МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра безопасности информационных систем

А. Н. Крутов, М. Е. Федина

МЕТОДЫ ПРОГРАММИРОВАНИЯ

Утверждено редакционно-издательским советом университета в качестве учебного пособия

Самара
Издательство «Самарский университет»
2014

УДК 681.3.07
ББК 32.973.26
К 64

Рецензенты: канд. физ.-мат. наук, доцент М. Н. Осипов,
д-р физ.-мат. наук, профессор В. И. Астафьев

Крутов, А. Н.

К 64 Методы программирования : учеб. пособие / А. Н. Крутов, М. Е. Федина – Самара : Изд-во «Самарский университет», 2014. – 48 с.

В учебном пособии описываются основные структуры данных, применяемые в программировании, обсуждаются наиболее распространенные в настоящее время алгоритмы обработки информации (сортировка, поиск и др.). В конце пособия предлагается список задач для лабораторных работ по курсу.

Предназначено для студентов специальности 10.05.01 – Компьютерная безопасность.

УДК 681.3.07
ББК 32.973.26

Содержание

Введение

В настоящее время компьютерная техника развивается стремительными темпами. Однако ключевая роль в достижении успеха большинства компьютеризованных систем принадлежит не используемому оборудованию, а программному обеспечению. Поэтому задача разработки качественных и надежных программных продуктов с использованием оптимальных алгоритмов никогда не перестанет быть актуальной.

Данное пособие посвящено описанию алгоритмов, ставших уже классическими, несмотря на относительно небольшую историю развития информатики и программирования.

В первой главе описываются основные структуры данных, применяемые в программировании. К ним относятся массивы, списки, стеки, очереди, а также деревья.

Вторая глава посвящена описанию алгоритмов сортировки.

В третьей главе приводятся алгоритмы поиска информации в упорядоченных и неупорядоченных массивах.

В конце пособия предлагается список задач для лабораторных работ по курсу.

При подготовке данного учебного пособия активно использовалась литература, приведенная в библиографическом списке, а также материалы internet-сайтов, таких как http://www.citforum.ru и http://algoritm.manual.ru. В начале каждой темы дается ссылка на первоисточник с тем, чтобы студент в случае необходимости смог обратиться за дополнительной информацией.

Учебное пособие предназначено для студентов специальности 10.05.01 – Компьютерная безопасность.

1. Структуры данных

Структуры данных и алгоритмы служат теми материалами, из которых строятся программы [1].

Под **структурой данных** в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты. Поэтому вводится дополнительная классификация структур данных, направления которой соответствуют различным аспектам их рассмотрения.

Понятие "**физическая структура данных**" отражает способ физического представления данных в памяти машины и называется еще структурой хранения, внутренней структурой или структурой памяти. В настоящем пособии физическая структура данных не рассматривается.

Структура данных без учета ее представления в машинной памяти называется абстрактной или **логической**. В общем случае между логической и, соответствующей ей, физической структурами существует различие, степень которого зависит от самой структуры и особенностей той среды, в которой она должна быть отражена.

Различают **простые** (базовые, примитивные) структуры (типы) данных и **интегрированные** (структурированные, композитные, сложные).

Простыми называются такие структуры данных, которые не могут быть расчленены на составные части, большие, чем биты. С логической точки зрения простые данные являются неделимыми единицами.

Интегрированными называются такие структуры данных, составными частями которых являются другие структуры данных – простые или интегрированные. Интегрированные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

[1] Вводная часть данной темы составлена на основе материалов монографии [3].

Одним из наиболее важных признаков структуры данных является ее **изменчивость** - изменение числа элементов и (или) связей между элементами структуры. В определении изменчивости структуры не отражен факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости.

По признаку изменчивости различают **простые базовые** структуры, структуры **статические**, **полустатические, динамические и файловые**. Классификация структур данных по признаку изменчивости приведена на рис. 1.

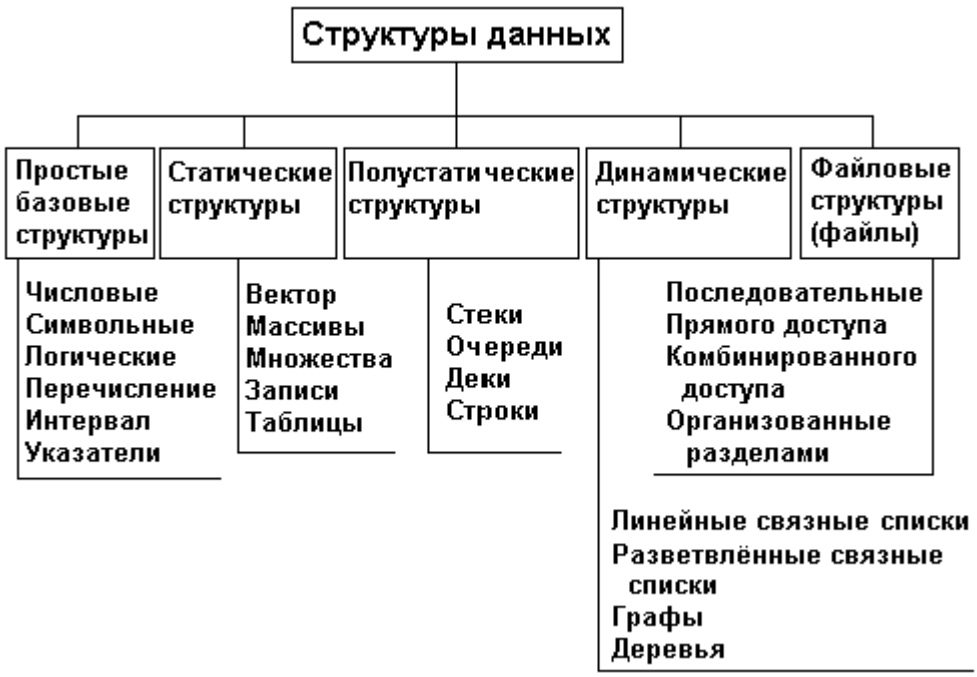

Рис. 1. Классификация структур данных

Базовые структуры данных, статические, полустатические и динамические характерны для оперативной памяти и часто называются **оперативными структурами**. Файловые структуры соответствуют структурам данных для внешней памяти.

Далее приводится описание интегрированных структур данных, широко используемых на практике при реализации тех или иных алгоритмов.

1.1. Треугольные массивы [2]

На практике часто приходится иметь дело с треугольными массивами. Примером могут служить операции с симметричными матрицами, которые часто появляются в сетевых задачах и задачах упругости. Если хранить соответствующую структуру данных в двумерном массиве, это вызовет излишний расход машинной памяти. Так, при размере матрицы в 1000 строк придется хранить более полумиллиона ненужных элементов.

Оптимальным выходом в данной ситуации является создание одномерного массива *B* и упаковка в него значимых элементов массива *A* (рис. 2). Формула для преобразования *A[i,j]* в *B[x]* имеет вид: $x=[i(i-1)/2+j]$ для $i>j$. Выражение в квадратных скобках в данной формуле означает операцию взятия целой части числа.

Массив *A*:

A[1,0]			
A[2,0]	*A[2,1]*		
A[3,0]	*A[3,1]*	*A[3,2]*	

Массив *B*:

A[1,0]	*A[2,0]*	*A[2,1]*	*A[3,0]*	*A[3,1]*	*A[3,2]*

Рис. 2. Упаковка треугольного массива в одномерный массив

Если треугольный массив содержит ненулевые диагональные элементы, то для его хранения в одномерном массиве можно также использовать предыдущую формулу, если перед ее применением увеличить значение индекса *i* на единицу.

[2] Составлено на основе материалов монографии [6].

1.2. Списки [3]

Простые списки

Если в программе нужен список постоянного размера, проще всего его реализовать при помощи массива. В этом случае можно легко исследовать элементы списка в цикле. Во многих программах приходится иметь дело со списками, размер которых может постоянно изменяться. Можно создать массив, размер которого будет максимально возможным размером списка, но такое решение для большинства случаев не является оптимальным.

Список переменного размера

Простой список переменного размера можно построить с помощью динамических массивов. Новый элемент в список добавляется следующим образом. Создается новый массив, размер которого на один элемент больше старого. Далее копируются элементы старого массива в новый и добавляется новый элемент. Затем освобождается старый массив и устанавливается указатель массива на новую страничку памяти. Эта простая схема хорошо работает для небольших списков, но у нее есть существенный недостаток, заключающийся в том, что приходится часто менять размер массива. Чтобы размер массива изменялся не так часто, при его увеличении можно вставлять дополнительные элементы, например, по 10 элементов вместо 1. Если впоследствии возникает необходимость добавлять новые элементы к списку, то они разместятся в уже существующих в массиве неиспользованных ячейках, не увеличивая размер массива. Точно так же можно избежать изменения размера каждый раз при удалении элемента из списка. Свободные ячейки позволяют избежать изменения размеров массива всякий раз, когда необходимо добавить или удалить элемент из списка.

[3] Составлено на основе материалов монографий [3], [6]

Неупорядоченные списки

В некоторых приложениях требуется удалять одни элементы из середины списка, добавляя другие в его конец. Это может быть в случае, когда порядок элементов не важен, но необходимо иметь возможность удалять определенные элементы из списка. Когда удаляется элемент из середины списка, оставшиеся элементы сдвигаются на одну позицию, заполняя образовавшийся промежуток. Однако удаление элемента массива подобным способом может занимать много времени, особенно если этот элемент находится в начале списка. Чтобы удалить первый элемент массива, содержащего 1000 записей, необходимо сдвинуть 999 элементов на одну позицию влево. Гораздо быстрее удалять элементы при помощи простой схемы «сборки мусора». Вместо удаления элементов из списка можно их отметить как неиспользуемые. Если элементы списка – данные простых типов, то их можно маркировать с помощью так называемого «мусорного» значения. Через некоторое время список может переполниться «мусором», в результате чего будет тратиться большое количество машинного времени на пропуск ненужных элементов, чем на обработку реальных данных. Во избежание такой ситуации надо периодически выполнять процедуру «сборки мусора».

Связанные списки

В отличие от неупорядоченных списков, элементы которых никак не связаны друг с другом [4], отличительной особенностью данного вида списков является именно наличие связей между его элементами. На рис. 3 представлена структура односвязного списка.

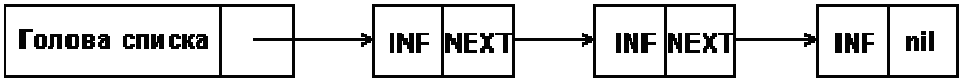

Рис. 3. Структура односвязного списка

[4] Во всяком случае, данная структура данных не требует наличия такой связи.

