# Сравнение эффективности методов сортировки массивов: Метод прямого выбора и метод сортировки с помощью дерева

## Лабораторная работа № 1

Сравнить эффективность методов сортировки массивов:

Метод прямого выбора и метод сортировки с помощью дерева.

## Сортировка с помощью прямого выбора

Этот прием основан на следующих принципах:

1. Выбирается элемент с наименьшим ключом.

2. Он меняется местами с первым элементом ai.

3. Затем этот процесс повторяется с оставшимися n-1 элементами, n-2 элементами и т.д. до тех пор, пока не останется один, самый большой элемент.

Процесс работы этим методом с теми же восемью ключами, что и в табл. 2.1, приведен в табл. 2.2. Алгоритм формулируется так:

FORi:=ITO n-1 DO

*присвоить k индекс наименьшего из* *a[i],,,* *a[nJ; поменять местами* a[i] *и a[j]*;

end

Такой метод – его называют *прямым выбором –* в не­котором смысле противоположен прямому включению. При прямом включении на каждом шаге рассматри­ваются только *один* очередной элемент исходной последовательности и *все* элементы готовой последо­вательности, среди которых отыскивается точка вклю­чения; при прямом выборе для поиска *одного* эле­мента с наименьшим ключом просматриваются все элементы исходной последовательности и найденный помещается как очередной элемент в готовую после­довательность. Полностью алгоритм прямого выбора приводится в прогр. 2.3.

Таблица 2.2. Пример сортировки с помощью прямого выбора

Начальные ключи

44 55 12 42 94 18 06 67

06 55 12 42 94 18 44 67

06 12 55 42 94 18 44 67

06 12 18 42 94 55 44 67

05 12 18 42 94 55 44 67

05 12 13 42 44 55 94 67

06 12 18 42 44 55 94 67

06 12 18 42 44 55 67 94

**PROCEDURE StraightSfcleclion;**

**VAR i,j,k: index; x: item; BEGIN**

**FORi:=1 TO n-1 DO k:= i; x := a[i]; FORj:= i+1TO n DO**

**IF a[j]а[k] := а[i]; a[i] ; = x END END StraightSelection**

Прогр. 2.3. Сортировка с помощью прямого выбора,

*Анализ прямого выбора.* Число сравнений ключей (С), очевидно, не зависит от начального порядка клю­чей. Можно сказать, что в этом смысле поведение этого метода менее естественно, чем поведение пря­мого включения. Для С имеем

*с = (n2 - n)/2*

Число перестановок минимально Mmin=3\*(n-l) (2.6)

в случае изначально упорядоченных ключей и макси­мально

Mmax ***=*** n2/4 +3(n-1)

если первоначально ключи располагались в обратном порядке. Для того чтобы определить Mavg, мы должны рассуждать так. Алгоритм просматривает массив, сравнивая каждый элемент с только что обнаружен­ной минимальной величиной; если он меньше первого, то выполняется некоторое присваивание. Вероят­ность, что второй элемент окажется меньше первого, равна 1/2, с этой же вероятностью происходят при­сваивания минимуму. Вероятность, что третий эле­мент окажется меньше первых двух, равна 1/3, а ве­роятность для четвертого оказаться наименьшим — 1/4 и т. д. Поэтому полное ожидаемое число пересы­лок равно Нn—1, где Нn — n-е гармоническое число:

Нn=1+1/2+1/3+ **...** +1/nНп можно выразить и так: Нп = In n+g+ 1/2n — 1/12n2 + ...

