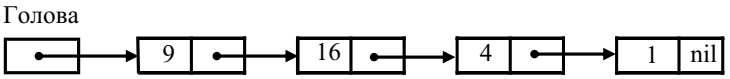


Рис. 11. Односвязная структура списка на базе адресных указателей

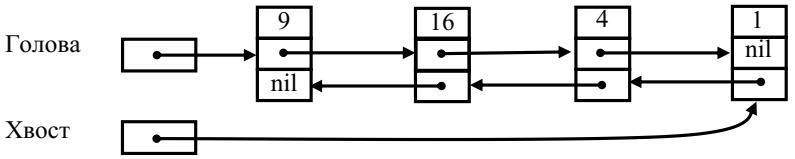


Рис. 12. Двусвязная структура списка на базе адресных указателей

Существуют разновидности связных списков, ориентированные на специфический режим использования. Например, система разделения времени хранит задания в очереди, построенной на базе связного списка. Система просматривает список от начала до конца, выделяя при этом каждому заданию ресурсы компьютера на конечный квант времени. При достижении конца очереди система возвращается к первому заданию в очереди и начинает новый цикл обслуживания. Для таких систем необходим логически замкнутый в кольцо список элементов, который можно построить на базе кольцевой связной структуры (рис. 13).

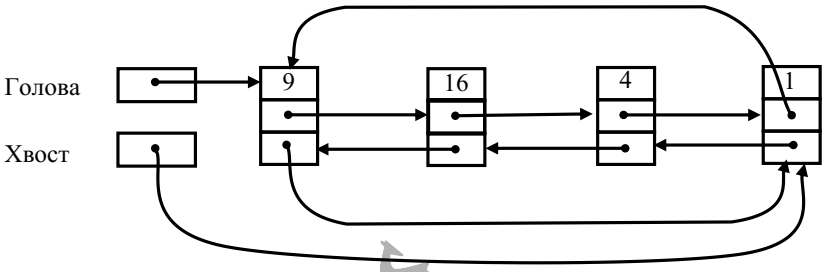


Рис. 13. Кольцевая двусвязная структура списка

При программировании операций для связных списков приходится обрабатывать особые ситуации при вставке или удалении элементов в начале или конце списка. Чтобы избавиться от этих ситуаций и тем самым ускорить выполнение операций вставки и удаления, используется кольцевая структура с фиктивным (барьерным) головным узлом [2]. Он существует в структуре всегда, даже если список пуст. Более того, фиктивный узел играет роль головы списка. Он хранит указатели на первый и последний узлы в структуре, когда список не пуст, и указатели на самого себя, когда список

пуст. Последний узел в непустом списке указывает на барьерный узел (рис. 14). Таким образом, структура всегда замкнута в кольцо. Благодаря этому в любой момент времени производится вставка или удаление какого-либо узла по одной и той же схеме.

У списков, использующих связную структуру на базе адресных указателей, есть один существенный недостаток — интенсивное использование механизма динамической памяти при вставке и удалении элементов. Если при работе со списком операции вставки и удаления выполняются часто, то это существенно замедляет работу программы, использующей такой список. В случае, когда заранее известен предельный размер списка, можно использовать связную структуру на базе массива с индексными указателями, свободную от этого недостатка [6].

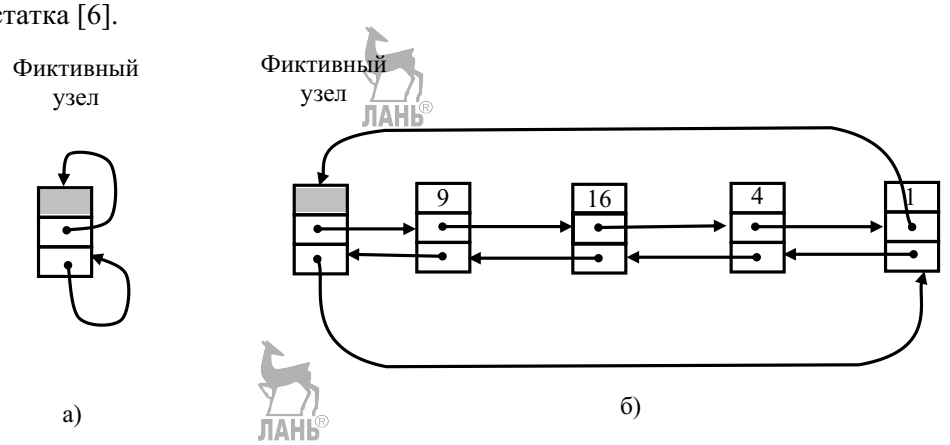


Рис. 14. Кольцевая двусвязная структура с фиктивным элементом:

а — пустой список; *б* — непустой список.

Связная структура представляет собой массив записей, поля которых аналогичны полям узла структуры с адресными указателями. Либо можно использовать несколько массивов для каждого из полей. В структуре адресные указатели заменяются индексами следующего и предыдущего элементов. На рисунке 15 представлена связная структура на базе индексных указателей, эквивалентная двусвязному списку.

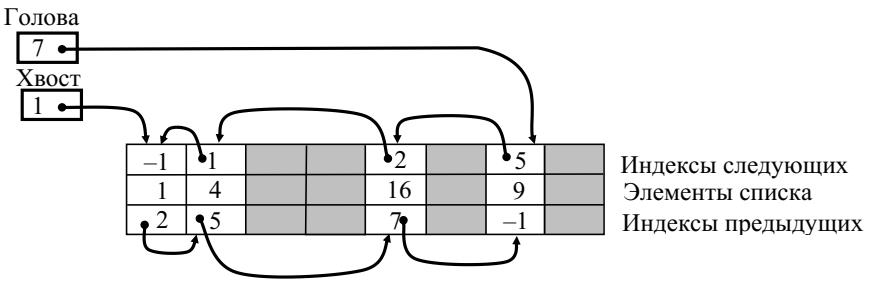


Рис. 15. Связная структура на базе индексных указателей

Голова списка представляет собой переменную, содержащую индекс позиции массива, в которую помещён первый элемент списка. По ходу работы со списком в результате вставок и удалений элементов в массиве образуются перемежающиеся свободные и занятые позиции. При добавлении нового элемента в список для него нужно выделять свободную позицию в массиве. Свободные позиции связываются в дополнительный односвязный список, используя поле индекса следующего элемента. Голова списка свободных позиций размещается в дополнительной индексной переменной. Список свободных позиций находится вперемешку со списком занятых позиций (рис. 16).

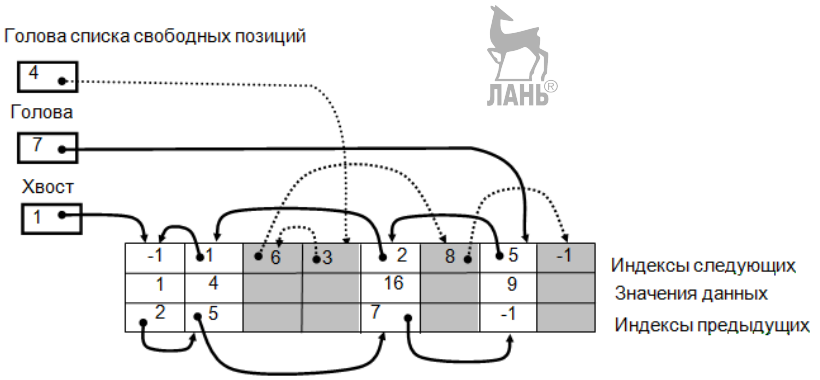


Рис. 16. Список свободных позиций в массиве с индексными указателями

Список свободных позиций используется как стек. При вставке элемента в список для него выбирается первая позиция из списка свободных позиций при удалении элемента списка освобождающаяся позиция помещается в начало списка свободных позиций.

При создании списка все позиции в массиве свободны и связаны в список свободных позиций.

3.2. Задание к практической работе

1. Спроектировать, реализовать и провести тестовые испытания АТД «Список» для коллекции, содержащей данные произвольного типа. Тип данных задаётся клиентской программой.

АТД «Список» представляет позиционно-ориентированную, линейную последовательность с доступом к элементам по номеру позиции или по значению.

Интерфейс АТД «Список» включает следующие операции:

- конструктор;
- конструктор копирования;
- деструктор;
- опрос размера списка;
- очистка списка;
- проверка списка на пустоту;
- опрос наличия заданного значения;
- чтение значения с заданным номером в списке;
- изменение значения с заданным номером в списке;
- получение позиции в списке для заданного значения;
- включение нового значения;
- включение нового значения в позицию с заданным номером;
- удаление заданного значения из списка;
- удаление значения из позиции с заданным номером;
- запрос прямого итератора *begin()*;
- запрос обратного итератора *rbegin()* //(для вариантов задания 1, 2, 5, 7);
- запрос «неустановленного» прямого итератора *end()*;
- запрос «неустановленного» обратного итератора *rend()* //(для вариантов задания 1, 2, 5, 7);

- прямой и обратный (для вариантов задания 1, 2, 5, 7) итераторы для доступа к значениям в списке с основными операциями (набор операций зависит от вида структуры для списка):

– операция доступа по чтению и записи к текущему значению *;

– операция инкремента для перехода к следующему (к предыдущему для обратного итератора) значению в списке ++;

– операция декремента для перехода к предыдущему (к следующему для обратного итератора) значению в списке --;

– проверка равенства однотипных итераторов ==;

– проверка неравенства однотипных итераторов !=.

Для отладки и тестирования операций интерфейс АТД «Список» включает дополнительные операции:

- запрос числа элементов списка, просмотренных операциями опроса наличия заданного значения, включения нового значения в позицию с заданным номером, удаления значения из позиции с заданным номером;
- вывод на экран последовательности значений данных из списка.

2. Выполнить отладку и тестирование всех операций АТД «Список» и итераторов с помощью меню операций.

3. Составить отчёт по лабораторной работе. Отчёт должен содержать следующие пункты:

1) титульный лист;

2) цель лабораторной работы;

3) общее задание и вариант задания;

4) формат АТД «Список»;

5) формат АТД «Прямой итератор списка»;

6) формат АТД «Обратный итератор списка» (для вариантов задания 1, 2, 5, 7);

7) определение шаблонного класса для коллекции «Список», предназначенное для клиентской программы;

8) выводы;

9) список использованной литературы;

10) приложение с текстами программ:

- полное определение класса и текстов методов класса;
- текст программы-меню тестирования отдельных операций АТД.

3.2.1. Варианты заданий

1. Структура данных — динамический массив с линейным законом изменения ёмкости.

2. Структура данных — динамический массив с экспоненциальным законом изменения ёмкости.

3. Структура данных — односвязная, на базе массива с индексными указателями.

4. Структура данных — двусвязная, на базе массива с индексными указателями.

5. Структура данных — двусвязная, на базе адресных указателей.

6. Структура данных — кольцевая, односвязная, на базе адресных указателей.

7. Структура данных — кольцевая, двусвязная, на базе адресных указателей.

8. Структура данных — кольцевая, односвязная, на базе адресных указателей, с использованием фиктивного элемента.

3.2.2. Методические указания к выполнению задания

1. Создание коллекции «Список» выполняется в соответствии с технологией реализации коллекций, изложенной в разделе 1. Для АТД «Список» разрабатываются формат АТД и шаблонный класс-контейнер. Параметр шаблона задаёт тип данных, хранящихся в списке. Для узлов связных структур списков и итераторов разрабатываются вспомогательные классы. Определения вспомогательных классов размещаются внутри определения класса списка.

2. Операции над списком учитывают специфику структуры, хранящей список. Операции для двусвязных списков выбирают проход по списку в

прямом или обратном направлениях с целью минимизации числа просматриваемых узлов.

3. Для тестирования операций разработанного класса-контейнера разрабатывается программа-меню. В меню следует ввести дополнительную операцию опроса числа просмотренных элементов списка в операциях поиска по значению, вставки и удаления по номеру, операцию вывода содержимого списка на экран. При выводе списка необходимо отражать только значения данных, хранящиеся в списке.

4. Программа-меню выполняет для каждой операции ввод исходных данных с клавиатуры, вызов соответствующего метода коллекции с исходными данными, вывод на экран **кода ответа** метода после возврата из метода.

3.3. Контрольные вопросы и упражнения

1. Перечислите достоинства и недостатки списка на базе массива.

2. Перечислите достоинства и недостатки списка на базе адресных указателей.

3. Перечислите достоинства и недостатки списка на базе массива с индексными указателями.

4. Приведите ожидаемые оценки трудоёмкости операций для списка на базе массива и надежного массива.

5. Приведите ожидаемые оценки трудоёмкости операций для списка на базе связной структуры.

6. Приведите схему структуры списка на базе массива с индексными указателями после серии операций: вставка (3), вставка (1), вставка (23), удаление (1), вставка (15), удаление (3), вставка (1), вставка (4), удаление (23). Размер массива равен 8, список первоначально пуст.

7. Приведите фрагмент программ поиска и вставки в список, базирующийся на односвязном списке.

8. Приведите фрагмент программ поиска и удаления в список, базирующийся на кольцевом, двусвязном списке с фиктивным элементом.

9. Напишите клиентскую программу, размещающую элементы в коллекции «Список» в обратном порядке. Какой вид структуры целесообразно использовать для хранения списка?

4. Практическая работа «Коллекция данных — дерево поиска»

Цель работы: освоение технологии реализации ассоциативных коллекций на примере АТД «Двоичное дерево поиска».

Двоичное дерево поиска (**B**inary **S**earch **T**ree — **BST**) представляет упорядоченное, иерархическое, ассоциативное множество элементов, между которыми существуют структурные отношения «предки — потомки». Каждый элемент ассоциативного множества состоит из данных и уникального ключевого значения, идентифицирующего данные среди прочих в множестве. Положение элемента в дереве определяется ключевым значением данных при сопоставлении его с другими ключами, присутствующими в дереве [1–4, 5–9].

Каждый элемент, называемый узлом **BST**-дерева, имеет потомков, разбитых на два подмножества — левое и правое поддеревья. Непосредственные потомки элемента называются левым и правым сыновьями. Каждый узел BST-дерева удовлетворяет следующим условиям (рис. 17):

- ключ элемента ***t*** больше всех ключей, содержащихся в его левом поддереве $\boldsymbol{R_t}$;
- ключ элемента ***t*** меньше всех ключей, содержащихся в его правом поддереве $\boldsymbol{R_t}$;
- деревья $\boldsymbol{L_t}$ и $\boldsymbol{R_t}$ являются бинарными деревьями поиска.

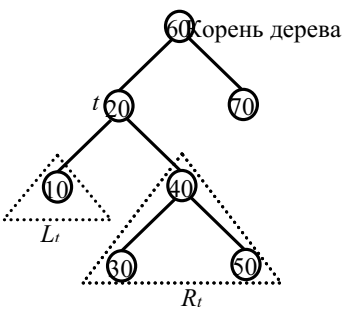


Рис. 17. Схема бинарного дерева поиска

Такая организация данных позволяет использовать **BST**-дерево для эффективного двоичного доступа по ключам к элементам множества.

Как абстрактный тип данных **BST**-дерево предусматривает операции поиска, вставки и удаления элементов по ключу. Используя эти операции, можно построить любое бинарное дерево. Операции вставки, удаления и поиска элементов для **BST**-дерева используют правило двоичного поиска при доступе к элементу с заданным значением ключа (см. приложение В). Поэтому трудоёмкость этих операций соответствует трудоёмкости бинарного поиска в упорядоченном множестве и имеет нотацию $O(\log_2 n)$.

Необходимо отметить структурную зависимость **BST**-дерева от порядка поступления и удаления элементов. Например, при последовательных вставках элементов со строго возрастающими ключами структура **BST**-дерева выродится в линейный список правых сыновей. В этом случае трудоёмкость операций возрастёт до нотации $O(n)$.

Важной операцией для **BST**-дерева является обход его элементов в определенном порядке для выполнения какой-либо операции над элементами дерева. Для **BST**-дерева существуют три основные схемы обхода — прямая, симметричная и обратная. Прямой обход выполняется по схеме $t \rightarrow L_t \rightarrow R_t$, то есть обход выполняется в порядке: посетить узел *t,* обойти узлы левого поддерева по схеме прямого обхода**,** обойти узлы правого поддерева по схеме прямого обхода. Аналогично, симметричный обход выполняется по схеме $L_t \rightarrow$ $\rightarrow t \rightarrow R_t$**,** а обратный обход — по схеме $L_t \rightarrow R_t \rightarrow t$. Существует также обход элементов дерева по уровням (см. приложение В). При обходе дерева по любой схеме каждый узел дерева посещается только один раз, и трудоёмкость операций обхода имеет нотацию $O(n)$.

Ещё одно важное свойство **BST**-дерева — рекурсивная природа его структуры. В самом деле, дерево можно определить как узел-корень и два его поддерева. В свою очередь, каждое поддерево можно определить точно так же. Эта особенность структуры определяет рекурсивный способ программирования операций **BST**-дерева. Рекурсивные алгоритмы операций компактны и

наглядны. При выполнении операции прямой ход рекурсии соответствует спуску в **BST**-дереве от корня к узлу с заданным значением ключа. Обратный ход рекурсии обеспечивает возврат по этому же пути от заданного узла к корню, что бывает важным для некоторых операций.

Но для вырожденных, больших деревьев глубина рекурсии может быть очень большой, что приводит к переполнению области памяти, отводимой под системный стек, обслуживающий вызовы функций. Следствием этого является отказ в работе программ коллекции и клиентской программы. Поэтому, если вероятность вырождения большого дерева высока, используется итеративный алгоритм, выполняющий спуск по дереву с помощью цикла. Если алгоритм операции после спуска к заданному узлу ведёт дополнительную обработку ранее пройденных узлов, то адреса этих узлов сохраняются в собственном стеке алгоритма. Нагрузка на такой стек значительно меньше, и вероятность отказа коллекции снижается.

4.1. Структуры BST-деревьев

Для хранения **BST**-деревьев используются две альтернативные структуры — массив элементов дерева и связная структура дерева на базе адресных указателей.

Массив элементов используется в случае, если задано предельное количество элементов (размер дерева), которые будут размещены в **BST**-дереве. В этом случае для хранения дерева выделяется массив заданного размера, тип которого совпадает с типом элементов множества, хранящегося в дереве. Корень дерева размещается в элементе массива с индексом 0, а его левый и правый сыновья — с индексом 1 и 2 соответственно. По индексу любого элемента легко вычислить индексы его сыновей и родителя в массиве. Если некоторый элемент имеет индекс ***i***, то его левый сын расположен по индексу ***2i + 1***, а правый сын — по индексу ***2i + 2***. Индекс родителя вычисляется как целая часть от деления ***(i – 1)/2*** (рис. 18).

