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

КАФЕДРА БЕЗОПАСНОСТИ ИНФОРМАЦИОННЫХ СИСТЕМ

А.Н. Крутов

МЕТОДЫ ПРОГРАММИРОВАНИЯ. КЛАССИЧЕСКИЕ АЛГОРИТМЫ

Учебное пособие

для студентов
механико-математического факультета
специальности 075200 – «Компьютерная безопасность»

Издательство «Самарский университет»
2004

Печатается по решению Редакционно-издательского совета Самарского государственного университета

УДК 519.688
ББК 32.973-01
К 846

Крутов А.Н. Методы программирования. Классические алгоритмы. Учебное пособие. Самара: Изд-во «Самарский университет», 2004. 78 с.

Основой данного учебного пособия являются материалы лекций, прочитанных автором по первой половине курса “Методы программирования” для специальности 075200 – «Компьютерная безопасность».

В пособии кратко рассматриваются наиболее распространенные в настоящее алгоритмы по обработки информации.

Предназначено для студентов механико-математического факультета Самарского государственного университета (специальность «Компьютерная безопасность»).

УДК 519.688
ББК 32.973-01

Рецензент доцент Е.В Рогачева

© Крутов А.Н., 2004

© Изд-во «Самарский университет», 2004

СОДЕРЖАНИЕ

Введение **4**

1. Структуры данных **5**

1.1 Треугольные массивы 7
1.2 Списки 8
1.3 Стеки 13
1.4 Очереди. Деки. 14
1.5 Деревья 15

2. Сортировка **33**

2.1 Внутренняя сортировка 33
2.2 Внешняя сортировка 44

3. Поиск **45**

3.1 Последовательный поиск 45
3.2 Поиск путем сравнения ключей 47
3.3 Хеширование 49

4. Сетевые алгоритмы **54**

4.1 Представления сетей 55
4.2 Обход сети 57
4.3 Наименьший каркас дерева 59
4.4 Кратчайший путь 60

5. Задачи для лабораторных работ **74**

Заключение **76**

Библиографический список **77**

ВВЕДЕНИЕ

В настоящее время компьютерная техника развивается стремительными темпами. Однако ключевая роль в достижении успеха большинства компьютеризованных систем принадлежит не используемому оборудованию, а программному обеспечению. Поэтому задача разработки качественных и надежных программных продуктов с использованием оптимальных алгоритмов никогда не перестанет быть актуальной. Настоящее пособие является первой частью цикла учебных пособий по курсу "Методы программирования" для специальности 075200 – Компьютерная безопасность. Данное пособие посвящено описанию алгоритмов, ставших уже классическими, несмотря на относительно небольшую историю развития информатики и программирования.

В первой главе настоящего пособия описываются основные структуры данных, применяемые в программировании. К ним относятся массивы, списки, стеки, очереди, а также деревья. Вторая глава посвящена описанию алгоритмов сортировки. В начале описываются алгоритмы внутренней сортировки, когда весь сортируемый массив данных считывается в оперативную память. Далее приводятся алгоритмы внешней сортировки, для которых характерно наличие такого объема данных, который не представляется возможным целиком разместить в оперативной памяти. В третьей главе приводятся различные алгоритмы поиска информации в упорядоченных и неупорядоченных массивах. Далее в пособии вводится понятие хеш-функции и предлагаются ряд алгоритмов для работы с ними. Четвертая глава рассматривает алгоритмы на графах. К ним относятся обход графа, поиск кратчайшего пути и задача о максимальном потоке. В конце пособия по итогам всех приведенных тем предлагается список задач для лабораторных работ по курсу.

Данное пособие представляет собой первую часть лекционного курса составителя по дисциплине "Методы программирования" для специальности 075200 – Компьютерная безопасность. При подготовке данного методического пособия активно использовалась литература, приведенная в библиографическом списке, а также материалы internet-сайтов, таких как http://www.citforum.ru и http://algoritm.manual.ru. Составитель данного пособия никоим образом не собирается посягать ни на авторство алгоритмов, ни на авторство материалов, приводимых в данном пособии. В начале каждой темы дается ссылка на первоисточник с тем, чтобы студент в случае необходимости смог обратиться за дополнительной информацией. Библиографический список данного пособия намеренно не содержит слишком большое количество монографий, чтобы акцентировать внимание студентов на самых главных их них. При желании можно открыть последнюю страницу любой из приведенных монографий с тем, чтобы получить более развернутый список литературы.

1. СТРУКТУРЫ ДАННЫХ

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

Под **структурой данных** в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты. Поэтому вводится дополнительная классификация структур данных, направления которой соответствуют различным аспектам их рассмотрения. Прежде чем приступать к изучению конкретных структур данных, необходимо привести их общую классификацию по нескольким признакам.

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

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

Различаются **простые** (базовые, примитивные) структуры (типы) данных и **интегрированные** (структурированные, композитные, сложные). **Простыми** называются такие структуры данных, которые не могут быть расчленены на составные части, большие, чем биты. С логической точки зрения простые данные являются неделимыми единицами. **Интегрированными** называются такие структуры данных, составными частями которых являются другие структуры данных – простые или в свою очередь интегрированные. Интегрированные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

Весьма важный признак структуры данных - ее **изменчивость** - изменение числа элементов и (или) связей между элементами структуры. В определении изменчивости структуры не отражен факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости. По признаку изменчивости различают структуры **статические**, **полустатические** и **динамические**. Классификация структур данных по признаку изменчивости приведена ниже на рис. 1. Ба-

---

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

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

Рис. 1. Классификация структур данных

Важный признак структуры данных - **характер упорядоченности** ее элементов. По этому признаку структуры можно делить на **линейные** и **нелинейные** структуры.

В зависимости от характера взаимного расположения элементов в памяти линейные структуры можно разделить на структуры с **последовательным распределением** элементов в памяти (векторы, строки, массивы, стеки, очереди) и структуры с **произвольным** связным распределением элементов в памяти (односвязные, двусвязные списки). Пример нелинейных структур - многосвязные списки, деревья, графы.

В языках программирования понятие "структуры данных" тесно связано с понятием "типы данных". Любые данные, т.е. константы, переменные, значения функций или выражения, характеризуются своими типами.

Информация по каждому типу однозначно определяет :

1) структуру хранения данных указанного типа, т.е. выделение памяти и представление данных в ней, с одной стороны, и интерпретация двоичного представления, с другой;

2) множество допустимых значений, которые может принимать тот или иной объект описываемого типа;

3) множество допустимых операций, которые применимы к объекту описываемого типа.

Далее в текущем разделе приводится описание интегрированных структур данных, которые широко распространены на практике и используются в настоящем пособии.

1.1 Треугольные массивы[2]

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

Массив А:

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

Массив В:

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

Рис. 2. Упаковка треугольного массива в одномерный массив

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

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

При практической реализации данного способа хранения представляется целесообразным в качестве В использовать динамический массив.

---

[2] Составлено на основе материалов монографии [6]

Безусловно, обращение к его элементам будет происходить медленнее, чем к элементам массива А. Поэтому использование данного алгоритма является оправданным лишь для больших массивов, когда появляются проблемы с хранением информации.

1.2 Списки[3]

В данном разделе описываются методы создания динамических списков. Различные виды списков имеют различные свойства. Некоторые из них достаточно просты и функционально ограничены. Другие же, такие, как циклические, двусвязные списки и списки с указателями, сложнее и поддерживают более развитые средства управления данными.

Простые списки

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

```
const
  MaxArrayIDX=1000;
type
  TIntArray=array[1..MaxArrayIDX] of integer;
  PIntArray=^TIntArray;
var Items:PIntArray;

procedure CreateAndFillArray(NumItems:integer);
  var i:integer;
begin
  GetMem(Items,NumItems*SizeOf(Integer));
  for i:=1 to NumItems do Items^[i]:=i;
end;

procedure DisposeArray;
begin
  FreeMem(Items);
end;
```

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

Приведенный выше код является примером того, как не надо писать программы, так как данный код имеет потенциальные возможности для ошибок в программе. Дело в том, что в приведенном фрагменте массив воспринимается как указатель, содержащий MaxArrayIDX ячеек. Если происходит выделение памяти только для десяти элементов, то исполнимая программа не определит попытку доступа к 100-му элементу массива как ошибку. В лучшем случае обращение к данной ячейке аварийно остановит программу. В худшем случае это вызовет неявный сбой, который будет достаточно сложно найти. Подобные же проблемы возникнут при неверном задании нижней границы массива.

Несмотря на подстерегающие опасности изменение размеров массива– очень мощная методика, позволяющая достигать большой производительности. Среда программирования Delphi, начиная с версии 4.0, поддерживает встроенный механизм динамических массивов.

Ниже представлен код программы, реализующий создание и уничтожение массивов данного типа.

```
var Items:array of integer;

procedure CreateAndFillArray(NumItems:integer);
  var i:integer;
begin
  SetLength(Items,NumItems);
  for i:=Low(Items) to High(Items) do Items[i]:=i;
end;

procedure DisposeArray;
begin
  SetLength(Items,0);{или Items:=nil}
end;
```

Список переменного размера

С помощью динамических массивов легко построить простой список переменного размера. Новый элемент в список добавляется следующим образом. Создается новый массив, который на один элемент больше старого. Далее копируются элементы старого массива в новый и добавляется новый элемент. Затем освобождается старый массив и устанавливается указатель массива на новую страничку памяти. Если использовать динамические массивы, реализованные в Delphi, то алгоритм добавления элемента в конец списка будет еще проще - при изменении размера массива программа автоматически создает новый и копирует в него содержимое старого:

```
var List:array of integer;

procedure AddItem(new_item:integer)
begin
  // Увеличиваем размер массива на 1 элемент.
  SetLength(List,Length(List)+l);
  // Сохранение нового элемента.
  List[High(List)]:=new_item;
end;
```

Эта простая схема хорошо работает для небольших списков, но у нее есть два существенных недостатка. Во-первых, приходится часто менять размер массива. Чтобы создать список из 1000 элементов, необходимо 1000 раз изменить размеры массива. Ситуация осложняется еще тем, что чем больше становится список, тем больше времени потребуется на изменение его размера, поскольку необходимо каждый раз копировать растущий список в заново выделенную память[4]. Чтобы размер массива изменялся не так часто, при его увеличении можно вставлять дополнительные элементы, например, по 10 элементов вместо 1. Если впоследствии возникает необходимость добавлять новые элементы к списку, то они разместятся в уже существующих в массиве неиспользованных ячейках, не увеличивая размер массива. Точно так же можно избежать изменения размера каждый раз при удалении элемента из списка. Свободные ячейки позволяют избежать изменения размеров массива всякий раз, когда необходимо добавить или удалить элемент из списка.

Неупорядоченные списки

В некоторых приложениях требуется удалять одни элементы из середины списка, добавляя другие в его конец. Это может быть в случае, когда порядок элементов не важен, но необходимо иметь возможность удалять определенные элементы из списка. Списки данного типа называются **неупорядоченными списками**.

Неупорядоченный список должен поддерживать следующие операции:

- добавление элемента к списку;
- удаление элемента из списка;
- определение наличия элемента в списке;

---

[4] Правильнее будет сказать, что функция SetLength выполняет копирование растущего списка в заново выделенную память только в том случае, если его не удается расширить текущий сегмент памяти, отведенный под хранение массива. Это является одним из достоинств динамических массивов в Delphi.

- выполнение каких-либо операций (например, печати или вывода не дисплей) для всех элементов списка.

Для управления подобным списком можно немного изменить простую схему, представленную в предыдущем разделе. Когда удаляется элемент из середины списка, оставшиеся элементы сдвигаются на одну позицию, заполняя образовавшийся промежуток. Однако удаление элемента массива подобным способом может занимать много времени, особенно если этот элемент находится в начале списка. Чтобы удалить первый элемент массива, содержащего 1000 записей, необходимо сдвинуть 999 элементов на одну позицию влево. Гораздо быстрее удалять элементы при помощи простой схемы “сборки мусора“. Вместо удаления элементов из списка можно их отметить как неиспользуемые. Если элементы списка – данные простых типов, то их можно маркировать с помощью так называемого "мусорного" значения. Если элементы списка - это структуры, определенные оператором type, то можно добавить к ним новое логическое поле IsGarbage. При удалении элемента из списка значение поля IsGarbage устанавливается в True.

Через некоторое время список может переполниться "мусором", в результате чего будет тратиться большое количество машинного времени на пропуск ненужных элементов, чем на обработку реальных данных. Во избежание такой ситуации надо периодически выполнять процедуру сборки мусора. Эта процедура перемещает все непомеченные элементы в начало массива. После этого они добавляются к неиспользуемым элементам в конце массива. Когда вам потребуется включить в список дополнительные элементы, можно повторно использовать помеченные ячейки без изменения размера массива. После добавления дополнительных неиспользуемых записей к другим свободным ячейкам полный объем свободного пространства может стать слишком большим. В этом случае следует уменьшить размер массива, чтобы освободить память.

Связанные списки

В отличие от неупорядоченных списков, элементы которых никак не связаны друг с другом [5], отличительной особенностью данного вида списков является именно наличие связей между его элементами. На рис. 3 представлена структура односвязного списка.

Рис. 3. Структура односвязного списка

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

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

Данный вид списка обладает некоторыми важными свойствами. Во-первых, в начало данного списка очень просто добавлять новые элементы (рис. 4). Для этого необходимо создать новую ячейку, указывающую на текущую вершину списка, а затем сделать так, чтобы новой вершиной списка стала только что созданная ячейка. Соответствующий код на Pascal для этой операции достаточно прост:

```
new_cell^.NextCell:=top_cell;
top_sell:=new_cell;
```

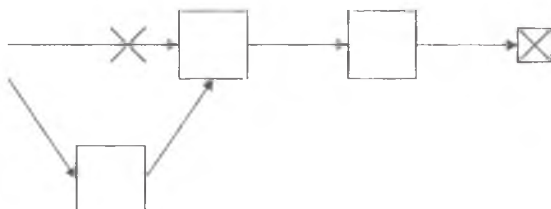


Рис. 4. Добавление элемента в начало связанного списка

Так же легко осуществляется вставка нового элемента и в середину связанного списка. Предположим, необходимо добавить новый элемент после ячейки, на которую указывает переменная after_me. В этом случае соответствующий код на Pascal имеет вид:

```
new_cell^.NextCell:=after_me^.NextCell;
after_me^.NextCell:=new_cell
```

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

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

Если в программе возникает необходимость обхода элементов списка как в прямом, так и обратном порядке, то для этого случая представляется

целесообразным использование другой разновидности связанных списков, а именно, двусвязных списков. Каждый элемент списка хранит указатель на следующий элемент и на предыдущий (рис. 5).

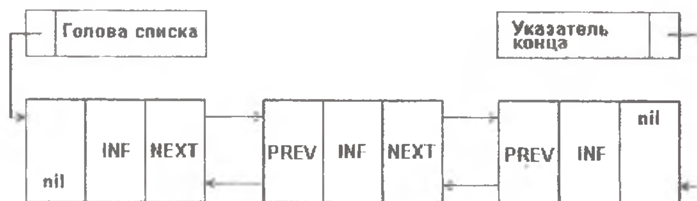


Рис. 5. Представление двусвязного списка

1.3 Стеки[6]

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

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

```
var Stack:array of integer;

procedure Push(value:integer);
begin
  SetLength(Stack, Length(Stack)+1);
  Stack[High(Stack)]:=value
end;

function Pop:integer;
begin
  Result:=Stack[High(Stack)];
  SetLength(Stack, Length(Stack)-1)
end;
```

Данный способ реализации стека сохраняет все недостатки, которые имели место для неупорядоченных списков, реализованных подобным образом. Однако вместе с тем допускает и те же возможности для его опти-

[6] Составлено на основе материалов монографии [6]

мизации, т.е. изменение размеров массива не поэлементно, а сразу для некоторой группы и т.п. Однако все же наиболее приемлемым как с точки зрения затрат памяти, так и с точки зрения скорости работы, способом реализации стека является использование односвязных списков. В этом случае в качестве операции помещения элемента в стек следует использовать процедуру добавления элемента в начало списка, а в качестве операции извлечения элемента из стека следует использовать операцию получения первого его значения с последующим удалением из списка.