В описанном списке поле *INF* – информационное поле, данные, *NEXT* – указатель на следующий элемент списка. Для последовательного обхода всех элементов списка необходимо иметь указатель на его вершину. Для того чтобы корректно обрабатывать случай конца списка, лучше всего в поле указателя последнего элемента списка записывать значение пустой ссылки.

Данный вид списка обладает следующими важными свойствами. Для того чтобы в начало списка добавить новые элементы, необходимо создать новую ячейку, указывающую на текущую вершину списка, а затем сделать так, чтобы новой вершиной списка стала только что созданная ячейка (рис. 4).

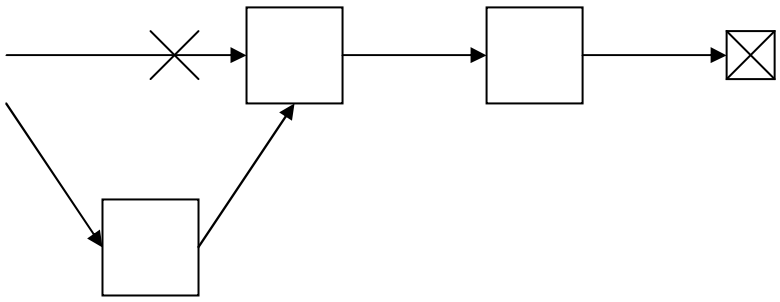

Рис. 4. Добавление элемента в начало связанного списка

Процедура удаления элемента как из начала, так и из середины списка также не вызывает никаких сложностей.

Обычно связанные списки удобнее, но списки на базе массивов имеют одно существенное преимущество – они используют меньше памяти. Каждый из указателей занимает дополнительные четыре байта памяти. Кроме этого, доступ к элементам связанного списка происходит последовательно, что занимает лишнее машинное время.

Если в программе возникает необходимость обхода элементов списка как в прямом, так и обратном порядке, то для этого случая целесообразно использовать другую разновидность связанных списков, а именно, двусвязные списки. Каждый элемент такого списка хранит указатель на следующий элемент и на предыдущий.

1.3. Стеки [5]

Стек - такой последовательный список с переменной длиной, включение и исключение элементов из которого выполняются только с одной стороны списка, называемого вершиной стека.

Примеры стека: винтовочный патронный магазин, тупиковый железнодорожный разъезд для сортировки вагонов. Традиционно операция помещения значения в стек называется *pushing* (проталкивание), а извлечение из стека – *popping* (выталкивание). Для реализации стека можно использовать как динамические массивы, так и односвязный список.

1.4. Очереди. Деки [6]

Очередь или односторонняя очередь - это линейный список, в котором все операции вставки выполняются на одном из концов списка, а все операции удаления (и, как правило, операции доступа к данным) - на другом.

Реализация очередей также может быть осуществлена с помощью динамических массивов или связанных списков. Однако в данном случае использование динамических массивов представляется достаточно затратным с точки зрения расходования вычислительных ресурсов. Ввиду того, что данные должны вводиться с одного конца массива, а удаляться с другого, получается, что необходимо постоянно сдвигать элементы массива, даже если после каждой вставки происходит операция удаления из очереди. Поэтому, для реализации очередей целесообразно использовать именно связанные списки, а из-за того, что должны осуществляться операции на его обоих концах, наиболее подходящими структурами являются двусвязные списки.

[5] Составлено на основе материалов монографии [6].
[6] Составлено на основе материалов монографий [4], [6].

Очереди с приоритетом

В данной очереди каждый элемент имеет определенный приоритет. При удалении элементов, в первую очередь удаляются элемент с самым высоким приоритетом. При этом не имеет значения, в каком порядке элементы хранятся в очереди.

Некоторые операционные системы используют очереди с приоритетом для планирования задания. В операционной системе *UNIX* все процессы имеют разные приоритеты. Когда процессор освобождается, выбирается готовый к исполнению процесс с максимальным приоритетом.

Самый простой способ организации очереди с приоритетами - поместить все элементы в список. Если требуется удалить элемент из очереди, надо найти в списке элемент с наивысшим приоритетом. Чтобы добавить элемент к очереди, необходимо просто разместить новый элемент в начале списка. Если очередь содержит *N* элементов, требуется *O(N)* шагов, чтобы определить положение и удалить из очереди элемент с максимальным приоритетом.

Чтобы добавить элемент в очередь, необходимо найти для него правильную позицию в списке и поместить его туда.

Чтобы удалить из списка элемент с самым высоким приоритетом, достаточно удалить первый элемент в очереди. Поскольку список отсортирован в порядке уменьшения приоритета, первый элемент всегда имеет наивысший приоритет.

1.5. Деревья [7]

Деревья являются одними из наиболее важных нелинейных структур, которые встречаются при работе с компьютерными алгоритмами.

Формально дерево определяется как конечное множество *T* одного или более узлов со следующими свойствами:

[7] Составлено на основе материалов монографий [4], [6].

a) существует один выделенный узел - **корень** данного дерева *T*;

b) остальные узлы (за исключением корня) распределены среди $m \geq 0$ непересекающихся множеств $T_1,....T_m$, каждое из котрых, в свою очередь, является деревом; деревья $T_1,..., T_m$ называются **поддеревьями** данного корня.

Данное определение является рекурсивным: дерево определено на основе понятия дерева.

На рис. 5 приведен пример дерева. Корневой узел *A* соединен с тремя поддеревьями, начинающимися узлами *B*, *C* и *D*. Эти узлы соединены с поддеревьями, имеющими корни в узлах *E*, *F* и *G*, которые в свою очередь связаны с поддеревьями с корнями *H*, *I* и *J*.

Данная терминология является смесью терминов, заимствованных из ботаники и генеалогии. Из ботаники взято определение **узла**, который представляет собой точку, где может возникнуть ветвь. **Ветвь** описывает связь между двумя узлами, **лист** - узел, откуда не выходят другие ветви.

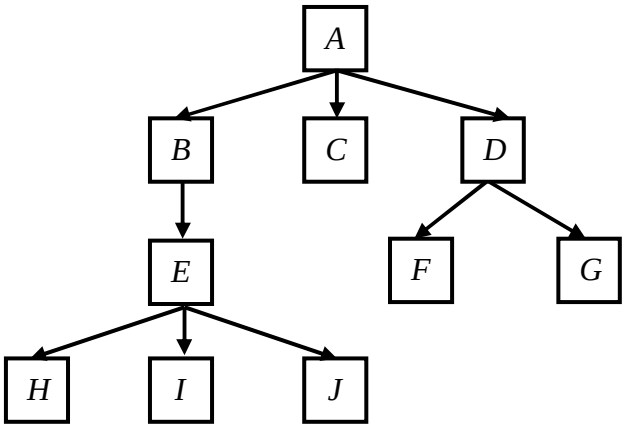

Рис. 5. Дерево

Из генеалогии пришли термины, описывающие отношения. Если узел находится непосредственно над другим, то он называется **родительским**, а нижний узел называется **дочерним**. Узлы на пути вверх от узла до корня принято считать **предками узла**. Например, на рис. 5 предками узла *I* являются узлы *E*, *B* и *A*. Все узлы, расположенные ниже какого-либо узла, называются его **потомками**. На рис. 5 потомками узла *B* являются узлы *E*, *H*, *I* и *J*. Узлы, имеющие одного и того же родителя, называются **сестринскими**.

Кроме того, существует несколько понятий, возникших собственно в программистской среде. **Внутренний узел** - это узел, не являющийся листом. **Порядком узла** или его **степенью** называется количество его дочерних узлов. **Степень дерева** - это максимальный порядок его узлов. Степень дерева, изображенного на рис. 5, равна трем, так как узлы *A* и *E*, имеющие максимальную степень, имеют по три дочерних узла.

Высота узла на единицу больше числа его предков. На рис. 5 узел *E* имеет высоту *3*. **Высота дерева** - это наибольшая высота всех его узлов. Высота дерева, изображенного на рис. 5, равна *4*.

Дерево степени *2* называется **двоичным** или **бинарным** деревом. Деревья степени *3* называются **троичными** или **тернарными**. Аналогично, дерево степени *N* называется *N*-ичным деревом.

Если в расположении узлов дерева имеет значение относительный порядок поддеревьев, то дерево является **упорядоченным**.

Лес - это множество, не содержащее ни одного непересекающегося дерева или содержащее несколько непересекающихся деревьев.

Между абстрактными понятиями леса и деревьев существует не очень заметная разница. При удалении корня дерева получается лес, и наоборот: при добавлении одного узла в лес, все деревья которого рассматриваются как поддеревья нового узла, получается дерево.

Обход деревьев

Последовательное обращение ко всем узлам дерева называется **обходом**. Существует несколько последовательностей обхода узлов двоичного дерева. Три самых простых - прямой, симметричный и обратный, которые реализуются с помощью достаточно простых рекурсивных алгоритмов. Для каждого заданного узла алгоритм выполняет следующие действия:

Прямой порядок:

1) обращение к узлу;
2) рекурсивный прямой обход левого поддерева;
3) рекурсивный прямой обход правого поддерева.

Симметричный порядок:

1) рекурсивный симметричный обход левого поддерева;
2) обращение к узлу;
3) рекурсивный симметричный обход правого поддерева.

Обратный порядок:

1) рекурсивный обратный обход левого поддерева;
2) рекурсивный обратный обход правого поддерева;
3) обращение к узлу.

