Работа с таблицами

---

1. Тип таблицы

Таблица - это массив записей. Основные операции с таблицами: ***включение записи, удаление записи, поиск записи по заданному признаку, сортировка записей таблицы по одному или нескольким полям***.

Назначение сортировки состоит в ускорении операции поиска, так как наличие каких бы то ни было элементов упорядоченности таблицы может быть использовано для этой цели. Сразу отметим, что сортировка таблицы - это достаточно трудоёмкая операция в вычислительном отношении. Очевидно, что в тех случаях, когда трудозатраты на сортировку не компенсируются в последующем их сокращением на поиск, следует использовать неупорядоченную таблицу.

Определим тип таблицы и саму таблицу:

```
const M = 7;                      {Макс. размер таблицы}
type TRec = record                {Тип записей таблицы}
            Tag:  Char;           {Тэг записи}
            Key:  string[5];      {Ключ записи}
            Info: string[10];     {Значение записи}
            end;
     TTab = array[1..M] of TRec; {Тип таблицы}

var  Tab:   TTab;                 {Таблица}
     Cnt:   Integer;              {Фактический размер таблицы}
```

Тэг - вспомогательное поле записи, которое может служить для указания каких-либо её признаков, например, что запись удалена.
Ключ - одно из полей записи, по которому предполагается выполнять поиск в таблице или её сортировку.
Значение - обобщённое обозначение других полей записи, которые используются в чисто информационном смысле.

В этом разделе мы будем исходить из предположения, что в таблице нет записей с одинаковыми значениями ключей.

Пример. Предположим, что таблица Tab - это телефонный справочник. Поле Key в этом случае может быть номером телефона (например, '23-34'), а поле Info - именем клиента (например, 'Bess').

2. Инициализация таблицы

Выполняется, как правило, в начале выполнения программы. Пример.

```
Cnt := 0;
```

и/или

```
for i:=1 to M do Tab[i].Tag:=' ';
```

3. Включение новой записи в неупорядоченную таблицу

Пример.

```
var NewRec: TRec;
. . .
{Ввод или определение значений переменных NewKey и NewInfo}
. . .
with NewRec do begin Key:=NewKey; Info:=NewInfo end;
if Cnt >= M then Error := 1;
Inc(Cnt);(
Tab[Cnt] := NewRec;
```

Здесь предполагается, что Error - это глобальная переменная, которая предназначена для фиксации кода ошибки. В данном случае будем считать, что 1 означает "Переполнение таблицы". Предполагается, что до выполнения операции управляющая часть программы обнуляет переменную Error, а после выполнения операции анализирует её значение и принимает соответствующие меры.

4. Поиск в неупорядоченной таблице

Задача поиска состоит в определении индекса записи таблицы, поле Key которой равно заданному значению - ***аргументу поиска***.

Оформим решение этой задачи в форме процедуры Search. Составим её ***спецификацию***.

Спецификация процедуры Search -
"Поиск записи в неупорядоченной таблице"

1. Заголовок: **procedure** Search(Arg: **string**; **var** Index, ErrCode: Integer);
2. Входные параметры: Arg - аргумент поиска.
3. Выходные параметры: Index - индекс искомой записи; ErrCode - код завершения.
4. Глобальные имена: Tab, Cnt.
5. Функция: Определение индекса Index, удовлетворяющего условию Tab[Index].Key = Arg, с формированием кода завершения ErrCode: 0 - запись найдена, 1 - запись не найдена.
6. Используемые процедуры: Нет.
7. Особые случаи: Нет.

Отметим, что в процедуре Search мы используем несколько другой метод обработки ошибок, чем выше, - формирование ***кода завершения процедуры***. По значению кода завершения вызывающая программа принимает решение о дальнейшем поведении.

```
procedure Search(Arg: string; var Index, ErrCode: Integer);
var i: Integer;
```

```
begin
for i:=1 to Cnt do
   if Tab[i].Key = Arg then
      begin Index:=i; ErrCode:=0; Exit end;
ErrCode:=1; {Запись не найдена}
end; {Search}
```

В заключение этого раздела остановимся на понятии спецификации. Большинство пунктов спецификации достаточно очевидны. Некоторого пояснения требуют лишь пункты 6 и 7. В пункте 6 перечисляются имена процедур, которые вызываются в данной процедуре. Пункт 7 содержит информацию о способах передачи данных между поцедурой и её окружением, минуя механизмы передачи параметров и глобальных переменных. Обычно такую роль выполняют файлы.