1.4 Очереди. Деки[7]

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

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

Очереди с приоритетом

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

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

Пожалуй, самый простой способ организации очереди с приоритетами - поместить все элементы в список. Если требуется удалить элемент из очереди, надо найти в списке элемент с наивысшим приоритетом. Чтобы

---

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

добавить элемент к очереди, необходимо просто разместить новый элемент в начале списка. Если очередь содержит N элементов, требуется O(N) шагов, чтобы определить положение и удалить из очереди элемент с максимальным приоритетом.

В качестве второго способа организации очереди с приоритетом можно также использовать связанный список, однако хранить элементы, располагая их в порядке уменьшения приоритета. Тип данных списка TPriorityCell можно определить следующим образом:

```
type
 PPriorityCell = ^TPriorityCell;
 TPriorityCell = record
   Value:string;              // Данные
   Priority:integer;          // Приоритет элемента
   NextCell:PPriorityCell;    // Следующая ячейка
 end;
```

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

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

1.5 Деревья[8]

Деревья, пожалуй, являются одними из наиболее важных нелинейных структур, которые встречаются при работе с компьютерными алгоритмами. Формально дерево определяется как конечное множество T одного или более узлов со следующими свойствами:

a) существует один выделенный узел, а именно- **корень** данного дерева T;
b) остальные узлы (за исключением корня) распределены среди m≥0 непересекающихся множеств T_1,....T_m, и каждое из этих множеств, в свою очередь, является деревом; деревья T_1,..., T_m называются **поддеревьями** данного корня.

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

[8] Составлено на основе материалов монографий [4,6]

На рис. 6 изображено дерево. Корневой узел A соединен с тремя поддеревьями, начинающимися узлами B, C и D. Эти узлы соединены с поддеревьями, имеющими корни в узлах E, F и G, которые в свою очередь связаны с поддеревьями с корнями H, I и J.

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

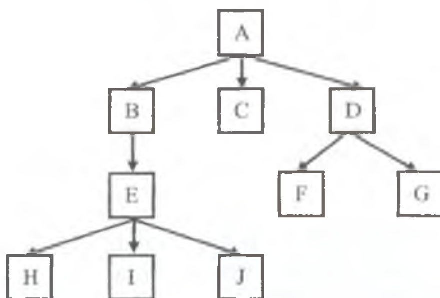


Рис. 6. Дерево

Из генеалогии пришли термины, описывающие отношения. Если узел находится непосредственно над другим, то он называется **родительским**, а нижний узел называется **дочерним**. Узлы на пути вверх от узла до корня принято считать **предками узла**. Например, на рис. 6 предками узла I являются узлы E, B и A. Все узлы, расположенные ниже какого-либо узла, называются его потомками. На рис. 6 потомками узла B являются узлы E, H, I и J. Узлы, имеющие одного и того же родителя, называются **сестринскими**.

Кроме того, существует несколько понятий, возникших собственно в программистской среде. **Внутренний узел** - это узел, не являющийся листом. **Порядком узла** или его **степенью** называется количество его дочерних узлов. **Степень дерева** - это максимальный порядок его узлов. Степень дерева, изображенного на рис. 6, равна 3, так как узлы A и E, имеющие максимальную степень, имеют по три дочерних узла.

Высота узла равна числу его предков плюс 1. На рис. 6 узел E имеет высоту 3. **Высота дерева** - это наибольшая высота всех узлов. Высота дерева, изображенного на рис. 6, равна 4.

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

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

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

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

Обход деревьев

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

Прямой порядок:

1. Обращение к узлу.
2. Рекурсивный прямой обход левого поддерева.
3. Рекурсивный прямой обход правого поддерева.

Симметричный порядок:

1. Рекурсивный симметричный обход левого поддерева.
2. Обращение к узлу.
3. Рекурсивный симметричный обход правого поддерева.

Обратный порядок:

1. Рекурсивный обратный обход левого поддерева.
2. Рекурсивный обратный обход правого поддерева.
3. Обращение к узлу.

Для дерева, изображенного на рисунке 7, порядок обхода будет следующим:

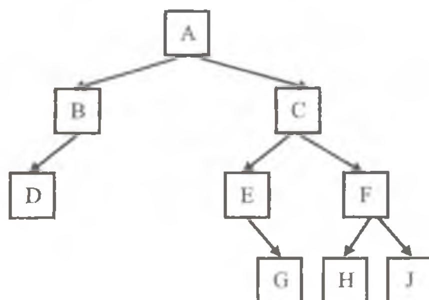


Рис. 7. Пример дерева для иллюстрации различных методов его обхода

Прямой порядок:	A, B, D, C, E, G, F, H, J
Симметричный порядок:	D, B, A, E, G, C, H, F, J
Обратный порядок:	D, B, G, E, H, J, F, C, A

Упорядоченные деревья

Двоичные деревья - обычный способ хранения и обработки информации в компьютерных программах. Если использовать внутренние узлы дерева, чтобы обозначить утверждение "левый дочерний узел меньше правого", то с помощью двоичного дерева можно построить сортированный список. На рис. 8 показано двоичное дерево, хранящее сортированный список с числами 1,2,4,6,7,9.

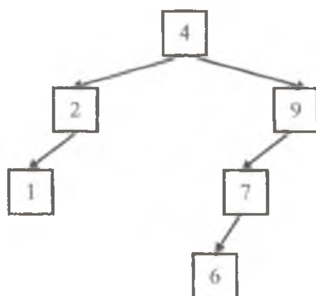


Рис. 8. Упорядоченный список 1, 2, 4, 6, 7, 9

Добавление элементов

Алгоритм добавления нового элемента в такой тип деревьев достаточно прост. С начала берется корневой узел. Далее следует по очереди срав-

нивать значения всех узлов со значением нового элемента. Если новое значение меньше или равно значению в рассматриваемом узле, то необходимо продолжать движение вниз по левой ветви. Если новое значение больше значения узла, то переходить следует вниз по правой ветви. Если достигнут конец дерева [9], необходимо вставить элемент в эту позицию.

Удаление элементов

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

Во-первых, если удаляемый узел не имеет потомков, допустимо просто убрать его из дерева. При этом порядок остальных узлов не изменяется. Во-вторых, если удаляемый узел имеет один дочерний узел, можно заменять его дочерним узлом. При этом порядок потомков данного узла остается так как эти узлы также являются и потомками дочернего узла. На рис. 9 показано дерево, где удаляется узел 4, имеющий только один дочерний узел.

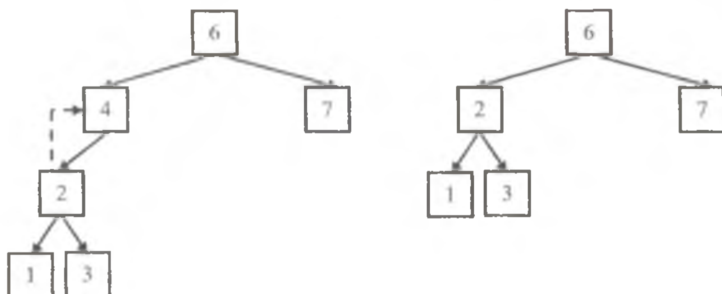


Рис. 9. Удаление узла с единственным потомком

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

---

[9] Под "достижением конца дерева" понимается попытка перехода с текущего узла по ссылке на дочерний, однако соответствующее значение ссылки является пустым

ветви. Если найденный узел является листом (на рис. 10 это узел 3), то им следует заменить удаляемый узел.

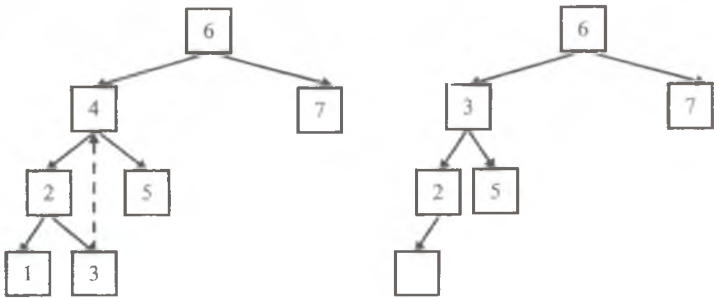


Рис. 10. Удаление узла с двумя потомком

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

Сбалансированные деревья

После выполнения ряда операций с упорядоченное дерево, таких как вставка и удаление узлов, оно может стать несбалансированным. Если подобное происходит, алгоритмы обработки дерева становятся менее эффективными. При сильной степени разбалансировки дерево фактически вытягивается в линейный список [10] , а у программы, использующей дерево, может резко снизиться производительность. В данном разделе рассматриваются методы сохранения баланса дерева даже при постоянной вставке и удалении элементов.

Балансировка

Форма упорядоченного дерева зависит от порядка добавления элементов. Высокие, тонкие деревья могут иметь глубину до O(N), где N- число узлов дерева. Добавление или размещение элемента в таком разбалансированном дереве может занимать O(N) шагов. Даже если новые элементы размещаются беспорядочно, в среднем они дадут дерево с глубиной N/2, обработка которого потребует так же порядка O(N) операций.

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

---

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

AVL-деревья

AVL-деревья были названы по имени российских математиков Адельсона-Вельского и Ландау, которые их изобрели. В каждом узле AVL-дерева глубина левого и правого поддеревьев отличаются не более чем на единицу. На рис. 11 изображено несколько AVL-деревьев.

Рис. 11. AVL – деревья

AVL-дерево имеет глубину $O(\log_2 N)$. Следовательно, и поиск узла в AVL-дереве занимает время порядка $O(\log_2 N)$, что при больших N существенно меньше, чем O(n).

Добавление узлов к AVL-дереву

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

Вращение AVL-деревьев

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

Предположим, что происходит добавление нового узла к AVL-дереву и оно теперь разбалансировано в узле X, как показано на рис. 12. На рисунке изображены только узел X и два его дочерних узла, остальные части дерева обозначены треугольниками, так как нет необходимости их рассматривать. Новый узел может быть вставлен в любое из этих четырех поддеревьев, изображенных в виде треугольников ниже узла X. В зависи-

мости от этого происходит соответствующий вид вращения. Однако следует помнить, что вращение следует применять только в случае, если вставляемый узел нарушает упорядоченность AVL-дерева.

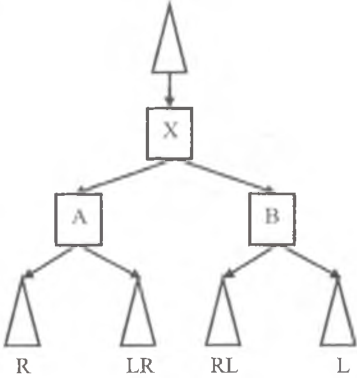


Рис. 12. Анализ разбалансированного AVL-дерева

Правое вращение

Предположим, что новый узел добавляется к поддереву R на рис. 12. В этом случае поддеревья RL и L изменяться не будут, поэтому их можно сгруппировать в один треугольник, как показано на рис. 13. Новый узел был добавлен к дереву T_1, при этом поддерево T_A с корнем в узле A становится по крайней мере на два уровня длиннее, чем поддерево T_3.[11] Механизм правого вращения представлен на рис. 13. Это вращение называется правым, поскольку узлы A и X сдвигаются на одну позицию вправо.

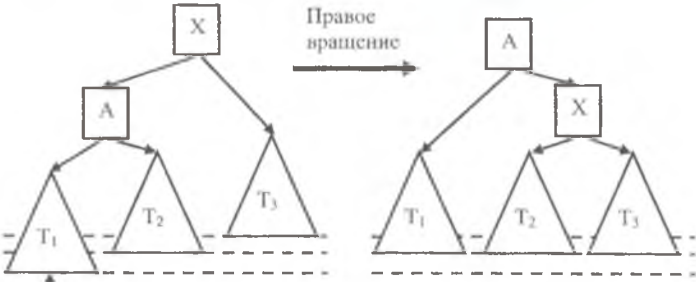


Рис. 13. Механизм правого вращения AVL-деревьев

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

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

Левое вращение

Левое вращение аналогично правому. Оно используется с целью перебалансировки дерева, когда новый узел добавляется к поддереву L, показанному на рис. 12. На рис. 14. изображено AVL-дерево до и после левого вращения.

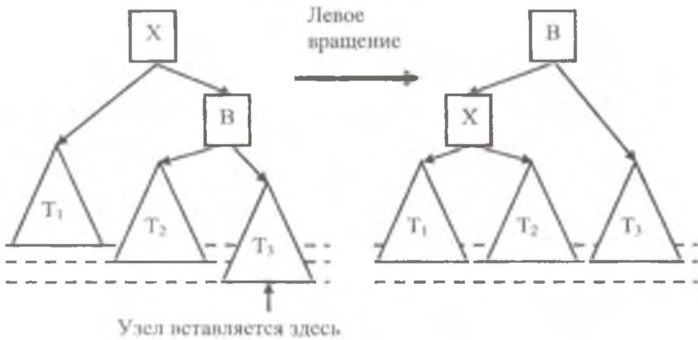


Рис. 14. Механизм левого вращения AVL-деревьев

Вращение влево-вправо

Когда узел добавляется в поддерево LR (см. рис. 12), необходимо рассмотреть еще один нижележащий уровень. На рис. 15 показано дерево, в котором новый узел вставляется в левую часть T_2 поддерева LR. В данном случае поддеревья T_A и T_C удовлетворяют свойству AVL, а поддерево T_X - нет.

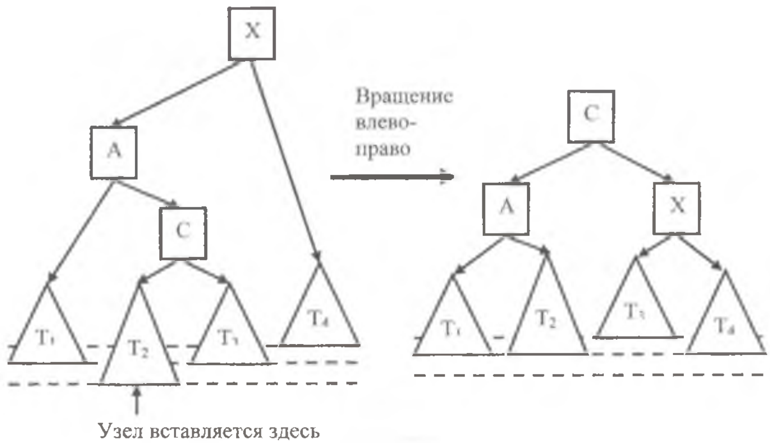


Рис. 15. Механизм вращения AVL-деревьев влево-вправо

Перебалансировка деревьев по принципу влево-вправо показана на рис. 15. Данный случай называется вращением влево-вправо, потому что узлы A и C сдвигаются на одну позицию влево, а узлы C и X - на одну позицию вправо.

