Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

В.В. ЛАНДОВСКИЙ

СТРУКТУРЫ ДАННЫХ

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2016

УДК 004.422.63(075.8)
Л 222

Рецензенты:
канд. техн. наук, доцент *В.А. Астапчук*;
канд. техн. наук, доцент *Т.А. Кондратьева*

Работа подготовлена на кафедре автоматизированных систем управления для студентов III курса АВТФ направления 09.03.01 – «Информатика и вычислительная техника»

Ландовский В.В.

Л 222 Структуры данных: учеб. пособие / В.В. Ландовский. – Новосибирск: Изд-во НГТУ, 2016. – 68 с.

ISBN 978-5-7782-3080-4

В пособии рассмотрены наиболее распространенные структуры данных – от обычных массивов до сложных многосвязных динамических структур. Рассмотрена концепция абстрактных типов данных, описаны основные АТД. Описание сопровождается примерами программных реализаций. Кратко изложены методы оценки временной и пространственной сложности алгоритмов. Отдельная глава посвящена сбалансированным деревьям и хеш-таблицам.

УДК 004.422.63(075.8)

ISBN 978-5-7782-3080-4

ОГЛАВЛЕНИЕ

ВМЕСТО ПРЕДИСЛОВИЯ

В XXI веке стало возможным быть квалифицированным, востребованным на рынке труда программистом, не имея при этом точного представления о том, как организованы данные в памяти вычислительной машины. Виной тому непрерывно развивающиеся программные средства, призванные ускорить и упростить процесс разработки программ. При использовании тяжелой артиллерии, такой как Microsoft Visual studio, Embarcadero RAD studio и им подобных на первый план выходит знание высокоуровневых средств, т. е. заранее созданных и отлаженных компонентов, из которых будет построена программа. Большинство проблем хранения и доступа к данным уже решено, и для достижения цели остается только реализовать необходимую логику поведения конкретного прикладного решения. Привлекательность такого подхода не вызывает сомнений – минимум действий для достижения результата. А это, в свою очередь, означает и уменьшение количества ошибок, и снижение требований к подготовке кадров.

Все замечательные плоды технического прогресса, двигателем которого, как известно, является лень, нужно использовать с осторожностью. В противном случае есть риск оказаться в зависимом положении и причинить вред здоровью. Среда программирования высокого уровня не исключение: одно лишь умение применять макросредства не дает достаточной глубины понимания полученного в результате программного кода. Вследствие этого эффективность разработанных программ как по времени выполнения, так и по затраченным объемам памяти может быть непредсказуемой.

Данная работа адресована в первую очередь студентам-специалистам, начинающим свой путь в области компьютерных наук, но уже имеющим представление о теоретических основах информатики

и владеющих базовыми навыками программирования с использованием языков высокого уровня, а также всем, кто интересуется одним из основополагающих вопросов построения программ – проектированием структур данных – и не против погрузиться вместе с автором в этот увлекательный мир. В работе рассмотрены структуры данных, давно ставшие классикой для большинства программистов, обсуждаются некоторые особенности их применения, приводятся примеры реализаций.

1. СТРУКТУРЫ И ТИПЫ ДАННЫХ (АБСТРАКТНЫЕ И НЕ ОЧЕНЬ)

При внимательном изучении литературы, посвященной структурам данных, нетрудно заметить путаницу в терминологии. Как отличить «структуру данных» от «абстрактного типа данных» (АТД), и как эти понятия соотносятся с термином «тип данных», что такое дерево-АТД или структура данных, можно ли рассматривать в одной категории массив, стек, очередь и список? Следует определиться с терминами, не претендуя на отыскание единственно правильных ответов на подобные вопросы, но для того, чтобы исключить двусмысленность при дальнейшем изложении.

Прежде всего, обратимся к фундаментальному для программистов понятию – «тип данных». Под ***типом данных*** принято понимать форму представления данных, которая характеризуется способом организации данных в памяти, множеством допустимых значений и набором возможных операций [1]. Типы данных обычно разделяют на ***базовые*** и ***производные*** – сконструированные программистом на основе базовых. Можно сказать, что набор базовых типов продиктован выбранным языком программирования. Если посмотреть шире, то особенности базовых типов напрямую связаны с аппаратной частью вычислительных машин – на низком уровне выбираются такие способы организации данных, с которыми наиболее эффективно работает процессор. Поэтому в большинстве языков базовые типы похожи, большинство из них – это формы представления чисел, целых и вещественных, различающихся разрядностью, наличием или отсутствием знака. Эту группу называют арифметическими типами, по критерию допустимых операций в некоторых языках сюда же можно отнести символьные и

логические типы. Наиболее известные производные типы данных – это массивы и структуры (записи). Массив представляет собой последовательность однотипных элементов, расположенных в памяти непосредственно друг за другом. Структуры (записи) объединяют элементы различных типов, последовательность размещения в различных языках может отличаться.

Несмотря на то, что производные типы данных могут иметь сложную внутреннюю структуру (т. е. включать элементы других типов), они все же являются статическими – неизменными на всем протяжении выполнения программы. Тип данных – это, прежде всего, способ интерпретации данных, содержащихся в некоторой области памяти. Пользуясь только возможностями конструирования новых типов данных, затруднительно или вовсе невозможно решать задачи, в которых необходимо работать с множеством взаимосвязанных элементов.

Термин «структура данных» не всегда воспринимается однозначно. Само по себе слово «структура» означает внутреннее устройство чего-либо, и в этом смысле все типы данных имеют свою структуру. При этом можно без сомнения отнести к структурам данных массивы, списки или деревья, а базовые типы в эту категорию не попадут. Выделим главные особенности ***структур данных***. Во-первых, будем использовать этот термин, когда речь идет о множестве элементов, во-вторых, будем считать обязательным наличие прямого отображения логической взаимосвязи элементов на физическом уровне, т. е. на уровне размещения данных и связей между ними в памяти. Механизмы реализации таких связей в языках программирования различны. Наиболее естественным выглядит явное использование указателей – типа данных, содержащего адрес ячейки памяти как в C++ или Pascal. В языке C# указатели не используются, однако существует понятие ссылочного типа данных, благодаря которому несколько переменных могут ссылаться на один и тот же объект. Если язык не предоставляет никаких возможностей для альтернативного обращения к ранее созданным объектам, возможно использование курсоров – целочисленных переменных, которые хранят индекс элемента в массиве. В этом случае появляется естественное ограничение – все элементы данных должны размещаться в массиве; по сути, это можно назвать эмуляцией работы с памятью.

Осталось определить последний термин – «абстрактный тип данных», сделаем это в контексте ответа на вопрос о возможности ставить в один ряд стек, очередь и список. Отличительная особенность ***абстрактного типа данных*** состоит в том, что он предоставляет интерфейс (набор функций) для работы с данными определенного типа [2], в то время как его внутренняя структура (особенности размещения и взаимосвязи элементов данных) остается закрытой и может быть реализована по-разному. Стек и очередь представляют собой линейную последовательность элементов, отличаются они только принципами работы операций добавления и извлечения, при этом абсолютно неважно, каким образом элементы будут храниться (будут ли они взаимосвязаны или расположены последовательно). Из этого следует, что стек и очередь – это абстрактные типы данных. Для списков характерно наличие определенных взаимосвязей элементов, поэтому список – это структура данных. Справедливости ради следует заметить, что если смотреть на список только с точки зрения обеспечения некоторого набора операций, то правильнее будет говорить о нем как об АТД.

2. АНАЛИЗ АЛГОРИТМОВ (АСИМПТОТИКА ПРОТИВ ТОЧНОСТИ)

Эффективность работы программ во многом зависит от выбора структур данных [3, 4]. Это утверждение сформулировано в те далекие времена, когда держа в руках элементы памяти вычислительных машин, можно было невооруженным глазом различить отдельные биты. С тех пор принципы конструирования вычислительной техники существенно изменились, а основные положения информатики по-прежнему актуальны.

При изучении структур данных полезно оговаривать их преимущества и недостатки в контексте решения конкретных практических задач. Поэтому вначале нужно определить критерии эффективности алгоритмов и научиться их сравнивать.

Основных критериев оценки эффективности алгоритмов два – временная и пространственная сложность. ***Временная сложность*** измеряется количеством элементарных операций, совершаемых для решения поставленной задачи. ***Пространственная сложность*** – это объем затраченной алгоритмом памяти. Чтобы сравнить алгоритмы по временной сложности, достаточно подсчитать количество операций, производимых каждым из них. В общем случае результатом подсчета будет не числовая константа, а функция от объема входных данных. Проиллюстрируем это на простом примере: поиске элемента массива по значению, фрагмент алгоритма представлен на рис. 2.1.

Учитывая, что обращение к элементу массива не зависит от его длины и может считаться элементарной операцией, в случае отсутствия элемента с искомым значением алгоритм выполнит $(4n + 2)$ операций (присвоение начального значения индекса, n обращений к элементам, n сравнений, n увеличений индекса, n проверок достижения максимального индекса, вывод результата). В случае, если искомый элемент есть и находится в середине массива, то $(2n + 2)$, а если в начале – операций будет всего две. Как можно увидеть, целесообразно

различать оценки в ***лучшем*** и ***худшем случаях***. Наибольший интерес представляют оценки в худшем случае, так как дают информацию о максимальном количестве операций, однако их улучшение не всегда гарантирует увеличение производительности. Для оценки ***сложности алгоритма в среднем*** необходимо вводить вероятностную модель входного потока [5].

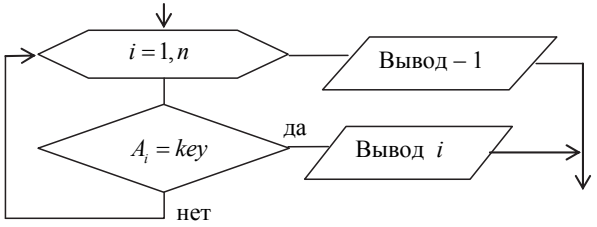


Рис. 2.1. Алгоритм поиска

Точные функции сложности алгоритмов применяются редко, главной причиной этого является их избыточность в сочетании с трудоемкостью определения. При значительном увеличении объема входных данных вклад постоянных множителей и слагаемых низших порядков становится ничтожным. На первый план выходят ***асимптотические оценки сложности***. Перечислим основные асимптотические обозначения, использующиеся при анализе алгоритмов:

1) O – верхняя асимптотическая оценка;

2) Ω – нижняя асимптотическая оценка;

3) Θ – верхняя и нижняя асимптотические оценки.

Формальные определения можно привести, приняв, что $f(n)$ и $g(n)$ – положительные функции положительного аргумента, так как и количество объектов на входе, и количество операций или памяти – числа положительные.

$f(n) = O(g(n))$ или $f(n) \in O(g(n))$ означает, что существуют такие положительные C и n_0, что для любого $n > n_0$ функция $f(n) \le Cg(n)$. Запись $O(g(n))$ обозначает класс таких функций, что все они растут не быстрее, чем функция $g(n)$ с точностью до постоянного множителя. Оценка $\Omega(g(n))$ определяет класс функций, которые растут не медленнее, чем $g(n)$ с точностью до постоянного множителя, т. е. $f(n) \in \Omega(g(n))$ означает $\exists C > 0, n_0 > 0 : \forall (n > n_0) \quad Cg(n) \le f(n)$. Оценку $f(n) \in \Theta(g(n))$ называют асимптотически точной оценкой

функции $f(n)$, так как по определению функция $f(n)$ не отличается от функции $g(n)$ с точностью до постоянного множителя, т. е.

$$\exists C_1 > 0, C_2 > 0, n_0 > 0 : \forall (n > n_0), \quad C_1 g(n) \le f(n) \le C_2 g(n) .$$

Пример того, как могут выглядеть графики f и g в случае различных оценок приведен на рис. 2.2.

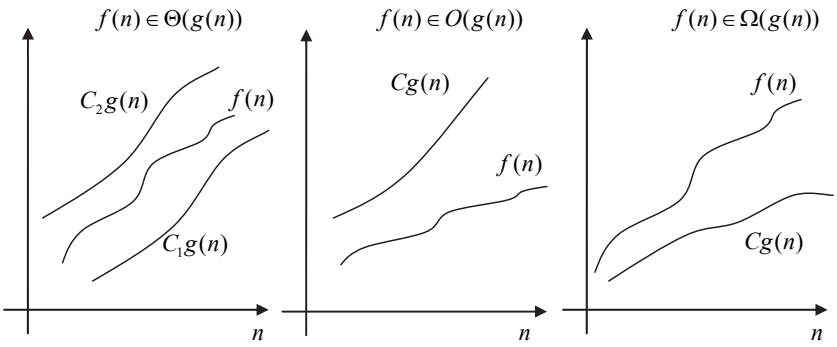


Рис. 2.2. Асимптотические оценки

Очевидно, что оценка Θ дает более точное представление о поведении функции, чем оценка O, однако в большинстве случаев используют именно верхнюю оценку. Рассмотрим основные правила асимптотического анализа.

1. Постоянный множитель отбрасывается: $O(Cf(n)) = O(f(n))$, где C – константа.

2. Оценка сложности произведения или частного двух функций равна произведению или частному их сложностей соответственно: $O(g(n)f(n)) = O(f(n))O(g(n))$, $O(g(n)/f(n)) = O(f(n))/O(g(n))$.

3. Оценка сложности суммы функций определяется как максимальная из оценок слагаемых: $O(g(n) + f(n)) = \max[O(f(n)), O(g(n))]$.

Если, например, алгоритм выполняет $2n^2 + 5n + 6$ операций, то его сложность следует оценить как $O(n^2)$, однако следуя формальной логике, утверждение, что сложность этого алгоритма $O(n^3)$ или $O(2^n)$, не будет ошибочным, так как эти функции растут быстрее, чем n^2.

Говоря о сложности алгоритмов, выделяют классы сложности, к наиболее распространенным из них можно отнести:

$O(1)$ – константная сложность;

$O(n)$ – линейная сложность;

$O(\log(n))$ – логарифмическая сложность;

$O(n^2)$ – квадратичная сложность;

$O(n^c)$ – полиномиальная сложность, где c – константа ($c > 1$);

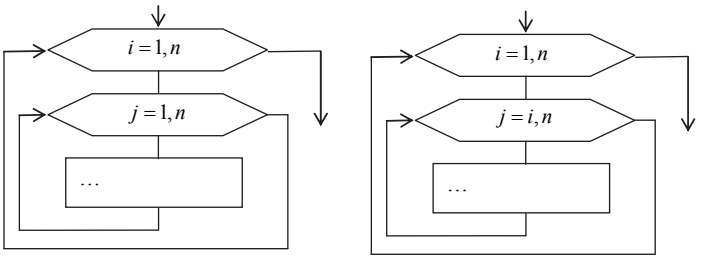
$O(c^{f(n)})$ – экспоненциальная сложность, где c – константа ($c > 1$), $f(n)$ – полиномиальная функция;

$O(n!)$ – факториальная сложность.

В логарифмических оценках основание логарифма не указывают, и связанно это, в первую очередь, не с тем, что большинство таких оценок подразумевает основание 2, а со следующим свойством логарифмов $\log_a(x) = \dfrac{\log_b(x)}{\log_b(a)}$. Смена основания логарифма эквивалентна изменению константы.