где *g=* 0.577216 **... —**константа Эйлера. Для доста­точно больших n мы можем игнорировать дробные составляющие и поэтому аппроксимировать среднее число присваиваний на i-м просмотре выражением

Fi-ln i+g+l

Среднее число пересылок Mavg в сортировке с вы­бором есть сумма Fi с *i* от 1 до n:

Mavg=n\*(g+l)+(Si: 1<i

Вновь аппроксимируя эту сумму дискретных членов интегралом Integral (1: п) ln x dx == x \* (ln x— 1) == n \* ln (п)— n + I

получаем, наконец, приблизительное значение Mavg = n(ln (n) + g)

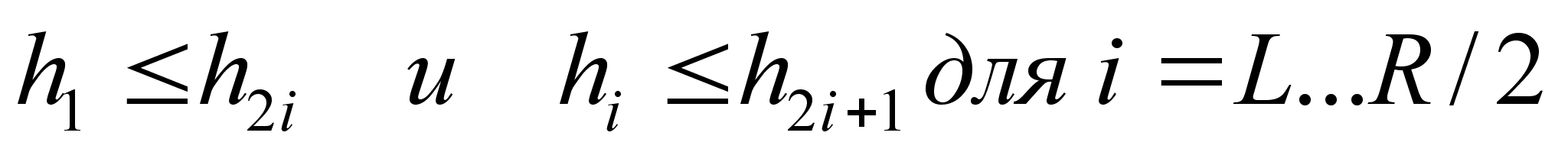
Отсюда можно сделать заключение, что, как правило, алгоритм с прямым выбором предпочтительнее стро­гого включения. Однако, если ключи в начале упоря­дочены или почти упорядочены, прямое включение будет оставаться несколько более быстрым.

## 2.3.2. Сортировка с помощью дерева

Метод сортировки с помощью прямого выбора ос­нован на повторяющихся поисках наименьшего ключа среди n элементов, среди оставшихся n —1 элемен­тов и т. д. Обнаружение наименьшего среди п эле­ментов требует—это очевидно — n — 1 сравнения, среди n — 1 уже нужно n — 2 сравнений и т. д. Сумма первых n — 1 целых равна 1/2\*(n2 — n). Как же в та­ком случае можно усовершенствовать упомянутый ме­тод сортировки? Этого можно добиться, только остав­ляя после каждого прохода больше информации, чем просто идентификация единственного минимального элемента. Например, сделав n/2 сравнений, можно определить в каждой паре ключей меньший. С по­мощью n/4 сравнений — меньший из пары уже вы­бранных меньших и т. д. Проделав n — 1 сравнений, мы можем построить дерево выбора вроде представ­ленного на рис. 2,3 и идентифицировать его корень как нужный нам наименьший ключ [2.21.

Второй этап сортировки — спуск вдоль пути, отме­ченного наименьшим элементом, и исключение его из дерева путем замены либо на пустой элемент (дырку) в самом низу, либо на элемент из соседней ветви в промежуточных вершинах (см. рис. 2.4 и 2.5). Элемент, передвинувшийся в корень дерева, вновь бу­дет наименьшим (теперь уже вторым) ключом, и его можно исключить. После п таких шагов дерево станет пустым (т. е. в нем останутся только дырки), и процесс сортировки заканчивается. Обратите внима­ние — на каждом из n шагов выбора требуется только log n сравнений. Поэтому на весь процесс понадобит­ся порядка n\*log n элементарных операций плюс еще n шагов на построение дерева. Это весьма суще­ственное улучшение не только прямого метода, тре­бующего *п2* шагов, но и даже метода Шелла, где нужно п^1.2 шага. Естественно, сохранение дополни­тельной информации делает задачу более изощренной, поэтому в сортировке по дереву каждый отдельный шаг усложняется. Ведь в конце концов для сохране­ния избыточной информации, получаемой при начальном проходе, создается некоторая древообраз­ная структура. Наша следующая задача — найти приемы эффективной организации этой информации.