Вращение вправо-влево

Вращение вправо-влево аналогично вращению влево-вправо. Оно используется для балансировки дерева после вставки узла в поддерево RL, изображенное на рис. 12. На рис. 16 показано AVL-дерево до и после вращения вправо-влево.

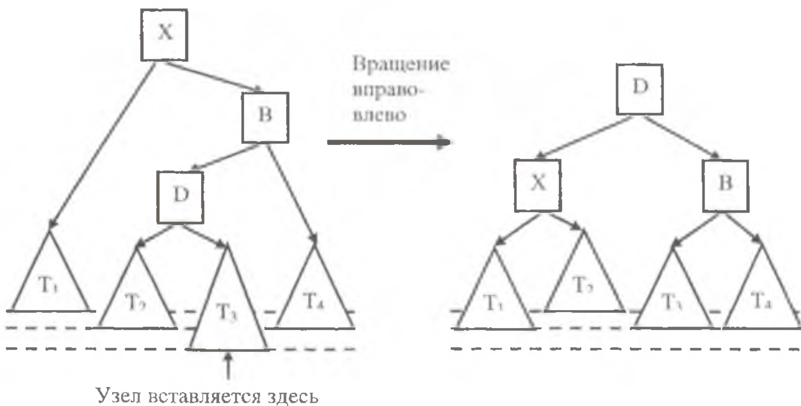


Рис. 16. Механизм вращения AVL-деревьев вправо-влево

Удаление узлов из AVL-дерева

Удаление узлов из AVL-деревьев осуществляется по тем же правилам, что и для обычных бинарных упорядоченных деревьев. Однако после этого необходимо проверить баланс дерева. Если найдется узел, где не выполняется свойство AVL-деревьев, необходимо осуществить соответствующее вращение, чтобы перебалансировать дерево.

Красно-черные деревья [12]

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

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

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

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

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

Вставка

Чтобы вставить узел, необходимо сначала найти соответствующее место в дереве. Новый узел всегда добавляется как лист, поэтому оба его потомка являются NIL-узлами и предполагаются черными. После вставки необходимо окрасить узел в красный цвет. После этого необходимо проверить все вышеприведенные свойства для его предка. Далее в случае необ-

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

ходимости осуществляется перекрашивание узла и/или поворот, чтобы сбалансировать дерево.

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

Красный предок, красный "дядя". Данную ситуацию иллюстрирует рис. 17. У нового узла X предок и "дядя" оказались красными. В данном случае простое перекрашивание избавляет от нарушения свойства 3. После перекраски нужно проверить "дедушку" нового узла (узел B), поскольку он может оказаться красным. Следует обратить внимание на то, что в данном случае меняется цвет корня дерева, т.к. в противном случае нарушилось бы свойство черной высоты.

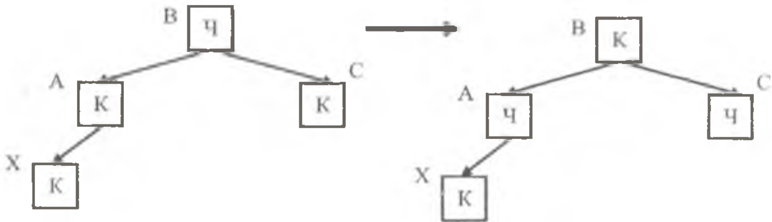


Рис. 17. Вставка – Красный предок, красный дядя

Красный предок, черный "дядя". На рис. 18 представлен другой пример нарушения свойства 3 красно-черных деревьев. В данном случае для коррекции необходимо применить дополнительно вращение узлов. Следует обратить внимание, что если узел X был в начале правым потомком, то для коррекции следует применять левое вращение, которое делает этот узел левым потомком.

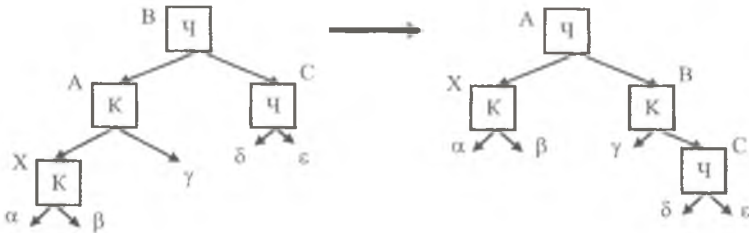


Рис. 18. Вставка – красный предок, черный дядя

Деревья со случайным поиском [13]

При вводе элемента в дерево случайного поиска этому элементу присваивается приоритет - некоторое вещественное число с равномерным распределением в диапазоне [0, 1]. Приоритеты элементов в дереве случайного поиска определяют их положение в дереве в соответствии с правилом: приоритет каждого элемента в дереве не должен быть больше приоритета любого из его последователей. Правило двоичного дерева поиска также остается справедливым: для каждого элемента x элементы в левом поддереве должны быть меньше, чем x, а в правом – больше, чем x. На рис. 19 приведен пример такого дерева случайного поиска, на котором приоритеты каждого элемента указаны в виде нижнего индекса.

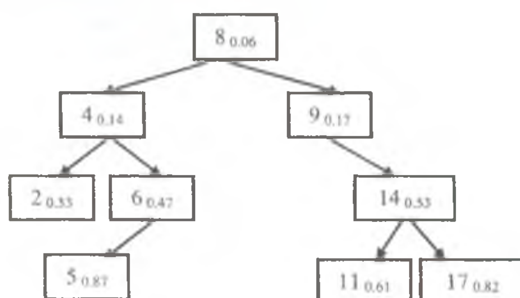


Рис. 19. Дерево случайного поиска. Приоритеты каждого элемента указаны в виде нижнего индекса

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

Б-деревья [14]

Б-деревья (B-tree) [15] – другая форма сбалансированного дерева. Каждый узел в Б-дереве содержит ключи и несколько указателей на дочерние узлы. Поскольку каждый узел хранит несколько элементов, узлы часто на-

[13] Составлено с использованием материалов сайта http://algolist.manual.ru
[14] Составлено на основе материалов монографии [6]
[15] Свое название Б-деревья получили по фамилии их разработчика Байера (Bayer)

зываются сегментами. Между каждой парой смежных указателей в узле находится ключ, который используется для определения ветви, по которой нужно двигаться при добавлении или поиске элемента. На рис. 20 изображено Б-дерево, содержащее два ключа – G и R.

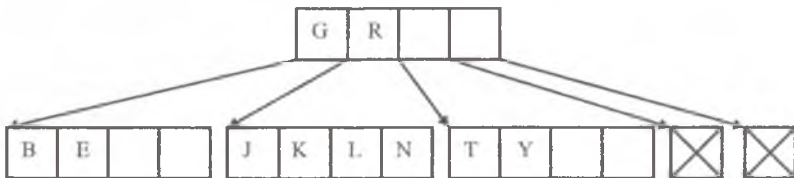


Рис. 20. Б-дерево

Чтобы разместить элемент со значением меньшим G, нужно следовать вниз по первой ветви. Чтобы найти значение между G и R, необходимо пройти вниз по второй ветви. Разместить элемент со значением большим R можно, выбрав третью ветвь.

Б-дерево порядка К обладает следующими свойствами:

- каждый узел содержит максимум 2К ключей;
- каждый узел, за исключением корня, содержит не менее К ключей;
- внутренний узел, где расположено М ключей, имеет М + 1 дочерних узлов;
- все листья дерева находятся на одном уровне.

Б-дерево на рис. 20 имеет порядок 2. Каждый узел может содержать до четырех ключей. Каждый узел, кроме корня, должен содержать не менее двух ключей.

Требование, чтобы каждый узел в Б-дереве порядка К содержал от К до 2К ключей, поддерживает баланс дерева. Поскольку каждый узел должен содержать, по крайней мере, К ключей, он должен иметь не меньше К+1 дочерних узлов, поэтому дерево не может стать слишком высоким и тонким.

Производительность Б-дерева

Применение Б-деревьев особенно полезно при создании приложений, предназначенных для работы с базами данных. При большом порядке Б-дерева любой его элемент можно найти после рассмотрения всего нескольких узлов. Например, Б-дерево 10-го порядка, содержащее миллион записей, может быть глубиной максимум шесть уровней. Для нахождения конкретного элемента необходимо исследовать максимум шесть узлов.

Добавление элементов в Б-дерево

Чтобы вставить новый элемент в Б-дерево, необходимо определить лист, в который должен быть помещен новый элемент. Если этот узел содержит менее чем 2К ключей, то в нем есть место для вставки нового элемента. В этом случае следует просто добавить новый элемент в соответствующую позицию, чтобы элементы узла были упорядочены.

Если узел уже содержит 2К элементов, то места для нового элемента в узле уже не остается. Чтобы создать необходимое пространство, необхоимо поделить узел на два новых узла.

Например, нужно вставить элемент Q в Б-дерево, показанное на рис. 20. Этот новый элемент принадлежит второму листу, который уже заполнен. Чтобы разбить узел, необходимо поделить элементы J, К, N, Р и Q между двумя новыми узлами. Для этого элементы J и К располагают в левом узле, Р и Q - в правом, а затем перемещают средний элемент N в родительский узел. На рис. 21 изображено полученное дерево.

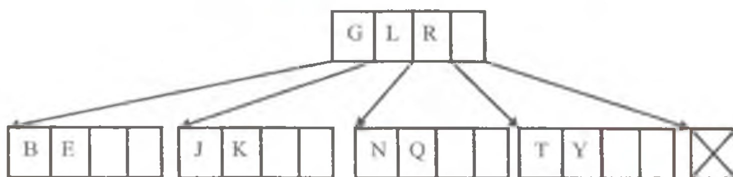


Рис. 21. Б-дерево после добавления элемента Q

Деление узла на два называется **дроблением сегмента**. Когда оно происходит, родительский узел получает новый ключ и новый указатель. Если родительский узел уже полон, то добавление нового ключа и указателя также может привести к его дроблению, что, в свою очередь, потребует добавления новой записи на более высоком уровне и т.д. В наихудшем случае добавление элемента вызывает "цепную реакцию" дробления узлов вплоть до корня.

Когда происходит дробление корня, Б-дерево становится глубже. Это единственный способ увеличить его глубину. Поэтому Б-деревья обладают необычным свойством - **они всегда растут от листьев к корню**.

Удаление элементов из Б-дерева

Если удаляемый элемент находится не в листе, его необходимо заменить его другим элементом, чтобы сохранить соответствующий порядок расположения. Это похоже на случай удаления элемента из сортированного дерева или AVL-дерева, поэтому можно обрабатывать этот случай по-

добным образом, т.е. заменить элемент крайним правым элементом из левой ветки. Этот элемент будет всегда находиться в листе. После замены элемента можно считать, что вместо него просто удален заменивший его лист.

Чтобы удалить элемент из листа, сначала необходимо сдвинуть все другие элементы влево, чтобы заполнить оставшееся место. Однако следует помнить, что каждый узел в Б-дереве порядка К должен содержать от К до 2К элементов. После удаления элемента из листа, он может содержать всего К-1 элементов. В этом случае необходимо взять несколько элементов з узлов на том же уровне, а затем перераспределить элементы так, чтобы узлы содержали не менее К элементов. На рис. 22 элемент был удален из крайнего левого листа в дереве, при этом узел остался всего с одним элементом. Перераспределение элементов между узлом и его правым сестринским дает обоим узлам, по крайней мере, по два ключа. Следует обратить внимание, что средний элемент J сдвинут в родительский узел.

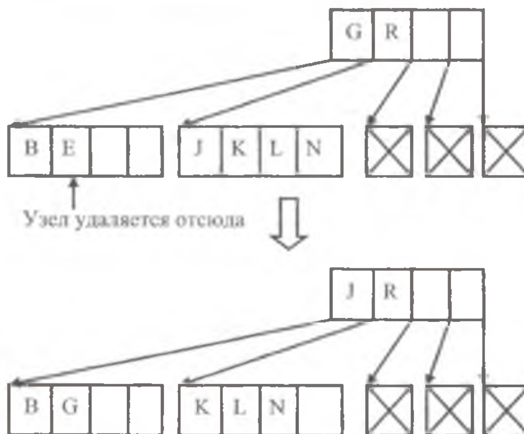


Рис. 22. Перераспределение узлов после удаления одного из них

При попытке сбалансировать дерево таким образом может оказаться, что соседний узел на том же уровне содержит всего К элементов, поэтому недостаточно использовать эти два узла. В этом случае все элементы в обоих узлах могут поместиться в пределах одного узла, поэтому их можно объединить их.

Процесс объединения двух узлов называется **слиянием сегментов**. На рис. 23 показан пример такого слияния.

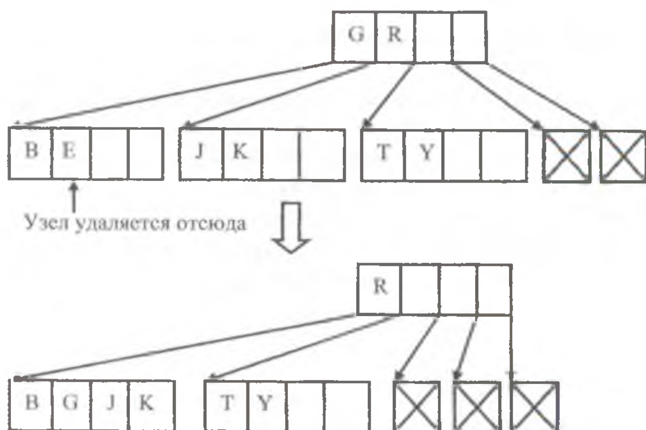


Рис. 23. Процесс объединения сегментов

При слиянии двух узлов из родительского удаляется ключ, и там остается К-1 элементов. В этом случае необходимо перебалансировать или объединить его с одним из сестринских узлов. Но если после этого в узле на более высоком уровне все равно останется К-1 элементов, процесс повторится. В наихудшем случае удаление вызовет цепную реакцию слияния сегментов узлов вплоть до корня.

Нисходящие Б-деревья

Данный вид Б-деревьев основан на немного отличном алгоритме добавления узлов. Так, в случае добавления узла в дерево происходит автоматическое разбиение всех полных узлов, которые встречаются на пути сверху вниз при поиске необходимого места для вставки. Так как все расположенные выше полные узлы уже разбиты, то в родительском узле всегда есть место для нового элемента.

Когда процедура достигает листа, в который нужно поместить элемент, в родительском узле обязательно будет место для размещения нового элемента. Если программа должна разбить лист, то всегда есть место для размещения среднего элемента листа в родительском узле. Поскольку эта система работает от вершины вниз, этот тип Б-деревьев называется **нисходящим Б-деревом** (top-down B-trees).

При этом разбиение блоков происходит ***чаще, чем необходимо***. Нисходящее Б-дерево разбивает полный узел, даже если в его дочерних узлах достаточно много свободного места. При нисходящем методе дерево содержит большее количество пустых записей, чем в случае обычных Б-

деревьев, однако тем самым сокращается риск возникновения длинного каскада разбиений сегментов.

К сожалению, **не существует нисходящей версии объединения узлов**. Процедура удаления узлов не может объединять встречающиеся полупустые узлы на пути вниз, потому что в этот момент еще неизвестно, нужно ли будет объединить два дочерних узла и удалить элемент из их родителя. Поскольку также неизвестно, будет ли удален элемент из родительского узла, нельзя заранее сказать, потребуется ли слияние родителя с одним из узлов, находящимся на том же уровне.

