Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Кафедра информатики и прикладной математики

ОПИСАНИЯ КОМБИНАТОРНЫХ АЛГОРИТМОВ

составил доц. Кукушкин Б.А. 01.02.1994

пытались дополнить и перевести в новый формат студенты кафедры ИПМ в 2010-2011 гг.

Содержание

[ЧАСТЬ 1. АЛГОРИТМЫ СОРТИРОВКИ 3](#_Toc348052223)

[1. Методы вставки. Алгоритм простых вставок 3](#_Toc348052224)

[1.1. Бинарные вставки 4](#_Toc348052225)

[1.2. Двухпутевые вставки 5](#_Toc348052226)

[1.3. Вставки одновременно нескольких элементов 6](#_Toc348052227)

[1.4. Вставки с убывающим шагом (метод Шелла) 7](#_Toc348052228)

[1.5. Вставки в связанный список 9](#_Toc348052229)

[1.6. Вставки в несколько связанных списков 10](#_Toc348052230)

[2. Обменная сортировка 11](#_Toc348052231)

[2.1. Метод пузырька 11](#_Toc348052232)

[2.2. Модификация метода пузырька 12](#_Toc348052233)

[2.3. Быстрая сортировка 12](#_Toc348052234)

[2.4. Обменная поразрядная сортировка 16](#_Toc348052235)

[2.5. Параллельная сортировка Бэтчера 18](#_Toc348052236)

[3. Сортировка посредством выбора 21](#_Toc348052237)

[3.1. Использование связанного списка для хранения информации о промежуточных максимумах 21](#_Toc348052238)

[3.2. Пирамидальная сортировка 22](#_Toc348052239)

[4. Сортировка посредством слияния 27](#_Toc348052240)

[4.1. Естественное двухпутевое слияние 27](#_Toc348052241)

[4.2. Простое двухпутевое слияние 28](#_Toc348052242)

[4.3. Слияние связанных списков 30](#_Toc348052244)

[5. Распределяющая сортировка 31](#_Toc348052246)

[ЧАСТЬ 2. АЛГОРИТМЫ ПОИСКА 35](#_Toc348052247)

[6. Последовательный поиск 35](#_Toc348052248)

[6.1. Последовательный поиск по связанному списку 35](#_Toc348052249)

[6.2. Последовательный поиск с перестановкой элементов 35](#_Toc348052250)

[6.3. Последовательный поиск в упорядоченном файле 35](#_Toc348052251)

[6.4. Бинарный поиск в упорядоченном файле 36](#_Toc348052252)

[6.5. Однородный бинарный поиск 36](#_Toc348052253)

[6.6. Интерполяционный поиск 36](#_Toc348052254)

[7. Поиск по бинарным деревьям 37](#_Toc348052255)

[7.1. Поиск по бинарному дереву 37](#_Toc348052256)

[7.2. Удаление из бинарного дерева 38](#_Toc348052257)

[7.3. Построение оптимальных бинарных деревьев поиска 39](#_Toc348052260)

[7.4. Цифровой поиск по дереву 41](#_Toc348052261)

[7.5. Цифровой поиск для длинных ключей, хранимых в текстовом массиве (Патриция) 41](#_Toc348052262)

# ЧАСТЬ 1. АЛГОРИТМЫ СОРТИРОВКИ

## 1. Методы вставки. Алгоритм простых вставок

Сортировка вставками – простой [алгоритм сортировки](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8). Хотя этот алгоритм сортировки уступает в эффективности более сложным (таким как [Быстрая сортировка](http://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)), у него есть ряд преимуществ:

- прост в реализации;

- эффективен на небольших наборах данных, на наборах данных до десятков элементов может оказаться лучшим;

- эффективен на наборах данных, которые уже частично отсортированы;

- это устойчивый алгоритм сортировки (не меняет порядок элементов, которые уже отсортированы);

- может сортировать список по мере его получения;

- не требует временной памяти, даже под стек.

Алгоритм использует прием, которым пользуются игроки в карты при сортировке только что розданной колоды: очередная карта вставляется между уже упорядоченными ранее.

На каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированном списке, до тех пор пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора. Обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по порядку их появления во входном массиве.

Файл, подлежащий сортировке, в общем случае состоит из элементов-записей, включающих информационную часть и ключи, по которым производится упорядочение по возрастанию. Поскольку информационная часть почти не влияет на процесс сортировки, будем предполагать, что файлы, используемые в примерах, состоит только из элементов-ключей, а информационная часть записи отсутствует. Здесь k[1], k[2], ... , k[N] - ключи, по которым производится упорядочение файла. X,i,j - рабочие переменные.

*for j:=2 to N do begin*

*i:=j-1;*

*X:=k[j];*

*while (X<k[i]) and (i>0) begin*

*k[i+1]:=k[i];*

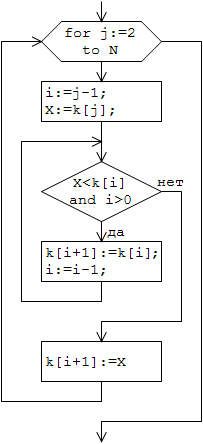
*i:=i-1;*

*end;*

*k[i+1]:=X;*

*end*

Блок-схема алгоритма будет выглядеть так:



Для примера возьмем файл, состоящий из 8 элементов:

Исх.файл.: 503 87 512 61 908 170 897 275

Алгоритм будет преобразовывать его следующим образом:

j=2 503 87 X= 87

Результат: 87 503

j=3 87 503 512 X=512

j=4 61 87 503 512 X= 61

j=5 61 87 503 512 908 X=908

j=6 61 87 170 503 512 908 X=170

j=7 61 87 170 503 512 897 908 X=897

j=8 61 87 170 275 503 512 897 908 X=275

Время работы алгоритма t примерно оценивается формулой:

t=a\*N2 + b\*N,

где a,b - неизвестные константы, зависящие от программной реализации алгоритма.

Таким образом, время выполнения алгоритма зависит от входных данных: чем большее множество нужно отсортировать, тем большее время выполняется сортировка. Также на время выполнения влияет исходная упорядоченность массива. Так, лучшим случаем является отсортированный массив, а худшим - массив, отсортированный в порядке, обратном нужному.

Алгоритм простых вставок порождает несколько модификаций.

### 1.1. Бинарные вставки

Этот метод упоминается Джоном Мочли в 1946г. в первой публикации по машинной сортировке.

Метод является улучшенной версией предыдущего метода простых вставок. Он основывается на том факте, что мы вначале ищем место в уже упорядоченной части массива, куда нужно вставить элемент, используя бинарный поиск, а затем уже вставляем. Это дает существенный выигрыш по числу сравнений, но число перестановок по прежнему остается большим.

Сначала Х сравнивается с элементом k[j/2], затем, в зависимости от результата сравнения, с элементом, лежащим посередине между 1 и j/2 или посередине между j/2+1 и j и т.д.

Например, если вставляется 64-я запись, то можно сравнить ее с 32-й, и если 64-я запись окажется меньше, то сравнить ее уже с 16-й, а если окажется больше, то сравнить с 48-й и т.д. Таким образом, место 64-й записи будет определено в худшем случае за шесть сравнений.

При этом число сравнений для каждого j равно примерно lg(j).Число пересылок элементов при этом не изменяется и остается примерно равным N/4.

Время работы алгоритма t примерно оценивается формулой:

t=a\*N2 + b\*N + c\*N\*lgN,

где a,b,c - неизвестные константы, зависящие от программной реализации алгоритма.

### 1.2. Двухпутевые вставки

Главный недостаток метода простых вставок заключается в том, что приходится сдвигать большое количество элементов. Метод бинарных вставок позволяет ускорить процесс поиска места для вставки очередного элемента. Однако для освобождения этого места по-прежнему необходимо подвинуть примерно 1/2j ранее рассортированных записей. В начале 50-х годов был предложен один из первых приемов, позволяющий сократить число необходимых переписей в памяти, - метод двухпутевых вставок.

Число пересылок можно сократить примерно в 2 раза (до N2/8), если допустить сдвиги элементов не только вправо, но и влево. Для выходного файла резервируется место в памяти, равное 2N-1,где N – число элементов в исходном файле. Первый элемент пересылается в середину выходного файла. В дальнейшем элементы выходного файла сдвигаются вправо или влево в зависимости от того, в какую сторону нужно сдвигать меньше элементов. Файл из предыдущего примера будет сортироваться следующим образом.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| исходный файл | | | | | | | | | | | | | | |  |
| 503 |  | 87 |  | 512 |  | *61* |  | 908 |  | 170 |  | 897 |  | 275 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| выходной файл | | | | | | | | | | | | | | | комментарий |
|  |  |  |  |  |  |  |  | 503 |  |  |  |  |  |  | X = 87 |
|  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | ***87*** |  | 503 |  |  |  |  |  |  | X = 512 |
|  |  |  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |
|  |  |  |  |  |  | 87 |  | 503 |  | ***512*** |  |  |  |  | X = 61 |
|  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | ***61*** |  | 87 |  | 503 |  | 512 |  |  |  |  | X = 908 |
|  |  |  |  |  |  |  |  |  |  |  | ^ |  |  |  |  |
|  |  |  |  | 61 |  | 87 |  | 503 |  | 512 |  | ***908*** |  |  | X = 170 |
|  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |
|  |  | 61 |  | 87 |  | ***170*** |  | 503 |  | 512 |  | 908 |  |  | X = 897 |
|  |  |  |  |  |  |  |  |  |  |  | ^ |  |  |  |  |
|  |  | 61 |  | 87 |  | 170 |  | 503 |  | 512 |  | ***897*** |  | 908 | X = 275 |
|  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |
| 61 |  | 87 |  | 170 |  | ***275*** |  | 503 |  | 512 |  | 897 |  | 908 | конеч. сост. |

Из таблицы видно, что присоединение новых элементов к выходному файлу происходит как справа, так и слева от центрального элемента 503 с возможным сдвигом вправо или влево.

Необходимость отвести для выходного отсортированного массива место под 2N-1 элементов объясняется тем, что изначально неизвестно, куда относительно первой записи будут добавляться элементы. Например, если первый элемент окажется минимальным, то все последующие будут расположены справа от него, а слева останется N-1 пустых записей. Аналогично для случая, когда первый элемент наибольший, заполненными будут только N левых элементов. При реализации алгоритма необходимо следить за правой и левой границей и освобождать неиспользованные области памяти.

Время работы алгоритма t примерно оценивается формулой:

t = a\*N2 + b\*N

где a, b - неизвестные константы, зависящие от программной реализации алгоритма.

### 1.3. Вставки одновременно нескольких элементов

Модификация метода простых вставок заключается в том, что вместо одной переменной Х можно использовать несколько переменных Х1, Х2, ... Xm, которые имеют значения элементов, подлежащих вставке в уже упорядоченную часть файла. Х1, X2, ... Xm упорядочены по возрастанию, поэтому, сравнивая Xm в цикле по переменной i с элементами упорядоченной части, мы можем гарантировать, что, если очередной элемент k[i] больше Xm, то он больше и остальных элементов. Перенос элементов исходного файла вперед в цикле по i выполняется на m элементов, то есть вместо k[i+1] = k[i] в исходном алгоритме в модифицированном алгоритме записывается k[i+m] = k[i]. При нахождении k[i] такого, что он меньше Хm, Хm ставится на место k[i+1] и m уменьшается на 1. Далее цикл по i продолжается с новым m. Экономия числа переносов элементов достигается за счет переносов сразу на m элементов.

Рассмотрим сортировку методом вставки одновременно нескольких элементов на конкретном примере.

Здесь m = 4.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 512 |  | 87 |  | 61 |  | 503 |  | 170 |  | 897 |  | 275 |  | 908 |  | 10 |  | 605 |  | 150 |  | 603 |  | 101 |  | 509 |  | 849 |  | 65 |  | I |
|  | 512 |  | 87 |  | 61 |  | 503 |  | 170 |  | 897 |  | 275 |  | 908 |  | 10 |  | 605 |  | 150 |  | 603 |  | 101 |  | 509 |  | 849 |  | 65 |  | II |
|  | 61 |  | 87 |  | 503 |  | 512 |  | 170 |  | 275 |  | 897 |  | 908 |  | 10 |  | 150 |  | 603 |  | 605 |  | 65 |  | 101 |  | 509 |  | 849 |  | III |

I - исходный массив;

II - массив, разделенный на подмножества, по m элементов в каждом;

III - подмножества отсортированы в порядке возрастания.