Как видно из примера, приведенного на рис. 18, хранение дерева приводит к неэффективному использованию памяти массива. Поэтому использовать массив рекомендуется для хранения полных деревьев, в которых каждый узел имеет двух сыновей.

Наиболее распространённой формой для хранения деревьев является связная структура на базе адресных указателей. Каждый узел дерева размещается в динамической памяти и содержит помимо ключа и данных два указателя на левого и правого сыновей. Таким образом, структура через указатели на сыновей обеспечивает спуск по дереву от корня к местоположению узла с заданным значением ключа. Для некоторых операций требуется подъём от некоторого узла к родительскому узлу и далее к корню. Для этого в структуру узла можно ввести дополнительный указатель на родителя. Но это приводит к неоправданным затратам памяти на дополнительные указатели в узлах. Поэтому эти указатели используются только в отдельных разновидностях деревьев. Как правило, естественное поведение рекурсивного алгоритма операции обеспечивает возврат по ранее пройденному пути в дереве.

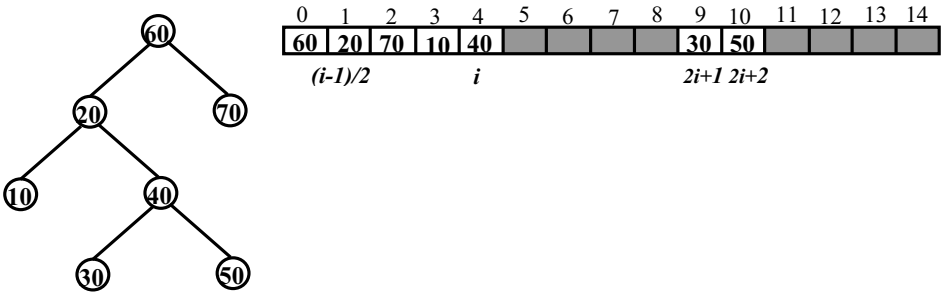


Рис. 18. Схема размещения **BST**-дерева в массиве

В дерево вводится вспомогательный указатель входа, содержащий адрес корневого узла. Как правило, работа операций начинается с этого узла. Если дерево пусто, то указатель на корень содержит значение ***nil*** (рис. 19).

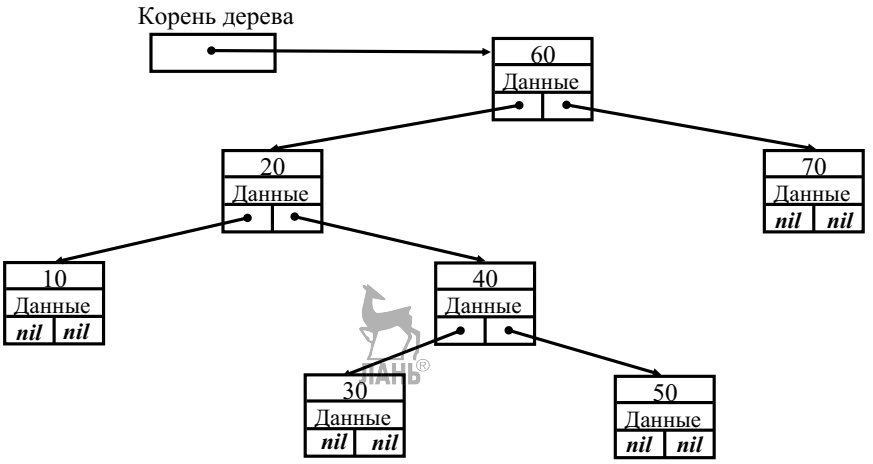


Рис. 19. Связная структура **BST**-дерева

4.2. Задание к практической работе

1. Спроектировать, реализовать и провести тестовые испытания АТД «**BST**-дерево» (Binary Search Tree — BST) для коллекции, содержащей ключи и данные произвольного типа. Тип ключа и тип данных задаются параметрами шаблона класса **BST**-дерева.

В формате АТД **BST**-дерево представляется как упорядоченное, иерархическое, ассоциативное множество элементов, между которыми существуют структурные отношения «предки — левые потомки — правые потомки».

Интерфейс АТД «**BST**-дерево» включает следующие операции:

- конструктор,
- конструктор копирования,
- деструктор,
- опрос размера дерева,
- очистка дерева,
- проверка дерева на пустоту,
- доступ по чтению/записи к данным по ключу,

- включение данных с заданным ключом,
- удаление данных с заданным ключом,
- формирование списка ключей в дереве в порядке обхода узлов по схеме, заданной в варианте задания,
- дополнительная операция, заданная в варианте задания,
- запрос прямого итератора, установленного на узел дерева с минимальным ключом *begin(),*
- запрос обратного итератора, установленного на узел дерева с максимальным ключом *rbegin()*,
- запрос «неустановленного» прямого итератора *end(),*
- запрос «неустановленного» обратного итератора *rend().*

Операции прямого и обратного итераторов выполняются по схеме симметричного обхода элементов дерева ***L — t — R***:

- операция доступа по чтению и записи к данным текущего узла *,
- операция перехода к следующему (для обратного — к предыдущему) по ключу узлу в дереве ++,
- операция перехода к предыдущему (для обратного — к следующему) по ключу узлу в дереве --,
- проверка равенства однотипных итераторов ==,
- проверка неравенства однотипных итераторов !=.

Для тестирования коллекции интерфейс АТД «**BST**-дерево» включает дополнительные операции:

- вывод структуры дерева на экран,
- опрос числа узлов дерева, просмотренных операцией.

2. Выполнить отладку и тестирование всех операций АТД «**BST**-дерево» с помощью программы-меню.

3. С помощью программы тестирования трудоёмкости (приложение Г) исследовать зависимость от размера дерева средней трудоёмкости операций

поиска, вставки и удаления. Тестирование провести для двух случаев: для случайного **BST**-дерева и вырожденного **BST**-дерева.

4. Провести сравнительный анализ теоретических и экспериментальных показателей трудоёмкости операций, трудоёмкости операций между собой.

5. Составить отчёт по лабораторной работе. Отчёт должен содержать следующие пункты:

1) титульный лист,

2) цель лабораторной работы,

3) общее задание и вариант задания,

4) формат АТД «**BST**-дерево»,

5) формат АТД «Прямой итератор **BST**-дерева»,

6) формат АТД «Обратный итератор **BST**-дерева»,

7) справочное определение класса для коллекции «**BST**-дерево», предназначенное для клиентской программы,

8) описание методики тестирования трудоёмкости операций **BST**-дерева,

9) таблицы и графики с полученными оценками средней трудоёмкости операций для случайного и вырожденного **BST**-дерева. Должны быть приведены графики среднего числа пройденных узлов для операций поиска, вставки и удаления (графики совмещены в одной системе координат),

10) сравнительный анализ теоретических и экспериментальных оценок трудоёмкости для операций **BST**-дерева,

11) сравнительный анализ экспериментальных оценок трудоёмкости операций поиска, вставки и удаления между собой,

12) выводы,

13) список использованной литературы,

14) приложение с текстами программ:

- текст полного определения класса и методов класса **BST**-дерева,
- текст программы-меню для тестирования отдельных операций **BST**-дерева,

- текст программы для тестирования средней трудоёмкости операций **BST**-дерева.

4.2.1. Варианты задания

1. Алгоритмы операций поиска, вставки и удаления реализуются в рекурсивной форме.

2. Вывод на экран последовательности ключей при обходе узлов дерева по схеме $t \rightarrow L_t \rightarrow R_t$.

3. Дополнительная операция: определение высоты дерева. Трудоёмкость операции — ***O(n)***.

4. Алгоритмы операций поиска, вставки и удаления реализуются в итерационной форме.

5. Вывод на экран последовательности ключей при обходе узлов дерева по схеме $L_t \rightarrow t \rightarrow R_t$.

6. Дополнительная операция: определение длины внешнего пути дерева. Трудоёмкость операции — ***O(n).***

7. Алгоритмы операций поиска, вставки и удаления реализуются в рекурсивной форме.

8. Вывод на экран последовательности ключей при обходе узлов дерева по схеме $L_t \rightarrow t \rightarrow R_t$.

9. Дополнительная операция: поиск и подъем в корень дерева узла с ближайшим ключом, большим заданного значения. Трудоёмкость операции — ***O(log n).***

10. Алгоритмы операций поиска, вставки и удаления реализуются в итерационной форме.

11. Вывод на экран последовательности ключей при обходе узлов дерева по схеме $t \rightarrow L_t \rightarrow R_t$.

12. Дополнительная операция: определение в дереве количества узлов с ключами, большими заданного значения. Трудоёмкость операции — ***O(log n).***

13. Алгоритмы операций поиска, вставки и удаления реализуются в рекурсивной форме.

14. Вывод на экран последовательности ключей при обходе узлов дерева по схеме $\boldsymbol{L_t \rightarrow R_t \rightarrow t}$.

15. Дополнительная операция: определение критерия сбалансированности bal_t для каждого узла дерева — t ($bal_t = h_r - h_l$, где h_r и h_l — высота правого и левого поддерева узла t). Трудоёмкость операции — ***O(n).***

16. Алгоритмы операций поиска, вставки и удаления реализуются в итерационной форме.

17. Вывод на экран последовательности ключей при обходе узлов дерева по схеме $\boldsymbol{L_t \rightarrow t \rightarrow R_t}$.

18. Дополнительная операция: вывод на экран горизонтального изображения дерева (размер ≤ 8) с симметричным позиционированием узлов относительно родителей. Трудоёмкость операции — ***O(n).***

19. Алгоритмы операций поиска, вставки и удаления реализуются в рекурсивной форме.

20. Вывод на экран последовательности ключей при обходе узлов дерева по схеме $\boldsymbol{t \rightarrow L_t \rightarrow R_t}$.

21. Дополнительная операция: определение порядкового номера для элемента с заданным ключом. Трудоёмкость операции — ***O(log n).***

22. Алгоритмы операций поиска, вставки и удаления реализуются в итерационной форме.

23. Вывод на экран последовательности ключей при обходе узлов дерева по схеме $\boldsymbol{L_t \rightarrow R_t \rightarrow t}$.

24. Дополнительная операция: объединение двух **BST-**деревьев. Трудоёмкость операции — ***O(n log n).***

4.2.2. Методические указания к выполнению задания

1. Создание коллекции «**BST**-дерево» выполняется в соответствии с технологией реализации коллекций, изложенной в разделе 1. Для АТД «**BST-**

дерево» разрабатываются формат АТД и шаблон класса-контейнера. Параметры шаблона задают тип ключа и тип данных. В классе дерева определены внутренние классы «Узел дерева» и «Прямой итератор», «Обратный итератор».

2. Для реализации операций АТД «**BST**-дерево» рекомендуется использовать алгоритмы, приведённые в приложении В.

3. Для тестирования разработанного класса-контейнера разрабатываются две программы: программа тестирования операций через меню и программа тестирования трудоёмкости операций поиска, вставки и удаления.

4. Тестирование операций через меню выполняется для **BST**-дерева небольшого размера (до 10 элементов). Тестирующая программа выполняет вызов метода коллекции для выбранной операции **без предварительной проверки входных параметров метода и состояния коллекции.** После выполнения операции можно вызвать метод вывода структуры дерева на экран. При выводе узлов дерева необходимо отражать только ключ поиска, хранящийся в узле. Также необходимо выводить вспомогательный параметр, если он введён в узел **BST**-дерева для дополнительной операции, заданной в варианте задания.

5. Тестирование трудоёмкости операций поиска, вставки и удаления выполняется в соответствии с технологией тестирования, изложенной в разделе 1.5 и реализованной в программе тестирования **BST**-дерева (приложение Г).

6. Перед тестированием эффективности операций задаётся размер дерева (100–100 000 элементов). После тестирования на экран выводятся размер дерева, теоретическая и экспериментальная оценки средней трудоёмкости операций поиска, вставки и удаления (среднее число пройденных узлов дерева).

4.3. Контрольные вопросы и упражнения

1. Перечислите достоинства и недостатки дерева на базе массива.

2. Перечислите достоинства и недостатки дерева на базе связной структуры с адресными указателями.

3. Перечислите основные операции **BST**-дерева и теоретические оценки их трудоёмкости.

4. Приведите схему структуры дерева на базе связной структуры после серии операций: вставка (10), вставка (6), вставка (23), вставка (15), вставка (4), вставка (9), вставка (7), вставка (13), вставка (14), удаление (23), удаление (9), удаление (10). Дерево первоначально пусто.

5. Приведите псевдокод операции инкремента для прямого итератора дерева.

6. Приведите псевдокод итерационного алгоритма обхода **BST**-дерева по схеме $L_t \to R_t \to t$.

7. Для дерева, построенного в упражнении 4, приведите последовательности элементов, полученные в результате обхода по схемам $t \to \to L_t \to R_t$, $L_t \to R_t \to t$ и $L_t \to t \to R_t$.

8. Для дерева, построенного в упражнении 4, приведите элемент с порядковым номером 6.

9. Приведите изображение изначально пустого дерева после вызовов алгоритма вставки в корень дерева последовательности ключей 15, 18, 7, 9, 4, 5, 14.

10. Приведите изображение дерева, полученного в результате работы алгоритма объединения исходных деревьев.

5. Практическая работа «Коллекция данных — сбалансированное дерево поиска»

Цели работы: изучение и исследование методов балансировки двоичных деревьев поиска на примере АТД «Сбалансированное дерево поиска»; освоение методики модификации коллекций с помощью механизма наследования классов.

5.1. Структуры сбалансированных деревьев

Как известно, **BST**-дерево сохраняет эффективность при условии, если ключи вставляемых элементов являются случайными, равномерно

распределёнными значениями. Если в дерево поступают упорядоченные серии ключей или удаляются, его сбалансированность нарушается, и в крайнем случае дерево вырождается в структуру, близкую к списку. Его эффективность резко ухудшается до оценки ***O(n).***

Существует несколько структур деревьев, которые аналогичны **BST**-дереву, но при этом их эффективность в любых условиях близка к оценке $O(\log_2 n)$. Такие деревья называются сбалансированными. Сбалансированным является такое дерево, высота которого логарифмически зависит от количества элементов в дереве ***n***. К наиболее известным сбалансированным деревьям относятся **рандомизированное** дерево [9], **AVL**-дерево [2, 3], **красно-черное дерево (RB**-дерево) [6, 9]. Также известны **2-3**–дерево и **2-3-4**-дерево, которые являются идеально сбалансированными [1, 6, 9].

В сбалансированные деревья введён механизм локальной балансировки, который контролирует и выравнивает высоты поддеревьев в процессе вставок и удалений узлов.

В **рандомизированном** дереве операция вставки принимает вероятностное решение о вставке нового узла либо в корень, либо в лист поддеревьев, через которые проходит путь операции. Вероятность появления нового узла в корне дерева или поддерева определяется как $1/(n+1)$, где ***n*** — число узлов в дереве или в поддереве. Тем самым воспроизводится псевдослучайный порядок поступающих ключей. Поскольку принимаемые алгоритмом решения являются случайными, при каждом выполнении алгоритма при одной и той же последовательности включений будут получаться деревья различной конфигурации.

Операция удаления в рандомизированном дереве также работает случайным образом. В основе удаления лежит замена удаляемого узла корнем его объединённых поддеревьев. В объединение поддеревьев внесена случайность: объединение ведётся на базе того поддерева, у которого большее значение ***n***. При объединении левого поддерева из ***n*** узлов с правым поддеревом из ***m*** узлов корнем объединённого дерева будет корень левого

поддерева с вероятностью ***n/(n+m)*** или корень правого поддерева с вероятностью ***m/(n+m)***.

Для поддержки алгоритмов вставки и удаления в структуру узла рандомизированного дерева введён дополнительный параметр ***n***, значение которого равно количеству узлов в поддереве данного узла. Этот параметр используется и корректируется операциями вставки и удаления элементов.

Несмотря на увеличение трудоёмкости операций при вычислении вероятностей с помощью генератора случайных чисел *rand()*, структура рандомизированного дерева остаётся близкой к сбалансированной структуре. Трудоёмкость операций в рандомизированном дереве близка к оценке случайного **BST**-дерева — $1{,}39 \log_2 n$.

Алгоритмы операций вставки, удаления элементов для рандомизированного дерева приведены в приложении Д.

Для контроля сбалансированности структуры в **AVL**-дерево введён критерий: для каждого узла высоты его двух поддеревьев различаются не более чем на 1. Операции вставки и удаления элементов могут привести к изменению высот поддеревьев в узлах, лежащих на пути к вставленному или удалённому элементу. Поэтому после вставки или удаления элемента выполняется восходящая по пройденному пути проверка и корректировка критериев сбалансированности этих узлов. Если в каком-либо узле обнаружено, что разность высот поддеревьев стала больше единицы, выполняется поворот этого узла для выравнивания высот поддеревьев. В **AVL**-дереве используются четыре вида поворотов в зависимости от конфигурации поддеревьев узла с нарушенным критерием сбалансированности. На рисунках 20 и 21 показаны однократный и двойной повороты узла, разбалансированного влево.

Повороты узла, разбалансированного вправо, выполняются по симметричной схеме и носят названия однократного L-поворота и двойного RL-поворота.

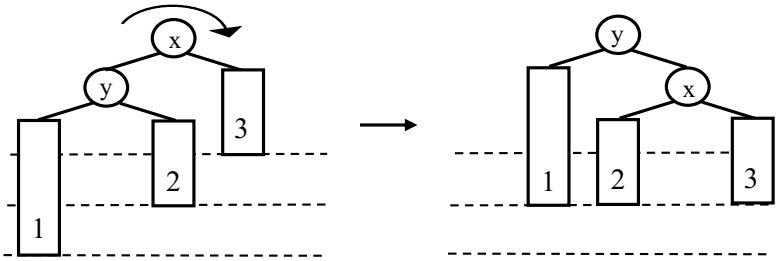


Рис. 20. Схема однократного R-поворота

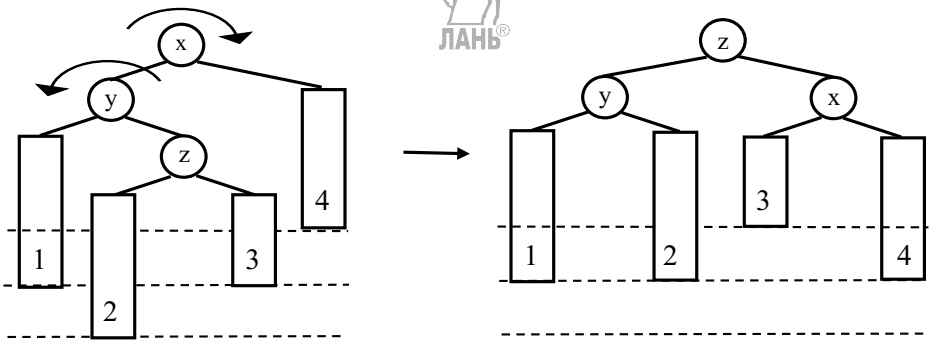


Рис. 21. Схема двойного LR-поворота

Для контроля сбалансированности узлов в их структуру введён параметр $bal = h_r - h_l$, где h_l и h_r — высота левого и правого поддеревьев узла. Узел сбалансирован, если значение ***bal*** равно –1, 0 или +1. Если в результате вставки или удаления значение ***bal*** становится равным –2 или +2, операция выполняет какой-либо из поворотов в зависимости от конфигурации поддеревьев разбалансированного узла.