Б+деревья

Б-деревья часто используются для хранения больших записей. Типичное Б-дерево может содержать записи о сотрудниках, каждая из которых занимает несколько килобайт памяти. Записи упорядочиваются по некоторому ключевому полю, например по имени служащего или идентификационному номеру. В этом случае переупорядочивание элементов будет происходить медленно. Чтобы объединить два сегмента, программа должна переместить много записей, каждая из которых довольно большая. Аналогично, для разбиения блока придется обработать не меньшее число, записей большого объема.

Чтобы избежать перемещения больших блоков данных, программа записывает во внутренних узлах Б-дерева только ключи записей. Узлы также содержат указатели на фактические записи данных, сохраненные в другом месте. Теперь, если программе требуется переупорядочить блоки, нужно будет переместить только ключи и указатели, а не сами записи. Данный тип Б-дерева называется Б+деревом (B+tree).

Поскольку элементы Б+ дерева довольно малы, программа сохраняет большее количество ключей в каждом узле. При том же размере узла программа может увеличить порядок дерева и сделать его короче.

2. СОРТИРОВКА

Под сортировкой записей $R_1, R_2, ..., R_N$ будем понимать сортировку в порядке не убывания их ключей $K_1, K_2, ..., K_N$.

2.1 Внутренняя сортировка[16]

В данном разделе рассматриваются алгоритмы внутренней сортировки, когда число записей, подлежащих сортировке, достаточно мало, так что их можно полностью разместить в оперативной памяти компьютера, обладающей высокой скоростью обмена.

Сортировка путем вставок

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

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

Бинарные и двухпутевые вставки

Поиск ячеек между которыми следует вставить новую запись можно модифицировать, учитывая тот факт, что первоначальный массив уже отсортирован. Например, если вставляется 64-я запись, можно сначала сравнить ключ K_{64} с K_{32} а затем, если он меньше, сравнить его с K_{16}, а если больше - с K_{48} и т. д. Данный метод называется **методом бинарных вставок**. Однако данный метод позволяет решить задачу только наполовину: после того как найдено место, куда следует вставлять запись R_j все равно

[16] Составлено на основе материалов монографий [5,6]

необходимо осуществить сдвиг примерно j/2 ранее рассортированных записей, чтобы освободить место для R_j.

Таким образом, в обоих описанных методах самой продолжительной по времени является операция сдвига элементов в массиве. Метод двухпутевых вставок, впервые предложенный в начале 50-х годов прошлого века позволяет несколько сократить ее продолжительность. Иллюстрация метода представлена на рисунке 24. Здесь первый элемент помещается в середину области вывода и место для последующих элементов освобождается при помощи сдвигов влево или вправо, туда, куда удобнее. Таким образом удается сэкономить примерно половину времени работы по сравнению с использованием метода простых вставок за счет некоторого усложнения программы.

```
                ^503
             087 503^
            ^087 503 512
         061 087 503 512^
         061 087^503 512 908
     061 087 170 503 512^908
     061 087 170^503 512 897 908
 061 087 170 275 503 512 897 908
```

Рис. 24. Процесс сортировки методом двухпутевых вставок

Метод Шелла

Для алгоритмов сортировки, который каждый раз перемещает запись только на одну позицию, среднее время выполнения будет в лучшем случае пропорционально N^2, потому что в процессе сортировки каждая запись должна пройти в среднем через N/3 позиций. Поэтому, если желательно получить метод, существенно превосходящий по скорости метод простых вставок, необходим механизм, с помощью которого записи могли бы перемещаться сразу на несколько позиций.

Один из таких методов был предложен в 1959 году Дональдом Л. Шеллом и известен во всем мире под именем своего автора. На рис. 25 проиллюстрирована общая идея, которая лежит в его основе для массива из 16 записей. Сначала они делятся на 8 групп по две записи в каждой: (R_1,R_9), (R_2,R_{10}), … , (R_8, R_{16}). Каждая из этих групп сортируется по возрастанию. В результате сортировки каждой группы записей по отдельности приходим ко второй строке рис. 25. Этот процесс называется первым проходом. Далее записи делятся на четыре группы по четыре в каждой: (R_1,R_5,R_9,R_{13}), (R_4, R_8,R_{12},R_{16}). Затем опять сортируется каждая группа в отдельности. Результат этого второго прохода показан в третьей строке.

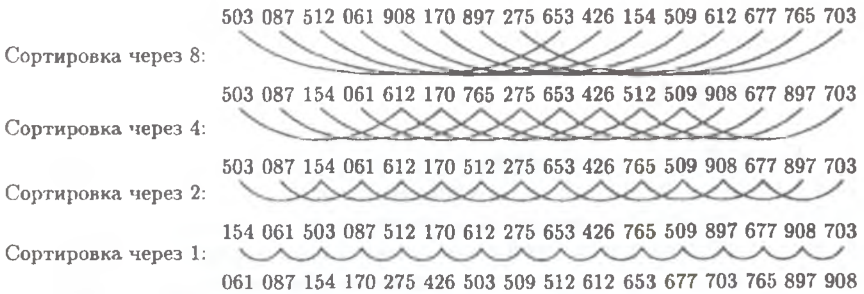


Рис. 25. Сортировка методом Шелла со смещениями 8, 4, 2, 1

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

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

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

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

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

которая обрывается на h_{t-1}, если $3h_t >= N$

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

Обменная сортировка

Для всех методов обменных сортировок характерно наличие обменов или транспозиций, предусматривающих систематический обмен местами

между элементами пар, в которых нарушается упорядоченность, до тех пор, пока таких пар не останется.

Метод пузырька

Пожалуй, наиболее очевидный способ обменной сортировки заключается в том, чтобы сравнить K_1 с K_2, меняя местами R_1 и R_2, если их ключи расположены не в нужном порядке, и затем проделать то же самое с R_2 и R_3, R_3 и R_4 и т. д. При выполнении этой последовательности операций записи с большими ключами будут продвигаться вправо; и действительно, все это закончится тем, что запись с наибольшим ключом займет положение R_N При многократном выполнении данного процесса соответствующие записи попадут в позиции R_{N-1}, R_{N-2} и т. д., так что, в конце концов, все записи будут упорядочены. Последовательность чисел удобно представлять не горизонтально, а вертикально, чтобы запись R_N была в самом верху, а R_1 - в самом низу. Метод назван "методом пузырька", потому что большие элементы, подобно пузырькам, "всплывают" на соответствующую позицию.

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

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

Однако несмотря на свою простоту, он поддается дальнейшему улучшению.

Первое улучшение алгоритма заключается в запоминании того, производился ли на данном проходе какой-либо обмен. Если нет - алгоритм заканчивает работу. Оптимизацию можно продолжить, если запоминать не только сам факт обмена, но и **индекс последнего обмена k**. Дальнейшие проходы можно заканчивать на индексе k, вместо того чтобы двигаться до установленной заранее верхней границы.

При сортировке методом пузырька можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют "***шейкер-сортировкой***". На практике метод пузырька, даже с улучшениями, работает, увы, слишком медленно. А потому – почти не применяется.

Параллельная сортировка Бэтчера

Чтобы получить алгоритм обменной сортировки, время работы которого имеет порядок, меньший $O(n^2)$, необходимо подобрать для сравнений

пары несоседних ключей (K_i, K_j); иначе придется выполнить столько операций обмена записей, сколько инверсий имеется в исходной перестановке. В 1964 году К. Э. Бэтчер открыл интересный способ программирования последовательности сравнсний, предназначенной для поиска возможных обменов. Его метод далеко не очевиден.

Схема сортировки Бэтчера несколько напоминает сортировку Шелла, но сравнения выполняются по-новому, а потому цепочки операций обмена записей не возникает. Бэтчера действует, как 8-, 4-, 2- и 1-сортировка, но сравнения не перекрываются. Поскольку в алгоритме Бэтчера, по существу, происходит слияние пар рассортированных подпоследовательностей, его можно назвать обменной сортировкой со слиянием. Схема метода представлена на рис. 26.

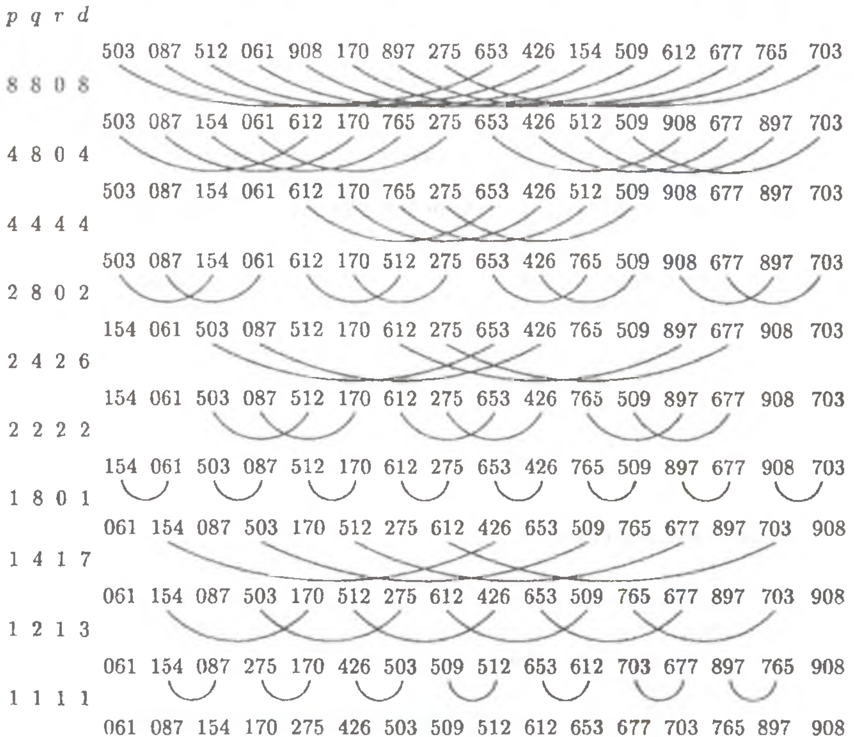
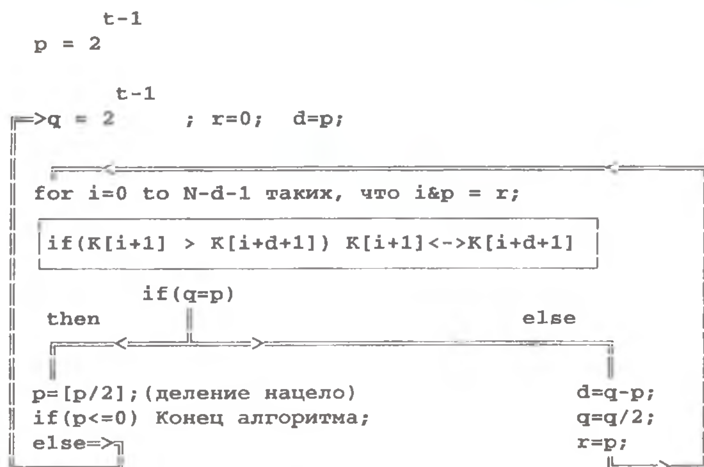


Рис. 26. Обменная сортировка со слиянием (метод Бэтчера)

Ниже представлена схема алгоритма. Здесь t=[logN] (целая часть числа). Переменная d имеет смысл шага сортировки, то есть сравниваются

ключи K[i] и K[i+d]. Символом "<->" обозначена операция обмена значений двух переменных.

К сожалению, количество вспомогательных операций, необходимых для управления последовательностью сравнений в методе Бэтчера, весьма велико, так что программа в итоге становится менее эффективна, чем многие другие рассмотренные выше методы. Однако она обладает одним важным компенсирующим качеством: все сравнения и/или обмены можно выполнять одновременно на компьютерах или в сетях, которые реализуют параллельные вычисления. С такими параллельными операциями сортировка осуществляется за [lgN]([lgN] + 1)/2 шагов, и это один из самых быстрых общих методов среди всех известных. Например, 1024 элемента можно рассортировать методом Бэтчера всего за 55 параллельных шагов.



Алгоритм сортировки Бетчера

Быстрая сортировка

Алгоритм быстрой сортировки был разработан более 40 лет назад, однако в настоящее время является, пожалуй, наиболее широко применяемым и одним их самых эффективных алгоритмов.

Метод основан на подходе "разделяй-и-властвуй". Общая схема такова:

1. из массива выбирается некоторый опорный элемент a[i],
2. запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], влево от него, а все ключи, боль-

шие, либо равные a[i] - вправо. В результате выполнения данной операции массив будет состоять из двух подмножеств, причем левое меньше либо равно правого:

≤ a[i]	a[i]	≥ a[i]

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

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

Модификации метода быстрой сортировки

1. Для сортировки малых подмассивов рекомендуется использовать другие методы сортировки. Увеличение скорости может составлять до 15%.

2. Если при реализации отказаться от рекурсии, то можно значительно сократить размер требуемой памяти.

3. Производительность метода будет выше, если массив будет делиться на равные части. Потому в качестве опорного элемента целесообразно брать средний из трех, а если массив достаточно велик - то из девяти произвольных элементов.

Сортировка посредством выбора

Еще одно важное семейство методов сортировки основано на идее многократного выбора. Вероятно, простейшая сортировка посредством выбора сводится к следующему.

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

Данный метод называется **сортировка путем простого выбора**. Заметим, что этот метод требует наличия всех исходных элементов до начала сортировки, а элементы вывода порождает последовательно, один за дру-

гим. Картина по существу, противоположна картине, возникающей при использовании метода вставок, в котором исходные записи поступают последовательно, но вплоть до завершения сортировки об окончательном результате ничего неизвестно.

Пирамидальная сортировка[17]

Назовем **пирамидой** (heap) бинарное дерево высоты k, в котором

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

Ниже на рисунке 27 показано дерево, удовлетворяющее свойству пирамиды

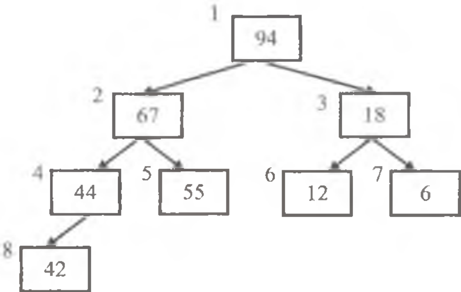


Рис. 27. Пример дерева, удовлетворяющего свойству пирамиды

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

Фаза 1: построение пирамиды

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

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

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

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

Фаза 2: сортировка

1. Необходимо взять верхний элемент пирамиды a[0]...a[n] (первый в массиве) и поменять с последним местами. Далее этот элемент из рассмотрения необходимо исключить, так как он уже занял нужное место в массиве. Рассматривается массив a[0]...a[n-1]. Для превращения его в пирамиду достаточно просеять лишь новый первый элемент описанным выше способом.
2. Необходимо повторять шаг 1, пока обрабатываемая часть массива не уменьшится до одного элемента.

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

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

Сортировка методом слияния[18]

Слияние означает объединение двух или более упорядоченных массивов в один упорядоченный массив. Можно, например, слить подмассивы 503 703 765 и 087 512 677 и получить 087 503 512 677 703 765. Простой способ сделать это - сравнить два наименьших элемента, вывести наименьший из них и повторить эту процедуру.
Начав с

$$\begin{cases} 503\ 703\ 765 \\ 087\ 512\ 677 \end{cases},$$

получим

$$087\ \begin{cases} 503\ 703\ 765 \\ 512\ 677 \end{cases},$$

затем

$$087\ 503\ \begin{cases} 703\ 765 \\ 512\ 677 \end{cases}$$

и

$$087\ 503\ 512\ \begin{cases} 703\ 765 \\ 677 \end{cases}$$

и т. д. пока не исчерпается хотя бы один подмассив. Если это произойдет, то в результирующий массив необходимо добавить все оставшиеся члены другого подмассива.