Для дерева, изображенного на рисунке 6, порядок обхода будет следующим:

Прямой порядок: *A, B, D, C, E, G, F, H, J;*

Симметричный порядок: *D, B, A, E, G, C, H, F, J;*

Обратный порядок: *D, B, G, E, H, J, F, C, A.*

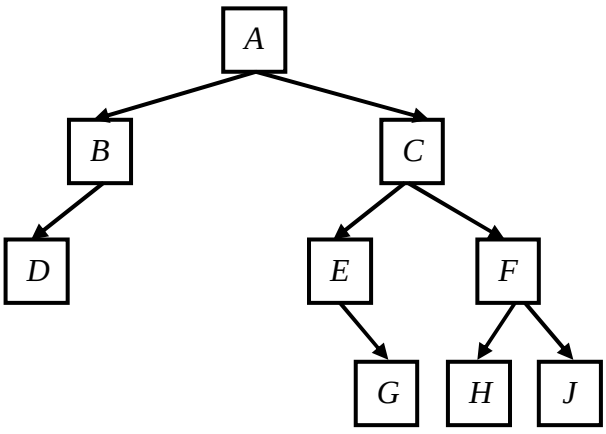

Рис. 6. Пример дерева для иллюстрации различных методов его обхода

Упорядоченные деревья

Двоичные деревья - обычный способ хранения и обработки информации в компьютерных программах. Если использовать внутренние узлы дерева, чтобы обозначить утверждение «левый дочерний узел меньше правого», то с помощью двоичного дерева можно построить сортированный список. На рис. 7 показано двоичное дерево, хранящее сортированный список с числами *1, 2, 4, 6, 7, 9.*

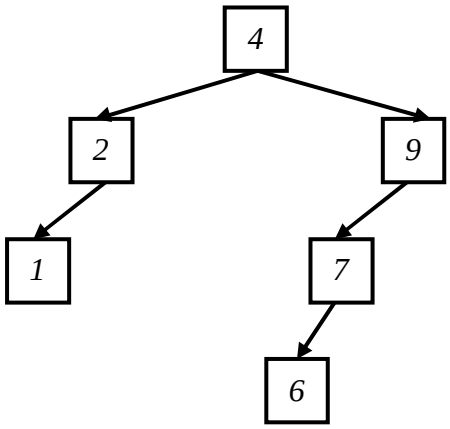

Рис. 7. Упорядоченный список *1, 2, 4, 6, 7, 9*

Добавление элементов

Алгоритм добавления нового элемента в описываемый тип деревьев достаточно прост. Сначала берется корневой узел. Далее следует по очереди сравнивать значения всех узлов со значением нового элемента. Если новое значение меньше или равно значению в рассматриваемом узле, то необходимо продолжать движение вниз по левой ветви. Если новое значение больше значения узла, то переходить следует вниз по правой ветви. Если достигнут конец дерева [8], необходимо вставить элемент в эту позицию.

[8] Под «достижением конца дерева» понимается попытка перехода с текущего узла по ссылке на дочерний, однако соответствующее значение ссылки является пустым.

Удаление элемента из сортированного дерева немного сложнее, чем добавление. После этой операции программа должна перестроить другие узлы, чтобы сохранить в дереве соотношение «меньше чем». Следует рассмотреть несколько существующих вариантов.

Во-первых, если удаляемый узел не имеет потомков, допустимо просто убрать его из дерева. При этом порядок остальных узлов не изменяется. Во-вторых, если удаляемый узел имеет один дочерний узел, можно заменить его дочерним узлом. При этом порядок потомков данного узла сохраняется, так как эти узлы также являются и потомками дочернего узла. На рис. 8 показано дерево, где удаляется узел *4*, имеющий только один дочерний узел.

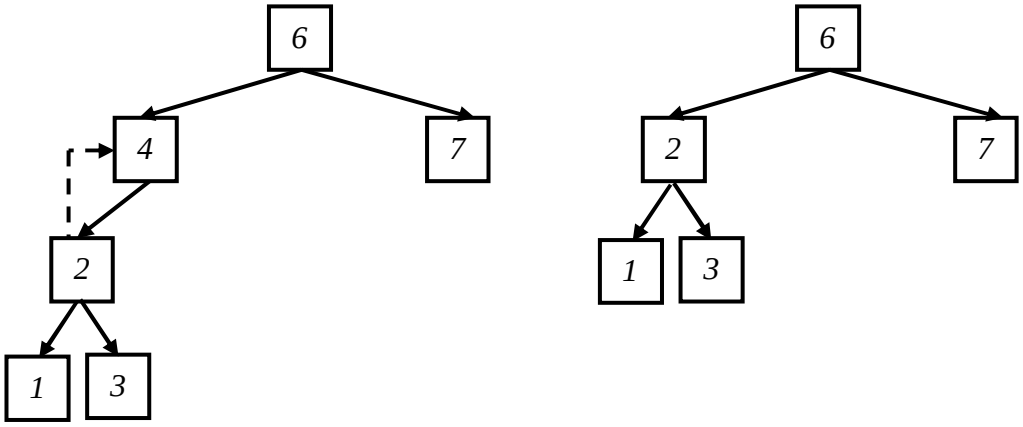

Рис. 8. Удаление узла с единственным потомком

Если удаляемый узел имеет два дочерних, вовсе не обязательно, что один из них займет его место. Если потомки узла также имеют по два дочерних, то для размещения всех дочерних узлов в позиции удаленного узла просто нет места. Чтобы решить эту проблему, необходимо заменить удаленный узел крайним правым узлом дерева из левой ветви. Другими словами, необходимо двигаться вниз от удаленного узла по левой ветви. Затем необходимо двигаться вниз по правым ветвям до тех пор, пока не найдется узел без правой ветви. Если найденный узел является листом (на рис. 9 это узел *3*), то им следует заменить удаляемый узел.

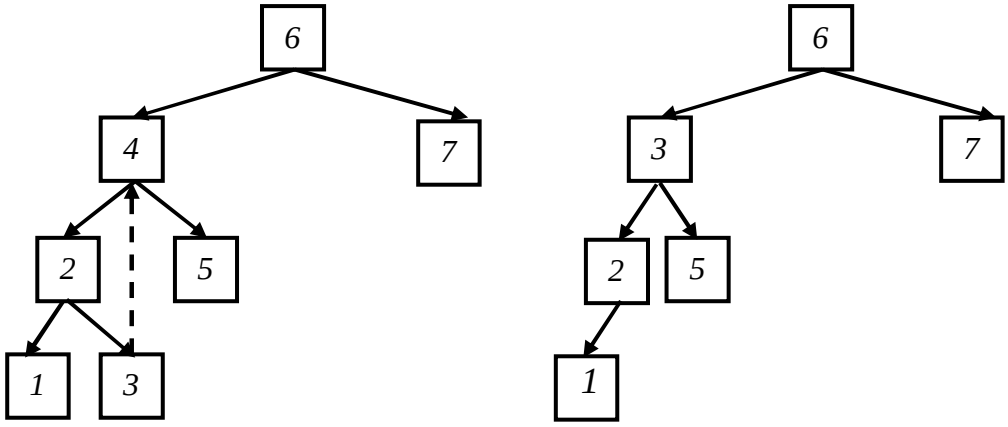

Рис. 9. Удаление узла с двумя потомком

В случае же когда найденный узел имеет левого потомка, тот встает на место заменяемого узла.

Сбалансированные деревья

После выполнения ряда операций с упорядоченным деревом, таких как вставка и удаление узлов, оно может стать несбалансированным. Если подобное происходит, алгоритмы обработки дерева становятся менее эффективными. При сильной степени разбалансировки дерево фактически вытягивается в линейный список[9], а у программы, использующей дерево, может резко снизиться производительность. Далее описываются методы сохранения баланса дерева даже при постоянных вставке и удалении элементов.

Балансировка

Форма упорядоченного дерева зависит от порядка добавления элементов. Высокие, тонкие деревья могут иметь глубину до *O(N)*, где *N*- число узлов дерева. Добавление или размещение элемента в таком разбалансированном дереве может занимать *O(N)* шагов. Даже если новые элементы разме-

[9] Самым худшим случаем с точки зрения разбалансировки дерева является последовательное добавление в дерево возрастающей последовательности элементов.

щаются беспорядочно, в среднем они дадут дерево с глубиной *N/2*, обработка которого потребует так же порядка *O(N)* операций.

Ниже рассматриваются методы, которые позволяют поддерживать деревья в сбалансированном состоянии при вставке и удалении узлов.

AVL-деревья

AVL-деревья были названы по имени российских математиков Адельсона-Вельского и Ландау, которые их изобрели. В каждом узле *AVL*-дерева глубина левого и правого поддеревьев отличаются не более чем на единицу. *AVL*-дерево имеет глубину $O(\log_2 N)$. Следовательно, и поиск узла в *AVL*-дереве занимает время порядка $O(\log_2 N)$, что при больших *N* существенно меньше, чем *O(n)*.

Добавление узлов к *AVL*-дереву

Каждый раз при добавлении узла к *AVL*-дереву необходимо проверять, соблюдаются ли условия, описывающие *AVL*-дерево. После вставки узла следует исследовать узлы в обратном порядке - к корню, проверяя, чтобы глубина поддеревьев отличалась не более чем на единицу. Если найдена ячейка, где это условие не выполняется, необходимо сдвинуть элементы, чтобы сохранить выполняемость условия *AVL*-дерева.

Вращение *AVL*-деревьев

При вставке узла в *AVL*-дерево в зависимости от того, в какую часть дерева добавляется узел, существует четыре варианта балансировки. Методы перебалансирования называют правым и левым вращением, вращением влево-вправо и вправо-влево. Сокращенно они обозначаются *R*, *L*, *LR* и *RL*.

Если к *AVL*-дереву добавить новый узел, то, как показано на рис. 10, оно становится разбалансированным в узле *X*. На рисунке изображены только узел *X* и два его дочерних узла, остальные части дерева обозначены треугольниками, так как нет необходимости их рассматривать. Новый узел может

быть вставлен в любое из этих четырех поддеревьев, изображенных в виде треугольников ниже узла *X*. В зависимости от того, в какое поддерево вставлен новый узел, происходит соответствующий вид вращения. Важно, что вращение следует применять только в случае, если вставляемый узел нарушает упорядоченность *AVL*-дерева.

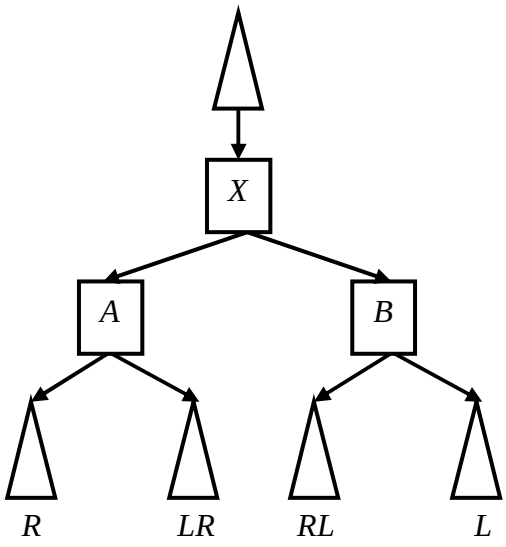

Рис. 10. Анализ разбалансированного *AVL*-дерева

Правое вращение

Если новый узел добавляется к поддереву *R* на рис. 10, то поддеревья *RL* и *L* изменяться не будут, поэтому их можно сгруппировать в один треугольник, как показано на рис. 11. Новый узел добавляется к дереву T_1, при этом поддерево T_A с корнем в узле *A* становится по крайней мере на два уровня длиннее, чем поддерево T_3.[10] Механизм правого вращения представлен на рис. 11. Это вращение называется правым, поскольку узлы *A* и *X* сдвигаются на одну позицию вправо.