Сортировку каждого подмножества целесообразнее проводить не перед началом всей сортировки, а на очередном шаге выполнения алгоритма. Приведенный здесь способ ориентирован на сокращение таблицы и облегчения понимания сути метода.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| выходной массив | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | X1 | X2 | X3 | X4 | m |
|  | 61 |  | 87 |  | 503 |  | 512 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 170 | 275 | 897 | 908 | 4 |
|  |  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 61 |  | 87 |  | 503 |  | 512 |  | ***908*** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 170 | 275 | 897 |  | 3 |
|  |  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 61 |  | 87 |  | 503 |  | 512 |  | ***897*** |  | 908 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 170 | 275 |  |  | 2 |
|  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 61 |  | 87 |  | ***275*** |  | 503 |  | 512 |  | 897 |  | 908 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 170 |  |  |  | 1 |
|  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 61 |  | 87 |  | ***170*** |  | 275 |  | 503 |  | 512 |  | 897 |  | 908 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 10 | 150 | 603 | 605 | 4 |
|  |  |  |  |  |  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 61 |  | 87 |  | 170 |  | 275 |  | 503 |  | 512 |  | ***605*** |  | 897 |  | 908 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 10 | 150 | 603 |  | 3 |
|  |  |  |  |  |  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 61 |  | 87 |  | 170 |  | 275 |  | 503 |  | 512 |  | ***603*** |  | 605 |  | 897 |  | 908 |  |  |  |  |  |  |  |  |  |  |  |  |  | 10 | 150 |  |  | 2 |
|  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 61 |  | 87 |  | ***150*** |  | 170 |  | 275 |  | 503 |  | 512 |  | 603 |  | 605 |  | 897 |  | 908 |  |  |  |  |  |  |  |  |  |  |  | 10 |  |  |  | 1 |
| ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **10** |  | 61 |  | 87 |  | 150 |  | 170 |  | 275 |  | 503 |  | 512 |  | 603 |  | 605 |  | 897 |  | 908 |  |  |  |  |  |  |  |  |  | 65 | 101 | 509 | 849 | 4 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 10 |  | 61 |  | 87 |  | 150 |  | 170 |  | 275 |  | 503 |  | 512 |  | 603 |  | 605 |  | ***849*** |  | 897 |  | 908 |  |  |  |  |  |  |  | 65 | 101 | 509 |  | 3 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 10 |  | 61 |  | 87 |  | 150 |  | 170 |  | 275 |  | 503 |  | ***509*** |  | 512 |  | 603 |  | 605 |  | 849 |  | 897 |  | 908 |  |  |  |  |  | 65 | 101 |  |  | 2 |
|  |  |  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 10 |  | 61 |  | 87 |  | ***101*** |  | 150 |  | 170 |  | 275 |  | 503 |  | 509 |  | 512 |  | 603 |  | 605 |  | 849 |  | 897 |  | 908 |  |  |  | 65 |  |  |  | 1 |
|  |  |  |  | ^ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 10 |  | 61 |  | ***65*** |  | 87 |  | 101 |  | 150 |  | 170 |  | 275 |  | 503 |  | 509 |  | 512 |  | 603 |  | 605 |  | 849 |  | 897 |  | 908 |  | конеч. сост. | | | | |

Из таблицы видно, что при каждом поиске места для вставки, элемент Xm-j-1 оказывается левее элемента Xm-j.

Время работы алгоритма t примерно оценивается формулой:

t = a\*N2 + b\*N + c\*N\*lgN

где a, b, c - неизвестные константы, зависящие от программной реализации алгоритма.

**Сетевые ресурсы:**

1. http://ru.wikipedia.org/wiki/Сортировка\_методом\_вставок – Википедия. Сортировка методом вставок

2. http://jsf.boom.ru/programm/algoritm/index.htm – Janis vs StarFox. Алгоритмы сортировки.

3. http://pascal.proweb.kz/index.php?page=78 – Программирование на паскале. Методы вставок.

4. http://www.dmtsoft.ru/bn/343/as/oneaticleshablon/

5. http://waidos32.narod.ru/otvet/1\_41.html

### 1.4. Вставки с убывающим шагом (метод Шелла)

Идея алгоритма состоит в обмене элементов, расположенных не только рядом, как в алгоритме простых вставок (п.1), но и далеко друг от друга, что значительно сокращает общее число операций перемещения элементов. Для примера возьмем файл из 16 элементов. Сначала просматриваются пары с шагом 8. Это пары элементов 1-9, 2-0, 3-11, 4-12, 5-13, 6-14, 7-15, 8-16. Если значения элементов в паре не упорядочены по возрастанию, то элементы меняются местами. Назовем этот этап 8-сортировкой. Следующий этап - 4-сортировка, на котором элементы в файле делятся на четверки: 1-5-9-13, 2-6-10-14, 3-7-11-15, 4-8-12-16. Выполняется сортировка в каждой четверке. Сортировка может выполняться методом простых вставок (п.1). Следующий этап - 2-сортировка, когда элементы в файле делятся на 2 группы по 8: 1-3-5-7-9-11-13-15 и 2-4-6-8-10-12-14-16. Выполняется сортировка в каждой восьмерке. Наконец весь файл упорядочивается методом простых вставок. Поскольку дальние элементы уже переместились на свое место или находятся вблизи от него, этот этап будет значительно менее трудоемким, чем при сортировке вставками без предварительных "дальних" обменов. Для данного примера результаты представлены в следующей таблице. Исходный файл. На нем выполняется 8-сортировка.Пары для возможного обмена соединены одинарными или двойными линиями. Двойными линиями обозначены пары, в которых произошел обмен.

503 87 512 61 908 170 897 275 653 426 154 509 612 677 765 703

└──┼───╫──┼──╫────┼───╫───┼───┘ │ ║ │ ║ │ ║ │

└───╫──┼──╫────┼───╫───┼───────┘ ║ │ ║ │ ║ │

╙──┼──╫────┼───╫───┼───────────╜ │ ║ │ ║ │

└──╫────┼───╫───┼───────────────┘ ║ │ ║ │

╙────┼───╫───┼───────────────────╜ │ ║ │

└───╫───┼───────────────────────┘ ║ │

╙───┼───────────────────────────╜ │

└───────────────────────────────┘

Файл после 8-сортировки. Линиями соединены четверки для следующего этапа.

503 87 154 61 612 170 765 275 653 426 512 509 908 677 897 703

└───┼──┼──┼───┴───┼───╫───┼───┴───┼───╫───┼───┘ │ │ │

└──┼──┼───────┴───╫───┼───────┴───╫───┼───────┘ │ │

└──┼───────────╨───┼───────────╨───┼───────────┘ │

└───────────────┴───────────────┴───────────────┘

Файл после 4-сортировки. Линиями соединены восьмерки для следующего этапа.

503 87 154 61 612 170 512 275 653 426 765 509 908 677 897 703

╙──╫───╨──╫───╨───┼───╨───┼───┴───┼───┴───┼───╨───┼───╜ │

╙──────╨───────┴───────┴───────┴───────┴───────┴───────┘

Файл после 2-сортировки:

154 61 503 87 512 170 612 275 653 426 765 509 897 677 908 703

Файл после окончательной сортировки (1-сортировки):

61 87 154 170 275 426 503 509 512 612 653 677 703 765 897 908

Время работы алгоритма t примерно оценивается формулой:

t=a\*N\*\*1.25

где a - неизвестная константа, зависящая от программной реализации алгоритма.

**Альтернативное описание:**

Суть метода в том, что в отличие от обычной сортировки вставками, а метод Шелла это не что иное, как модификация метода простых вставок, здесь происходит упорядочивание элементов стоящих друг от друга на шаге d, т.е. упорядочиваются элементы с индексами отличающимися на d. Затем элементов стоящих друг от друга на шаге h (с индексами отличающимися на h причем h<d) и т.д.. Например: имея массив длинной 15 элементов, и сортируя его на шаге 5, выйдет так, что мы возьмем 0-й, 5-й и10-й элементы, и отсортируем их, затем 1-й, 6-й и 11-й и отсортируем их, затем 2-й, 7-й 12-й и их тоже отсортируем, будем повторять эти шаги, пока не дойдем до 5-го элемента(считая с нуля), т.е. до группы элементов 4-й,9-й,14-й. После их сортировки мы прекращаем рассматривать и сортировать массив на этом шаге, и обрабатываем полученную последовательность чисел уже с меньшим шагом(например 3, получатся группы {0,3,6,9,12},{1,4,7,10,13},{2,5,8,11,14}). Очевидно, что если продолжать уменьшать шаг, то мы прийдем к единице, а это и есть сортировка простыми вставками, но в отличие от нее, мы выиграли во времени и количестве перестановок т.к. начинали рассматривать элементы не рядом стоящие, а находящиеся на некотором расстоянии друг от друга, таким образом прийдя к сортировке простыми вставками, массив оказался гораздо более упорядоченным.

Время работы алгоритма в худшем случае t примерно оценивается формулой:

t=a\*N\*\*1.5 ,

где a - неизвестная константа, зависящая от программной реализации алгоритма.

**Сетевые ресурсы:**

1. http://ru.wikibooks.org/wiki/Примеры\_реализации\_сортировки\_Шелла - (здесь исходные коды на множество популярных языков программирования таких как Java, C#, C++, PHP, Python).

2. http://algolist.manual.ru/sort/shell\_sort.php - (довольно популярно объяснено описание алгоритма).

3. http://ru.wikipedia.org/wiki/Сортировка\_Шелла - более детальное описание.

4. http://www.sorting-algorithms.com/shell-sort - визуализация (из инвертированного состояния, почти упорядоченного состояния, с часто повторяющимися элементами, произвольного состояния)

### 1.5. Вставки в связанный список

Среди общих способов улучшения алгоритма простых вставок можно рассмотреть способ, основанный на изменении структуры данных. Сортировка простыми вставками состоит из двух основных операций:

просмотра исходного файла со сравнением переменной Х с элементами K[i] файла;

вставки нового элемента путем сдвига оставшихся элементов вправо.

Файл до сих пор рассматривался как линейный список и для выполнения операции вставки в нем необходимо переместить в среднем половину элементов. Известно, что для операций вставки идеально подходит связанный список, так как в этом случае вставка требует всего лишь изменения нескольких связей. Операция последовательного просмотра для связанного списка почти так же проста, как и для линейного списка. Поскольку файл всегда просматривается в одном направлении, то достаточно иметь список только с одной связью. С другой стороны связанное распределение делает невозможным бинарный поиск, поэтому приобретая преимущество в выполнении операции вставки, мы теряем по сравнению с бинарным поиском в эффективности операции просмотра и сравнения. Рассмотрим алгоритм простых вставок на связанном вперед списке.

Дан файл в виде связанного списка, каждый элемент которого содержит кроме ключа K[i] еще и указатель на следующий элемент L[i]. Кроме того есть еще дополнительная переменная L[0], содержащая указатель на последний N-й элемент файла. Указатель L[N] равен нулю, что является признаком конца списка элементов.

K[1]

L[1]

K[2]

L[2]

K[N]

L[N]

…

L[0]

= 0

Упорядоченная часть файла формируется в конце списка и L[0] всегда указывает на начало упорядоченной части, в конце алгоритма - на логическое начало списка.

Алгоритм имеет следующий вид:

for j = N - 1 to 1

P = L[0]; q = 0; X = K[j];

Продвижение по списку

q = p;

p = L[p];

вставка

L[q] = j;

L[j] = p;

while(p > 0 & X > K[p])

q

p

L[q]

K[q]

K[p]

L[p]

Переменные p и q служат указателями на текущий элемент, причем =l[q] (q всегда на один шаг отстает от p). Если p=0, то Х - наибольший элемент и должен попасть в конец списка.

Время работы алгоритма t примерно оценивается формулой:



где a, b - неизвестные константы, зависящие от программной реализации алгоритма.

### 1.6. Вставки в несколько связанных списков

Идея метода основывается на предположении, что ключи в исходном файле имеют значения в некотором известном диапазоне MAX и в этом диапазоне они распределены довольно равномерно. Тогда по аналогии с методом вставки в один связанный список можно организовать несколько, например, Q списков. Величина Q зависит от ожидаемого среднего количества элементов M в каждом списке, то есть Q=N/M, N - количество ключей. При разработке программы нужно проанализировать зависимость времени работы метода от параметра М для различных исходных файлов и дать рекомендации по выбору оптимального значения.

Схема алгоритма имеет следующий вид. Через Q обозначено количество списков, массив B[1]...B[Q] служит для хранения указателей на начала отдельных списков. Перед началом работы алгоритма элементы массива В предполагаются равными 0. Каждый i-й элемент исходного файла содержит ключ K[i] и указатель L[i] на следующий элемент списка.  Значение L[i]=0 соответствует последнему элементу в списке, указатель B[1] указывает на начало первого подсписка и одновременно на начало всего списка. Через minK обозначено минимальное значение ключа в файле, через М - среднее выбранное значение количества элементов в подсписке. d - номер текущего списка, в который должен быть вставлен элемент K[j]. Величина R=MAX/Q есть диапазон значений ключей, приходящийся на один список.

for j = N - 1 to 1

X = K[j];

d = (X - minK) / R + 1; (деление нацело)

p = B[d];

q = 0;

Продвижение по списку

q = p;

p = L[p];

Вставка элемента Х

if(q=0) B[d]=j; else L[q]=j;

L[j]=p;

while(p > 0 & X > K[p])

Время работы алгоритма t примерно оценивается формулой:



где a,b - неизвестные константы, зависящие от программной реализации алгоритма.

## 2. Обменная сортировка

### 2.1. Метод пузырька

Сортировка простыми обменами, сортиро́вка пузырько́м (англ. bubble sort) — простой алгоритм сортировки. Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов.