Перечисленные выше правила упрощают оценку сложности алгоритма. Алгоритм, не содержащий циклов и рекурсий, имеет константную сложность, сложность цикла зависит от количества итераций, рекурсии – от ее глубины. Два вложенных цикла, как правило, порождают квадратичную сложность. Итоговая оценка двух последовательных фрагментов алгоритма будет совпадать с наибольшей из оценок.

Рассмотрим несколько примеров. Вложенные циклы: обе конструкции, приведенные на рис. 2.3, имеют квадратичную сложность относительно n, в случае рис. 2.3, *а* это очевидно, рис. 2.3, *б*, возможно, нуждается в доказательстве.

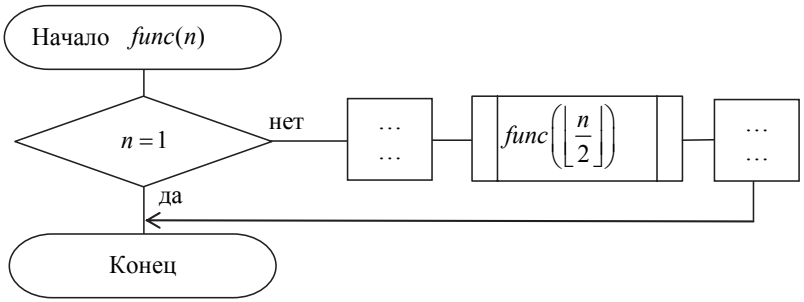


а *б*

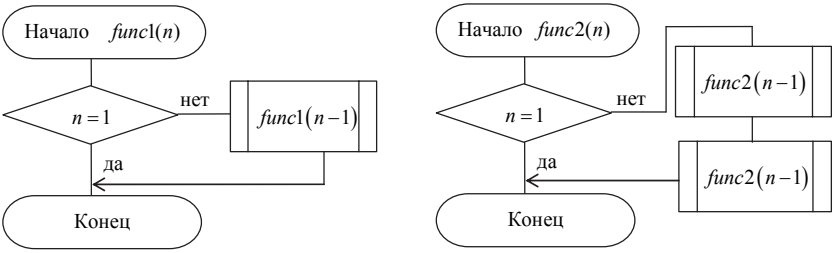
Рис. 2.3. Вложенные циклы

Количество итераций внутреннего цикла образует арифметическую прогрессию, сумма которой равна $\frac{n+1}{2}n$, следовательно, сложность алгоритма $O\left(\frac{n^2}{2}+\frac{n}{2}\right)=O(n^2)$.

Рекурсия: алгоритм, представленный на рис. 2.4, *а*, принимает на входе целочисленный параметр *n*, глубина рекурсии $\log_2(n)$, следовательно, сложность алгоритма $O(\log(n))$. Рассмотрим два содержащих рекурсию алгоритма, представленных на рис. 2.4, *б*, *в*.



а



б *в*

Рис. 2.4. Примеры рекурсии

Очевидно, что вариант (*а*) имеет сложность $O(n)$, в случае (*б*) каждый вызов порождает две ветки, рекурсия разворачивается в виде дерева с общим количеством вершин $(2^n - 1)$, следовательно, алгоритм имеет сложность $O(2^n)$.

Оценка Ω используется, когда утверждают, что алгоритм или группа алгоритмов не может работать быстрее. Например, сложность алгоритма поиска максимального элемента в неупорядоченном массиве есть Ω(*n*), где *n* – размер массива. Можно доказать, что сложность алгоритмов сортировки, основанных на попарном сравнении элементов, есть Ω(*n* log(*n*)). То же самое можно выразить через верхнюю оценку, сказав, что алгоритм не может работать лучше, чем *O*(…).

Основные подходы к оценке пространственной сложности аналогичны применяемым для временных оценок. Выбор тех или иных структур данных для хранения информации, использование вспомогательных структур для ускорения доступа и улучшения временных характеристик напрямую влияет на необходимое количество памяти. Как правило, улучшение одних оценок ведет к ухудшению других.

В заключение хотелось бы обратить внимание на одну особенность асимптотических оценок, рассмотрим ее на примере: справедливо утверждение $n^{32} \in O(1,01^n)$. Полином, разумеется, растет медленнее экспоненты, но можно ли на практике ожидать таких высоких значений *n*, при которых экспонента, даже умноженная на большую константу, обгонит полином такой высокой степени. Этот искусственный пример должен показать, что нельзя однозначно предпочитать алгоритмы с более хорошей асимптотикой, всегда нужно помнить об особенностях решаемой задачи и о константах, которые мы отбрасываем. Например, если *n* не превосходит десять, то линейный алгоритм, в котором скрывается множитель, превосходящий двадцать, уступает квадратичному с константой менее двух.

3. СТРУКТУРЫ ДАННЫХ (ОТ ПРОСТОГО К СЛОЖНОМУ)

В этом разделе рассмотрены основные структуры данных, т. е. те, которые применяются наиболее часто. Для решения практических задач бывает достаточно самых примитивных структур данных, а в других случаях возникает необходимость комбинировать наиболее сложные подходы к представлению данных. Хорошее знание устройства базовых структур и особенностей их применения позволяет принимать правильные решения еще на этапе проектирования программ.

При обсуждении структур данных наибольший интерес представляют связи между элементами. Для наглядной иллюстрации на схемах используются стрелки, которые проводятся из ячеек указателей, курсоров или ссылок на другие элементы к ячейкам этих элементов.

Примеры приведенных программных реализаций выполнены с использованием языков C/C++.

3.1. МАССИВЫ

Массив – это последовательность однотипных элементов, расположенных непосредственно друг за другом (рис. 3.1).

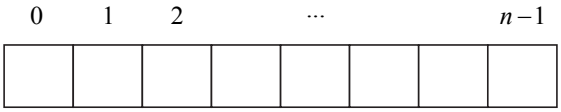


Рис. 3.1. Массив

Связей между элементами массива нет, положение элемента однозначно определяется его индексом, так как все элементы одного типа и, следовательно, занимают один и тот же объем памяти. Благодаря таким особенностям обращение к элементу по индексу занимает постоянное время. В языках, поддерживающих указатели, работать с

массивом можно, передвигая указатель на величину индекса, умноженную на размер элемента.

Обычно массив – это статическая структура, т. е. количество элементов неизменно на всем протяжении работы программы, но иногда говорят о динамических массивах. Динамическим называют массив, который определяют не на этапе компиляции программы, а в процессе ее выполнения. На протяжении работы с динамическим массивом его размер, как правило, также остается постоянным. В языке Си присутствуют возможности расширения выделенной области памяти, однако результатом выполнения операции может стать перемещение содержимого блока памяти на новое место, поэтому применяется она редко. Пример создания динамического массива показан на рис. 3.2, *а*, а схема размещения данных в памяти – на рис. 3.2, *б*.

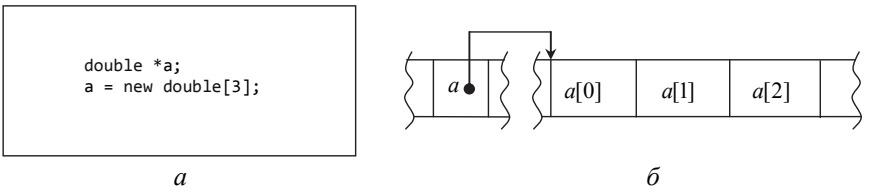


Рис. 3.2. Динамический массив

Массивы могут быть многомерными, это означает, что элемент массива сам является массивом. Схема размещения двумерного массива в памяти представлена на рис. 3.3.

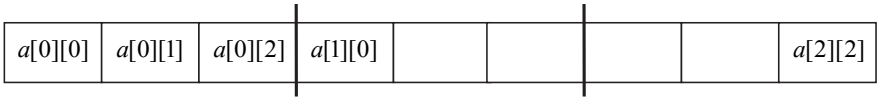


Рис. 3.3. Двумерный массив

Логический порядок элементов в массиве полностью совпадает с физическим. Чтобы изменить позицию элемента, его требуется копировать. В качестве примера можно рассмотреть фрагмент программы, выполняющий перестановку первой и последней строк квадратной матрицы размера 3×3, он представлен на рис. 3.4.

Если элемент массива является указателем, то говорят о «массиве указателей». Такая структура обладает преимуществом, когда элементы данных, занимающие большой объем памяти, предполагается часто менять местами. Перестановки строк матрицы, характерные для неко-

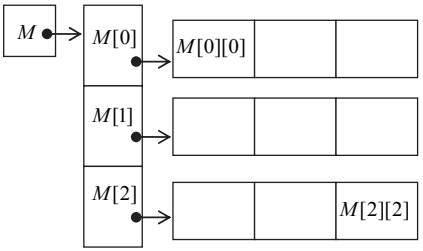
торых численных методов, можно реализовать без копирования элементов, если реализовать матрицу с помощью массива указателей, по каждому из которых будет расположен массив с соответствующей строкой матрицы. В примере реализации, показанном на рис. 3.5, *а*, перестановка, аналогичная предыдущему примеру, – это последние три строки программы, цикл отвечает за создание динамической структуры, схема которой приведена на рис. 3.5, *б*.

```
double M[3][3] = { { 1, 1, 1 }, { 2, 2, 2 }, { 3, 3, 3 } }, tmp;
for (int j = 0; j < 3; j++)
{
tmp = M[0][j];
M[0][j] = M[2][j];
M[2][j] = tmp;
}
```

Рис. 3.4. Перестановка строк матрицы

```
double **M = new double* [3],*tmp;
for (int i = 0; i < 3; i++)
{
       M[i] = new double[3];
       for (int j = 0; j < 3; j++)
              M[i][j] = i + 1;
}
tmp = M[0];
M[0] = M[2];
M[2] = tmp;
```

а



б

Рис. 3.5. Реализация матрицы с помощью массива указателей

Как можно видеть, меняют положение только указатели, элементы данных остаются на прежнем месте.

Массивы можно применять и в ситуациях, когда количество элементов меняется. В этом случае размер массива должен превосходить максимальное количество элементов, а для определения текущего количества занятых ячеек можно использовать отдельный счетчик – индекс последнего занятого элемента. Если возможно обойтись добавлением элементов в конец и не требуется удаление, то такая реализация будет вполне эффективной. Если же необходимо удалять или вставлять элементы между другими, то массив – плохое решение. При удалении еще можно использовать специальное значение, которое будет интерпретироваться как пустое, но при вставке не избежать сдвига всех элементов, находящихся правее добавленного. Чтобы массив оставался именно

массивом, операции добавления и удаления в общем случае должны сдвигать элементы, и время их выполнения составит $O(n)$, где n – количество элементов. Для достижения константного времени этих операций требуется принципиально иная организация данных.

3.2. СПИСКИ

Структура данных, о которой ниже пойдет речь, имеет более точное название – «связный список», основной ее особенностью являются интегрированные с элементами данных поля, отвечающие за связи.

3.2.1. ОДНОСВЯЗНЫЕ СПИСКИ

В зависимости от количества и направления связей выделяют различные типы списков, наиболее простым является односвязный список, его схема изображена на рис. 3.6. На схеме последняя стрелка упирается в вертикальную линию, это означает равенство указателя значению, определяющему пустой указатель.

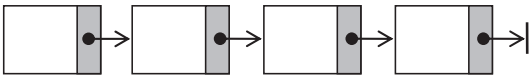


Рис. 3.6. Односвязный список

В зависимости от решаемой задачи для доступа к списку используют один или несколько отдельных указателей, обычно это указатели на первый и последний элементы.

Вставка элемента в любое место при условии, что это место заранее определено, выполняется за $O(1)$, так как требуется только разорвать прежние связи и создать новые. Пример вставки приведен на рис. 3.7, новые связи обозначены пунктирными стрелками.

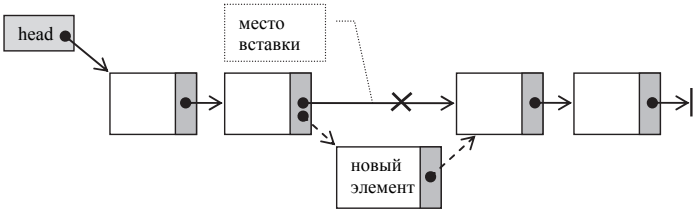


Рис. 3.7. Вставка элемента в список

Асимптотически время вставки элемента на заданную позицию останется таким же как у массива, но трудоемкость уменьшится, так как в случае массива требуется ***копирование его элементов*** – сдвиг справа, а для списка доступ к интересующей позиции предполагает последовательное перемещение от начала – ***переход по указателям*** слева от места вставки. Пример реализации такого алгоритма показан на рис. 3.8.

```
struct SomeData
{
	int data_n;
	char data_s[128];
	bool data_b;
};
struct LElem
{
	SomeData D;	// Данные.
	LElem *next;	// Указатель на следующий элемент.
};
LElem* insert(LElem *head, int pos,SomeData newData) // Функция вставки в список начинающийся
{	// по указателю head, на позицию pos (начальная позиция 0), элемента с данными заданными
	// параметром newData.
	LElem *newElem = new LElem; // Выделение памяти для нового элемента списка.
	newElem->D = newData; // Копирование данных.
	if (pos == 0)			// Вставка в позицию ноль выполняется всегда
	{				// и возвращает изменившийся адрес начала списка.
		newElem->next = head;
		return newElem;
	}
	LElem *Current = head;
	int index=0;
	while (index != (pos-1))		// Поиск элемента, после которого должен
	{					// быть добавлен новый. Если текущее количество
		if (Current == NULL)		// элементов меньше позиции - выход из функции.
			return head;		// При выходе из функции возвращается указатель
		Current = Current->next;	// на начало списка, во всех случаях кроме pos=0
		index++;			// он остается прежним.
	}
	LElem *tmp = Current->next; // Вставка newElem между Current и Current->next.
	Current->next = newElem;
	newElem->next = tmp;
	return head;
}
int _tmain(int argc, _TCHAR* argv[])
{
	LElem *h = NULL;
	h = insert(h, 0, { 10, "string 1", true });
	h = insert(h, 1, { 20, "string 2", true });
	h = insert(h, 2, { 30, "string 3", true });
```

Рис. 3.8. Добавление в список

В отличие от массива списки допускают добавление элементов с обоих концов за время $O(1)$. При работе с односвязным списком для этого требуется контролировать положение последнего элемента с помощью отдельного указателя, как показано на рис. 3.9.