[18] Составлено на основе материалов монографии [5]

Задачу сортировки можно свести к слиянию, сливая все более длинные подмассивы до тех пор, пока не будет рассортирован весь массив. Такой подход можно рассматривать как развитие идеи сортировок методом вставок: вставка нового элемента в упорядоченный массив – это частный случай слияния при n=1. Чтобы ускорить процесс вставок, можно рассмотреть вставку нескольких элементов за один раз, группируя несколько операций, а это естественным образом приведет к общей идее сортировки методом слияния.

На рисунке 28 проиллюстрирована сортировка методом слияния, когда возрастающие серии выделяются как с начала массива, так и с его конца.

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

503 703 765 061 612 908 154 275 426 653 897 509 170 677 512 087

087 503 512 677 703 765 154 275 426 653 908 897 612 509 170 061

061 087 170 503 509 512 612 677 703 765 897 908 653 426 275 154

061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

Рис. 28. Сортировка методом естественного двухпутевого слияния

Вертикальными линиями на рисунке 28 отмечены границы между сериями. Сложность данного алгоритма составляет O(n log n) операций.

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

Сортировка методом распределения

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

Поразрядная сортировка

Отличительными особенностями данного способа сортировки являются:

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

До сортировки необходимо знать два параметра: k и m, где

k - количество разрядов в самом длинном ключе
m - разрядность данных: количество возможных значений разряда ключа

Эти параметры нельзя изменять в процессе работы алгоритма.

Поразрядная сортировка для списков

Предполагается, что элементы линейного списка L есть k-разрядные десятичные числа, разрядность максимального числа известна заранее. Через d(j,n) обозначается j-я справа цифра числа n

Пусть L_0, L_1,..., L_9 - вспомогательные списки (карманы), которые вначале пусты. Поразрядная сортировка состоит из двух процессов, называемых **распределение** и **сборка** и выполняемых для j=1,2,...,k.

Фаза распределения разносит элементы L по карманам: элементы l_i списка L последовательно добавляются в списки L_m, где $m = d(j, l_i)$. Таким образом получается десять списков, в каждом из которых j-тые разряды чисел одинаковы и равны m.

Фаза сборки состоит в объединении списков L_0, L_1,..., L_9 в общий список $L = L_0 \Rightarrow L_1 \Rightarrow L_2 \Rightarrow \ldots \Rightarrow L_9$

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

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

2.2 Внешняя сортировка[19]

Будем понимать под **"внешней" сортировкой** сортировку последовательных файлов, располагающихся во внешней памяти и слишком больших, чтобы их можно было целиком переместить в основную память и применить один из рассмотренных выше методов внутренней сортировки. Наиболее часто внешняя сортировка применяется в системах управления базами данных. Практически все методы внешней сортировки, излагаемые в настоящем пособии, основаны на слиянии.

Прямое слияние

Предположим, что имеется последовательный файл А, состоящий из записей $a_1, a_2, ..., a_n$ (для простоты предполагается, что n представляет собой степень числа 2). Будем считать, что каждая запись состоит ровно из одного элемента, представляющего собой ключ сортировки. Для сортировки используются два вспомогательных файла В и С, размер каждого из них будет равен n/2.

Сортировка состоит из последовательности шагов, в каждом из которых выполняется распределение содержимого файла А в файлы В и С, а затем слияние файлов В и С в файл А. На первом шаге для распределения последовательно читается файл А, и нечетные записи пишутся в файл В, а четные - в файл С (начальное распределение). Начальное слияние производится над парами (a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n), и результат записывается в файл А. На втором шаге снова последовательно читается файл А, и в файл В записываются последовательные пары с нечетными номерами, а в файл С – с четными. При слиянии образуются и пишутся в файл А упорядоченные четверки записей. Перед выполнением последнего шага файл А будет содержать две упорядоченные подпоследовательности размером n/2 каждая. При распределении первая из них попадет в файл В, а вторая - в файл С. После слияния файл А будет содержать полностью упорядоченную последовательность записей. Заметим, что для выполнения внешней сортировки методом прямого слияния в основной памяти требуется расположить всего лишь две переменные - для размещения очередных записей из файлов В и С. Файлы А, В и С будут O(log n) раз прочитаны и столько же раз записаны.

Естественное слияние

При использовании метода прямого слияния не принимается во внимание то, что исходный файл может быть частично отсортированным, т.е. содержать упорядоченные подпоследовательности записей. **Серией** называется подпоследовательность записей $a_i, a_{i+1}, ..., a_j$ такая, что $a_k \le a_{k+1}$ для

---

[19] Составлено с использованием материалов сайта http://www.citforum.ru

всех $i \le k \le j$, $a_i < a_{i-1}$ и $a_j > a_{j+1}$. Метод естественного слияния основывается на распознавании серий при распределении и их использовании при последующем слиянии.

Как и в случае прямого слияния, сортировка выполняется за несколько шагов, в каждом из которых сначала выполняется распределение файла А по файлам В и С, а потом слияние В и С в файл А. При распределении распознается первая серия записей и переписывается в файл В, вторая - в файл С и т.д. При слиянии первая серия записей файла В сливается с первой серией файла С, вторая серия В со второй серией С и т.д. Если просмотр одного файла заканчивается раньше, чем просмотр другого (по причине разного числа серий), то остаток файла целиком копируется в конец файла А. Процесс завершается, когда в файле А остается только одна серия.

Сбалансированное многопутевое слияние

В основе метода внешней сортировки сбалансированным многопутевым слиянием является распределение серий исходного файла по m вспомогательным файлам $B_1, B_2, ..., B_m$ и их слияние в m вспомогательных файлов $C_1, C_2, ..., C_m$. На следующем шаге производится слияние файлов $C_1, C_2, ..., C_m$ в файлы $B_1, B_2, ..., B_m$ и т.д., пока в B_1 или C_1 не образуется одна серия. Данный метод является естественным развитием идеи обычного двухпутевого слияния.

3. ПОИСК

Предполагается, что имеется набор из N записей и задача состоит в нахождении одной из них. Также полагается, что каждая запись включает специальное поле, именуемое ее ключом. В общем случае требуется N различных ключей для того, чтобы каждый из них однозначно идентифицировал связанную с ним запись. Набор всех записей именуется таблицей или файлом. Алгоритм поиска имеет так называемый аргумент K, и задача заключается в нахождении записи, для которой K служит ключом. Результатом поиска может быть одно из двух: либо поиск завершился успешно, и уникальная запись, содержащая K, найдена, либо поиск оказался неудачным, и запись с ключом K не найдена.

3.1 Последовательный поиск [20]

Принцип данного способа поиска достаточно прост и прозрачен - "начать с начала и продолжать, пока не будет найден искомый ключ; затем

[20] Составлено на основе материалов монографии [5]

остановиться". Эта последовательная процедура представляет собой очевидный путь поиска и может служить отличной отправной точкой для рассмотрения множества алгоритмов поиска

Ниже приводится данный способ поиска, реализованный на языке Pascal.

Дан массив записей R_1, R_2, ... , R_n с ключами K_1, K_2,..., K_n соответственно. Алгоритм предназначен для поиска записи с заданным ключом Value. Предполагается, что $N \geq 1$.

Последовательный поиск

```
procedure ConsistentFind(Value:integer);
 var i:integer;
    MustContinue:boolean;
begin
 i:=1;
 MustContinue:=true;
 while (i<=N) and MustContinue do
 begin
   MustContinue:=K[i]<>Value;
   inc(i);
 end;
 if not MustContinue then writeln('Запись найдена');
                         else writeln('Поиск неудачен')
end;
```

Данный алгоритм в той или иной вариации, несомненно, знаком всем программистам, но лишь некоторые из них знают, что этот способ - не самая лучшая реализация последовательного поиска! Необходимо произвести небольшие изменения и алгоритм будет выполняться существенно быстрее при условии, что записей в исходном массиве достаточно много.

Ниже представлен тот же алгоритм, что и предыдущий, однако в нем имеется дополнительное предположение о наличие фиктивной записи R_{n+1} в конце массива.

Быстрый последовательный поиск

```
procedure Consistent2Find(Value:integer);
 var i:integer;
begin
 i:=1;
 K[N+1]:=Value;
 while true do
 begin
```

```
  if K[i]=Value then
  if i<=N then
  begin
    writeln('Запись найдена');
    exit;
  end else
  begin
    writeln('Поиск неудачен');
    exit;
  end;
  inc(i);
 end;
end;
```

Как видно из приведенного кода скорость повышается благодаря тому, что второе условие (i<=N) будет проверяться не на каждой итерации, а только после того, как требуемый элемент будет найден. Ниже представлен способ сделать данный алгоритм еще быстрее:

Быстрый последовательный поиск, вариант 2

```
procedure Consistent3Find(Value:integer);
 var i:integer;
begin
 i:=1;
 K[N+1]:=Value;
 Q3:
 inc(i,2);
 if K[i]=Value then goto Q4;
 if K[i+1]<>Value then goto Q3;
 inc(i)
 Q4:
 if i<=N then writeln('Запись найдена') else writeln('Поиск неудачен');
end;
```

Внутренний цикл данного алгоритма дублирован, что позволяет избежать выполнения половины инструкций i:=i+1, тем самым снижая затраты времени. При работе с большими таблицами это сохраняет до 30% машинного времени. Одно из упражнений, приведенных в конце данного пособия, посвящено решению этой задачи.

3.2 Поиск путем сравнения ключей

В данном разделе обсуждаются методы поиска, основанные на линейном упорядочении ключей. После сравнения данного аргумента K с ключом K_i из таблицы поиск продолжается одним из трех путей, в зависимости от того, какое из условий $K < K_i$, $K = K_i$ или $K > K_i$ будет выполняться. По-

следовательные методы поиска, рассмотренные ранее, по сути, имели только два варианта ветвления поиска - в зависимости от выполнения или не выполнения условия $K = K_i$.

Поиск в упорядоченной таблице

Бинарный поиск

Пожалуй, самым простым способом поиска ключа в упорядоченном массиве является следующий: сравнить К со средним ключом в таблице, в результате этого сравнения определить, в какой половине таблицы находится искомый ключ, и снова применить ту же процедуру к половине таблицы. Таким образом, максимум за величину порядка $\log_2 N$ сравнений искомый ключ будет либо найден, либо будет установлено, что его нет в таблице. Эта процедура известна как "логарифмический поиск" или "метод деления пополам" но чаще всего употребляется термин бинарный поиск.

Поиск Фибоначчи

Данный алгоритм поиска представляет собой альтернативу бинарному поиску. Для некоторых компьютеров предлагаемый метод предпочтителен в связи с тем, что в нем используются только сложение и вычитание, в отличие от бинарного поиска, где применяется деление. На рис. 29 показано дерево поиска Фибоначчи порядка 6.

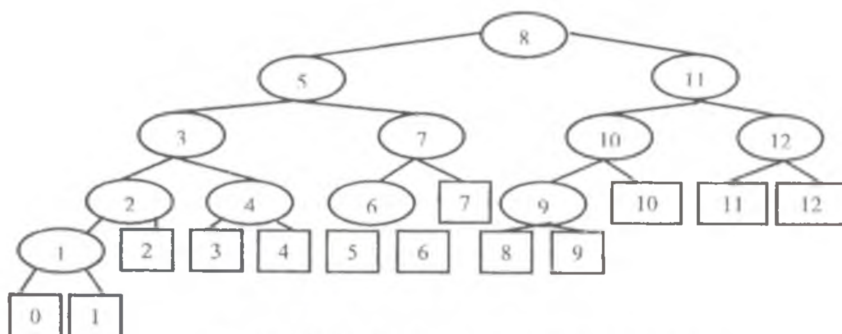


Рис. 29. Дерево Фибоначчи порядка 6

Последовательность чисел Фибоначчи F_k описывается следующей зависимостью:

$$F_0=0,\ F_1=1,\ F_{k+2}= F_{k+1}+ F_k,\ k=0,1,\ldots$$

В целом, дерево Фибоначчи порядка k имеет F_{k+1}-1 внутренних и F_{k+1} внешних узлов. Оно строится по следующим правилам:

- Если k = 0 или k = 1, то дерево вырождается в [0].
- Если k ≥ 2, корнем является F_k; левое поддерево представляет собой дерево Фибоначчи порядка k - 1; а правое поддерево представляет собой дерево Фибоначчи порядка k - 2 с числами, увеличенными на F_k.

Комбинируя эти наблюдения с механизмом распознавания внешних узлов, получается алгоритм поиска Фибоначчи.

3.3 Хеширование[21]

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

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

Хеш-функции

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

$$0 \le h(K) < M.$$

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

[21] Составлено на основе материалов монографии [5]

Метод деления весьма прост. В качестве хеш-функции берется остаток от деления ключа К на М:

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

Как показала практика, лучше всего в качестве М использовать простые числа.

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

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

Одна из положительных сторон мультипликативной схемы заключается в отсутствии потери информации, т.е. в возможности восстановить значение ключа К по значению функции h(K). Дело в том, что А и w взаимно просты и при помощи алгоритма Евклида можно найти константу А', такую, что

А А' mod w = 1; отсюда следует, что К = (А'(АК mod w)) mod w.

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

Хеширование ключей, состоящих из нескольких слов

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

Разрешение коллизий методом "цепочек".

Как уже упоминалось выше, что некоторые хеш-адреса могут быть одинаковыми для различных ключей. Вероятно, самый очевидный путь решения этой проблемы - поддержка M связных списков, по одному для каждого возможного значения хеш-кода. После хеширования ключа можно выполнить последовательный поиск в списке с номером номер h(K)+1. На рисунке 30 приводится иллюстрация метода разрешения коллизий методом цепочек.

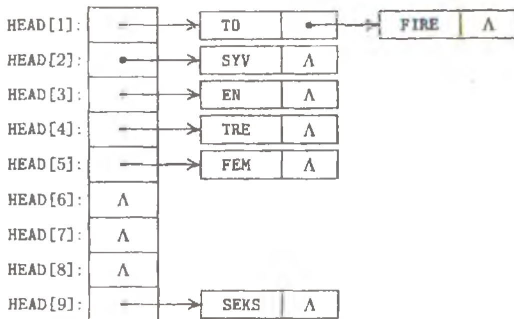


Рис. 30. Разрешение коллизий методом цепочек

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

Разрешение коллизий методом открытой адресации.

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

деленной этим ключом последовательности встречается пустая позиция, можно сделать вывод о том, что K в таблице отсутствует. Этот общий класс методов назван **открытой адресацией**.

Простейшая схема открытой адресации, известная как **линейное исследование** использует циклическую последовательность проверок
h(K), h(K) - 1, ..., 0, M - 1, M - 2, ..., h(K) + 1
Ниже на рисунке 31 приведен пример распределения ключей в хеш таблицы, если исходными данными были ключи:K=EN, TO, TRE, FIRE, FEM, SEEK, SYV с хеш кодами 2, 7, 1, 8, 2, 8, 1

0	FEM
1	TRE
2	EN
3	
4	
5	SYV
6	SEEK
7	TO
8	FIRE

Рис. 31. Линейная хеш-адресация

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

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

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

где α=N/M – коэффициент заполненности таблицы

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

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

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

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

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

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

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

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

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

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

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

4. СЕТЕВЫЕ АЛГОРИТМЫ[22]

Сеть, или **граф** - это набор узлов, связанных ребрами, или дугами. В отличие от дерева, в сети нет предков и потомков. Узлы, соединенные с другими узлами, являются скорее соседями, чем родительскими или дочерними узлами.

