Михаил Дворкин, Андрей Станкевич, Анатолий Шалыто

О применении автоматов при реализации алгоритмов дискретной математики (на примере АВЛ-деревьев)

Не потому что я лучше других деревьев.
Нет, я этого не говорю.
Просто, я другое дерево.

Из песни «Я другое дерево».
Стихи Г. Поженяна, музыка М. Таривердиева.

Введение

Задачи управления постоянно возникают в программировании. Одним из подходов к решению таких задач является *автоматное программирование* или *программирование с явным выделением состояний*, которое заключается в построении автомата, управляющего поведением объекта управления при применении к нему управляющих воздействий [1]. При этом строится *управляющий автомат*, который имеет прямую и обратную связь с объектом управления и может осуществлять воздействия при попадании в состояние или при переходе из одного состояния в другое.

Опишем, чем отличаются автоматы, рассматриваемые в настоящей работе, от традиционной математической абстракции – конечных автоматов, применяемых в теории компиляции [2].

Детерминированным конечным автоматом называют пятерку $\langle Q, \Sigma, \boldsymbol{d}, S, F \rangle$, где Q – конечное множество состояний, Σ – входной алфавит (также, обычно, конечный), $\boldsymbol{d} : Q \times \Sigma \to Q$ – функция переходов, $S \in Q$ – начальное состояние, а $F \subset S$ – множество допускающих состояний [3]. Определенный таким образом автомат последовательно считывает символы входного алфавита с входной ленты и в соответствии с функцией переходов изменяет свое состояние. Результатом работы автомата является допуск (или, соответственно, недопуск) строки над входным алфавитом, записанной на входной ленте. Дальнейшее развитие идеи конечного автомата приводит к построению более сложных моделей – автоматов с магазинной памятью, преобразующих автоматов (имеющих также выходную ленту) и, наконец, машин Тьюринга. Такие автоматы также называют *абстрактными*.

Автоматы, рассматриваемые в настоящей работе, представляют собой другой тип модели – *структурные* конечные автоматы [1], которые в дальнейшем будем называть *автоматами*. В автоматах, также как и в абстрактных автоматах, имеется конечное множество состояний, начальное состояние и функция переходов. Однако в отличие от абстрактных автоматов, автоматы вместо входной ленты используют конечное число булевых входов, называемых *входными переменными*. Кроме входных переменных автоматы допускают также и другой тип входных воздействий – *события* [4]. Автоматы обычно имеют выходы, обеспечивающие формирование *выходных воздействий* на объект управления.

Автоматы могут получать информацию о состояниях других автоматов, с которыми они *взаимодействуют*. В общем случае взаимодействие автоматов – это вложенность, вызываемость и обмен сообщениями.

Опишем, как автоматы осуществляют управление. Разумеется, управление при программной реализации осуществляет не автомат, а программа, его реализующая. Основу такой программы представляет собой оператор выбора. В языках *C*, *C*++ и *Java* это оператор `switch`, поэтому программирование с явным выделением состояний называют также *switch-технологией* [1]. Каждый из вариантов этого оператора (метка `case`) соответствует одному состоянию автомата.

Исходно автомат находится в начальном состоянии. На каждом шаге происходит анализ входных переменных и событий и в соответствии с их значениями осуществляется переход в новое состояние (которое может совпадать с текущим). При этом на переходе могут выполняться

выходные воздействия, определенные для этого перехода. Выходные воздействия реализуются путем вызова соответствующих функций.

Отметим, что входные переменные в общем случае также являются функциями. Они анализируют состояние объекта управления и/или органов управления. Отличие события от входной переменной состоит в том, что оно является асинхронным, в то время как опрос входных переменных происходит синхронно. События могут обрабатываться различным образом. В настоящей работе произошедшее событие помещается его обработчиком в *очередь событий* и может быть извлечено оттуда автоматом позже, когда он находится в состоянии, из которого возможен переход по этому событию.

Кроме того, каждому состоянию могут быть сопоставлены операции, которые выполняются в данном состоянии. Если в рассматриваемое состояние вложены другие автоматы, то для каждого из них выполняется шаг. Затем вызываются выходные воздействия, определенные для данного состояния.

В настоящей работе используется нотация, введенная в работе [4]. Пример состояния автомата и переходов, которые исходят из этого состояния, приведен на рис. 1.

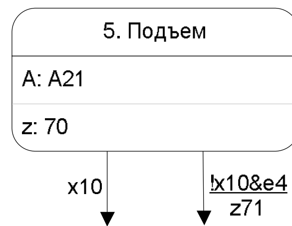

Рис. 1. Нотация состояния автомата

Обозначения для входных переменных начинаются с символа “*x*”, обозначения для выходных воздействий – с символа “*z*”, а обозначения для событий – с символа “*e*”. Для каждого перехода задано *условие перехода* – формула, содержащая входные переменные и/или события. Если переход помечен условием “1”, то переход выполняется всегда (*безусловный переход*). В корректном автомате из состояния, из которого исходит такой переход, не могут исходить другие переходы. Кроме условия для некоторых переходов (под горизонтальной чертой) указывается одно или несколько выходных воздействий.

Состояние автомата обозначается номером состояния, названием, списком из нуля или более вложенных в это состояние автоматов (поле “A:”), а также списком выходных воздействий, выполняемых при попадании в это состояние (поле “z:”).

Изложенная модель соответствует так называемым *смешанным* автоматам [1]. Автоматы, в которых осуществление выходных воздействий возможно только в состоянии называются автоматами Мура, автоматы, в которых осуществление выходных воздействий возможно только на переходах – автоматами Мили.

Итак, процесс программирования с явным выделением состояний происходит следующим образом. Сначала строится автомат, управляющий объектом. Интерфейс автомата задается схемой связей с его «окружением», а его поведение – графом переходов. Затем автомат формально преобразуется в программу, его реализующую, в которой сохраняется структура графа переходов. Также реализуются функции входных переменных и выходных воздействий. Эти два процесса – проектирование и реализация – и рассматриваются в настоящей работе.

Под проектированием в настоящей работе понимается построение управляющего автомата. Ключевым понятием в рассматриваемой технологии является *состояние*. Состояние, в котором находится автомат, определяет его дальнейшее поведение, и, следовательно, тесно связано с логикой программы. Соответственно, наиболее сложный вопрос, который возникает при проектировании, – как определить состояния автомата.

Для решения этой задачи существует несколько возможных подходов. При этом они тесно связаны с формой постановки задачи.

Первый подход заключается в том, что по некоторой программе, написанной на одном из языков программирования, строится автомат, эмулирующий ее поведение. При этом возможно два варианта построения автомата.

Первый вариант – формальное преобразование программы, при котором программные элементы по определенным правилам преобразуются в состояния и переходы автомата (эта процедура подробно рассмотрена в работе [5]). Полученный таким образом автомат будет полностью отражать поведение программы, по которой он построен.

Второй вариант состоит в анализе программы и построении автомата, который отражает поведение не программы, а объектов, управляемых программой (при этом происходит в определенном смысле reengineering программы). Этот способ близок к описанным далее вариантам, применяемым в случаях, когда постановка задачи осуществляется в виде формального или даже неформального (вербального) описания необходимого поведения. Рассмотренный вариант, на первый взгляд, кажется применимым исключительно для случая, когда имеется объект управления, у которого можно явно выделить состояния. Однако это не так. При рассмотрении практически любого алгоритма возможно выделение состояний, если не у объекта управления, то, по крайней мере, у самого алгоритма. Более подробное обсуждение этого вопроса проведем после изучения основного примера, рассматриваемого в этой работе.

Построение автомата *после* написания программы кажется представляющим лишь академический интерес, поскольку программа уже имеется. Однако это не так. Формальный переход от программы к автомату может иметь различные области применения, например, построение визуализаторов [6]. Кроме того, поскольку задача верификации программы является алгоритмически неразрешимой [3], переход от программы к объекту с конечным числом состояний является существенным шагом при решении задач верификации. В частности, автомат является формальной конечной моделью, а значит, к нему применимы методы проверки модели (*Model Checking* [7]).

Перейдем к описанию **второго подхода**. При этом осуществляется построение автомата *до* написания программы – *проектирование* программы с явным выделением состояний. При данном подходе снова возможны два варианта.

Первый из них состоит в построении автомата по алгоритму (формальному описанию). В отличие от построения автомата по программе в данном случае полностью формальный переход затруднителен: описание алгоритма допускает использование инструкций, формальный перевод которых на язык программирования сложен. Например, это имеет место для инструкции «*поместить вершину графа в конец очереди*», так как от программиста требуется самостоятельно принять решение о способе реализации очереди и о способе представления вершин графа.

Формализм в описании алгоритма, с одной стороны, помогает в построении автомата, так как строго задает последовательность действий в различных случаях, а с другой, как будет показано ниже, может привести к построению неэффективного автомата, так как он будет слишком детально наследовать логику и структурные элементы алгоритма.

Второй вариант подхода состоит в том, что постановка задачи осуществляется в виде вербального описания. При этом возможно выявление противоречий или неполноты в постановке задачи. Проектирование с явным выделением состояний позволяет *до* построения программы и алгоритма (так как автомат и является алгоритмом) устранить эти проблемы. Однако такой подход полностью исключает возможность применения формальных методов для выделения состояний автомата.

Введенная классификация подходов к проектированию автоматов и их недостатки приведены на рис. 2.

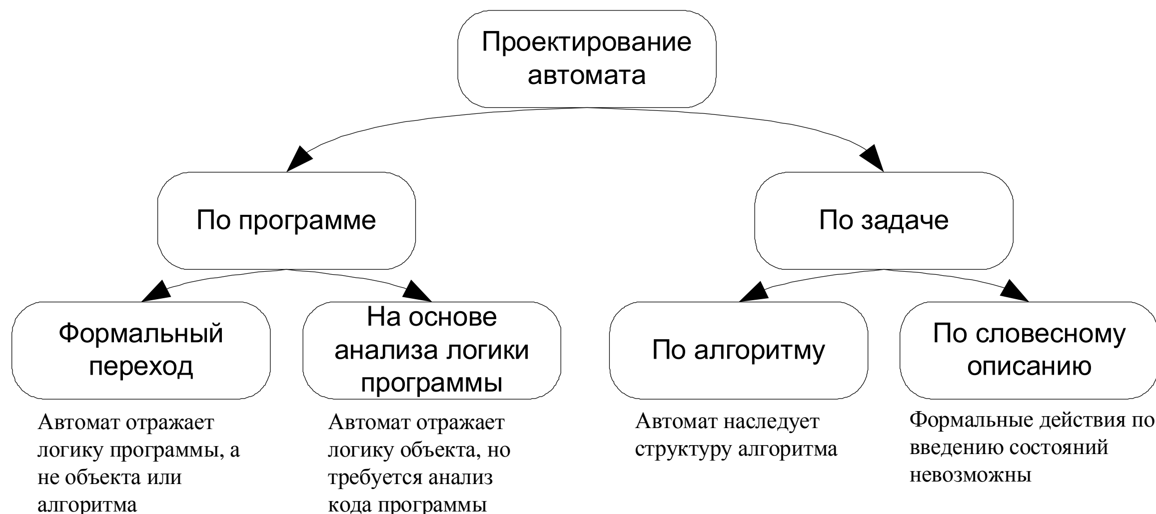

Рис. 2. Возможные подходы к проектированию автомата

В настоящей работе подробно рассмотрены три последних подхода. Первый (максимально формальный) подход изложен, например, в работе [5], и поэтому здесь не рассматривается.

Наиболее естественной областью применения программирования с явным выделением состояний являются задачи управления сложными техническими объектами, такими как дизель-генератор [8] или танк [9]. Однако отсутствие знаний в соответствующей предметной области и сложность объекта управления затрудняют понимание процесса построения управляющего автомата многими программистами. В то же время такая технология программирования может быть применена не только в задачах управления техническими объектами. Так, даже в таких областях, как структуры данных и вычислительные алгоритмы, явное выделение состояний может существенно упростить этап проектирования логики управления сложными объектами, что и будет показано в настоящей работе. Кроме того, поскольку состояния автомата отражают суть сложного процесса более естественно и наглядно, чем код программы, поиск и устранение ошибок заметно упрощаются после явного выделения состояний.

Еще одной сферой применения программирования с явным выделением состояний являются задачи визуализации, когда объект управления отображается в промежуточных состояниях, что позволяет подробно изучить его структуру и функционирование. В данном случае применение программирования с явным выделением состояний особенно естественно, поскольку состояния управляющего автомата оказываются как раз визуализируемыми состояниями [6]. Таким образом, после реализации структуры данных или вычислительного алгоритма с помощью рассматриваемой технологии, имеется возможность построить визуализатор, для которого любая сложная последовательность действий является лишь цепочкой переходов управляющего автомата. В таких условиях становится возможной не только пошаговая визуализация работы алгоритма, но и, при правильно организованном хранении истории действий, визуализация шагов назад [10].

В настоящей работе предлагаемые подходы демонстрируются на примере построения автоматов, управляющих структурой данных – *АВЛ-деревом*. Рассматривается именно эта структура данных, поскольку работа с АВЛ-деревьями связана помимо технических моментов с содержательной логикой управления. При этом объектом управления является само дерево, а в качестве системы управления предлагается использовать автомат.

Следующий раздел содержит краткое описание АВЛ-деревьев. Более подробное описание АВЛ-деревьев и аналогичных структур данных можно найти, например, в работе [11].

1. Двоичные деревья поиска

Рассмотрим некоторое множество *U*. Часто возникает задача организации *коллекции* из некоторых его элементов. Коллекция представляет собой структуру данных, поддерживающую следующие операции:

– создание пустой коллекции;

– добавление элемента;
– удаление элемента;
– проверка наличия элемента в коллекции.

Если U представляет собой линейно упорядоченное множество элементов (например, множество целых чисел, упорядоченных естественным образом, или множество строк над некоторым алфавитом, которые упорядочены лексикографически), то при организации коллекции из элементов такого множества одной из наиболее часто используемых является ее реализация в виде двоичного дерева поиска.