Сложность алгоритма: O(n²).

Алгоритм считается учебным и практически не применяется вне учебной литературы, вместо него на практике применяется сортировка вставками.

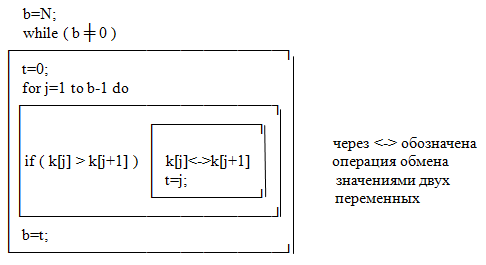
**Алгоритм**

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.

После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.

Алгоритм имеет следующий вид.



### 2.2. Модификация метода пузырька

Рассмотрим ситуацию, когда на каком-либо из проходов не произошло ни одного обмена. Что это значит?

Это значит, что все пары расположены в правильном порядке, так что массив уже отсортирован. И продолжать процесс не имеет смысла (особенно, если массив был отсортирован с самого начала!).

Итак, первое улучшение алгоритма заключается в запоминании, производился ли на данном проходе какой-либо обмен. Если нет - алгоритм заканчивает работу.

Процесс улучшения можно продолжить, если запоминать не только сам факт обмена, но и индекс последнего обмена k. Действительно: все пары соседих элементов с индексами, меньшими k, уже расположены в нужном порядке. Дальнейшие проходы можно заканчивать на индексе k, вместо того чтобы двигаться до установленной заранее верхней границы i.

Качественно другое улучшение алгоритма можно получить из следующего наблюдения. Хотя легкий пузырек снизу поднимется наверх за один проход, тяжелые пузырьки опускаются с минимальной скоростью: один шаг за итерацию. Так что массив 2 3 4 5 6 1 будет отсортирован за 1 проход, а сортировка последовательности 6 1 2 3 4 5 потребует 5 проходов. Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют "шейкер-сортировкой".

**Сетевые ресурсы:**

Визуальные представления метода можно посмотреть на следующих ресурсах:

1. http://algolist.ru/sort/bubble\_sort.php

2. http://www.sorting-algorithms.com – анимационное представление.

### 2.3. Быстрая сортировка

Быстрая сортировка (англ. quicksort) — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром в 1960 году – является одним из наиболее быстрых известных универсальных алгоритмов (в среднем O(n log n) обменов при упорядочении n элементов).

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

1) из массива выбирается некоторый опорный элемент;

2) запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные опорному элементу, влево от него, а все ключи, большие, либо равные – вправо;

3) теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правого;

4) для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

В конце получится полностью отсортированная последовательность

Рассмотрим алгоритм более детально:

Исходным является массив А с номерами элементов от First до Last. В алгоритме используются еще два индекса массива, обозначенные как Index и ContrIndex. Первый из них всегда указывает на переставляемый элемент, а второй — на элемент, который сравнивается по значению с переставляемым. В процессе вычислений применяются переменная h (равная либо 1, либо -1) - шаг движения индексов навстречу друг другу, используемая для обозначения направления движения индекса ContrIndex, и логическая переменная Condition (равная либо TRUE, либо FALSE), используемая для изменения условия сравнения на противоположное при обратном движении индекса ContrIndex.

Шаг 1. Пока Index не равно ContrIndex, делаются шаги: Шаг 2а - Шаг 2b.

Шаг 2а. Если справедливо ((A[Index]>A[ContrIndex])=Condition), то переставляются как сами элементы, на которые указывают Index и ContrIndex, (Val:=A[Index], A[Index]:=A[ContrIndex], A[ContrIndex]:=Val), так и сами вспомогательные индексы массивов (Val:=Index, Index:=ContrIndex, ContrIndex:=Val). Затем меняется направление движения (h:= -h) и условие сравнения (Condition: = not Condition). В процессе таких перестановок слева от переставляемого элемента всегда будут находиться меньшие значения, а справа - большие значения.

Шаг 2b. Сдвигается вспомогательный индекс массива ContrIndex навстречу индексу Index , т.е. ContrIndex:= ContrIndex + h.

Шаг 3. Перед выполнением этого шага индексы Index = ContrIndex и элемент A[Index] находится на нужном месте. Т.е. исходный массив разбит на три части: часть массива до этого элемента, значения в котором меньше величины A[Index], часть массива после этого элемента с значениями большими значения A[Index] и сам этот элемент A[Index]. Поэтому для дальнейшего упорядочивания массива достаточно рекурсивно обратиться к алгоритму быстрой сортировки два раза: для первой и второй частей массива. Т.к. длина сортируемых участков массива уменьшается, то в итоге алгоритм конечен и после применения алгоритма массив будет полностью отсортирован.

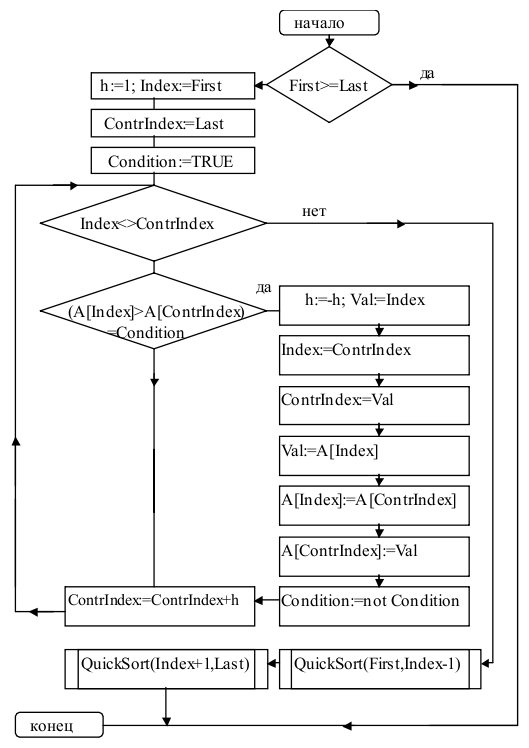
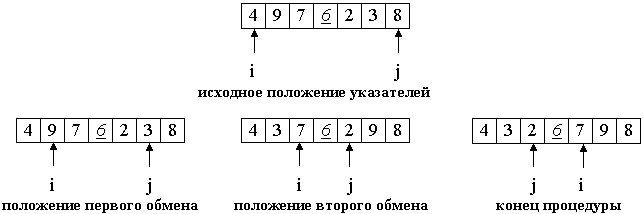


Рис. 1. Блок-схема быстрой сортировки

Рассмотрим работу процедуры для массива a[0]...a[6] и опорного элемента p = a[3].



Теперь массив разделен на две части: все элементы левой меньше либо равны p, все элементы правой - больше, либо равны p. Разделение завершено.

**Оценка эффективности:**

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного, в том числе, своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь меняются местами наиболее удалённые друг от друга элементы массива. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате самый эффективный улучшенный метод.

Лучший случай. Для этого алгоритма самый лучший случай — если в каждой итерации каждый из подмассивов делился бы на два равных по величине массива. В результате количество сравнений, делаемых быстрой сортировкой, было бы равно значению рекурсивного выражения CN = 2CN/2+N. Это дало бы наименьшее время сортировки.

Среднее. Даёт в среднем O(n lg n) обменов при упорядочении n элементов. В реальности именно такая ситуация обычно имеет место при случайном порядке элементов и выборе опорного элемента из середины массива либо случайно.

На практике быстрая сортировка значительно быстрее, чем другие алгоритмы с оценкой O(n lg n), по причине того, что внутренний цикл алгоритма может быть эффективно реализован почти на любой архитектуре. 2CN/2 покрывает расходы по сортировке двух полученных подмассивов; N — это стоимость обработки каждого элемента, используя один или другой указатель. Известно также, что примерное значение этого выражения равно CN = N lg N.

Худший случай. Худшим случаем, очевидно, будет такой, при котором на каждом этапе массив будет разделяться на вырожденный подмассив из одного опорного элемента и на подмассив из всех остальных элементов. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых.

Худший случай даёт O(n²) обменов, но количество обменов и, соответственно, время работы — это не самый большой его недостаток. Хуже то, что в таком случае глубина рекурсии при выполнении алгоритма достигнет n, что будет означать n-кратное сохранение адреса возврата и локальных переменных процедуры разделения массивов. Для больших значений n худший случай может привести к исчерпанию памяти во время работы алгоритма. Впрочем, на большинстве реальных данных можно найти решения, которые минимизируют вероятность того, что понадобится квадратичное время.

**Наглядный пример работы алгоритма:** http://upload.wikimedia.org/wikipedia/commons/6/6a/Sorting\_quicksort\_anim.gif

**Литература:**

Дональд Кнут Искусство программирования, том 3. Сортировка и поиск. 2-е изд. — М.: «Вильямс», 2007.

### 2.4. Обменная поразрядная сортировка

Данный метод использует двоичное представление ключей. Файл сортируется последовательно по битам двоичного представления ключей, начиная со старшего. Ключи, имеющие значение данного бита, равное нулю, ставятся в левую половину файла, а ключи со значением бита 1 - в правую. Ниже приведена схема алгоритма поразрядной сортировки.

l=1; r=N; b=1;

║

╔══>═╬══<══════════════════════════════════════════════╗

║ ║ ╔═══════════<═══════════════════════════╬═══════════════╗

║ ║ ┌──╫─────────────────────────────────────┐ ║ ║

║ ║ │ if(стек пуст) Конец; │ ║ ║

║ if(l=r) │ ┌──────────────────────────────┐ │ ║ ║

║ │ else │l=r+1; │ │ ║ ║

║ │ │Взять из стека элемент (r',b')╞>═╪═╝ ║

║ │ │r=r'; b=b'; │ │ ║

║ │ └──────────────────────────────┘ │ ║

║ └────────────────────────────────────────┘ ║

║ i=l; j=r; ║

║ ┌────────────────────────────────────────┐ ║

║ ╔═if(b(K[i])=1)│j=j-1;══════════<═══════════╗ │ ║

║ ║ │ ┌──────────────────╫─────────┐ │ ║

║ ║ │if(i<=j) │ if(b(K[j+1])=1) ═╝ │ │ ║

║ ║ │ │ ┌───────────────┐ │ │ ║

║ ║ │ │ else │ K[i]<->K[j+1] ╞>═╗ │ │ ║

║ ║ │ │ └───────────────┘ ║ │ │ ║

║ ║ │else ═>╗ └─────────────────────────╫──┘ │ ║

║ ║ └───────╫───────────────────────────╫────┘ ║

║ ╚════<═════════════╗ ║ ║ ║

║ ┌──────────╫───╨───────────────────────────╨────────────┐ ║

║ else │ i=i+1; ║ │ ║

║ │ if(i<=j)═╝ │ ║

║ │ ┌─────────────────────────────────────────────┐ │ ║

║ │ │ b=b+1; │ │ ║

║ │ else │ if(b>m) ═>══════════════════════════════════╪══╪════╝

╚════<════╪══════╪══════════════════╦═<═╦══════════════<═╗ │ │

│ │ if(j<l или j=r)═>╝ ║ ║ │ │

│ │ ┌────────────── ╫────────────────╫───┐ │ │

│ │ else │if(j=l) l=l+1═>╝ ║ │ │ │

│ │ │ ┌────────────────────────┐ ║ │ │ │

│ │ │else │ Поместить в стек (r,b);╞>╝ │ │ │

│ │ │ │ r=j; │ │ │ │

│ │ │ └────────────────────────┘ │ │ │

│ │ └────────────────────────────────────┘ │ │

│ └─────────────────────────────────────────────┘ │

└───────────────────────────────────────────────────────┘

Рассматриваемый алгоритм существенно отличается от других.

Во-первых, он совсем не использует сравнений сортируемых элементов.

Во-вторых, ключ, по которому происходит сортировка, необходимо разделить на части, разряды ключа. Например, слово можно разделить по буквам, 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

При сортировке русских слов m = 33, так как буква может принимать не более 33 значений. Если в самом длинном слове 10 букв, k = 10.

Аналогично, для шестнадцатиричных чисел m=16, если в качестве разряда брать цифру, и m=256, если использовать побайтовое деление.

Эти параметры нельзя изменять в процессе работы алгоритма. В этом - еще одно отличие метода от остальных.

**Список используемых источников:**

1. http://algolist.manual.ru/sort/radix\_sort.php

2. Роберт Седжевик – Фундаментальные Алгоритмы С++

3. http://wiki-linki.ru/Page/190151

4. http://dmtsoft.ru/bn/345/as/oneaticleshablon/

5. http://ru.wikibooks.org/wiki/Примеры\_реализации\_поразрядной\_сортировки

### 2.5. Параллельная сортировка Бэтчера

Суть – происходит слияние пар отсортированных подпоследовательностей. Проблема – как формировать эти пары.