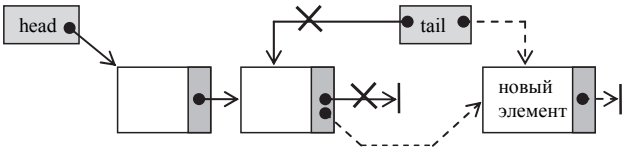


Рис. 3.9. Вставка элемента в конец списка

С увеличением количества вспомогательных переменных реализация функций, работающих со структурой данных, становится громоздкой. Проблема заключается еще и в том, что значения этих переменных могут меняться внутри функции, и потребуется сообщить вызывающей функции не одно, а несколько изменений. Для этого можно передавать указатели по ссылке. Изменения в рассмотренной реализации добавления показаны на рис. 3.10. Вместо возврата указателя на новое положение начала списка функция возвращает код ошибки или успеха операции, а входной параметр *head* одновременно является выходным за счет механизма передачи по ссылке.

```
int insert(LElem *&head, int pos, SomeData newData)
{               ....
                head = newElem;
                return 0;       // Возврат кода успешного завершения.
                ....
                        return 1;               // Возврат кода ошибки.
                ....
        return 0;       // Возврат кода успешного завершения.
}
int _tmain(int argc, _TCHAR* argv[])
{
        LElem *h = NULL;
        if (insert(h, 0, { 10, "string 1", true }) != 0)
        {
                printf("ERROR");
                ....
        }
```

Рис. 3.10. Передача параметра по ссылке при добавлении в список

Следует отметить, что реализация таких структур с использованием объектно-ориентированного подхода выглядит привлекательней. Проиллюстрируем это на примере односвязного списка с методами добавления в начало, добавления в конец и удаления элемента по номеру (рис. 3.11). Использование переменной *count*, содержащей текущее количество элементов, позволит на первом этапе удаления выявить некорректно заданный индекс.

```
class List1
{
	LElem *head, *tail;
	int count;
public:
	List1(){ tail = head = NULL; count = 0; }
	~List1(){ while (count > 0) Del(0); }
	void InsetHead(SomeData newData){
		LElem *tmp = head;
		LElem *newElem = new LElem;
		newElem->D = newData;
		head = newElem;
		newElem->next = tmp;
		if (tmp == NULL)		// Если до операции список был пуст, то
			tail = newElem;		// указатель на последний элемент необходимо изменить,
		count++;			// установив его на добавленный элемент.
	}
	void InsetTail(SomeData newData){
		LElem *newElem = new LElem;
		newElem->D = newData;
		newElem->next = NULL;
		if (tail == NULL)		// Если до операции список был пуст, то необходимо
			head = tail = newElem;	// изменить указатели на первый и последний элемент.
		else			{
			tail->next = newElem;	// В противном случае меняется только поле next
			tail = newElem;		// последнего элемента и сам указатель tail.
		}
		count++;
	}
	int Del(int index)	{
		if (index > (count - 1)) // Если индекс выходит
			return 1;	// за границы - возврат кода ошибки.
		LElem *tmp;
		if (index == 0) {// Если требуется удалить первый элемент.
			tmp = head;
			if (head == tail) // Если первый элемент единственный.
				head = tail = NULL;
			else
				head = head->next;
		}
		else			{
			LElem *Current = head;
			int cur_index = 0;
			while (cur_index != (index - 1))
			{	// Поиск элемента, следующий за которым необходимо удалить.
				Current = Current->next;
				cur_index++;
			}
			tmp = Current->next; // Установить tmp на удаляемый элемент.
			if (tail == tmp)// Если удаляемый элемент - последний.
				tail = Current;
			Current->next = tmp->next; // Исключение из списка.
		}
		delete tmp;	// Освобождение памяти.
		count--;
		return 0;
	}
	void Print()	{
		LElem *Current = head;
		while (Current != NULL)		{
			printf("%d, %s\n", Current->D.data_n, Current->D.data_s);
			Current = Current->next;
		}
	}
};
int _tmain(int argc, _TCHAR* argv[])
{
	List1 L;
	L.InsetTail({ 20, "string 2", true });
	L.InsetHead({ 10, "string 1", true });
	if (L.Del(5) != 0) printf("ERROR\n"); else printf("Deleted\n");
	if (L.Del(1) != 0) printf("ERROR\n"); else printf("Deleted\n");
	L.InsetHead({ 0, "string 0", true });
L.Print();
```

Рис. 3.11. Класс «Список»

Приведенный пример показывает, что при реализации структур данных требуется учитывать множество частных случаев. Для многих это является камнем преткновения и поводом использовать готовые реализации, применение которых не всегда оправдано.

Односвязные списки позволяют решить множество практических задач, но встречаются ситуации, когда связей, направленных только в одну сторону, недостаточно. Например, если часто требуется обращаться к предыдущему элементу, то это потребует времени $O(n)$, а если бы элементы были связаны как в прямом, так и в обратном направлении, эта операция превратилась бы в обращение по прямому указателю и оценивалась бы как $O(1)$.

3.2.2. ДВУСВЯЗНЫЕ (ДВУНАПРАВЛЕННЫЕ) СПИСКИ

Элемент двусвязного списка имеет два указателя – на следующий элемент и на предыдущий, схема связи изображена на рис. 3.12.

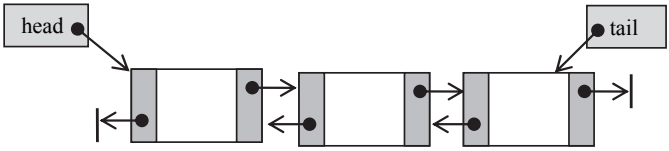


Рис. 3.12. Двусвязный список

За счет наличия дополнительных возможностей алгоритмы работы с таким списком более гибкие. Рассмотрим в качестве примера алгоритм, представленный на рис. 3.13, символом ∅ обозначен пустой указатель.

В отличие от предыдущего случая нет необходимости запоминать элемент, после которого будет производиться удаление, так как к нему ведет прямой указатель. Глядя на подробный алгоритм, представленный на рис. 3.14, можно заметить, что частных случаев, которые скрываются в соответствующем блоке рис. 3.13, три. Кроме явно записанных в условии равенств отдельно рассматривается ситуация, когда из списка удаляется единственный существующий элемент (при этом «голова» и «хвост» равны).

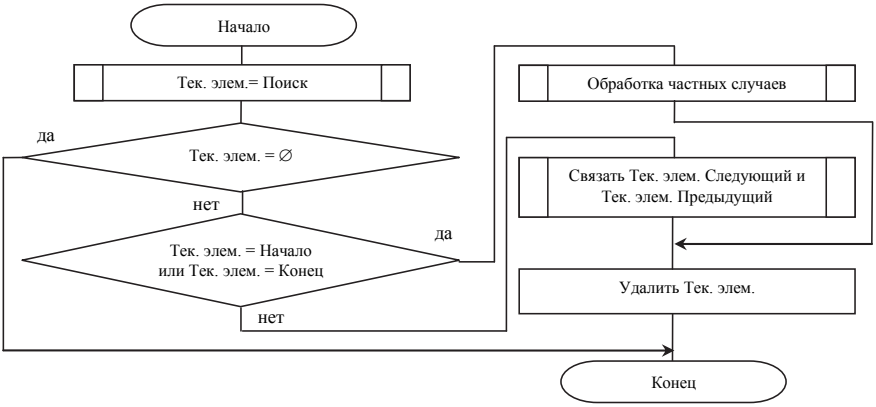


Рис. 3.13. Укрупненный алгоритм поиска и удаления элемента двусвязного списка

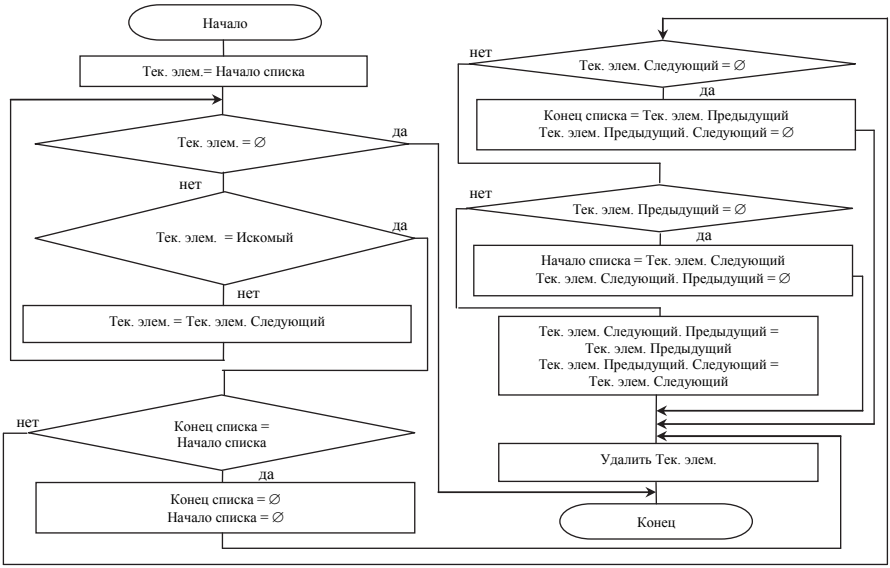


Рис. 3.14. Подробный алгоритм поиска и удаления элемента двусвязного списка

Запись алгоритма на языке программирования является исчерпывающей: не только дает наиболее детальное представление всех операций, но и не оставляет возможности что-то оставить за кадром. Излишние подробности могут затруднить восприятие основных идей, которые представляют главную ценность, в то время как использование блок-схем, псевдокода или иных форм описания алгоритмов позволяют выбрать достаточный уровень детализации, абстрагироваться от специфики синтаксиса конкретного языка и представить алгоритм в наиболее удобном для понимания виде.

3.2.3. ЦИКЛИЧЕСКИЕ (ЗАМКНУТЫЕ, КОЛЬЦЕВЫЕ) СПИСКИ

Иногда требуется выполнять циклический сдвиг элементов, для этого наилучшим образом подходят замкнутые списки. Отличительной особенностью этих структур является то, что последний элемент указывает на первый, а для двусвязного варианта добавляется связь в обратном направлении – первого с последним. Схема циклических списков изображена на рис. 3.15.

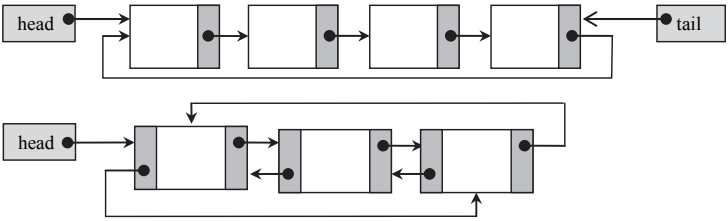


Рис. 3.15. Циклические списки

Чтобы осуществить циклический сдвиг элементов, достаточно переставить указатели начала и конца на нужное количество позиций. Для двусвязного кольцевого списка не имеет смысла контролировать указатель на последний элемент, так как его всегда можно получить за $O(1)$, используя первый.

3.2.4. МНОГОСВЯЗНЫЕ СПИСКОВЫЕ СТРУКТУРЫ

Если на одном и том же множестве элементов данных определено несколько списков, то такую структуру данных называют многосвязным или иерархическим списком. Забегая вперед, следует отметить, что это одна из форм реализации графов, достойная персонального внимания. Рассмотрим схему связей, представленную на рис. 3.16.

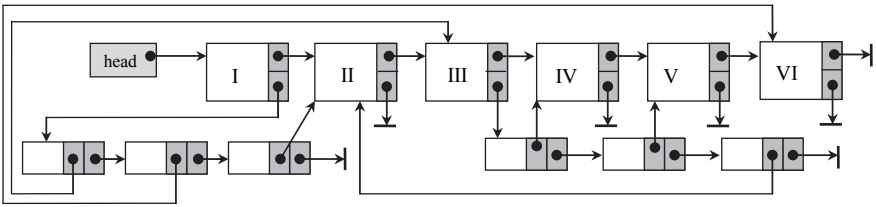


Рис. 3.16. Многосвязная списковая структура

Элементы данных, обозначенные римскими цифрами, образуют линейный односвязный список. У каждого из них предусмотрен указатель на начало нового списка, элементы которого указывают на такие же элементы первого списка. Название иерархический происходит от того, что, по сути, элементы первого списка также являются списками. Если выделить все связи между элементами первого списка и отбросить способ их реализации, то можно построить упрощенную модель. Схеме, приведенной на рис. 3.16, будет соответствовать рис. 3.17.

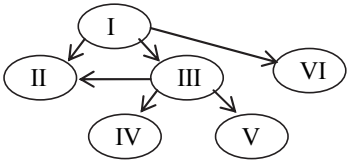


Рис. 3.17. Упрощенная модель
связей между элементами

Примером использования такой структуры может служить задача учета изделий, состоящих из деталей и сложных узлов, которые, в свою очередь, имеют состав. Элементы списков, отвечающих за состав, могут содержать число равное количеству вхождений указываемого узла в состав изделия. Может показаться, что, исходя из условий задачи, связь между элементами предполагает строгую иерархию, но это не так. В приведенном примере связь первого и третьего элементов со вторым нарушает иерархию и требует пояснения. Пусть второй элемент – это некоторый крепеж, например, шуруп, соединяющий детали IV и V в составе узла III, тогда можно предположить, что такой же крепеж применяется для соединения узла III и детали VI при конструировании изделия I.

3.3. ДЕРЕВЬЯ

Существует множество определений дерева. Того, кто плохо знаком с математикой, может шокировать утверждение, что дерево – это связный ациклический граф. Другое распространенное в сети Интернет определение звучит так: «Дерево – множество, состоящее из элемента, называемого корнем, и конечного (возможно пустого) множества деревьев, называемых поддеревьями данного дерева». Оно чем-то похоже на определение, сформулированное в [6], однако в оригинальном тексте есть важная оговорка, что поддеревья не пересекаются. Определение, которое дано в [2] звучит примерно так: дерево – это совокупность элементов (один из которых определен как корень), называемых узлами (или вершинами), и отношений, образующих иерархическую структуру узлов. Об отношениях между вершинами дерева удобно говорить как о родительских. Корень дерева можно определить как узел, не имеющий предка. Узел, не имеющий потомков, обычно называют листом. Высотой узла дерева называется длина самого длинного пути из этого узла до какого-либо листа.

Наиболее естественным вариантом реализации древовидной структуры выглядит наличие у элемента, представляющего узел, указателей на потомков, как показано на рис. 3.18.

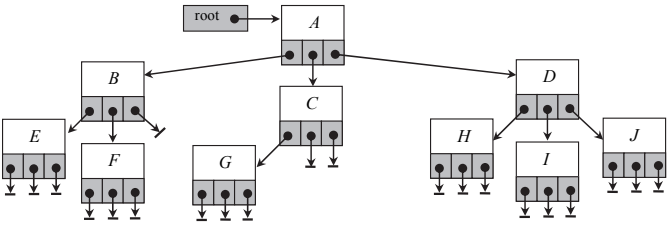


Рис. 3.18. Связи элементов дерева

Для упрощения реализации алгоритмов, в которых необходимо обращение к родительскому узлу, к элементу добавляют указатель на родителя. Подобная структура предполагает знание максимального количества потомков и не эффективна в ситуациях, когда это количество велико, а среднее число реально использующихся указателей мало. Для такого случая подойдет структура, представленная на рис. 3.19. Главными структурообразующими связями здесь являются указатели на младшего брата и старшего сына, таким образом, элементы, являющиеся потомками некоторого узла, объединены в список, доступ к ко-

торому возможен из узла-предка. Указатели на предка – вспомогательные, их добавляют только в случае необходимости.