Такая структура данных представляет собой двоичное дерево с корнем, каждая вершина которого содержит элемент множества U – *ключ*. Обозначим ключ, хранящийся вершине x, как v_x. При этом для каждой вершины x в двоичном дереве поиска должны выполняться следующие два условия:

1. Для каждой вершины y в левом поддереве вершины x хранящийся в ней ключ v_y строго меньше ключа v_x.
2. Для каждой вершины z в правом поддереве вершины x хранящийся в ней ключ v_z строго больше ключа v_x.

На рис. 3 приведено двоичное дерево поиска, и показаны левое и правое поддеревья некоторых вершин. На рис. 3, а – поддеревья вершины с ключом 7, а на рис. 3, б – поддеревья вершины с ключом 4.

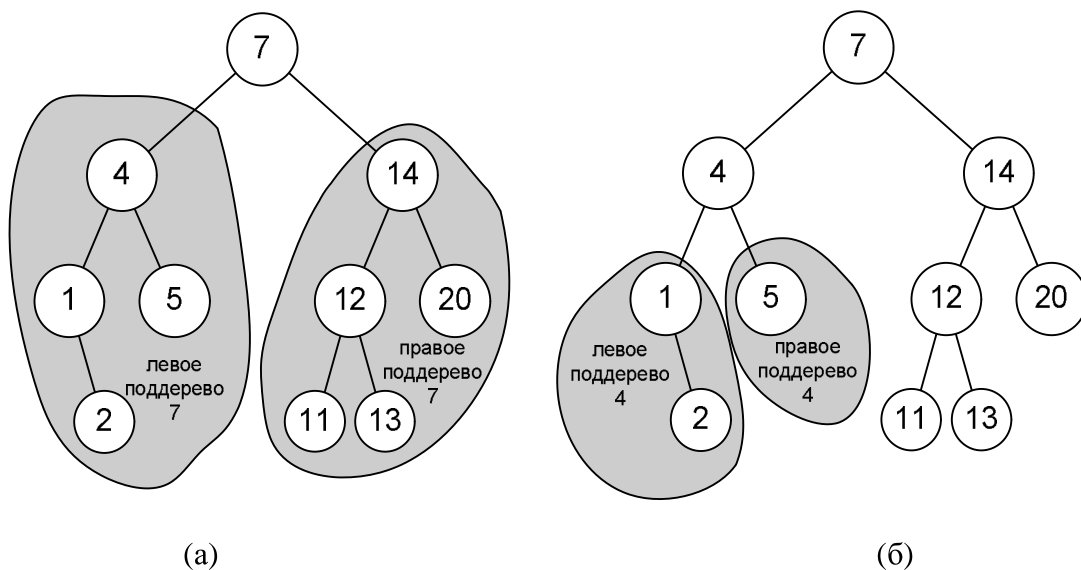

Рис. 3. Двоичное дерево поиска

Поиск, добавление и удаление вершины из этого дерева выполняются естественным образом, благодаря указанному упорядочению вершин. Так, начиная поиск от корня, при рассмотрении очередного узла дерева, достаточно сравнить ключ, хранящийся в этом узле, с искомым ключом. Результат этого сравнения соответствует тому, что в левом либо в правом поддереве следует искать наш элемент. Двигаясь таким образом вниз по дереву, либо обнаруживается узел с искомым ключом, либо будет необходимо перейти в пустое поддерево какого-либо узла, что говорит об отсутствии искомого ключа в данном дереве.

Отметим, что если непосредственно реализовать двоичное дерево поиска, то производительность основных операций оказывается неудовлетворительной (табл. 1).

Таблица 1. Асимптотическая производительность операций с деревом

Операция	**Производительность**
Создание пустой коллекции	$O(1)$
Добавление элемента	$O(N)$
Удаление элемента	$O(N)$
Проверка наличия элемента в коллекции	$O(N)$

В табл. 1 число элементов в коллекции в момент выполнения соответствующей операции обозначено *N*. Можно построить пример последовательности операций с деревом, при котором достигаются указанные верхние оценки – каждая операция будет выполняться за $\Omega(N)$. Это, например, имеет место при добавлении элементов в возрастающем порядке.

Отметим, что столь низкая производительность обусловлена тем, что в общем случае дерево является *несбалансированным* – левое и правое поддеревья вершины могут сильно отличаться по высоте. В частности, в крайнем случае, дерево вообще вырождается в линейный список (рис. 4).

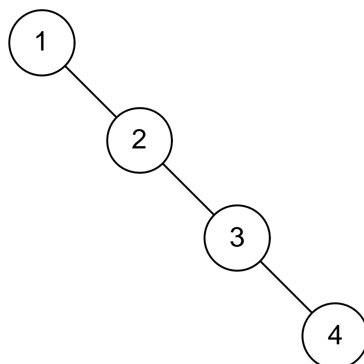

Рис. 4. Дерево, выродившееся в линейный список

Для борьбы с указанным недостатком используются различные способы *балансировки* двоичных деревьев поиска. Первый и один из наиболее эффективных способов балансировки был предложен в 1962 году Г. М. Адельсоном-Вельским и Е. М. Ландисом в работе [12]. Деревья, при работе с которыми используется предложенный ими способ балансировки, называются АВЛ-деревьями. Этот способ балансировки основан на операциях перестраивания дерева, за счет которых поддерживается следующее дополнительное (третье) условие, характеризующее такие деревья:

3. Для каждой вершины разность высот левого и правого поддеревьев не превышает единицы по абсолютной величине.

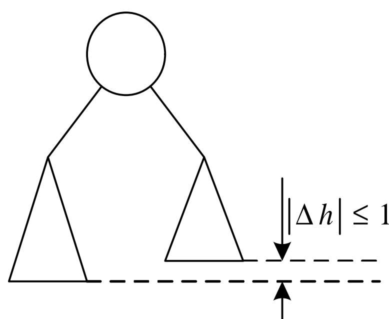

Рис. 5. АВЛ-дерево

При выполнении этого свойства высота дерева равна $O(\log N)$. Благодаря этому, производительность операций с деревом из линейной становится логарифмической (табл. 2).

Таблица 2. Асимптотическая производительность операций с АВЛ-деревом

Операция	**Производительность**
Создание пустой коллекции	$O(1)$
Добавление элемента	$O(\log N)$
Удаление элемента	$O(\log N)$
Проверка наличия элемента в коллекции	$O(\log N)$

Балансировка основана на возможности перестраивания дерева с сохранением свойства упорядоченности. Она осуществляется при добавлении и удалении элементов из дерева, в случае

если для текущей вершины третье свойство нарушается – разность высот поддеревьев превышает единицу по абсолютной величине. При этом можно показать [11], что она не может превысить значения два по абсолютной величине, если перед выполнением операции дерево было сбалансировано.

При работе с АВЛ-деревьями выполняются четыре вида балансировки – LL-балансировка, LR-балансировка, и симметричные им RR-балансировка и RL-балансировка. Применение конкретного вида балансировки зависит от того, какое – левое или правое поддерево текущей вершины является более высоким, а также от того, какое – левое или правое поддерево ребенка текущей вершины, являющегося корнем более высокого поддерева, является более высоким. Так, на рис. 6 показан случай, когда применяется LL-балансировка, и соответствующее ей преобразование дерева. Все треугольники на этом рисунке считаются поддеревьями, имеющими одну и ту же высоту, кроме треугольников 1 и 2, один из которых может иметь высоту на единицу меньше, чем все остальные.

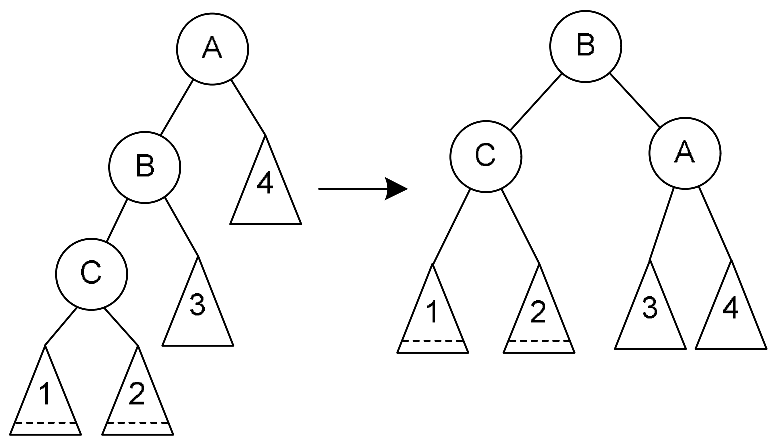

Рис. 6. LL-балансировка АВЛ-дерева

На рис. 7 аналогичным образом показано, как выполняется LR-балансировка.

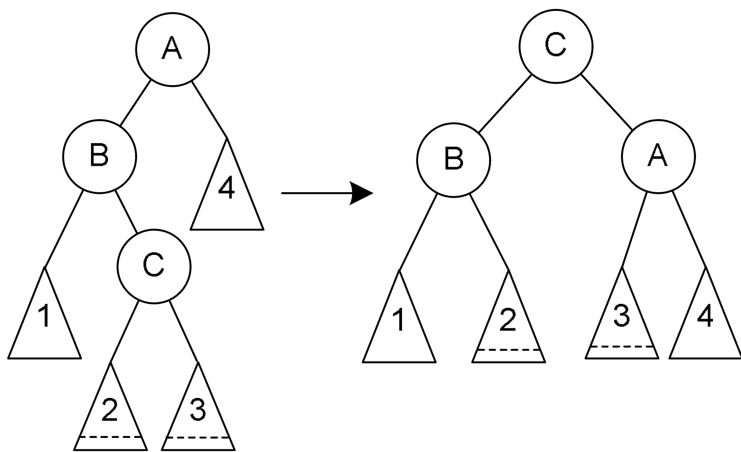

Рис. 7. LR-балансировка АВЛ-дерева

Алгоритмы работы с АВЛ-деревом (поиск, добавление и удаление элемента) будут рассмотрены ниже при построении соответствующих автоматов.

2. *Построение управляющих автоматов*

2.1. Начальное приближение

Сформулируем задачу следующим образом. Имеется объект управления – дерево. Цель построения автоматов – управлять деревом так, чтобы при осуществлении различных операций поведение дерева соответствовало поведению АВЛ-дерева.

Предположим первоначально, что дерево представляет собой черный ящик, на который могут оказываться три воздействия – осуществить поиск, добавить или удалить элемент. Заметим, что каждое из этих воздействий имеет внешний параметр – элемент, который необходимо найти, добавить или удалить из коллекции. Этот элемент и его соотношение с другими элементами, которые уже есть в дереве, определяет значения входных переменных. Будем считать, что имеется доступ к параметру текущей операции из функций входных переменных и выходных воздействий, и работа с ним сосредоточена в этих функциях.

Схема связей [1] автомата управления черным ящиком представлена на рис. 8.

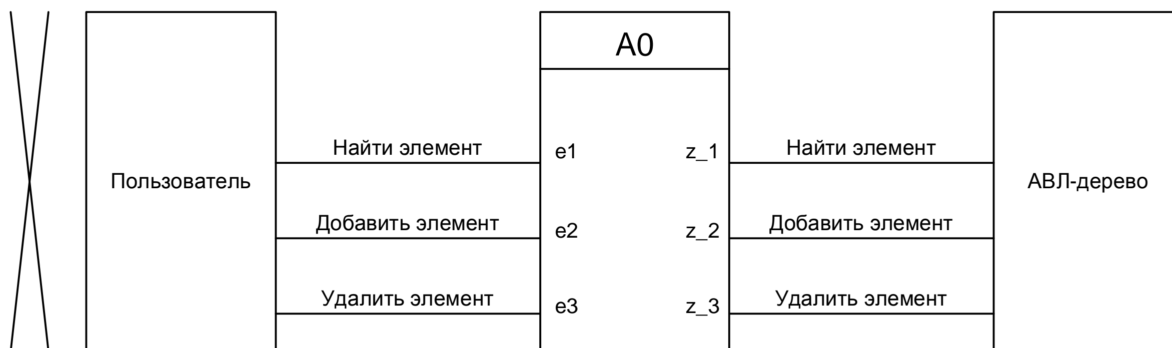

Рис. 8. Схема связей автомата управления черным ящиком

Граф переходов управляющего автомата приведен на рис. 9.

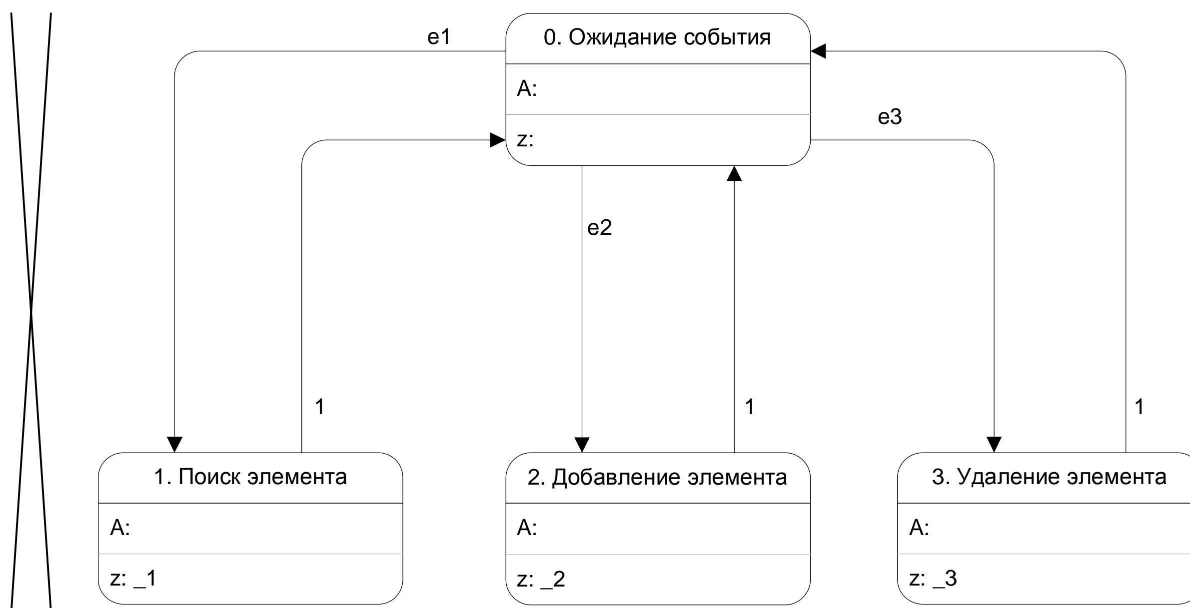

Рис. 9. Граф переходов автомата управления черным ящиком

Отметим, что в настоящей работе используется тот же методический прием, что и в книге [13] – в процессе изложения в качестве промежуточных шагов могут получаться схемы связей и графы переходов, которые в силу различных причин не могут или не должны использоваться. Такие диаграммы будем отмечать «крестом».