[10] Имеется в виду ситуация, когда нарушается условие *AVL*-дерева

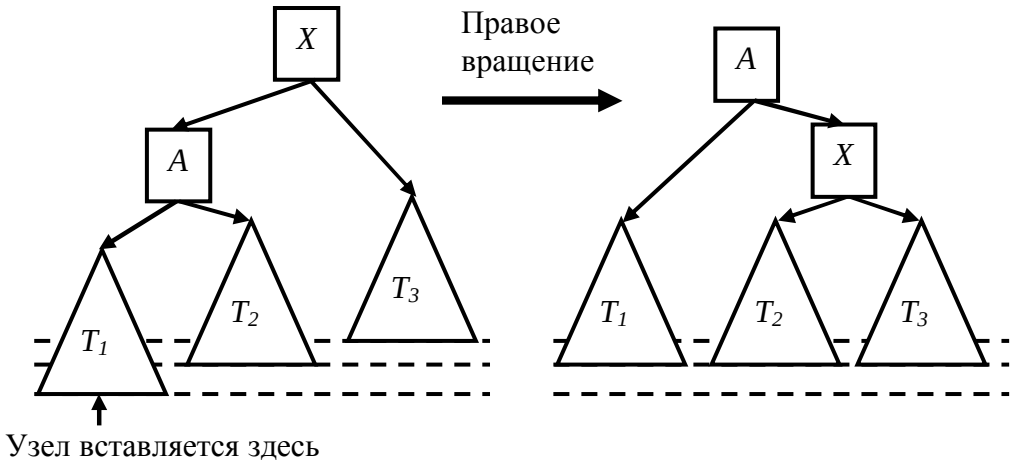

Рис. 11. Механизм правого вращения *AVL*-деревьев

После добавления узла и применения правого вращения глубина поддерева не увеличивается. Любая часть дерева, лежащая выше узла *X*, остается сбалансированной, поэтому дальнейшая балансировка не нужна.

Левое вращение

Левое вращение аналогично правому. Оно используется с целью перебалансировки дерева, когда новый узел добавляется к поддереву *L*, показанному на рис. 10. На рис. 12. изображено *AVL*-дерево до и после левого вращения.

Вращение влево-вправо

Когда узел добавляется в поддерево *LR* (см. рис. 10), необходимо рассмотреть еще один нижележащий уровень. На рис. 13 показано дерево, в котором новый узел вставляется в левую часть T_2 поддерева *LR*. В данном случае поддеревья T_A и T_C удовлетворяют свойству *AVL*, а поддерево T_X - нет.

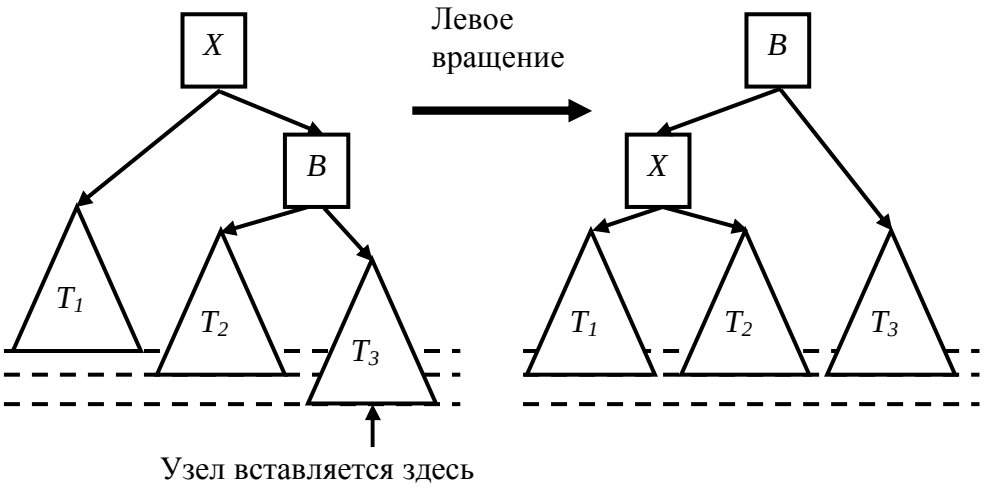

Рис. 12. Механизм левого вращения *AVL*-деревьев

Перебалансировка деревьев по принципу влево-вправо показана на рис. 13. Данный случай называется вращением влево-вправо, потому что узлы *A* и *C* сдвигаются на одну позицию влево, а узлы *C* и *X* - на одну позицию вправо.

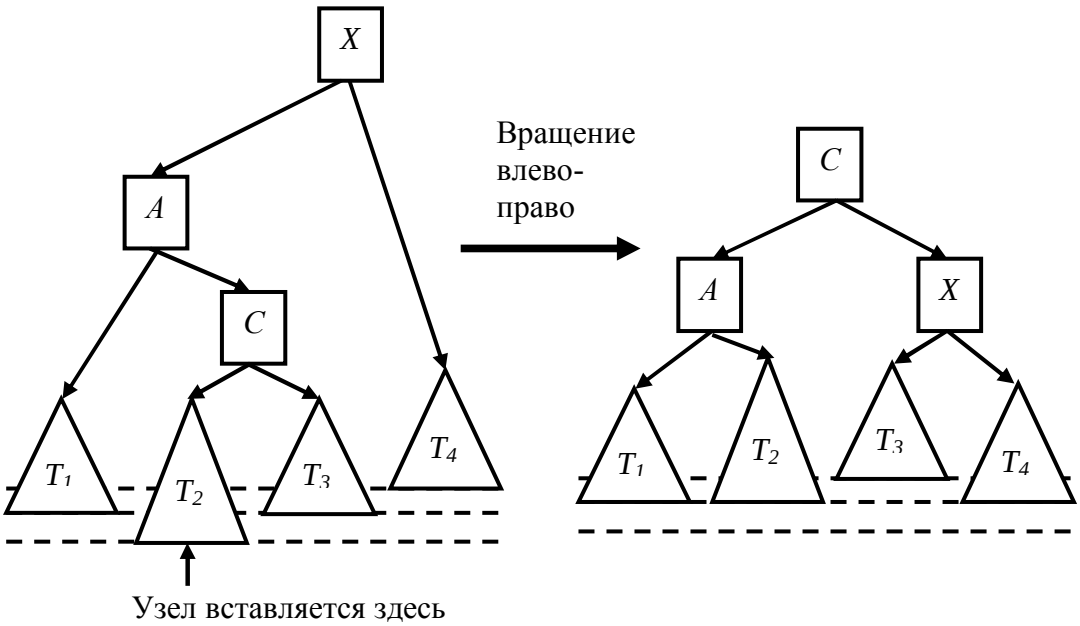

Рис. 13. Механизм вращения *AVL*-деревьев влево-вправо

Вращение вправо-влево

Вращение вправо-влево аналогично вращению влево-вправо. Оно используется для балансировки дерева после вставки узла в поддерево *RL*, изображенного на рис. 10. На рис. 14 показано *AVL*-дерево до и после вращения вправо-влево.

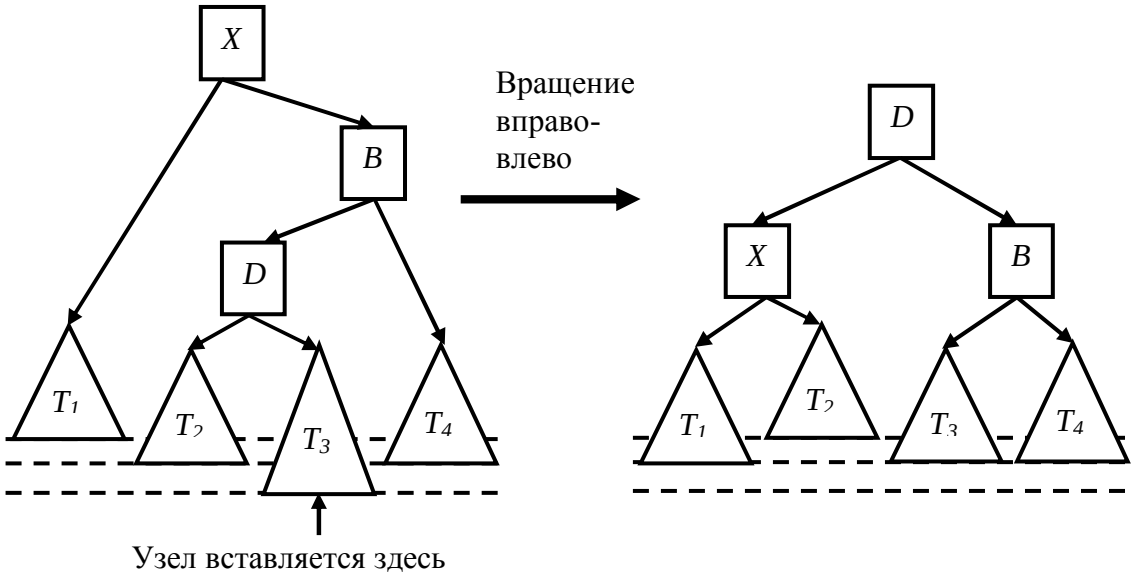

Рис. 14. Механизм вращения AVL-деревьев вправо-влево

Красно-черные деревья [11]

Красно-черные деревья – еще один из способов балансировки деревьев. Название происходит от стандартной раскраски узлов таких деревьев в красный и черный цвета. Цвета узлов используются при балансировке дерева. Во время операций вставки и удаления может понадобиться повернуть поддеревья, чтобы достигнуть сбалансированности дерева.

Красно-черное дерево - это бинарное дерево со следующими свойствами:

1) каждый узел дерева покрашен либо в черный, либо в красный цвет;

[11] Составлено с использованием материалов сайта http://algolist.manual.ru

2) листьями объявляются *nil*-узлы (т.е. "виртуальные" узлы, наследники узлов, которые обычно называют листьями; на них "указывают" *nil* указатели). Листья покрашены в черный цвет;
3) если узел красный, то оба его потомка черны;
4) на всех ветвях дерева, ведущих от его корня к листьям, число черных узлов одинаково.

