How to Code Arrays (aka Tables) …

and Other Data Structures in Rexx

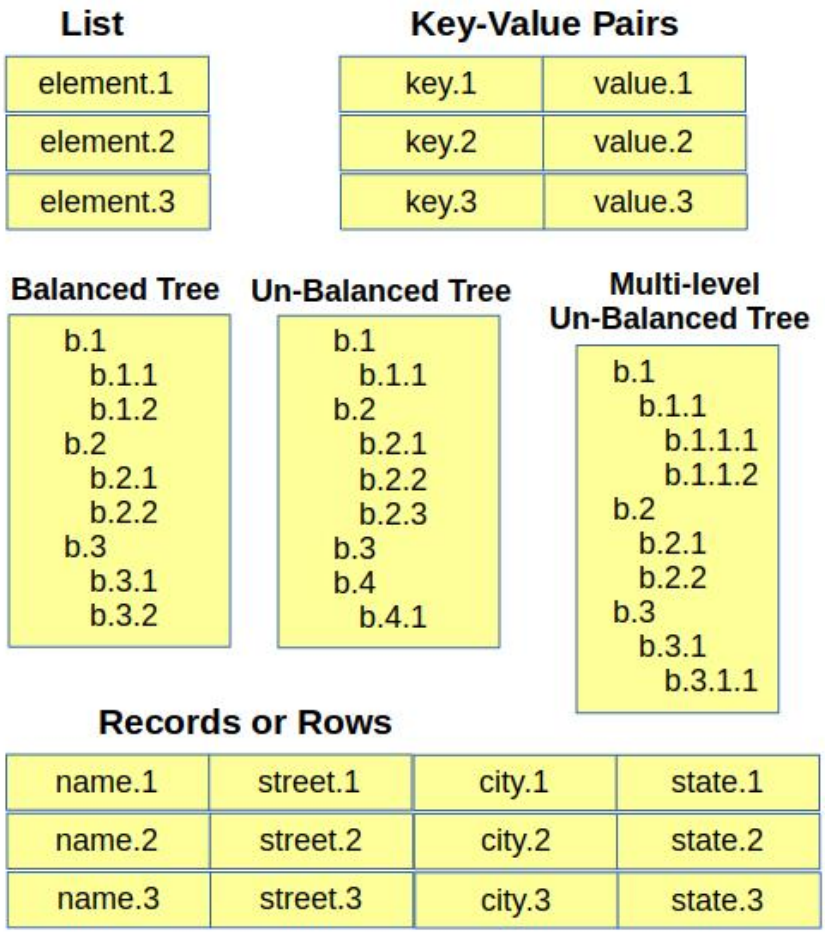
by Howard Fosdick (Originally published in *Enterprise Systems Journal*)

Rexx excels in the manipulation of data through **arrays** -- often called **tables**. It's a key feature of the language that accounts for much of its power.

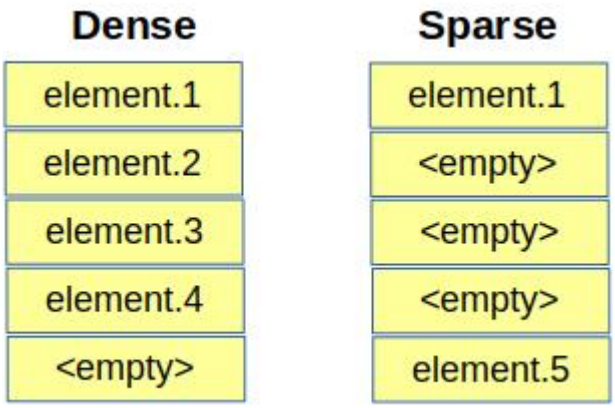
The **compound variable** is the Rexx feature that allows you to create and manage arrays. You use compound variables not only to create arrays or tables, but also to create and use virtually any other kind of data structure including:

- Two- or three- or N- dimensional arrays or tables
- Lists
- Key/value pairs
- Balanced trees
- Unbalanced trees
- Multi-level trees
- Records
- Rows
- and more

This graphic shows how you can use Rexx compound variables to create many different data structures:



Rexx supports both **dense arrays**, in which consecutive table positions each contain a value, and **sparse arrays**, in which some table slots are empty or even uninitialized:



Compound variables also support **associative arrays**, in which the index or subscript that describes array position is a character string, rather than a numeric value. (I'll explain this more later).

While providing this tremendous flexibility, compound variables make coding simple. You don't have to waste time with the complicated syntax that some other scripting languages require.

This article tells you everything you need to know about how to use compound variables to code Rexx arrays. Let's get started.

The Basics

A Rexx **variable** is a name or string that does not begin with a digit and does not contain a period.

A **compound variable** also does not begin with a digit, but it contains one or more periods. Compound variables are used to refer to array (table) elements. Here are some examples:

> **list.i** -- **list** is the name of an array or table. The **stem** of this compound variable is **list.** (it includes the period). **i** is referred to as the **subscript** or **index**.
>
> **books.j.k** -- **books.** is the stem, while **j** and **k** are two subscripts. Arrays can have multiple subscripts.

You do not have to declare an array or its maximum size prior to referring to it. Rexx will automatically size and then expand the array up to the size of available memory.