К диаграммам этого класса относятся диаграммы, изображенные на рис. 8 и 9, так как введенный автомат бесполезен. Действительно, он пока не делает *ничего*, кроме как передает

запросы дереву. При этом дерево, управление которым выполняется, является уже АВЛ-деревом, и ему требуется лишь подавать запросы. Так что автомат абсолютно бесполезен – можно напрямую «закоротить» пользователя непосредственно на дерево, убрав в схеме автомат.

Разумеется, это не то, что требуется, так как было сделано предположение, что дерево настолько умное, что умеет выполнять запросы высокого уровня такие, как, например, «*добавить в коллекцию элемент*». Конечно, нельзя рассчитывать на то, что какой-нибудь другой объект, которым требуется управлять, окажется таким же умным – например, весьма сомнительно, чтобы трактор смог самостоятельно исполнить команду «*повернуть налево*». Скорее всего, потребуется давать гораздо более низкоуровневые команды. Придется проанализировать и реализовать то, что пока скрывается внутри черного ящика.

Отметим, что построенный автомат не является совсем бессмысленным. В разд. 2.5 он будет преобразован в главный автомат, координирующий работу автоматов, управляющих деревом при отдельных операциях.

2.2. Построение автомата по программе

Первый рассматриваемый способ построения автомата использует имеющуюся программу, причем здесь речь идет не о формальном преобразовании программы с сохранением ее структурных элементов, а о построении наиболее естественного автомата, повторяющего функциональность исходной программы. Можно сказать, что описываемое преобразование является частным случаем рефакторинга.

В качестве примера, применим данный способ для построения автомата, реализующего поиск в АВЛ-дереве. Сначала рассмотрим стандартную реализацию процедуры поиска элемента, приведенную в фрагменте листинга 1 (все тексты программ в этой работе приведены на языке *Java*).

Фрагмент листинга 1. Поиск в двоичном дереве

```
/** Узел дерева */
private static class TreeNode {
    /** Левый ребенок */
    public TreeNode left;
    /** Правый ребенок */
    public TreeNode right;
    /** Родитель */
    public TreeNode parent;
    /** Ключ */
    public int key;
}

/** Корень дерева */
public TreeNode root;

/** Найти элемент в дереве */
boolean find(int key) {
    TreeNode currentNode = root;
    while (currentNode != null) {
        if (currentNode.key > key) {
            currentNode = currentNode.left;
        } else if (currentNode.key < key) {
            currentNode = currentNode.right;
        } else {
            // Нашли значение
            return true;
        }
    }
    return false;
}
```

Из приведенного фрагмента следует, что имеется *текущий указатель* на некоторый узел дерева, который сначала устанавливается в корне, а затем, в зависимости от текущего элемента,

перемещается от узла к его левому или правому сыну, если это возможно. При невозможности перемещения результат поиска отрицательный, а если указатель оказывается в узле, хранящем элемент, поиск которого выполняется, то результат положительный.

Реализуем этот процесс с помощью автомата поиска, который будет *вложен* в главный автомат управления. Для этого необходимо получить доступ к внутреннему устройству АВЛ-дерева. Будем считать, что внутри объекта управления имеется указатель, который в каждый момент времени указывает на некоторый узел дерева и который может проверить, что он не указывает на пустое место, и может перемещаться к левому или правому сыну. Также он может сравнивать значение ключа в узле, на который указывает, со значением, поиск которого выполняется. Реализуем это с помощью входных переменных и выходных воздействий автомата. Также в качестве выходных воздействий передадим результат поиска – найден или не найден элемент.

Входные переменные приведены в табл. 3.

Таблица 3. Входные переменные, используемые при реализации автомата поиска

Вход	**ОУ**	**Значение**
$x10$	Д	Текущий узел есть пустая ссылка
$x20$	Д	Значение ключа в текущем узле больше значения ключа-параметра
$x30$	Д	Значение ключа в текущем узле меньше значения ключа-параметра

Выходные воздействия приведены в табл. 4.

Таблица 4. Выходные воздействия, используемые при реализации автомата поиска

Выход	**ОУ**	**Смысл**	**Действие**
$z10$	Д	Перейти в корень	Текущий узел ← корень дерева
$z40$	Д	Пойти налево	Текущий узел ← левый сын текущего узла
$z50$	Д	Пойти направо	Текущий узел ← правый сын текущего узла
$z190$	Р	Сбросить результат	Установить результат операции как неопределенный
$z200$	Р	Элемент найден	Сообщить оператору, что элемент найден
$z210$	Р	Элемент не найден	Сообщить оператору, что элемент не найден

Схема связей автомата поиска приведена на рис. 10.

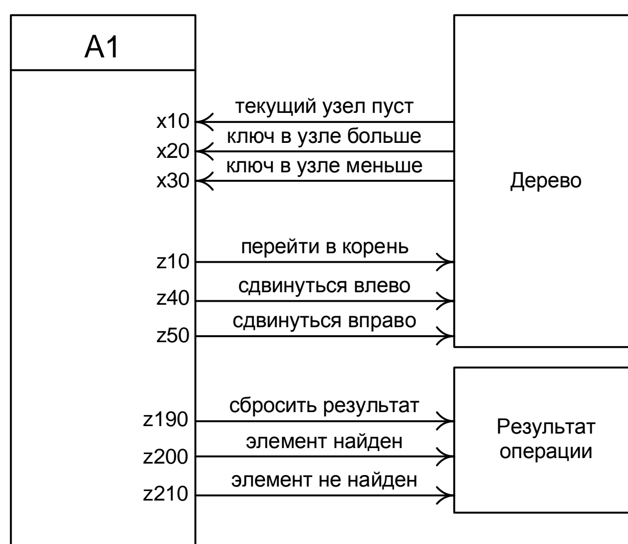

Рис. 10. Схема связей автомата поиска

Отметим, что при задании входных переменных и выходных воздействий возникает вопрос, имеющий принципиальное значение при дальнейшей реализации автомата в виде программы: принадлежат ли входные переменные и выходные воздействия объекту управления или управляющему автомату. Поскольку технология программирования с явным выделением

состояний появилась в эпоху процедурного программирования, то сначала такой проблемы не возникало. Действительно, входные переменные и выходные воздействия могли быть реализованы в виде глобальных функций, имеющих доступ к глобальной переменной, содержащей объект управления. Однако в условиях объектно-ориентированного программирования (ООП) такой подход, разумеется, нецелесообразен.

Здесь важно сделать следующее замечание. Программирование с явным выделением состояний является технологией, *независимой* от способа представления объектов в программе и способа реализации связей между объектами – от применения процедурного или объектно-ориентированного программирования. Можно сказать, что технология реализации логики (в нашем случае выбрана технология с явным выделением состояний) и технология представления объектов и выполнения взаимодействия между ними в программе «ортогональны» (рис. 11).

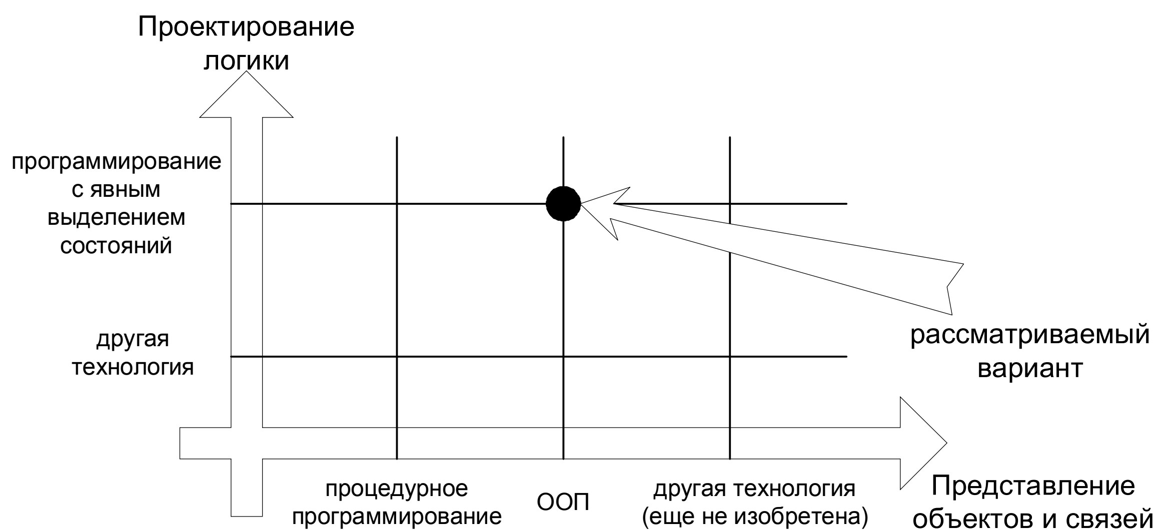

Рис. 11. Независимость технологии проектирования от реализации

Наиболее естественным при применении объектно-ориентированного подхода является отнесение входных переменных и управляющих воздействий к объекту управления. Однако это порождает некоторые проблемы, которые необходимо решить. При проектировании в случае такого подхода возникает проблема, состоящая в том, что при отнесении входных переменных и выходных воздействий к объекту управления требуется указывать (при наличии нескольких объектов), к какому объекту относится наше воздействие. Так, в рассматриваемом случае появился новый объект – «результат операции», к которому можно применить выходные воздействия $z190$, $z200$ и $z210$. Эта проблема решается сравнительно легко, в частности, она уже решена в табл. 3 и 4. В них введен столбец ОУ (объект управления), возможные значения в котором – Д (дерево) и Р (результат).

Другие проблемы, возникающие при реализации, будут проанализированы в разд. 3, и для них будут предложены пути решения.

Сейчас же вернемся к рассмотрению автомата поиска. Граф переходов этого автомата приведен на рис. 12.

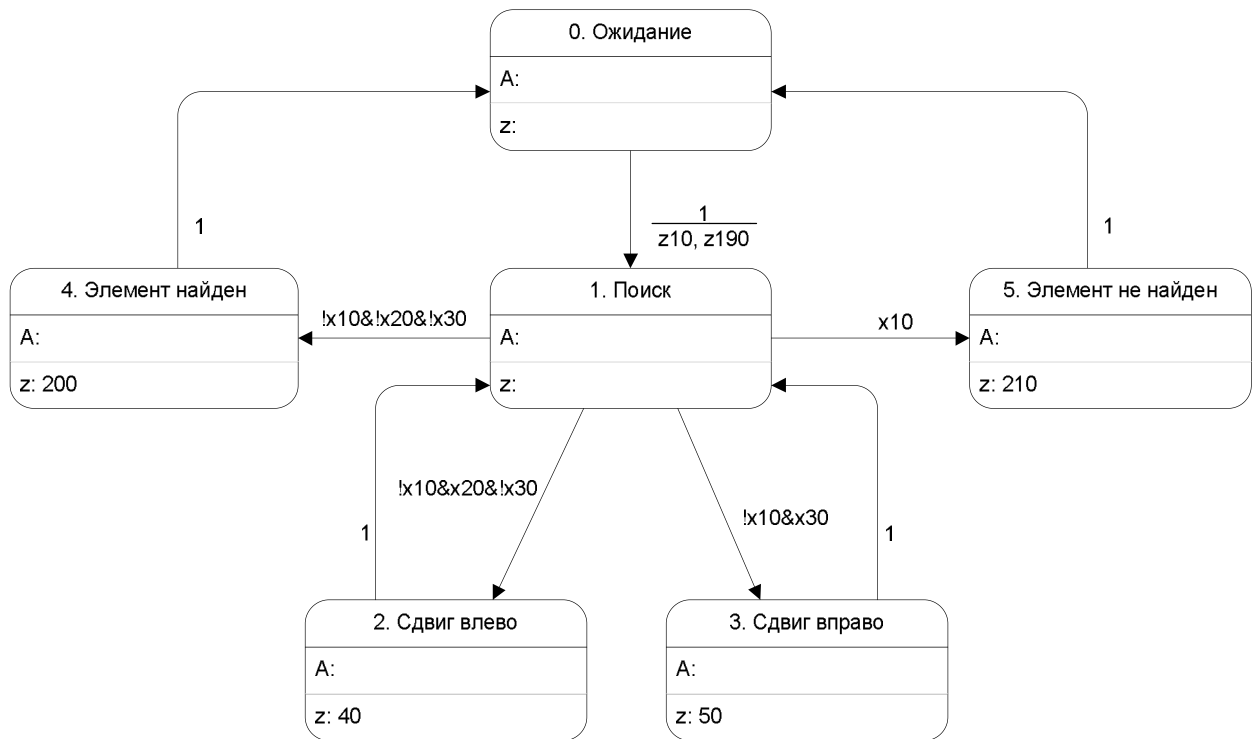

Рис. 12. Граф переходов автомата поиска элемента

Отметим важный момент. Как уже упоминалось, имеется возможность формально построить автомат по приведенной программе, который бы в точности повторял ее структуру. Однако был выбран альтернативный вариант – действия программы были проанализированы, и был построен автомат, реализующий эти действия. При этом конкретная программная реализация не была использована.

Поясним, почему был выбран такой подход. Результат формального преобразования программы в автомат обычно неудачен с точки зрения его структуры – дело в том, что программа, записанная на языке программирования, таком, как используемый в настоящей работе язык *Java*, теряет в определенном смысле связь с исходной логикой, заложенной в нее, поскольку она оказывается выраженной с помощью специального набора операторов. Полученный автомат действует как интерпретатор языка программирования, выполняя инструкции программы. Аналогией может служить формальное преобразование программы с одного языка программирования на другой – эффективность и понятность программы при этом может быть утеряна безвозвратно, особенно если языки существенно отличаются. Поэтому вместо того, чтобы формально преобразовывать программу в автомат, правильнее выяснить, *что* делает программа, а затем построить автомат, реализующий эти действия.

