

ПОИСК И СОРТИРОВКА МАССИВОВ

Организация работы с базами данных

- Для создания и обработки всевозможных баз данных широко применяются массивы структур. Обычно база данных накапливается и хранится в виде файла на магнитном диске. К ней часто приходится обращаться, обновлять, перегруппировывать, осуществлять поиск. Работа с базой может быть организована двумя способами:
 - 1. Вносить изменения и осуществлять поиск можно прямо в файле, при этом временные затраты на обработку данных (поиск, сортировку) значительно возрастают, но нет ограничений на оперативную память.
 - 2. В начале работы вся база (или ее необходимая часть)

- считывается в массив структур и обработка производится в оперативной памяти, что значительно сокращает ее время, однако требует затрат оперативной памяти. Наиболее частыми операциями при работе с базами данных являются «поиск» и «сортировка». При этом алгоритмы решения этих задач существенно зависят от того, организованы ли структуры в виде массива или файла. Обычно структура содержит некое ключевое поле (*ключ*), по которому ее находят среди множества других аналогичных структур. В зависимости от решаемой задачи ключом может служить, например, фамилия или номер расчетного счета. Основное требование к ключу в задачах поиска состоит в том, чтобы операция проверки на равенство была корректной. Поэтому, например, в качестве ключа не следует выбирать действительные

- числа, т. к. из-за ошибки округления поиск нужного ключа может оказаться безрезультатным, хотя этот ключ в массиве имеется.

Поиск в массиве записей

- Задача поиска требуемого элемента в массиве структур $a[k]$, $k = 0, \dots, n-1$ заключается в нахождении индекса k , удовлетворяющего условию $a[k].kluch = isk$. Здесь поле структуры $kluch$ выступает в качестве ключа, isk – искомый ключ. После нахождения значения индекса k обеспечивается доступ ко всем другим полям найденной структуры $a[k]$.
- **Линейный (последовательный) поиск** используется, когда нет никакой дополнительной информации о разыскиваемых данных. Он представляет собой последовательный перебор массива до обнаружения требуемого ключа или до конца, если ключ не обнаружен:
 - $k=0$;
 - `while (k<n && a[k].kluch != isk) k++;`
 - `if (k==n) cout<< "элемент не найден"<<endl;`
 - `else cout<<"индекс искомого элемента=" <<k<<endl;`

- **Поиск с барьером.** В линейном поиске на каждом шаге вычисляется логическое выражение. Уменьшить затраты на поиск можно упростив логическое выражение с помощью введения вспомогательного элемента – **барьера**, который предохраняет от выхода за пределы массива. Для этого в **конец** массива добавляется элемент с искомым ключом. Количество проверок на каждом шаге уменьшается (проверка одного, а не двух условий), т. к. нет необходимости проверки выхода за пределы массива – элемент с искомым ключом обязательно будет найден до выхода за пределы расширенного массива.
- `a[n].kluch = isk; k=0;`
- `while (a[k].kluch != isk) k++;`
- `if (k==n) cout<<"элемент не найден";`
- `else cout<<"индекс искомого элемента="<<k<<endl;`

• **Поиск делением пополам (бинарный поиск)** используется, когда данные упорядочены по возрастанию ключа **kluch**. Основная идея поиска – берется «средний» (**m-й**) элемент массива. Если **a[m].kluch < isk**, то все элементы массива **a[k]** с индексами **k ≤ m** можно исключить из дальнейшего поиска, в противном случае исключаются элементы с индексами **k > m**:

- `i = 0; j = n-1;`
- `while (i < j)`
- `{`
- `m = (i+j) / 2;`
- `if (a[m].kluch < isk) i = m+1;`
- `else j = m;`
- `}`
- `if (a[i].kluch == isk)`
- `cout << "индекс искомого элемента=" << i << endl;`
- `else cout << "элемент не найден" << endl;`

- В этом алгоритме отсутствует проверка внутри цикла совпадения ключа `a[m].kluch==isk`. На первый взгляд это кажется странным, однако тестирование показывает, что в среднем выигрыш от уменьшения количества проверок превосходит потери от нескольких «лишних» вычислений до выполнения условия `i=j`.