You can initialize all elements in an array by referring to its stem:

> **list. = 0** /* initializes all possible entries in the array **list** to 0 */
>
> **books. = ''** /* initializes all possible entries in array **books** to the null string */

If your program refers to an array element that has not yet been assigned a value, it returns its own name in uppercase. For example, if you refer to the array element **list.i** and it has not yet been given a value, it will show this value to your program: **LIST.I**

By convention, the 0th element of an array contains the number of elements in the array. So for example, **list.0** would contain the number of elements in the table **list**. Note that since this is only a convention, if you wish to

follow it, it is up to you to code your program to update and maintain the value in the 0th array element. Rexx does not automatically do that for you.

An Example Program

This example program retrieves book titles from a list. To be selected for retrieval, a book title must match at least the number of keyword descriptors the user designates.

The program uses two arrays. The first table contains a list of keywords. It's named the **keyword** table in the code.

The second array contains a list of book titles, each of which has three associated descriptors. This array is named **title** in the code.

The script reads a command line argument from the user called **weight**. This value determines how many of the keywords must match title descriptors in order to retrieve a specific title as a match.

Here's the code:

```
/*  FIND BOOKS:                                                    */
/*                                                                 */
/*     This program illustrates basic arrays by retrieving book    */
/*     titles based on keyword weightings.                         */

keyword. = ''         /* initialize both arrays to all null strings  */
title.   = ''

/* the array of keywords to search for among the book descriptors    */

keyword.1 = 'earth'   ;   keyword.2 = 'computers'
keyword.3 = 'life'    ;   keyword.4 = 'environment'

/* the array of book titles, each having 3 descriptors               */

title.1 = 'Saving Planet Earth'
  title.1.1 = 'earth'
  title.1.2 = 'environment'
  title.1.3 = 'life'
title.2 = 'Computer Lifeforms'
  title.2.1 = 'life'
  title.2.2 = 'computers'
  title.2.3 = 'intelligence'
title.3 = 'Algorithmic Insanity'
  title.3.1 = 'computers'
  title.3.2 = 'algorithms'
  title.3.3 = 'programming'


arg weight .    /* get number keyword matches required for retrieval */

say 'For weight of' weight 'retrieved titles are:'  /* output header */

do j=1 while title.j <> ''                  /* look at each book    */
   count = 0
   do k=1 while keyword.k <> ''             /* inspect its keywords */
      do l=1 while title.j.l <> ''
         if  keyword.k = title.j.l  then count = count + 1
      end
   end

   if count >= weight then   /* display titles matching the criteria */
      say title.j
end
```

Here are some sample runs of the program from the command line. (A blank line has been added after each run of the program for clarity.)

```
bob@Compaq:~/Desktop/testing$ regina find_books 1
For weight of 1 retrieved titles are:
Saving Planet Earth
Computer Lifeforms
Algorithmic Insanity

bob@Compaq:~/Desktop/testing$ regina find_books 2
For weight of 2 retrieved titles are:
Saving Planet Earth
Computer Lifeforms

bob@Compaq:~/Desktop/testing$ regina find_books 3
```

```
For weight of 3 retrieved titles are:
Saving Planet Earth
```

To explain the code, the first two lines of the program initialize all possible entries in the **keyword** and **title** tables to the null string (").

The next code block initializes the **keyword** table to contain four keywords. The **keyword** table is simple list data structure.

Next the **title** array is initialized to contain three book titles. Each book title is initialized so that it is associated with three keyword descriptors. The **title** array is a balanced tree data structure.

The **arg** instruction reads the **weight** assigned by user input. This value should be a number coded as a command line argument or input parameter to the program. This number indicates the minimal number of keywords that must match book descriptors in order for a book title to be retreived.

The **do** loops then determine which book titles match enough keywords to be retrieved. The last line in the program displays those book titles to the user.

Associative Arrays

In the above example program, all the subscripts to the tables are numeric values. Rexx also supports **associative arrays**, in which you subscript table entries by character strings, instead of numbers. Associative arrays allow you to code compact solutions to problems that otherwise might require much more complicated code.

Here's an example. This program contains a table of towns and their associated area codes. The user types in the name of a town to the program, and the program displays its area code in response.

Note how the code uses the town -- a character string -- that is input by the user to look up the proper area code in the table. This is the essence of any associative array:

```
/* CODE LOOKUP:                                                  */
/*                                                               */
/*     Looks up the areacode for the town the user enters.       */

area. = ''      /* pre-initialize all entries to the null string */

area.Chicago  = 312   ;   area.Wrigleyville = 773
area.Homewood = 708   ;   area.Geneva       = 630
area.Zion     = 847   ;   area.Schaumburg   = 847

do while town <> ''
   say 'For which town do you want the area code?'
   pull town .
   if town <> '' then do
      if area.town = ''
         then  say 'Town' town 'is not in my database'
         else  say 'The area code for' town 'is' area.town
   end
end
```

Here is a sample interaction with this program:

```
bob@Compaq:~/Desktop/testing$ regina code_lookup
For which town do you want the area code?
Geneva
The area code for GENEVA is 630

For which town do you want the area code?
Zion
The area code for ZION is 847

For which town do you want the area code?
Louisville
Town LOUISVILLE is not in my database

[The program ends when user presses ENTER with no keystrokes]
```

Summary

This tutorial gave you a quick introduction to how to code arrays (or tables) in Rexx.

Associative arrays can do *much* more than we show here. For real world examples of how their clever use can save hours of processing time, see this article.