Для того чтобы убедиться, что полученный таким образом результат действительно лучше, чем формальный, построим автомат формально по тексту программы (рис. 13).

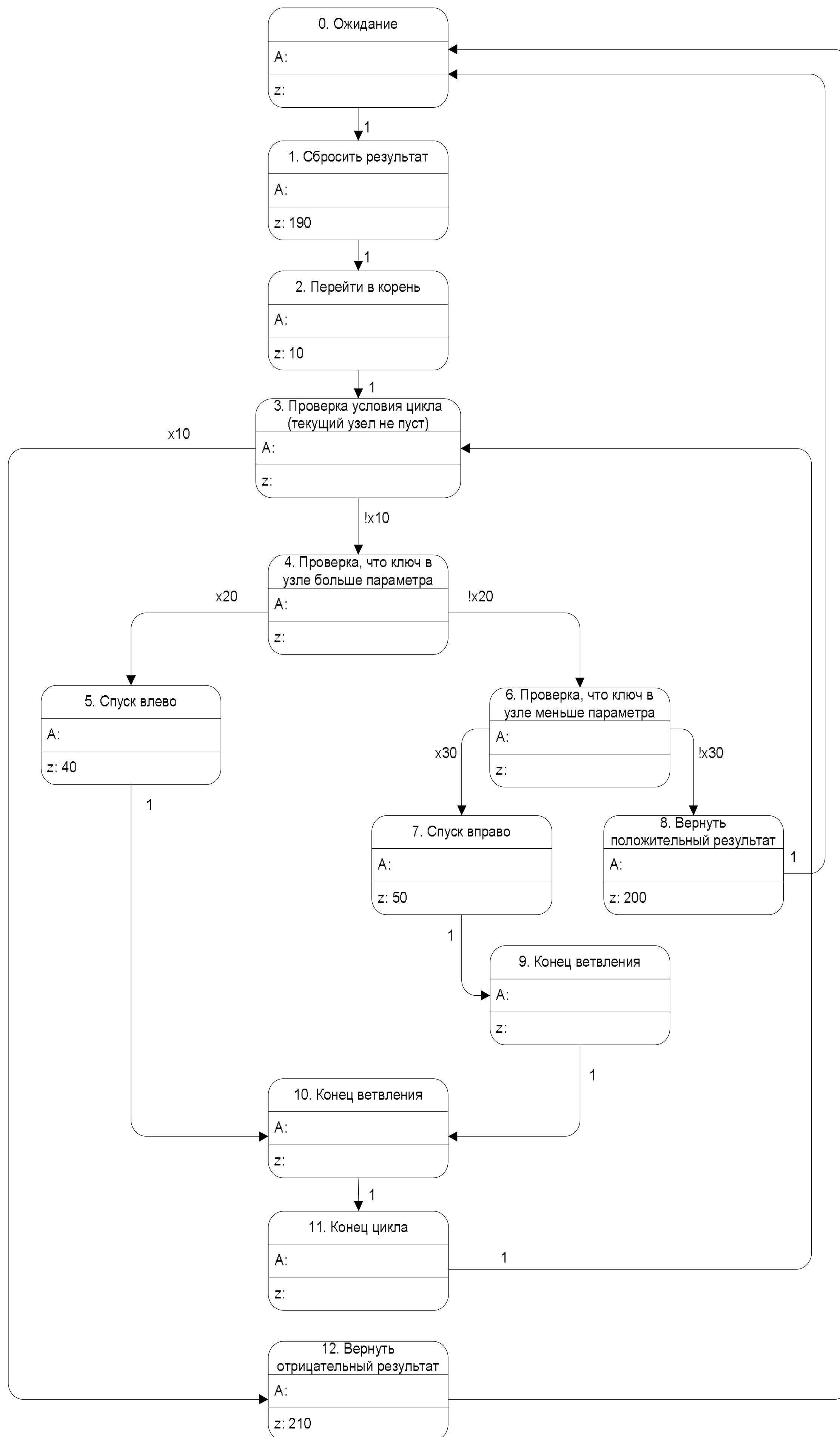

Рис. 13. Граф переходов автомата, построенного формально по тексту программы

В первом построенном автомате (рис. 12) четко видны *состояния* алгоритма поиска (поиск, спуск, возвращение конкретного результата). В формально же построенном автомате состояния соответствуют операторам программы – циклу и ветвлению. В этом случае нет соответствия между состояниями *объекта* и *алгоритма* и состояниями *автомата*. Поэтому затруднены анализ и модификация построенного таким образом автомата. Проще анализировать и модифицировать исходную программу.

2.3. Построение автомата по формальному описанию алгоритма

С использованием подхода, изложенного в предыдущем разделе, можно было бы по тексту процедур построить управляющие автоматы и для остальных операций с АВЛ-деревьями. Однако это довольно бесполезное занятие – если уже есть программа, реализующая операцию, то не имеет смысла строить автомат. Поэтому будем сначала строить автоматы, и лишь потом по ним создавать программу.

Построим автомат по описанию операции добавления элемента в АВЛ-дерево. Пусть дано следующее алгоритмическое описание операции добавления.

I. Поиск места в дереве, в которое следует добавить элемент.
   1. Текущий узел ← корень дерева, текущий отец ← пустая ссылка.
   2. Если текущий узел – пустая ссылка, то перейти к шагу II.
   3. Если ключ в текущем узле больше ключа добавляемого элемента, то сдвинуться налево:
      i. Текущий отец ← текущий узел.
      ii. Текущий узел ← левый сын текущего узла.
      iii. Перейти к пункту 2.
   4. Если ключ в текущем узле меньше ключа добавляемого элемента, то сдвинуться направо:
      i. Текущий отец ← текущий узел.
      ii. Текущий узел ← правый сын текущего узла.
      iii. Перейти к пункту 2.

II. Создание нового узла дерева.
   1. Текущий узел ← новый узел.
   2. Отец текущего узла ← текущий отец.
   3. Левый сын текущего узла ← пустая ссылка.
   4. Правый сын текущего узла ← пустая ссылка.
   5. Ключ текущего узла ← ключ, добавление которого осуществляется.
   6. Глубина текущего узла ← 0.

III. Возврат и балансировка.
   1. Текущий узел ← отец текущего узла.
   2. Если текущий узел – пустая ссылка, то выход.
   3. Обновить высоту текущего узла.
   4. Если высота левого поддерева текущего узла больше высоты правого поддерева на два, то:
      i. Если высота левого поддерева левого поддерева больше высоты правого поддерева левого поддерева, то выполнить балансировку типа LL.
      ii. Если высота правого поддерева левого поддерева больше высоты левого поддерева левого поддерева, то выполнить балансировку типа LR.
   5. Если высота правого поддерева текущего узла больше высоты левого поддерева на два, то
      i. Если высота правого поддерева правого поддерева больше высоты левого поддерева правого поддерева, то выполнить балансировку типа RR.
      ii. Если высота левого поддерева правого поддерева больше высоты правого поддерева правого поддерева, то выполнить балансировку типа RL.
   6. Перейти к пункту 1.

Добавим входные переменные (табл. 5) и выходные воздействия (табл. 6), необходимые для реализации операции добавления.

Таблица 5. Новые входные переменные, введенные при реализации автомата добавления

Вход	**ОУ**	**Значение**
$x40$	Д	Высота левого поддерева текущего узла больше высоты правого поддерева текущего узла на 2
$x50$	Д	Высота правого поддерева текущего узла больше высоты левого поддерева текущего узла на 2
$x60$	Д	Высота левого поддерева левого поддерева больше высоты правого поддерева левого поддерева
$x70$	Д	Высота правого поддерева левого поддерева больше высоты левого поддерева левого поддерева
$x80$	Д	Высота левого поддерева правого поддерева больше высоты левого поддерева правого поддерева
$x90$	Д	Высота правого поддерева правого поддерева больше высоты правого поддерева правого поддерева

Таблица 6. Новые выходные воздействия, введенные при реализации автомата добавления

Выход	**ОУ**	**Смысл**	**Действие**
$z20$	Д	Обнулить текущего отца	Текущий отец ← пустая ссылка
$z30$	Д	Спуститься вниз	Текущий отец ← текущий узел
$z60$	Д	Создать новый узел	Создать в текущем узле новый узел в соответствии с шагом II
$z70$	Д	Подняться наверх	Текущий узел ← отец текущего узла
$z71$	Д	Обновить высоту	Установить высоту текущего узла как максимум высот его детей плюс один
$z80$	Д	LL-балансировка	Выполнить LL-балансировку текущего узла
$z90$	Д	LR-балансировка	Выполнить LR-балансировку текущего узла
$z100$	Д	RL-балансировка	Выполнить RL-балансировку текущего узла
$z110$	Д	RR-балансировка	Выполнить RR-балансировку текущего узла

Заметим, что можно было бы разбить выходное воздействие $z60$ на несколько более мелких действий в соответствии с пунктами в шаге II, однако в этом нет никакой необходимости, так как все эти действия *всегда* выполняются вместе. Поэтому можно считать воздействие $z60$ в некотором роде «макросом», состоящим из нескольких простых действий, которые можно собрать в функции, реализующей это воздействие.

Выделим состояния автомата, который будет выполнять добавление элемента в дерево. Трем основным стадиям алгоритма соответствуют состояния *поиска*, *создания вершины* и *подъема* соответственно. При поиске, как и в случае реализации операции поиска элемента, введем состояния перемещения по дереву влево и вправо. При подъеме потребуются состояния для анализа баланса и балансировки. Схема связей построенного автомата приведена на рис. 14, а его граф переходов – на рис. 15.

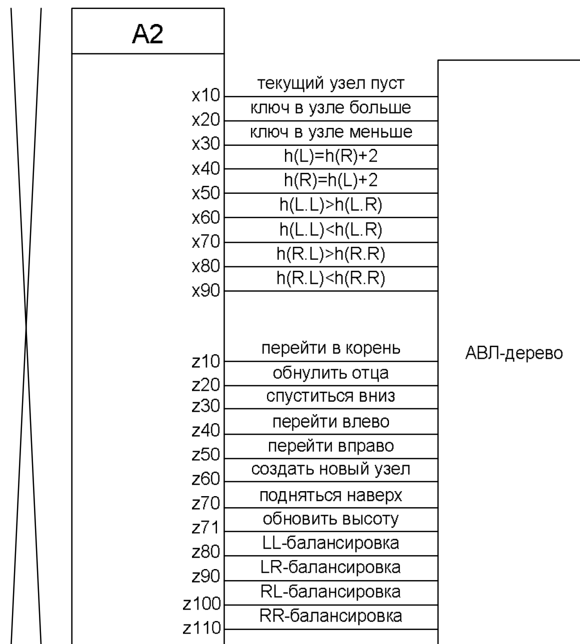

Рис. 14. Схема связей автомата добавления, первая версия

0. Ожидание
A:
z:

1
z10, z20

1. Поиск
A:
z:

x10

4. Создание вершины
A:
z: 60

1
!x10&x20
!x10&x30
1

2. Сдвиг влево
A:
z: 30, 40

3. Сдвиг вправо
A:
z: 30, 50

1

5. Подъем
A:
z: 70

!x10
z71

x40&x60

8. LL-балансировка
A:
z: 80

6. Проверка левого баланса
A:
z:

1
1

9. LR-балансировка
A:
z: 90

x40&x70

x10

1

!x50

!x40

12. Завершение добавления
A:
z:

x50&x80

10. RL-балансировка
A:
z: 100

7. Проверка правого баланса
A:
z:

1
1

11. RR-балансировка
A:
z: 110

x50&x90

Рис. 15. Автомат добавления, первая версия

В отличие от автомата поиска, который был построен по готовой программе, автомат добавления был создан по формальному описанию алгоритма. Как следствие, если исходный код, по которому строится автомат поиска, мог быть протестирован и ошибки в нем могли быть устранены, то для приведенного выше текста алгоритма это не могло быть сделано. Поэтому посмотрим на него более внимательно.

Интересно, что полученный автомат оказался некорректным. В состоянии 1 (поиск) не определено поведение автомата при комбинациях $!x10x20x30$ и $!x10!x20!x30$. В первом случае автомат может перейти как в состояние 2, так и в состояние 3, а во втором случае переход вообще не задан.

Первая ситуация логически невозможна, и то, что она имеет место, означает, что произошла некоторая фатальная ошибка. Это является следствием того, что входные переменные не являются независимыми – одновременное выполнение условий $x20$ и $x30$ невозможно. Для устранения противоречия необходимо *ортогонализовать* соответствующие переходы автомата – усилить условие на одной из дуг, например, заменив условие $!x10x20$ на $!x10x20!x30$. Формально ортогонализация автомата состоит в приведении выходящих из состояния дуг к форме, когда условия на любых двух дугах несовместны.

Итак, был рассмотрен один из вариантов – с помощью ортогонализации было устранено противоречие. Однако второй случай хуже – такая комбинация входных переменных вполне возможна. Это означает, что число, которое автомат пытается добавить, уже есть в дереве (в имеющейся коллекции), и указатель находится как раз в узле, который содержит это число.

Таким образом, была найдена ошибка в алгоритме – он неправильно обрабатывает случай добавления в коллекцию числа, которое там уже присутствует. Отметим, что эту ошибку можно было заметить уже на этапе построения автомата – когда разрабатывали фрагмент, соответствующий поиску ключа в дереве, – ведь данный фрагмент очень напоминает автомат, который был использован для реализации операции поиска элемента в дереве, а там подобная ситуация корректно обрабатывалась. Исправим автомат. Пусть он сразу переходит в состояние, соответствующее концу добавления.

В имеющемся автомате есть еще два состояния, из которых не все переходы определены – *состояние 6* при $x40x60x70$ или $x40!x60!x70$ и *состояние 7* при $x50x80x90$ или $x50!x80!x90$. В обоих случаях первая проблема устраняется путем ортогонализации. Вторая же заслуживает особого внимания.