Конечно, хотелось бы, в частности, избавиться от дырок, которыми в конечном итоге будет заполнено все дерево и которые порождают много ненужных сравнений. Кроме того, надо было бы поискать такое представление дерева из п элементов, которое требует лишь п единиц памяти, а не 2n—1, как это было раньше. Этих целей действительно удалось добиться в методе *Heapsort**[1](" \l "sdfootnote1sym)\*)* изобретенном Д. Уилльямсом (2.14), где было получено, очевидно, существенное улучшение традиционных сортировок с помощью де­ревьев. Пирамида определяется как последователь­ность ключей hi., hL+1, .. , hr, такая, что



Если любое двоичное дерево рассматривать как мас­сив по схеме на рис. 2.6, то можно говорить, что де­ревья сортировок на рис. 2.7 и 2.8 суть пирамиды, а элемент h1, в частности, их наименьший элемент: hi==min(hi, h2, ..., hn). Предположим, есть некото­рая пирамида с заданными элементами hL+1, ..., hR для некоторых значений L и R и нужно добавить но­вый элемент х, образуя расширенную пирамиду hi., .. . ..., *li R.* Возьмем, например, в качестве исходной пи­рамиду hi, ..., *hr,* показанную на рис. 2.7, и добавим к ней слева элемент h1==44[2](" \l "sdfootnote2sym) Новая пирамида по­лучается так: сначала х ставится наверх древовидной структуры, а затем он постепенно опускается вниз каждый раз по направлению наименьшего из двух примыкающих к нему элементов, а сам этот элемент передвигается вверх. В приведенном примере значе­ние 44 сначала меняется местами с 06, затем с 12 ц в результате образуется дерево, представленное на рис. 2.8. Теперь мы сформулируем этот сдвигающий алгоритм так: i, j — пара индексов, фиксирующих эле­менты, меняющиеся на каждом шаге местами. Чита­телю остается лишь убедиться самому, что предложен­ный метод сдвигов действительно сохраняет неизмен­ным условия (2.13), определяющие пирамиду.

Р. Флойдом был предложен некий «лаконичный» способ построения пирамиды «на том же месте». Его процедура сдвига представлена как прогр. 2.7. Здесь hi . .. hn — некий массив, причем Ьщ ...hn (пп = ==(nDIV2)+1) уже образуют пирамиду, поскольку индексов i, j, удовлетворяющих отношениям j = 2i (или j = 2i+1), просто не существует. Эти элементы образуют как бы нижний слой соответствующего двоичного дерева (см. рис. 2.6), для них никакой

PROCEDURE sift(L, R; index);

VAR i, j: index; x: item;

BEGIN i : = L; J:= 2\*L; x := a[L];

IF (j < R) & (a[J+l] < a[j] THEN j:=j+l END;

WHILE (j < =R)&(a[j]

a[i]:= a[j]: i:=j; i := 2\*j;

IF(j<R)&(a[j+1]

END

END sift

Прогр. 2.7. Sift.

упорядоченности не требуется. Теперь пирамида рас­ширяется влево; каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент. Табл. 2.6 иллюстрирует весь этот процесс, а получаю­щаяся пирамида показана на рис. 2.6.

Следовательно, процесс формирования пирамиды из п элементов hi ... hn на том же самом месте опи­сывается так:

L :== (n DIV 2) + 1;

WHILE L > 1 DO L :== L — 1; sift(L, n) END

Для того чтобы получить не только частичную, но и полную упорядоченность среди элементов, нужно про­делать n сдвигающих шагов, причем после каждого шага на вершину дерева выталкивается очередной (наименьший) элемент. И вновь возникает вопрос: где хранить «всплывающие» верхние элементы и мож­но ли или нельзя проводить обращение на том же месте? Существует, конечно, такой выход: каждый раз брать последнюю компоненту пирамиды (скажем, это будет х), прятать верхний элемент пирамиды в освободившемся теперь месте, а х сдвигать в нужное место. В табл. 2.7 приведены необходимые в этом слу-

.Таблица 2.6, Построение пирамиды

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
| 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
| 44 | 55 | 06 | 42 | 94 | 18 | 12 | 67 |
| 44 | 42 | 06 | 55 | 94 | 18 | 12 | 67 |
| 06 | 42 | 12 | 55 | 94 | 18 | 44 | 67 |