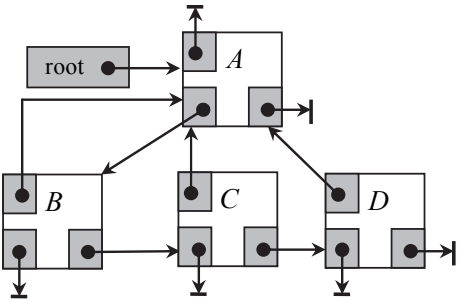


Рис. 3.19. Связи элементов дерева с произвольным количеством потомков

Дерево может быть организовано и с помощью многосвязных списков, которые рассмотрены выше. Такой способ в контексте дерева называют ***списком потомков***.

Тот факт, что дерево может быть представлено с помощью массива, может вызывать недоверие. Но это возможно благодаря тому, что любой элемент имеет не более одного предка. В примере, приведенном на рисунке 3.20, элемент массива содержит числовое поле, в котором записан индекс его родителя в дереве, корень дерева имеет в этом поле значение, которое недопустимо в качестве индекса.

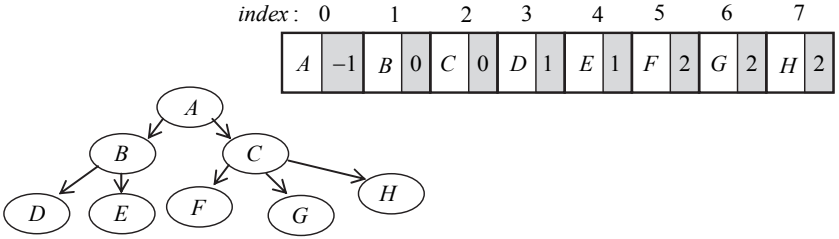


Рис. 3.20. Реализация дерева с помощью массива

Получить всех непосредственных потомков любого узла заданного такой структурой дерева удастся только за время $O(n)$, где n – размер массива или общее количество элементов в дереве. Переход в обрат-

ном направлении – от листа к корню – будет выполнен также за линейное время, но уже по отношению к высоте дерева.

Другие особенности работы с деревьями рассмотрены в разделах 4.4–4.6 и 5.1.

3.4. ГРАФЫ

Довольно трудно говорить о графе как о структуре данных. Граф слишком широкое понятие, все рассмотренные выше структуры являются частными случаями графа. Алгоритмы на графах – область достойная отдельного внимательного изучения. Ограничимся здесь рассмотрением основных способов реализации графов.

Учитывая, что граф – это совокупность множества вершин и связей между ними, его реализация должна обеспечивать возможность связывания любых пар элементов. Таким свойством обладает одна из рассмотренных ранее структур – иерархический список, в контексте работы с графами его называют ***списком смежности***. Пример реализации такого списка представлен на рис. 3.21, в нем в целях упрощения схемы используются курсоры.

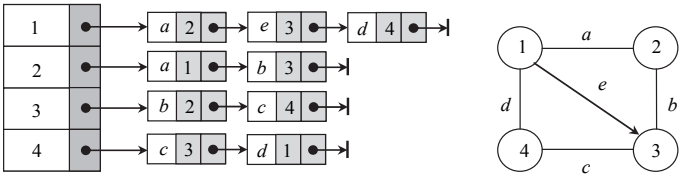


Рис. 3.21. Реализация графа с помощью списка смежности

Для неориентированных графов все элементы списков связей будут парными, поле данных этих элементов может содержать метку соответствующей дуги.

Представление графа списками позволяет без труда реализовать добавление и удаление вершин. При применении курсоров изменение графа будет более трудоемким по сравнению с реализацией, использующей указатели, но это позволит получать доступ к вершинам, обращаясь по номеру, за время $O(1)$.

Чтобы гарантировать константное время операции проверки существования некоторой дуги, можно использовать другой вариант представления графа – ***матрицу смежности***. Это квадратная матрица, элемент которой с индексами i и j равен единице при наличии дуги,

идущей из вершины i в вершину j, в противном случае он равен нулю. На месте единиц можно хранить метки соответствующих ребер, помечая отсутствующие ребра символом, которого нет в алфавите, условно обозначим его \$. Пример матрицы смежности и соответствующего ей графа приведен на рис. 3.22. Для неориентированных графов матрица будет симметричной.

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \text{или} \quad \begin{pmatrix} \$ & a & e & d \\ a & \$ & b & \$ \\ \$ & b & \$ & c \\ d & \$ & c & \$ \end{pmatrix}$$

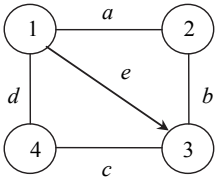


Рис. 3.22. Представление графа с помощью матрицы смежности

Изменение графа, заданного матрицей, будет связано с очевидными трудностями, а при использовании статических массивов станет практически невозможным. Еще один недостаток такой реализации состоит в том, что она требует $\Omega(n^2)$ памяти, где n – количество вершин; и даже если дуг значительно меньше, чем n^2, чтение такого графа будет выполняться за время $O(n^2)$.

Еще один вариант представления графа с помощью матрицы – ***матрица инцидентности*** – это матрица размера $n \times m$, где n – число вершин, а m – число ребер. Строки этой матрицы соответствуют вершинам, а столбцы – ребрам, элемент с индексами i и j может принимать значения: 1 – вершина инцидентна ребру и является его началом, 0 – вершина не инцидентна ребру, –1 – вершина инцидентна ребру и является его концом. Пример приведен на рис. 3.23.

	a	*b*	*c*	*d*	*e*
1	1	0	0	1	1
2	1	1	0	0	0
3	0	1	1	0	–1
4	0	0	1	1	0

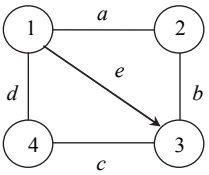


Рис. 3.23. Представление графа с помощью матрицы инцидентности

Недостатки по сравнению с матрицей смежности очевидны: время чтения (в худшем случае, когда каждая вершина связана со всем остальными) $O(n^3)$, время проверки существования ребра $O(n \times m)$.

Другой, малораспространенный вариант компьютерного представления графа – ***список ребер***. Элемент такого списка содержит номер или указатель начальной вершины и номер или указатель конечной вершины и может включать вес ребра. Пример изображен на рис. 3.24.

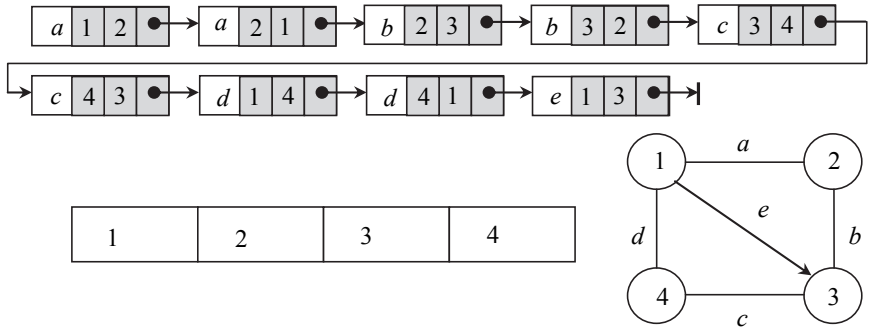


Рис. 3.24. Представление графа с помощью списка ребер, реализованного как односвязный список

Особых преимуществ такая динамическая реализация не даст, даже если будет полностью построена на указателях, и поэтому ее допустимо выполнить с помощью трех массивов, как показано на рис. 3.25.

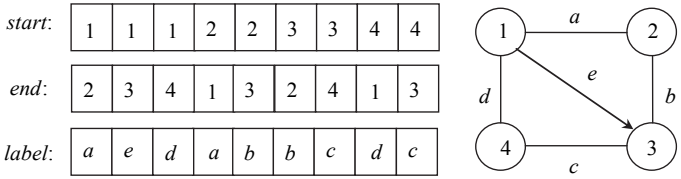


Рис. 3.25. Представление графа с помощью списка ребер, реализованного с помощью массивов

4. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ (ИНКАПСУЛЯЦИЯ НА СТАДИИ ПРОЕКТИРОВАНИЯ ПРОГРАММ)

Построение уровней абстракции распространяется практически на все области знаний человека. Очень часто мы рассуждаем о сложных механизмах, не вспоминая их отдельных деталей, а говоря о свойствах материалов, не вдаемся в подробности их молекулярной структуры. Процессы, происходящие при работе компьютера трудно описать, не выделяя различных уровней: уровень системы команд процессора, уровень операционной системы и, наконец, уровень прикладных программ. В программировании невозможно обойтись без абстрагирования: в конечном счете, программа будет преобразована в машинный код, который будет исполнен, но машинный код не может родиться в голове у нормального человека.

Базовые типы данных некоторого языка программирования – это абстракция, так как в действительности они представляют собой последовательность битов. Как было показано в предыдущем разделе, одни и те же структуры данных могут иметь различные реализации, обеспечивающие необходимые логические связи между элементами. Следовательно, вновь абстракция. Абстрактные типы – еще более высокий уровень, так как они могут быть построены на базе различных структур данных, позволяющих реализовать заданный набор операций.

Рассмотрим в качестве примера некоторую задачу, в которой присутствует необходимость работы с множествами. Если рассматривать понятие множества в широком математическом смысле, то кроме операций добавления, исключения и проверки принадлежности множеству заданного элемента потребуются еще и такие операции, как пересечение, объединение, разность множеств. Алгоритм решения этой задачи может быть выражен в терминах абстрактного типа данных «множество» (SET), с операциями INSERT(x, A), DELETE(x, A), FIND(x, A), INTERSECTION(A, B, C), UNION(A, B, C), DIFFERENCE(A, B, C), где A

и *B* – входные параметры типа множество, *C* – выходной параметр типа множество, *x* – имеет тип, совпадающий с типом элемента множества. Использование такого уровня абстракции поможет сконцентрироваться на особенностях конкретной задачи, временно «спрятав» нюансы реализации.

Набор операторов конкретного АТД обусловлен задачей, в которой он применяется, однако можно выделить несколько наиболее общих абстрактных типов и рассмотреть их основные операторы.

4.1. СТЕК

Стек представляет собой линейную последовательность элементов, добавление и удаление в которую осуществляется с одного конца, называемого вершиной. Говорят, что стек функционирует по принципу LIFO (Last In – First Out).

Основные операторы, свойственные стеку:

– PUSH(*x*, *S*) – помещает значение *x* в стек *S*;

– POP(*S*) – извлекает элемент из стека *S*;

– TOP(*S*) – возвращает значение элемента в вершине стека *S*;

– CLEAR(*S*) – удаляет все элементы из стека *S*;

– ISEMPTY(*S*) – возвращает значение «истина», если стек *S* пустой, «ложь» – в противном случае.

Схематично работа со стеком представлена на рис. 4.1.

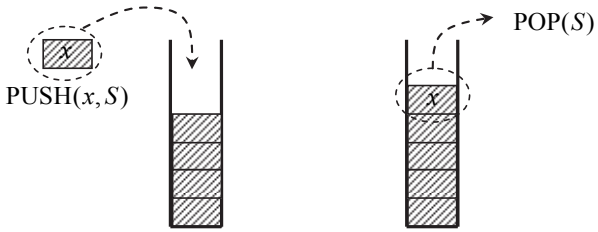


Рис. 4.1. Работа со стеком

При условии, что максимальное количество элементов известно, стек может быть эффективно реализован с помощью массива, как показано на рис. 4.2. Зафиксировав «дно» стека как начало массива, можно использовать курсор или указатель *top*, определяющий положение вершины.

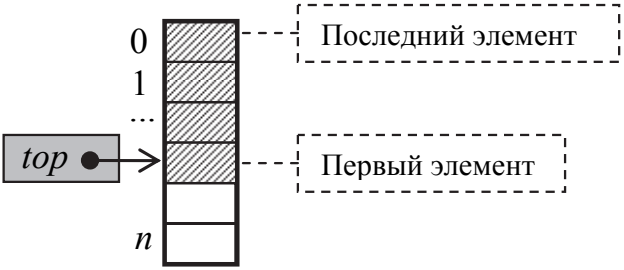


Рис. 4.2. Реализация стека с помощью массива

Использование списков выглядит более естественным, но при этом следует учитывать временные затраты на создание новых элементов (выделение памяти).

4.2. ОЧЕРЕДЬ

Работа с очередью в отличие от стека осуществляется по принципу FIFO (First In – First Out). Элементы добавляются с одного конца – заднего (rear), а удаляются с другого – переднего (front). Внешне операторы, выполняемые над очередями, похожи на операторы стеков, но имеют другие устоявшиеся названия:

– ENQUEUE(x, Q) – помещает значение x в очередь Q;
– DEQUEUE(Q) – извлекает элемент из очереди Q;
– FRONT(Q) – возвращает значение первого элемента очереди Q;
– CLEAR(Q) – удаляет все элементы из очереди Q;
– ISEMPTY(Q) – возвращает значение «истина», если очередь Q пуста, «ложь» – в противном случае.

Схематично работа с очередью представлена на рис. 4.3.

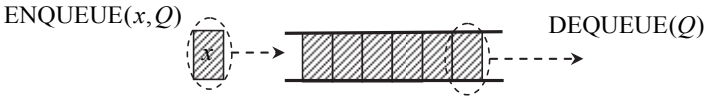


Рис. 4.3. Работа с очередью

Списки являются наиболее удобной для очереди структурой данных, так как время добавления и удаления элементов с любого конца есть $O(1)$. Реализация очередей с помощью массивов, позволяющая

избежать сдвига всех элементов, находящихся в очереди, не столь очевидна как у стека. Для этого организуем работу с массивом как с циклической структурой, где первая ячейка следует за последней. Это возможно, если определить процедуру увеличения индекса на единицу следующим образом: $i = i \bmod n + 1$, где i – текущий индекс, n – размер массива. При добавлении элемента будем помещать его в позицию следующую за индексом *rear* и изменим значение *rear* (рис. 4.4, *а*), если достигнем конца массива, то согласно введенному правилу увеличения индекса перейдем к началу, как показано на рис. 4.4, *б*. При удалении элемента сдвинем указатель *front* в сторону увеличения индексов.

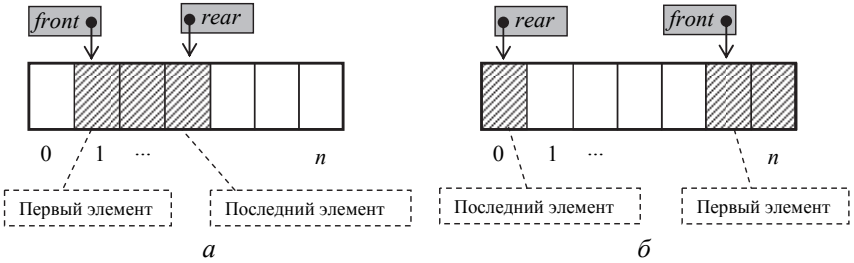


Рис. 4.4. Реализация очереди с помощью «циклического массива»

4.3. ДЕК

Двусторонняя очередь, которую также называют «дек», позволяет добавлять и удалять элементы с обоих концов, как схематично представлено на рис. 4.5.

Рис. 4.5. Двусторонняя очередь

Для дека характерны следующие операции:

– PushBack(x, D), PopBack(D), PushFront(x, D), PopFront (D) – помещают значение x и извлекают элемент в конце и начале дека D соответственно;

– CLEAR(D), ISEMPTY(D) – удаление всех элементов и проверка наличия элементов в деке D.