При *добавлении* элемента в АВЛ-дерево такая комбинация невозможна. Однако при *удалении* элемента может случиться, что левое и правое поддеревья более высокого поддерева текущего узла будут иметь равную высоту. В этом случае следует выполнять более простую балансировку – LL и RR соответственно. Конечно, сейчас это кажется не важным, так как сейчас реализуется автомат для операции добавления. Обратим внимание на тот факт, что балансировка, а поэтому и анализ высоты узлов, необходимый для определения, какой тип балансировки следует применить для текущего узла, необходимы как в операции добавления, так и в операции удаления элемента. Это подталкивает к решению выделить данные действия в отдельные автоматы, которые будут осуществлять балансировку. Изложенное напоминает известный принцип структурного программирования: одно и то же действие, выполняющееся в двух местах, следует оформить в виде процедуры.

Итак, построим два дополнительных автомата – для левой и правой балансировки. При этом удобно перенести в эти автоматы и проверку того, какой вид балансировки следует осуществлять – LL или LR (или RR или RL соответственно), и удалить из автомата добавления группу состояний, соответствующих выбору балансировки. Что касается проблемы с неопределенностью в поведении автоматов балансировки в случае, если высоты поддеревьев равны, то она решается следующим образом. Удалим входные переменные $x60$ и $x90$, заменив условия $x40x60$ и $x50x90$ на $x40!x70$ и $x50!x80$ соответственно.

Помимо сделанных исправлений обращает на себя внимание не совсем разумное поведение автомата после балансировки, связанной с «перекосом» влево – даже после балансировки все равно анализируется возможность более высокого правого поддерева. Эта особенность рассматриваемого автомата связана с тем, что при последовательном изложении алгоритма сложно описать «троичное ветвление» – левое поддерево существенно выше, правое поддерево

существенно выше, высоты поддеревьев соотносятся в рамках правила 3. Однако в автомате ничто не мешает осуществить выход из состояния по трем различным дугам в зависимости от условий. Сделаем это.

В результате всех перечисленных исправлений схема связей автомата и сам автомат немного упрощаются и принимают вид, изображенный на рис. 16 и 17 соответственно. Заметим, что из схемы связей исчезли не только удаленные входные переменные $x60$ и $x90$, но и входные переменные $x70$ и $x80$, а также выходные воздействия $z80$, $z90$, $z100$ и $z110$, общение с которыми было перенесено во вложенные автоматы балансировки.

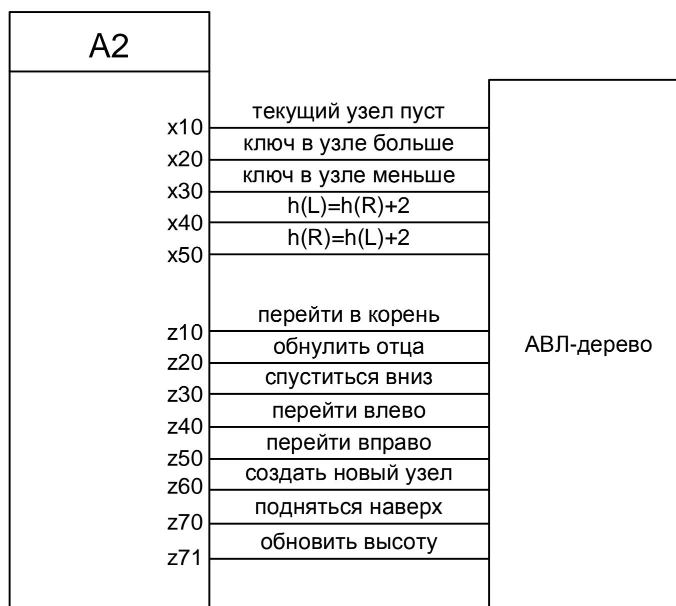

Рис. 16. Схема связей автомата добавления

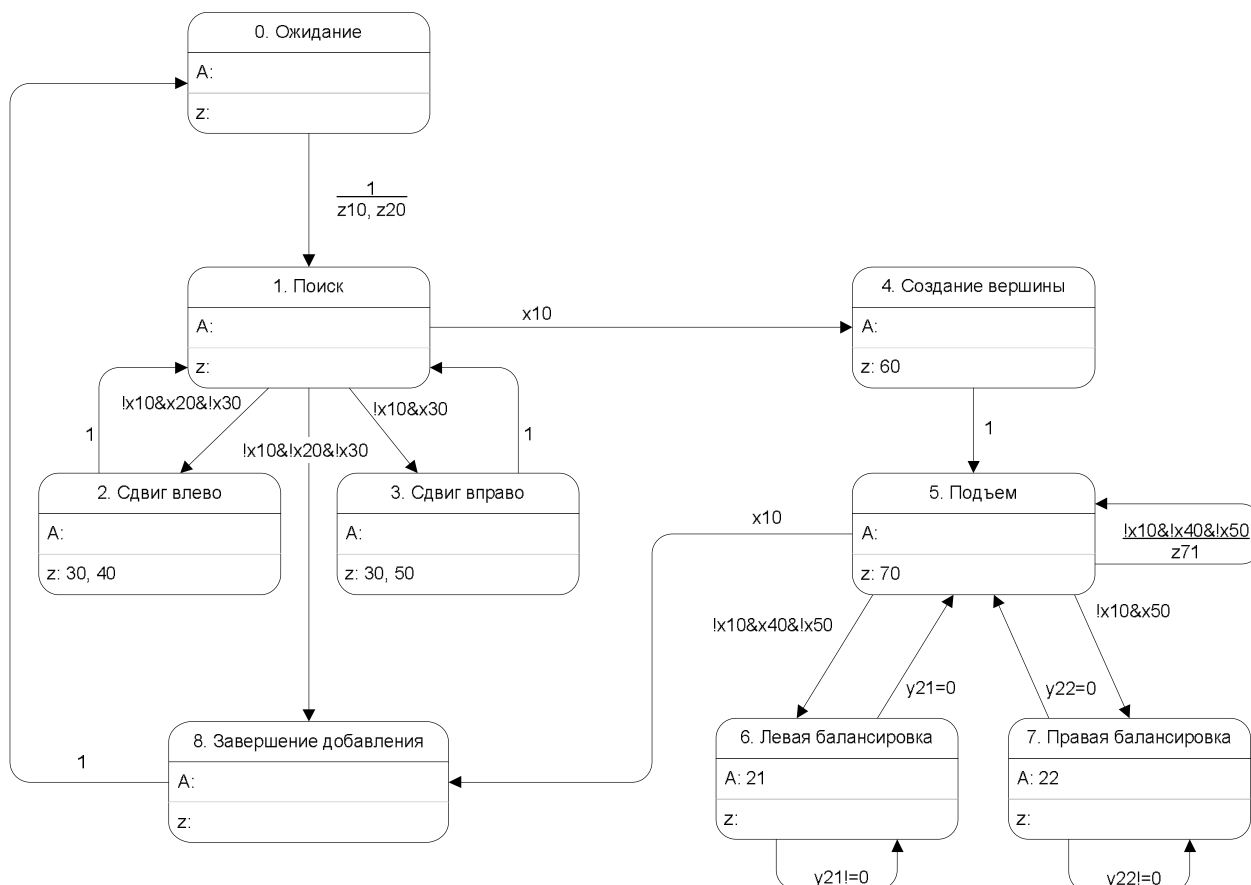

Рис. 17. Граф переходов автомата добавления элемента

Отметим, что новый вид автомата является гораздо более естественным и простым, так как с его помощью стала отчетливо видна структура алгоритма добавления элемента в АВЛ-дерево. Заметим также, что теперь при попадании в состояния 6 и 7 автомат вызывает автоматы балансировки.

Может показаться, что упрощение автомата мнимое, но это не так. Был применен типичный в программировании прием – «разделяй и властвуй». При этом задача была разбита на подзадачи, и тем самым снизилась вероятность ошибки.

Построим теперь автоматы балансировки. При этом потребуется два практически идентичных автомата – для левых и для правых балансировок. Процедуры балансировки достаточно просты и вынесение их содержания в автоматы нецелесообразно – это приведет к появлению большого числа дополнительных выходных воздействий автомата по перемещению различных указателей и изменения глубины узлов дерева. Однако, поскольку одной из целей исследования является визуализация операций с деревом, то введем выходные воздействия, осуществляющие «выделение» узлов, которые будут участвовать в операциях балансировки (рис. 18 – рис. 21). Все выходные воздействия (включая ранее введенные) приведены в табл. П1 приложения 1.