Роль спецификаций в программировании трудно переоценить. Для программиста-разработчика процедуры спецификация это документ, играющий роль задания на разработку. Для программиста-пользователя она даёт исчерпывающую информацию о способе включения процедуры в программную систему.

5. Метод барьера

Позволяет несколько увеличить производительность предыдущей процедуры за счёт сокращения количества проверок условий в теле цикла.

```
procedure Search(Arg: string; var Index, ErrCode: Integer);
var i: Integer;
begin
Tab[Cnt+1].Key:=Arg;   { Установка баръера }
i:=1;
while Tab[i].Key <> Arg do Inc(i);
if i = Cnt + 1 then begin ErrCode:=1; Exit end;  {Запись не найдена}
Index:=i; ErrCode:=0;
end; {Search}
```

Метод барьера требует, чтобы в таблице всегда находился хотя-бы один свободный элемент, следующий за элементом с индексом Cnt. Это необходимо учитывать при определении факта переполнения таблицы.

6. Двоичный поиск

Может быть использован в случае упорядоченной таблицы. Идея двоичного поиска состоит в последовательном делении области поиска пополам. Характер убывающей геометрической прогрессии этого процесса приводит к резкому сокращению времени поиска. Предположим, что записи в таблице упорядочены по возрастанию ключа. Далее предположим, что Up..Down - диапазон индексов записей таблицы, одна из которых, возможно, содержит искомый ключ. Определим середину диапазона Up..Down - i = (Up + Down) **div** 2. Теперь, если Tab[i].Key < Arg, то это в силу упорядоченности таблицы означает, что индекс искомой записи может находиться только в диапазоне i+1..Down. Если же Tab[i].Key > Arg, то - в диапазоне Up..i-1. Мы видим, что проверка всего двух условий приводит к сокращению области поиска сразу в два раза! На следующем шаге полученная область уменьшится ещё в два раза, и т.д.

```
procedure Search(Arg: string; var Index, ErrCode: Integer);
var i: Integer;
begin
```

```
Up:=1; Down:=Cnt;
repeat
   i:=(Up + Down) div 2;
   if Tab[i].Key <= Arg then Up:=i+1;
   if Tab[i].Key >= Arg then Down:=i-1;
until Up > Down;
if Up = Down + 2 then
   begin Index:=i; ErrCode:=0; Exit end;
ErrCode:=1;  {Запись не найдена}
end; {Search}
```

Сравним данный метод с предыдущим. По объёму кода несколько больше процедура двоичного поиска. Оценим производительность каждого метода. Время поиска методом перебора, который применяется для поиска в неупорядоченной таблице, может быть оценено по формуле Тп = Кп*Cnt/2. Для двоичного поиска - Тд = Кд*log2(Cnt). Здесь Кп и Кд - это коэффициенты, пропорциональные времени выполнения операторов тела цикла соответствующих методов. Грубая оценка даёт Кд ~= 5*Кп. Если предположить, что Кп=1, то для таблицы, содержащей 1000 записей, получим Тп = 500, а Тд = 50 некоторых условных единиц времени. Вывод очевиден. Однако, следует заметить, что поддержание таблицы в упорядоченном состоянии тоже требует затрат времени и окончательный выбор метода следует делать с учетом всех факторов. На практике двоичный поиск применяют тогда, когда таблица изменяется редко или не изменяется совсем.