Количество черных узлов на ветви от корня до листа называется **черной высотой** дерева. Перечисленные свойства гарантируют, что самая длинная ветвь от корня к листу не более чем вдвое длиннее любой другой ветви от корня к листу.

Все вышеперечисленные свойства должны сохраняться при вставке элементов в дерево и удалении из него.

Вставка

Чтобы вставить узел, необходимо сначала найти соответствующее место в дереве. Новый узел всегда добавляется как лист, поэтому оба его потомка являются *nil*-узлами и предполагаются черными. После вставки необходимо окрасить узел в красный цвет. Затем проверяются все вышеприведенные свойства для его предка. Далее в случае необходимости осуществляется перекрашивание узла и/или поворот, чтобы сбалансировать дерево.

Вставив красный узел с двумя *nil* -потомками, свойство черной высоты (свойство 4) сохранится. Однако при этом может оказаться нарушенным свойство 3, согласно которому оба потомка красного узла обязательно черны. В рассматриваемом случае оба потомка нового узла черны по определению (поскольку они являются *nil*-узлами). Поэтому необходимо рассмотреть ситуацию, когда предок нового узла красный: при этом будет нарушено свойство 3. Достаточно рассмотреть следующие два случая:

Красный предок, красный «дядя». Данную ситуацию иллюстрирует рис. 15. У нового узла *X* предок и «дядя» оказались красными. В рассматриваемом случае простое перекрашивание избавляет от нарушения свойства 3.

После перекраски нужно проверить «дедушку» нового узла (узел *B*), поскольку он может оказаться красным. Следует обратить внимание на то, что в данном случае меняется цвет корня дерева, т.к. в противном случае нарушилось бы свойство черной высоты.

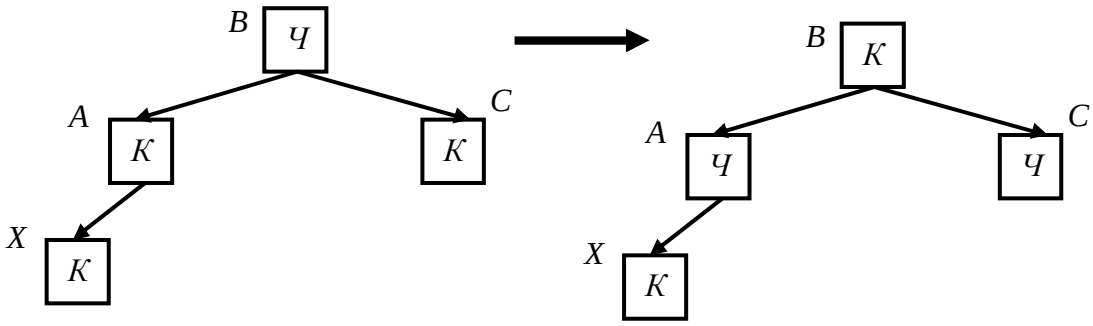

Рис. 15. Вставка – Красный предок, красный «дядя»

Красный предок, черный «дядя». На рис. 16 представлен другой пример нарушения свойства 3 красно-черных деревьев. В данном случае для коррекции необходимо применить дополнительно вращение узлов. Следует обратить внимание, что если узел *X* был в начале правым потомком, то для коррекции следует применять левое вращение, которое делает этот узел левым потомком.

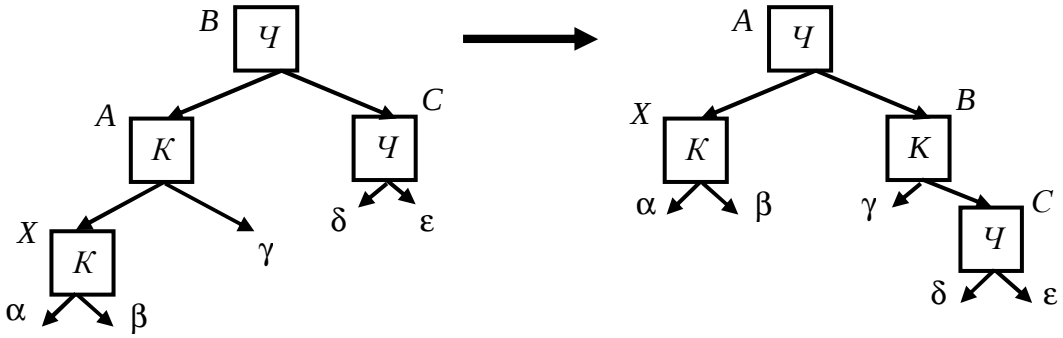

Рис. 16. Вставка – красный предок, черный «дядя»

Деревья со случайным поиском [12]

При вводе элемента в дерево случайного поиска этому элементу присваивается приоритет - некоторое вещественное число с равномерным распределением в диапазоне [0, 1]. Приоритеты элементов в дереве случайного поиска определяют их положение в дереве в соответствии с правилом: приоритет каждого элемента в дереве не должен быть больше приоритета любого из его последователей. Правило двоичного дерева поиска также остается справедливым: для каждого элемента *x* элементы в левом поддереве должны быть меньше, чем *x*, а в правом – больше, чем *x*. На рис. 17 приведен пример такого дерева случайного поиска, на котором приоритеты каждого элемента указаны в виде нижнего индекса.

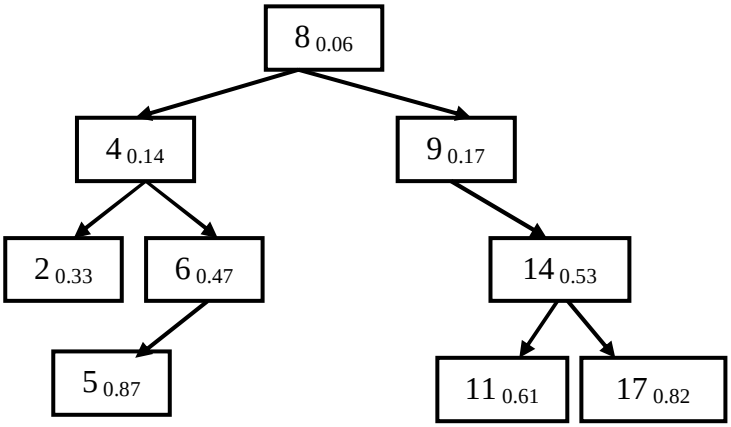

Рис. 17. Дерево случайного поиска

Ввод элемента *x* в дерево случайного поиска выполняется в два этапа. На первом этапе необходимо определить позицию узла в дереве в соответствии с правилами обычного бинарного упорядоченного дерева. На втором этапе необходимо изменить форму дерева в соответствии с приоритетами элементов. В этом случае может потребоваться либо правое, либо левое вращение.

[12] Составлено с использованием материалов сайта http://algolist.manual.ru

2. Сортировка

Под сортировкой записей $R_1, R_2, ..., R_N$ понимается сортировка в порядке не убывания их ключей $K_1, K_2, ..., K_N$. В данном разделе рассматриваются алгоритмы внутренней сортировки, когда число записей, подлежащих сортировке, достаточно мало, так что их можно полностью разместить в оперативной памяти компьютера, обладающей высокой скоростью обмена.

2.1. Сортировка путем вставок

Предполагается, что перед рассмотрением записи R_j предыдущие записи R_1, ..., R_{j-1} уже упорядочены. Задача состоит в необходимости вставить данную запись таким образом, чтобы массив оставался упорядоченным.

Метод простых вставок

Пусть $1<j<N$ и записи $R_1,...,R_{j-1}$ уже размещены так, что $K_1 \le K_2 \le ... \le K_{j-1}$. Необходимо производить сравнения ключа K_j с ключами K_{j-1}, K_{j-2}, ... до тех пор, пока не обнаружится, что запись R_j следует вставить между R_i и R_{i+1}. В этом случае необходимо сдвинуть записи R_{i+1}, ..., R_{j-1} на одну позицию вверх и поместить новую запись в позицию $i+1$. Операции сравнения и перемещения удобно совмещать, перемежая их между собой. Поскольку запись R_j как бы «погружается» на положенный ей уровень, этот способ часто называют **просеиванием** или **погружением**.

Бинарные и двухпутевые вставки

Поиск ячеек, между которыми следует вставить новую запись можно модифицировать, учитывая тот факт, что первоначальный массив уже отсортирован. Например, если вставляется 64-я запись, можно сначала сравнить

ключ K_{64} с K_{32} а затем, если он меньше, сравнить его с K_{16}, а если больше - с K_{48} и т. д. Данный метод называется **методом бинарных вставок**. Однако данный метод позволяет решить задачу только наполовину: после того как найдено место, куда следует вставлять запись R_j, все равно необходимо осуществить сдвиг примерно *j/2* ранее рассортированных записей, чтобы освободить место для R_j.

Таким образом, в обоих описанных методах самой продолжительной по времени является операция сдвига элементов в массиве. Метод двухпутевых вставок, впервые предложенный в начале 50-х годов прошлого века, позволяет несколько сократить ее продолжительность. Иллюстрация метода представлена на рисунке 18. Здесь первый элемент помещается в середину области вывода и место для последующих элементов освобождается при помощи сдвигов влево или вправо, туда, куда удобнее. Таким образом, удается сэкономить примерно половину времени работы по сравнению с использованием метода простых вставок за счет некоторого усложнения программы.

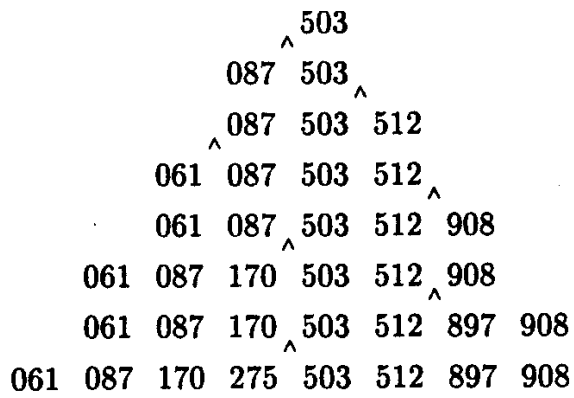

Рис. 18. Процесс сортировки методом двухпутевых вставок

Метод Шелла

Для алгоритмов сортировки, которые каждый раз перемещают запись только на одну позицию, среднее время выполнения будет в лучшем случае пропорционально N^2, потому что в процессе сортировки каждая запись долж-

на пройти в среднем через *N/3* позиций. Поэтому, если желательно получить метод, существенно превосходящий по скорости метод простых вставок, необходим механизм, с помощью которого записи могли бы перемещаться сразу на несколько позиций.