4.4. ОТОБРАЖЕНИЯ

АТД Mapping полезен в ситуациях, когда элементам некоторого множества требуется поставить в соответствие значения другого множества, и это невозможно сделать иначе, чем хранением каждой пары соответствий. Рассмотрим операторы отображений:

– CLEAR(M) – удаляет все соответствия;

– ASSIGN(*M*, *d*, *r*) – делает *M*(*d*) равным *r*;

– COMPUTE(*M*, *d*, *r*) – возвращает значение «истина» и присваивает *r* значение *M*(*d*), если оно определено, возвращает «ложь» – в противном случае.

Когда тип элементов области определения отображений позволяет использовать их как индексы массива, т. е. является конечным интервалом значений простого типа, процедуру COMPUTE можно выполнить за константное время. Рассмотрим конкретный пример: пусть множеству символов {*a*, *b*, *c*, *i*, *j*} нужно ставить в соответствие неотрицательные целые числа. Для получения индекса заданного символа достаточно отнять от него символ с наименьшим кодом. Как неопределенное соответствие можно рассматривать любое отрицательное число. Пример такой реализации приведен на рис. 4.6.

4.5. ДЕРЕВО КАК АТД

4.5.1. ОСНОВНЫЕ ОПЕРАЦИИ

Для работы с деревьями можно определить следующий типичный набор операций:

– PARENT(*n*, *T*) – возвращает родителя узла *n* в дереве *T*;

– LEFTMOST__CHILD(*n*, *T*) – возвращает самого левого сына узла *n* в дереве *T*;

– RIGHT_SIBLING(*n*, *T*) – возвращает правого брата узла *n* в дереве *T*;

– ROOT(*T*) возвращает узел, являющимся корнем дерева *T*.

Все перечисленные функции могут возвращать специальное значение – признак отсутствия нужного узла.

Для создания дерева необходимо некоторым образом определить процедуру CREATE(…), а для получения метки узла *n* в дереве *T* – процедуру LABEL(*n*, *T*).

```
class Mapping
{
	int M[10];
public:
	Mapping() { Clear(); }
	void Assign(char d, int Md){ M[d - 'a'] = Md; }
	bool Compute(char d, int &r)
	{
		if (M[d - 'a'] >= 0)
		{
			r = M[d - 'a'];
			return true;
		}
		else
			return false;
	}
	void Clear() { for (int i = 0; i < 10; i++) M[i] = -1; }
}M;
int _tmain(int argc, _TCHAR* argv[])
{
	M.Assign('a', 15);
	M.Assign('c', 115);
	M.Assign('i', 25);
	int Md;
	char c = 'a';
	for (int i = 0; i < 10; i++,c++)
	{
		if (M.Compute(c, Md))
			printf("%c <=> %d\n", c, Md);
		else
			printf("%c <=> undefined\n", c);
	}
	_tsystem(L"pause");
	return 0;
}
```

Рис. 4.6. Реализация отображения с помощью массива

4.5.2. ОБХОДЫ ДЕРЕВЬЕВ

Деревья используются при решении широкого спектра задач. Во многих из них порядок размещения и обхода элементов играет ключевую роль. Различные способы обхода дерева можно трактовать как различные правила упорядочивания его узлов. Рассмотрим три способа обхода дерева ***в глубину*** на примере ***деревьев выражений***, метки внутренних узлов таких деревьев соответствуют операциям, метки листьев – операндам. Пример дерева выражений приведен на рис. 4.7.

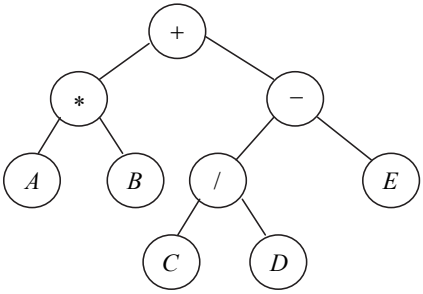


Рис. 4.7. Дерево выражений

Префиксный* (*прямой*) *обход состоит в том, что сначала обрабатывается текущий узел, затем левое и правое поддеревья. Для дерева, изображенного на рис. 4.7, будет получена следующая последовательность узлов: + * A B – / C D E .

Постфиксный* (*обратный*) *обход – сначала обрабатываются левое и правое поддеревья текущего узла, затем сам узел. Последовательность узлов для приведенного примера: A B * C D / E – + .

Инфиксный* (*симметричный*) *обход – сначала обрабатывается левое поддерево текущего узла, затем сам узел, затем правое поддерево. Последовательность узлов для приведенного примера: A * B + C / D – E .

Другой способ обхода, называемый ***обходом в ширину***, выводит узлы по уровням – вначале корень, затем все его непосредственные потомки, затем все непосредственные потомки сыновей корня, начиная с самого левого, последними будут выведены листья. В алгоритме обхода в ширину используется очередь. Первоначально в очередь помещается корень, затем, пока очередь не пуста, выполняются следующие действия:

1) из очереди выталкивается очередной узел;

2) этот узел обрабатывается;

3) в очередь добавляются сыновья обработанного узла;

4) переход к шагу 1.

Пример реализации представленного алгоритма для двоичного дерева приведен на рис. 4.8. ***Двоичным***, или ***бинарным***, называют дерево, каждый узел которого имеет не более двух потомков. Для тестирования алгоритма в программе создается дерево, изображенное на рис. 4.9. В результате работы программы метки узлов будут выведены в алфавитном порядке.

```
#include <queue>
using namespace std;
struct node     // структура узла дерева
{
        char label;
        node *left;
        node *right;
        node(char c) { left = right = NULL; label = c; }
};
class Queue : public queue<node*>       // реализация очереди указателей
{                                       // на узлы на основе STL queue
public:
        void enqueue(node *x)   {       push(x); }
        node * dequeue(){ node *tmp;  tmp = front(); pop(); return tmp; }
}Q;
void width(node *root)  // функция вывода меток узлов при обходе в ширину
{
        node* x;
        Q.enqueue(root);        // поместить корень в очередь
        while (!Q.empty())
        {
                x = Q.dequeue();        // извлечь из очереди и
                printf("%c ", x->label);// обработать узел x
                if (x->left != NULL)    // поместить в очередь левое поддерево x
                        Q.enqueue(x->left);
                if (x->right != NULL)   // поместить в очередь правое поддерево x
                        Q.enqueue(x->right);
        }
}
int _tmain(int argc, _TCHAR* argv[])
{
        node r('A'), nB('B'), nC('C'), nD('D'),nE('E'),nF('F'),nG('G');
        r.left = &nB;   r.right = &nC;  // создадим дерево с корнем r
        nB.left = &nD;  nB.right = &nE;
        nC.left = &nF;  nC.right = &nG;
        width(&r);
        system("pause");
        return 0;
}
```

Рис. 4.8. Реализация обхода двоичного дерева в ширину

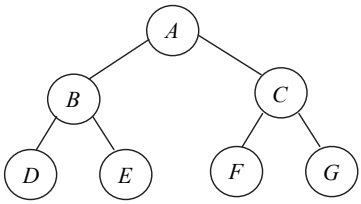


Рис. 4.9. Двоичное дерево

4.6. ПИРАМИДА (КУЧА)

Сбалансированное двоичное дерево, в котором значения дочерних узлов всегда меньше (или всегда больше) родительского, называют ***пирамидой***, ***двоичной кучей*** или ***частично упорядоченным деревом***. «Частично» означает, что порядок, в котором расположены потомки, не важен, важно только взаимное расположение потомков и родителя. Сбалансированность означает, что высоты левого и правого поддеревьев любого узла отличаются не более, чем на единицу. По способу упорядочивания различают max-heap, где родительский узел больше, и min-heap, где меньше.

Для таких деревьев характерны операции DeleteMin и DeleteMax, которые извлекают соответствующий минимальный или максимальный элемент за время $O(\log n)$, где n – высота дерева. Пример пирамиды изображен на рис. 4.10.

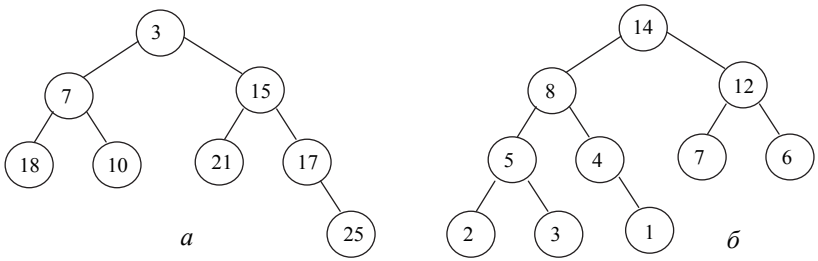


Рис. 4.10. Примеры пирамид:

а – min-heap; *б* – max-heap

Основное применение пирамид – сортировка, с этим связано еще одно их название – ***сортирующее дерево***.

4.7. ПРЕФИКСНОЕ ДЕРЕВО (БОР)

Для представления множеств, элементами которых являются символьные строки, используются специальные структуры, называемые префиксными деревьями, у них существуют и другие названия: бор или нагруженное дерево [2]. Применение таких структур не ограничивается только строками символов, это могут быть последовательности значений других типов, например, целых чисел.

В нагруженных деревьях элементу множества соответствует не метка узла, а путь, пройденный от корня к листу, либо (в других реализациях) путь, пройденный от корня к узлу со специальной меткой. При изображении таких деревьев принято писать символы не в узлах, а на ребрах. На рис. 4.11 представлены примеры дерева, содержащего строки he, she, his, him, her, hers. Дерево (рис. 4.11, *а*) построено с использованием дополнительного символа $, которого нет в алфавите исходного множества строк, добавление его в конце каждого слова обеспечивает завершение слова в листе. Дерево (рис. 4.11, *б*) иллюстрирует альтернативный подход – заштрихованные узлы соответствуют концам слов.

Узлы префиксных деревьев можно рассматривать как отображение множества символов в указатели на узел. Если реализация отображения позволяет получить соответствие за константное, не зависящее от размера алфавита время, то поиск образца занимает время, линейное относительно его длины, и не зависит от размера множества.

Основными операторами АТД «префиксное дерево» будут операторы множеств INSERT(x, T), DELETE(x, T), FIND(x, T), где T – префиксное дерево, x – строка. FIND(x, T) будет выполняться за время $O(|x|)$ и не будет зависеть от размеров T.

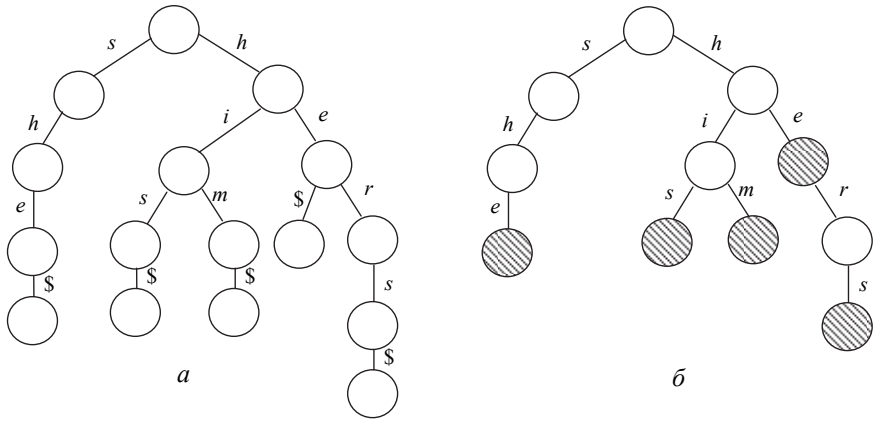


Рис. 4.11. Префиксное дерево для строк he, she, his, him, her, hers

5. БЫСТРЫЙ ПОИСК (ГАРАНТИРОВАННЫЙ ЛОГАРИФМ И МИФИЧЕСКАЯ КОНСТАНТА)

В этом разделе рассмотрены специальные подходы к реализации больших множеств, позволяющие ускорить процедуру поиска элемента.

5.1. ДЕРЕВЬЯ ПОИСКА

Для представления множеств, чьи элементы могут быть упорядочены посредством некоторого отношения линейного порядка, возможно использовать древовидные структуры. Большинство таких структур обеспечивают порядок времени выполнения поиска $O(\log n)$, где n – количество элементов.

5.1.1. ДВОИЧНЫЕ ДЕРЕВЬЯ ПОИСКА (BINARY SEARCH TREE, BST)

Двоичное дерево поиска – это двоичное дерево, узлы которого помечены или содержат элементы множества, все элементы левого поддерева некоторого узла меньше элемента этого узла, а правого – больше [2]. Такой подход к организации дерева поиска представляется наиболее очевидным. Для того, чтобы найти заданное значение в дереве, нужно, начиная от корня, сравнивать искомое значение с элементом текущего узла, и если равенство не выполнено, то в зависимости от результата сравнения переходить к левому или правому потомку. В случае, если дерево сбалансировано, как показано на рис. 5.1, *а*, то после каждого сравнения число оставшихся претендентов на равенство уменьшается в два раза, обеспечивая время поиска в худшем случае (элемента нет или он является листом) $O(\log_2 n)$.

Если операция вставки не предусматривает никаких изменений расположения элементов, а только находит позицию в соответствии с

текущим состоянием и добавляет новый лист, то баланс дерева будет зависеть от порядка добавления элементов. При этом дерево может выглядеть как на рис. 5.1, *б* или вовсе выродится в список с соответствующим линейным временем поиска.

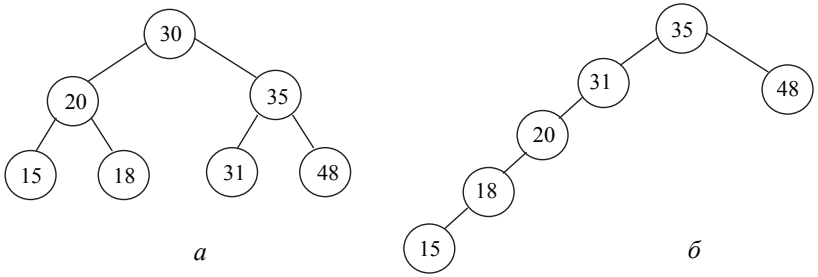


Рис. 5.1. Двоичные деревья поиска:

а – сбалансированное; *б* – нарушение баланса

Для обеспечения баланса операции вставки и удаления должны перестраивать дерево – изменять взаимное расположение узлов. Далее рассмотрены некоторые известные подходы.

5.1.2. АВЛ-ДЕРЕВЬЯ

Это один из первых (а возможно, и самый первый) подход к балансировке, он был предложен в 1962 году советскими математиками Г.М. Адельсоном-Вельским и Е.М. Ландисом [7] и назван в их честь.

Ограничение, накладываемое на структуру, формулируется крайне просто: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более, чем на единицу. Этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов и лежала в пределах от $\log_2(n+1)$ до $1,44\log_2(n+2)$.

Отдельного рассмотрения заслуживают операции, обеспечивающие выполнение этого свойства, – повороты. Трансформации, которые они выполняют, показаны на рис. 5.2.

Рассмотрим ситуацию нарушения баланса, когда высота правого поддерева узла *p* на 2 больше высоты левого поддерева (обратный случай является симметричным).

