DATA STRUCTURES

Hanan Samet*
Computer Science Department
University of Maryland
College Park, Maryland 20742

The field of data structures is important in geographic information science as it is the foundation for the implementation of many operations. In particular, the efficient execution of algorithms depends on the efficient representation of the data. There has been much research on data structures in computer science with the most prominent work being the encyclopedic treatises of Knuth. In this article we only review the basic data structures.

The simplest way to understand a *data structure* is to view it as a mechanism for capturing information that is usefully kept together. To most people who have any computer programming experience, the first data structure that comes to mind is in the form of a table where the rows correspond to objects where each column represents a different piece of information of a particular type for the object associated with the row. When all of the columns are of the same type, then we have an *array* data structure. As an example of a table, consider an airline reservation system which makes use of a *passenger* data structure where such information as name, address, phone, flight number, destination (e.g., on a multi-stop flight), requiring assistance, etc. would be stored. We use the term *record* to describe the collection of such information for each passenger. We also use the term *field* (also called a *key*) to refer to the individual items of information in the record.

There are several ways in which records are differentiated from arrays. Perhaps the most distinguishing characteristic is the fact that each field can be of a different type. For example, some fields contain numbers (e.g., ``phone number''), others contain letters (e.g., ``name''), while others contain alphanumeric data (e.g., ``address''). Of course, there are other possibilities as well.

Another very important distinguishing characteristic is that each field can occupy a different amount of storage. For example, the ``requiring assistance'' field is binary (i.e., of type *Boolean*) and only requires one bit, while the ``phone number'' field is a number and usually requires just one word. On the other hand, the ``address'' field is a string of characters and requires a variable amount of storage. This is not true for a two-dimensional array representation of a collection of records where all columns contain information of the same type and of the same size.

The manner in which a particular data type is used by the program may also influence its representation. For example, there are many ways of representing numbers. They can be represented as integers, sequences of decimal digits such as binary-coded-decimal (i.e., BCD), or even as character strings using representations such as ASCII, EBCDIC, UNICODE.

So far, we have looked at records as a means of aggregating data of different types. The data that is aggregated need not be restricted to the types that we have seen. It can also consist of instances of the same record type or other record types. In this case our fields contain information in the form of addresses of (called *pointers* or *links* to) other records. For example, returning to the airline reservation system described above, we can enhance the passenger record definition by observing that the reason for the existence of these records is that they are usually part of a flight.

Suppose that we wish to determine all the passengers on a particular flight. There are many ways of answering this query. They can be characterized as being implicit or explicit. First, we must decide on the representation of the flight. Assume that the passenger records are the primitive entities in the system. In this case, to determine all passengers on flight *f*, we need to examine the entire database and check each record individually to see if the contents of its ``flight number'' field is *f*. This is an implicit response and is rather costly.

Instead, we could use an explicit response which is based on aggregating all the records corresponding to all passengers on a particular flight and storing them together. Aggregation of data in this way is usually done by use of a *list* or a *set*, where the key distinction between a set and a list is the concept of order. In a *set*, the order of the items is irrelevant -- the crucial property is the presence or absence of the item. In particular, each item can only appear once in the set. In contrast, in a *list*, the order of the items is the central property. An intuitive definition of a list is that item x_k appears before item x_{k+1}. Items may appear more than once in a list.

There are two principal ways of implementing a list. We can either make use of sequential allocation or linked allocation. Sequential allocation should be familiar as this is the way arrays are represented. In particular, in this case, elements x_k and x_{k+1} are stored in contiguous locations. In contrast, using linked allocation implies the existence of a pointer (also termed *link*) field in each record, say NEXT. This field points to the next element in the list -- that is, NEXT(x_k) contains the address of x_{k+1}.

Both linked and sequential allocation have their advantages and disadvantages. The principal advantage of sequential allocation is that random access is very easy and takes a constant amount of time. In particular, accessing the k^{th} element is achieved by adding a multiple of the storage required by an element of the list to the base address of the list. In contrast, in order to access the *kth* element using linked allocation, we must traverse pointers and visit all preceding *k-1* elements. When every element of the list must be visited, the random access advantage of sequential allocation is somewhat diminished. Nevertheless, sequential allocation is still better in this case. The reason is that for sequential allocation we can march through the list by using indexing. This is implemented in assembly language by using an index register for the computation of the successive addresses that must be visited. In contrast, for linked allocation, the successive addresses are computed by accessing pointer fields which contain physical addresses. This is implemented in assembly language by using indirect addressing, which results in an additional memory access for each item in the list.