7. Сортировка таблицы

Задача сортировки состоит в перестановке записей таблицы таким образом, чтобы выполнялось условие Tab[i].Key < Tab[i+1].Key, i = 1..Cnt-1. Сортировка таблицы это достаточно трудоёмкая операция. Мы рассмотрим сортировку ***методом простых вставок.***

```
var R:   TRec;
    i,j: Integer;
. . .
for i:=2 to Cnt do
   begin
   R:=Tab[i]; j:=i-1;
   while (j > 0) and (R.Key < Tab[j].Key) do
      begin Tab[j+1]:=Tab[j]; Dec(j) end;
   Tab[j+1]:=R;
   end;
```

Идея метода простых вставок состоит в следующем. Предположим что начальная часть таблицы 1..i-1 уже отсортирована. В частности она может состоять всего из одной записи, первой. Запись Tab[i] сохраняется в R. Далее определяется местоположение записи R в части таблицы 1..i-1. Для этого R.Key последовательно сравнивается с ключами записей таблицы, начиная с j = i - 1 в сторону уменьшения индекса j, пока не будет нарушено условие R.Key < Tab[j].Key. Попутно записи, для которых условие R.Key < Tab[j].Key выполняется, сдвигаются на одну позицию в сторону конца таблицы, освобождая тем самым место для включения записи R. В результате выполнения этого процесса размер отсортированной части таблицы увеличивается на 1. Остаётся продолжить эти действия для оставшихся записей массива.

Это наиболее эффективный метод сортировки из числа наиболее простых. Он отлично работает, когда таблица состоит из 30..50 элементов. Для таблиц большего размера приходится применять другие, более изощрённые методы.

8. Включение новой записи в упорядоченную таблицу

Другой способ получения упорядоченной таблицы состоит в выполнении включения каждой новой записи в таблицу таким образом, чтобы свойство её упорядоченности сохранялось.

Спецификация процедуры Insert -
"Включение записи в упорядоченную таблицу"

1. Заголовок: **procedure** Insert(R: TRec);
2. Входные параметры: R - включаемая запись.
3. Выходные параметры: нет.
4. Глобальные имена: TRec, Tab, Cnt, M, ErrCode.
5. Функция: Включение записи R в упорядоченную таблицу Tab с сохранением свойства её упорядоченности. Формирование кода завершения ErrCode: 0 - нормальное включение, 1 - таблица переполнена.
6. Используемые модули: Нет.
7. Особые случаи: Нет.

```
procedure Insert(R: TRec);
var j:  Integer;

begin                       {Tab is full}
if Cnt >= M then begin ErrCode:=1; Exit end;
ErrCode:=0; j:=Cnt;
while (j>0) and (R.Key < Tab[j].Key) do
   begin Tab[j+1]:=Tab[j]; Dec(j) end;
Tab[j+1]:=R; Inc(Cnt);
end; {Insert}
```

Включение в этом примере выполняется так же, как внутренний цикл в алгоритме сортировки простыми вставками. Кроме этого пример иллюстрирует ещё раз способ передачи кода завершения, при котором используется глобальная переменная.

9. Удаление записи из таблицы

Предполагаем, что индекс удаляемой записи (Index) уже известен. Как правило, он является результатом выполнения предыдущей операции поиска.

```
var i: Integer;
. . .
for i:=Index to Cnt-1 do Tab[i]:=Tab[i+1];
Dec(Cnt);
```

Основная часть времени выполнения операции уходит на сдвиг элементов хвоста таблицы на одну позицию, чтобы заполнить брешь, возникающую при удалении записи.

Для тех программ, в которых велика интенсивность выполнения операций удаления, возникает задача ускорения этой операции. Стандартный подход в решении этой проблемы основан на использовании поля записи Tag - в него просто записывают соответствующий признак, обозначающий, что запись удалена. На самом деле она удаляется лишь логически. Место в таблице, занимаемое подобными записями, называют ***мусором***. Естественно, что наличие таких записей в таблице должно учитываться в алгоритмах выполнения других операций. По мере накопления мусора появляется

необходимость в его удалении. Для этого предназначена операция сжатия массива. Она запускается оператором по мере необходимости, или автоматически при неудачной попытке включить запись в таблицу.

```
var i,j: Integer;
. . .
j:=0;
for i:=1 to Cnt do
   if Tab[i].Tag<>'D' then
      begin Inc(j); Tab[j]:=Tab[i] end;
Cnt:=j;
```
