1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

Appendix C: Basics of data structures

Data structures are ways of storing things inside a computer. The efficiency of an artificial intelligence program can be affected a good deal by the choices made for its data structures. We summarize the main kinds in Figure C-1. The many books on this subject provide further information.

Basic data structures are ways of storing small single items of data. Programming languages typically provide as basic data structures (or *data types*) integers, real numbers (in floating point notation), characters(alphanumeric), and bits (things that can be either true or false). Some programming languages also have pointers (actual memory addresses).

The simplest of composite data structures is the *array*. It's just a group of data items stored in successive locations in computer memory. The data items can be any of the aforementioned basic types. So we can speak of an *integer array*, a *real array* or a *character array*; the last is usually called a *string*. To find the data item in the Kth position in a plain (*one-dimensional*) array, you give the *index* K. For a *two-dimensional* array, you give two numbers, a *row* and *column* for a data item. Use of an array requires the start of the array and the length of each dimension.

But arrays are a poor data structure when you must frequently change the data. The *linked lists*, often called just *lists*, are better. Linked lists store sequences of data items like arrays do, but the items do not have to be in successive locations in computer memory; each is paired with a pointer giving the memory location of the next data item in sequence, with the last data item in the list having a special pointer called *nil*. So a list can be scattered around in computer memory, provided you keep item-pointer pairs together and store the location of the first data item. Linked lists need more space to store data items than an array would, but allow more flexibility in the use of computer memory. Lists are more common in artificial-intelligence programming than arrays, and in fact the programming language Lisp is built entirely around lists: lists even represent programs in Lisp.

Several common data structures are special cases of arrays and lists. If you only insert and delete items at the very end of an array or list, then you have a *stack* data structure. Stacks are essential for managing procedure calls and recursion in programming languages, though their operation is usually hidden from the user; such stacks contain pointers to memory locations of program pieces. If you only delete items at the very front of an array or list, and only insert items at the very end, then you have a *queue* data structure. Queues are used when you want to be "fair" to data items and want the data item that has been in the array or list the longest to be the first removed. So queues are first in, first out. Analogously stacks are last in, first out.

Arrays and lists order their items. An unordered set can be stored as an array or list in some random order, but there is an alternative. If data items can take on one of N data values--as for instance, data items for ages of employees in years, which must be positive integers less than 100--then we can represent sets as an N-element *bit array* (an array whose data items are bits) where a "true" in location K of the array means that the Kth possible is present in the set. Bit arrays can make set operations like intersection run very fast, and many computers have bit-handling instructions to speed things even more.

Data structures can arrange data in more complicated ways too. For instance, consider as data items the names of procedures, and suppose we want to represent how those procedures call one another in a nonrecursive program in which every procedure is called only once (e.g. Figure 7-3). A procedure can call

several others, and each of those can call several others, and so on. This sort of branching data structure is called a *tree*. *Binary trees* are those for which every data item has one branch entering and either two or zero branches leaving; they're the most common kind of tree. A tree is usually stored in computer memory as an array of subarrays, each subarray holding a data value and pointers to records of other data items that are linked from it. Trees are useful whenever there are hierarchies or taxonomies in the data. Trees can also be built as an index to sequential data, when you want to be able to find things fast (*search trees*). Trees can be implemented in a list-processing language like Lisp by embedding lists within lists; the same thing can be done in Prolog.

If the branches of a tree can join together, then technically you don't have a tree but a *lattice*. Lattices and trees can be implemented the same way, and they are often confused. Lattices occur often in computer science. For instance, if the same procedure is used in several different places in a program, the hierarchy of procedures can be represented as a lattice. In general, lattices arise whenever you have partial information about ordering of things, but not complete ordering information.

If your data structure has loops so that you can return to some data item by following a sequence of pointers from it, then technically you have a *directed graph*, often just called a *graph*. Graphs are good at modeling arbitrary connections between things. You can implement graphs like trees, as arrays of subarrays containing pointers to other records. Another way is to extend a bit array to a *bit matrix*: for N possible data items, create on N by N matrix of bits so that the entry in row I and column J of this matrix is true if and only if there is a connection from data entry in row I to data item J in the graph. Some graphs require that you store data for each connection, not just for each data item, and for these graphs you can use a matrix but store data other than bits in it. Matrix array implementations are good whenever there are many connections between relatively few data items.

Still more complex data structures can be made by combining the aforementioned ones in various ways. Data items can themselves be data structures; for instance, data items representing English words can be character arrays. Lists can be items of lists themselves, and such *embedded lists* are used extensively in the language Lisp. If you want to represent a graph in Lisp, you can use a list of lists where each sublist contains a data item value followed by a list of pointers. And all the standard list processing of Lisp can be done analogously in Prolog. (But a word of warning: Lisp uses parentheses to denote lists while Prolog uses brackets.)

Recursion (see Appendix B) is useful for manipulating all these data structures. Lists and arrays suggest tail recursion in the form of a single basis case followed by a single induction case, where the basis case concerns a single-item array or list, and the induction case decomposes the problem by eliminating a single item from the list or array. Trees, lattices, graphs, and embedded data structures require more complicated forms of recursion involving either multiple induction cases or multiple recursive procedures. Data structures books provide classic recursive algorithms for all these things and they should be consulted for most simple needs instead of your trying to write programs yourself.

Go to book index