Каждое звено в сети может иметь соответствующее направление. В этом случае сеть называется **направленной сетью**. Каждая дуга может также иметь соответствующую **стоимость**. В сети улиц, например, стоимость равна времени, которое требуется, чтобы проехать по участку дороги, представленному дугой сети. На рис. 32 показана небольшая направленная сеть, в которой числа рядом с дугами соответствуют их стоимости.

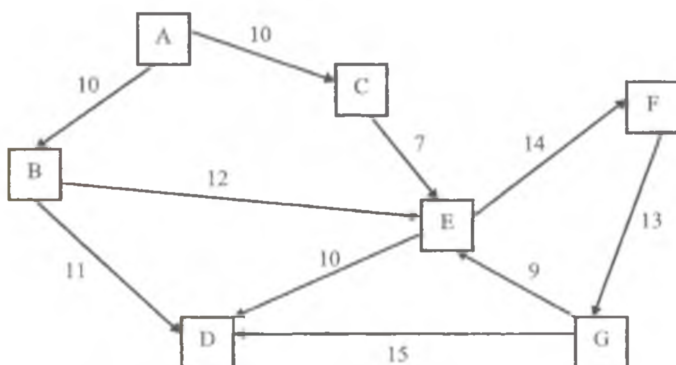


Рис. 32. Направленная сеть со стоимостью связей

Путь между узлами A и B - это последовательность дуг, которые соединяют эти узлы. Если между любыми двумя узлами сети есть не больше одного ребра, то путь можно описать, перечислив входящие в него узлы. Поскольку такое описание проще представить наглядно, пути по возможности описываются именно так. На рис. 32 путь, содержащий узлы B, E, F, G, E, и D, соединяет узлы B и D.

Цикл - это путь, который соединяет узел с самим собой. Путь E, F, G, E на рис. 32 является циклом.

Путь называется **простым**, если он не содержит циклов. Путь B, E, F, G, E, D не является простым, потому что он содержит цикл E, F, G, E.

Если между двумя узлами существует какой-либо путь, то должен существовать и простой путь между ними. Его можно найти, удалив все циклы из первоначального пути. Например, если заменить цикл E, F, G, E

---

[22] Составлено на основе материалов монографии [6]

узлом E в пути B, E, F, G, E, D, получится простой путь B, E, D между узлами B и D.

Сеть называется **связанной**, если между любыми двумя узлами сети есть хотя бы один путь. В направленной сети не всегда очевидно, существует такая связь или нет. На рис. 33 сеть слева связана. Сеть справа не является таковой, потому что от узла E к узлу C нет ни одного пути.

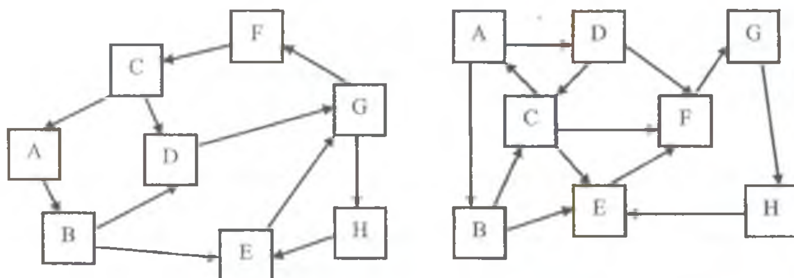


Рис. 33. Пример связанной и несвязанной сети

4.1 Представления сетей

Для представления сетей можно использовать большинство представлений, которые использовались для деревьев. Например, для сохранения сетей могут использоваться такие представления, как метод полных узлов, списки дочерних узлов (для сетей список соседних узлов) и представление нумерацией связей.

Для разных приложений лучше подходят разные представления сети. Представление полными узлами приводит к хорошим результатам, если каждый узел сети связан с ограниченным числом ребер. Список соседних узлов обеспечивает большую гибкость, чем метод полных узлов. Представление с помощью нумерации связей, хотя его сложнее изменять, требует меньше памяти для сохранения сети. Кроме того, несколько вариантов представления ребер могут упростить управление определенными типами сетей. Эти форматы используют один класс для представления узлов и другой - для представления связей. Применение класса для связей облегчает работу со свойствами ребра, такими как стоимость.

Например, направленная сеть со стоимостью ребер может использовать следующее определение для класса узла и дуги. Каждое ребро хранит указатели на узлы, где оно начинается и заканчивается. Каждый узел хранит связанный список ребер, исходящих из него. Узел также имеет указатель NextNode, и программа может сохранять все узлы сети в связанном списке.

```
type
  PLink=^TLink;
  PNode=^TNode;
  TLink=record
    ToNode:PNode;    // Конечный узел звена
    Cost:integer;    // Стоимость
    NextLink:PLink;  // Следующее звено в списке звеньев начального узла
  end;

  TNode=record
    ID:integer;           // Идентификатор узла
    LinkSentinel:TLink; // Звенья, исходящие из данного узла
    NextNode:PNode;  // Следующий узел в списке узлов
  end;
```

Классы узла и дуги часто расширяют для удобства работы с конкретными алгоритмами. Например, к классу узла часто добавляется логическое поле Marked. Когда программа обращается к узлу, она устанавливает Marked в True, чтобы впоследствии легко можно было его отыскать.

Для ненаправленной сети используется несколько другое представление. Класс узла остается таким же, а вот класс дуги включает указатели на оба соединяемых узла, которые на каждом конце ребра могут указывать на одну и ту же его структуру.

```
type
  TLink=record
    Node1: PNode;     // Узел на одном конце дуги
    Node2: PNode;     // Узел на другом конце
    Cost:integer;     // Стоимость
    NextLink:PLink;   // Следующее звено в списке звеньев начального узла
  end;
```

Управление узлами и связями

Корень дерева уникален, это единственный узел в дереве, который не имеет родителя. Начав от корневого узла и следуя дочерним указателям, можно найти все остальные узлы дерева. Это делает корень удобным дескриптором дерева. Если вы сохраните указатель на вершину дерева, то сможете потом обратиться ко всем его узлам.

Сети не всегда содержат узел, который обладает такими уникальными свойствами. В графах может и не быть способа обойти все узлы по связям, начав с одной точки. По этой причине сетевые программы часто включают в себя полный список всех узлов сети, а также могут хранить список всех ребер. Данные списки существенно упрощают работу со всеми связями и узлами сети.

4.2 Обход сети

Обход сети подобен обходу дерева. Для этого можно использовать обход либо в глубину, либо в ширину.

Обход в ширину обычно похож на прямой обход деревьев, хотя для сети можно также определить также обратный и симметричный обход.

Как известно, алгоритм прямого обхода двоичного дерева формулируется следующим образом:

1. Обратиться к узлу.
2. Выполнить рекурсивный прямой обход левого поддерева.
3. Выполнить рекурсивный прямой обход правого поддерева.

В дереве между связанными узлами существует отношение "родительский-дочерний". Поскольку алгоритм начинает с корня и всегда движется вниз через дочерние узлы, он никогда не обратится к узлу дважды.

В сети узлы не обязательно соединены сверху вниз. Если попытаться реализовать в сети алгоритм прямого обхода, то возможно возникновение бесконечного цикла. Чтобы предотвратить это, алгоритм должен помечать посещаемые узлы. При поиске в соседних узлах обращение происходит только к тем узлам, которые еще не были помечены. Когда алгоритм заканчивается, все узлы в сети будут помечены как посещенные (если сеть связана).

Таким образом, алгоритм прямого обхода сети выполняется в следующем порядке:

1. Пометить узел.
2. Посетить узел.
3. Выполнить рекурсивный обход непомеченных соседних узлов.

Для хранения информации о посещенных узлах можно добавить логическую переменную Visited к записи TNode:

```
TNode=record
  ID:integer;            // Идентификатор узла
  LinkSentinel:TLink; // Звенья, исходящие из данного узла
  NextNode:PNode;  // Следующий узел в списке узлов
  Visited:boolean;     // Был ли узел уже посещен при обходе
end;
```

Следующий код демонстрирует, как процедура может обойти все непомеченные узлы, начиная с данного. В связанной сети алгоритм обратится к каждому узлу.

```
procedure Traverse(node : PNode);
  var
    link : PLink;
    neighbor : PNode;
begin
  // Помечаем узел как посещенный.
  Node^.Visited := True;
  // Посещение непомеченных соседних узлов.
  link := node^.LinkSentinel.NextLink;
  while (link<>nil) do
  begin
    // Какой узел является соседним для данного.
    if (link^.Node1 = node) then neighbor := link^.Node2
                                 else neighbor := link^.Nodel;
    // Посещаем соседний узел, если он еще не помечен
    if (not neighbor^.Visited) then Traverse(neighbor);
    // Исследуем следующее ребро
    Link:=link^.NextLink;
  end;
end;
```

Поскольку эта процедура не обращается дважды ни к одному узлу, набор обходимых связей не содержит циклов и образует дерево. В связанной сети дерево будет обходить каждый узел. Поскольку дерево охватывает каждый узел сети, оно названо **остовным деревом**, или каркасом. На рис. 34 показана небольшая сеть. Каркас ее дерева с корнем в узле А изображен жирными линиями.

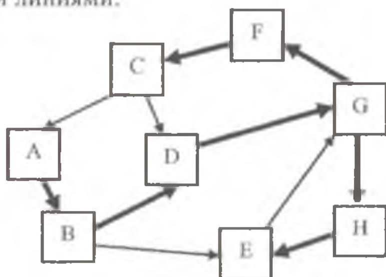


Рис. 34. Каркас дерева

Методику пометки узлов можно также использовать для того, чтобы преобразовать алгоритм обхода дерева в ширину в сетевой алгоритм. Алгоритм обхода дерева начинает работу, помещая корневой узел дерева в очередь. Затем первый узел из очереди удаляется, происходит обращение к узлу, и его дочерние узлы помещаются в конце очереди. Этот процесс повторяется до тех пор, пока очередь не опустеет.

Прежде чем выполнять обход сети, необходимо убедиться, что узел не проверялся раньше или уже не находится в очереди. Чтобы удостовериться в этом, необходимо помечать каждый узел, который помещается в очередь, как в предыдущем случае.

Ниже приводится алгоритм обхода сети в ширину:

1. Пометить первый узел (это будет корень остовного дерева) и добавить его в конец очереди.
2. Повторять следующие шаги, пока очередь не опустеет:
   - удалить первый узел из очереди и обратится к нему;
   - пометить каждый из непомеченных соседних узлов и добавить его в конец очереди.

4.3 Наименьший каркас дерева

Если задана сеть со стоимостями ребер, то **минимальным**, или **наименьшим каркасом дерева** именуется каркас, общая стоимость всех ребер в котором минимальна. Вы можете использовать минимальный каркас, чтобы выбрать самый дешевый способ соединения всех узлов сети.

Предположим, что требуется спроектировать телефонную сеть, соединяющую шесть городов. Можно проложить магистральный кабель между каждой парой городов, но это нерентабельно. Следует соединить города связями, которые содержатся в минимальном каркасе дерева. На рис. 35 показано шесть городов, каждые два из которых соединены междугородными линиями. Наименьшее остовное дерево выделено жирными линиями.

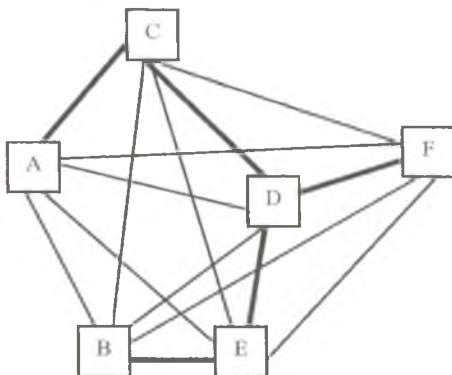


Рис. 35. Телефонные линии, соединяющие шесть городов

Следует обратить внимание на то, что сеть может содержать больше одного минимального каркаса. Для поиска минимального остовного дерева существует простой алгоритм. Сначала необходимо поместить любой

узел в остовное дерево. Затем найти связь с минимальной стоимостью, которая соединяет узел дерева с узлом, еще не помещенным в него. Данное ребро и соответствующий узел необходимо добавить к дереву. Эту процедуру следует повторять до тех пор, пока к дереву не добавятся все узлы.

4.4 Кратчайший путь

Алгоритмы поиска кратчайшего пути, рассматриваемые в данном разделе, находят все кратчайшие пути от одной точки сети до любой другой при условии что сеть связана. Набор связей, используемых всеми кратчайшими путями, образует дерево кратчайших путей.

На рис. 36 изображена сеть, в которой дерево кратчайших путей с корнем в узле A выделено жирными линиями. Оно показывает кратчайшие пути от узла A до любого другого узла сети. Например, кратчайший путь от узла A к узлу F проходит через узлы A, C, E, F

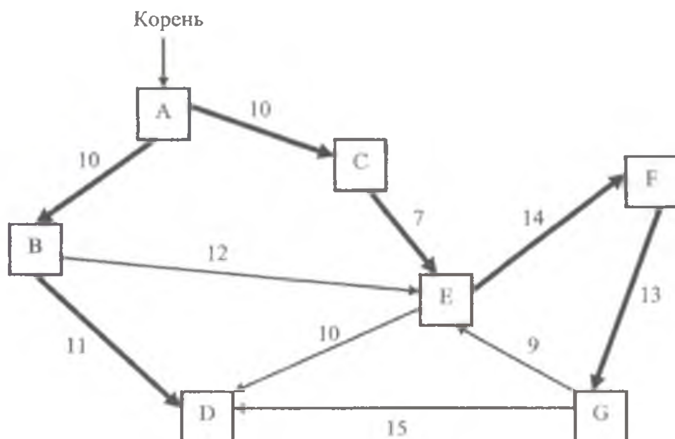


Рис. 36. Дерево кратчайших путей

Большинство алгоритмов поиска кратчайшего пути начинают с пустого дерева и затем добавляют к дереву по одному ребру то тех пор, пока дерево не будет построено. Эти алгоритмы можно разделить на две категории по способу выбора следующего ребра, которое прибавляется к дереву.

Алгоритм расстановки меток всегда выбирает ребро, которое гарантированно является частью конечного дерева кратчайших путей. Он работает аналогично алгоритму построения минимального остовного дерева. Если ребро было добавлено к дереву, оно уже не будет удалено.

Алгоритм коррекции меток добавляет ребра, которые могут быть, а могут и не быть частью конечного дерева кратчайших путей. В процессе

выполнения алгоритм может определить, что вместо уже имеющегося ребра в дерево должно быть добавлено другое. В этом случае алгоритм заменяет старое ребро новым и продолжает работу. Замена ребра в дереве может открыть дополнительные пути. Чтобы проверить их, алгоритму приходится повторно исследовать пути, которые были добавлены к дереву раньше и использовали удаленное ребро.

Алгоритмы расстановки и коррекции меток используют аналогичные классы для представления узлов и связей. Класс узла TNode содержит поле Dist, которое указывает расстояние от корня до узла в растущем дереве кратчайшего пути. В алгоритме расстановки меток для Dist задается True, как только узел добавлен к дереву и этот параметр уже не будет изменяться. В алгоритме коррекции меток параметр Dist может быть исправлен позже, когда ребро будет заменено другим.