Один из таких методов был предложен в 1959 году Дональдом Л. Шеллом и известен во всем мире под именем своего автора. На рис. 19 проиллюстрирована общая идея, которая лежит в его основе для массива из 16 записей. Сначала они делятся на 8 групп по две записи в каждой: (R_1,R_9), (R_2,R_{10}), … , (R_8, R_{16}). Каждая из этих групп сортируется по возрастанию. В результате сортировки каждой группы записей по отдельности получается вторая строка на рис. 19. Этот процесс называется первым проходом. Далее записи делятся на четыре группы по четыре в каждой: (R_1,R_5,R_9,R_{13}), (R_4, R_8,R_{12},R_{16}). Затем опять сортируется каждая группа в отдельности. Результат этого второго прохода показан в третьей строке на рис. 19.

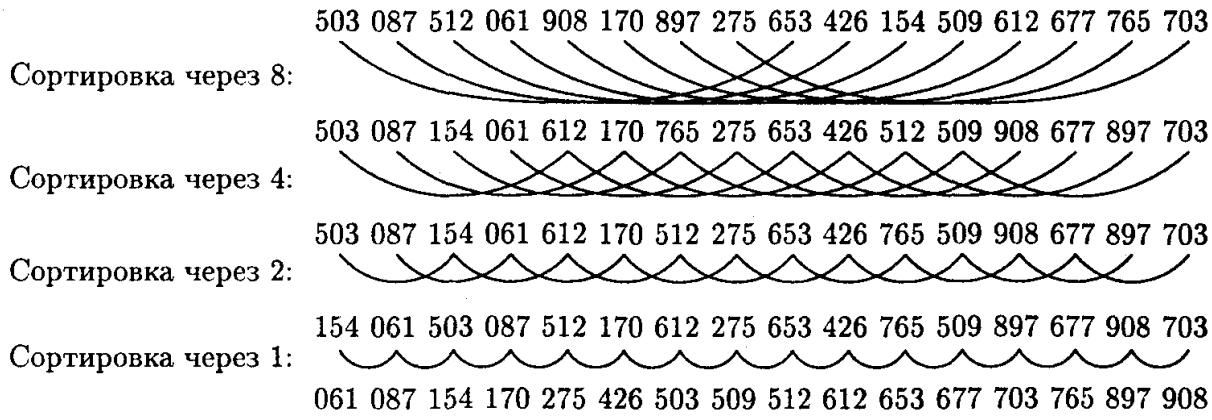

Рис. 19. Сортировка методом Шелла со смещениями 8, 4, 2, 1

На третьем проходе сортируются две группы по восемь записей; процесс завершается четвертым проходом, во время которого сортируются все 16 записей. В каждой из промежуточных стадий сортировки участвуют либо сравнительно короткие массивы, либо уже сравнительно хорошо упорядоченные массивы, поэтому на каждом этапе можно пользоваться методом простых вставок и сортировка выполняется довольно быстро.

Метод сортировки Шелла также известен под именем метода сортировки с «**убывающим смещением**», поскольку каждый проход характеризуется смещением h, таким, что сортируются записи, каждая из которых отстоит от предыдущей на h позиций. На практике можно пользоваться любой последовательностью $h_{t-1},h_{t-2},\ldots,h_0$, но последнее смещение h_0 должно быть равно 1. Выбор значений смещений на последовательных проходах имеет существенное значение для скорости сортировки.

Для задания значений $h_0,h_1,\ldots$ предлагается следующее правило.

Если N достаточно большое (больше 1000), то рекомендуется использовать последовательность Седгевика:

$$h_s = \begin{cases} 9*2^s - 9*2^{s/2} + 1, & s \text{ четно}; \\ 8*2^s - 6*2^{(s+1)/2} + 1, & s \text{ нечетно}, \end{cases}$$

которая обрывается на h_{t-1}, если $3h_t \le N$.

Седгевик доказал, что когда используются сформированные по этому правилу смещения $(h_0, h_1, h_2,\ldots)=(1, 5, 19, \ldots)$, то время выполнения в худшем случае не превышает $O(N^{4/3})$.

2.2. Обменная сортировка

Для всех методов обменных сортировок характерно наличие обменов или транспозиций, предусматривающих систематический обмен местами между элементами пар, в которых нарушается упорядоченность, до тех пор, пока таких пар не останется.

Метод пузырька

Наиболее очевидный способ обменной сортировки заключается в том, чтобы сравнить K_1 с K_2, меняя местами R_1 и R_2, если их ключи расположены не в нужном порядке, и затем проделать то же самое с R_2 и R_3, R_3 и R_4 и т. д. При выполнении этой последовательности операций записи с большими ключами бу-

дут продвигаться вправо. Все это закончится тем, что запись с наибольшим ключом займет положение R_N. При многократном выполнении данного процесса соответствующие записи попадут в позиции R_{N-1}, R_{N-2} и т. д., в итоге, все записи будут упорядочены. Последовательность чисел удобно представлять не горизонтально, а вертикально, чтобы запись R_N была в самом верху, а R_1 - в самом низу. Метод называется «методом пузырька», потому что большие элементы, подобно пузырькам, «всплывают» на соответствующую позицию.

После каждого просмотра последовательности все записи, расположенные выше самой последней, которая участвовала в обмене, и сама эта запись должны занять свои окончательные позиции, так что их не нужно проверять при последующих просмотрах.

Среднее число сравнений и обменов имеют квадратичный порядок роста: $O(N^2)$, отсюда можно заключить, что «алгоритм пузырька» очень медленен и малоэффективен.

Быстрая сортировка

Алгоритм быстрой сортировки был разработан более 40 лет назад, однако в настоящее время является, пожалуй, наиболее широко применяемым и одним их самых эффективных алгоритмов. Метод основан на подходе «разделяй-и-властвуй». Общая схема быстрой сортировки такова:

1) из массива выбирается некоторый опорный элемент *a[i]*;
2) запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные *a[i]*, влево от него, а все ключи, большие, либо равные *a[i]* – вправо. В результате выполнения данной операции массив будет состоять из двух подмножеств, причем левое меньше либо равно правого:

≤ a[i]	a[i]	≥ a[i]

;

3) для обоих подмассивов проверяется условие: если хотя бы в одном из них содержится более двух элементов, то для него рекурсивно запускается та же процедура.

В среднем производительность метода составляет *O(N log N)* операций. Однако при определенных входных данных она падает до $O(N^2)$ операций. Такое происходит, если каждый раз в качестве опорного элемента выбирается максимум или минимум входной последовательности.

2.3. Сортировка посредством выбора

Еще одно важное семейство методов сортировки основано на идее многократного выбора. Простейшая сортировка посредством выбора сводится к следующему:

1) найти наименьший ключ, переслать соответствующую запись в область вывода и заменить ключ значением ∞, которое по предположению больше любого реального ключа;
2) повторить шаг (1). На этот раз будет выбран ключ, наименьший из оставшихся, так как ранее наименьший ключ был заменен значением ∞;
3) повторять шаг (1) до тех пор, пока не будут выбраны *N* записей.

Данный метод называется **сортировка путем простого выбора**. Этот метод требует наличия всех исходных элементов до начала сортировки, а элементы вывода порождает последовательно, один за другим. Ситуация по существу, противоположна ситуации, возникающей при использовании метода вставок, в котором исходные записи поступают последовательно, но вплоть до завершения сортировки об окончательном результате ничего неизвестно.

Пирамидальная сортировка [13]

Пирамидой *(heap)* называется бинарное дерево высоты *k*, в котором

- все узлы имеют глубину *k* или *k-1*;
- уровень *k-1* дерева полностью заполнен, а уровень *k* заполнен слева направо;
- выполняется «свойство пирамиды»: каждый элемент меньше, либо равен родителю.

Ниже на рисунке 20 показано дерево, удовлетворяющее свойству пирамиды.

[13] Составлено с использованием материалов сайта http://algolist.manual.ru

Для хранения пирамиды можно использовать массив. В *a[0]* хранится корень дерева; левый и правый потомки элемента *a[i]* хранятся в *a[2i+1]* и *a[2i+2]* соответственно.

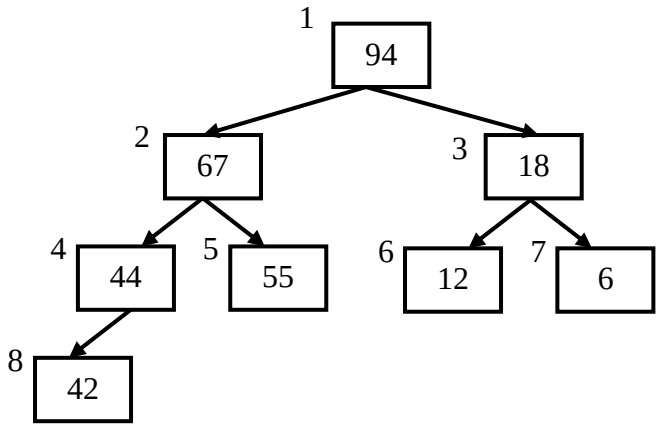

Рис. 20. Пример дерева, удовлетворяющего свойству пирамиды

Фаза 1: построение пирамиды

Построение пирамиды начинается с a*[k]...a[n], k = [size/2]*. Эта часть массива уже удовлетворяет свойству пирамиды.

Далее необходимо расширять данную часть массива, добавляя по одному элементу за шаг. Для этого необходимо просмотреть правого и левого потомков - в массиве это *a[2i+1]* и *a[2i+2]* и выбрать наибольшего из них.

Если этот элемент больше *a[i]*, то необходимо поменять его с *a[i]* местами. Таким образом новый элемент «просеивается» сквозь пирамиду.

Фаза 2: сортировка

1) взять верхний элемент пирамиды *a[0]...a[n]* (первый в массиве) и поменять с последним местами. Далее этот элемент из рассмотрения необходимо исключить, так как он уже занял нужное место в массиве. Рассматривается массив *a[0]...a[n-1]*. Для превращения его в пирамиду достаточно просеять лишь новый первый элемент описанным выше способом;

2) повторять шаг 1, пока обрабатываемая часть массива не уменьшится до одного элемента.

Очевидно, что в конец массива каждый раз будет попадать максимальный элемент из текущей пирамиды, поэтому в правой части постепенно возникает упорядоченная последовательность.

Вторая фаза занимает *O(N log N)* операций. К положительным сторонам данного способа сортировки относится тот факт, что пирамидальная сортировка не использует дополнительной памяти.