Трудоёмкость AVL-дерева соответствует оценке $O(\log_2 n)$, точнее $\log_2 n + 0{,}25$.

Алгоритмы операций вставки и удаления для **AVL**-дерева приведены в приложении Д.

В отличие от описанных выше бинарных сбалансированных деревьев **2-3**-дерево не является двоичным деревом. Его узлы могут иметь 2 или 3 сына. Все пути от корня до любого листа имеют одинаковую длину. Данные с ключами располагаются только в листьях дерева по возрастанию ключей.

2-3-дерево (рис. 22) соответствует полному, идеально сбалансированному **BST-**дереву, если все его узлы являются 2-узлами. В этом случае его высота

равна $\boldsymbol{log_2 n}$. Если же все узлы в 2-3-дереве — 3-узлы, то высота 2-3-дерева равна $\boldsymbol{log_3 n}$. Таким образом, длина пути от корня дерева к листьям с данными и, следовательно, трудоёмкость операций лежит в пределах $\boldsymbol{O(log_3 n)–O(log_2 n).}$

Для организации поиска элементов внутренние узлы 2-3-дерева содержат дубликат наименьшего ключа во втором поддереве и, если у узла есть третий сын, дубликат наименьшего ключа в третьем поддереве. Эти ключи служат границами поиска в 2-3-дереве нужного листа с заданным ключом и данными. При проходе от корня к листу операции поиска, вставки и удаления сравнивают в каждом внутреннем узле искомый ключ с одной или двумя границами и выбирают соответствующего сына для дальнейшего спуска по дереву.

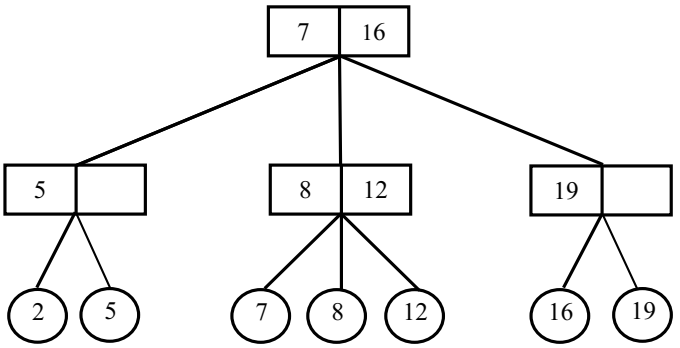


Рис. 22. Структура 2-3-дерева

Операция вставки добавляет данные в соответствующий лист 2-3-дерева. При этом в родительском узле этого листа может возникнуть нарушение ограничения на число сыновей. Если в результате операции вставки новый лист становится четвертым сыном, то родительский узел расщепляется на два 2-узла. Образовавшийся при расщеплении новый внутренний узел фиксируется в структуре родительского узла и также может вызвать расщепление родительского узла, если становится его четвёртым сыном. Процесс расщепления может достигнуть корня 2-3-дерева и привести к созданию нового корня.

Операция удаления может привести к тому, что у родительского узла останется один сын. Это может вызвать слияние родительского узла с соседним

узлом. Процесс слияния также может достичь корня и вызвать его замену на узел, образованный в результате слияния двух сыновей корня.

Если в операциях вставки или удаления создаётся новый корень, то одновременно изменяется длина всех путей от корня к листьям, и 2-3-дерево остаётся идеально сбалансированным.

Алгоритмы операций вставки и удаления для 2-3-дерева приведены в приложении Д.

Аналогом 2-3-дерева является **2-3-4-дерево** [3, 9]. Узлы 2-3-4-дерева содержат 2, 3 или 4 сына. Но ключи и данные хранятся не только в листьях, но и во внутренних узлах 2-3-4-дерева. Ключи во внутренних узлах одновременно служат границами поиска в дереве. Операции вставки и удаления, как и в 2-3-дереве, используют процедуру расщепления и слияния узлов при нарушении ограничения на количество сыновей. Точкой роста 2-3-4-дерева также является корень дерева, и, следовательно, 2-3-4-дерево является идеально сбалансированным. Трудоёмкость операций в 2-3-4-дереве лежит в границах $O(\log_4 n)$–$O(\log_2 n)$.

Ввиду сложности структуры узлов 2-3-4-дерева алгоритмы операций для него становятся громоздкими и непроизводительными. Поэтому непосредственная реализация 2-3-4-дерева редко используется. Взамен предлагается **красно-чёрное дерево (RB**-дерево), моделирующее структуру и алгоритмы 2-3-4-дерева. В RB-дереве 2-, 3- и 4-узлы заменяются группами 2-узлов (рис. 23).

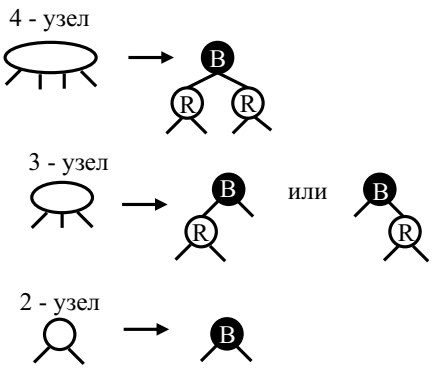


Рис. 23. Модели узлов 2-3-4-дерева

Для различения моделируемых узлов 2-3-4-дерева в группах принята раскраска узлов. Корневой узел группы имеет чёрный цвет (B), внутренние узлы окрашиваются в красный цвет (R). Таким образом, формируется двоичный эквивалент 2-3-4-дерева — RB-дерево. (рис. 24).

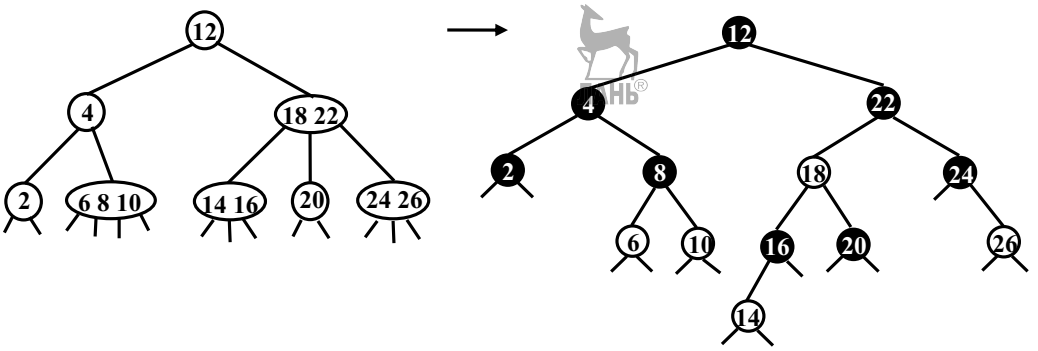


Рис. 24. Структуры 2-3-4-дерева и RB-дерева

Для раскраски в узел RB-дерева вводится дополнительный параметр — цвет узла ***color***, который имеет значение «красный» или «чёрный».

Благодаря переходу к двоичному аналогу 2-3-4-узлов в RB-дереве значительно упрощаются алгоритмы операций для 2-3-4-дерева. В приложении Д приведён рекурсивный алгоритм вставки в RB-дерево [9], сохранивший действия операций для 2-3-4-дерева (вставка в узел с изменением его типа, расщепление 4-узла).

Широкое распространение получили также итеративные алгоритмы вставки и удаления, которые в своей работе базируются на формальных свойствах структуры RB-дерева [6]:

1) каждый узел либо красный, либо чёрный,
2) каждый лист — чёрный,
3) если узел красный, то оба его сына — чёрные,
4) все пути, ведущие вниз от корня к листьям, содержат одинаковое количество чёрных узлов, то есть все пути имеют одинаковую чёрную высоту,
5) корень — чёрный.

Необходимо отметить ещё одну особенность структуры RB-дерева — наличие в дереве фиктивного чёрного узла. В терминальных узлах RB-дерева

вместо адресного значения ***nil*** хранится адрес этого фиктивного узла. Узел введён для упрощения программирования итеративного алгоритма удаления.

Начальная стадия операций вставки и удаления выполняется в RB-дереве точно так же, как и в **BST**-дереве. Новый узел вставляется как лист дерева и всегда окрашивается в красный цвет. Это может привести к нарушению третьего свойства структуры RB-дерева, если родитель нового узла тоже красный. При удалении узла из RB-дерева может быть нарушено четвертое свойство структуры, если удалённый узел был чёрным. Алгоритмы операций вставки и удаления восстанавливают свойства RB-дерева путём восходящих поворотов и перекрасок узлов, лежащих на пути вставки или удаления. Тем самым достигается сбалансированность RB-дерева. Средняя трудоёмкость операций в RB-дереве соответствует оценке $O(\log_2 n)$. Трудоёмкость RB-дерева, сформированного значениями ключей с равномерным законом распределения, соответствует оценке $1{,}002 \log_2 n$.

5.2. Задание к практической работе

1. Спроектировать, реализовать и провести тестовые испытания АТД «Сбалансированное дерево поиска» для коллекции, содержащей данные произвольного типа. Тип данных задаётся клиентской программой.

АТД «Сбалансированное дерево поиска» представляет собой модифицированную версию АТД «**BST**-дерево» с трудоёмкостью операций $O(\log_2 n)$.

Интерфейс АТД «Сбалансированное дерево поиска» включает следующие операции:

- конструктор;
- конструктор копирования;
- деструктор;
- опрос размера дерева;
- очистка дерева;
- проверка дерева на пустоту;

- доступ по чтению/записи к данным по ключу;
- включение данных с заданным ключом;
- удаление данных с заданным ключом;
- запрос прямого итератора, установленного на узел дерева с минимальным ключом *begin()*;
- запрос обратного итератора, установленного на узел дерева с максимальным ключом *rbegin()*;
- запрос «неустановленного» прямого итератора *end()*;
- запрос «неустановленного» обратного итератора *rend()*.

Операции прямого и обратного итераторов:

- операция доступа по чтению и записи к данным текущего узла *,
- операция перехода к следующему (для обратного — к предыдущему) по ключу узлу в дереве ++,
- операция перехода к предыдущему (для обратного — к следующему) по ключу узлу в дереве --,
- проверка равенства однотипных итераторов ==,
- проверка неравенства однотипных итераторов !=.

Для тестирования коллекции интерфейс АТД «Сбалансированное дерево поиска» включает дополнительные операции:

- вывод структуры дерева на экран (для узлов отображать ключи и параметр для балансировки),
- опрос числа просмотренных операцией узлов дерева.

2. Выполнить отладку и тестирование всех операций АТД «Сбалансированное дерево поиска» с помощью меню операций.

3. Выполнить сравнительное тестирование трудоёмкости операций вставки, удаления и поиска для коллекций «**BST**-дерево» и «Сбалансированное дерево поиска» для случайной и вырожденной структуры дерева.

4. Выполнить сравнительный анализ теоретических и экспериментальных показателей трудоёмкости операций.

5. Составить отчёт по лабораторной работе. Отчёт должен содержать следующие пункты:

1) титульный лист;

2) цель лабораторной работы;

3) общее задание и вариант задания;

4) формат АТД «Сбалансированное дерево поиска»;

5) формат АТД «Прямой итератор сбалансированного дерева поиска»;

6) формат АТД «Обратный итератор сбалансированного дерева поиска»;

7) справочное определение класса для коллекции «Сбалансированное дерево поиска», предназначенное для клиентской программы;

8) описание методики сравнительного тестирования трудоёмкости операций BST-дерева и сбалансированного дерева поиска;

9) таблицы и графики с полученными оценками трудоёмкости для худшего и среднего случаев функционирования BST-дерева и сбалансированного дерева поиска. Должны быть приведены графики трудоёмкости для операций поиска, вставки и удаления (графики обеих коллекций совмещены в одной системе координат);

10) сравнительный анализ теоретических и экспериментальных оценок трудоёмкости для операций BST-дерева и сбалансированного дерева поиска;

11) выводы;

12) список использованной литературы;

13) приложение с текстами программ:

- текст полного определения и методов класса сбалансированное дерево поиска;
- текст программы-меню для тестирования отдельных операций сбалансированного дерева поиска;
- текст программы сравнительного тестирования трудоёмкости операций **BST**-дерева и сбалансированного дерева поиска.

5.2.1. Варианты заданий

1. AVL-дерево как модификация **BST**-дерева. Алгоритмы операций вставки, удаления и поиска реализуются в рекурсивной форме.

2. AVL-дерево как модификация **BST**-дерева. Алгоритмы операций вставки, удаления и поиска реализуются в итерационной форме.

3. RB-дерево как модификация **BST**-дерева. Алгоритмы операций вставки, удаления и поиска реализуются в рекурсивной форме.

4. RB-дерево как модификация **BST**-дерева. Алгоритмы операций вставки, удаления и поиска реализуются в итерационной форме.

5. 2-3-дерево. Алгоритмы операций вставки, удаления и поиска реализуются в рекурсивной форме.

6. 2-3-дерево. Алгоритмы операций вставки, удаления и поиска реализуются в итерационной форме.

7. Рандомизированное дерево как модификация **BST**-дерева. Алгоритмы операций вставки, удаления и поиска реализуются в рекурсивной форме.

8. Рандомизированное дерево как модификация **BST**-дерева. Алгоритмы операций вставки, удаления и поиска реализуются в итерационной форме.

5.2.2. Методические указания к выполнению задания

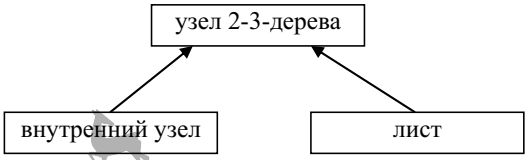
1. Коллекции «Сбалансированное дерево поиска», использующие рандомизированное дерево, AVL-дерево, RB-дерево, разрабатываются как модификация класса «**BST**-дерево» с использованием технологии наследования классов.

2. Коллекция на базе 2-3-дерева разрабатывается как самостоятельный класс. Для реализации двух разновидностей узлов в 2-3-дереве использовать иерархию классов.

3. Для реализации операций АТД рекомендуется использовать алгоритмы, приведённые в приложении Д.

4. Для тестирования разработанного класса-контейнера разрабатываются две программы: программа тестирования операций через

меню и программа тестирования трудоёмкости операций поиска, вставки и удаления.



5. Тестирование операций через меню выполняется для небольшого размера дерева (до 10 элементов). Тестирующая программа выполняет вызов метода коллекции для выбранной операции без предварительной проверки входных параметров и состояния коллекции. После выполнения операций необходимо вывести на экран содержимое дерева с помощью операции вывода структуры дерева. При выводе узла дерева необходимо отражать хранящийся в нём ключ поиска и дополнительный параметр, используемый для балансировки узла.

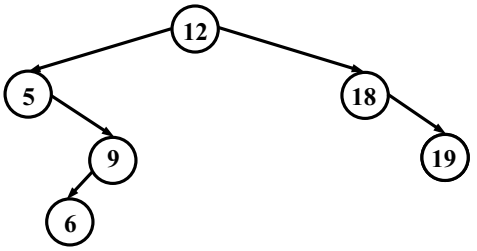
6. Сравнительное тестирование трудоёмкости операций коллекций «**BST**-дерево» и «Сбалансированное дерево поиска» выполняется в соответствии с технологией тестирования, изложенной в разделе 1.5.

7. Перед тестированием эффективности операций для обеих коллекций задаётся размер деревьев. Размер варьируется в пределах от 10 до 100 000 элементов. После тестирования на экран выводятся размер деревьев и средняя трудоёмкость операций поиска, вставки и удаления (среднее число пройденных узлов дерева).

8. Для рандомизированного дерева реализуются две версии для операций вставки и удаления. Также проводится сравнительное тестирование трудоёмкости операций для рандомизированных деревьев, использующих различные версии операций вставки и удаления.

5.3. Контрольные вопросы и упражнения

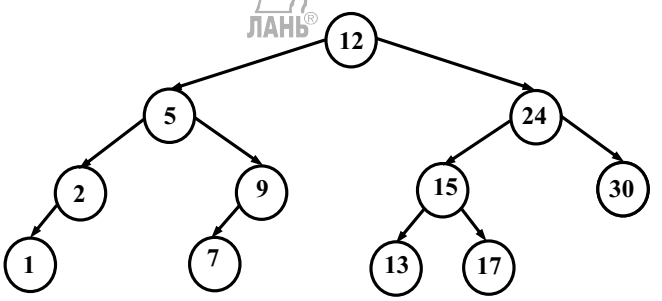
1. Приведите изображение структуры, полученной в результате работы операции вставки ключа 7 в рандомизированное дерево:



Значение RAND_MAX = 32767.

В процессе работы алгоритма генератор случайных чисел Rand() вычисляет следующую последовательность значений: 6782, 12653, 5187, 154, 23567, … .

2. Приведите изображение структуры AVL-дерева после последовательного выполнения операции вставки ключей 3, 14, 18:



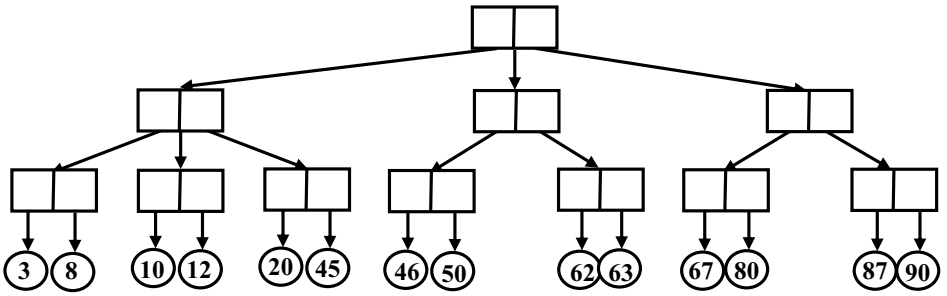
Приведите псевдокод алгоритма определения высоты AVL-дерева, имеющего трудоёмкость $O(\log_2 n)$.

3. Приведите изображение структуры изначально пустого RB-дерева после вставки ключей 30, 40, 20, 90, 10, 50, 70, 60, 80.

4. Приведите изображение структуры RB-дерева, полученного в упражнении 3, после удаления ключей 20, 40.