Класс TNode также содержит поле Status, которое определяет, есть ли в настоящее время данный узел в дереве или списке возможных связей. В поле InLink указывается ребро, ведущее к узлу в растущем дереве кратчайшего пути.

```
type
 TStatus=(nsNotInList, nsWasInList, nsNowInList);

 TNode=record
  ID:integer;              // Идентификатор узла
  LinkSentinel:TLink;  // Звенья, исходящие из данного узла
  NextNode:PNode;     // Следующий узел в списке узлов
  Status:TStatus;       // Есть ли узел в дереве
  Dist:integer;           // Расстояние от корня
  InLink:PLink;          // Ребро, ведущее в данный узел
 end;
```

Класс TLink включает поле InTree, которое указывает, является ли ребро частью дерева кратчайшего пути.

```
type
 TLink=record
  Node1: PNode;      // Узел на одном конце дуги
  Node2: PNode;      // Узел на другом конце
  Cost:integer;        // Стоимость
  NextLink:PLink;    // Следующее звено в списке звеньев
  InTree:boolean;    // Есть ли ребро в дереве
 end;
```

Алгоритмы расстановки и коррекции меток используют список возможных связей, чтобы отслеживать узлы, которые могут быть добавлены к дереву кратчайшего пути, но по-разному управляют этим списком. Алгоритм расстановки меток всегда выбирает связь, которая обязательно окажется частью дерева кратчайшего пути. Алгоритм коррекции меток, описанный в этом разделе, выбирает любой узел в начале списка возможных связей.

Расстановка меток

Алгоритм расстановки меток начинает с присвоения полям Status всех узлов значения nsNotInList и полями Dist значения INFINITY, которое должно быть больше суммы стоимостей по всем дугам графа. Затем он присваивает полю Dist корневого узла значение 0 и помещает корневой узел в список возможных связей. Полю Status корня присваивается значение nsNowInList - это указывает, что корень в настоящее время находится в списке возможных связей.

Затем алгоритм выполняет поиск узла с минимальным значением Dist. Сначала будет найден корневой узел, так как он единственный в списке.

Алгоритм удаляет выбранный узел из списка и устанавливает для него значение поля Status в значение nsWasInList, поскольку теперь он является постоянной частью дерева кратчайшего пути. Поля Dist и InLink узла уже имеют правильные значения. Для каждого корневого узла значение поля InLink равно nil, а значение поля Dist- нулю.

После этого алгоритм исследует каждую связь, исходящую из выбранного узла. Если соседний узел на другом конце ребра не был в списке возможных связей, то алгоритм добавляет его к списку. Он устанавливает значение поля Status соседнего узла равным nsNowInList, а значение поля Dist - расстоянию от корневого узла до выбранного узла плюс стоимость ребра. И наконец, он присваивает полю соседнего узла InLink значение, которое указывает на связь с соседним узлом.

Если во время проверки алгоритмом связей, исходящих из выбранного узла, значение поля соседнего узла Status равно nsNowInList, то он уже занесен в список возможных. Алгоритм исследует текущее значение поля Dist соседнего узла, определяя, будет ли новый путь через выбранный узел короче. Если это так, он обновляет поля InLink и Dist соседнего узла и оставляет его в списке возможных связей. Алгоритм повторяет весь описанный процесс, удаляя узлы из списка возможных связей, исследуя соседние с ними узлы и добавляя их в список, пока он не опустеет.

На рис. 37 показана часть дерева кратчайшего пути. В этой точке алгоритм уже проверил узлы А и В, удалил их из списка возможных и исследовал их связи. Узлы А и В уже добавлены к дереву кратчайшего пути, и список возможных связей теперь содержит узлы C, D и E. Жирные стрелки на рисунке указывают значение InLink в этой точке. Например, значение поля InLink для узла E соответствует связи между узлами E и B.

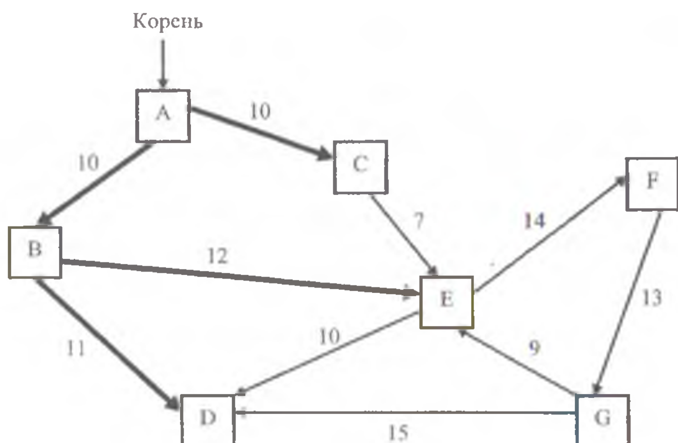


Рис. 37. Часть дерева кратчайшего пути

Важно, чтобы алгоритм обновлял значения полей InLink и Dist только для узлов со значением Status = nsNowInList. Для большинства сетей нельзя получить более короткий путь, добавляя узлы, не имеющиеся в списке возможных. Однако если сеть содержит цикл с отрицательной общей длиной, алгоритм обнаружит, что можно уменьшить расстояние до некоторых узлов, которые уже находятся в дереве. Он соединит две ветви дерева таким образом, чтобы оно перестало быть деревом. На рис. 38 показана сеть с отрицательной стоимостью цикла и «дерево» кратчайшего пути, которое получится, если алгоритм модифицирует стоимость уже имеющихся в дереве узлов.

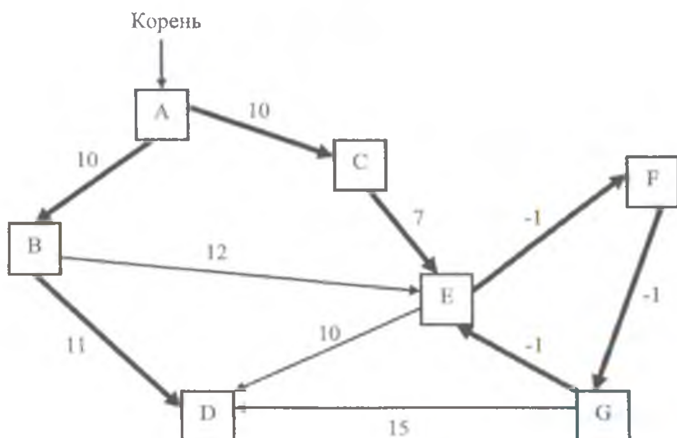


Рис. 38. Решение задачи для сети с циклом отрицательной стоимости

Вариации алгоритма расстановки меток

Основной проблемный момент в этом алгоритме - нахождение узла в списке возможных связей, который имеет минимальное значение поля Dist. Несколько вариаций этого алгоритма используют различные структуры данных для сохранения списка возможных связей, например упорядоченный связанный список. В этом случае потребуется всего один шаг, чтобы найти следующий узел, который будет добавлен к дереву кратчайших путей. Список всегда будет отсортирован, поэтому искомый узел всегда будет в начале списка.

Это облегчает поиск правильного узла в списке, но усложняет вставку узла. Вместо того чтобы просто добавлять узел в начало списка, его придется помещать в соответствующую позицию.

Иногда необходимо перераспределять узлы в списке. Если при добавлении узла к дереву уменьшилось кратчайшее расстояние до другого узла, который уже есть в списке, то нужно переместить этот элемент ближе к началу списка.

Предыдущий алгоритм и его только что описанный новый вариант представляют два крайних случая управления списком возможных связей. Первый алгоритм совершенно не упорядочивает список и тратит много времени на поиск узла в сети. Второй выполняет множество операций для поддержания упорядоченного связанного списка, на что тратится значительная часть времени, но это окупается возможностью очень быстро выбирать узлы. Другие варианты алгоритма используют промежуточную стратегию.

Например, можно хранить список кандидатов в очереди с приоритетом на основе пирамиды. В этом случае программа будет просто выбирать следующий узел из вершины пирамиды. Добавление элемента в пирамиду и ее пересортировка будут выполняться быстрее, чем такие же операции над упорядоченным связанным списком. Другие стратегии используют блочное расположение, чтобы упростить
поиск возможных узлов.

Некоторые из этих вариантов достаточно сложны. Часто для небольших сетей данные алгоритмы выполняются медленнее, чем более простые алгоритмы. Но для очень большой сети или для сети, в которой каждый узел имеет огромное число связей, выигрыш во времени от использования этих алгоритмов может стоить дополнительного усложнения алгоритма.

Коррекция меток

Как и алгоритм расстановки меток, метод коррекции меток начинает работать, присваивая полю Dist корневого узла нулевое значение и поме-

щая корневой узел в список возможных. Значения Dist для других узлов устанавливается в бесконечность. Затем из списка возможных узлов выбирается первый узел и добавляется к дереву кратчайшего пути.

После этого алгоритм исследует все соседние узлы, сравнивая расстояние от корня до выбранного узла плюс стоимость связи с текущим значением Dist соседнего узла. Если это расстояние меньше Dist, то алгоритм обновляет значения Dist и InLink соседнего узла таким образом, чтобы кратчайший путь к соседнему узлу проходил через выбранный узел. Если соседнего узла в настоящее время нет в списке возможных, то он также добавляется к списку.

Обратите внимание, что этот алгоритм не проверяет, был ли элемент в списке раньше. Если в результате подстановки путь от корня до соседнего узла становится короче, алгоритм всегда добавляет данный узел в список возможных. Алгоритм продолжает удалять узлы из списка возможных, проверяя соседние узлы и добавляя их в список до тех пор, пока список не опустеет.

Если **сравнить алгоритмы расстановки и коррекции меток**, видно, что они похожи. Разница заключается в том, как каждый из них выбирает элементы из списка возможных узлов для вставки в дерево кратчайшего пути.

Алгоритм расстановки меток всегда выбирает связь, которая гарантированно находится в дереве кратчайших путей. После удаления из списка возможных узел добавляется к дереву и уже не будет помещен в список.

Алгоритм коррекции меток всегда выбирает первый узел из списка возможных, который не всегда является лучшим выбором. Значения полей Dist и InLink данного узла могут и не быть лучшими возможными значениями. Но в конце концов алгоритм отыщет в списке узел, через который проходит более короткий путь к выбранному узлу. Тогда алгоритм обновляет поля Dist и InLink неправильно выбранного узла и помещает этот узел обратно в список возможных.

Алгоритм может использовать новый путь для формирования других путей, которые ранее могли быть пропущены. Помещая обновленный узел опять в список возможных, алгоритм гарантирует, что этот узел снова будет проверен и найдутся все такие пути.

В отличие от алгоритма расстановки меток этот алгоритм не может обрабатывать сети, которые содержат циклы с отрицательной стоимостью. Если встречается такой цикл, то алгоритм бесконечно перемещается по связям внутри него. Каждый раз, когда алгоритм проходит через цикл, сокращается расстояние до входящих в него узлов, при этом алгоритм снова помещает узлы в список возможных узлов и снова проверяет их. При следующей проверке узлов расстояние до них также уменьшится и т.д.

Варианты алгоритма коррекции меток

Данный алгоритм позволяет быстро выбирать узлы из списка возможных. Он может также добавлять узел в список всего за один или два шага. Недостаток алгоритма коррекции меток состоит в том, что когда он выбирает узел из списка возможных, этот выбор не всегда оказывается удачным. Если алгоритм выбирает узел до того, как поля Dist и InLink этого узла получат свои конечные значения, он должен потом исправить значения этих полей и снова поместить узел в список возможных. Чем чаще алгоритм помещает узлы назад в список, тем больше времени занимает его выполнение.

Варианты этого алгоритма направлены на улучшение качества выбора узлов без выполнения дополнительной работы. Один из методов, который на практике работает достаточно хорошо, заключается в том, чтобы добавлять узлы одновременно и в начало, и в конец списка возможных узлов. Если узел прежде не был в списке, он как обычно добавляется в конец списка. Если узел уже был в списке возможных узлов, но в данный момент его там нет, алгоритм помещает его в начало списка. При этом повторное обращение к узлу выполняется практически сразу, иногда при следующем же обращении к списку.

Основанный на этом методе подход состоит в следующем: если алгоритм совершил ошибку, она должна исправляться как можно скорее. Если ошибка не будет устранена в течение долгого времени, алгоритм может использовать неправильную информацию при построении длинных ложных путей, которые затем придется исправлять. Благодаря быстрому устранению ошибок число неверных путей, нуждающихся в исправлении, будет сокращено. В наилучшем случае, если соседние узлы все еще находятся в списке возможных, повторная проверка данного узла перед исследованием соседних устраняет построение неправильных путей.

Варианты поиска кратчайшего пути

Предыдущие алгоритмы поиска кратчайшего пути вычисляют все кратчайшие пути от одного корневого узла до всех остальных узлов сети. Есть много других типов задач по нахождению кратчайшего пути. В этом разделе обсуждаются два из них: двухточечный кратчайший путь и кратчайший путь для всех пар.

Двухточечный кратчайший путь

В некоторых приложениях необходимо находить кратчайший путь между двумя точками, при этом остальные пути в полном дереве кратчайшего маршрута не важны. Простой способ нахождения двухточечного

кротчайшею пути состоит в том, чтобы вычислить полное дерево кратчайших путей с помощью алгоритмов расстановки или коррекции меток, а затем выбрать из дерева кратчайший путь между двумя точками.

Другой способ заключается в использовании метода расстановки меток, который прекращает работу, когда находит путь к конечному узлу. Алгоритм расстановки меток помещает в дерево кратчайшего маршрута только действительно существующие пути, следовательно, в тот момент, когда алгоритм добавит конечный узел в дерево, будет найден искомый кратчайший маршрут. В алгоритме коррекции меток это происходит, когда удаляется конечный узел из списка возможных узлов.

Единственное изменение требуется внести в ту часть алгоритма расстановки меток, которая выполняется сразу после того, как алгоритм нашел в списке возможных узлов узел с наименьшим значением Dist. Перед удалением узла из списка алгоритм должен проверить, является ли данный узел конечным. Если это так, то дерево кратчайшего пути уже содержит правильный путь от начального до конечного узла, и алгоритм может завершить работу.

На практике, если две точки находятся далеко друг от друга, этот алгоритм обычно выполняется дольше, чем вычисление полного дерева кратчайших путей. Это объясняется тем, что в каждом цикле алгоритма проверяется, достигнут ли искомый узел. С другой стороны, если узлы расположены близко друг к другу, выполнение этого алгоритма займет меньше времени, чем построение полного дерева кратчайших путей. Поэтому на практике целесообразно использовать комбинацию этих двух методов.

Вычисление кратчайшего пути для всех пар

В некоторых приложениях требуется быстро найти кратчайший путь между всеми нарами узлов сети. Если необходимо вычислять большую часть из всех возможных N^2 путей, то проще заранее найти все возможные кратчайшие пути, а не высчитывать их каждый по мере надобности.

Можно сохранить кратчайшие пути в двумерных массивах Dists и inLinks. В ячейке Dists[i,j] содержится кратчайшее расстояние от узла i до узла j, а в ячейке InLinks[i,j] - связь, которая ведет к узлу j в кратчайшем пути от узла i до узла j. Эти значения подобны значениям Dist и InLink в классе узла из предыдущих алгоритмов.