Сортировка массивов

Под сортировкой понимается процесс перегруппировки элементов массива, приводящий к их упорядоченному расположению относительно ключа. Цель сортировки – облегчить последующий поиск элементов. Метод сортировки называется устойчивым, если в процессе перегруппировки относительное расположение элементов с равными ключами не изменяется. Основное условие при сортировке массивов – не вводить дополнительных массивов, т.е. все перестановки элементов должны выполняться в исходном массиве. Сортировку массивов принято называть **внутренней** в отличие от сортировки файлов (списков), которую называют **внешней**.

- Методы внутренней сортировки классифицируются по времени их работы. Мерой эффективности может быть число сравнений ключей C и число пересылок элементов P . Эти числа являются функциями $C(n)$, $P(n)$ от числа сортируемых элементов n . **Быстрые** (но **сложные**) алгоритмы сортировки требуют (при $n \rightarrow \infty$) порядка $n \cdot \log(n)$ сравнений, **прямые (простые)** методы – n^2 .
- Прямые методы коротки, просто программируются. Быстрые, усложненные методы требуют меньшего числа операций, но эти операции обычно сами более сложны, чем операции прямых методов, поэтому для достаточно малых n ($n < 50$) прямые методы работают быстрее. Значительное преимущество быстрых методов начинает проявляться при $n > 100$.

- Среди **простых** методов наиболее популярны:
- 1) *Метод прямого обмена* (пузырьковая сортировка).
- 2) *Метод прямого выбора*.
- 3) *Сортировка с помощью прямого (двоичного) включения*.
- 4) *Шейкерная сортировка* (модификация пузырьковой).
-
- **Улучшенные** методы сортировки:
- 1) *Метод Шелла* , усовершенствование метода прямого включения.
- 2) *Сортировка с помощью дерева*, метод *HeapSort*, Д. Уильямсон.
- 3) *Сортировка с помощью разделения*, метод *QuickSort*, Ч. Хоар. На сегодняшний день это самый эффективный метод сортировки.
- Рассмотрим алгоритмы и реализацию некоторых методов.

Метод пузырьковой сортировки

- При сортировке методом пузырька сравниваются два соседних элемента. Если предыдущий элемент больше последующего, то выполняется перестановка этих элементов. Сортировка осуществляется путем многократного прохода по списку элементов. В общем случае для сортировки n элементов требуется $n-1$ проход по массиву. Чтобы исключить ненужные проходы, если массив уже был полностью или частично отсортирован, нужно использовать не оператор цикла с заданным количеством повторений `for`, а оператор `while` с флажком `p`. Оба варианта пузырьковой сортировки приведены ниже:

- **Сортировка с явно заданным числом проходов**

- `void puzsort(double a[], int n)`

- `{ double w;`

- `for (int i=1; i<n; i++)` **Номер прохода**

- `for (int k=0; k<n-i; k++)` **Номер элемента**

- `if (a[k]>a[k+1])`

- `{ w=a[k];`

- `a[k]=a[k+1];`

- `a[k+1]=w;`

- `}`

- `}`

- **Сортировка с использованием флага**
- `void puzsort(double a[], int n)`
- `{ int i,k; double w; bool p;`
- `i=1; Номер прохода`
- `do`
- `{ p=false; Флажок`
- `for (k=0; k<n-i; k++)`
- `if (a[k]>a[k+1])`
- `{ w=a[k];`
- `a[k]=a[k+1];`
- `a[k+1]=w;`
- `p=true; Была перестановка`
- `}`
- `i++;`
- `} while (p);`
- `} Сортировка прекращается, если на каком-то проходе не было перестановок (p=false)`

Метод прямого выбора

- Сортировка осуществляется путем многократного прохождения по списку элементов. На каждом (**k-ом**) проходе находится номер минимального элемента начиная с **k-го**, который затем переставляется с **k-м** элементом. Сортировка методом прямого выбора имеет вид:
- `void PramSort(double a[], int n)`
- `{ int k, i, m; double w;`
- `for (k=0; k<n-1; k++)`
- `{ m=k;`
- `for (i=k+1; i<n; i++)`
- `if (a[i]<a[m]) m=i;`
- `w=a[m]; a[m]=a[k]; a[k]=w;`
- `}`
- `}`

Метод Шелла

- Метод сортировки Шелла намного эффективнее, чем похожий на него метод пузырька. Здесь также сравниваются два элемента, но не соседние, а разделенные интервалом kol . Начальное значение kol равно половине количества элементов. В процессе сортировки значение kol уменьшается в два раза на каждом проходе, пока не начнет выполняться сравнение соседних элементов, как и в методе пузырька, т.е. сначала сравниваются отдаленные, а затем все более близкорасположенные элементы. Сортировка методом Шелла имеет вид:



- void ShellSort(double a[], int n)
- { int k, kol; double w; bool p;
- kol=n/2; **Начальный интервал**
- do
- { do
- { p=false;
- for (k=0; k < n-kol; k++)
- if (a[k]>a[k+kol])
- { w=a[k];
- a[k]=a[k+kol];
- a[k+kol]=w;
- p=true;
- }
- } while (p);
- kol=kol/2; **Новый интервал**
- } while (kol>0);
- }

- **Полезный совет:** перестановка элементов со сложными типами данных (с объемами занимающими десятки-сотни байтов) занимает значительно больше времени чем перестановка 4-х байтовых значений. Поэтому рекомендуется вместо перестановки самих данных переставлять их номера (индексы). Такой прием используется практически во всех коммерческих приложениях. В этом случае метод сортировки Шелла с перестановкой индексов имеет вид:

- void ShellSortInd(double a[], int n, int nom[])
- { int k, kol, w; bool p;
- for (k=0; k<n; k++) nom[k]=k; **Начальные номера**
- kol=n/2; **Начальный интервал**
- do {
- do {
- p=false;
- for (k=0; k<n-kol; k++)
- if (a[nom[k]]>a[nom[k+kol]])
- {
- w=nom[k];
- nom[k]=nom[k+kol];
- nom[k+kol]=w;
- p=true;
- }
- } while (p);
- kol=kol/2; **Новый интервал**
- } while (kol>0);
- }

Метод Хоара (Hoare)

- Идея метода быстрой сортировки Хоара в следующем. Сначала выбирается так называемое опорное значение op , в качестве которого может быть использовано значение какого-либо внутреннего элемента. Массив просматривается слева-направо до тех пор, пока не будет обнаружен элемент $a[i] > op$. Затем массив просматривается справа-налево, пока не будет обнаружен элемент $a[j] < op$. Элементы $a[i]$ и $a[j]$ меняются местами. Процесс просмотра и обмена продолжается до тех пор, пока i не станет больше j , т.е. элементы со значением, меньше опорного, переносятся влево, а со значением, большим или равным ему, – вправо. Таким образом, на первом этапе алгоритм Хоара делит элементы на два раздела: со значениями, меньшими опорного, и со значениями, большими или равными ему. Затем алгоритм разделения применяется к левому и правому разделам до тех пор, пока каждый из разделов не будет состоять из одного единственного элемента. На каждом этапе запоминаются номер разделения и границы разделения. Эффективность метода зависит от объема данных и выбора опорного сечения. В приведенных ниже алгоритмах в качестве опорного выбирается средний элемент раздела. Рекурсивный вариант сортировки Хоара имеет вид:

- void QuickSort(double a[], int low, int high)
- { int i, j; **Рекурсивный метод Хоара**
- double op, w;
- op=a[(low+high)/2]; **Опорный элемент**

Перенос элементов, меньших опорного, влево, а больших - право

- i=low; j=high;
- do {
- while ((i<=high) && (a[i]<op)) i++;
- while ((j>=low) && (a[j] >op)) j--;
- if (i<=j) { w=a[i]; a[i]=a[j]; a[j]=w;
- i++; j--;
- }
- } while (i<j);
- if (j>low) QuickSort(a,low,j);
- if (i<high) QuickSort(a,i,high);
- }

- void QuickSort(double a[], int n)
- { struct
- { int l;
- int r;
- } w[50];
- int i, j, left, right, op, s=0; double t;
- w[s].l=0; w[s].r=n-1;
- while (s!=-1)
- { left=w[s].l;
- right=w[s].r;
- s--;
- while (left<right)
- { i=left;
- j=right;
- op=a[(left+right)/2];
- while (i<=j)
- { while(a[i]<op) i++;
- while(a[j]>op) j--;

- if (i<=j) { t=a[i]; a[i]=a[j]; a[j]=t;
- i++; j--;
- }
- }
- if ((j-left) < (right-i))
- { if (i<right) { s++;
- w[s].l=i;
- w[s].r=right;
- }
- right=j;
- }
- else { if (left<j) { s++;
- w[s].l=left;
- w[s].r=j;
- }
- left=i;
- }
- } } }