5. Приведите изображение структуры изначально пустого 2-3-дерева после вставки ключей 14, 4, 3, 45, 6, 30, 4, 35, 10.

6. Приведите вид заданной структуры 2-3-дерева после удаления ключей 8, 20, 67, 90.



6. Практическая работа «Коллекция данных — хеш-таблица»

Цели работы: изучение методов построения таблиц с постоянным временем доступа к элементам; освоение технологии реализации таблиц на примере АТД «Хеш-таблица».

Рассмотренные ранее коллекции на основе деревьев поиска позволяют строить абстрактный тип данных «Таблица», типичными операциями которой являются поиск, вставка и удаление элементов по ключу. Но эти коллекции обладают логарифмической трудоёмкостью операций, зависящей от числа элементов в таблице. Встречаются приложения, требующие быстрого и гарантированного по времени выполнения, не зависящего от объёма данных в таблице. В частности, это требование свойственно программным системам, работающим в реальном времени. В этом случае используется специальная организация таблицы с постоянным временем выполнения операций — хеш-таблица [1, 3, 6, 9].

Прообразом хеш-таблицы является таблица с прямой адресацией, представляющая собой массив, каждая позиция которого соответствует отдельному значению ключа. В этом случае ключ используется как индекс массива, и тем самым обеспечивается быстрый, прямой доступ к элементам таблицы. Реализовать такую таблицу можно, если разброс значений ключей невелик. На практике это условие редко выполнимо.

Если количество позиций в таблице существенно меньше, чем количество возможных значений ключей, то используется хеш-таблица. Хеш-таблица представляет собой массив с размером m, который может хранить $n > m$ значений. В нём прямое индексирование по ключу заменяется преобразованием

значения ключа в индекс массива с помощью специальной функции. Процесс преобразования ключа ***k*** в индекс ***i*** называется хешированием, и в хеш-таблице для этого используется специальная хеш-функция ***h(k)→ i, I = 0..m–1***.

6.1. Методы хеширования ключей

При выборе хеш-функции заранее неизвестно, какие именно значения ключей будут храниться в хеш-таблице. На всякий случай хеш-функцию разумно сделать «случайной», обеспечивающей равномерное хеширование, т. е. для любого ключа все ***m*** индексов таблицы должны быть равновероятны. С другой стороны, случайная функция должна быть детерминированной в том смысле, что при повторных вызовах с одним и тем же ключом она должна возвращать в качестве индекса одно и то же хеш-значение.

6.1.1. Преобразование ключей перед хешированием

Существует много удобных способов хеширования, использующих преобразование натурального ключа к натуральному индексу, лежащему в диапазоне ***0..m***. Если ключ имеет иную природу, то перед хешированием значение ключа *k* приводится к натуральному числу – значению *k'*. Например, если ключ — строка текста, то её отдельные символы можно интерпретировать как цифры числа, записанного в системе счисления с основанием 256. Вещественные числа можно привести к натуральному числу с некоторой заданной точностью, предварительно умножив на соответствующую степень 10.

Рассмотрим несколько методов преобразования ключей различной природы к натуральным значениям перед хешированием.

Если натуральные многоразрядные ключи принадлежат большому числовому интервалу, то можно уменьшить разрядность ключей и тем самым перевести их в меньший числовой интервал. Метод **выбора цифр** формирует новое значение *k'* как комбинацию отдельных цифр ключа *k*. Например, из 12-значного ключа можно выбрать нулевую, пятую и десятую цифры. Запись этих цифр в позиционной десятичной системе и будет давать трёхзначное натуральное число, которое будет превращено в индекс с помощью хеш-

функции. Но такой способ оправдывает себя, если выбор цифр не приводит к коллизии различных ключей ещё на стадии выбора отдельных цифр. Например, при нумерации цифр справа налево ключи k_1 = 1**2**3456**7**8912**3** и k_2 = 4**2**5426**7**5432**3** при выборе 0, 5 и 12-й цифр дадут одно и то же число k' = **273**. Эту проблему можно решить, если значения выбираемых цифр модифицировать суммой пропущенных цифр. Например:

для k_1 = 1**2**3456**7**8912**3** k' = (1 + **2**) mod 10, (3 + 4 + 5 + 6 + **7**) mod 10, (8 + 9 + 1 + + 2 + **3**) mod 10 = **353**,

для k_2 = 4**2**5426**7**5432**3** k' = (4 + **2**) mod 10, (5 + 4 + 2 + 6 + **7**) mod 10, (5 + 4 + 3 + + 2 + **3**) mod 10 = **647.**

Ещё один способ, основанный на выборе цифр из многоразрядного натурального ключа, известен как метод «середины квадрата» [4]. Значение ключа возводится в квадрат, и из середины полученного числа выбирается нужное количество цифр.

Например, пусть k = 134295445 и k^2 = 18035266**574**60711401. Можно выбрать 3 цифры из середины k^2 и сформировать число k'= **574** для функции хеширования.

Метод свёртки выделяет в многоразрядном натуральном ключе k группы по n цифр и складывает группы цифр по модулю 10^n. Например, для группы размером n = 3 ключ k_1 = 123342954451 можно преобразовать по схеме

$$k' = (123 + 342 + 954 + 451) \bmod 10^3 = \mathbf{870}.$$

Если другой ключ k_2 = 123342451954 имеет те же группы цифр в ином порядке, то после свёртки получится такое же число:

$$k' = (123 + 342 + 451 + 954) \bmod 10^3 = \mathbf{870}.$$

Чтобы избежать коллизий ключей при свёртке, можно модифицировать значения групп с учётом их порядкового номера. Например, с помощью операции «исключающее ИЛИ» — xor:

для k_1 = 123342954451 k' = (123 xor 1 + 342 xor 2 + 954 xor 3 + 451 xor 4) mod 10^3 = (122 + 340 + 953 + 455) mod 10^3 = **870**,

для k_2 = 123342451954 k' = (123 xor 1 + 342 xor 2 + 451xor 3 + 954 xor 4) mod 10^3 = = (122 + 340 + 448 + 958) mod 10^3 = **868**.

Если ключ k является строкой символов произвольной длины, то нужно применить преобразование строки к натуральному числу k' перед хешированием. Например, если строка включает только заглавные буквы латинского алфавита, то каждой букве от A до Z можно поставить в соответствие числа от 1 до 26. Строка символов заменяется **конкатенацией битовых образов** этих чисел. Десятичное значение получившейся битовой последовательности используется в качестве k' для хеширования [3]. Например, строковый ключ ALPHA можно преобразовать этим способом.

Буква A кодируется числом 1, или 00001_2.

Буква L кодируется числом 12, или 01100_2.

Буква P кодируется числом 16, или 10000_2.

Буква H кодируется числом 8, или 01000_2.

Собирая эти битовые образы в целочисленной переменной, получаем двоичное значение 00001011001000001000000001, которое интерпретируется как десятичное значение k' = 1458433.

Другой способ преобразования ключа-строки — это вычисление k' как числа в позиционной системе счисления по **правилу Горнера.** Например, если ключ-строка состоит только из заглавных букв латинского алфавита, то ASCII — код каждой буквы — преобразуется в её порядковый номер в алфавите. Этот номер используется в качестве значения цифры в позиционной системе счисления с основанием 32, наиболее соответствующим мощности латинского алфавита (26 букв) и кириллицы (33 буквы). Пример: k = "ALPHA",

$k' = 1 \times 32^4 + 12 \times 32^3 + 16 \times 32^2 + 8 \times 32^1 + 1 \times 32^0 =$ **1458433.**

Такие степенные полиномы эффективнее вычисляются с помощью схемы Горнера:

$k' = (((1 \times 32 + 12) \times 32 + 16) \times 32 + 8) \times 32 + 1 =$ **1458433.**

6.1.2. Хеш-функции

После преобразования ключа ***k*** к натуральному значению ***k'*** можно вычислять индекс для хеш-таблицы с помощью хеш-функции ***h(k').***

В дальнейшем изложении обозначим для хеш-функции преобразованный ключ ***k'*** как ***k.***

Хеш-функция, основанная на **методе деления с остатком (модульное хеширование),** даёт простое и равномерное вычисление индексов для хеш-таблицы. Ключу ***k*** ставится в соответствие остаток от деления ***k*** на размер хеш-таблицы (***m***). Хеш-функция вычисляется как уравнение ***h(k) = k mod m***.

Метод хорошо работает, если в качестве ***m*** выбирать простые числа, близкие к степени двойки. Ниже приведены рекомендуемые размеры хеш-таблиц (числа Мерсенне) для модульного хеширования [9]:

Таблица 2. Числа Мерсенне

n	**δ**	**$2^n - δ$**
8	5	251
9	3	509
10	3	1021
11	9	2039
12	3	4093
13	1	8191
14	3	16381
15	19	32749
16	15	65521
17	1	131071
18	5	262139
19	1	524287
20	3	1048573

Другой вариант хеш-функции использует **метод умножения (мультипликативный).** Функция вычисляет хеш-значение по формуле

$$h(k) = \lfloor m \times (k \times A \bmod 1) \rfloor,$$

где

- A — некоторая константа, удовлетворяющая условию $0 < A < 1$,
- $k \times A \bmod 1$ — дробная часть произведения $k \times A$,
- $\lfloor m \times (k \times A \bmod 1) \rfloor$ — целая часть произведения $m \times (k \times A \bmod 1)$.

Достоинство метода умножения заключается в том, что качество хеш-функции мало зависит от выбора значения m. Но всё-таки в качестве m выбирают степень двойки, так как в большинстве компьютеров умножение на степень двойки реализуется быстро, как сдвиг слова.

Метод умножения работает при любом выборе константы A, но некоторые значения A могут быть лучше других. Удачным значением является число — «золотое сечение» [6]:

$$A = (\sqrt{5} - 1)/2 \approx \mathbf{0{,}6180339887}.$$

6.1.3. Качество хеширования

Теоретический выбор способа преобразования ключей и хеш-функции не даёт гарантии хорошего качества распределения ключей в хеш-таблице. На стадии проектирования хеш-таблицы полезно дополнительное экспериментальное исследование равномерности распределения ключей с помощью выбранного способа хеширования.

Показателем равномерности распределения ключей по пространству таблицы может служить статистический критерий χ^2 — распределения [9]:

$$\chi^2 = \frac{m}{P} \times \sum_{0 \le i \le m} \left(f_i - \frac{P}{m} \right)^2,$$

где P — количество ключей в выборке, использованной для получения распределения; m — размер хеш-таблицы; f_i — количество ключей с хеш-значением, равным индексу i.

Если использованные ключи являются случайными на всём пространстве значений ключей U, значение функции χ^2 должно быть равно $m \pm \sqrt{m}$ с вероятностью $1 - 1/c$, где $c = U/m$.

Для обеспечения достоверных результатов исследования необходимо обеспечить объём выборки ключей $P \geq 20 \times m$, мощность пространства ключей $|U|$, обеспечивающую вероятность $1 - 1/c \approx 1$, т. е. величину U/m, приближающуюся к «бесконечно» большому числу.

6.2. Разрешение коллизий и структуры хеш-таблиц

Проблема, свойственная хеш-таблицам, заключается в том, что индексы (хеш-значения) нескольких разных ключей могут совпадать. Такой случай называется **коллизией,** или столкновением. Эта ситуация неизбежна, поскольку количество возможных значений ключей U значительно превышает размер хеш-таблицы.

Существуют два подхода к разрешению коллизий. Во-первых, можно задать структуру хеш-таблицы таким образом, чтобы каждая ячейка таблицы хранила несколько элементов с одинаковым хеш-значением ключей. Во-вторых, можно найти другую ячейку в хеш-таблице и поместить туда элемент, вступивший в коллизию. В соответствии с этими подходами существуют два типа хеш-таблиц с различной структурой.

Хеш-таблица с цепочками коллизий (рис. 25) представляет собой массив списков коллизий. В позиции i массива хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно i. Если таких элементов в таблице нет, то позиция i содержит нулевой адрес ***nil***. Такой способ разрешения коллизий снимает ограничения на количество элементов, включённых в таблицу.

Хеш-таблица с открытой адресацией представляет собой массив T, в который помещаются данные по индексу, равному хеш-значению ключа. При возникновении коллизии выполняется поиск другой ячейки массива (зондирование), в которой может разместиться элемент. Эта ячейка должна быть свободной (открытой). Для пометки состояния в ячейки таблицы вводится специальный признак состояния ячейки ***state***. Ячейка хеш-таблицы может быть

в одном из трех состояний: свободна (***free***), занята (***busy***), свободна после удаления из неё элемента (***deleted***). Открытой считается ячейка, находящаяся в состоянии ***free*** или ***deleted***.

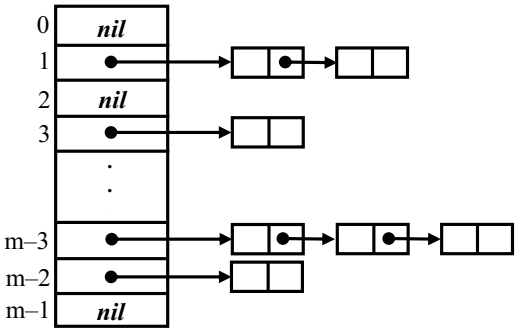


Рис. 25. Хеш-таблица с цепочками коллизий

При поиске свободной ячейки вычисляется и просматривается последовательность индексов ячеек массива. Последовательность зависит от ключа *k* и номера попытки зондирования ***i***. Для вычисления последовательности к хеш-функции добавляется второй аргумент — номер попытки (нумерация начинается с 0). Последовательность ячеек для некоторого ключа ***k*** имеет вид ***<h (k, 0), h (k, 1), …, h (k, m–1)>***.

Обычно используются три способа поиска свободных ячеек, называемых «линейное зондирование», «квадратичное зондирование» и «зондирование с двойным хешированием».

При **линейном зондировании** хеш-функция при каждой попытке поиска вычисляет новую ячейку по формуле

$$h(k, i) = (h(k) + i) \bmod m,$$

где ***h(k)*** — обычная хеш-функция, ***i*** — номер попытки поиска свободной ячейки.

При работе с ключом ***k*** первой выбирается ячейка ***T*** **[*h(k)*]**, а затем перебираются ячейки таблицы подряд: ***T*** **[*h(k)+1*]**, ***T*** **[*h(k)+2*]**, …. После ячейки ***T*** **[*m–1*]** следует ячейка ***T*** **[*0*]**. У метода очевидный недостаток: он может

привести к образованию кластеров — длинных последовательностей занятых ячеек, идущих подряд. Это удлиняет поиск.

При **квадратичном зондировании** хеш-функция задаёт квадратичную последовательность ячеек по формуле

$$h(k, i) = (h(k) + c_1 \times i + c_2 \times i^2) \bmod m,$$

где $c_1 \neq 0$ и $c_2 \neq 0$ — некоторые константы. Пробы начинаются с ячейки $T[h(k)]$, но дальше ячейки просматриваются не подряд. Номер пробуемой ячейки квадратично зависит от номера попытки. Для того чтобы использовались все ячейки таблицы, c_1 и c_2 нужно тщательно подбирать.

При квадратичном зондировании тенденция к образованию кластеров частично устраняется. Хотя и остается эффект кластеризации в более мягкой форме — образование вторичных кластеров.

Двойное хеширование — один из лучших методов зондирования в хеш-таблице с открытой адресацией. Последовательности индексов, возникающие при зондировании с двойным хешированием, близки к равномерному хешированию. При двойном хешировании функция h имеет вид:

$$h(k, i) = (h(k) + i \times h_1(k)) \bmod m,$$

где $h(k)$ и $h_1(k)$ — обычные хеш-функции. То есть последовательность проб при зондировании является арифметической прогрессией по модулю m с первым членом $h(k)$ и шагом $h_1(k)$.

Чтобы зондирование ячеек охватывало все индексы таблицы, значение $h_1(k)$ должно быть взаимно простым с размером хеш-таблицы m. Так, для мультипликативной хеш-функции $h(k)$ размер хеш-таблицы m задаётся как степень двойки, а функция $h_1(k)$ — любое нечётное натуральное число. Если $h(k)$ использует модульное хеширование и m — простое число, то значения $h_1(k)$ — любое натуральное число, меньшее m:

$$h(k) = k \bmod m,$$

$$h_1(k) = 1 + (k \bmod m_1),$$

где m_1 чуть меньше, чем m ($m_1 = m-1$ или $m-2$).

6.3. Трудоёмкость операций

Алгоритмы чтения или модификации, вставки и удаления элементов зависят от структуры хеш-таблицы. Если используется хеш-таблица с цепочками коллизий, то алгоритмы поиска, вставки и удаления элемента действуют по однотипной схеме:

- приведение ключа к натуральному значению: $k \rightarrow k'$,
- вычисление индекса списка в массиве цепочек коллизий: $i = h(k')$,
- поиск в списке элемента с ключом k,
- выполнение операции над элементом.

Трудоёмкость операций зависит от средней длины списков, которую можно вычислить как $\alpha = n/m$, где n — число занесенных в таблицу элементов, m — размер хеш-таблицы. С учётом первоначального хеширования ключа перед просмотром списка оценку трудоёмкости можно записать как $O(1+\alpha/2)$ [3]. Величина α учитывает фактор заполнения таблицы и называется коэффициентом заполнения таблицы. Трудоёмкость операций в хеш-таблице зависит не от числа элементов n, содержащихся в ней, а от коэффициента заполнения α. Для хеш-таблиц с цепочками коллизий величина α может быть меньше и больше единицы.

Зависимость трудоёмкости операций от заполнения справедлива и для хеш-таблиц с открытой адресацией. Чем более заполнена хеш-таблица, тем больше возникает коллизий и проб при зондировании. Приведённые в приложении Е алгоритмы трёх основных операций для хеш-таблицы с открытой адресацией демонстрируют этот процесс.

Так же как при анализе хеширования с цепочками коллизий, трудоёмкость операций с открытой адресацией оценивается в терминах коэффициента заполнения α. Для хеш-таблицы с открытой адресацией $\alpha \leq 1$. Анализ трудоёмкости операций для хеш-таблиц с открытой адресацией более сложен, чем для хеш-таблиц с цепочками коллизий [3, 6]. Помимо коэффициента α трудоёмкость операции зависит от её алгоритма, метода зондирования, вероятности промахов операций. Поскольку в основе всех

операций лежит определение наличия или отсутствия ключа в таблице, то трудоёмкость операций оценивается в количестве проб зондирования, выполненных при поиске ключа. Ниже приведены оценки трудоёмкости успешного или неуспешного поиска ключа в зависимости от метода зондирования [9]:

Таблица 3

Метод зондирования	Успешный поиск ключа	Неуспешный поиск ключа
Линейное зондирование	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$
Квадратичное зондирование	$\frac{-\ln(1-\alpha)}{\alpha}$	$\frac{1}{1-\alpha}$
Зондирование с двойным хешированием	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$	$\frac{1}{1-\alpha}$

6.4. Задание к практической работе

АТД «Хеш-таблица» представляет ассоциативное, неупорядоченное множество данных, доступ к которым выполняется с использованием ключей.