Алгоритм сортировки Бэтчера не является наиболее эффективным алгоритмом сортировки, однако он обладает одним важным компенсирующим качеством: все сравнения и/или обмены, определяемые данной итерацией алгоритма можно выполнять одновременно. С такими параллельными операциями сортировка осуществляется за image1

Например, 1024 элемента можно рассортировать методом Бэтчера всего за 55 параллельных шагов. Схема сортировки Бэтчера несколько напоминает сортировку Шелла, но сравнения выполняются по-новому, а потому цепочки операций обмена записей не возникает. Поскольку в алгоритме Бэтчера, по существу, происходит слияние пар рассортированных подпоследовательностей, его можно назвать обменной сортировкой со слиянием.

Алгоритм Бэтчера (обменная сортировка со слиянием). Записиimage2 перекомпоновываются в пределах того же пространства в памяти. После завершения сортировки их ключи будут упорядочены:image3 Предполагается, что image4

1)[начальная установка image5.] Установить image6 , где image7— наименьшее целое число, такое, что image8. (Шаги 2-5 будут выполняться с image9 .)

2)[начальная установка image10 .] Установить image11.

3)[цикл по image12.] Для всех image12 , таких, что image13 и image14, выполнять шаг 4) Затем перейти к шагу 5. (Здесь через image14 обозначена операция "поразрядное логическое И" над представлениями целых чисел image12 и image5 ; все биты результата равны 0, кроме тех битов, для которых в соответствующих разрядах image12 и image5 находятся 1. Так image15.

К этому моменту d — нечётное кратное p (т.е. частное от деления d на p нечётно), а p — степень двойки, так что image17. Отсюда следует, что шаг 4 можно выполнять при всех нужных значениях I в любом порядке или даже одновременно.)

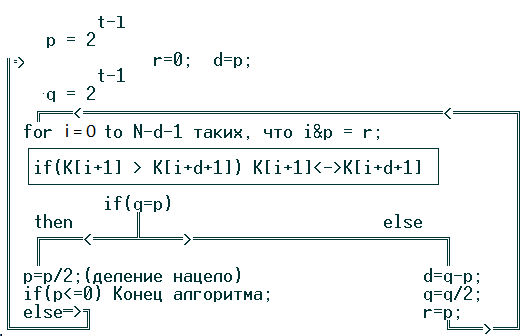
4)[Сравнение/обмен image18.] Если image19 , поменять местами записи image20 .

5)[Цикл по q.] Если q != p , установить image23 и возвратиться к шагу 3.

6)[Цикл по p ] (К этому моменту перестановка image24 будет p -упорядочена.) Установить image25 . Если p>0 , возвратиться к шагу 2.

Для получения алгоритма обменной сортировки, время работы которого меньше, чем необходимо выбирать для сравнения и обмена ключи, расположенные возможно дальше друг от друга. Эта идея уже была реализована в алгоритме сортировки Шелла вставок с убывающим шагом, однако в данном алгоритме сравнения выполняются по-другому.

Рассмотрим схему алгоритма. Здесь t= [logN] - наименьшее целое, равное или превышающее logN ( N>=2). Переменная d имеет смысл шага сортировки, то есть сравниваются ключи K[i] и K[i+d]. Через <-> обозначена операция обмена значений двух переменных.

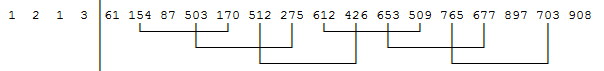
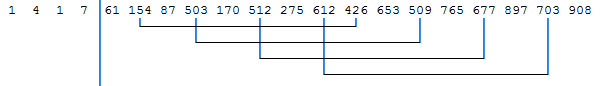
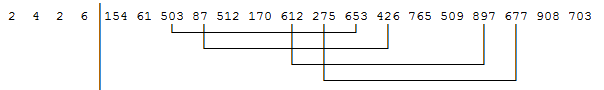
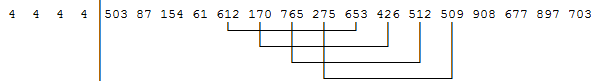
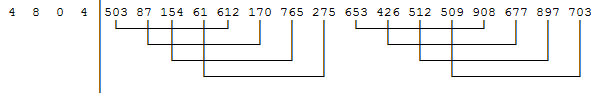
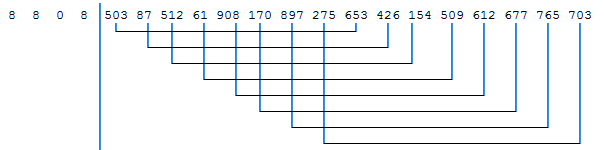


На схеме алгоритма через & обозначена операция поразрядного И, то есть выполнение условия i&p = r означает, что в цикле по i нужно выбирать только такие значения i, которые в одном разряде, изменяющемся в цикле по p (операция p=p/2 имеет смысл сдвига начального значения 1 в разряде t-1), имеют сначала значение 0 (поскольку r=0), а затем 1 (поскольку r=p).

Проиллюстрируем работу алгоритма примером исходного файла из 16 элементов. Линиями соединены сравниваемые на данном шаге ключи.При d=1 происходит фактически слияние упорядоченных подфайлов K[1],K[3],K[5], ... и K[2],K[4],K[6], ... , то есть сортировку Бэтчера можно назвать также обменной сортировкой со слиянием.

Время работы алгоритма t примерно оценивается формулой: C:\Users\fanta239\Desktop\1111.pngгде a, b - неизвестные константы, зависящие от программной реализации алгоритма.





**Список используемых источников:**

1. http://pascal.proweb.kz/index.php?page=79

2. http://waidos32.narod.ru/otvet/1\_44.html

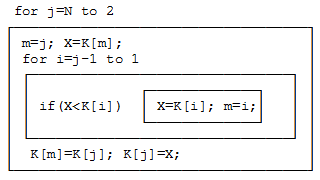
3.http://techn.sstu.ru/TFI/site\_tfi/TFI/PVS/material/murashev/programming/term2/books/sort.htm#\_Toc38711143

4. http://www.viva64.com/ru/a/0032/

5. http://saod.narod.ru/saod2/List012.html

## 3. Сортировка посредством выбора

Идея метода довольно проста: найти наибольший элемент файла и поставить его на место N, найти следующий максимум и поставить его на место N-1 и т.д. до 2-го элемента. Схема алгоритма имеет следующий вид.



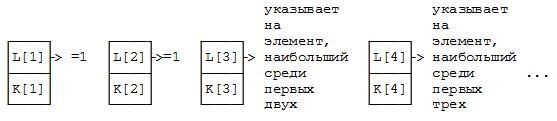
Время работы алгоритма t примерно оценивается формулой:

t=a\*N^2 + b\*N\*lgN

где a,b - неизвестные константы, зависящие от программной реализации алгоритма.

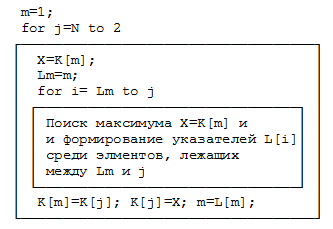
### 3.1. Использование связанного списка для хранения информации о промежуточных максимумах

В приведенном выше алгоритме максимум среди K[1] ... K[j-1] определяется в цикле от j-1 до 1 c целью обеспечить меньшее число обменов в случае равенства ключей и сохранении прежнего порядка равных элементов. Однако, если изменить порядок просмотра элементов на противоположный и изменить структуру данных, введя дополнительные указатели, можно примерно в два раза сократить число повторений в цикле поиска максимума. Каждый ключ K[i] снабжается указателем L[i] на элемент, максимальный среди первых i-1 элементов так, как показано ниже.



Тогда после обмена элементов K[j] и K[m] поиск следующего максимума можно осуществлять среди элементов K[L[j]] ... K[j-1], т.к. максимум может "обновиться" только за счет элементов, лежащих правее локального максимума. Таким образом среднее количество просматриваемых при поиске максимума элементов сокращается примерно в два раза.

Модифицированный алгоритм имеет следующий вид.



Время работы алгоритма t примерно оценивается формулой:

t=a\*N^2 + b\*N\*lgN

где a,b - неизвестные константы, зависящие от программной реализации алгоритма.

### 3.2. Пирамидальная сортировка

Это алгоритм сортировки массива произвольных элементов; требуемый им дополнительный объём памяти не зависит от количества исходных данных. Время работы алгоритма — O\left(n\cdot\ln n\right) в среднем, а так же в лучшем и худшем случаях.

Идея алгоритма:

• Пирамида — двоичное дерево, в котором значение каждого элемента больше либо равно значений дочерних элементов.

• Заполнив дерево элементами в произвольном порядке, можно легко его отсортировать, превратив в пирамиду.

• Самый большой элемент пирамиды находится в её вершине.

• Отделяем вершинный элемент, и записываем его в конец результирующего массива.

• На место вершинного элемента записываем элемент из самого нижнего уровня дерева.

• Восстанавливаем (пересортировываем) пирамиду.

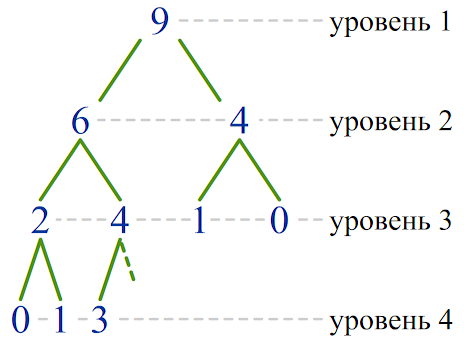
• Самый большой элемент из оставшихся снова в вершине. Снова отделяем его и записываем его в качестве предпоследнего элемента результата, и так далее...

• Весь фокус алгоритма в том, что пирамида без дополнительных затрат хранится прямо в исходном массиве. По мере того, как размер пирамиды уменьшается, она занимает всё меньшую часть массива, а результат сортировки записывается, начиная с конца массива на освободившиеся от пирамиды места.

**Пирамида**

Воз­рас­таю­щей пи­ра­ми­дой на­зы­ва­ет­ся по­чти за­пол­нен­ное де­ре­во, в ко­то­ром зна­че­ние каж­до­го эле­мен­та боль­ше ли­бо рав­но зна­че­ний всех его по­том­ков. Ана­ло­гич­но, в убы­ваю­щей пи­ра­ми­де зна­че­ние каж­до­го эле­мен­та мень­ше ли­бо рав­но зна­че­ний по­том­ков.

При­мер воз­рас­таю­щей пи­ра­ми­ды по­ка­зан на ри­сун­ке:



Оче­вид­но, са­мое боль­шое зна­че­ние в воз­рас­таю­щей пи­ра­ми­де име­ет кор­не­вой эле­мент. Од­на­ко, из свойств пи­ра­ми­ды не сле­ду­ет, что зна­че­ния эле­мен­тов умень­ша­ют­ся с уве­ли­че­ни­ем уров­ня. В част­но­сти, на ри­сун­ке 4 эле­мент со зна­че­ни­ем 3 на чет­вёр­том уров­не боль­ше зна­че­ний всех эле­мен­тов, кро­ме од­но­го, на тре­тьем уров­не.

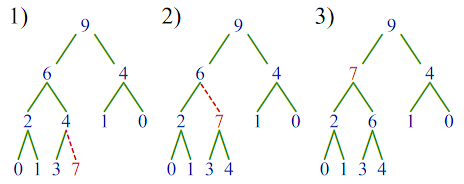
Важ­ной опе­ра­ци­ей яв­ля­ет­ся от­де­ле­ние по­след­не­го (в смыс­ле ну­ме­ра­ции) эле­мен­та от пи­ра­ми­ды. Не­труд­но до­ка­зать, что в этом слу­чае дво­ич­ное де­ре­во оста­ёт­ся по­чти за­пол­нен­ным, и все свой­ства пи­ра­ми­ды со­хра­ня­ют­ся.

**Просеивание вверх**

Рас­смот­рим те­перь за­да­чу при­со­еди­не­ния эле­мен­та с про­из­воль­ным зна­че­ни­ем к воз­рас­таю­щей пи­ра­ми­де. Ес­ли про­сто до­ба­вить эле­мент в ко­нец мас­си­ва, то свой­ство пи­ра­ми­ды (зна­че­ние лю­бо­го эле­мен­та зна­че­ния его ро­ди­те­ля) мо­жет быть на­ру­ше­но. Для вос­ста­нов­ле­ния свой­ства пи­ра­ми­ды к до­бав­лен­но­му эле­мен­ту при­ме­ня­ет­ся про­це­ду­ра про­сеи­ва­ния вверх, ко­то­рая опи­сы­ва­ет­ся сле­дую­щим ал­го­рит­мом:

1. если элемент корневой, или его значение \leqslant значения родителя, то конец;
2. меняем местами значения элемента и его родителя;
3. переходим к родителю, и выполняем для него этот же алгоритм, начиная с пункта 1.