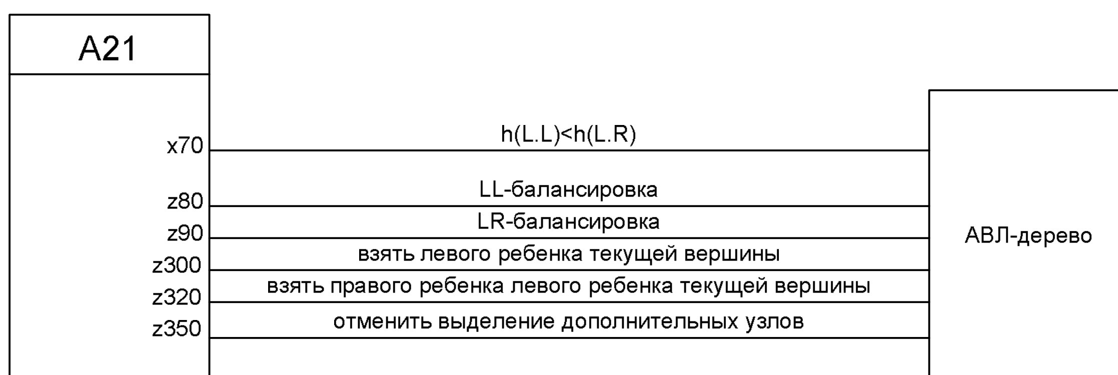

Рис. 18. Схема связей автомата левой балансировки

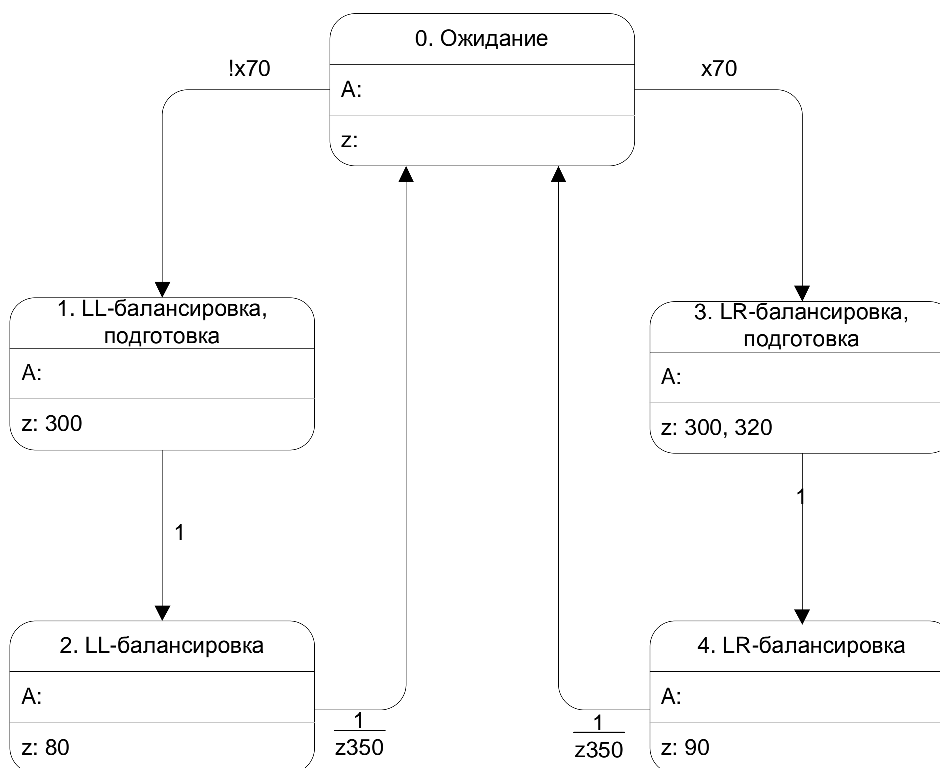

Рис. 19. Граф переходов автомата левой балансировки

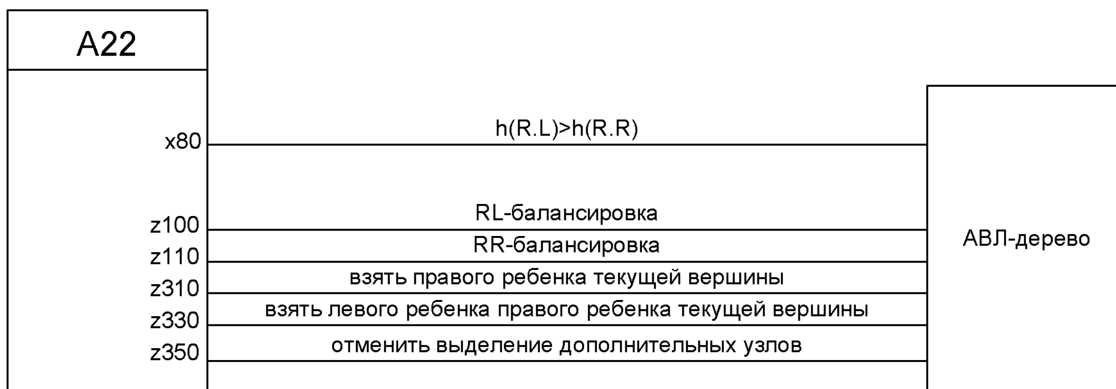

Рис. 20. Схема связей автомата правой балансировки

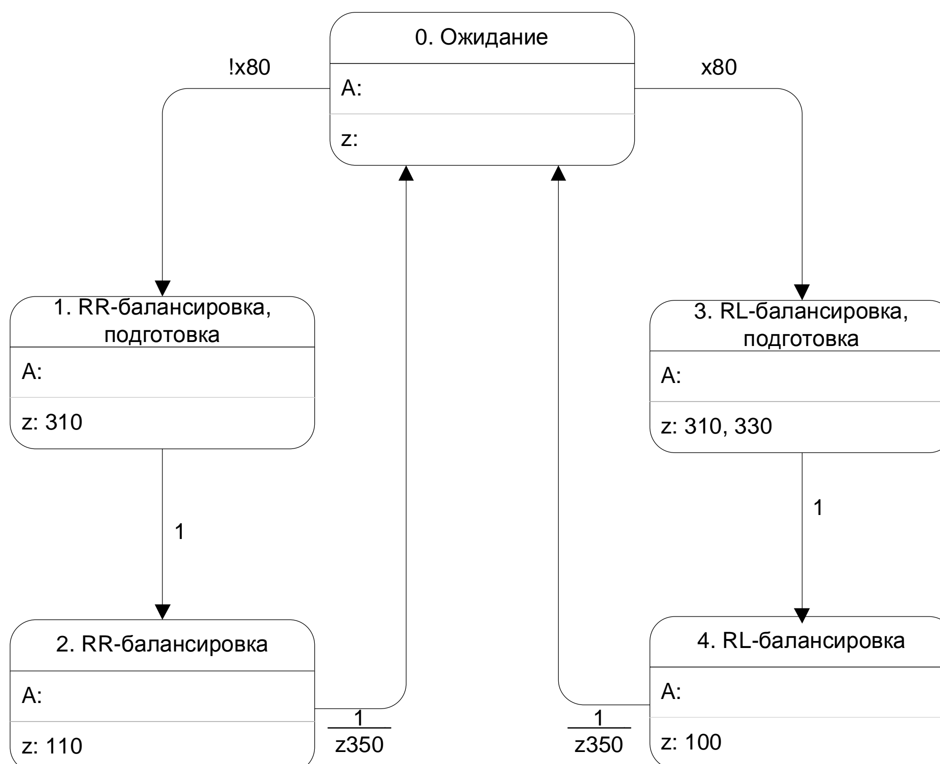

Рис. 21. Граф переходов автомата правой балансировки

2.4. *Построение автомата по неформальному описанию алгоритма*

Следующей операцией, которую необходимо реализовать, является удаление элемента из коллекции. При реализации поиска был выполнен переход от обычной программы к автомату, а при реализации добавления – построен автомат по формальному описанию алгоритма. При этом была найдена ошибка в этом формальном описании. Пойдем еще дальше – построим автомат по неформальному описанию операции удаления. Эта задача встречается наиболее часто.

Пусть дано следующее неформальное описание операции удаления.

Для удаления элемента следует сначала найти в дереве узел, его содержащий. Если такого узла нет, то ничего делать не требуется. Если этот узел является листом, то удалим его. Иначе, если у найденного узла нет левого поддерева, то удалим этот узел, поставив его правого сына на его место. Наконец, если у него есть левое поддерево, то найдем в нем узел X, содержащий максимальный элемент в этом поддереве. Переместим ключ из узла X на место ключа удаляемого элемента, а затем удалим узел X тем же способом. Возвращаясь вверх по дереву от последнего удаленного элемента, будем при необходимости осуществлять балансировку.

Построим автомат по этому описанию. Его основными состояниями будут следующие:

- поиск удаляемого элемента;
- удаление узла;
- поиск максимального элемента в поддереве;
- возвращение вверх по дереву.

Используем еще одну идею. До сих пор использовались смешанные автоматы (автоматы, в которых действия могли выполняться как в состояниях, так и на переходах). Как отмечалось выше, автоматы, в которых действия выполняются только на переходах, называются автоматами Мили. Автоматы Мили особенно удобны для визуализации поведения объектов, поскольку при этом все действия выполняются при переходе между состояниями, и состояние автомата отвечает стабильному состоянию объекта. Построим для реализации операции удаления автомат Мили.

Схема взаимодействий автомата удаления с объектом управления (*АВЛ*-деревом) показана на рис. 22, а сам автомат – на рис. 23. Полные таблицы всех входных переменных и выходных воздействий (табл. П2 и П3) приведены в приложении 1.

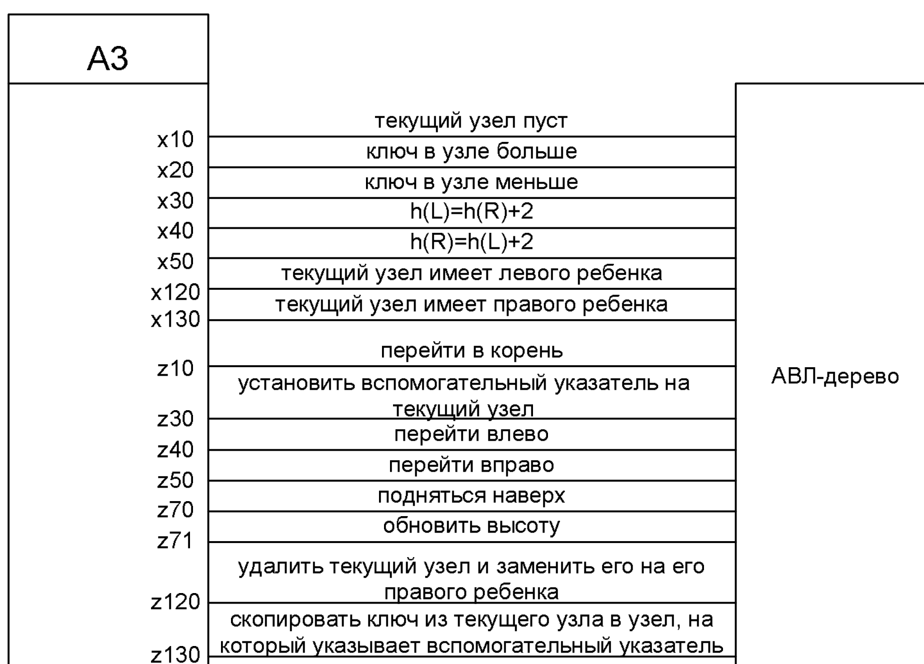

Рис. 22. Схема взаимодействий автомата удаления

Отметим, что в качестве вспомогательного указателя используется тот же указатель, что применялся в качестве указателя на текущего отца в алгоритме удаления. Это позволяет использовать уже определенное выходное воздействие $z30$.

Отметим также, что переход к автомату Мили позволил существенно уменьшить число состояний, но усложнил переходы.

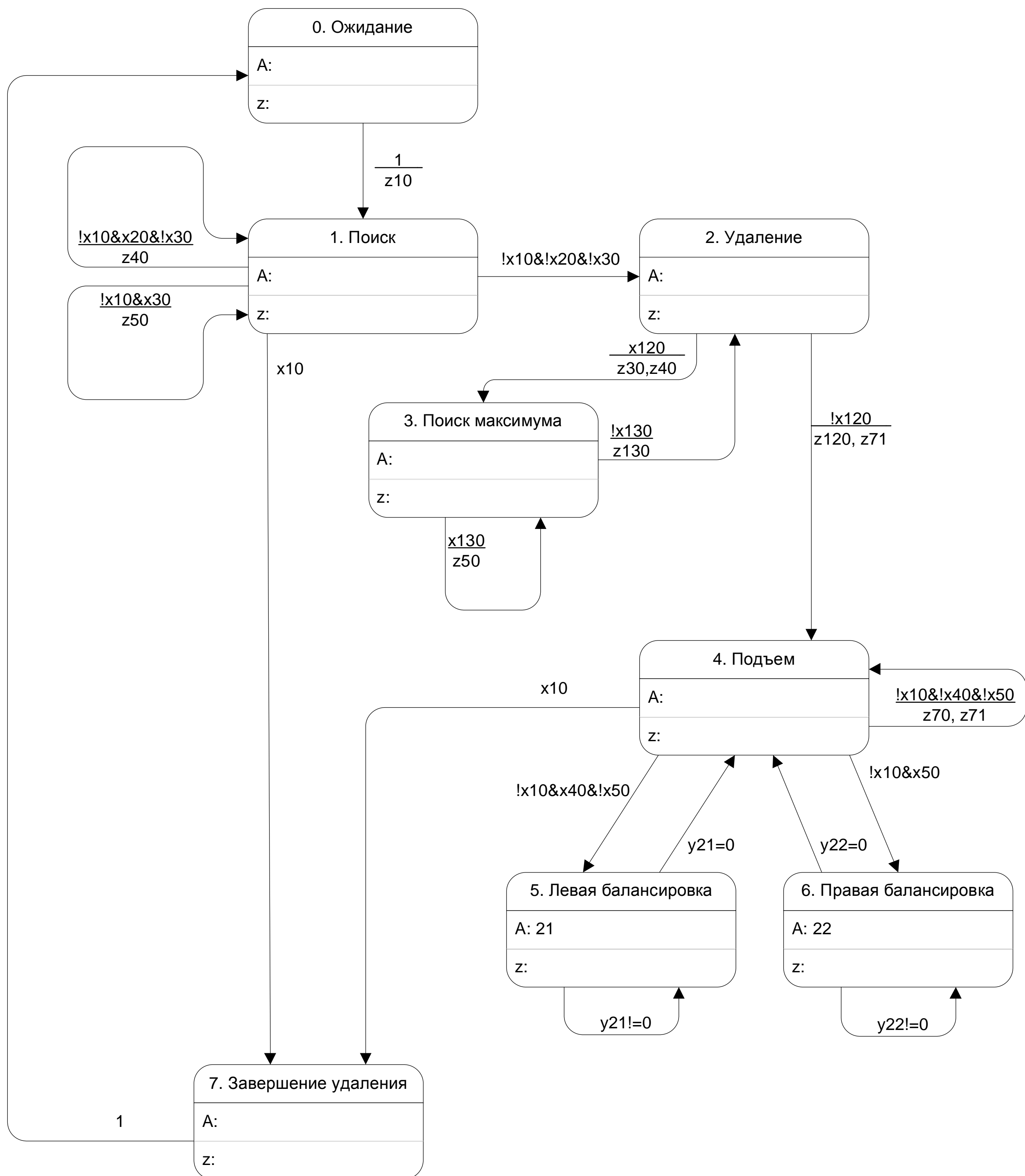

Рис. 23. Граф переходов автомата удаления

2.5. Построение главного автомата

Последнее, что осталось – построить главный автомат. Для этого вспомним первый автомат, который управлял черным ящиком. Преобразуем его в главный автомат, координирующий действия остальных автоматов. Вместо фиктивных выходных воздействий, которые выполняли операции с деревом, будем вызывать вложенные автоматы. В результате получается автомат, схема связей которого приведена на рис. 24, а граф переходов – на рис. 25.

Рис. 24. Схема связей главного автомата

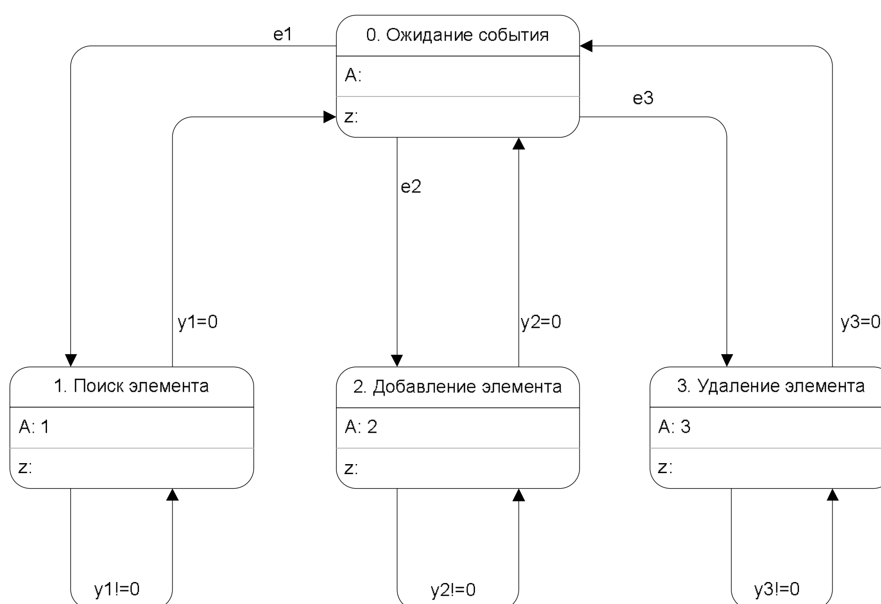

Рис. 25. Граф переходов главного автомата

3. *Реализация автоматов*

Итак, разработаны автоматы для всех операций с деревом (автомат поиска – рис. 12, автомат добавления – рис. 17, автомат удаления – рис. 23), разработан и главный автомат (рис. 25), координирующий их работу. Кроме того, были разработаны автоматы балансировки (рис. 19 и рис. 21), которые вложены в автоматы добавления и удаления.

Теперь необходимо преобразовать эти автоматы в программу. При этом требуется:

- реализовать объекты управления, реализовать у них методы, отвечающие входным переменным и выходным воздействиям автоматов;
- реализовать каждый автомат;
- реализовать внешнюю программу – генератор событий.

Первое требование означает, что необходимо написать класс, объект которого будет объектом управления (если объектов управления несколько, то требуется несколько классов). Это не вызывает проблемы, так как дерево и «результат операции», используемый при поиске, реализованы в виде классов `AVLTree` и `OperationResult`. Их полные исходные тексты приведены в приложении 2 (листинги 1 и 2).

Теперь реализуем автоматы. Каждый автомат будет реализован в виде класса – наследника стандартного класса `AbstractAutomata`. Диаграмма автоматных классов приведена на рис. 26.