1. Спроектировать, реализовать и провести тестовые испытания АТД «Хеш-таблица» для коллекции, содержащей данные произвольного типа. Тип ключей и данных задаётся клиентской программой.

Хеш-таблица может быть представлена в одной из двух форм:

- хеш-таблица с цепочками коллизий;
- хеш-таблица с открытой адресацией.

Интерфейс АТД «Хеш-таблица» включает следующие операции:

- опрос размера таблицы;
- опрос количества элементов в таблице;
- опрос пустоты таблицы;
- очистка таблицы;
- поиск элемента по ключу k;

- вставка элемента по ключу *k*;
- удаление элемента по ключу *k*;
- задание формы представления — хеш-таблица с цепочками коллизий;
- задание формы представления — хеш-таблица с открытой адресацией;
- опрос формы представления;
- запрос прямого итератора *begin()*;
- запрос «неустановленного» прямого итератора *end()*;
- прямой итератор для доступа к значениям в хеш-таблице с основными операциями:
  - доступ по чтению и записи к значению текущего элемента;
  - переход к следующему по индексу элементу в хеш-таблице;
  - проверка равенства итераторов ==;
  - проверка неравенства итераторов !=.

Для тестирования коллекции интерфейс АТД «Хеш-таблица» включает дополнительные операции:

- вывод структуры хеш-таблицы на экран;
- опрос числа проб, выполненных последней операцией.

2. Для метода хеширования ($k \to k' + \boldsymbol{h(}k'\boldsymbol{)}$) получить экспериментальную оценку качества хеширования χ^2, усреднённую по нескольким экспериментам.

3. Выполнить отладку и тестирование всех операций для хеш-таблицы с цепочками коллизий и для хеш-таблицы с открытой адресацией с помощью меню операций.

4. Выполнить тестирование средней трудоёмкости операций поиска, вставки и удаления для хеш-таблицы с цепочками коллизий и хеш-таблицы с открытой адресацией в зависимости от коэффициента заполнения $\boldsymbol{\alpha}$.

5. Провести сравнительный анализ теоретических и экспериментальных показателей трудоёмкости операций.

6. Составить отчёт по лабораторной работе. Отчёт должен содержать следующие пункты:

1) титульный лист;

2) цель лабораторной работы;

3) общее задание и вариант задания;

4) формат АТД «Хеш-таблица»;

5) формат АТД «Прямой итератор хеш-таблицы»;

6) справочное определение класса для коллекции «Хеш-таблица», предназначенное для клиентской программы;

7) описание методики получения экспериментальной оценки χ^2 и полученная оценка χ^2, усреднённая по нескольким экспериментам;

8) описание методики тестирования трудоёмкости операций вставки, удаления и поиска;

9) таблицы и графики с полученными оценками трудоёмкости операций. Должны быть приведены графики трудоёмкости для операций поиска, вставки и удаления для хеш-таблицы с цепочками коллизий и хеш-таблицы с открытой адресацией;

10) выводы;

11) список использованной литературы;

12) приложение с текстами программ:

– полное определение классов и текстов методов для хеш-таблицы с цепочками коллизий и хеш-таблицы с открытой адресацией,

– текст программы-меню для тестирования отдельных операций хеш-таблицы,

– текст программы получения оценки χ^2,

– текст программы тестирования трудоёмкости операций АТД.

6.4.1. Варианты заданий

1. Тип ключа *k* — строка текста произвольной длины (символы — маленькие буквы латинского алфавита).

Преобразование к *k'* методом конкатенации битовых образов символов.

Метод хеширования — модульный.

Метод разрешения коллизий — двойное хеширование.

2. Тип ключа k — вещественное число на интервале [–10 000,000, +10 000,000].

Преобразование k к натуральному значению k' с точностью 10^{-3} и выбор 6 цифр из середины квадрата k'.

Метод хеширования — мультипликативный.

Метод разрешения коллизий — квадратичный.

3. Тип ключа k — целое число на интервале [0, +1 000 000 000].

Преобразование к k' методом выбора цифр.

Метод хеширования — модульный.

Метод разрешения коллизий — линейное хеширование.

4. Тип ключа k — строка текста произвольной длины (символы — заглавные буквы латинского алфавита).

Преобразование к k' по схеме Горнера.

Метод хеширования — мультипликативный.

Метод разрешения коллизий — линейное хеширование.

5. Тип ключа k — вещественное число на интервале [10 000,0000, +15 000,0000].

Преобразование k к натуральному значению с точностью 10^{-4} и свёртка к значению k' на интервале [10 000, +20 000].

Метод хеширования — модульный.

Метод разрешения коллизий — квадратичное хеширование.

6. Тип ключа k — строка текста произвольной длины (символы — заглавные буквы латинского алфавита).

Преобразование к k' методом конкатенации битовых образов символов.

Метод хеширования — мультипликативный.

Метод разрешения коллизий — квадратичный.

7. Тип ключа k — натуральное число на интервале [100 000 000, 300 000 000].

Преобразование к k' методом свёртки.

Метод хеширования — модульный.

Метод разрешения коллизий — квадратичный.

8. Тип ключа k — строка текста произвольной длины (символы — заглавные буквы кириллицы).

Преобразование к k' по схеме Горнера.

Метод хеширования — мультипликативный.

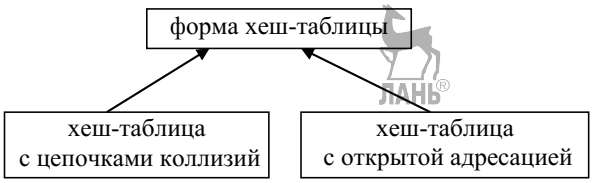
Метод разрешения коллизий — линейный.

6.4.2. Методические указания к выполнению задания

1. Параметрами шаблона класса «Хеш-таблица» являются тип ключа, тип данных.

2. Размер хеш-таблицы вычисляется конструктором коллекции в зависимости от заданного количества хранящихся элементов и формы хеш-таблицы, метода хеширования, указанного в варианте задания, и рекомендуемой оптимальной величины α.

3. Для изменения формы представления в классе хеш-таблицы рекомендуется использовать внутреннюю иерархию классов:



4. Для реализации операций хеш-таблицы с открытой адресацией рекомендуется использовать алгоритмы, приведённые в приложении Е.

5. Для получения оценки χ^2 использовать количество ключей, минимум в 20 раз превышающее размер хеш-таблицы.

6. Для тестирования разработанного класса-контейнера разрабатываются две программы — программа тестирования отдельных операций через меню и программа тестирования средней трудоёмкости операций поиска, вставки и удаления.

7. Тестирование операций через меню выполняется для небольшого размера таблицы (до 10 элементов). Тип данных, хранящихся в ней, задаётся с клавиатуры перед началом тестирования.

8. После выполнения операций для вывода на экран структуры хеш-таблицы используется отдельный пункт меню с вызовом операции вывода содержимого таблицы. При выводе структуры хеш-таблицы с цепочками коллизий в строках экрана отражать индекс таблицы и значения ключей k в цепочках. При выводе структуры хеш-таблицы с открытой адресацией в строках экрана отражать индекс таблицы, значение ключа k, состояние ячейки хеш-таблицы (***free /busy /deleted***).

9. Тестирование трудоёмкости операций коллекций «Хеш-таблица с цепочками коллизий» и «Хеш-таблица с открытой адресацией» выполняется в соответствии с технологией тестирования, изложенной в разделе 1.5.

10. Перед тестированием эффективности операций для обеих коллекций задаётся количество элементов, для хранения которых предназначена хеш-таблица. В качестве исходных данных для тестирования задаётся коэффициент заполнения $\boldsymbol{\alpha}$. Для хеш-таблицы с цепочками коллизий изменение $\boldsymbol{\alpha}$ определяется неравенством $0{,}1 \le \boldsymbol{\alpha} \le 10$. Для хеш-таблицы с открытой адресацией изменение $\boldsymbol{\alpha}$ задаётся неравенством $0{,}1 \le \boldsymbol{\alpha} \le 1$. После тестирования на экран выводятся коэффициент заполнения и полученные оценки трудоёмкости для операций поиска, вставки и удаления элементов.

6.5. Контрольные вопросы и упражнения

1. Приведите размер хеш-таблицы с открытой адресацией, предназначенной для хранения 200 элементов, если она использует мультипликативный метод хеширования, модульный метод хеширования.

2. Приведите вид первоначально пустой хеш-таблицы с цепочками коллизий после вставки ключей 1, 89, 45, 76, 2, 13, 33, 4. Хеш-таблица предназначена для хранения 20 элементов, использует модульное хеширование.

3. Приведите вид первоначально пустой хеш-таблицы с открытой адресацией после вставки ключей 1, 89, 45, 76, 2, 13, 33, 4. Хеш-таблица предназначена для хранения 10 элементов, использует модульное хеширование.

4. Приведите в соответствующий строке JKRSDS ключ, полученный методом конкатенации битовых образов для хеш-таблицы размером 1000.

5. Приведите соответствующий числу 1263454756 ключ, полученный методом свёртки для хеш-таблицы размером 1000.

6. Какая последовательность ячеек получается в результате квадратичного зондирования, если $h(k) = k \bmod 11$ и $k = 19$?

7. Какая последовательность зондируемых ячеек получается в результате двойного хеширования: $h(k) = k \bmod 11$, $h_1(k) = 1 + (k \bmod 10)$ и $k = 19$?

ПРИЛОЖЕНИЕ А. Основные правила и соглашения псевдокода

Чтобы не отвлекаться при изучении алгоритмов на особенности кодирования на конкретном языке программирования, принято вести записи алгоритмов на псевдокоде. Действия алгоритма описываются полуформально. Можно пропустить технические подробности программирования, затеняющие суть алгоритма.

1. Отступ от левого поля указывает на уровень вложенности. Это делает излишним использование блочных скобок ({} — в C++, begin–end — в Паскале).

2. Циклы **while, for, repeat, until** и условные операторы **if — then — else** имеют тот же смысл, что и в C++ и Паскале.

3. Символ ▷ начинает комментарий, идущий до конца строки.

4. **i←e** означает присваивание переменной **i** значения **e**.

5. **i←j←e** означает одновременное присваивание значения **e** переменной **i** и **j**.

6. Индекс массива указывается в квадратных скобках **A[i].** Знак "**..**" выделяет часть массива **A[i..j]** от **i**–го до **j**–го значения.

7. Индекс начинается с 1.

8. Переменные локальны внутри функции.

9. Часто используются объекты, состоящие из нескольких полей, называемых атрибутами. Значение поля записывается как **имя поля [имя объекта]**. Например, поле **left** с указателем на левого сына узла дерева **t** обозначается как **left[t].**

10. Переменная, означающая массив или объект, считается указателем на его данные. После присваивания **y←x** для любого поля **f** выполняется **f[y] = f[x].** Более того, если выполним **f[x]←3**, то будет и **f[y]=3**, поскольку после **y←x** переменные **y** и **x** указывают на один объект.

11. Указатель может иметь специальное значение ***nil***.

12. Параметры передаются функциям по значению: вызванная функция получает собственную копию параметров. Изменение параметра внутри функции снаружи не видно.

13. При передаче массивов или объектов функция получает указатель на данные.

ПРИЛОЖЕНИЕ Б. Алгоритмы внутренней сортировки

Алгоритм сортировки методом выбора

Selection_Sort (*A, n*)

```
1.  ▷ A — массив
2.  ▷ n — размерность массива
3.  for i ← 1 to n – 1
4.    do imin ← i
5.      for i ← i + 1 to n
6.        do if A[j] < A[imin]
7.          then imin ← j
8.      temp ← A[imin]
9.      A[imin] ← A[i]
10.     A[i] ← temp
```

Алгоритм сортировки методом вставок

Insertion_Sort (*A, n*)

```
1.  ▷ A — массив
2.  ▷ n — размерность массива
3.  for i ← 2 to n
4.    do x ← A[j]
5.      j ← I – 1
6.      while j > 0 and A[j] > x
7.        do A[j + 1] ← A[j]
```

```
8.          j ← j – 1
9.      A[j + 1] ← x
```

Алгоритм обменной сортировки

Bubble_Sort (*A, n*)

```
1.  ▷ A — массив
2.  ▷ n — размерность массива
3.  for i ← 1 to n – 1
4.    do for j ← n down i – 1
5.       do if A[j–1] < A[j]
6.          then  temp ← A[j]
7.                A[j] ← A[j – 1]
8.                A[j – 1] ← temp
```

Алгоритм шейкер-сортировки

Sheker_Sort (*A, n*)

```
1.  ▷ A — массив
2.  ▷ n — размерность массива
3.  l ← 1
4.  r ← n
5.  while l < r
6.    do for  j ← r down l + 1
7.       do if A[j – 1] > A[j]
8.          then temp ← A[j – 1]
9.               A[j – 1] ← A[j]
10.              A[j] ← temp
11.              k ← j
12.    l ← k + 1
13.    for  j ← l to r
14.      do if A[j – 1] > A[j]
15.         then temp ← A[j – 1]
```

16. A[j – 1] ← A[j]
17. A[j] ← temp
18. k ← j
19. r ← k – 1

Алгоритм сортировки Шелла

Shell_Sort (*A*, *n*)

1. ▷ A — массив
2. ▷ n — размерность массива
3. h ← 1
4. **while** h ≤ (n – 1)/9
5. **do** h ← 3h + 1
6. **while** h > 0
7. **do** for i ← h **to** n
8. **do** j ← i
9. temp ← A[i]
10. **while** j ≥ h **and** temp < A[j – h]
11. **do** A[j] ←A[j – h]
12. j ← j – h
13. A[j] ← temp
14. h = h/3

Алгоритм пирамидальной сортировки

Heap_Sort (*A*, *n*)

1. ▷ A — массив
2. ▷ n — размерность массива
3. l = [n / 2] + 1
4. r ← n
5. ▷ Формирование пирамиды
6. **while** l >1
7. **do** l ← l –1

8. **Shift** (***A***, ***l***, ***r***)

9. ▷ Сортировка

10. **while** ***r*** > 1

11. **do** x ← *A*[1]

12. ***A***[1] ← *A*[***r***]

13. ***A***[***r***] ← x

14. ***r*** ← ***r*** – 1

15. **Shift** (A, 1, ***r***)

Shift (*A, l, r)*

1. ▷ просеивание в пирамиду

2. i ← ***l***

3. j ← 2i

4. x ← A[i]

5. **if** j < ***r*** **and** A[j + 1] < A[j]

6. **then** j ← j + 1

7. **while** j ≤ ***r*** **and** A[j]<x

8. **do** ***A***[i] ← ***A***[j]

9. i ← j.

10. j ← 2i

11. if j < ***r*** **and** ***A***[j + 1] < ***A***[j]

12. **then** j ← j + 1

13. ***A***[i] ← x

Рекурсивный алгоритм сортировки разделением

QSort (*A, l, r*)

1. ▷ ***A*** — массив

2. ***l, r*** — левая и правая границы части

3. **if** ***l*** < ***r***

4. **then** k ← Partition (***A***, ***l, r***)

5. **QSort** (***A***, ***l***, k)

6. **QSort** (***A***, k + 1, ***r***)

Partition (A, *l, r*)

1. x ← ***A*** [(***l*** + ***r***)]/2
2. i← ***l*** –1
3. j← ***r*** + 1
4. **while** TRUE
5. **do** **repeat** j ← j –1
6. **until** ***A***[j] ≤ x
7. **repeat** i ← i + 1
8. **until** ***A***[i] ≥ x
9. **if** i ≤ j
10. **then** t ← ***A***[i]
11. ***A***[i] ← ***A***[j]
12. ***A***[j] ← t
13. else return j

Итеративный алгоритм сортировки разделением

Iterative_QSort (*A, n*)

1. ▷ ***A*** — массив
2. ▷ ***n*** — размерность массива
3. ▷ stack[1..2***n***] — массив для стека
4. stack[1] ← 1
5. stack[2] ← ***n***
6. top ← 1
7. **repeat**
8. ***l*** ← stack[top]
9. ***r*** ← stack[top + 1]
10. top ← top – 2

```
11.      if l < r
12.       then
13.     repeat
14.          i ← l – 1
15.          j ← r + 1
16.          x ← A (l + r)/2
17.          repeat
18.                repeat i ← i + 1
19.                until A[i] ≥ x
20.                repeat j ← j – 1
21.                until A[j] ≤
22.                if i ≤ j
23.                 then
24.                   t ← A[i]
25.                        A[i] ← A[j]
26.                        A[j] ← t
27.                until i ≥ j
28.                if i – l > r – i
29.                      then  top ← top + 2
30.                            stack[top] ← l
31.                            stack[top + 1] ← i – 1
32.                            l ← i + 1
33.                      else
34.                            top ← top + 2
35.                            stack[top] ← i + 1
36.                            stack[top + 1] ← r
37.                            r ← i – 1
38.               until l ≥ r
39.   until top = –1
```

Рекурсивный алгоритм сортировки слиянием

Merge_Sort (*A, l, r, C*)

1. ▷ ***A*** — массив
2. ▷ ***C*** — вспомогательный массив
3. ▷ ***l***, ***r*** — левая и правая границы части
4. **if** $r \le l$
5. **then** return
6. $m \leftarrow [(l + r)]/2$
7. **Merge_Sort** (*A*, *l*, *m*, *C*)
8. **Merge_Sort** (*A*, *m* + 1, *r*, *C*)
9. **Merge** (*A*, *l*, *m*, *r*, *C*)

Merge (*A*, *l*, *m*, *r*, *C*)

1. **for** i ← *m* + 1 **down** *l* + 1
2. **do** *C*[I – 1] ← *A*[I – 1]
3. **for** j ← *m* **to** *r* –1
4. **do** *C*[*r* + *m* – j] ←*A*[j + 1]
5. **for** k ← *l* **to** *r*
6. **do if** *C*[j] < *C*[i]
7. **then** *A*[k] ← *C*[j]
8. j ← j – 1
9. **else** *A*[k] ← *C*[i]
10. i ← i + 1

Итеративный алгоритм сортировки слиянием

Iterative_Merge_Sort (*A, n, C*)

1. ▷ ***A*** — массив
2. ▷ ***n*** — размерность массива
3. ▷ ***C*** — вспомогательный массив
4. **for** h ← 1 **to** *n* – 1

```
5.  do for i ← 1 to n – h
6.      do r ← i + h + h – 1
7.          if r >n then r ← n
8.          Merge (A, i, i + h – 1, r, C)
9.          i ← i + h + h
10.     h ← h + h
```