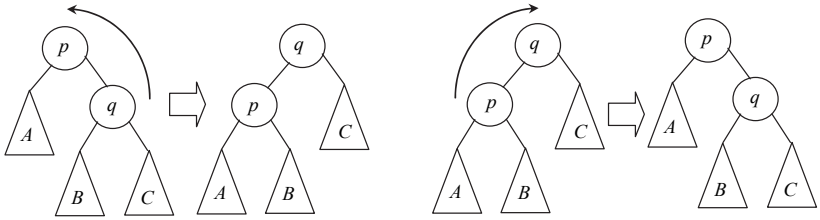


Рис. 5.2. Повороты дерева:

а – левый; *б* – правый

Если *q* – правый дочерний узел *p*, то при условии, что высота правого поддерева узла *q* больше высоты его левого поддерева, выполняется простой левый поворот, как показано на рис. 5.3.

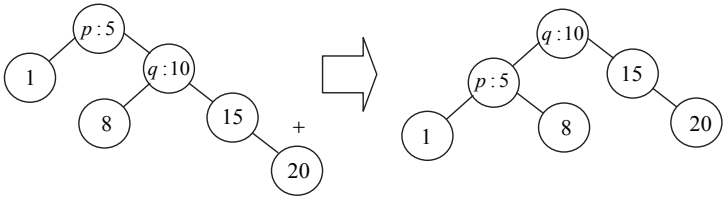


Рис. 5.3. Балансировка АВЛ-дерева, простой поворот

В случае, когда высота левого поддерева узла *q* больше высоты его правого поддерева, выполняют большой поворот: вначале правый поворот вокруг *q* и затем левый вокруг *p*, как показано на рис. 5.4.

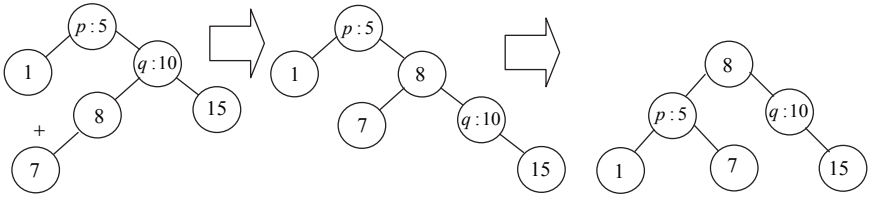


Рис. 5.4. Балансировка АВЛ-дерева, большой поворот

Вставка нового элемента в АВЛ-дерево выполняется так: спускаемся вниз по дереву, выбирая правое или левое направление движения

в зависимости от результата сравнения, добавляем лист, а при возвращении из рекурсии выполняется балансировка текущего узла.

При удалении спускаемся вниз по дереву в поисках нужного элемента, если удаляемый элемент не является листом, то заменяем его минимальным элементом его правого поддерева (который либо является листом, либо имеет единственного потомка справа), как показано на рис. 5.5. На выходе из рекурсии как в самом удалении, так и в перемещении минимального элемента правого поддерева выполняем балансировку.

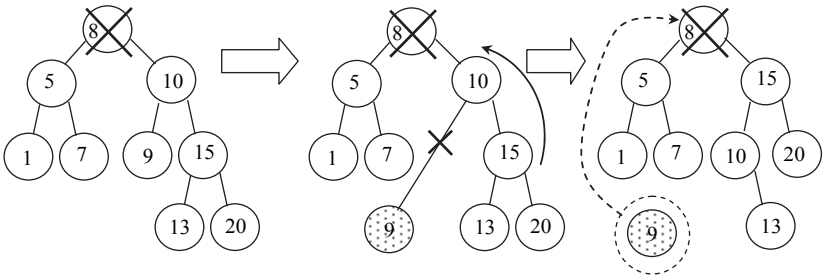


Рис. 5.5. Удаление из АВЛ-дерева

Таким образом, все основные операции по работе с АВЛ-деревом требуют времени порядка $O(\log(n))$.

5.1.3. КРАСНО-ЧЕРНЫЕ ДЕРЕВЬЯ

Это другой весьма популярный подход к балансировке двоичных деревьев поиска, опубликованный Робертом Седжвиком в 1978 году [8]. В основе лежит пометка узлов одним из двух вариантов цвета (красным или черным) и два основных ограничения. Первое состоит в том, что два красных узла не могут появляться непосредственно друг за другом, а второе – все пути от корня к листьям содержат одинаковое число черных узлов. Кроме того, принято считать, что листья не содержат данных и являются черными. Высота этих деревьев

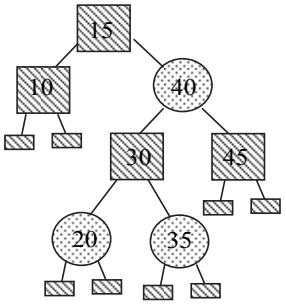


Рис. 5.6. Пример красно-черного дерева, черные узлы обозначены прямоугольниками, красные – окружностями

не превосходит $2\log(n+1)$, где n – количество элементов, пример дерева изображен на рис. 5.6. По высоте, и, следовательно, по балансировке красно-черное дерево немного уступает АВЛ-дереву, преимущества заключаются в меньшем количестве операций необходимых для поддержания баланса. Асимптотика операций остается прежней – $O(\log(n))$. При вставке новый узел обычно помечается красным, что позволяет сохранить «черную» длину путей. Затем, исходя из первого ограничения, выполняются повороты или изменения цветов. Частные случаи, возникающие при этом, рассмотрены в литературе [9], [13].

5.1.4. РАНДОМИЗИРОВАННЫЕ ДЕРЕВЬЯ

Ключевая идея данного способа балансировки, предложенная в 1998 году [10], состоит в следующем. Если заранее перемешать все ключи (если элемент имеет сложную структуру, то следует говорить о ключе элемента – поле данных, по которому элементы упорядочиваются) и потом построить из них дерево обычной процедурой добавления, то построенное дерево окажется неплохо сбалансированным. При этом корнем с одинаковой вероятностью может оказаться любой из исходных ключей. Таким образом, если любой ключ может оказаться корнем с вероятностью $\frac{1}{n+1}$ (n – размер дерева до вставки), то можно выполнять с указанной вероятностью вставку в корень, а с вероятностью $1-\frac{1}{n+1}$ – рекурсивную вставку в правое или левое поддерево в зависимости от значения ключа. Вставка в корень проиллюстрирована на рис. 5.7, а на рис. 5.8 приведен соответствующий алгоритм, использующий операции левого и правого поворотов вокруг заданного узла p – RotateLeft(p) и RotateRight(p), которые возвращают в качестве результата новый корень поддерева.

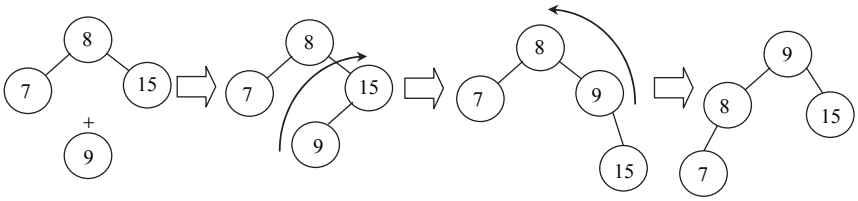


Рис. 5.7. Вставка в корень

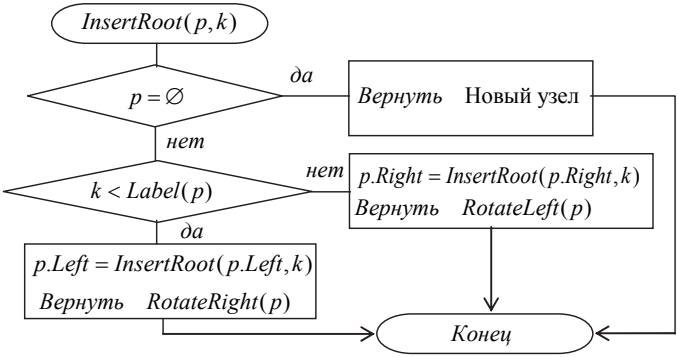


Рис. 5.8. Алгоритм вставки в корень

После удаления элемента возникает задача объединения деревьев. Пусть даны два дерева поиска с корнями p и q, причем любой ключ первого дерева меньше любого ключа во втором дереве. Требуется объединить эти два дерева в одно. В качестве корня нового дерева можно взять любой из двух корней. В рандомизированной реализации выбор между этими альтернативами делается случайным образом. Пусть размер левого дерева равен n, правого – m. Тогда p выбирается новым корнем с вероятностью $\frac{n}{n+m}$, а q – с вероятностью $\frac{m}{n+m}$. Объединение деревьев, схематично показанное на рис. 5.9 (корнем объединения выбран корень левого поддерева), также является рекурсивной функцией.

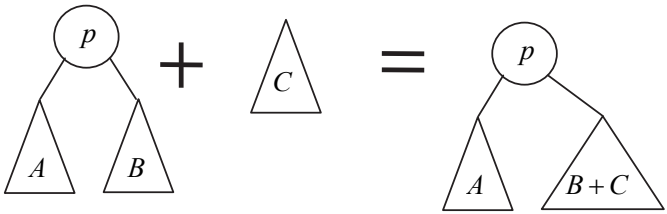


Рис. 5.9. Объединение деревьев

Теоретическая оценка высоты дерева $4,3\log(n)-1,9\log(\log(n))-4$.

5.1.5. ДЕКАРТОВЫ ДЕРЕВЬЯ

Декартово дерево [11] хранит пары (x, y) в виде бинарного дерева таким образом, что оно является бинарным деревом поиска по x и бинарной пирамидой по y (y – называют приоритетом). Тот факт, что эти пары значений можно рассматривать как координаты на плоскости, объясняет наиболее литературное название таких деревьев. Другое название – дерамида (дерево + пирамида) в английском варианте treap (tree + heap).

Основная идея заключается в следующем: из множества ключей можно построить много различных корректных деревьев поиска, однако после добавления к ним приоритетов дерево из данных ключей можно построить единственное, вне зависимости от порядка поступления ключей. Если приоритеты случайны, то полученное декартово дерево с высокой вероятностью будет иметь высоту, не превосходящую $4\log(n)$.

Добавление и удаление элементов реализуется с помощью двух вспомогательных операций: Merge и Split. Операция Merge принимает на вход два декартовых дерева L и R, ключи дерева L не превышают ключей R. От нее требуется соединить их в одно корректное декартово дерево T. Корнем будущего дерева станет корень с наибольшим приоритетом. Пусть это будет левое дерево, тогда оно останется слева, и вновь появятся два дерева, причем ключи в одном меньше ключей в другом. Фактически эта процедура аналогична объединению рандомизированных деревьев, рассмотренных выше.

На вход операции Split поступает корректное декартово дерево T и некий ключ x_0. Задача операции – разделить дерево на два так, чтобы в одном из них (L) оказались все элементы исходного дерева с ключами, меньшими x_0, а в другом (R) – с большими. Если ключ корня исходного дерева меньше x_0, то он должен оказаться в L, иначе – в R. Предположим, что ключ корня оказался меньше x_0. Тогда все элементы левого поддерева T также окажутся в L. Корень T станет корнем L, поскольку его приоритет наибольший во всем дереве.

Левое поддерево корня полностью сохранится без изменений, а правое уменьшится – из него нужно убрать элементы с ключами больше x_0 и вынести в дерево R, т. е. вновь разделить по ключу x_0. Работа операции Split изображена на рис. 5.10.

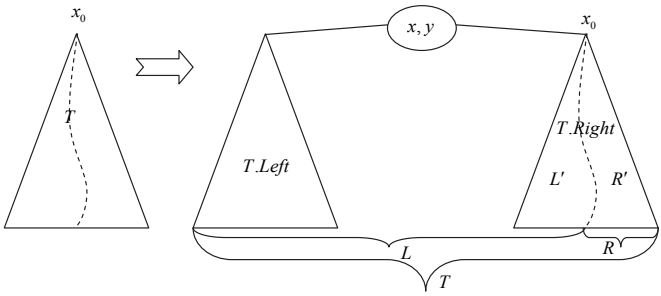


Рис. 5.10. Разделение дерева T по ключу x_0

Для вставки нового элемента с ключом x в дерево T требуется выполнить три операции, как показано рис. 5.11.

1. Разделим T по ключу x на L с ключами меньше x, и на R – с ключами больше x.

2. Создадим дерево M из единственной вершины (x, y), где y – случайный приоритет.

3. Объединим по очереди L с M, то, что получилось, – с R.

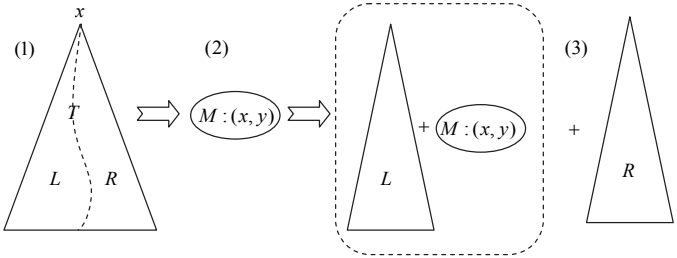


Рис. 5.11. Добавление в декартово дерево T ключа x

Удаление элемента с ключом x из дерева T также можно выполнить с помощью трех операций, как показано рис. 5.12.

1. Разделим дерево по ключу x-1. Все элементы, меньшие либо равные x-1, будут в дереве L, значит, искомый элемент – в R.

2. Разделим R по ключу x. В новый правый результат R' будут помещены все элементы с ключами больше x, а в «средний» – все ключи меньше либо равные x. Поскольку строго меньшие после первого шага были отсеяны, то среднее дерево содержит только искомый элемент.

3. Объединим снова L с R'.

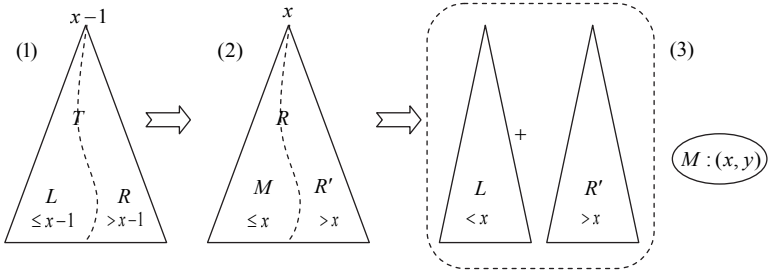


Рис. 5.12. Удаление из декартова дерева T ключа x

5.1.6. ДРУГИЕ ВИДЫ СБАЛАНСИРОВАННЫХ ДЕРЕВЬЕВ

Сбалансированное двоичное дерево поиска наилучшим образом олицетворяет концепцию «разделяй и властвуй», но, как было показано, поддержание баланса – непростая задача. Существуют альтернативные подходы к построению сбалансированных деревьев, позволяющие узлам иметь более двух потомков. Ниже рассмотрены два вида таких деревьев.

5.1.7. 2-3 ДЕРЕВЬЯ

Основные свойства 2-3 дерева состоят в следующем.

1. Каждый внутренний узел имеет два или три непосредственных потомка (их называют соответственно 2- или 3-узлами).

2. Пути от корня до любого листа имеют одинаковую длину.

3. Элементы множества располагаются только в листьях дерева в возрастающем порядке.