2.4. Сортировка посредством слияния[14]

Слияние означает объединение двух или более упорядоченных массивов в один упорядоченный массив. Можно, например, слить подмассивы 503 703 765 и 087 512 677 и получить 087 503 512 677 703 765. Простой способ сделать это - сравнить два наименьших элемента, вывести наименьший из них и повторить эту процедуру. Задачу сортировки можно свести к слиянию, сливая все более длинные подмассивы до тех пор, пока не будет рассортирован весь массив. Такой подход можно рассматривать как развитие идеи сортировок методом вставок: вставка нового элемента в упорядоченный массив – это частный случай слияния при *n=1*. Чтобы ускорить процесс вставок, можно рассмотреть вставку нескольких элементов за один раз, группируя несколько операций, а это естественным образом приведет к общей идее сортировки методом слияния.

На рисунке 21 проиллюстрирована сортировка методом слияния, когда возрастающие серии выделяются как с начала массива, так и с его конца.

[14] Составлено на основе материалов монографии [5]

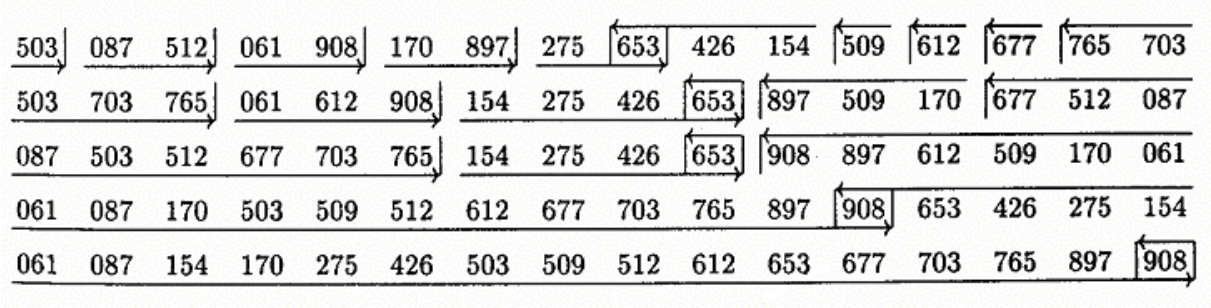

Рис. 21. Сортировка методом естественного двухпутевого слияния

Вертикальными линиями на рисунке 21 отмечены границы между сериями. Сложность данного алгоритма составляет *O(N log N)* операций.

Основной сложностью практической реализации данного алгоритма является необходимость создания дополнительных массивов для хранения результатов, либо большой объем по перемещению данных внутри одного массива. Поэтому, наиболее оптимальной структурой данных для данного алгоритма являются односвязные списки, с помощью которых удастся избежать излишнего расхода памяти в случае временных массивов, а также перемещения данных внутри них.

2.5. Сортировка методом распределения

Данный тип сортировки по своей сути является обратным слиянию. Данный тип сортировки начал широко использоваться со времен машин с перфокарточным оборудованием. Она общеизвестна под названиями «поразрядная сортировка», «карманная сортировка» *(bucket sorting)* и «цифровая сортировка» *(digital sorting)*, поиск базируется на анализе цифр ключей.

Поразрядная сортировка

Отличительными особенностями данного способа сортировки являются:

1) отсутствие сравнений сортируемых элементов;
2) ключ, по которому происходит сортировка, делится на части, разряды ключа. Например, слово можно разделить по буквам, а число - по цифрам, ...

До сортировки необходимо знать два параметра: k и m, где k - количество разрядов в самом длинном ключе; m - разрядность данных: количество возможных значений разряда ключа. Эти параметры нельзя изменять в процессе работы алгоритма.

Предполагается, что элементы линейного списка L есть k-разрядные десятичные числа, разрядность максимального числа известна заранее. Через $d(j,n)$ обозначается j-я справа цифра числа n. Пусть $L_0, L_1,..., L_9$ - вспомогательные списки (карманы), которые вначале пусты. Поразрядная сортировка состоит из двух процессов, называемых **распределение** и **сборка** и выполняемых для $j=1, 2,..., k$. Фаза распределения разносит элементы L по карманам: элементы l_i списка L последовательно добавляются в списки L_m, где $m=d(j, l_i)$. Таким образом получается десять списков, в каждом из которых j-тые разряды чисел одинаковы и равны m. Фаза сборки состоит в объединении списков $L_0, L_1,..., L_9$ в общий список $L = L_0 \Rightarrow L_1 \Rightarrow L_2 \Rightarrow \ldots \Rightarrow L_9$.

Поразрядная сортировка растет линейным образом по n, так как k, m - константы.

Данный тип сортировки является наиболее эффективным для компьютеров с **магистральной архитектурой**. Эти машины имеют несколько арифметических устройств и схему «опережения», так что обращения к памяти и вычисления могут в значительной степени совмещаться во времени.

3. Поиск

Предполагается, что имеется набор из N записей и задача состоит в нахождении одной из них. Также полагается, что каждая запись включает специальное поле, именуемое ее ключом. В общем случае требуется N различных ключей для того, чтобы каждый из них однозначно идентифицировал связанную с ним запись. Набор всех записей именуется таблицей или файлом. Алгоритм поиска имеет так называемый аргумент K, и задача заключается в нахождении записи, для которой K

служит ключом. Результатом поиска может быть одно из двух: либо поиск завершился успешно, и уникальная запись, содержащая *K*, найдена, либо поиск оказался неудачным, и запись с ключом *K* не найдена.

3.1. Последовательный поиск

Принцип данного способа поиска достаточно прост – «начать с начала и продолжать, пока не будет найден искомый ключ; затем остановиться». Эта последовательная процедура представляет собой очевидный путь поиска и может служить отправной точкой для рассмотрения множества алгоритмов поиска. Данный тип поиска осуществляется в неупорядоченной таблице.

3.2. Бинарный поиск

Самым простым способом поиска ключа в упорядоченном массиве является следующий: сравнить *K* со средним ключом в таблице, в результате этого сравнения определить, в какой половине таблицы находится искомый ключ, и снова применить ту же процедуру к половине таблицы. Таким образом, максимум за величину порядка log_2N сравнений искомый ключ будет либо найден, либо будет установлено, что его нет в таблице. Эта процедура известна как «логарифмический поиск» или «метод деления пополам», но чаще всего употребляется термин бинарный поиск.

3.3. Хеширование [15]

Рассмотренные ранее методы поиска основывались на сравнении данного аргумента *K* с ключами в таблице. При хешировании данную операцию заменяют арифметическими действиями над ключом *K*, позволяющими вычислить некоторую функцию *f(K)*. Последняя укажет адрес в таблице, где хранится *K* и связанная с ним информация.

[15] Составлено на основе материалов монографии [5]

Идея хеширования состоит в использовании некоторой частичной информации, полученной из ключа, в качестве основы поиска. С помощью ключа происходит вычисление **хеш-адреса** *h(K)* и он используется в дальнейшем для проведения поиска. Возможна ситуация, когда найдутся различные ключи K_i и K_j, имеющие одинаковое значение хеш-функции $h(K_i) = h(K_j)$. Такое событие именуется **коллизией** *(collision)*; для разрешения коллизий разработаны различные способы. При использовании хеширования необходимо решить две практически независимые задачи - выбрать хеш-функцию *h(K)* и способ разрешения коллизий.

Хеш-функции

Предполагается, что хеш-функция имеет не более *M* различных значений, причем для всех ключей *K* выполняется неравенство

$$O \le h(K) < M.$$

Многочисленные тесты показали хорошую работу двух основных типов хеш-функций, один из которых основан на делении, а другой - на умножении.

Метод деления: в качестве хеш-функции берется остаток от деления ключа *K* на *M*

$$h(K) = K \bmod M.$$

Как показывает практика, лучше всего в качестве *M* использовать простые числа.

Мультипликативная схема хеширования описывается немного сложнее. Пусть *w* - размер машинного слова, понимая под этим максимальное количество представляемых им значений. Данный метод состоит в выборе некоторой целой константы *A*, взаимно простой с *w*, после чего можно положить

$$h(K) = \left\lfloor M \left(\left(\frac{A}{w} K \right) \bmod 1 \right) \right\rfloor.$$

Одна из положительных сторон мультипликативной схемы заключается в отсутствии потери информации, т.е. в возможности восстановить значение

ключа *K* по значению функции *h(K)*. Так как *A* и *w* взаимно просты, то при помощи алгоритма Евклида можно найти константу *A'*, такую, что
A A' mod w = 1; отсюда следует, что *K = (A'(AK mod w)) mod w*.

Хорошие результаты для хеш-функций получаются в случае, когда *A/w* приближенно равно золотому сечению $\varphi^{-1} = \left(\sqrt{5}-1\right)/2 = 0.61803$. В этом случае последовательность значений *h(k), h(k+1), h(k+2)* ведет себя точно также как последовательность *h(0), h(1), h(2).*

Хеширование ключей, состоящих из нескольких слов

С ключами, состоящими из нескольких слов и с ключами переменной длины можно работать как с числами с многократной точностью и применять к ним все рассмотренные методы. Второй способ состоит в комбинировании слов в одно и в выполнении единственной операции умножения или деления, как описано выше. Эта комбинация может быть осуществлена путем сложения по модулю *w* или выполнения операции «исключающее или». Обе операции используют все биты обоих аргументов; «исключающее или», однако, предпочтительнее, поскольку позволяет избежать арифметического переполнения. Основной недостаток такого метода заключается в том, что указанные операции коммутативны, а значит, хеш-адреса *(X, Y)* и *(Y, X)* будут совпадать. Чтобы устранить этот недостаток, Г. Д. Кнотт предложил выполнять перед арифметической операцией циклический сдвиг.

Разрешение коллизий методом "цепочек"

Некоторые хеш-адреса могут быть одинаковыми для различных ключей. Очевидный путь решения этой проблемы - поддержка *M* связных списков, по одному для каждого возможного значения хеш-кода. После хеширования ключа можно выполнить последовательный поиск в списке с номером номер *h(K)+1*. На рисунке 22 приводится иллюстрация метода разрешения коллизий методом цепочек.