Рекурсивный алгоритм поразрядной MSD-сортировки

MSD_Sort (*A, l, r, d, C*)

```
1.  ▷ A — массив
2.  ▷ C — вспомогательный массив для «карманов»
3.  ▷ l, r — левая и правая границы части массива A
4.  ▷ R — основание системы счисления для поразрядной сортировки
5.  ▷ k — максимальное количество цифр в значениях
6.  ▷ d  — номер текущей цифры
7.  ▷ M — пороговый размер кармана для окончания рекурсии
8.  k ← bitsword / bitsdigit ▷ bitsword — число битов в ячейке массива A
9.                           ▷ bitsdigit — число битов в цифре
10. if d > k
11.   then return
12. if r – l ≤ M
13.   then Insertion_Sort(A + l, r – l)
14.        return
15. ▷ вычисление размеров «карманов»
16. for j ← 1 to R + 1
17.   do count[j] ← 0
18. for i ← l to r
19.   do j ← digit(A[i], d, R)     ▻ digit(…) — извлечение d-й цифры
20.     count[j + 1] ← count[j + 1] + 1
21. for j ← 2 to R
```

```
22.  do count[j] ← count[j] + count[j – 1]
23. ▻ распределение значений по «карманам»
24. for j ← l to r
25.  do j ← digit(A[i], d, R)
26.    C[l + count[j]] ← A[i]
27.    count[j] ← count[j] + 1
28. ▹ перенос значений из «карманов» в массив A
29. for i ← l to r
30.  do A[i] ← C[i]
31. MSD_Sort (A, l, l + count[1], d + 1, C)
32. for i ← 2 to R
33.  do MSD_Sort (A, l + count[i], l + count[i + 1] – 1, d + 1, C)
```

Итеративный алгоритм поразрядной LSD-сортировки

```
LSD_Sort (A, n, C)
1.  ▹ A — массив
2.  ▹ C — вспомогательный массив для «карманов»
3.  ▹ R — основание системы счисления для поразрядной сортировки
4.  ▹ k — максимальное количество цифр в значениях
5.  ▹ d — номер текущей цифры
6.  k ← bitsword / bitsdigit      ▹ bitsword — число битов в ячейке массива A
7.                                ▹ bitsdigit — число битов в цифре
8.  for d ← k down 0
9.  do
10.   ▹ — вычисление размеров «карманов»
11.   for j ← 1 to R + 1
12.    do count[j] ← 0
13.   for i ← 1 to n
14.    do j ← digit(A[i], d, R) ▹ digit(…) — извлечение d-й цифры
15.      count[j + 1] ← count[j + 1] + 1
```

16. **for** j ← 2 **to** ***R***
17. **do** count[j] ← count[j] + count[j –1]
18. ▷ распределение значений по «карманам»
19. **for** j ← 1 **to** ***n***
20. **do** j ← digit(***A***[i], ***d***, ***R***)
21. ***C***[count[j]] ← ***A***[i]
22. count[j] ← count[j] + 1
23. ▷ перенос значений из «карманов» в массив ***A***
24. **for** i ← 1 **to** ***n***
25. **do** ***A***[i] ← ***C***[i]

ПРИЛОЖЕНИЕ В. Алгоритмы для BST-дерева

Рекурсивный алгоритм обхода BST-дерева по схеме $t \rightarrow L_t \rightarrow R_t$

1. ▷ ***t*** – корень дерева/поддерева
2. **if** ***t*** = ***nil*** **then return**
3. операция над узлом ***t***
4. **Lt_ Rt _t** (left[***t***])
5. **Lt_ Rt _t** (right[***t***])

Итеративный алгоритм обхода BST-дерева по схеме $t \rightarrow L_t \rightarrow R_t$

t_Lt_ Rt (*root*)

1. ▷ ***root*** — корень дерева
2. ▷ S — вспомогательный стек
3. **Push**(S, ***root***)
4. **while** S не пуст
5. **do** ***t*** ← **Pop**(S)
6. операция над узлом ***t***
7. **if** right[***t***] ≠ ***nil***
8. **then Push**(S, right[***t***])

9. **if** left[***t***] ≠ ***nil***
10. **then Push**(S, left[***t***])

Рекурсивный алгоритм обхода BST-дерева по схеме $L_t \rightarrow R_t \rightarrow t$

Lt_ Rt _t (*t*)

1. ▷ ***t*** — корень дерева/поддерева
2. **if** ***t*** = ***nil*** **then return**
3. **Lt_ Rt _t** (left[***t***])
4. **Lt_ Rt _t** (right[***t***])
5. операция над узлом ***t***

Алгоритм обхода BST-дерева по схеме $L_t \rightarrow t \rightarrow R_t$

Lt_t_ Rt (*t*)

1. ▷ ***t*** — корень дерева/поддерева
2. **if** ***t*** = ***nil*** **then return**
3. **Lt_t_ Rt** (left[***t***])
4. операция над узлом ***t***
5. **Lt_t_ Rt** (right[***t***])

Алгоритм обхода BST-дерева по схеме по уровням

Traverse (*t*)

1. ▷ ***t*** — корень дерева
2. ▷ Q — вспомогательная очередь
3. **Enqueue**(Q, ***t***)
4. **while Empty**(Q) = FALSE
5. **do** ***t*** ← **Dequeue**(Q)
6. операция над узлом ***t***
7. **if** left[***t***] ≠ ***nil***
8. **then Enqueue** (Q, left[***t***])

9. **if** right[*t*] ≠ ***nil***
10. **then Enqueue** (Q, right[*t*])

Рекурсивный алгоритм поиска элемента в BST-дереве

BST_Search (*t, k*)

1. ▷ ***t*** — корень дерева или поддерева
2. ▷ ***k*** — значение искомого ключа
3. **if *t* = *nil***
4. **then** сообщение об ошибке
5. **if *k*** = key[*t*]
6. **then return** data[*t*]
7. **if *k*** < key[t]
8. **then return BST_Search**(left[*t*], ***k***)
9. **else return BST_Search**(right[*t*], ***k***)

Итеративный алгоритм поиска элемента в BST-дереве

BST_Iterative_Search (*t*, *k*)

1. ▷ ***t*** — корень дерева
2. ▷ ***k*** — значение искомого ключа
3. **while *t* ≠ *nil* and *k*** ≠ key[t]
4. **do if *k*** < key[t]
5. **then *t*** ← left[*t*]
6. **else *t*** ← right[*t*]
7. **if *t* = *nil***
8. **then** сообщение об ошибке
9. **return** data[*t*]

Рекурсивный алгоритм вставки элемента в BST-дерево

BST_Insert (*t, k, data, inserted*)

1. ▷ ***t*** — корень дерева или поддерева
2. ▷ ***k*** — ключ элемента
3. ▷ ***data*** — данные элемента

```
4.  ▷ inserted — возвращаемый признак вставки
5.  if t = nil
6.     then inserted ← TRUE
7.          return Create_Node(k, data)
8.  if k = key[t]
9.     then inserted ← FALSE
10.         return t
11. if k < key[t]
12.    then left[t] ← BST_Insert(left[t], k, data, ins)
13.    else right[t] ← BST_Insert(right[t], k, data, ins)
14. inserted ← ins
15. return t
```

Итеративный алгоритм вставки элемента в BST-дерево

BST_Iterative_Insert (*root, k, data*)

```
1.  ▷ root — корень дерева
2.  ▷ k — ключ элемента
3.  ▷ data — данные элемента
4.  if root = nil
5.     then root ← Create_Node(k, data)
6.          return TRUE
7.  t ← root
8.  while t ≠ nil
9.     do
10.    pred ← t
11.    if k = key[t]
12.       then return FALSE
13.    if k < key[t]
14.       then t ← left[t]
15.       else t ← right[t]
```

16. **if** ***k*** < key[pred]
17. **then** left[pred] ← **Create_Node**(***k***, ***data***)
18. **else** right[pred] ← **Create_Node**(***k***,***data***)
19. **return** TRUE

Рекурсивный алгоритм удаления элемента из BST-дерева

BST_Delete (*t, k, deleted*)

1. ▷ ***t*** — корень дерева или поддерева
2. ▷ ***k*** — значение удаляемого ключа
3. ▷ ***deleted*** — возвращаемый признак
4. **if** ***t*** = ***nil***
5. **then** ***deleted*** ← FALSE
6. **return** ***t***
7. **if** ***k*** < key[***t***]
8. **then** left[***t***] ← **BST_Delete**(left[***t***], ***k,*** del)
9. ***deleted*** ← del
10. **return** ***t***
11. **if** ***k*** > key[***t***]
12. **then** right[***t***] ← **BST_Delete**(right[***t***], ***k***, del)
13. ***deleted*** ← del
14. **return** ***t***
15. ***deleted*** ← TRUE
16. **if** left[***t***] = ***nil*** **and** right[***t***] = ***nil***
17. **then Delete_Node** (***t)***
18. **return** ***nil***
19. **if** left[***t***] **=** ***nil***
20. **then** x ← right[***t***]
21. **Delete_Node** (***t)***
22. **return** x
23. **if** right[***t***] = ***nil***

24. **then** x ← left[***t***]
25. **Delete_Node** (***t***)
26. **return** x
27. right[***t***] ← **Del**(right[***t***], ***t***)
28. **return** ***t***

Del (*t*, t_0)

1. **if** left[***t***] ≠ ***nil***
2. **then** left[***t***] ← **Del**(left[***t***], t_0)
3. **return** ***t***
4. key[t_0] ← key[***t***]
5. data[t_0] ← data[***t***]
6. x ← right[***t***]
7. **Delete_Node (*t*)**
8. **return** x

Итеративный алгоритм удаления элемента из BST-дерева

BST_Iterative_Delete (*root, k*)

1. ▷ ***root*** — корень дерева
2. ▷ ***k*** — ключ удаляемого элемента
3. t ← ***root***
4. **while** t ≠ ***nil*** **and** [key]t ≠ ***k***
5. **do**
6. pred ← t
7. **if** ***k*** < key[t]
8. **then** t ← left[t]
9. **else** t ← right[t]
10. **if** t = ***nil***
11. **then return** FALSE
12. pred← ***nil***

```
13. if left[t]= nil and right[t] = nil
14.  then x ← nil
15.  else if left[t] = nil
16.       then x ← right[t]
17.       else if right[t] = nil
18.            then x ← left[t]
19.            else pred ← t
20.                 y ← right[t]
21.                 while left[y] ≠ nil
22.                    do pred ← y
23.                       y ← left[y]
24.                 key[t] ← key[y]
25.                 data[t] ← data[y]
26.                 x ← right[y]
27.                 t ← y
28. if pred = nil
29. then root ← x
30. else
31.    if key[t] < key[pred]
32.      then left[pred] ← x
33.      else right[t] ← x
34. Delete_Node (t)
35. return TRUE
```

Алгоритм вставки элемента в корень BST-дерева

BST_Root_Insert (*t, k, data, inserted*)

1. ▷ ***t*** — корень дерева или поддерева
2. ▷ ***k*** — ключ нового элемента
3. ▷ ***data*** — данные нового элемента
4. ▷ ***inserted*** — возвращаемый признак вставки

```
5.  if t = nil
6.    then inserted ← TRUE
7.         return Create_Node(k, data)
8.  if k = key[t]
9.    then inserted ← FALSE
10.        return t
11. if k < key[t]
12.   then left[t] ← BST_Root_Insert(left[t], k, data, ins)
13.        inserted ← ins
14.        if ins = TRUE
15.          then return R(t)
16.          else return t
17.   else  right[t]←BST_Root_Insert(right[t], k, data, ins)
18.        inserted ← ins
19.        if ins = TRUE
20.          then return L(t)
21.          else return t
```

L (*t*)

```
1.  if t = nil
2.    then return nil
3.    else
4.       x ← right[t]
5.       right[t] ← left[x]
6.       left[x] ← t
7.       return x
```

R (*t*)

```
1.  if t = nil
2.    then return nil
```

3. **else**
4. x ← left[***t***]
5. left[***t***] ← right[x]
6. right[x] ← ***t***
7. **return** x

Алгоритм поиска предыдущего по ключу узла BST-дерева

BST_Predecessor (*root, x*)

1. ▷ ***root*** — корень дерева
2. ▷ ***x*** — узел с заданным ключом
3. **if** left[***x***] ≠ ***nil***
4. **then return Max**(left[***x***])
5. **else return R_Parent** (***root, x***)

Max (*t*)

1. **if** ***t*** = ***nil***
2. **then return** ***nil***
3. **while** right[***t***] ≠ ***nil***
4. **do** ***t*** ← right[***t***]
5. **return** ***t***

R_Parent (*t, x*)

1. ▷ ***t*** — корень дерева/поддерева
2. ▷ ***x*** — узел с заданным ключом
3. **if** ***t*** = ***x*** **then return** ***nil***
4. **if** key[***x***] > key[***t***]
5. **then** rp ← **R_Parent**(right[***t***],***x***)
6. **if** rp ≠ ***nil*** **then return** rp
7. **else return** ***t***
8. **else return R_Parent**(left[***t***], ***x***)

Алгоритм поиска *k*-го по значению ключа узла в BST-дереве

BST_Selectk (*t, k*)

1. ▷ ***t*** — корень дерева/поддерева
2. ▷ ***k*** – порядковый номер узла в BST-дереве
3. **if** ***t*** = ***nil*** **then** return ***t***
4. **if** left[***t***] = ***nil***
5. **then** m ← 0
6. **else** m ← n[left[***t***]]
7. **if** m > ***k***
8. **then return BST_Selectk**(left[***t***], ***k***)
9. **if** m < ***k***
10. **then return BST_Selectk**(right[***t***],***k***–m–1)
11. **return** ***t***

Алгоритм разбиения дерева на части

BST_Part (***root, k***)

1. ▷ ***root*** — корень дерева
2. ▷ ***k*** — порядковый номер узла в BST-дереве
3. x ← **BST_Selectk(*root, k*)**
4. **if** x = ***nil***
5. **then return** FALSE
6. ***root*** ← **Partition(*root*, x)**
7. **return** ***root***

Partition (***t, x***)

1. if ***t*** = ***x*** **then return** ***t***
2. if key[***x***] < key[***t***]
3. **then** left[***t***] ← **Partition**(left[***t***], ***x***)
4. **return Rn**[***t***]

5. **else** right[*t*] ← **Partition**(right[*t*], *x*)
6. **return Ln**[t]

Rn[*t*]

1. if ***t* = *nil***
2. **then return *nil***
3. **else**
4. x ← left[t]
5. left[t] ← right[x]
6. right[x] ← t
7. **Calc_n**(***t***)
8. **Calc_n**(x)
9. **return** x

Calc_n (*t*)

1. if t ≠ ***nil***
2. **then**
3. **if** left[*t*] ≠ ***nil***
4. **then** n[*t*] ← 1 + n[left[*t*]]
5. **else** n[*t*] ← 1
6. **if** right[*t*] ≠ ***nil***
7. **then** n[*t*] ← n[*t*] + n[right [*t*]]
8. **Return**

Алгоритм удаления из BST-дерева на основе объединения поддеревьев

BST_Delete_Join (*t, k, deleted*)

1. ▷ ***t*** — корень дерева или поддерева
2. ▷ ***k*** — значение удаляемого ключа
3. ▷ ***deleted*** — возвращаемый признак удаления
4. **if *t* = *nil***

```
5.    then deleted ← FALSE
6.        return t
7.  if k < key[t]
8.    then left[t] ← BST_Delete_Join(left[t], k, del)
9.        Calc_n(t)
10.       deleted ← del
11.       return t
12. if k > key[t]
13.   then right[t] ← BST_Delete_Join(right[t], k, del)
14.       Calc_n(t)
15.       deleted ← del
16.       return t
17. deleted ← TRUE
18. x ← JoinLR(left[t],right[t])
19. Delete_Node (t)
20. return x
```

JoinLR (*tl,tr*)

```
1.  ▷ tl,tr — корни поддеревьев
2.  if tr = nil then return tl
3.  tr ← BST_Part(tr, 0)
4.  left[tr] ← tl
5.  Calc_n(tr)
6.  return tr
```

Алгоритм объединения BST-деревьев

BST_Join (*a,b*)

```
1.  ▷ a, b — корни деревьев/поддеревьев
2.  if a = nil then return b
3.  if b = nil then return a
```

4. left ← left[***a***]
5. right ← right[***a***]
6. left[***a***] ← ***nil***
7. right[***a***] ← ***nil***
8. **if** key[k] = key[b]
9. **then** Delete_Node (a)
10. **return** b
11. **if** key[***a***] = key[***b***]
12. **then Delete_Node**(***a***)
13. **return** ***b***
14. ***b*** ← **Root_Insert**(***b, a***)
15. left[***b***] ← **BST_Join**(left, left[***b***])
16. right[***b***] ← **BST_Join**(right, right[***b***])
17. **return** ***b***

Алгоритм вертикальной печати BST-дерева

BST_Show (*t, level*)

1. ▷ ***t*** — корень дерева/поддерева
2. ▷ ***level*** — глубина узла ***t***
3. **if** ***t*** = ***nil***
4. **then return**
5. **Show**(right[***t***], ***level***+1)
6. **for** i← 0 **to** 3 × ***level***
7. **do** печать пробела
8. печать key[***t***]
9. перевод курсора на начало следующей строки
10. **Show**(left[***t***], ***level***+1)

ПРИЛОЖЕНИЕ Г. Тесты трудоёмкости операций BST-дерева

```
#include <time.h>
#include <math.h>
#include <iostream>
#include "bst.h"
using namespace std;
typedef unsigned long long INT_64;
```

Генератор случайных чисел большой разрядности

```
//переменная и константы генератора LineRand()
static INT_64 RRand=15750 const INT_64 mRand =(1<<63) -1;
const INT_64 aRand=6364136223846793005;
const INT_64 cRand=1442695040888963407;

//функция установки первого случайного числа от часов компьютера
void sRand () { srand(time(0)); RRand=(INT_64)rand();}

//функция генерации случайного числа
//линейный конгруэнтный генератор  Xi+1=(a*Xi+c)%m
//habr.com/ru/post/208178
INT_64 LineRand ()
{
 INT_64 y1,y2;
 y1= (aRand*RRand+cRand)%mRand;
 y2= (aRand*y1+cRand)%mRand;
 RRand = y1 & 0xFFFFFFFF00000000LL ^ (y2 &
0xFFFFFFFF00000000LL)>>32;
 return RRand;
}
```