4. В каждый внутренний узел записывается ключ наименьшего элемента второго потомка и ключ наименьшего элемента третьего потомка, если он есть.

Очевидно, что при выполнении этих свойств высота дерева, содержащего n элементов, будет находиться в интервале от $1+\log_3(n)$ до $1+\log_2(n)$. 2-3 деревья предложены Д.Е. Хопкрофтом в 1973 году, подробное описание можно найти в [2].

При поиске ключ искомого элемента сравнивается с первым ключем текущего внутреннего узла, и если искомый ключ меньше, то осуществляется переход в левое поддерево. Иначе он сравнивается со вторым ключом (если второго ключа нет, то поддерева всего два, и сразу можно перейти во второе), и если искомый не превосходит второй

ключ, то осуществляется переход в среднее поддерево, а если превосходит, то в правое.

В процессе вставки элемента можно выделить два случая. В первом новый лист добавляется к 2-узлу, превращая его в 3-узел, и достаточно будет изменить ключи родителя в соответствии с новым порядком сыновей. Во втором случае, когда родитель нового листа уже является 3-узлом, он расщепляется на два 2-узла. Процесс такого восходящего расщепления может достичь корня, при этом будет создан новый корень. Пример вставки показан на рис. 5.13.

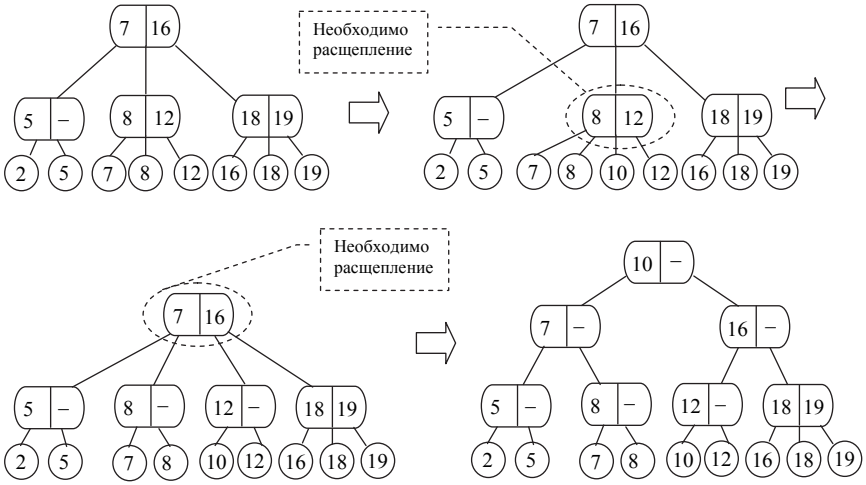


Рис. 5.13. Добавление элемента «10»

При удалении листа может оказаться, что у его родителя остался единственный сын, если левый или правый брат родителя удаленного листа имеет трех сыновей, то одного из них можно переместить. В противном случае родитель удаленного листа удаляется, а его оставшийся сын переходит к его брату. При таком восходящем слиянии может быть удален корень. Пример удаления показан на рис. 5.14.

5.1.8. 2-3-4 ДЕРЕВЬЯ

2-3-4 деревья являются частным случаем обобщенного подхода к построению сбалансированных деревьев поиска – B-деревьев, предложенных Р. Бэйером и Е. МакКрейтом в 1970 году [12]. Подробное опи-

сание алгоритмов работы с 2-3-4 деревьями можно найти в [13]. Ниже перечислены основные свойства 2-3-4 дерева.

1. Элементы множества располагаются во внутренних узлах и листьях дерева (не более трех на узел).

2. Внутренний узел имеет два, три или четыре непосредственных потомка (их называют соответственно 2-, 3- или 4-узлами).

3. Ключи элементов, хранящихся во внутренних узлах, служат разделяющими границами при поиске в дереве.

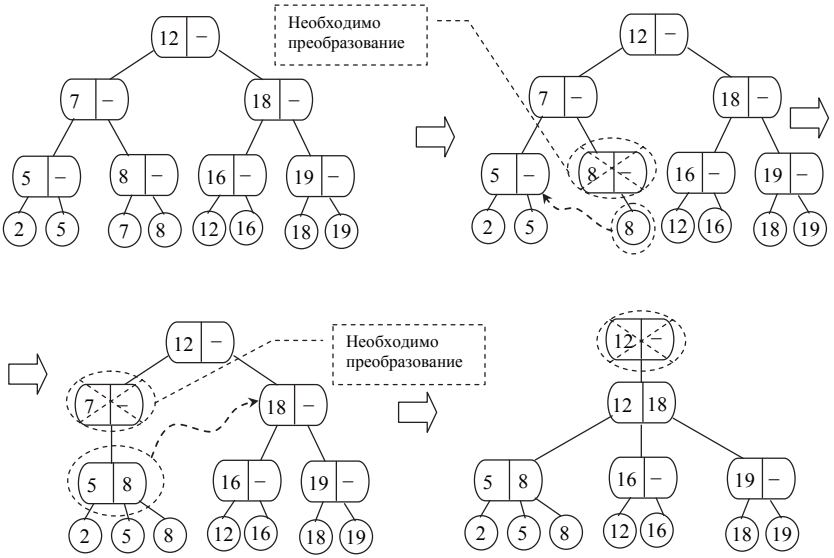


Рис. 5.14. Удаление элемента «7»

Поиск в таком дереве аналогичен поиску в 2-3 дереве с той лишь разницей, что нет необходимости спускаться до листа, если элемент найден.

Алгоритм добавления узла можно построить таким образом, чтобы избежать восходящего расщепления, характерного для 2-3 деревьев. Для этого нужно исключить ситуации, в которых родитель 4-узла также является 4-узлом. Этого можно добиться, если разбивать 4-узлы на прямом пути – пути поиска места вставки. Пример добавления изображен на рис. 5.15.

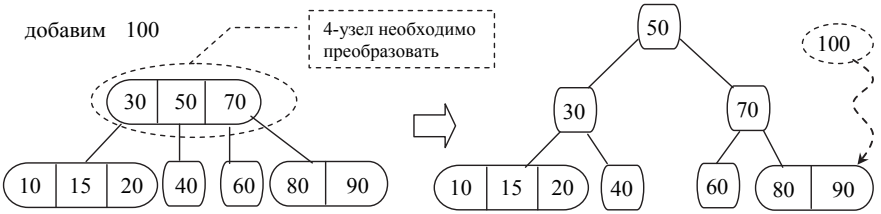


Рис. 5.15. Вставка в 2-3-4 дерево

В примере, показанном на рис. 5.15, был преобразован корень дерева, если бы это был внутренний узел, то вытесненный при расщеплении элемент следовало бы передать родителю, учитывая его тип (2- или 3-узел).

Удаление всегда производится с листа. Если удаляемый элемент оказался в середине дерева, то ищется преемник, которым его можно заменить. Удаляя элемент «20» из дерева на рис. 5.16, на его место будет перемещен элемент «17».

Чтобы избавить алгоритм удаления от необходимости возвращаться, объединяя узлы, нужно предусмотреть преобразование 2-узлов в 3- или 4-узлы по пути поиска удаляемого элемента. При этом возможно либо заимствование из элементов братьев, являющихся 3- или 4-узлами (с изменением элемента родителя), либо объединение двух братьев (2-узлов) в 4-узел с заимствованием элемента родителя (ситуация обратная расщеплению при вставке). Примеры удаления изображены на рис. 5.17, 5.18.

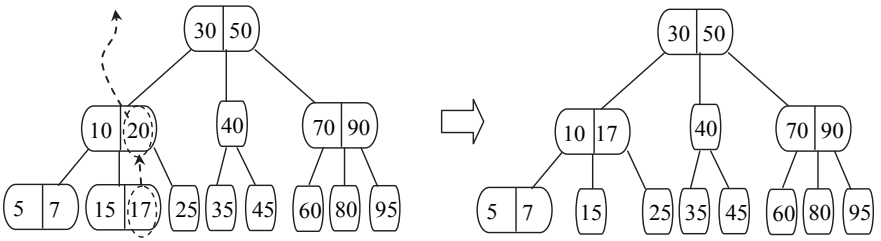


Рис. 5.16. Удаление элемента «20» (простой случай)

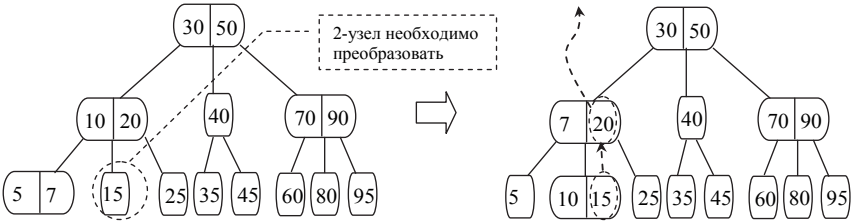


Рис. 5.17. Удаление элемента «20» (другая ситуация)

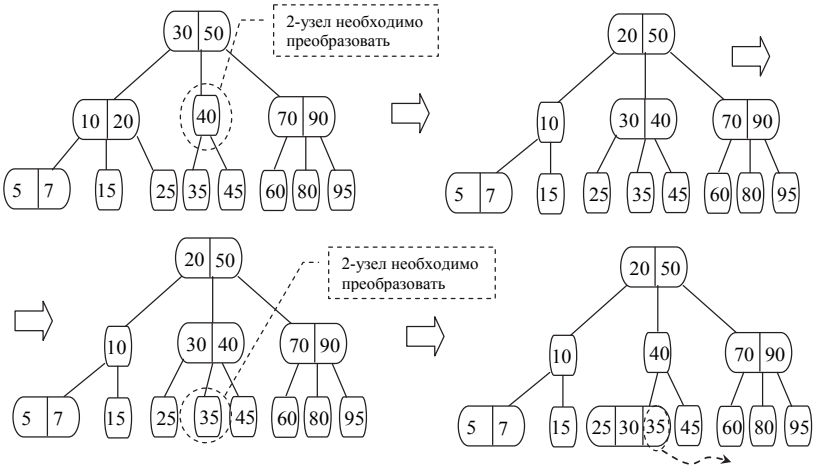


Рис. 5.18. Удаление элемента «35» (два преобразования по пути поиска)

5.2. ХЕШ-ТАБЛИЦЫ

Рассмотренные выше сбалансированные деревья гарантируют логарифмическую временную сложность выполнения операций добавления, удаления и поиска элементов. Лучше логарифма может быть разве что константа. Такими характеристиками обладают хеш-таблицы. В ***хеш-таблице*** элементу отводится индекс $h(k)$, где k – ключ элемента (в случае если элемент имеет сложную структуру, это может быть одно или несколько из его свойств), h – хеш-функция (функция, принимающая значения 0,…*size*-1, где *size* – размер хеш-таблицы).

Различают два способа размещения данных в хеш-таблице – ***открытое*** (внешнее или расширенное) и ***закрытое*** (внутреннее или прямое) ***хеширование***.

5.2.1. ЗАКРЫТОЕ ХЕШИРОВАНИЕ

Закрытое хеширование сложнее в реализации, однако на его примере проще продемонстрировать суть распространенного утверждения о том, что хеш-таблица – это обычный массив с необычной адресацией. В этом случае элементы данных (или указатели на них) располагаются в массиве, размер которого выбирается таким образом, чтобы вместить максимально возможное по условиям задачи число элементов. В примере, показанном на рис. 5.19, данные представляют собой строки ограниченного размера.

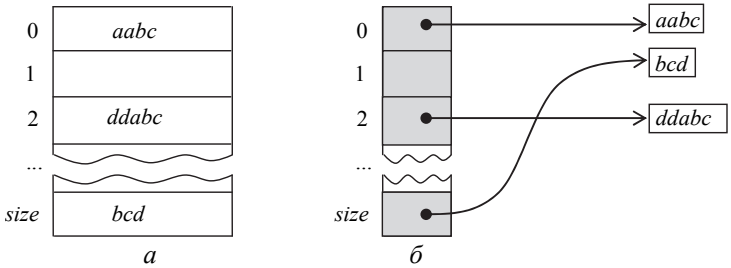


Рис. 5.19. Организация данных при закрытом хешировании:
а – таблица хранит данные; *б* – таблица хранит указатели

Описание хеш-таблиц можно было бы завершить при условии, что хеш-функция может гарантировать уникальное значение для любого ключа. Можно привести пример такой приятной ситуации: ключи – целые неотрицательные числа из интервала от 0 до m, $size = m$, $h(x) = x$. Но даже в таком примере значение m может оказаться слишком большим для размера массива, а максимальное количество хранимых элементов (обозначим его N) будет во много раз меньше m. Если выбрать $size$ исходя из N и определить хеш-функцию как остаток от деления x на $size$, то могут встретиться несколько ключей, для которых $h(x)$ примет одинаковое значение, эту ситуацию принято называть ***коллизией*** (столкновением).

Если новый элемент, который требуется добавить, претендует на уже занятое место, то для него отыскивается ближайшее справа (в сторону увеличения индексов) свободное место, перемещение выполняется циклически (дойдя до края – переход в начало). При таком подходе для любого элемента участок справа от его исконного места до его фактического места полностью заполнен, поэтому возможно корректно организовать проверку принадлежности заданного элемента таблице. Для ключа t необходимо встать на позицию $h(t)$ и двигаться направо, пока не встретится искомый элемент или пустое место, означающее его отсутствие.

Создав пустое место после удаления элемента, необходимо двигаться направо, пока не встретится еще одно пустое место (на этом можно закончить), или на элемент, стоящий не на исконном месте. Во втором случае нужно проверить: не тот ли это элемент, место которого освободилось. Если нет, то движение продолжается, если да, то после перемещения элемента образуется новое свободное место, с которым поступают аналогично. Описанный способ разрешения коллизий называют ***линейным зондированием***. Линейное зондирование можно представить следующим соотношением $h(k, i) = h(k) + i$, где h – хеш-функция, k – ключ элемента, i – номер попытки (0, 1, 2,…).

Другие наиболее популярные способы борьбы с коллизиями:

- квадратичное зондирование: $h(k, i) = (h(k) + c_1 i + c_2 i^2)$, где $c_1 \neq 0$ и $c_2 \neq 0$ – некоторые константы;
- двойное хеширование: $h(k, i) = (h(k) + h_1(k)i)$, где h_1 – вспомогательная хеш-функция.

Каким бы образом не решалась проблема коллизий при закрытом хешировании, размер таблицы должен существенно превосходить количество хранимых элементов. В противном случае возникает риск получить время выполнения операций пропорциональное количеству элементов.

5.2.2. ОТКРЫТОЕ ХЕШИРОВАНИЕ

При открытом хешировании данные, имеющие одинаковые значения хеш-функций, помещаются в списки, заголовки которых, или указатели на них расположены в массиве, как показано на рис. 5.20.

Такой подход позволяет не только упростить разрешение коллизий, но и разместить широкий диапазон возможных значений (максимальное количество которых не всегда легко оценить) в малом объеме памяти.