Таблица 2.7. Пример процесса сортировки с помощью Heapsort

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 06 | 42 | 12 | 55 | 94 | 18 | 44 | 67 |
| 12 | 42 | 18 | 55 | 94 | 67 | 44 | 06 |
| 18 | 42 | 44 | 55 | 94 | 67 | 12 | 06 |
| 42 | 55 | 44 | 67 | 94 | 18 | 12 | 06 |
| 44 | 55 | 94 | 67 | 42 | 18 | 12 | 06 |
| 55 | 67 | 94 | 44 | 42 | 18 | 12 | 06 |
| 67 | 94 | 55 | 44 | 42 | 18 | 12 | 06 |
| 94 | 67 | 55 | 44 | 42 | 18 | 12 | 06 |

чае n — 1 шагов. Сам процесс описывается с помощью процедуры *sift* (прогр. 2.7) таким образом:

R:=n; WHILE R>1 DO х := а[l]; a[l] :*=* a[R]; a[R] := x: R:=R-l; sift(l,R) END

Пример из табл. 2.7 показывает, что получающийся порядок фактически является обратным. Однако это можно легко исправить, изменив направление «упо­рядочивающего отношения» в процедуре *sift.* В конце концов получаем процедуру Heapsort (прогр. 2.8),

PROCEDURE HeapSort; VAR L, R: index; х: item; PROCHDURE sift(L, R: index); VAR i,j:index; x: item; BEGIN i: = L; j := 2\*L: x := a[L]; IF(j < R) & (a[j] < a[j+1]) THEN j := j+l END; WHILE(j<=R)&(x<R)&(a[i]

BEGIN L:=:(nDIV2)+l:R:=n; WHILE L > 1 DO L: = L-l; sift(L, R) END; WHILER>1 DO x :*=* a[l]; a[l] := a[R]; a[R] := x; R := R-l; sifl(L, R) END END HeapSort

Прогр. 2.8. HeapsorL

*Анализ Heapsort.* На первый взгляд вовсе не оче­видно, что такой метод сортировки дает хорошие ре­зультаты. Ведь в конце концов большие элементы, прежде чем попадут на свое место в правой части, сначала сдвигаются влево. И действительно, про­цедуру не рекомендуется применять для небольшого, вроде нашего примера, числа элементов. Для боль­ших же n Heapsort очень эффективна; чем больше п, тем лучше она работает. Она даже становится срав­нимой с сортировкой Шелла.

В худшем случае нужно п/2 сдвигающих шагов, они сдвигают элементы на log (n/2), log (п/2—1), **...** ..., log(n—l) позиций (логарифм (по основанию 2)] «обрубается» до следующего меньшего целого). Сле­довательно, фаза сортировки требует n—1 сдвигов с самое большое log(n—1), log(n—2), ..., 1 переме­щениями. Кроме того, нужно еще n —1 перемещений для просачивания сдвинутого элемента на некоторое расстояние вправо. Эти соображения показывают, что даже в самом плохом из возможных случаев Heap-sort потребует n\*log n шагов. Великолепная произ­водительность в таких плохих случаях—одно из при­влекательных свойств Heapsort.

Совсем не ясно, когда следует ожидать наихудшей (или наилучшей) производительности. Но вообще-то кажется, что Heapsort «любит» начальные последо­вательности, в которых элементы более или менее отсортированы в обратном порядке. Поэтому ее по­ведение несколько неестественно. Если мы имеем дело с обратным порядком, то фаза порождения пирамиды не требует каких-либо перемещений. Среднее число перемещений приблизительно равно п/2\* log(n), при­чем отклонения от этого значения относительно неве­лики.

[1](" \l "sdfootnote1anc) Здесь мы оставляем попытки перевести названия соответ­ствующих методов на русский язык, ибо ни уже не более чем собственные имена, хотя в названиях первых упоминавшихся методов еще фигурировал некоторый элемент описания сути са­мого приема сортировки. — *Прим. перев.*

[2](" \l "sdfootnote2anc) Это несколько противоречит утверждению, что lu+i ..) ...hr—пирамида, но надеемся, сам читатель разберется, *Что* хотел сказать автор. — *Прим. перев.*