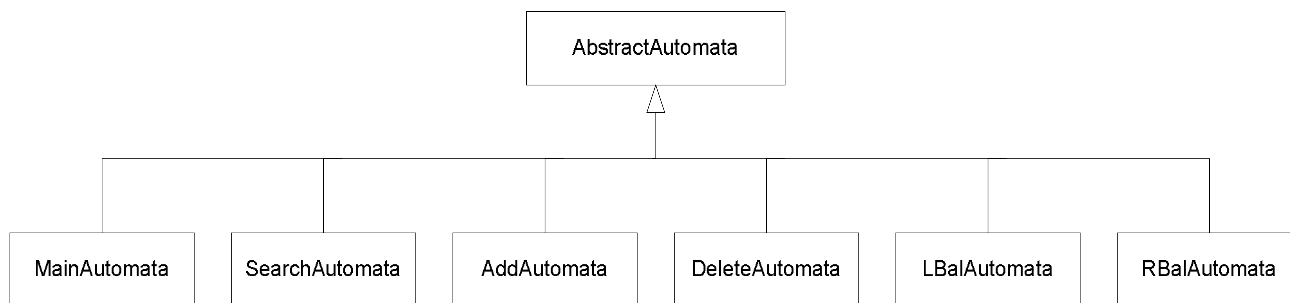

Рис. 26. Диаграмма автоматных классов

Переход из состояния в состояние реализуется методом `nextStep`. Он состоит из одного оператора `switch`, каждый вариант `case` в котором соответствует одному состоянию. При этом в каждом варианте последовательно проверяются условия на дугах, которые ведут из данного состояния, и при выполнении условия осуществляется переход. Сначала выполняются действия, соответствующие переходу (записанные на дуге на графах), а затем вызывается метод `switchState`, осуществляющий замену состояния. Этот метод изменяет текущее состояние и выполняет операции, определенные при входе в данное состояние.

При реализации автоматов, таким образом, возникают следующие проблемы:

– как обеспечить доступ автоматов к объектам управления;
– как обеспечить доступ автоматов к вложенным в них автоматам;
– как передавать автоматам события.

Предлагается следующее решение изложенных проблем. Для обеспечения доступа автоматов к объектам управления и вложенным автоматам используем параметры конструктора. Соответственно, первоначальное инстанциирование объектов управления и автоматов происходит в главном классе, внешнем по отношению к автоматам. Для передачи автоматам событий, создается очередь событий, которая также передается автоматам в качестве параметра конструктора. Автоматы, которым не требуется получать события, могут не иметь в конструкторе этот параметр. При наступлении события, его обработчик (находящийся во внешней программе) создает объект типа *событие* (`AutomataEvent`) и помещает его в очередь. При анализе условий на переходах автоматы, поведение которых зависит от событий, могут обращаться к очереди с целью выяснения, произошло ли то или иное событие.

Исходные тексты реализаций всех автоматов приведены в приложении 2, листинги 3–10.

Заключение

В настоящей работе рассмотрена задача управления АВЛ-деревом с помощью системы взаимодействующих автоматов. В процессе их построения постепенно выполнялась детализация от первоначальной абстракции, в которой дерево рассматривалось как «умный» черный ящик, до реализации, в которой дерево рассматривалось как сложный механизм, для управления которым применялись более мелкие операции. При этом была постепенно построена иерархия автоматов, взаимодействующих друг с другом. В результате задача управления оказалась разбита на множество небольших взаимосвязанных подзадач.

В заключение отметим, что хотя построенная реализация АВЛ-дерева не является оптимальной с точки зрения производительности, такой задачи и не ставилось. Ясно, что применение рассмотренного подхода для построения высокопроизводительных структур данных, где основная проблема не в сложной логике взаимодействия различных компонент, а в алгоритмической сложности самих структур, не оправдано. Тем не менее, явное выделение состояний в данном случае позволило подробно рассмотреть процесс работы с деревом и легко визуализировать работу с ним.

В задачах управления техническими объектами применение рассмотренного подхода построения программы с помощью конечных автоматов позволяет четко структурировать процесс управления и запрограммировать сложную логику взаимодействия различных составных частей программы. Рассмотренный в настоящей работе пример позволяет лишь изучить этот подход без

привлечения специальной предметной области. Управление более сложными объектами (дизель-генератором и танком) рассматривается в работах [8, 9].

Литература

1. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1.
2. *Ахо А., Сети Р., Ульман Дж.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2001.
3. *Хопкрофт Дж., Мотвани Р., Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
4. *Шалыто А. А., Туккель Н. И.* SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. № 5. http://is.ifmo.ru/works/switch/2.
5. *Шалыто А. А., Туккель Н. И.* Преобразование итеративных алгоритмов в автоматные // Программирование. 2002. № 5. http://is.ifmo.ru/works/iter/.
6. *Казаков М. А., Корнеев Г. А., Шалыто А. А.* Разработка логики визуализаторов алгоритмов на основе конечных автоматов // Телекоммуникация и информатизация образования. 2003, № 6. http://is.ifmo.ru/works/vis/.
7. *Кларк Э. М., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
8. *Туккель Н. И., Шалыто А. А.* Система управления дизель-генератором. Программирование с явным выделением состояний. Программная документация. http://is.ifmo.ru/projects/dg/.
9. *Туккель Н. И., Шалыто А. А.* Система управления танком для игры «Robocode». Объектно-ориентированное программирование с явным выделением состояний. Программная документация. http://is.ifmo.ru/projects/tanks/.
10. *Корнеев Г. А.* Технология разработки визуализаторов алгоритмов // Труды II межвузовской конференции молодых ученых. СПб.: СПбГУ ИТМО, 2005. http://is.ifmo.ru/works/_a_process.pdf.
11. *Вирт Н.* Алгоритмы и структуры данных. СПб.: Невский диалект. 2001.
12. *Адельсон-Вельский Г. М., Ландис Е. М.* Один алгоритм организации информации // Доклады АН СССР. 1962. Вып. 2.
13. *Керниган Б., Пайк Р.* Практика программирования. СПб.: Невский Диалект. 2001.
14. *Кузнецов Б. П.* Психология автоматного программирования // BYTE/Россия. 2000. № 11. http://www.avrorasystems.com/235/016012001004/1/.

Об авторах

Дворкин Михаил Эдуардович – студент кафедры «Компьютерные технологии» Санкт-Петербургского государственного университета информационных технологий, механики и оптики (СПбГУ ИТМО), обладатель золотой медали чемпионата мира по программированию ACM ICPC 2006 года.

Станкевич Андрей Сергеевич – старший преподаватель кафедры «Компьютерные технологии» СПбГУ ИТМО, лауреат премии Президента Российской Федерации в области образования 2003 года, обладатель золотой медали чемпионата мира по программированию ACM ICPC 2001 года.

Шалыто Анатолий Абрамович – доктор технических наук, профессор, заведующий кафедрой «Технологии программирования» СПбГУ ИТМО.

Приложение 1. Сводные таблицы входных переменных автоматов, выходных воздействий автоматов и событий

Таблица П1. Входные переменные автоматов

Вход	Значение
*x*10	Текущий узел есть пустая ссылка
*x*20	Значение ключа в текущем узле больше значения ключа-параметра
*x*30	Значение ключа в текущем узле меньше значения ключа-параметра
*x*40	Высота левого поддерева текущего узла больше высоты правого поддерева текущего узла на 2
*x*50	Высота правого поддерева текущего узла больше высоты левого поддерева текущего узла на 2
*x*60	Высота левого поддерева левого поддерева больше высоты правого поддерева левого поддерева
*x*70	Высота правого поддерева левого поддерева больше высоты левого поддерева левого поддерева
*x*80	Высота левого поддерева правого поддерева больше высоты левого поддерева правого поддерева
*x*90	Высота правого поддерева правого поддерева больше высоты правого поддерева правого поддерева
*x*120	Текущий узел имеет левого ребенка
*x*130	Текущий узел имеет правого ребенка

Таблица П2. Выходные воздействия автоматов

Выход	Смысл	Действие
*z*10	Перейти в корень	Текущий узел ← корень дерева
*z*20	Обнулить текущего отца	Текущий отец ← пустая ссылка
*z*30	Спуститься вниз	Текущий отец ← текущий узел
*z*40	Пойти налево	Текущий узел ← левый сын текущего узла
*z*50	Пойти направо	Текущий узел ← правый сын текущего узла
*z*60	Создать новый узел	Создать в текущем узле новый узел в соответствии с шагом II
*z*70	Подняться наверх	Текущий узел ← отец текущего узла
*z*71	Обновить высоту	Установить высоту текущего узла как максимум высот его детей
*z*80	*LL*-балансировка	Осуществить *LL*-балансировку текущего узла
*z*90	*LR*-балансировка	Осуществить *LR*-балансировку текущего узла
*z*100	*RL*-балансировка	Осуществить *RL*-балансировку текущего узла
*z*110	*RR*-балансировка	Осуществить *RR*-балансировку текущего узла
*z*120	Удаление узла	Удалить текущий узел и заменить его на его правого ребенка
*z*130	Перенос узла	Скопировать узел из текущего узла в узел, на который указывает вспомогательный указатель
*z*190	Сбросить результат	Установить результат операции как неопределенный
*z*200	Элемент найден	Сообщить оператору, что элемент найден
*z*210	Элемент не найден	Сообщить оператору, что элемент не найден
*z*300	Выделить L	• Взять левого ребенка текущей вершины
*z*310	Выделить R	Взять правого ребенка текущей вершины
*z*320	Выделить L.R	Взять правого ребенка левого ребенка текущей

		вершины
$z330$	Выделить R.L	Взять левого ребенка правого ребенка текущей вершины
$z350$	Отменить выделение	Отменить выделение дополнительных узлов

Таблица П3. События

Событие	**Значение**
$e1$	Поиск значения в дереве
$e2$	Добавление элемента в дерево
$e3$	Удаление элемента из дерева

Приложение 2. Тексты программ

Листинг 1. Объект управления «дерево» (AVLTree.java)

```
package ru.ifmo.ips.visualizers.avltree;

public class AVLTree {
    public static class Node {
        Node left;
        Node right;
        Node parent;

        int key;

        int depth;
    }

    public Node root;

    public Node currentNode;
    public Node currentParent;
    public Node selectedNode;
    public Node selectedNode2;
    public int direction;

    public int pendingKey;

    public AVLTree() {
        root = null;
    }

    private int getDepth(Node node) {
        if (node == null) {
            return 0;
        } else {
            return node.depth;
        }
    }

    private void updateDepth(Node node) {
        int ld = getDepth(node.left);
        int rd = getDepth(node.right);
        if (ld > rd) {
            node.depth = ld + 1;
        } else {
            node.depth = rd + 1;
        }
    }

    private void lchild(Node parent, Node child) {
        if (parent != null) {
            parent.left = child;
        } else {
            root = child;
        }
        if (child != null) {
            child.parent = parent;
        }
    }

    private void rchild(Node parent, Node child) {
        if (parent != null) {
            parent.right = child;
        } else {
            root = child;
```

```
        }
        if (child != null) {
            child.parent = parent;
        }
    }

    private void llBal(Node nodeA) {
        Node parent = nodeA.parent;
        int which = 0;
        if (parent != null) {
            if (nodeA == parent.right) {
                which = 1;
            }
        }

        Node nodeB = nodeA.left;
        lchild(nodeA, nodeB.right);
        rchild(nodeB, nodeA);
        if (which == 0) {
            lchild(parent, nodeB);
        } else {
            rchild(parent, nodeB);
        }
        updateDepth(nodeA);
        updateDepth(nodeB);
    }

    private void rrBal(Node nodeA) {
        Node parent = nodeA.parent;
        int which = 0;
        if (parent != null) {
            if (nodeA == parent.right) {
                which = 1;
            }
        }

        Node nodeB = nodeA.right;
        rchild(nodeA, nodeB.left);
        lchild(nodeB, nodeA);
        if (which == 0) {
            lchild(parent, nodeB);
        } else {
            rchild(parent, nodeB);
        }
        updateDepth(nodeA);
        updateDepth(nodeB);
    }

    private void lrBal(Node nodeA) {
        Node parent = nodeA.parent;
        int which = 0;
        if (parent != null) {
            if (nodeA == parent.right) {
                which = 1;
            }
        }

        Node nodeB = nodeA.left;
        Node nodeC = nodeB.right;
        lchild(nodeA, nodeC.right);
        rchild(nodeC, nodeA);
        rchild(nodeB, nodeC.left);
        lchild(nodeC, nodeB);
        if (which == 0) {
            lchild(parent, nodeC);
        } else {
```

```
            rchild(parent, nodeC);
        }
        updateDepth(nodeA);
        updateDepth(nodeB);
        updateDepth(nodeC);
    }

    private void rlBal(Node nodeA) {
        Node parent = nodeA.parent;
        int which = 0;
        if (parent != null) {
            if (nodeA == parent.right) {
                which = 1;
            }
        }

        Node nodeB = nodeA.right;
        Node nodeC = nodeB.left;
        rchild(nodeA, nodeC.left);
        lchild(nodeC, nodeA);
        lchild(nodeB, nodeC.right);
        rchild(nodeC, nodeB);
        if (which == 0) {
            lchild(parent, nodeC);
        } else {
            rchild(parent, nodeC);
        }
        updateDepth(nodeA);
        updateDepth(nodeB);
        updateDepth(nodeC);
    }

    /* Automata interface functions */

    /* Automata inputs */

    public boolean x10() {
        return currentNode == null;
    }

    public boolean x20() {
        return currentNode.key > pendingKey;
    }

    public boolean x30() {
        return currentNode.key < pendingKey;
    }

    public boolean x40() {
        return getDepth(currentNode.left) == getDepth(currentNode.right) + 2;
    }

    public boolean x50() {
        return getDepth(currentNode.right) == getDepth(currentNode.left) + 2;
    }

    public boolean x70() {
        return getDepth(currentNode.left.left) < getDepth(currentNode.left.right);
    }

    public boolean x80() {
        return getDepth(currentNode.right.left) > getDepth(currentNode.right.right);
    }

    public boolean x120() {
        return currentNode.left != null;
```

```
    }

    public boolean x130() {
        return currentNode.right != null;
    }

    /* Automata outputs */

    public void z10() {
        currentNode = root;
    }

    public void z20() {
        currentParent = null;
    }

    public void z30() {
        currentParent = currentNode;
    }

    public void z40() {
        direction = 0;
        currentNode = currentNode.left;
    }

    public void z50() {
        direction = 1;
        currentNode = currentNode.right;
    }

    public void z60() {
        currentNode = new Node();
        currentNode.parent = currentParent;
        currentNode.left = null;
        currentNode.right = null;
        currentNode.key = pendingKey;
        currentNode.depth = 1;
        if (currentParent != null) {
            if (direction == 0) {
                currentParent.left = currentNode;
            } else {
                currentParent.right = currentNode;
            }
        } else {
            root = currentNode;
        }
    }

    public void z70() {
        currentNode = currentNode.parent;
    }

    public void z71() {
        if (currentNode != null) {
            updateDepth(currentNode);
        }
    }

    public void z80() {
        llBal(currentNode);
        currentNode = currentNode.parent;
    }

    public void z90() {
        lrBal(currentNode);
        currentNode = currentNode.parent;
```

```
    }

    public void z100() {
        rlBal(currentNode);
        currentNode = currentNode.parent;
    }

    public void z110() {
        rrBal(currentNode);
        currentNode = currentNode.parent;
    }

    public void z120() {
        Node parent = currentNode.parent;
        if (parent != null && parent.left == currentNode) {
            lchild(parent, currentNode.right);
        } else {
            rchild(parent, currentNode.right);
        }
        currentNode = parent;
    }

    public void z130() {
        currentParent.key = currentNode.key;
    }

    public void z300() {
        selectedNode = currentNode.left;
    }

    public void z310() {
        selectedNode = currentNode.right;
    }

    public void z320() {
        selectedNode2 = currentNode.left.right;
    }

    public void z330() {
        selectedNode2 = currentNode.right.left;
    }

    public void z350() {
        selectedNode = null;
        selectedNode2 = null;
    }
}
```