Тест трудоёмкости операций случайного BST-дерева

```
void test_rand(int n)
{
 //создание дерева для 64-разрядных ключей типа INT_64
 BST< INT_64,int > tree;
```

```
 //массив для ключей, которые присутствуют в дереве
 INT_64* m=new INT_64 [n];
//установка первого случайного числа
 sRand();
//заполнение дерева и массива элементами со случайными ключами
 for(int i=0; i<n; i++)
  {
  m[i]=LineRand();
  tree.add(m[i],1);
  }
//вывод размера дерева до теста
 cout<<"items count:"<<tree.Size()<<endl;
//обнуление счётчиков трудоёмкости вставки, удаления и поиска
 double I=0;
 double D=0;
 double S=0;
//генерация потока операций, 10% — промахи операций
 for(int i=0;i<n/2;i++)
  if(i%10==0)       //10% промахов
  {
  tree.remove(LineRand ());
  D+=tree.CountNodes();
  tree.add(m[rand()%n],1);
  I+=tree.CountNodes();
  try{
    tree.getItem(LineRand ());
    S+=tree.CountNodes();
    }
  //обработка исключения при ошибке операции поиска
  catch(int){S+=tree.CountNodes();}}
  else //90% успешных операций
  {
  int ind=rand()%n;
  tree.remove(m[ind]);
  D+=tree.CountNodes();
  INT_64 key=LineRand ();
```

```
  tree.add(key,1);
  I+=tree.CountNodes();
  m[ind]=key;
  try{
    tree.getItem(m[rand()%n]);
    S+=tree.CountNodes();
    }
  //обработка исключения при ошибке операции поиска
  catch(int){S+=tree.CountNodes();}
  }  //конец теста

//вывод результатов:
//вывод размера дерева после теста
 cout<<"items count:"<<tree.Size()<<endl;
//теоретической оценки трудоёмкости операций BST
 cout<<"1.39*log2(n)="<<1.39*(log((double)n)/log(2.0))<<endl;
//экспериментальной оценки трудоёмкости вставки
 cout<<"Count insert: " << I/(n/2) <<endl;
//экспериментальной оценки трудоёмкости удаления
 cout<<"Count delete: " << D/(n/2) <<endl;
//экспериментальной оценки трудоёмкости поиска
 cout<<"Count search: " << S/(n/2) <<endl;
//освобождение памяти массива m[]
 delete[] m;
}
```

Тест трудоёмкости операций вырожденного BST-дерева

```
void test_ord(int n)
{
 //создание дерева для 64-разрядных ключей типа INT_64
 BST< INT_64,int > tree;
 //массив для ключей, которые присутствуют в дереве
 INT_64* m=new INT_64 [n];
```

```
//заполнение дерева и массива элементами с возрастающими чётными
//ключами на интервале [0, 10000, 20000, ... ,10000*n]
 for(int i=0; i<n; i++) {
  m[i]= i*10000;
  tree.add(m[i],1);
  }
//вывод размера дерева до теста
 cout<<"items count:"<<tree.Size()<<endl;
//обнуление счётчиков трудоёмкости вставки, удаления и поиска
 double I=0;
 double D=0;
 double S=0;
//установка первого случайного числа
 sRand ();
//генерация потока операций, 10% — промахи операций
 for(int i=0;i<n/2;i++)
  {
          if(i%10==0)         //10% промахов
            {int k=LineRand()%(10000*n);
               k=k+!(k%2);    //случайный нечётный ключ
               tree.remove(k);
               D+=tree.CountNodes();
               tree.add(m[rand()%n],1);
               I+=tree.CountNodes();
               k=LineRand()%(10000*n);
               k=k+!(k%2);    // случайный нечётный ключ
               try{
                    tree.getItem(k);
                    S+=tree.CountNodes();
                 }
               //обработка исключения при ошибке операции поиска
               catch(int){S+=tree.CountNodes();}
             }
          else //90% успешных операций
          {
               int ind=rand()%n;
```

```
                tree.remove(m[ind]);
                D+=tree.CountNodes();;
                int k=LineRand()%(10000*n);
                k=k+k%2;        // случайный чётный ключ
                tree.add(k,1);
                I+=tree.CountNodes();;
                m[ind]=k;
                try{
                     tree.getItem(m[rand()%n]);
                     S+=tree.CountNodes();;
                  }
                //обработка исключения при ошибке операции поиска
                catch(int){S+=tree.CountNodes();}
          }
     }
//вывод результатов:
// вывод размера дерева после теста
 cout<<"items count:"<<tree.Size()<<endl;
//теоретической оценки трудоёмкости операций BST
 cout<<"n/2 ="<<n/2<<endl;
//экспериментальной оценки трудоёмкости вставки
 cout<<"Count insert: " << I/(n/2) <<endl;
//экспериментальной оценки трудоёмкости удаления
 cout<<"Count delete: " << D/(n/2) <<endl;
//экспериментальной оценки трудоёмкости поиска
 cout<<"Count search: " << S/(n/2) <<endl;
//освобождение памяти массива m[]
 delete[] m;
}    //конец теста
```

ПРИЛОЖЕНИЕ Д. Алгоритмы для сбалансированных деревьев

Алгоритм вставки элемента в рандомизированное дерево

RND_Insert (*t, k, data, inserted*)

```
1.  ▷ t — корень дерева или поддерева
2.  ▷ k — ключ нового элемента
3.  ▷ data — данные нового элемента
4.  ▷ inserted — возвращаемый признак вставки
5.  if t = nil
6.    then t ← Create_Node(k, data)
7.         n[t] ← 1
8.         inserted ← TRUE
9.         return t
10. if Rand() < RAND_MAX/(n[t]+1)
11.   then t ← Root_Insert(t, k, data, ins)
12.        inserted ← ins
13.        return t
14.if k = key[t]
15.   then inserted ← FALSE
16.      return t
17. if k < key[t]
18.   then left[t] ← RND_Insert (left[t], k, data, ins)
19.   else right[t] ← RND_Insert (right[t], k, data, ins)
20. inserted ← ins
21. if inserted = TRUE
22.   then n[t] ← n[t] + 1
23. return t
```

Алгоритм удаления элемента из рандомизированного дерева

RND_Delete (*t, k, deleted*)

```
1.  if t = nil
```

```
2.    then deleted ← FALSE
3.       return t
4.  if k < key[t]
5.    then left[t] ← RND_Delete (left[t], k, del)
6.    else if k > key[t]
7.         then right[t] ← RND_Delete (right[t], k, del)
8.         else x ← RND_Join (left[t], right[t])
9.              Delete_Node (t)
10.             t ← x
11.             del ← TRUE
12. deleted ← del
13. if deleted = TRUE
14.   then Calc_n(t)
15. return t
```

RND_Join (*a, b*)

```
1.  if a = nil
2.    then return b
3.  if b = nil
4.    then return a
5.  if(rand()/(RAND_MAX/(n[a] + n[b] + 1))<n[a]
6.  then right[a] ← RND_Join (right[a], b)
7.     return a
8.  else left[b] ← RND_Join (a, left[b])
9.     return b
```

Рекурсивный алгоритм вставки элемента в AVL-дерево

AVL_Insert (*t*, *k*, *data*, *h*, *inserted*)

1. ▷ ***t*** — корень дерева / поддерева
2. ▷ ***k*** — ключ нового элемента

```
3.  ▷ data — данные нового элемента
4.  ▷ inserted — возвращаемый признак вставки
5.  ▷ h — возвращаемый признак увеличения высоты
6.  if1 t = nil
7.   then1 t ← Create_Node(k, data)
8.        bal[t] ← 0
9.        h ← TRUE
10.       inserted ← TRUE
11.      return t
12. h ← FALSE
13. if2 k = key[t]
14.  then2 inserted ← FALSE
15.       return t
16. if3 k < key[t]
17.  then3 left[t] ← AVL_Insert(left[t], k, data, hh, ins)
18.     if4 hh = TRUE
19.       then4 ▷ выросла левая ветвь
20.          if5 bal[t] = 1
21.            then5 bal[t] ← 0
22.            else5 if6 bal[t] = 0
23.                 then6 bal[t] ← –1
24.                      h ← TRUE
25.                      ▷ балансировка
26.                 else6 if7 bal[left[t]] = –1
27.                      then7 t ← R (t)
28.                      else7 t ← LR (t)
29.  else3 right[t] ← AVL_Insert(right[t], k, data, hh, ins)
30.     if8 hh = TRUE
31.       then8 ▷ выросла правая ветвь
32.          if9 bal[t] = –1
```

```
33.          then9 bal[t] ← 0
34.        else9 if10 bal[t] = 0
35.             then10 bal[t] ← 1
36.                  h ← TRUE
37.                    ▷ балансировка
38.             else10 if11 bal[right[t]] = 1
39.                  then11 t ← L (t)
40.                  else11 t ← RL (t)
41. inserted ← ins
42. return t
```


Рекурсивный алгоритм удаления элемента из AVL-дерева

```
AVL_Delete (t, k, h, deleted)
1. ▷ t — корень дерева / поддерева
2. ▷  k — ключ удаляемого элемента
3. ▷ h — возвращаемый признак уменьшения высоты
4. ▷ deleted — возвращаемый признак удаления
5. h ← FALSE
6. if t = nil
7.  then deleted ← FALSE
8.      return nil
9. if k < key[t]
10.  then left[t] ← AVL_Delete(left[t], k, hh, del)
11.     if hh = TRUE
12.        then if bal[t] = –1
13.             then bal[t] ← 0;
14.                  h ← TRUE
15.           else if bal[t] = 0
16.                then bal[t] ← 1
17.                else if bal[right[t]] = 0
```

```
18.                 then t ← L(t)
19.                 else  h ← TRUE
20.                     if bal[right[t]] = 1
21.                      then t ← L(t)
22.                      else t ← RL(t)
23.     deleted ← del
24.     return t
25. if k > key[t]
26.  then right[t] ← AVL_Delete(right [t], k, hh, del)
27.     if hh = TRUE
28.       then if bal[t] = 1
29.            then bal[t] ← 0;
30.              h ← TRUE
31.            else if bal[t] = 0
32.              then bal[t] ← –1
33.              else if bal[left[t]] = 0
34.                   then t ← R(t)
35.                   else h ← TRUE
36                      if bal[left[t]] = –1
37.                      then t ← R(t)
38.                      else t ← LR(t)
39.     deleted ← del
40.     return t
41. ▷ найден удаляемый узел t
42. deleted ← TRUE
43. if left[t] = nil and right[t] = nil
44.   then Delete_Node (t)
45.       h ← TRUE
46.       return nil
47. if left[t]= nil
```

```
48.   then x ← right[t]
49.        Delete_Node (t)
50.        h ← TRUE
51.        return x
52. if right[t] = nil
53.   then x ← left[t]
54.        Delete_Node (t)
55.        h ← TRUE
56.        return x
57. right[t] ← Del (right[t], t, hh)
58. if hh = TRUE
59.   then if bal[t] = 1
60.         then bal[t] ← 0
61.              h ← TRUE
62.         else if bal[t] = 0
63.              then bal[t] ← –1
64.              else x ← left[t]
65.                   if bal[x] = 0
66.                   then t ← R(t)
67.                   else h ← TRUE
68                         if bal[x] = –1
69.                        then t ← R(t)
70.                        else t ← LR(t)
71. return t
```

Del(*t, t0, h*)

```
1. h ← FALSE
2. if left[t] ≠ nil
3.   then left[t] ← Del (left[t], t0,hh)
4.        if hh = TRUE
```

```
5.       then if bal[t] = –1
6.            then bal[t] ← 0
7.              h ← TRUE
8.            else if bal[t] = 0
9.                 then bal[t] ← 1
10.                else if bal[right[t]] = 0
11.                     then t ← L(t)
12.                     else h ← TRUE
13.                       if bal[right[t]] = 1
14.                         then t ← L(t)
15.                         else t ← RL (t)
16.     return t
17.  else key[t0] ← key[t]
18.    data[t0] ← data[t]
19.    x ← right[t]
20.    Delete_Node (t)
21.    h ← TRUE
22.    return x
```

R (*t*)

```
1.  x← left[t]
2.  left[t] ← right[x]
3.  right[x] ← t
4.  if bal[x] = 1
5.    then bal[t] ← 0
6.       bal[x] ← 0
7.    else bal[t] ← 1
8.       bal[x] ← –1
9.  return x
```

LR (*t*)

1. x ← left [***t***]
2. y ← right [x]
3. right [x] ← left [y]
4. left [y] ← x
5. left [***t***] ← right [y]
6. right [y] ← ***t***
7. **if** bal [y] = –1
8. **then** bal [***t***] ← 1
9. bal [x] ← 0
10. **if** bal [y] = 0
11. **then** bal [***t***] ← bal [x] ← 0
12. **if** bal [y] = +1
13. **then** bal [***t***] ← 0
14. bal [x] ← –1
15. bal [y] ← 0
16. **return** y

Итеративный алгоритм удаления элемента из AVL-дерева

It_AVL_Delete (***root, k***)

1. **if** ***root*** = ***nil***
2. **then return** FALSE
3. t ← ***root***
4. **while** t ≠ ***nil*** **and** key[t] ≠ ***k***
5. **do** Push(St, t) ▷ St – стек
6. **if**(***k*** < key[t]
7. **then** t ← left[t]
8. **else** t ← right[t]
9. **if** t = ***nil***

```
10.   then return FALSE
11. if left[t] = nil and right[t] = nil
12.   then k ← key[t]
13.      Delete_Node (t)
14.      son ← nil
15.   else if left[t] = nil
16.      then k ← key[t]
17.           son ←right[t]
18.           Delete_Node (t)
19.      else if right[t] = nil
20.           then k ← key[t]
21.                son ← left[t]
22.                Delete_Node (t)
23.           else t0 ← t
24.                Push(St, t)
25.                t ← right[t]
26.                while left[t] ≠ nil
27.                  do Push(St, t)
28.                     t ← left[t]
29.                key[t0] ← key[t]
30.                data[t0] ← data[t]
31.                k ← key[t]
32.                son ← right[t]
33.                Delete_Node (t)
34. h ← TRUE
35. if St пуст
36.   then root ← son
37.      return TRUE
38.   else while St не пуст
39.        do t ← Pop(St)
```

```
40.          if k < key[t]
41.            then left[t] ← son
42.                 if h = TRUE
43.                   then if bal[t] = –1
44.                          then bal[t] ← 0
45.                          else if bal[t] = 0
46.                                 then bal[t] ← 1
47.                                      h ← FALSE
48.                                 else x ← right[t]
49.                                      if bal[x] = 0 or bal[x] = 1
50.                                        then t ← L(t)
51.                                             h ← FALSE
52.                                        else t ← RL(t)
53.            else right[t] ← son
54.                 if h = TRUE
55.                   then if bal[t] = 1
56.                          then bal[t] ← 0
57.                          else if bal[t] = 0
58.                                 then bal[t] ← –1
59.                                      h ← FALSE
60.                                 else x ← left[t]
61.                                      if bal[x] = 0 or bal[x] = –1
62.                                        then t ← R(t)
63.                                             h ← FALSE
64.                                        else t ← LR(t)
65.          son ← t
66.          k ← key[son]
67.        ▷ конец цикла
68. root ← son
69. return TRUE
```

Рекурсивный алгоритм вставки элемента в RB-дерево

RB_Insert (*root*, *k*, *data*)

1. ▷ ***root*** — корень RB-дерева
2. ▷ ***k*** — ключ нового элемента
3. ▷ ***data*** — данные нового элемента
4. ***root*** ← **RB_Insert1**(***root***, ***k***, ***data***, 0, ins)
5. color [***root***] ← BLACK
6. **return** ins

RB_Insert1 (*t, k, data, s, inserted*)

1. ▷ ***t*** — корень дерева/поддерева
2. ▷ ***k*** — ключ нового элемента
3. ▷ ***data*** — данные нового элемента
4. ▷ ***s*** — тип родителя (левый — 0/правый — 1)
5. ▷ ***iserted*** — возвращаемый признак вставки
6. ▷ ***tnil*** — фиктивный узел RB-дерева
7. **if** ***k*** = key[***t***]
8. **then** ***inserted*** ← FALSE
9. **return** ***t***
10. **if** ***t*** = ***tnil***
11. **then** ***t*** ← **Create_RB_Node**(***k***, ***data***)
12. color[***t***] ← RED
13. ***inserted*** ← TRUE
14. **return** ***t***
15. **if** color[left[***t***]] = RED **and** color[right[***t***]] = RED
16. **then** color[***t***] ← RED
17. color[left[***t***]] ← color[right[***t***]] ← BLACK
18. **if** ***k*** < key[***t***]
19. **then** left[***t***] ← **RB_Insert1**(left[***t***], ***k***, ***data***, 0, ins)
20. **if** color[***t***] = RED **and** color[left[***t***]] = RED **and** ***s*** = 1

```
21.            then t ← R(t)
22.          if color [left[t]] = RED and color [left[left[t]]] = RED
23.            then t ← R(t)
24.                 color[t] ← BLACK
25.                 color[right[t]] ← RED
26.       else right[t] ← RB_Insert1(right[t], k, data, 1, ins)
27.         if color[t] = RED and color[right[t]] = RED and s = 0
28.           then t ← L(t)
29.         if color[right[t]] = RED and color[right[right[t]]] = RED
30.            then t ← L(t)
31.                 color[t] ← BLACK
32.                 color[left[t]] ← RED
33.    inserted ← ins
34.    return t
```

Итеративный алгоритм вставки элемента в RB-дерево

```
It_RB_Insert (root, k, data)
1.  ▷ root — корень дерева
2.  ▷ k — ключ нового элемента
3.  ▷ data — данные нового элемента
4.  ▷ tnil — фиктивный узел RB-дерева
5.  if root = tnil
6.    then root ← Create_RB_Node(k, data)
7.         color[root] ← BLACK
8.         return TRUE
9.  t ← root
10. while t ≠ tnil
11.   do
12.    pred ← t
13.    if k = key[t]
```

```
14.       then return FALSE
15.    if k < key[t]
16.       then t ← left[t]
17.       else t ← right[t]
18. if k < key[pred]
19.   then t ← left[pred] ← Create_RB_Node(k, data)
20.   else t ← right[pred] ← Create_RB_Node(k,data)
21. color[t] ← RED
22. while t ≠ root and color[p[t]] = RED
23.   do if p[t] = left[p[p[t]]]
24.       then y ← right[p[p[t]]]
25.          if color[y]=RED                    ▷ случай 1
26.            then color[p[t]] ← color[y] ← BLACK
27.                 color[p[p[t]]] ← RED
28.                 t ← p[p[t]]
29.            else if t = right[p[t]]          ▷ случай 2
30.                 then t ← p[t]
31.                      It_L(root, t)
32.                 color[p[t]] ← BLACK ▷ случай 3
33.                 color[p[p[t]]] ← RED
34.                 It_R(root, p[p[t]])
35.       else ▷ симметричный текст (строки 24–34) с заменой left на right
36.   ▷ конец цикла
37. color[root] ← BLACK
38. return TRUE
```

It_L (***root***, ***t***)

```
1.  x ← right[t]
2.  right[t] ← left[x]
3.  if left[x] ≠ tnil
```