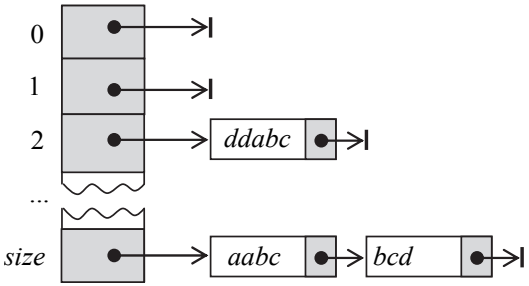


Рис. 5.20. Организация данных при открытом хешировании

5.2.3. ХЕШ-ФУНКЦИИ

Основные требования, предъявляемые к хеш-функциям, можно сформулировать так:

1) быстрота вычисления;

2) равномерность распределения элементов;

3) при повторных вызовах с одним и тем же ключом должно возвращаться одинаковое значение.

Ниже перечислены наиболее популярные и простые хеш-функции для ключей целочисленного типа и строк.

ДЕЛЕНИЕ С ОСТАТКОМ

Самая простая хеш-функция, о которой уже говорилось, представлена на рис. 5.21.

```
int Hash(int Key)
{
       return Key % HashTableSize;
}
```

Рис. 5.21. Хеш-функция – деление с остатком

Чтобы получить более случайное распределение ключей, в качестве размера таблицы (HashTableSize) нужно брать простое число, не слишком близкое к степени двух.

МЕТОД УМНОЖЕНИЯ (МУЛЬТИПЛИКАТИВНЫЙ МЕТОД)

Пусть $\lceil x \rceil$ обозначает дробную часть числа x, а $\lfloor x \rfloor$ – целую часть числа x, тогда $h(k) = \lfloor \text{HashTableSize} \lceil kA \rceil \rfloor$, где $0 < A < 1$, например, $\frac{\sqrt{5}-1}{2} \approx 0{,}6180339887$.

МЕТОД ВЫБОРА ЦИФР

Хеш-функция, основанная на выборе цифр, формирует свое значение как комбинацию отдельных цифр ключа. Например, для хеш-таблицы с размером 1000 элементов из 12-значного ключа **1**237694501**2**3 можно выбрать первую, шестую и двенадцатую цифры и сформировать индекс 143 или 341, если выбирать цифры в обратном порядке.

МЕТОД СВЕРТКИ

Метод свертки группирует цифры ключа и складывает группы цифр по модулю HashTableSize. Например, для hashTableSize = 1000 ключ 123769450123 можно преобразовать по схеме: (123 + 769 + 450 + 123) mod 1000 = 465.

КОНКАТЕНАЦИЯ БИТОВЫХ ОБРАЗОВ

Значение хеш-функции для строки можно получить, соединив битовые образы отдельных символов строки и интерпретировав результат как число, взятое по модулю HashTableSize.

Например, строковый ключ "ALPHA" можно преобразовать этим способом в следующем виде.

Буква A кодируется числом 1, или 00001 в двоичной системе.

Буква L кодируется числом 12, или 01100 в двоичной системе.

Буква P кодируется числом 16, или 10000 в двоичной системе.

Буква H кодируется числом 8, или 01000 в двоичной системе.

Конкатенируя двоичные величины, получаем двоичное число 00001011001000001000000001, которое равно 1458433.

АДДИТИВНЫЙ МЕТОД ДЛЯ СТРОК ПЕРЕМЕННОЙ ДЛИНЫ

Для строк, максимальный размер которых оценить трудно, простым и эффективным методом является сложение кодов символов.

Пример реализации представлен на рис. 5.22 с учетом размера типа char, равного одному байту, сложение выполняется по модулю 256.

```
unsigned char Hash(char *str)
{
	unsigned char h = 0;
	while (*str) h += *str++;
	return h;
}
```

Рис. 5.22. Хеш-функция для строк переменной длины

6. ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

6.1. ЛАБОРАТОРНАЯ РАБОТА № 1. ЛИНЕЙНЫЕ СТРУКТУРЫ

6.1.1. ЦЕЛЬ РАБОТЫ

Изучить линейные динамические и статические структуры данных и приобрести практические навыки программирования таких структур.

6.1.2. ЗАДАНИЕ

Разработать программную реализацию заданных согласно табл. 6.1 АТД с использованием заданной структуры данных.

Таблица 6.1

Варианты заданий

№ п/п	АТД	Операторы	Структура данных	Тип элемента
1	Стек	типичный набор (раздел 4.1)	массив	натуральное число
	Список	вставка с сохранением упорядоченности, удаление по значению, поиск позиции (номера элемента) по значению, очистка	односвязный список	символ русского алфавита
2	Очередь	типичный набор (раздел 4.2)	массив	натуральное число
	Список	вставка и удаление по номеру, поиск позиции (номера элемента) по значению, очистка	двусвязный список	строка (8 символов)
3	Дек	типичный набор (раздел 4.3)	массив	натуральное число
	Отображение	типичный набор (раздел 4.4)	односвязный список	строка (8 символов), натуральное число
4	Дек	типичный набор (раздел 4.3)	двусвязный циклический список	вещественное число

Окончание табл. 6.1

№ п/п	АТД	Операторы	Структура данных	Тип элемента
	Отображение	типичный набор (раздел 4.4)	массив	символ русского алфавита, натуральное число
5	Список	вставка с сохранением упорядоченности, удаление по значению, поиск позиции (номера элемента) по значению, очистка	массив	натуральное число
	Очередь	типичный набор (раздел 4.2)	двусвязный список	вещественное число
6	Список	вставка и удаление по номеру, поиск позиции (номера элемента) по значению, очистка	массив	любой печатный символ
	Стек	типичный набор (раздел 4.1)	односвязный список	вещественное число

6.1.3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет по работе должен содержать: цель работы; схемы структур данных; алгоритмы основных операций в виде блок-схем или псевдокода; текст программы; пример работы программы с описанием интерфейса пользователя.

6.2. ЛАБОРАТОРНАЯ РАБОТА № 2. МНОГОСВЯЗНЫЕ СПИСКОВЫЕ СТРУКТУРЫ

6.2.1. ЦЕЛЬ РАБОТЫ

Познакомиться с многосвязными структурами данных, получить представление об использовании таких структур для описания реальных данных, приобрести навыки программирования функций доступа к данным.

6.2.2. ЗАДАНИЕ

Рассмотрим две задачи. Первая состоит в учете изделий, состоящих из деталей и сложных узлов, которые, в свою очередь, имеют состав. Изделия и детали считаются элементами одного множества, отношения между которыми можно представить в виде графа, например, как показано на рис. 3.17. Вторая задача заключается в формировании маршру-

тов проезда по населенным пунктам. При наличии прямого пути из одного города в другой в графе появляется соответствующее ребро.

Пусть в первой задаче требуются функции добавления, удаления изделия (или детали), включение (и исключение) одного изделия в состав другого (с указанием количества) и вывода узлов и деталей, входящих в состав заданного изделия, вывода полного состава заданного изделия (с учетом состава всех сложных узлов).

Во второй задаче необходимо добавлять и удалять населенные пункты; добавлять и удалять пути между ними (с указанием расстояний); выводить все пункты, смежные с заданным; выводить все возможные маршруты для переезда из одного пункта в другой.

Для реализации обеих задач подойдет иерархический список, рассмотренный в разделе 3.2.4. Организовать такую структуру можно в памяти, предусмотрев сохранение и восстановление из файла, организованного как список ребер графа (раздел 3.4). В альтернативной реализации работа с иерархическим списком может выполняться непосредственно в файлах, первый содержит список вершин, второй – списки смежности. В качестве указателей в файлах используется смещение относительно начала файла (отрицательное значение – пустой указатель), вначале файла предусматривают заголовочную запись, которая содержит указатель на первый элемент и может содержать дополнительную информацию, например, размеры элемента данных, количество записей и т. п. При удалении элементы не удаляются физически, а помечаются как удаленные и исключаются из списка. Для эффективного использования дискового пространства на место удаленных элементов можно записывать новые, сами удаленные при этом целесообразно помещать в стек, пометка удаления в этом случае избыточна. При большом количестве удаленных элементов навигация по файлу при просмотре списков будет занимать много времени. Исправить ситуацию поможет процедура упаковки файлов, которая перезаписывает их таким образом, чтобы все активные записи списков были физически смежными и находились в начале файлов, а все удаленные были сгруппированы в конце.

6.2.3. ПОРЯДОК ВЫПОЛНЕНИЯ И ВАРИАНТЫ ЗАДАНИЙ

Выполнение данной работы предполагает коллектив из двух подгрупп исполнителей, каждая из которых реализует свою часть функций, для заданного в табл. 6.2. варианта задачи и способа организации списков.

Таблица 6.2

Варианты заданий

№ п/п	Задача	Организация списка	Функции
1.1	«Состав изделий»	В памяти, с возможностью сохранить в файл и прочитать из файла	Редактирование структуры
1.2			Работа с файлом, вывод полного состава
2.1		Непосредственно в файлах	Редактирование структуры
2.2			Упаковка файлов, вывод полного состава
3.1	«Населенные пункты»	В памяти, с возможностью сохранить в файл и прочитать из файла	Редактирование структуры
3.2			Работа с файлом, вывод всех маршрутов между парой пунктов
4.1		Непосредственно в файлах	Редактирование структуры
4.2			Упаковка файлов, вывод всех маршрутов между парой пунктов

6.2.4. СОДЕРЖАНИЕ ОТЧЕТА

Отчет по работе должен содержать: цель работы; схемы структур данных; описание формата использующихся файлов; текст программы; пример работы программы с описанием интерфейса пользователя.

6.3. ЛАБОРАТОРНАЯ РАБОТА № 3. ДЕРЕВЬЯ ПОИСКА

6.3.1. ЦЕЛЬ РАБОТЫ

Познакомиться со способами балансировки двоичных деревьев поиска и другими видами сбалансированных деревьев, приобрести навыки их реализации.

6.3.2. ЗАДАНИЕ

Для заданного в табл. 6.3 вида дерева разработать программу, позволяющую оценить число выполняемых операций при вставке (удалении) n элементов. Построить графики изменения среднего числа операций на вставку (удаление) одного элемента в зависимости от n. Каж-

дый из вариантов деревьев, как и в предыдущей работе, разрабатывает коллектив из двух подгрупп.

Таблица 6.3

Варианты заданий

№ п/п	Вид дерева	Операция
1.1	АВЛ-дерево	Добавление
1.2		Удаление
2.1	Декартово дерево	Добавление
2.2		Удаление
3.1	Рандомизированное дерево	Добавление
3.2		Удаление
4.1	2-3 дерево	Добавление
4.2		Удаление
5.1	2-3-4 дерево	Добавление
5.2		Удаление
6.1	Красно-черное дерево	Добавление
6.2		Удаление

6.3.3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет по работе должен содержать: цель работы; схемы структур данных; алгоритм вставки (или удаления, в зависимости от варианта) элемента в виде блок-схемы или псевдокода; результаты экспериментов в виде графиков; выводы; текст программы.

6.4. ЛАБОРАТОРНАЯ РАБОТА № 4. ХЕШ-ТАБЛИЦЫ

6.4.1. ЦЕЛЬ РАБОТЫ

Изучить различные виды хеш-функций, познакомиться с хеш-таблицами и приобрести навыки их программной реализации.

6.4.2. ЗАДАНИЕ

Для заданных в табл. 6.4 способов организации хеш-таблицы и типов элементов данных разработать программу, позволяющую добавлять элементы, подсчитывая число коллизий. Построить графики зави-

симости числа коллизий от текущего количества элементов. Провести эксперименты на одном множестве неповторяющихся элементов с использованием двух различных хеш-функций.

Таблица 6.4

Варианты заданий

№ п/п	Вид таблицы	Разрешение коллизий	Тип элементов
1	Открытая	–	Строка – идентификаторы из текстов программ
2		–	Строка – слова из текста на русском языке
3		–	Строка – наименование товара из прайс-листа
4	Закрытая	Линейное зондирование	Экспоненциально-распределенная СВ, с заданным математическим ожиданием, округленная к ближайшему целому
5			Нормально-распределенная СВ, с заданными параметрами, округленная к ближайшему целому
6		Квадратичное зондирование	Экспоненциально-распределенная СВ, с заданным математическим ожиданием, округленная к ближайшему целому
7			Нормально-распределенная СВ, с заданными параметрами, округленная к ближайшему целому
8		Двойное хеширование	Экспоненциально-распределенная СВ, с заданным математическим ожиданием, округленная к ближайшему целому
9			Нормально-распределенная СВ, с заданными параметрами, округленная к ближайшему целому

6.4.3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет по работе должен содержать: цель работы; схемы структур данных; описание выбранных хеш-функций; результаты экспериментов в виде графиков; выводы; текст программы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Романов Е.Л.* Практикум по программированию на C++: учеб. пособие. – СПб: БХВ-Петербург; Новосибирск: Изд-во НГТУ, 2004. – 432 с.

2. *Ахо А.В., Хопкрофт Д.Э., Ульман Д.Д.* Структуры данных и алгоритмы: учеб. пособие; пер. с англ. – М.: Издательский дом «Вильямс», 2000. – 384 с.

3. *Вирт Н.* Алгоритмы+структуры данных = программы: пер. с англ. – М.: Мир, 1985. – 406 с.

4. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование: пер. с англ. – М.: Мир, 1975. – 246 с.

5. *Кнут Д.Э.* Искусство программирования. Т. 3: Сортировка и поиск: пер. с англ. / общ. ред. Ю.В. Козаченко. – 2-е изд., испр. и доп. – М.: Вильямс, 2003. – 832 с.

6. *Кнут Д.Э.* Искусство программирования. Т. 1: Основные алгоритмы: пер. с англ. / общ. ред. Ю.В. Козаченко. – 3-е изд., испр. и доп. – М.: Вильямс, 2002. – 720 с.

7. *Адельсон-Вельский Г.М., Ландис Е.М.* Один алгоритм организации информации // Доклады АН СССР. – 1962. Т. 146. № 2. – С. 263–266.

8. *Guibas L.J., Sedgewick R.* A dichromatic framework for balanced trees // Proceedings of the 19th Annual Symposium on Foundations of Computer Science, 1978. – Pp. 8–21.

9. *Седжвик Р.* Фундаментальные алгоритмы на C++. Анализ. Структуры данных. Сортировка. Поиск. – Киев: ДиаСофт, 2001. – 688 с.

10. Martinez C., Roura S. Randomized binary search trees // Journal of the ACM (ACM Press). 1998. № 45 (2). – Pp. 288–323.

11. Aragon C.R., Seidel R.G. Randomized search trees // 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, 30 October – 1 November 1989. IEEE. Pp. 540–545.

12. Bayer R., McCreight E. Organization and Maintenance of Large Ordered Indices // Mathematical and Information Sciences Report. 1970. № 20, Boeing Scientific Research Laboratories.

13. Каррано Ф.М., Причард Дж. Абстракция данных и решение задач на C++. Стены и зеркала. 3-е издание: пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 848 с.

Ландовский Владимир Владимирович

СТРУКТУРЫ ДАННЫХ

Учебное пособие

В авторской редакции

Выпускающий редактор *И.П. Брованова*
Корректор *Е.Н. Николаева*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *С.И. Ткачева*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 21.11.2016. Формат 60 × 84 1/16. Бумага офсетная. Тираж 100 экз.
Уч.-изд. л. 3,95. Печ. л. 4,25. Изд. № 179. Заказ № 31. Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20