Пример просеивания вверх добавленного элемента показан на рисунке:



По­сле вы­пол­не­ния дан­ной про­це­ду­ры свой­ство пи­ра­ми­ды бу­дет вос­ста­нов­ле­но, так как:

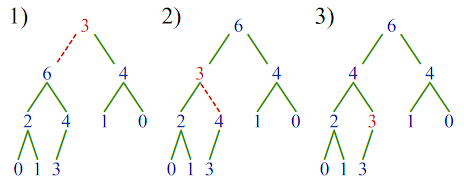
* Ес­ли вновь до­бав­лен­ный эле­мент боль­ше ро­ди­те­ля, то он боль­ше и вто­ро­го по­том­ка ро­ди­те­ля (ес­ли вто­рой по­то­мок есть), так как до до­бав­ле­ния но­во­го эле­мен­та бы­ло вы­пол­не­но свой­ство пи­ра­ми­ды, и ро­ди­тель был не мень­ше сво­е­го по­том­ка (на ри­сун­ке 5 сле­ва: се­мёр­ка боль­ше сво­е­го ро­ди­те­ля — чет­вёр­ки, — по­это­му она боль­ше и вто­ро­го по­том­ка чет­вёр­ки — трой­ки). По­сле об­ме­на зна­че­ний эле­мен­та и ро­ди­те­ля свой­ство пи­ра­ми­ды бу­дет вос­ста­нов­ле­но в под­де­ре­ве, кор­нем ко­то­ро­го яв­ля­ет­ся ро­ди­тель с но­вым зна­че­ни­ем (на­при­мер, по­сле об­ме­на ме­ста­ми се­мёр­ки и чет­вёр­ки, се­мёр­ка трой­ка и чет­вёр­ка бу­дут об­ра­зо­вы­вать пра­виль­ное под­де­ре­во).
* В ре­зуль­та­те об­ме­на свой­ство пи­ра­ми­ды мо­жет быть на­ру­ше­но в от­но­ше­нии ро­ди­те­ля и ро­ди­те­ля ро­ди­те­ля (так как зна­че­ние ро­ди­те­ля ста­ло боль­ше). Про­це­ду­ра вы­зы­ва­ет­ся для ро­ди­тель­ско­го уз­ла, что­бы вос­ста­но­вить это свой­ство.

**Просеивание вниз**

Что де­лать, ес­ли нам нуж­но за­ме­нить кор­не­вой эле­мент на ка­кой-ли­бо дру­гой? В этом слу­чае пи­ра­ми­даль­ную ст­рук­ту­ру дво­ич­но­го де­ре­ва мож­но вос­ста­но­вить с по­мо­щью про­це­ду­ры про­сеи­ва­ния вниз:

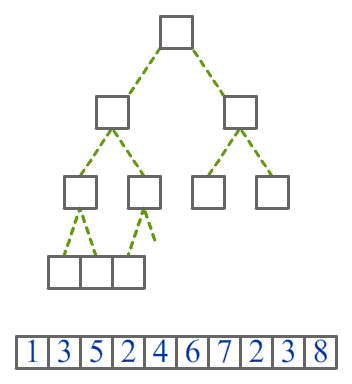
1. если элемент листовой, или его значение \geqslant значений потомков, то конец;
2. иначе меняем местами значения элемента и его потомка, имеющего максимальное значение;
3. переходим к изменившемуся потомку, и выполняем для него этот же алгоритм, начиная с пунк­та 1.

При­мер про­сеи­ва­ния вниз по­ка­зан на ри­сун­ке:



**Построение пирамиды**

До­пу­стим, у нас есть про­из­воль­ный на­бор из n эле­мен­тов, и мы хо­тим по­стро­ить из не­го пи­ра­ми­ду. Са­мый про­стой спо­соб это­го до­бить­ся — на­чать с пу­стой пи­ра­ми­ды, и до­бав­лять в неё эле­мен­ты один за дру­гим, каж­дый раз вы­зы­вая про­це­ду­ру про­сеи­ва­ния вверх для но­во­го эле­мен­та. На­зо­вём этот ме­тод по­строе­ния пи­ра­ми­ды «ме­то­дом про­сеи­ва­ния вверх». При­мер ра­бо­ты ал­го­рит­ма по­ка­зан на ри­сун­ке:



Рас­счи­та­ем быст­ро­дей­ствие это­го ме­то­да в са­мом пло­хом слу­чае. Пусть мас­сив уже от­сор­ти­ро­ван по воз­рас­та­нию, и каж­дый до­бав­ляе­мый эле­мент вы­нуж­ден про­сеи­вать­ся вверх до са­мо­го кор­ня. То­гда при до­бав­ле­нии эле­мен­та но­мер i нам по­тре­бу­ет­ся O\left(\ln\left(i+1\right)\right) операций. Мы делаем эту процедуру для всех i от 0 до n-1:

|  |  |
| --- | --- |
| T_1\left(n\right)=\sum_{i=0}^{n-1}{O\left(\ln\left(i+1\right)\right)}=O\left(\sum_{i=2}^n{\ln i}\right). | (1) |

Оценим сумму в выражении (1) с помощью определённого интеграла:

|  |  |
| --- | --- |
| \int\limits_2^{n+1}{\ln\left(x-1\right)dx} < \sum_{i=2}^n{\ln i} < \int\limits_2^{n+1}{\ln x dx}. | (2) |

Вычисляя интегралы, получаем:

|  |  |
| --- | --- |
| n\ln n -n+1 < \sum_{i=2}^n{\ln i} < \left(n+1\right)\cdot\ln \left(n+1\right) -n-2\ln2+1. | (3) |

Это означает, что:

|  |  |
| --- | --- |
| T_1\left(n\right)=O\left(n\cdot\ln n\right). | (4) |

На са­мом де­ле пи­ра­ми­ду мож­но по­стро­ить быст­рее. Для это­го за­пол­ним де­ре­во эле­мен­та­ми в про­из­воль­ном по­ряд­ке (на­при­мер, так, как они идут в мас­си­ве из­на­чаль­но), и бу­дем его ис­прав­лять.

За­ме­тим, что ес­ли у нас есть две пи­ра­ми­ды, то, ес­ли их со­еди­нить об­щим кор­нем, а за­тем про­се­ять ко­рень вниз, то мы по­лу­чим но­вую боль­шую пи­ра­ми­ду. Де­ре­вья, со­сто­я­щие из од­но­го эле­мен­та, за­ве­до­мо яв­ля­ют­ся пи­ра­ми­да­ми, по­это­му нач­нём с на­бо­ра ли­сто­вых эле­мен­тов, а за­тем бу­дем со­еди­нять имею­щие­ся пи­ра­ми­ды по­пар­но до тех пор, по­ка не по­лу­чим од­ну боль­шую пи­ра­ми­ду. Пер­вый до­бав­ляе­мый ко­рень име­ет ин­декс  n_0=\left\lfloor n/2 \right\rfloor-1. Алгоритм следующий: «для всех i от n_0 до 0 вызываем процедуру просеивания элемента вниз». Назовём этот метод построения пира­миды «методом просеивания вниз».

Вы­чис­лим быст­ро­дей­ствие это­го ал­го­рит­ма в худ­шем слу­чае. Пусть мас­сив от­сор­ти­ро­ван по воз­рас­та­нию, и каж­дый до­бав­ляе­мый ко­рень дол­жен быть про­се­ян в са­мый низ стро­я­щей­ся пи­ра­ми­ды. Вы­со­та де­ре­ва при­мер­но рав­на \log_2n, глу­би­на за­ле­га­ния эле­мен­та но­мер i в дереве равна \log_2i, поэтому он будет просеиваться вниз \log_2n-\log_2i шагов. Тогда общее время построения пирамиды будет равно:

|  |  |
| --- | --- |
| T_2\left(n\right)=O\left(\sum_{i=n/2}^1{\left(\ln n-\ln i\right)}\right). | (5) |

Оценивая сумму интегралами аналогично (2), (3), получим:

|  |  |
| --- | --- |
| T_2\left(n\right)=O\left(n\right). | (6) |

Отсюда следует, что для большого числа элементов T_2\left(n\right) будет меньше, чем T_1\left(n\right). Более точ­ные оценки дадут, что T_2\left(n\right) < T_1\left(n\right) для любых n, так как, во пер­вых, при по­строе­нии пи­ра­ми­ды ме­то­дом про­сеи­ва­ния вверх про­це­ду­ра про­сеи­ва­ния вы­зы­ва­ет­ся в 2 ра­за боль­ше раз, чем при по­строе­нии ме­то­дом про­сеи­ва­ния вниз, и, во вто­рых, эле­мен­ты с но­ме­ра­ми от  n/2 до n про­сеи­ва­ют­ся вверх на боль­шее рас­стоя­ние, чем рас­стоя­ние, на ко­то­рое про­сеи­ва­ют­ся вниз эле­мен­ты от от n/2до 0. Поэтому в дальнейшем будем строить пирамиду методом просеивания вниз.

**Сортировка**

Имея по­стро­ен­ную пи­ра­ми­ду, не­слож­но реа­ли­зо­вать сор­ти­ров­ку. Так как кор­не­вой эле­мент пи­ра­ми­ды име­ет са­мое боль­шое зна­че­ние, мы мо­жем от­де­лить его и по­ме­стить в от­сор­ти­ро­ван­ный спи­сок. Вме­сто от­сут­ствую­ще­го кор­не­во­го эле­мен­та мож­но по­ста­вить по­след­ний (в смыс­ле ну­ме­ра­ции) эле­мент де­ре­ва и, про­се­яв его вниз, сно­ва по­лу­чить пи­ра­ми­ду.

В но­вой умень­шен­ной пи­ра­ми­де ко­рень име­ет са­мое боль­шое зна­че­ние сре­ди остав­ших­ся эле­мен­тов. Его сно­ва мож­но от­де­лить и по­ме­стить в от­сор­ти­ро­ван­ный спи­сок пе­ред имею­щи­ми­ся там эле­мен­та­ми, и так да­лее.

Для хра­не­ния от­сор­ти­ро­ван­но­го спис­ка бу­дем ис­поль­зо­вать эле­мен­ты ис­ход­но­го мас­си­ва, осво­бож­даю­щие­ся в ре­зуль­та­те умень­ше­ния раз­ме­ра пи­ра­ми­ды. Ал­го­ритм по­лу­ча­ет­ся сле­дую­щий:

1. По­ме­нять ме­ста­ми зна­че­ния пер­во­го и по­след­не­го эле­мен­тов пи­ра­ми­ды.

2. От­де­лить по­след­ний эле­мент от де­ре­ва, умень­шив раз­мер де­ре­ва на еди­ни­цу (эле­мент оста­ёт­ся в мас­си­ве).

3. Вос­ста­но­вить пи­ра­ми­ду, про­се­яв вниз её но­вый кор­не­вой эле­мент.

4. Пе­рей­ти к пунк­ту 1.

5. По ме­ре ра­бо­ты ал­го­рит­ма, часть мас­си­ва, за­ня­тая де­ре­вом, умень­ша­ет­ся, а в кон­це мас­си­ва на­кап­ли­ва­ет­ся от­сор­ти­ро­ван­ный ре­зуль­тат.

6. Тео­ре­ти­че­ское вре­мя ра­бо­ты это­го ал­го­рит­ма не­труд­но оце­нить, ес­ли за­ме­тить, что пи­ра­ми­даль­ная сор­ти­ров­ка (без учё­та на­чаль­но­го по­строе­ния пи­ра­ми­ды) пол­но­стью ана­ло­гич­на по­строе­нию пи­ра­ми­ды ме­то­дом про­сеи­ва­ния вверх, толь­ко про­из­во­дит­ся «в об­рат­ном по­ряд­ке»: пи­ра­ми­да не стро­ит­ся, а «раз­би­ра­ет­ся» элемент за эле­мен­том, и эле­мен­ты про­сеи­ва­ют­ся не вверх, а вниз. По­это­му вре­мя ра­бо­ты ал­го­рит­ма пи­ра­ми­даль­ной сор­ти­ров­ки  T_3 без учёта времени построения пи­рамиды будет определяться аналогично формуле (4):

|  |  |
| --- | --- |
| T_3\left(n\right)=O\left(n\cdot\ln n\right). | (7) |

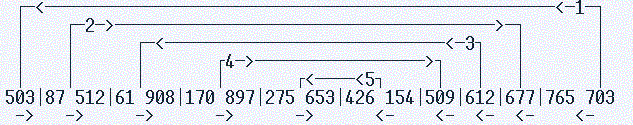
7. Тогда общее время сортировки (с учётом построения пирамиды) будет равно:

|  |  |
| --- | --- |
| T\left(n\right) = T_2\left(n\right)+T_3\left(n\right) = O\left(n\right)+O\left(n\cdot\ln n\right) = O\left(n\cdot\ln n\right). | (8) |

## 4. Сортировка посредством слияния

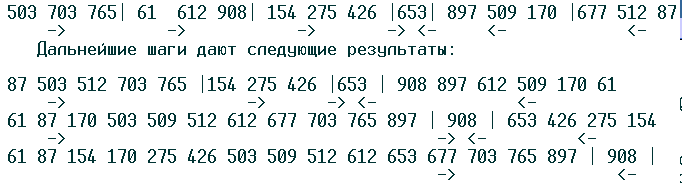
### 4.1. Естественное двухпутевое слияние

Этот алгоритм ищет упорядоченные отрезки с двух концов файла и переписывает их по очереди также в оба конца(слово «Естественное» указывает на то, что мы используем уже имевшиеся в файле последовательности элементов). Повторяя эту процедуру в цикле, мы приходим к середине файла, что означает окончание сортировки. Проиллюстрируем работу алгоритма на примере файла из 16 элементов:



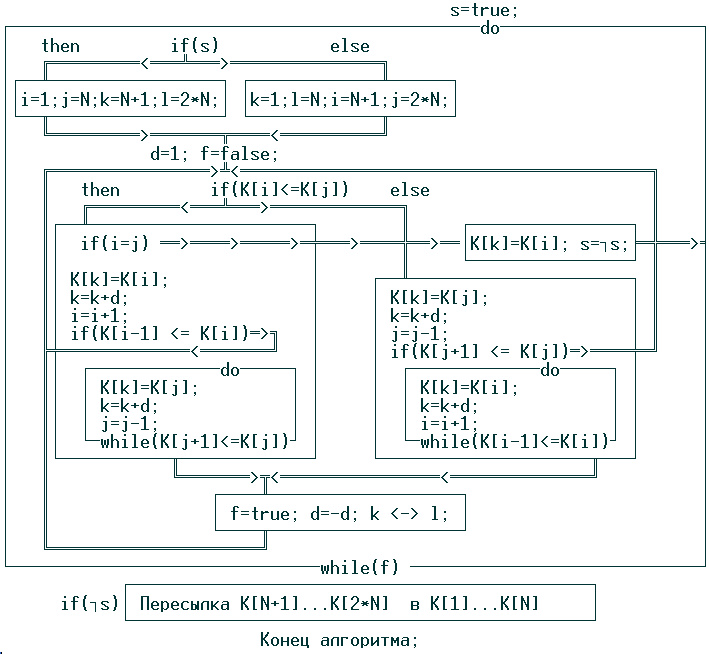
Черточками разделены упорядоченные отрезки, стрелками показаны направления упорядочения внутри отрезков. На первом шаге сливаются отрезки 503 слева и 703 765 справа в один отрезок, который записывается в левый конец файла, на втором шаге сливаются отрезки 87 512 слева и 677 справа, которые записываются в правый конец файла и т.д. Это показано на рисунке линиями со стрелками и номерами действий.

В результате файл принимает следующий вид:



Рассмотрим алгоритм. Для хранения второго "экземпляра" файла образована новая область, располагающаяся вслед за исходной областью K[1], K[2], ... K[N] c индексами от N+1 до 2\*N: K[N+1] ... K[2\*N].

Элементы файла пересылаются из одной области в другую, меняя направление пересылки. Для запоминания направления пересылки служит переменная s, принимающая значения TRUE и FALSE попеременно. Другой логический признак f служит сигналом продолжения-окончания алгоритма, если все области слились в конце концов в одну. Переменная d принимает попеременно значения +1 -1 и указывает направление просмотра файла: вперед или назад. Операция <-> обозначает обмен значениями двух переменных. Операция обозначает инверсию логической переменной или выражения.



Время работы алгоритма t примерно оценивается формулой:

t=a\*N\*lgN + b\*N

где a, b - неизвестные константы, зависящие от программной реализации алгоритма.

### 4.2. Простое двухпутевое слияние

В алгоритме естественного двухпутевого слияния упорядоченные отрезки файла определялись случайным расположением элементов в исходном файле. В данном алгоритме длина отрезков фиксируется на каждом шаге. В исходной файле все отрезки имеют длину 1, после первого шага она равна 2, после второго 4, после третьего - 8, после к-го шага – 2K. Иллюстрация работы алгоритма приведена ниже.

Исходный файл имеет вид:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 503 | 87 | 512 | 61 | 908 | 170 | 897 | 275 | 653 | 426 | 154 | 509 | 612 | 677 | 765 | 703 |

Чертами разделены отрезки. Стрелками обозначено направление упорядочивания. В дальнейшем файл преобразуется следующим образом:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 503 | 703 | 512 | 677 | 509 | 908 | 426 | 897 | 653 | 275 | 170 | 154 | 612 | 61 | 765 | 87 |
| → | | → | | → | | → | | ← | | ← | | ← | | ← | |
| 87 | 503 | 703 | 765 | 154 | 170 | 509 | 908 | 897 | 653 | 426 | 275 | 677 | 612 | 512 | 61 |
| → | | | | → | | | | ← | | | | ← | | | |
| 61 | 87 | 503 | 512 | 612 | 677 | 703 | 765 | 908 | 897 | 653 | 509 | 426 | 275 | 170 | 154 |
| → | | | | | | | | ← | | | | | | | |
| 61 | 87 | 154 | 170 | 275 | 426 | 503 | 509 | 512 | 612 | 653 | 677 | 703 | 765 | 897 | 908 |
| → | | | | | | | | | | | | | | | |

Рассмотрим алгоритм:

Для хранения второго "экземпляра" файла образовывается новая область, располагающаяся вслед за исходной областью K[1], K[2], … K[N] c индексами от N+1 до 2N: K[N+1]…K[2N].

Элементы файла пересылаются из одной области в другую, меняя направление пересылки. Для запоминания направления пересылки служит переменная s, принимающая значения TRUE и FALSE попеременно. Переменная p имеет значение размера отрезков, которые будут сливаться на предстоящем шаге, q и r - количества еще неслитых элементов в отрезках. Переменная d принимает попеременно значения +1, -1 и указывает направление просмотра файла: вперед или назад. Операция <-> обозначает обмен значениями двух переменных. Операция ┐ обозначает инверсию логической переменной или выражения.

Алгоритм имеет следующий вид:

s=true; p=1;

┌─────────────────────────────────────────────────do───────────────────┐

│ then if(s) else │

│ ╔═════════<═══╩═══>═══════════════╗ │

│ ┌──╨─────────────────┐ ┌─────────────╨──────┐ │

│ │i=1;j=N;k=N+1;l=2\*N;│ │k=1;l=N;i=N+1;j=2\*N;│ │

│ └──╥─────────────────┘ └─────────────╥──────┘ │

│ ╚═════════>═══════╦════<══════════╝ │

│ d=1; q=p; r=p; │

│ ╔══════════════>══╬══<═══════════════════════════════════════╗ │

│ ║ if(K[i] <= K[j]) ║ │

│ ║ then ║ else ║ │

│ ║ ╔═════════<═══╩═══>═════════════╗ ║ │

│ ║┌──╨──────────────────────┐ ┌──╨────────────────┐ ║ │

│ ║│ k=k+d; │ │ k=k+d; │ ║ │

│ ║│ K[k]=K[i]; │ │ K[k]=K[j]; │ ║ │

│ ║│ i=i+1; │ │ j=j-1; │ ║ │

│ ║│ q=q-1; │ │ r=r-1; │ ║ │

│ ║│ if(q>0)════════════>╗ │ │ if(r>0)═══════════╪═══════╝ │

│ ╠╪═════════════<═══════╝ │ │ ┌──────────do──┐ │ │

│ ║│ ┌─────────do───┐ │ │ │ k=k+d; │ │ │

│ ║│ │ k=k+d; │ │ │ │ if(k=l)═>══╗ │ │ ┌──────┐ │

│ ║│ │ if(k=l) ══>══╪═══════╪══>══╪══╪═════════>══╩═╪═╪══╡p=p+p;╞═>╡

│ ║│ │ K[k]=K[j] │ │ │ │ K[k]=K[i] │ │ │s=┐s; │ │

│ ║│ │ j=j-1;r=r-1; │ │ │ │ i=i+1; q=q-1;│ │ └──────┘ │

│ ║│ └while(r>0)────┘ │ │ └while(q>0)────┘ │ │

│ ║└───────────╥─────────────┘ └───────────╥───────┘ │

│ ║ ╚═══════>╦<════════════════<════╝ │

│ ║ ┌────╨───────────────────┐ │

│ ║ │ q=p;r=p; d=-d; k<->l; │ │

│ ║ └────╥───────────────────┘ │

│ ╚═══════<═════════════╝ │

└────────────────────────────────────────while(p<N)────────────────────┘

┌────────────────────────────────────┐

if(s) │ Пересылка K[N+1]…K[2N] в K[1]…K[N] │

└────────────────────────────────────┘

Конец алгоритма

Время работы алгоритма t примерно оценивается формулой: t = a⋅N⋅lgN + b⋅N, где a, b - неизвестные константы, зависящие от программной реализации алгоритма.

### 4.3. Слияние связанных списков

В процессе работы алгоритмов 4.1 и 4.2 половина памяти постоянно не используется, а остается зарезервированной для последующего заполнения. Более логичной структурой данных является связанный список. В этом случае все манипуляции с данными заменяются операциями со связями, присутствующими в каждом элементе файла. Затраты памяти в этом случае те же, что и при добавлении второго пространства памяти под исходный файл, однако время работы может быть меньше, так как записи не перемещаются. Файл данных содержит ключи K[1]…K[N] и поля связи L[1]…L[N], в которых могут храниться числа от -(N+1) до +(N+1). В начале и в конце файла имеются искусственные элементы K[0], L[0] и K[N+1], L[N+1]. После завершения сортировки L[0] указывает на наименьший ключ, для записи к с наибольшим ключом L[k]=0. В начале каждого шага алгоритма K[0] и K[N+1] являются головами двух подсписков, которые в данный момент сливаются. Отрицательная связь означает конец списка, о котором известно, что он упорядочен. Нулевая связь означает конец последнего списка. Через │L[s]│=p обозначена операция присваивания значения p с сохранением старого знака переменной L[s].

Иллюстрация работы алгоритма дана для файла из 16 элементов:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| K[j] | - | 503 |  | 512 |  | 908 |  | 897 |  | 653 |  | 154 |  | 612 |  | 765 |  | - |
|  |  |  | 87 |  | 61 |  | 170 |  | 275 |  | 426 |  | 509 |  | 677 |  | 703 |  |
| L[j] | 1 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 | -11 | -12 | -13 | -14 | -15 | -16 | 0 | 0 | 2 |
| L[j] | 2 | -6 | 1 | -8 | 3 | -10 | -5 | -11 | 7 | -13 | 9 | 12 | -16 | 14 | 0 | 0 | 15 | 4 |
| L[j] | 4 | 3 | 1 | -11 | 2 | -13 | 8 | 5 | 7 | 0 | 12 | 10 | 9 | 14 | 16 | 0 | 15 | 6 |
| L[j] | 4 | 3 | 6 | 7 | 2 | 0 | 8 | 5 | 1 | 14 | 12 | 10 | 13 | 9 | 16 | 0 | 15 | 11 |
| L[j] | 4 | 12 | 11 | 13 | 2 | 0 | 8 | 5 | 10 | 14 | 1 | 6 | 3 | 9 | 16 | 7 | 15 | 0 |

Рассмотрим схему алгоритма:

L[0]=1; L[N-1]=L[N]=0; L(N+1)=2;

┌────────────┐

for i=1 to N-2 │L[i]=-(i+2);│

└────────────┘

╔> s=0; t=N+1;

║ p=L[s]; q=L[t];

║ if(q=0) Конец алгоритма

║ ╔═════>══════════════════╦═════════<══════════════════╦══<═╗

║ ║ if(K[p]<=K[q]) ║ ║

║ ║ then ║ else ║ ║

║ ║ ╔═══════════════<═══╩════>════════════╗ ║ ║

║ ║ │L[s]│=p; │L[s]│=q; ║ ║

║ ║ s=p; p=L[p]; s=q; q=L[q]; ║ ║

║ ║ if(p>0) ═>╗ if(q>0) ═>═════╝ ║

║ ╚════<══════╝ ║

║ │L[s]│=q; L[s] =p; ║

║ s=t; s=t; ║

║ ╔═══<═════════╗ ╔═══<═════════╗ ║

║ t=q; q=L[q]; ║ t=p; p=L[p]; ║ ║

║ if (q>0) ═>════╝ if (p>0) ═>════╝ ║

║ else ═════>═════════════════════╗ else ═>═╗ ║

║ ╠══<══════════╝ ║

║ p=-p; ║

║ q=-q; ║

║ if(q=0) ║

║ then ║ else ║

║ ╔════════════════════<════╩════>══════════════════╝

║ │L[s]│=p; │L[t]│=0;

╚══════<════╝

Время работы алгоритма t примерно оценивается формулой: t = a⋅N⋅lgN + b⋅N, где a, b - неизвестные константы, зависящие от программной реализации алгоритма.

Этот метод допускает естественную модификацию, заключающуюся в том, что вместо начальной установки длин всех списков, равных 1 (оператор L[i]=-(i+2);), анализируется естественное упорядочивание участков исходного файла, и длины устанавливаются максимально возможными.

## 5. Распределяющая сортировка

Эта группа методов использует стратегию, прямо противоположную методам слияния. Ключи K[i] элементов файла рассматриваются в этом алгоритме как некоторые последовательности из s символов в известном алфавите K=(A1,A2,A3,...,As). Например, такой порядок используется для расположения слов в словаре. Если множество слов упорядочить сначала по правой букве, затем, не меняя порядка по правой букве, упорядочить их по следующей букве и т.д. до конца слова, то слова будут расположены в лексикографическом порядке. Можно начинать выбор букв как с конца, так и с начала слов. В процессе упорядочивания по букве множество слов разбивается на подмножества с одинаковыми значениями данной буквы, а затем подмножества снова объединяются в одно множество в порядке, соответствующем алфавитному порядку для значений этой буквы. Распределение множества по подмножествам и дало название этой группы методов.