4. **then** p[left[x]] ← ***t***
5. p[x] ← p[***t***]
6. **if** p[***t***] = ***tnil***
7. **then** ***root*** ← x
8. **else if** ***t*** = left[p[***t***]]
9. **then** left[p[***t***]] ← x
10. **else** right[p[***t***]] ← x
11. left[x] ← ***t***
12. p[***t***] ← x

Итеративный алгоритм удаления элемента из RB-дерева

It_RB_Delete ***(root, k)***

1. ▷ ***root*** — корень RB-дерева
2. ▷ ***k*** — ключ удаляемого элемента
3. t ← ***root***
4. **while** k ≠ key[t] **and** t ≠ ***tnil*** ▷ ***tnil*** — фиктивный узел RB-дерева
5. **do if** ***k*** < key[t]
6. **then** t ← left[t]
7. **else** t ← right[t]
8. **if** t = ***tnil***
9. **then return** FALSE
10. **if** left[t]= ***tnil*** **or** right[t] = ***tnil***
11. **then** x ← t ▷ x — удаляемый узел
12. **else** x ← BST_Successor(***root,*** t)
13. **if** left[x] ≠ ***tnil***
14. **then** y ← left[x] ▷ y — сын удаляемого узла, возможно ***tnil***
15. **else** y ← right[x]
16. p[y] ← p[x]
17. **if** p[x]=***tnil*** ▷ x — корень

```
18.   then root ← y
19.   else if x =left[p[x]]
20.        then left[p[x]] ← y
21.        else right[p[x]] ← y
22. if x ≠ t    ▷ x — замещающий узел
23.   then key[t] ← key[x]
24.      data[t] ← data[x]
25. if color[x]=BLACK
26.   then RB_Fixup(root, y)
27. Delete_Node(x)
28. return TRUE
```

RB_Fixup (*root*, *y*)

```
1.      ▷ root — корень дерева
2.      ▷ y — узел с двойной чернотой
3.      while y ≠ root and color[y] = BLACK
4.        do if y = left[p[y]]
5.           then w ← right[p[y]]
6.              if color[w] = RED                          ▷ случай 1
7.                 then color[w] ← BLACK
8.                      color[p[y]] ← RED
9.                      It_L(root, p[y])
10.                     w ← right[p[y]]
11.             if color[left[w]] =BLACK and color[right[w]]=BLACK ▷ случай 2
12.                then color[w] ← RED
13.                     y ← p[y]
14.                else if color[left[w]] = RED               ▷ случай 3
15.                      then color[left[w]] ← BLACK
16.                           color[w] ← RED
17.                           It_R(root, w)
```

```
18.                 w ← right[p[y]]
19.               ▷ случай 4
20.               color[w] ← color[p[y]]
21.               color[p[y]] ← BLACK
22.               color[right[w]] ← BLACK
23.               It_L(root, p[y])
24.               y ← root ▷ для прекращения цикла
25.         else
26.           ▷ симметричный фрагмент для y = right[p[y]]с заменой left на right
27.       ▷ конец цикла
28.   color[y] ← BLACK        ▷ если y — красный узел или y — корень
```

Алгоритм вставки элемента в 2-3-дерево

```
T23_Insert (root, k, data)
1. ▷ root — корень 2-3-дерева
2. ▷ k — ключ
3. ▷ data — данные
4. lt ← Create_Leaf (k, data)
5. if root = nil
6.  then root ← Create_Internal
7.      son1[root] ← lt
8.      son2[root] ← son3[root] ← nil
9.      return TRUE
10. if son2[root] = nil
11.  then if key[son1] < k
12.       then son2[root] ← lt
13.            low2[root] ← k
14.            return TRUE
15.       else if key[son1] > k
16.            then son2[root] ← son1[root]
```

```
17.              low2[root] ← key[son1[root]]
18.              son1[root] ← lt
19.              return TRUE
20.           else Delete_Leaf (lt)
21.              return FALSE
22. inserted ← Insert1(root, lt, tbk, lbk)
23. if inserted = FALSE
24.   then Delete_Node(lt)
25.      return FALSE
26. if tbk ≠ nil
27.   then temp ← root
28.      root ← new Internal
29.      son1[root] ← temp
30.      son2[root] ← tbk
31.      low2[root] ← lbk
32.      son3[root] ← nil
33. return TRUE
```

T23_Insert1 (*t*, *lt*, *tup*, *lup*)

```
1. ▷ t — корень 2-3-дерева/2-3-поддерева
2. ▷ lt — новый узел — лист
3. ▷ tup — возвращаемый после расщепления адрес нового узла
4. ▷ lup — возвращаемый минимальный ключ в поддереве нового узла
5. tup ← nil
6. if1 type[t] = LEAF
7.  then1 if2 key[t] = key[lt]
8.       then2 return FALSE
9.       else2 tup ← lt
10.          if3 key[t] < key[lt]
11.            then3 lup ← key[lt]
```

```
12.          else3 lup ← key[t]
13.             temp1 ← key[t]
14.             key[t] ← key[lt]
15.             key[lt] ← temp1
16.             temp2 ← data[t]
17.             data[t] ← data[lt]
18.             data[lt] ← temp2
19.          return TRUE
20. ▷ t — внутренний узел
21. if4 key[lt] < low2[t]
22.  then4 child ← 1
23.       w ← son1[t]
24.  else4 if5 son3[t] = nil or (son3[t] ≠ nil and key[lt] < low3[t])
25.       then5 child ← 2
26.          w ← son2[t]
27.       else5 child ← 3
28.          w ← son3[t]
29. inserted ← T23_Insert1(w, lt, tbk, lbk)
30. if6 inserted = TRUE
31.   then6 if7 tbk ≠ nil
32.        then7 if8 son3[t] = nil ▷ узел имел 2-х сыновей
33.             then8 if9 child = 2
34.                  then9 son3[t] ← tbk
35.                     low3[t] ← lbk
36.                  else9 son3[t] ← son2[t]
37.                     low3[t] ← low2[t]
38.                     son2[t] ← tbk
39.                     low2[t] ← lbk
40.             else8 ▷ узел имел 3-х сыновей
41.                tup ← Create_Internal( )
```

```
42.              if10 child = 3
43.               then10 ▷ выполнялась вставка в 3-е поддерево
44.                   son1[tup] ← son3[t]
45.                   son2[tup] ← tbk
46.                   son3[tup] ← nil
47.                   low2[tup] ← lbk
48.                   son3[t] ← nil
49.                   lup ← low3[t]
50.               else10 ▷ выполнялась вставка в 1-е или 2-е поддерево
51.                   son2[tup] ← son3[t]
52.                   low2[tup] ← low3[t]
53.                   son3[tup] ← nil
54.                   if11 child = 2 ▷ выполнялась вставка во 2-е поддерево
55.                    then11 son1[tup] ← tbk
56.                        lup ← lbk
57.                   if12 child = 1 ▷ выполнялась вставка в 1-е поддерево
58.                    then12 son1[tup] ← son2[t]
59.                        son2[t] ← tbk
60.                        lup ← low2[t]
61.                        low2[t] ← lbk
62.              son3[t] ← nil
63. return inserted
```

Алгоритм удаления элемента из 2-3-дерева

T23_Delete (*root*, *k*)

```
1. ▷ root — корень 2-3-дерева
2. ▷ k — ключ удаляемого элемента
3. if root = nil
4.    then return FALSE
```

```
5.  if son2[root] = nil
6.     then if key[son1[root]] = k
7.          then Delete_Node (son1[root])
8.               Delete_Node (root)
9.               root ← nil
10.              return TRUE
11.         else return FALSE
12. deleted ← T23_Delete1(root, k, tmin, one)
13. if deleted = TRUE
14.    then if one = TRUE
15.         then if son1[root] ≠ LEAF
16.                 then t ← son1[root]
17.                      Delete_Node (root)
18.                      root ← t
19. return deleted
```

T23_Delete1 (*t*, *k*, *tlow1*, *one_son*)

```
1. ▷ t — корень дерева / поддерева
2. ▷ k = ключ удаляемого элемента
3. ▷ tlow1 — возвращаемый адрес узла с минимальным ключом в первом поддереве
4. ▷ one_son — возвращаемый признак узла с одним сыном
5. tlow1 ← nil
6. one_son ← FALSE
7. if₁ type [son1[t]] = LEAF
8.      then₁ if₂ key[son1[t]] = k
9.              then₂ Delete_Node (son1[t])
10.                   son1[t] ← son2[t]
11.                   son2[t] ← son3[t]
12.                   son3[t] ← nil
13.                   low2[t] ← low3[t]
```

```
14.          else2 if3 key[son2[t]] = k
15.               then3 Delete_Node (son2[t])
16.                    son2[t] ← son3[t]
17.                    son3[t] ← nil
18.                    low2[t] ← low3[t]
19.               else3 if4 son3[t] ≠nil & key[son3[t]]=k
20.                    then4 Delete_Node (son3[t])
21.                         son3[t] ← nil
22.                    else4 return FALSE
23.          tlow1 ← son1[t]    ▷ общее завершение вариантов удаления листа
24.          if5 son2[t] = nil
25.               then5 one_son ← TRUE
26.          return TRUE
27. if6 k < low2[t]
28.     then6 child ← 1
29.          w ← son1[t]
30.     else6 if7 son3[t]=nil or k < low3[t]
31.          then7 child ← 2
32.               w ← son2[t]
33.          else7 child ← 3
34.               w ← son3[t]
35. if8 T23_Delete1(w, k, tlow1_bk, one_son_bk)=FALSE
36.     then8 return FALSE
37. ▷ удаление произошло
38. tlow1 ← tlow1_bk
39. if9 tlow1_bk ≠ nil
40.     then9 if10 child = 2
41.               then10 low2[t] ← key[tlow1_bk]
42.               tlow1 ← nil
43.          if11 child = 3
```

```
44.                 then11 low3[t] ← key[tlow1_bk]
45.                     tlow1 ← nil
46. if12 one_son_bk = FALSE
47.     then12 return TRUE
48. if13 child = 1
49.     then13 y ← son2[t]
50.         if14 son3[y] ≠ nil
51.             then14 son2[w] ← son1[y]
52.                 low2[w] ← low2[t]
53.                 low2[t] ← low2[y]
54.                 son1[y] ← son2[y]
55.                 son2[y] ← son3[y]
56.                 low2[y] ← low3[y]
57.                 son3[y] ← nil
58.             else14 ▷ y не имеет 3-х сыновей
59.                 son3[y] ← son2[y]
60.                 low3[y] ← low2[y]
61.                 son2[y] ← son1[y]
62.                 low2[y] ← low2[t]
63.                 son1[y] ← son1[w]
64.                 Delete_Node (w)
65.                 son1[t] ← son2[t]
66.                 son2[t] ← son3[t]
67.                 low2[t] ← low3[t]
68.                 son3[t] ← nil
69.                 if15 son2[t]=nil
70.                     then15 one_son ← TRUE
71.         return TRUE
72. if16 child = 2
73.     then16 y ← son1[t]
```

```
74.          if17 son3[y] ≠ nil
75.               then17 son2[w] ← son1[w]
76.                    low2[w] ← low2[t]
77.                    son1[w] ← son3[y]
78.                    son3[y] ← nil
79.                    low2[t] ← low3[y]
80.                    return TRUE
81.               else17 z ← son3[t]
82.                    if18 z≠nil and son3[z] ≠nil
83.                         then18 son2[w] ← son1[z]
84.                                low2[w] ← low3[t]
85.                                low3[t] ← low2[z]
86.                                son1[z] ← son2[z]
87.                                son2[z] ← son3[z]
88.                                low2[z] ← low3[z]
89.                                son3[z] ← nil
90.                                return TRUE
91.          ▷ y t нет сыновей с тремя узлами
92.          son3[y] ← son1[w]
93.          low3[y] ← low2[t]
94.          Delete_Node (w)
95.          son2[t] ← son3[t]
96.          low2[t] ← low3[t]
97.          son3[t] ← nil
98.          if19 son2[t]=ni
99.               then19 one_son ← TRUE
100.         return TRUE
101. ▷ child=3
102.   y ← son2[t]
103. if20 son3[y] ≠nil
```

104. **then**$_{20}$ son2[w] ← son1[w]

105. low2[w] ← low3[***t***]

106. son1[w] ← son3[y]

107. low2[***t***] ← low3[y]

108. son3[y] ← ***nil***

109. **else**$_{20}$ ▷ y не имеет третьего сына

110. son3[y] ← son1[w]

111. low3[y] ← low3[***t***]

112. son3[***t***] ← ***nil***

113. **Delete_Node** (w)

114. **return** TRUE

ПРИЛОЖЕНИЕ Е. Алгоритмы для хеш-таблицы с открытой адресацией

Алгоритм вставки элемента

Hash_Insert (T, *k, data*)

1. ▷ ***k, data* —** вставляемые ключ и данные
2. ▷ T[{state, key, data}] — массив хеш-таблицы T
3. ▷ state[T[j]] — состояние ячейки таблицы (***free / busy / deleted***)
4. ▷ key[T[j]] — ключ поиска элемента таблицы
5. ▷ data[T[j]] — данные элемента таблицы
6. i← 0
7. pos ← –1
8. **repeat** j ← h(***k***, i)
9. **if** state[T[j]] = ***deleted*** **and** pos = –1
10. **then** pos ← j
11. **if** state[T[j]] = ***busy*** **and** key[T[j]] = ***k***
12. **then return** FALSE
13. i ← i+1
14. **until** i = m **or** state[T[j]] = ***free***

15. **if** i = m **and** pos = –1
16. **then return** FALSE
17. **if** pos = –1
18. **then** pos ← i
19. key[T[pos]] ← ***k***
20. data[T[pos]] ← ***data***
21. state[T[pos]] ← ***busy***
22. **return** TRUE

Алгоритм удаления элемента

Hash_Delete (T, *k*)

1. ▷ ***k*** — удаляемый ключ
2. ▷ T[{state, key, data}] — массив хеш-таблицы T
3. ▷ state[T[j]] — состояние ячейки таблицы (***free / busy / deleted***)
4. ▷ key[T[j]] — ключ поиска элемента таблицы
5. ▷ data[T[j]] — данные элемента таблицы
6. i← 0
7. **repeat** j← h(k, i)
8. **if** state[T[j]] = ***busy*** **and** key[T[j]] = ***k***
9. **then** state[T[j]] ← ***deleted***
10. **return** TRUE
11. I ← i + 1
12. **until** state[T[j]] = ***free*** **или** i = m
13. **return** FALSE

Алгоритм поиска элемента

Hash_Search (T, *k*)

1. ▷ ***k*** — ключ поиска
2. ▷ T[{state, key, data}] — массив хеш-таблицы T
3. ▷ state[T[j]] — состояние ячейки таблицы (***free / busy / deleted***)
4. ▷ key[T[j]] — ключ поиска элемента таблицы

5. ▷ data[T[j]] – данные элемента таблицы
6. i← 0
7. **repeat** j← h(k, i)
8. **if** state[T[j]] = ***busy*** **and** key[T[j]] = ***k***
9. **then return** data[T[j]]
10. i← i + 1
11. **until** state[T[j]] = ***free*** **или** i = m
12. **return** сообщение об ошибке

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Ахо, А. В.* Структуры данных и алгоритмы / А. В. Ахо, Д. Э. Хопкрофт, Дж. Д. Ульман ; [пер. с англ. и ред. А. А. Минько]. — М., 2001. — 382 с.

2. *Вирт, Н.* Алгоритмы и структуры данных : с примерами на Паскале / Н. Вирт ; [пер. с англ. Д. Б. Подшивалова]. — СПб., 2007. — 350 с.

3. *Каррано, Ф. М.* Абстракция данных и решение задач на C++. Стены и зеркала / Ф. М. Каррано, Д. Дж. Причард ; [пер. с англ. и ред. Д. А. Клюшина]. — Москва [и др.], 2003. — 847 с.

4. *Кнут, Д.* Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. — М. : Мир, 1978. — 844 с.

5. *Коллинз, У. Дж.* Структуры данных и стандартная библиотека шаблонов. — М. : ООО «Бином-Пресс», 2004. — 624 с.

6. *Кормен, Т.* Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест ; [пер. с англ. К. Белова и др. ; под ред. А. Шеня]. — М., 2001. — 955 с.

7. *Кубенский, А. А.* Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++. — СПб. : БХВ-Петербург, 2004. — 464 с.

8. *Макконелл, Дж.* Анализ алгоритмов. Вводный курс. — М. : Техносфера, 2002. — 304 с.

9. *Седжвик, Р.* Фундаментальные алгоритмы на C++. Ч. 1–4 : [пер. с англ.] / Р. Седжвик. — М. [и др.], 2002. — 687 с.

ОГЛАВЛЕНИЕ

Татьяна Александровна РОМАНЕНКО

ПРОГРАММНЫЕ КОЛЛЕКЦИИ ДАННЫХ
ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ
Учебник

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*
Ответственный редактор *В. В. Яески*
Корректор *Н. Ю. Наумкина*
Выпускающий *В. А. Невара*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com
196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А
Тел./факс: (812) 336-25-09, 412-92-72
Бесплатный звонок по России: 8-800-700-40-71

ГДЕ КУПИТЬ

ДЛЯ ОРГАНИЗАЦИЙ:

Для того, чтобы заказать необходимые Вам книги, достаточно обратиться в любую из торговых компаний Издательского Дома «ЛАНЬ»:

по России и зарубежью
«ЛАНЬ-ТРЕЙД». 196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А
тел.: (812) 412-85-78, 412-14-45, 412-85-82; тел./факс: (812) 412-54-93
e-mail: trade@lanbook.ru; ICQ: 446-869-967

www.lanbook.com
пункт меню **«Где купить»**
раздел **«Прайс-листы, каталоги»**

в Москве и в Московской области
«ЛАНЬ-ПРЕСС». 109387, Москва, ул. Летняя, д. 6
тел.: (499) 722-72-30, (495) 647-40-77; e-mail: lanpress@lanbook.ru

в Краснодаре и в Краснодарском крае
«ЛАНЬ-ЮГ». 350901, Краснодар, ул. Жлобы, д. 1/1
тел.: (861) 274-10-35; e-mail: lankrd98@mail.ru

ДЛЯ РОЗНИЧНЫХ ПОКУПАТЕЛЕЙ:

интернет-магазин
Издательство «Лань»: http://www.lanbook.com

магазин электронных книг
Global F5: http://globalf5.com/

Подписано в печать 09.07.21.
Бумага офсетная. Гарнитура Школьная. Формат 70×100 $^{1}/_{16}$.
Печать офсетная/цифровая. Усл. п. л. 12,35. Тираж 30 экз.

Заказ № 875-21.

Отпечатано в полном соответствии с качеством
предоставленного оригинал-макета в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.