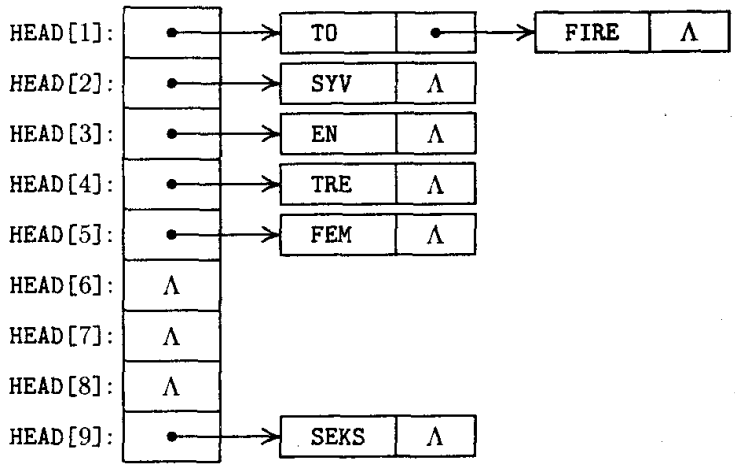

Рис. 22. Разрешение коллизий методом цепочек

Если цепочки коротки, то рассматриваемый метод является быстродействующим. В общем же случае, если имеется *N* ключей и *M* списков, средний размер списка будет равен *N/M*. Для повышения быстродействия желательны большие значения *M*, однако в этом случае многие списки будут пусты и будут зря расходовать память на содержание заголовков списков. Данный способ можно немного модифицировать, если производить сортировку в каждом из получаемых списков. Таким образом, поиск в них можно будет осуществлять одним из способов поиска путем сравнения ключей, а не простым последовательным поиском.

Разрешение коллизий методом открытой адресации

Другой путь решения проблемы, связанной с коллизиями, состоит в том, чтобы полностью отказаться от ссылок, и просматривать различные записи таблицы одну за одной до тех пор, пока не будет найден ключ *K* или пустая позиция. Идея заключается в формулировании правила, согласно которому по данному ключу *K* определяется «пробная последовательность», т.е. последовательность позиций таблицы, которые должны быть просмотрены при вставке или поиске *K*. Если при поиске *K* согласно определенной этим ключом последовательности встречается пустая позиция, можно сделать вывод о

том, что *K* в таблице отсутствует. Этот общий класс методов назван **открытой адресацией**.

Простейшая схема открытой адресации, известная как **линейное исследование,** использует циклическую последовательность проверок *h(K), h(K)-1, ..., 0, M - 1, M - 2, ..., h(K) + 1.*

Данный алгоритм хорошо работает в начале заполнения таблицы, однако по мере заполнения процесс замедляется, а длинные серии проб становятся все более частыми. Так, среднее количество проб, требуемое данным методом составляет:

$$C'_N \approx \frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right), \qquad \text{(неудачный поиск);}$$

$$C_N \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right), \qquad \text{(удачный поиск),}$$

где $\alpha = N/M$ – коэффициент заполненности таблицы.

Одним из способов защиты от последовательных хеш-кодов состоит в установке способа обхода элементов не по одному, т.е. *i:=i-1*, а сразу на несколько шагов: *i:=i-C*. Можно использовать любое положительное значение *C*, взаимно простое с *M*, поскольку последовательность проб в этом случае будет проверять все позиции таблицы без исключения. Такое изменение немного замедлит работу программы.

Хотя фиксированное значение *C* не приводит к устранению длинных серий проб, можно существенно улучшить ситуацию, сделав *C* зависящим от *K*. Эта идея позволяет выполнить важную модификацию алгоритма.

Данная модификация называется **открытая адресация с двойным хешированием**. Этот алгоритм почти идентичен предыдущему алгоритму, но проверяет таблицу немного иначе, используя две хеш-функции: $h_1(K)$ и $h_2(K)$. Как обычно, значения $h_1(K)$ находятся в диапазоне от *0* до *M - 1* включительно; функция $h_2(K)$ должна порождать значения от *1* до *M - 1*, взаимно простые

с *M*. Сравнение проб осуществляется путем сдвига переменной, отвечающей за текущий сдвиг на значение $h_2(K)$.

Существует несколько различных вариантов для вычисления $h_2(K)$. Если *M* - простое число и $h_1(K) = K \bmod M$, то можно положить $h_2(K)=1+(K \bmod (M-1))$; однако, поскольку *M-1* четно, можно положить $h_2(K) = 1 + (K \bmod (M - 2))$. Кроме того, можно взять в качестве $h_2(K)$ величину $1+([K/M] \bmod (M - 2))$ в связи с тем, что частное $[K/M]$ может быть получено в регистре как побочный результат вычисления $h_1(K)$.

Если $M=2^m$ и используется мультипликативное хеширование, то $h_2(K)$ может быть вычислена методом простого сдвига на *m* дополнительных битов влево с выполнением операции «ИЛИ» с 1. Эти операции выполняются быстрее метода деления.

Можно также допустить зависимость $h_2(K)$ от $h_1(K)$, как было предложено Гари Кноттом. Например, при простом *M* можно положить

$$h_2(K) = \begin{cases} 1, & h_1(K) = 0; \\ M - h_1(K), & h_1(K) > 0. \end{cases}$$

Этот метод выполняется быстрее повторного деления, однако он вызывает определенную вторичную кластеризацию, что приводит к небольшому увеличению числа проб из-за повышения вероятности того, что два или несколько ключей пойдут одному и тому же пути.

Для приведенной выше формулы вычислены средние значения проб при неудачном и удачном поисках:

$$C'_N \approx (1-\alpha)^{-1} - \alpha - \ln(1-\alpha), \qquad \text{(неудачный поиск);}$$

$$C_N \approx 1 - \ln(1-\alpha) - \frac{1}{2}\alpha, \qquad \text{(удачный поиск).}$$

4. Задачи для лабораторных работ

1. Составить программу, решающую системы линейных уравнений методом Гаусса. Исходная матрица уравнения является треугольной. Для хранения структуры данных использовать одномерные массивы.

Примечание: Программа не должна оперировать с двумерными массивами

2. Написать программу, производящую арифметические операции с помощью польской бесскобочной записи. Для хранения чисел использовать стеки.

Примечание: Обычная запись числа: *(10+2)*5*. Польская бесскобочная запись для данного выражения имеет вид: *10 ↑2 ↑+ 5 ↑**, где символом ↑ обозначается помещение элемента в стек. Программа должная выглядеть следующим образом. На экране в любой момент времени можно ввести либо какое-либо число, либо арифметическую операцию (+,-,*,/). Если вводится число, то оно автоматически помещается в стек. Если вводится какая-либо арифметическая операция, то должны извлечься два последних числа из стека, произвестись над ними арифметические действия и затем результат помещается в стек и выводится на экран. Необходимо предусмотреть возможность предупреждения пользователя, когда он собирается выполнить какое-либо арифметическое действие над двумя числами, а в стеке только одно число, например: *10 ↑2 ↑+ 5 ↑** - +.

3. Написать программу, которая умеет вставлять, удалять элементы из бинарного упорядоченного дерева, а также отображать текущую структуру дерева.

Примечание: Отображать структуру дерева можно в любом виде, однако самым простым способом представляется отображение дерева в режиме каталога

4. На основе программы 3 реализовать один из алгоритмов балансировки деревьев.

Примечание: Балансировка дерева должна осуществляться на момент вставки. Удаление элементов из сбалансированного дерева можно опустить. Также в программе должна быть предусмотрена возможность отображения текущей конфигурации дерева.

5. Реализовать сравнительный анализ не менее 5 способов сортировки, которые принадлежат не менее четырем семействам методов (сортировка путем вставок, обменная сортировка, сортировка посредством выбора, сортировка методом слияния, сортировка методом распределения).

Примечание: Для сравнения методов должен использоваться один и тот же массив случайных чисел. Количество элементов в массиве должно задаваться с клавиатуры. Результатом работы программы должна быть таблица, в которой приводятся реализованные методы сортировок и затраченное ими машинное время на выполнение алгоритма.

6. Написать программу, на вход которой подается текстовый файл. Входной текстовый файл должен быть разобран по словам, которые необходимо поместить в хеш-таблицу. Программа должна уметь запрашивать слова у пользователя и отвечать на вопрос: есть ли текущее слово в тексте или оно отсутствует.

Заключение

Высокопроизводительные алгоритмы обработки информации непрерывно совершенствуются. Ежегодно в научных журналах публикуются новые способы поиска информации, анализируются достоинства и недостатки предлагаемых алгоритмов.

Целью данного учебного пособия является не только знакомство с базовыми алгоритмами, но и выработка у студентов творческого мышления и способности к критическому анализу.

После знакомства с основными алгоритмами обработки информации, необходимо переходить к рассмотрению современных способов разработки программного обеспечения – объектно-ориентированному проектированию и технологии *RUP*.

Библиографический список

1. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. 536 с.

2. Вирт Н. Алгоритмы и структуры данных. М.: Мир, 1989. – 360 с.

3. Далека В. Д., Деревянко А.С., Кравец О.Г., Тимановская Л.Е. Модели и структуры данных. Харьков: ХГПУ, 2000. 241с.

4. Кнут Д. Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: пер. с англ.: уч. пос. М.: Издательский дом “Вильямс” , 2000. 720 с.

5. Кнут Д. Искусство программирования, том 3. Сортировка и поиск, 2-е изд.: пер. с англ.: уч. пос. М.: Издательский дом “Вильямс” , 2000. 832 с.

6. Стивенс Р. Delphi. Готовые алгоритмы. М.:ДМК Пресс, 2004. 384 с.

7. Разбор арифметического (и не только) выражения. Классические алгоритмы. [Электронный ресурс]. 2014. Дата обновления: 09.02.2014. URL: http://algolist.manual.ru/syntax/parsear.php (дата обращения: 09.02.2014).

8. Общие алгоритмы сжатия и кодирования. [Электронный ресурс]. 2014. Дата обновления: 09.02.2014. URL: http://algolist.manual.ru/compress/-standard/index.php (дата обращения: 09.02.2014).

Учебное издание

Крутов Алексей Николаевич, **Федина** Мария Ефимовна

МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Публикуется в авторской редакции
Титульное редактирование *Л. А. Кузнецовой*
Компьютерная верстка, макет *Н. П. Бариновой*

Подписано в печать 25.11.2014. Формат 60x84/16. Бумага офсетная. Печать оперативная.
Усл.-печ. л 2,8; уч.-изд. л. 3,0. Гарнитура Times.
Тираж 100 экз. Заказ № 2568.
Издательство «Самарский университет», 443011, г. Самара, ул. Акад. Павлова, 1.
Тел. 8 (846) 334-54-23.
Отпечатано на УОП СамГУ.