р = АДР(K[N]); j = N;с

for k = 1 to s

for i = 0 to M

TOP[i] = АДР(BOTM[i]);

BOTM[i] = @

i = (k-я младшая буква ключа, на который указывает переменная р)

L[ЭЛЕМ(ТОР[i])] = p;

TOP[i] = p;

if (k = 1)

then:

p = L[ЭЛЕМ(p)];

else:

if(p != @)

then:

j = j – 1;

if(j != 0)

else:

p = TOP[0]

for i = 1 to M-1

if(BOTM[i] = @)

L[ЭЛЕМ(p)] = BOTM[i]

p = TOP[i]

then:

p = АДР(K[j]);

L[ЭЛЕМ(p)] = @

p = BOTM[0]

Алгоритм имеет следующий вид. Функция АДР(переменная) возвращает указатель на переменную. Функция ЭЛЕМ(указатель) позволяет выполнять операции над элементом файла, на который указывает аргумент функции. Подмножества, отсортированные по одной букве ключа, хранятся в виде М очередей, где М - количество возможных значений каждой буквы. Для каждой из М очередей существует два массива указателей: BOTM[0] ... BOTM[N] и TOP[0] ... TOP[M], указывающие соответственно на начала и концы очередей - подмножеств. BOTM[0] указывает в конце алгоритма на начало объединенного списка. Каждый элемент файла имеет поле ключа K[i] и поле связи L[i], указывающее на следующий элемент в отсортированном файле. Символом @ обозначен пустой указатель. Его конкретное значение может быть выбрано произвольно и использоваться в алгоритме только в этом смысле.

Работу алгоритма проиллюстрируем на примере файла из 16 элементов, сортировка будет проводится по цифрам ключей, начиная с младшей. Исходный файл:

503 87 512 61 908 170 897 275 653 426 154 509 612 677 765 703

Файл после 1-го прохода (k=1)

TOP[0] TOP[1] TOP[2] TOP[3] TOP[4] TOP[5] TOP[6] TOP[7] TOP[8] TOP[9]

509

908

87

462

275

154

503

512

170

61

612

653

765

897

677

703

BOTM[0] BOTM[1] BOTM[2] BOTM[3] BOTM[4] BOTM[5] BOTM[6] BOTM[7] BOTM[8] BOTM[9]

Файл после 2-го прохода (k=2)

TOP[0] TOP[1] TOP[2] TOP[3] TOP[4] TOP[5] TOP[6] TOP[7] TOP[8] TOP[9]

897

87

677

765

154

426

509

512

653

275

908

612

61

170

503

BOTM[0] BOTM[1] BOTM[2] BOTM[3] BOTM[4] BOTM[5] BOTM[6] BOTM[7] BOTM[8] BOTM[9]

Файл после 3-го прохода (k=3)

TOP[0] TOP[1] TOP[2] TOP[3] TOP[4] TOP[5] TOP[6] TOP[7] TOP[8] TOP[9]

908

897

765

677

512

275

87

170

426

509

703

61

154

653

503

612

BOTM[0] BOTM[1] BOTM[2] BOTM[3] BOTM[4] BOTM[5] BOTM[6] BOTM[7] BOTM[8] BOTM[9]

После 3-го прохода видно, что сцепление всех очередей от 0 до 9 дает отсортированный файл.

Время работы алгоритма t примерно оценивается формулой:

t=a\*N + b

где a, b - неизвестные константы, зависящие от программной реализации алгоритма.

**Примеры реализации:**

1. http://en.wikibooks.org/wiki/Algorithm\_implementation/Sorting/Radix\_sort

2. http://bitbucket.org/ais/usort/wiki

Визуализация:

1. http://wiki.algoviz.org/AlgovizWiki//RadixSort

2. http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix/

**Литература:**

1. Дональд Кнут Искусство программирования, том 3. Сортировка и поиск = The Art of Computer Programming, vol.3. Sorting and Searching. — 2-е изд. — М.: «Вильямс», 2007. — С. 824. — ISBN 0-201-89685-0. Глава 5.2.5: Распределяющая сортировка

2. Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн “Алгоритмы. Построение и анализ. Издание 2-е”, Издательство: Вильямс, 2007 г. Глава 8.3

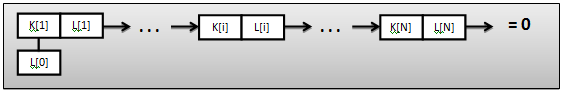
# ЧАСТЬ 2. АЛГОРИТМЫ ПОИСКА

## 6. Последовательный поиск

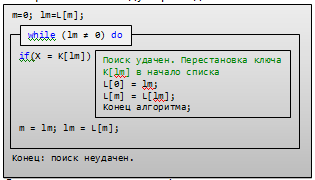
Название этой группы методов произошло от использования последовательного списка в качестве структуры данных для хранения ключей. Список называют упорядоченным, если он упорядочен по значениям ключей, в противном случае он не называется упорядоченным, хотя некоторый порядок ему можно придать за счет размещения в начале списка элементов, которые ищутся наиболее часто, то есть в порядке убывания вероятностей запросов P[1] ... P[N].

### 6.1. Последовательный поиск по связанному списку

Вероятности появления запросов на поиск часто не известны заранее. В этом случае можно использовать эмпирический алгоритм, дающий на практике хорошие результаты: каждый раз при удачном поиске найденный ключ переставляется в начало списка. Естественно такие перестановки удобно делать на связанном списке. Каждому ключу K[i] соответствует поле связи L[i], указывающее на следующий логически за ним ключ. На первый элемент указывает поле связи L[0].



Алгоритм имеет следующий вид:



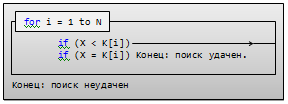
Этот алгоритм легко модифицируется и для случая, когда после неудачного поиска ключ Х должен быть вставлен первым в список. Время работы алгоритма t примерно оценивается формулой: t=a\*N, где a - неизвестная константа, зависящая от программной реализации алгоритма, а также от результата поиска: удачен - неудачен.

### 6.2. Последовательный поиск с перестановкой элементов

### 6.3. Последовательный поиск в упорядоченном файле

Если расположить ключи в возрастающем порядке, то время неудачного поиска можно существенно сократить. В простейшем алгоритме поиска в упорядоченном файле поиск прекращается, если при последовательном просмотре величина Х становится больше или равной очередному ключу K[i]. В среднем при этом сокращается время неудачного поиска до N/2, среднее время удачного поиска остается по-прежнему равным N/2.

Алгоритм имеет вид:

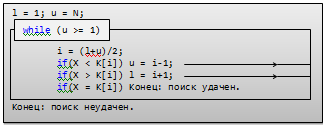


Время работы алгоритма t примерно оценивается формулой: t=a\*N, где a - неизвестная константа, зависящая от программной реализации алгоритма и не зависящая для данного алгоритма от результата поиска: удачен - неудачен.

### 6.4. Бинарный поиск в упорядоченном файле

Алгоритм предполагает деление упорядоченного списка ключей пополам до тех пор, пока в результате очередного деления число элементов в отрезке файла не уменьшится до одного. В этом случае уже можно сказать, был поиск удачным или нет.

Алгоритм имеет вид:

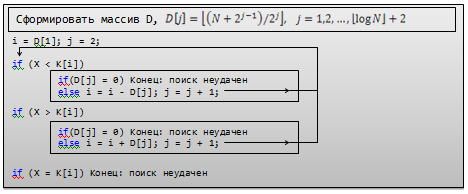


Время работы алгоритма t примерно оценивается формулой: t=a\*lgN, где a - неизвестная константа, зависящая от программной реализации алгоритма, а также от результата поиска: удачен - неудачен. Лучших результатов не может дать ни один метод, основанный на сравнении ключей, появляющихся с равной вероятностью. Однако, метод предназначен только для поиска, как и все методы, основанные на структуре исходного файла в виде последовательного списка. При включении новых ключей очевидны большие затраты по пересылке элементов.

### 6.5. Однородный бинарный поиск

Эта модификация бинарного поиска требует сохранения таблицы значений величин D[1] ... D[max] max= │lgN│+2. Эти величины отражают возможное изменение текущего индекса i на каждом шаге поиска i=i(+-)d, где d - один из элементов массива D. При этом может достигаться экономия времени за счет более эффективной программы.

Схема алгоритма имеет вид:

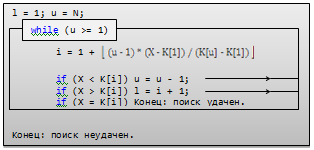


Время работы алгоритма t примерно оценивается формулой: t=a\*lgN, где a - неизвестная константа, зависящая от программной реализации алгоритма, а также от результата поиска: удачен - неудачен.

### 6.6. Интерполяционный поиск

Алгоритм интерполяционного поиска предполагает, что исходный файл упорядочен по величинам ключей поиска. Идея алгоритма состоит в том, что делается предположение о равномерном распределении величин в некотором их диапазоне от u до l. Поэтому, зная величину Х ключа поиска, можно предсказать более точное положение искомой записи, чем просто в середине некоторого отрезка файла. Формула нахождения положения следующего элемента для сравнения следует из деления длины отрезка u-l пропорционально величинам разностей ключей K[u]-K[l] и X-K[l]. Интерполяционный поиск асимптотически предпочтительнее бинарного, если только соблюдается гипотеза о равномерном распределении величин ключей.

Алгоритм имеет вид:



Время работы алгоритма t примерно оценивается формулой: t=a\*lgN, где a - неизвестная константа, зависящая от программной реализации алгоритма, а также от результата поиска: удачен - неудачен.

## 7. Поиск по бинарным деревьям

### 7.1. Поиск по бинарному дереву

Бинарное (двоичное) дерево (binary tree) - это упорядоченное дерево, каждая вершина которого имеет не более двух поддеревьев, причем для каждого узла выполняется правило: в левом поддереве содержатся только ключи, имеющие значения, меньшие, чем значение данного узла, а в правом поддереве содержатся только ключи, имеющие значения, большие, чем значение данного узла.

Бинарное дерево является рекурсивной структурой, поскольку каждое его поддерево само является бинарным деревом и, следовательно, каждый его узел в свою очередь является корнем дерева.

Узел дерева, не имеющий потомков, называется листом.

Бинарное дерево может представлять собой пустое множество.

Популярным приложением бинарных деревьев, например, в технологиях баз данных является поиск требуемых элементов во входном потоке информации. Наиболее быстрый поисковый алгоритм основан на использовании бинарного дерева поиска.

Использование бинарного дерева для организации поиска позволяет сохранить эффективность на уровне бинарного поиска, обеспечивая при этом достаточно простую реализацию вставки и удаления элемента.

Для выполнения поиска бинарное дерево стоится следующим образом (предполагается, что никакие две записи не имеют одинаковых ключей):

1) Первую запись входной последовательности сопоставляют с корнем дерева.

2) Для каждой следующей записи ключ сначала сравнивается с ключом корня. Если он меньше чем ключ корня, то далее сравнивается с ключом правого потомка и т.д. до тех пор, пока потомок не будет отсутствовать. Место этого отсутствующего потомка занимает новая вершина, которой сопоставляется очередная запись.

Шаг 2 повторяется до тех пор, пока не будет просмотрена вся входная последовательность записей.

Поиск записи с заданным ключом осуществляется в точности по рассматриваемой процедуре. Сложность алгоритма поиска по дереву определяется высотой дерева и колеблется в пределах от O(Log2n) до O(n).

**Схема работы:**

k – ключ поиска

p – индекс некоторой вершины

a[p] – значение ключа для этой вершины

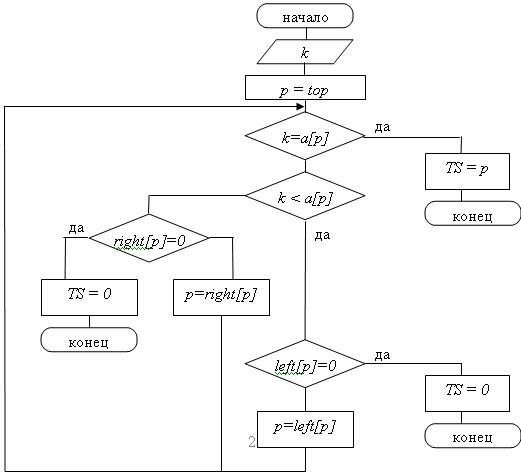
left[p] – номер левого потомка вершины p, если p=0, то потомок отсутствует

right[p] – номер левого потомка вершины p

top – номер вершины, которая является корнем дерева

TS = номеру вершины – в случае успешного поиска

TS = 0 – в случае неуспешного поиска



**Информационые ресурсы**:

http://www.codingrus.ru/readarticle.php?article\_id=2152

### 7.2. Удаление из бинарного дерева

Дано: дерево Т с корнем n и ключом K.

Задача: удалить из дерева Т узел с ключом K (если такой есть).