Листинг 2. Объект управления «результат операции» (OperationResult.java)

```
package ru.ifmo.ips.visualizers.avltree;

public class OperationResult {
    public int result;

    public static final int RESULT_UNDEFINED = 0;
    public static final int RESULT_SUCCESS = 1;
    public static final int RESULT_FAIL = 2;

    public void z190() {
        result = RESULT_UNDEFINED;
    }

    public void z200() {
        result = RESULT_SUCCESS;
    }
```

```
    public void z210() {
        result = RESULT_FAIL;
    }

}
```

AbstractAutomata.java

```
package ru.ifmo.ips.visualizers.avltree;

import java.util.LinkedList;

public abstract class AbstractAutomata {
    protected int state;

    private LinkedList eventQueue;

    public AbstractAutomata() {
        state = 0;
        eventQueue = new LinkedList();
    }

    public void pushEvent(AutomataEvent event) {
        eventQueue.add(event);
    }

    protected AutomataEvent peekEvent() {
        if (eventQueue.isEmpty()) {
            return null;
        } else {
            AutomataEvent event = (AutomataEvent) eventQueue.get(0);
            return event;
        }
    }

    protected AutomataEvent popEvent() {
        if (eventQueue.isEmpty()) {
            return null;
        } else {
            AutomataEvent event = (AutomataEvent) eventQueue.removeFirst();
            return event;
        }
    }

    public int getState() {
        return state;
    }

    public void invokeChildAutomata(AbstractAutomata a) {
        a.nextStep();
    }

    public void switchState(int newState) {
        state = newState;
    }

    public abstract void nextStep();
}
```

AutomataEvent.java

```
package ru.ifmo.ips.visualizers.avltree;

public class AutomataEvent {
    private final int type;
```

```
    private final Object value;

    public AutomataEvent(int eventType, Object value) {
        this.type = eventType;
        this.value = value;
    }

    public int getType() {
        return type;
    }

    public Object getValue() {
        return value;
    }
}
```

Внутренняя логика автомата сосредоточена в методах `nextStep()`, который осуществляет переход по дуге, и `switchState()`, в котором реализованы действия, происходящие при попадании в состояние – вызов вложенных автоматов и выходные воздействия. Автомат получает вложенные в него автоматы от главного класса `AVLTreeApplet`, который будет рассмотрен ниже. События, поступающие в автоматы, содержатся с объектах специального класса `AutomataEvent`.

MainAutomata.java

```
package ru.ifmo.ips.visualizers.avltree;

public class MainAutomata extends AbstractAutomata {
    private final AddAutomata addAutomata;
    private final SearchAutomata searchAutomata;
    private final DeleteAutomata deleteAutomata;

    private final AVLTree tree;

    public MainAutomata(AVLTree tree) {
        this.tree = tree;
        addAutomata = AVLTreeApplet.addAutomata;
        searchAutomata = AVLTreeApplet.searchAutomata;
        deleteAutomata = AVLTreeApplet.deleteAutomata;
    }

    public void nextStep() {
        AutomataEvent event = peekEvent();
        switch (state) {
            case 0: {
                if (event != null) {
                    switch (event.getType()) {
                        case 1: {
                            popEvent();
                            tree.pendingKey = ((Integer)event.getValue()).intValue();
                            switchState(1);
                            break;
                        }
                        case 2: {
                            popEvent();
                            tree.pendingKey = ((Integer)event.getValue()).intValue();
                            switchState(2);
                            break;
                        }
                        case 3: {
                            popEvent();
                            tree.pendingKey = ((Integer)event.getValue()).intValue();
                            switchState(3);
                            break;
```

```
                    }
                }
            }
            break;
        }
        case 1: {
            if (searchAutomata.getState() == 0) {
                switchState(0);
            } else {
                switchState(1);
            }
            break;
        }
        case 2: {
            if (addAutomata.getState() == 0) {
                switchState(0);
            } else {
                switchState(2);
            }
            break;
        }
        case 3: {
            if (deleteAutomata.getState() == 0) {
                switchState(0);
            } else {
                switchState(3);
            }
            break;
        }
    }
}

public void switchState(int newState) {
    super.switchState(newState);

    switch (state) {
        case 0: {
            break;
        }
        case 1: {
            invokeChildAutomata(searchAutomata);
            break;
        }
        case 2: {
            invokeChildAutomata(addAutomata);
            break;
        }
        case 3: {
            invokeChildAutomata(deleteAutomata);
            break;
        }
    }
}
}
```

SearchAutomata.java

```
package ru.ifmo.ips.visualizers.avltree;

public class SearchAutomata extends AbstractAutomata {
    private final AVLTree tree;

    public SearchAutomata(AVLTree tree) {
        this.tree = tree;
    }
```

```
public void nextStep() {
    switch (state) {
        case 0: {
            tree.z10();
            tree.z190();
            switchState(1);
            break;
        }
        case 1: {
            if (tree.x10()) {
                switchState(5);
            } else if (!tree.x10() && tree.x20()) {
                switchState(2);
            } else if (!tree.x10() && tree.x30()) {
                switchState(3);
            } else if (!tree.x10() && !tree.x20() && !tree.x30()) {
                switchState(4);
            } else {
            }
            break;
        }
        case 2: {
            switchState(1);
            break;
        }
        case 3: {
            switchState(1);
            break;
        }
        case 4: {
            switchState(0);
            break;
        }
        case 5: {
            switchState(0);
            break;
        }
    }
}

public void switchState(int newState) {
    super.switchState(newState);

    switch (state) {
        case 0: {
            break;
        }
        case 1: {
            break;
        }
        case 2: {
            tree.z40();
            break;
        }
        case 3: {
            tree.z50();
            break;
        }
        case 4: {
            tree.z200();
            break;
        }
        case 5: {
            tree.z210();
            break;
        }
```

```
        }
    }
}
```

AddAutomata.java

```
package ru.ifmo.ips.visualizers.avltree;

public class AddAutomata extends AbstractAutomata {
    private final AVLTree tree;

    private final LBalAutomata lBalAutomata;
    private final RBalAutomata rBalAutomata;

    public AddAutomata(AVLTree tree) {
        this.tree = tree;

        lBalAutomata = AVLTreeApplet.lBalAutomata;
        rBalAutomata = AVLTreeApplet.rBalAutomata;
    }

    public void nextStep() {
        switch (state) {
            case 0: {
                tree.z10();
                tree.z20();
                switchState(1);
                break;
            }
            case 1: {
                if (tree.x10()) {
                    switchState(4);
                } else if (!tree.x10() && tree.x20()) {
                    switchState(2);
                } else if (!tree.x10() && tree.x30()) {
                    switchState(3);
                } else if (!tree.x10() && !tree.x20() && !tree.x30()) {
                    switchState(8);
                } else {
                }
                break;
            }
            case 2: {
                switchState(1);
                break;
            }
            case 3: {
                switchState(1);
                break;
            }
            case 4: {
                switchState(5);
                break;
            }
            case 5: {
                if (tree.x10()) {
                    switchState(8);
                } else if (!tree.x10() && tree.x40()) {
                    switchState(6);
                } else if (!tree.x10() && tree.x50()) {
                    switchState(7);
                } else if (!tree.x10() && !tree.x40() && !tree.x50()) {
                    tree.z71();
                    switchState(5);
```

```
            } else {
            }
            break;
        }
        case 6: {
            if (lBalAutomata.getState() == 0) {
                switchState(5);
            } else {
                switchState(6);
            }
            break;
        }
        case 7: {
            if (rBalAutomata.getState() == 0) {
                switchState(5);
            } else {
                switchState(7);
            }

            break;
        }
        case 8: {
            switchState(0);
            break;
        }
    }
}

public void switchState(int newState) {
    super.switchState(newState);
    switch (state) {
        case 0: {
            break;
        }
        case 1: {
            break;
        }
        case 2: {
            tree.z30();
            tree.z40();
            break;
        }
        case 3: {
            tree.z30();
            tree.z50();
            break;
        }
        case 4: {
            tree.z60();
            break;
        }
        case 5: {
            tree.z70();
            break;
        }
        case 6: {
            invokeChildAutomata(lBalAutomata);
            break;
        }
        case 7: {
            invokeChildAutomata(rBalAutomata);
            break;
        }
        case 8: {
            break;
        }
```

```
        }
    }
}
```

LBalAutomata.java

```
package ru.ifmo.ips.visualizers.avltree;

public class LBalAutomata extends AbstractAutomata {
    private final AVLTree tree;

    public LBalAutomata(AVLTree tree) {
        this.tree = tree;
    }

    public void nextStep() {
        switch (state) {
            case 0: {
                if (!tree.x70()) {
                    switchState(1);
                } else if (tree.x70()) {
                    switchState(3);
                }
                break;
            }
            case 1: {
                switchState(2);
                break;
            }
            case 2: {
                switchState(0);
                break;
            }
            case 3: {
                switchState(4);
                break;
            }
            case 4: {
                switchState(0);
                break;
            }
        }
    }

    public void switchState(int newState) {
        super.switchState(newState);
        switch (state) {
            case 0: {
                break;
            }
            case 1: {
                tree.z300();
                break;
            }
            case 2: {
                tree.z80();
                tree.z350();
                break;
            }
            case 3: {
                tree.z300();
                tree.z320();
                break;
            }
            case 4: {
```

```
                tree.z90();
                tree.z350();
                break;
            }
        }
    }
}
```

RBalAutomata.java

```
package ru.ifmo.ips.visualizers.avltree;

public class RBalAutomata extends AbstractAutomata {
    private final AVLTree tree;

    public RBalAutomata(AVLTree tree) {
        this.tree = tree;
    }

    public void nextStep() {
        switch (state) {
            case 0: {
                if (!tree.x80()) {
                    switchState(1);
                } else if (tree.x80()) {
                    switchState(3);
                }
                break;
            }
            case 1: {
                switchState(2);
                break;
            }
            case 2: {
                switchState(0);
                break;
            }
            case 3: {
                switchState(4);
                break;
            }
            case 4: {
                switchState(0);
                break;
            }
        }
    }

    public void switchState(int newState) {
        super.switchState(newState);
        switch (state) {
            case 0: {
                break;
            }
            case 1: {
                tree.z310();
                break;
            }
            case 2: {
                tree.z110();
                tree.z350();
                break;
            }
            case 3: {
```

```
                tree.z310();
                tree.z330();
                break;
            }
            case 4: {
                tree.z100();
                tree.z350();
                break;
            }
        }
    }
}
```

DeleteAutomata.java

```
package ru.ifmo.ips.visualizers.avltree;

public class DeleteAutomata extends AbstractAutomata {
    private final AVLTree tree;

    private final LBalAutomata lBalAutomata;
    private final RBalAutomata rBalAutomata;

    public DeleteAutomata(AVLTree tree) {
        this.tree = tree;

        lBalAutomata = AVLTreeApplet.lBalAutomata;
        rBalAutomata = AVLTreeApplet.rBalAutomata;
    }
    public void nextStep() {
        switch (state) {
            case 0: {
                tree.z10();
                switchState(1);
                break;
            }
            case 1: {
                if (tree.x10()) {
                    switchState(7);
                } else if (!tree.x10() && tree.x20() && !tree.x30()) {
                    tree.z40();
                    switchState(1);
                } else if (!tree.x10() && tree.x30()) {
                    tree.z50();
                    switchState(1);
                } else if (!tree.x10() && !tree.x20() && !tree.x30()) {
                    switchState(2);
                } else {
                }
                break;
            }
            case 2: {
                if (tree.x120()) {
                    tree.z30();
                    tree.z40();
                    switchState(3);
                } else if (!tree.x120()) {
                    tree.z120();
                    tree.z71();
                    switchState(4);
                }
                break;
            }
            case 3: {
```

```
            if (tree.x130()) {
                tree.z50();
                switchState(3);
            } else if (!tree.x130()) {
                tree.z130();
                switchState(2);
            }
            break;
        }
        case 4: {
            if (tree.x10()) {
                switchState(7);
            } else if (!tree.x10() && tree.x40() && !tree.x50()) {
                switchState(5);
            } else if (!tree.x10() && tree.x50()) {
                switchState(6);
            } else if (!tree.x10() && !tree.x40() && !tree.x50()) {
                tree.z70();
                tree.z71();
                switchState(4);
            }
            break;
        }
        case 5: {
            if (lBalAutomata.getState() == 0) {
                switchState(4);
            } else {
                switchState(5);
            }
            break;
        }
        case 6: {
            if (rBalAutomata.getState() == 0) {
                switchState(4);
            } else {
                switchState(6);
            }

            break;
        }
        case 7: {
            switchState(0);
            break;
        }
    }
}

public void switchState(int newState) {
    super.switchState(newState);

    switch (state) {
        case 0: {
            break;
        }
        case 1: {
            break;
        }
        case 2: {
            break;
        }
        case 3: {
            break;
        }
        case 4: {
            break;
        }
```

```
            case 5: {
                invokeChildAutomata(lBalAutomata);
                break;
            }
            case 6: {
                invokeChildAutomata(rBalAutomata);
                break;
            }
            case 7: {
                break;
            }
        }
    }
}
```