Один из способов поиска кратчайших путей состоит в том, чтобы построить деревья кратчайших путей с корнями в каждом узле сети при помощи одного из предыдущих алгоритмов, а затем сохранить результаты в массивах InLinks и Dists.

Другой метод вычисления всех кратчайших путей последовательно строит пути через сеть, используя все более увеличивающееся количество

узлов. Сначала алгоритм отыскивает все кратчайшие пути, которые обходят только первый узел плюс конечные узлы пути. Другими словами, для узлов j и k алгоритм находит кратчайший путь между этими узлами, который использует только узел с номером 1 и узлы j и k, если такой путь существует.

Затем алгоритм находит все кратчайшие пути, содержащие только первые два узла. Потом он строит пути, использующие первые три узла, первые четыре узла и т.д., пока не построит все самые кратчайшие пути, обходящие все узлы. На этом этапе, поскольку кратчайшие пути могут включать в себя любой узел, алгоритм найдет все кратчайшие пути в сети.

Кратчайший путь от узла j к узлу k, использующий только первые i узлов будет включать узел i только в случае выполнения неравенства Dist[j,k]>Dist[j,i]+Dist[i,k]. В противном случае кратчайшим будет предыдущий кратчайший путь, использующий только первые i - 1 узлов. Это означает, что когда алгоритм рассматривает узел i, надо проверить только это условие. Если оно выполняется, алгоритм обновляет кратчайший путь от узла j к узлу k. Иначе старый кратчайший путь между этими двумя узлами все еще является таковым.

Применение алгоритмов поиска кратчайшего пути

Задача о пожарных депо

Предположим, что имеется карта города, которая показывает расположение всех пожарных депо. Необходимо определить для каждой точки города ближайшее к ней депо. Для решения задачи можно попытаться вычислить деревья кратчайших путей с корнями в каждом узле сети, чтобы найти, какое из пожарных депо расположено ближе всего к тому или иному узлу. Или создать дерево кратчайших путей, с корнями в каждом из пожарных депо и сохранить расстояния от каждого пожарного депо до каждого узла сети. Но есть и более эффективный метод.

Необходимо создать ложный корневой узел и соединить его с каждым пожарным депо связями нулевой стоимости. Затем найти дерево кратчайших путей с корнем в этом фиктивном узле. Для каждой точки сети кратчайший путь от ложного корневого узла к данной точке пройдет через ближайшее к ней пожарное депо. Чтобы найти ближайшее к данной точке пожарное депо, необходимо проследовать по кратчайшему пути от точки к корню, пока на пути не встретится одно из депо. Таким образом, построив всего одно дерево кратчайших путей, можно найти все ближайшие пожарные депо для каждой точки сети.

Максимальный поток

Во многих сетях звенья имеют кроме стоимости еще и пропускную способность. Через каждый узел сети проходит поток, который не может превышать ее пропускной способности. Например, каждая улица может пропустить только определенное количество автомобилей в час. Если число машин превышает пропускную способность связи, образуется автомобильная пробка. Сеть с заданными пропускными способностями связей называется **нагруженной сетью**. Если задана нагруженная сеть, то задачей о максимальном потоке будет определение самого большего потока через сеть от заданного источника до заданного стока.

На рис. 39 изображена небольшая нагруженная сеть. Числа рядом с ребрами - это не стоимость связи, а ее пропускная способность. В данном примере максимальный поток, равный 4, получается, если две единицы потока направляются по пути A, B, E, F и еще две- по пути A, C, D, F.

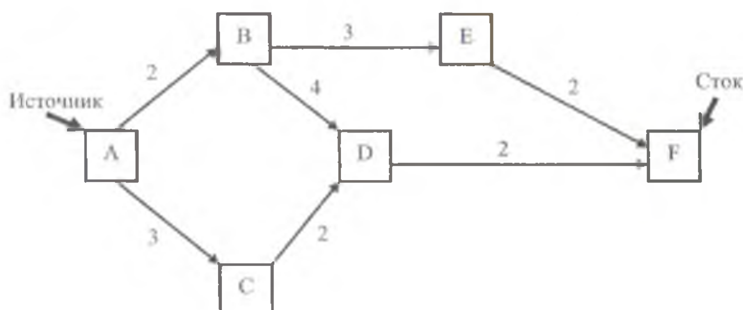


Рис. 39. Нагруженная сеть

Описанный здесь алгоритм начинает работу с того, что поток во всех связях равен нулю. Затем он постепенно увеличивает потоки, чтобы улучшить найденное решение. Когда сделать это уже невозможно, алгоритм завершает работу.

Чтобы найти способы увеличения полного потока, алгоритм исследует разностную пропускную способность связи. **Разностная пропускная способность связи** между узлами i и j равна максимальному дополнительному сетевому потоку, который можно направить из узла i в узел j, используя связь между i и j и связь между j и i. Этот сетевой поток может включать дополнительный поток через связь i-j, если есть резерв пропускной способности. Он может также исключать часть из потока j-i, если по данной связи идет поток.

Например, предположим, что в сети, соединяющей узлы A и C на рис. 39, существует поток, равный 2. Поскольку пропускная способность этой

связи равна 3, к ней можно добавить единицу потока, поэтому ее разностная пропускная способность равна 1. Хотя сеть, изображенная на рис. 39, не имеет связи C-A, разностная пропускная способность для этой связи существует. В данном примере, так как по связи C-A идет поток, равный 2, то можно удалить до двух единиц данного потока. Это увеличит сетевой поток от узла C к узлу A на 2, поэтому разностная пропускная способность связи C-A равна 2.

Сеть, состоящая из всех связей с положительной разностной пропускной способностью, называется **разностной сетью**. На рис. 40 изображена сеть с рис. 39, каждой связи в которой присвоен поток. Для каждой связи первое число равно потоку через связь, а второе - ее пропускной способности. Метка "1/2", например, означает, что связь проводит поток 1 и имеет пропускную способность 2. Связи, несущие потоки больше нуля, нарисованы жирными линиями.

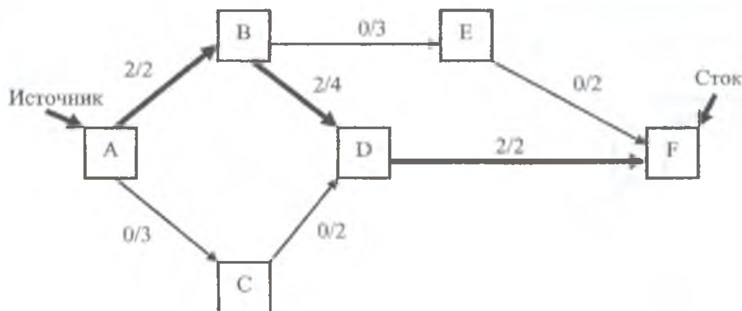


Рис. 40. Сетевые потоки

На рис. 41 изображена разностная сеть, соответствующая потокам, приведенным на рис. 40. Показаны только те связи, которые действительно могут иметь разностную пропускную способность. Например, между узлами A и D не нарисовано ни одной связи. Исходная сеть не имеет связи A-D или связи D-A, поэтому эти связи всегда будут иметь нулевую разностную пропускную способность.

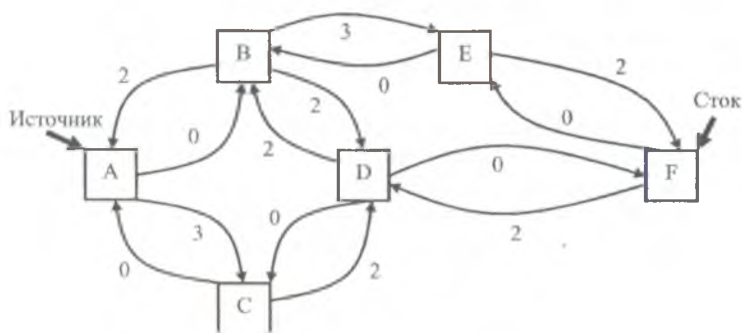


Рис. 41. Разностная сеть

Важное свойство разностных сетей заключается в том, что любой путь, использующий связи с разностной пропускной способностью больше нуля, который соединяет источник со стоком, показывает способ увеличения потока сети. Этот путь называется **расширяющим путем**. На рис. 42 изображена разностная сеть с рис. 41, расширяющий путь в ней выделен жирными линиями.

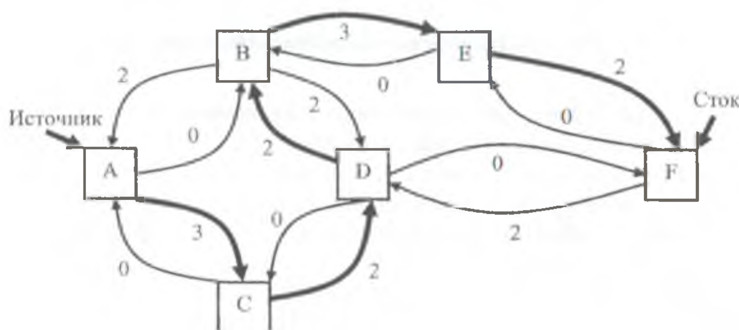


Рис. 42. Расширяющийся путь через разностную сеть

Чтобы улучшить решение с помощью расширяющего пути, необходимо найти наименьшую разностную пропускную способность на этом пути. Затем следует скорректировать потоки в пути в соответствии с данной величиной. На рис. 42, например, наименьшая разностная пропускная способность любого ребра на расширяющем пути равна 2. Чтобы обновить потоки в сети, следует прибавить поток 2 к любой связи пути i-j, а из всех обратных им связей j-i вычесть поток 2.

Гораздо проще изменить разностную сеть, а не корректировать потоки и затем перестраивать разностную сеть. Затем после завершения работы алгоритма можно использовать результат для вычисления потоков для связей в исходной сети.

Чтобы изменить разностную сеть в данном примере, необходимо проследовать по расширяющему пути и вычитать 2 из разностной пропускной способности любого ребра i-j на этом пути, и добавлять 2 к разностной пропускной способности соответствующего ребра j-i. На рис. 43 изображена измененная разностная сеть для данного примера.

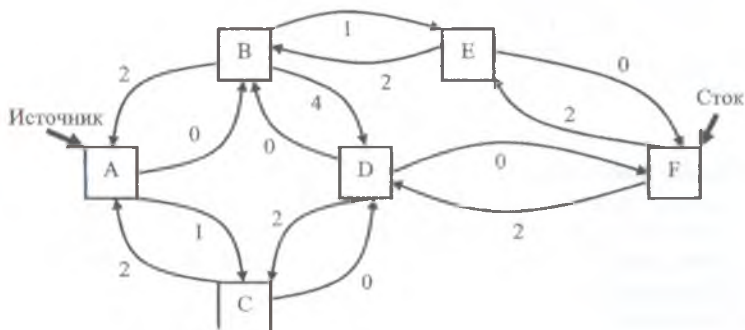


Рис. 43. Измененная разностная сеть

Если нельзя больше найти ни одного расширяющего пути, то можно использовать разностную сеть для вычисления потоков в исходной сети. Для каждой связи между узлами i и j, если разностный поток между узлами i и j меньше, чем пропускная способность связи, поток должен равняться пропускной способности минус разностный поток. В противном случае поток должен быть равен нулю.

Например, на рис. 43 разностный поток от узла A к узлу C равен 1, а пропускная способность связи A-C равна 3. Поскольку 1 меньше 3, поток через узел будет равен 3-1=2. На рис. 44 показаны сетевые потоки, соответствующие разностной сети на рис. 43.

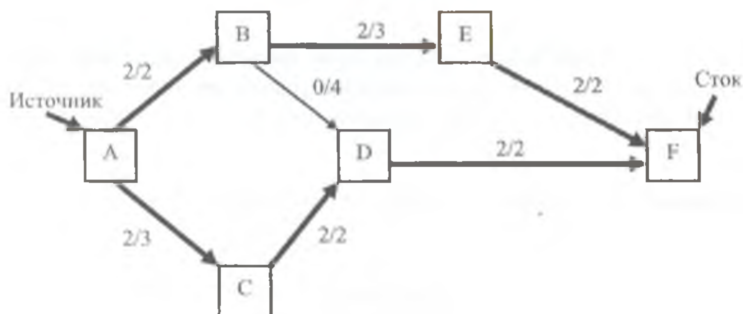


Рис. 44. Максимальные потоки

На данный момент алгоритм не содержит методов поиска расширяющих путей в разностной сети. Один из подходящих для этой задачи методов похож на алгоритм коррекции меток для поиска кратчайшего пути. Сначала следует поместить узел-источник в список возможных узлов. Затем необходимо удалять элементы из списка, пока список не опустеет, исследуя при этом все соседние узлы, соединенные с выбранным узлом связью с разностной пропускной способностью больше нуля. Если соседний узел еще не был в списке возможных, его необходимо туда добавить.

5. ЗАДАЧИ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

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

Примечание: Программа не должна оперировать с двумерными массивами

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

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

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

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

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

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

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

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

6. Написать программу, реализующую сравнительный анализ трех способов последовательного поиска

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

8. Написать программу, находящую минимальное остовное дерево для графа

9. Написать программу, определяющую кратчайший путь в графе от выбранной вершины до всех остальных с помощью любого известного алгоритма

10. Написать программу, определяющую максимальный поток в графе.

ЗАКЛЮЧЕНИЕ

Разработка высокопроизводительных алгоритмов по обработке информации не прекращается ни на минуту. Каждый год в научных журналах появляются новые способы поиска информации, анализ того или иного алгоритма и т.д. И поэтому внимательный читатель данного пособия может подвергнуть сомнению целесообразность практического использования ряда алгоритмов приведенных в нем. Однако целью данной части курса было не только знакомство с оптимальными алгоритмами, но и выработка у студентов творческого мышления и способности к критическому анализу сделанного ранее. После того, как произошло знакомство с основными алгоритмами по обработке информации, необходимо переходить к рассмотрению современных способов разработки программного обеспечения – объектно-ориентированному проектированию и технологии RUP. Рассмотрению этих и других вопросов будут посвящены следующие части данного учебно-методического пособия.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979. – 536 с.
2. Вирт Н. Алгоритмы и структуры данных. – М.: Мир, 1989. – 360 с.
3. Далека В. Д., Деревянко А.С., Кравец О.Г., Тимановская Л.Е. Модели и структуры данных. – Харьков: ХГПУ, 2000. – 241с.
4. Кнут Д. Искусство программирования. Т. 1. Основные алгоритмы, 3-е изд. / Пер. с англ. – М.: Издательский дом “Вильямс” , 2000. – 720 с.
5. Кнут Д. Искусство программирования. Т. 3. Сортировка и поиск, 2-е изд. / Пер. с англ. – М.: Издательский дом “Вильямс” , 2000. – 832 с.
6. Стивенс Р. Delphi. Готовые алгоритмы. -М.:ДМК. Пресс, 2001. – 384с.

Крутов Алексей Николаевич

МЕТОДЫ ПРОГРАММИРОВАНИЯ.
КЛАССИЧЕСКИЕ АЛГОРИТМЫ

Учебное пособие

Печатается в авторской редакции

Компьютерная верстка, макет Н.П.Бариновой

Лицензия ИД № 06178 от 01.11.01. Подписано в печать 01.09.04. Гарнитура «Times New Roman». Формат 60x84/16. Бумага офсетная. Печать оперативная. Объем 4,65 усл. печ. л., 5,0 уч. -изд. л. Тираж 150 экз. Заказ № 210
Издательство «Самарский университет», 443011, г. Самара, ул. Ак. Павлова, д.1.
Отпечатано ООО «Универс-групп»