Алгоритм:

1.Если дерево T пусто, остановиться

Иначе сравнить K с ключом X корневого узла n.

2.Если K>X, рекурсивно удалить K из правого поддерева Т.

3.Если K<X, рекурсивно удалить K из левого поддерева Т.

4.Если K=X, то необходимо рассмотреть три случая.

1.)Если обоих детей нет, то удаляем текущий узел и обнуляем ссылку на него у родительского узла.

2.)Если одного из детей нет, то значения полей второго ребёнка m ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом m.

3.)Если оба ребёнка присутствуют, то:

- найдём узел m, являющийся самым левым узлом правого поддерева;

- присвоим ссылке Left(m) значение Left(n)

- ссылку на узел n в узле Parent(n) заменить на Right(n);

- освободим память, занимаемую узлом n (на него теперь никто не указывает).

Случай 4.3 лучше рассмотреть на отдельном примере с рисунком для наглядности. Например, если из дерева на рис. 1 удаляется узел 20, его сначала нужно заменить на узел 37. Это даст дерево, изображенное на рис. 2 Рассуждения здесь примерно следующие. Нам нужно найти потомка узла 20, справа от которого расположены узлы с большими значениями. Таким образом, нам нужно выбрать узел с наименьшим значением, расположенный справа от узла 20. Чтобы найти его, нам и нужно сначала спуститься на шаг вправо (попадаем в узел 38), а затем на шаг влево (узел 37); эти двухшаговые спуски продолжаются, пока мы не придем в концевой узел, лист дерева.

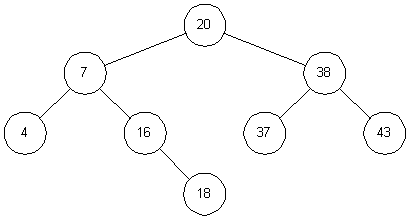


Рис.1(удаление узла 20)

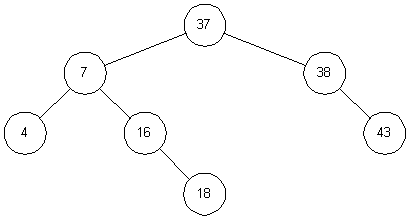
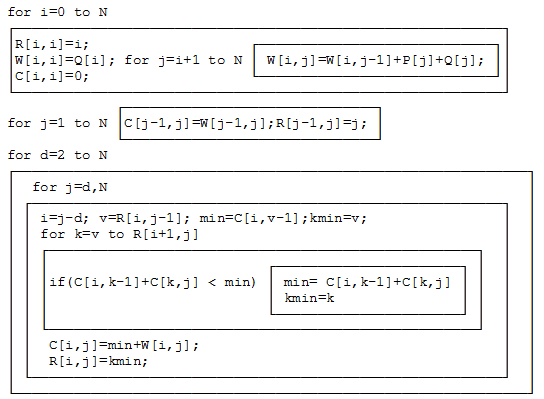


Рис.2(узел 20 удален)

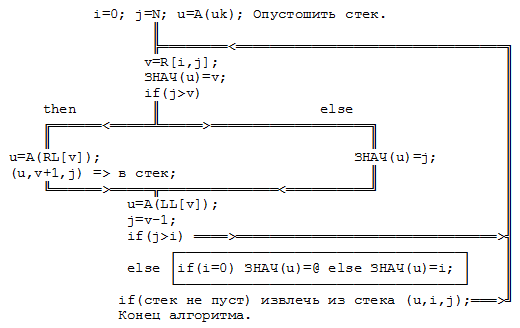
Бинарные деревья представляют собой необычайно мощный ,гибкий и эффективный инструмент. Поскольку при поиске по сбалансированному бинарному дереву(наихудший случай) выполняется log2(n) сравнений, оно по сравнению со связанным списком представляет собой структуру, гораздо более удобную для поиска информации.

### 7.3. Построение оптимальных бинарных деревьев поиска

Деревья, получаемые данным алгоритмом, минимизируют цену дерева - взвешенную длину путей в дереве. Весами узлов являются частоты (вероятности) P[1] P[2] ... P[N] поиска данного ключа, одного из K[1] K[2] ... K[N] (значения ключей упорядочены в порядке возрастания) или частоты (вероятности) попадания искомого значения Х в промежуток между соседними значениями ключей Q[0] Q[1] ... Q[N]. Узлы занумерованы от 1 до N. Алгоритм состоит из двух этапов. На первом этапе вычисляется матрица размером (N+1)\*(N+1), каждая строка и столбец которой соответствует узлу (строка и столбец 0 соответствуют внешнему узлу - листу, отражающему значения ключа поиска меньше наименьшего K[1]). В каждом элементе матрицы с индексами [i,j] хранятся три параметра оптимального поддерева, образованного множеством узлов от узла i до узла j. R[i,j] есть корень оптимального поддерева, W[i,j] есть сумма весов узлов, C[i,j] есть цена поддерева. Из оптимальных поддеревьев в результате составляется оптимальное дерево.



На втором этапе на основе этой матрицы формируется традиционная структура данных для представления дерева. Каждый узел i имеет три поля: значение ключа K[i], ссылку на номер узла - корня левого поддерева LL[i], ссылку на номер узла - корня правого поддерева RL[i]. Ссылка @ есть пустая ссылка на лист дерева. В схеме используются две функции: A(v) - адрес или индекс в массиве некоторой переменной v и ЗНАЧ(u) - значение переменной по адресу u. uk - переменная, которая имеет значение номера корневого узла оптимального дерева. Стек используется для хранения троек (u,i,j), соответствующих поддеревьям, где u - адрес или индекс в массиве корня данного поддерева, i,j - начальный и конечный индексы поддерева. Схема второго этапа имеет следующий вид.



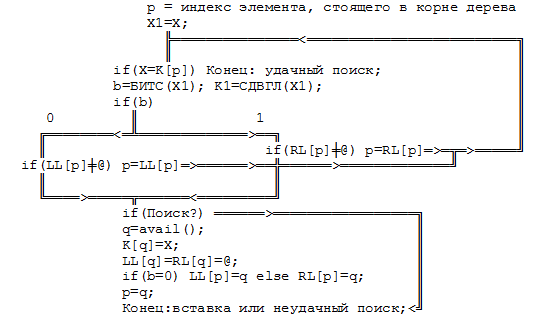
Время работы алгоритма t примерно оценивается формулой:

t=a\*N^2

где a - неизвестная константа, зависящая от программной реализации алгоритма.

### 7.4. Цифровой поиск по дереву

В этом алгоритме предполагается, что искомый ключ Х представляет собой последовательность бит. Функция БИТС(Х) возвращает старший бит ключа Х. Функция СДВГЛ(Х) сдвигает Х на 1 бит влево, добавляя справа 0. Схема алгоритма имеет вид.



### 7.5. Цифровой поиск для длинных ключей, хранимых в текстовом массиве (Патриция)

Название Патриция(Patricia) произошло от сокращения: Practical Algorithm To Retrieve Information Coded In Alphanumeric. Ключи, по которым производится поиск, имеют вид текстовых фраз различной длины и хранятся в массиве TEXT. В узлах дерева поиска хранятся лишь ссылки на TEXT, которые будем обозначать через KEY(p), где p - номер узла. Поиск выполняется путем последовательного просмотра битов ключа поиска Х и "путешествия" по дереву в зависимости от их значения.

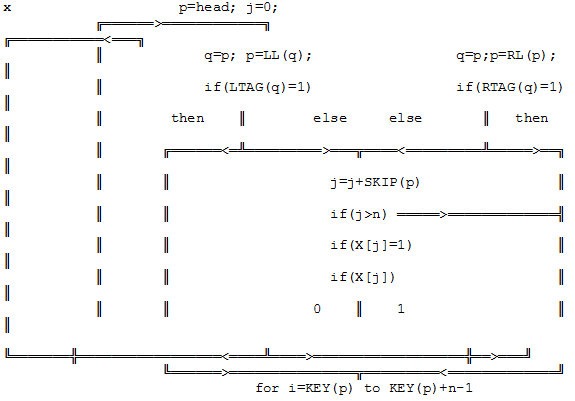
Каждый узел p дерева кроме ссылки KEY(p) на массив TEXT содержит:

- ссылки на левое LL(p) и правое RL(p) поддерево;

- битовые признаки LTAG(p) и RTAG(p) для LL(p) и RL(p) соответственно, обозначающие окончание поиска: если признак равен 0, то ссылка LL(p) или RL(p) не пуста и имеет обычный смысл; если признак равен 1, то ссылка выполнена на одного из предков этого узла или на него самого и обозначает необходимость окончания просмотра дерева и обращения к массиву TEXT из этого узла;

- значения SKIP(p), обозначающие количество бит в аргументе поиска, которые нужно пропустить перед сравнением в следующем узле, то есть номер следующего бита, по которому выполняется сравнение, есть не j+1 (j - номер бита, по которому выполнялось сравнение в данном узле p), а j+SKIP(p). Данный признак позволяет удалить из дерева узлы, из которых ведет только одна ссылка, а другая пуста.

Начальный узел поиска назван "голова" (head), узел head имеет только левую ссылку LL(head) и ccылку на массив TEXT - KEY(head). Через n обозначено количество бит в аргументе поиска Х. Через Х[i] обозначим i-й бит аргумента Х, через TEXT[i] - i-й бит в массиве TEXT, считая от ссылки на него из какого-либо узла. Будем рассматривать два алгоритма: поиска (алгоритм П) и вставки новых узлов (алгоритм В). Второй алгоритм как составную часть использует первый. Схема алгоритма П имеет вид.



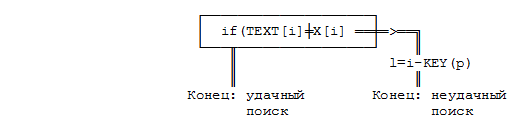
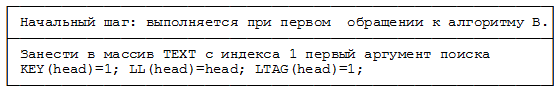
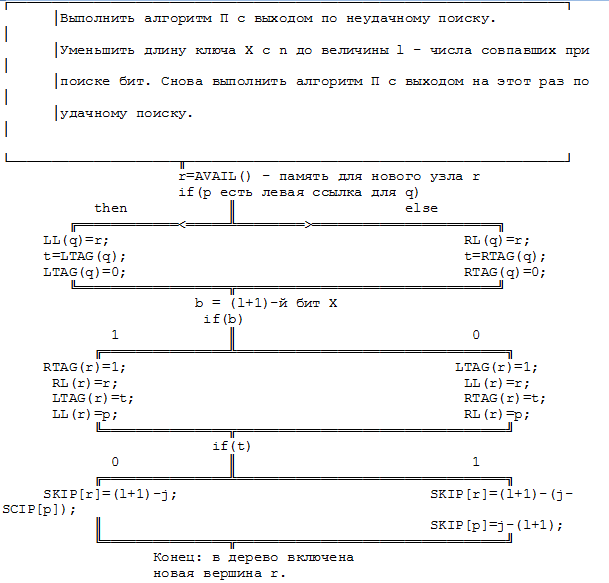


Схема алгоритма В имеет вид.



Следующие шаги выполняются при каждом последующем входе.



Рассмотрим пример работы алгоритмов П и В. Пусть в дерево Патриции нужно последовательно включить следующие элементы: '0 ','А ','Б ','С '. В качестве конечного символа в каждом заказе на поиск используется символ пробела. В 16-ричном и двоичном виде заказы на поиск имеют следующий вид:

'0 '= 4820=0100 1000 0010 0000 Шаг 1

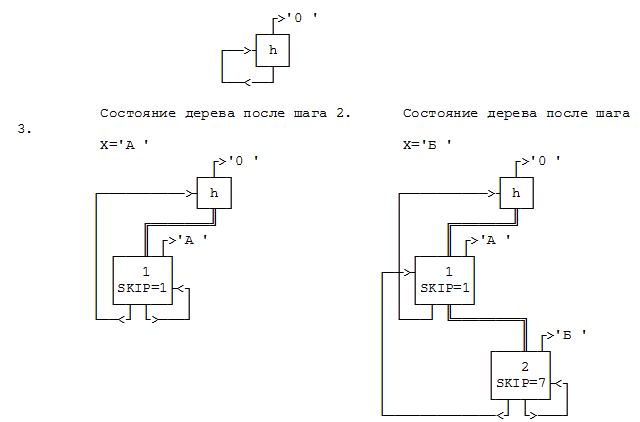
'А '= 8020=1000 0000 0010 0000 Шаг 2

'Б '= 8120=1000 0001 0010 0000 Шаг 3

'С '= 9120=1001 0001 0010 0000 Шаг 4

Шаг 1. Х='0 '

Состояние дерева после шага 1. Дерево состоит только из головной вершины, в которой определены ссылки: на TEXT: KEY(h)=1 (на рисунке показан непосредственно текст '0 ', на который указывает ссылка), на левое поддерево: LL(h)=h и LTAG(h)=1.



На рисунке одинарными линиями показаны ссылки с TAG=1, а двойными - ссылки с TAG=0.

Состояние дерева после шага 4. X='С '