The main drawback of linked allocation is the necessity of additional storage for the pointers. However, when implementing complex data structures (e.g., the passenger records in the airline reservation system), this overhead is negligible since each record contains many fields. Linked allocation has the advantage that sharing of data can be done in a flexible manner. In other words, the parts that are shared need not always be contiguous. Insertion and deletion are easy with linked allocation -- that is, there is no need to move data as is the case for sequential allocation. It is relatively easy to merge and split lists with linked allocation. Finally, it may be the case that we need storage for an m element list and we have sufficient storage available for a list of $n > m$ elements, yet by using sequential allocation it could be that we are unable to satisfy the request without repacking (a laborious process of moving storage) because the storage is non-contiguous. Such a problem does not arise when using linked allocation.

In some applications we are given an arbitrary item in a list, say at location j, and we want to remove it in an efficient manner. When the list is implemented using linked allocation, this operation requires that we traverse the list starting at the first element and find the element immediately preceding j. This search can be avoided by adding an additional field called `PREV` which points to the immediately preceding element in the list. The result is called a *doubly-linked* list. Doubly-linked lists are frequently used when we want to make sure that arbitrary elements can be deleted from a list in constant time. The only disadvantage of a doubly-linked list is the extra amount of space that is needed.

The linear list can be generalized to handle data aggregations in more than one dimension. The result is termed an *array*. Arrays are usually represented using sequential allocation. Their principle advantage such a representation is that the cost of accessing each element is the same, whereas this is not the case for data structures that are represented using linked allocation.

The *tree* is another important data structure which is a branching structure between nodes. There are many variations of trees and the distinctions between them are subtle. Formally, a *tree* is a finite set T of one or more nodes such that one element of the set is distinguished. It is called the *root* of the tree. The remaining nodes in the tree form m ($m \geq 0$) disjoint subsets (T_1, T_2.. T_m) where each subset is itself a tree. The subsets are called the *subtrees* of the root. Figure 1 is an example of a tree. The tree is useful for representing hierarchies.

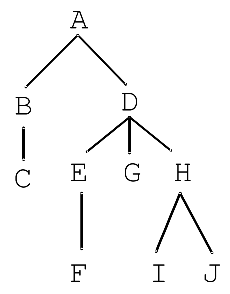

Figure1: Example tree.

The tree is not to be confused with its relative, the *binary tree*, which is a finite set of nodes that is either empty, or contains a root node and two disjoint binary trees called the *left* and *right* subtrees of the root. At a first glance, it would appear that a binary tree is a special case of a tree (i.e., the set of binary trees is a subset of the set of trees). This is wrong! The concepts are completely different. For example, the empty tree is not a ``tree'' while it is a ``binary tree''. Further evidence of the difference is provided by examining the two simple trees in Figure 2. The binary trees in Figures 2a and 2b are different because the former has an empty right subtree while the latter has an empty left subtree. However, as ``trees'' Figures 2a and 2b are identical.

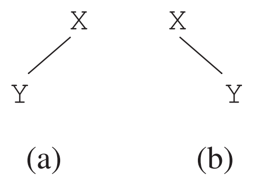

Figure 2: Example of two different binary trees.

Trees find use as a representation of a search structure. In particular, in the case of a set of numbers, a *binary search tree* stores the set in such a way that the values of all nodes in the left subtree are less than the value of the root, and the values of all nodes in the right subtree are greater than the value of the root. For example, Figure 3 is an example of a binary search tree for the set `{10, 15, 20, 30, and 45}`. Binary search trees enable a search to be performed in expected logarithmic time (i.e., proportional to the logarithm of the number *N* of elements in the set). This is in contrast to the list representation where the average time to search the list successfully is *N/2*.

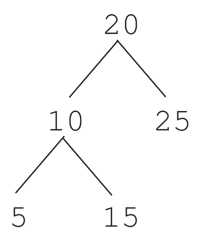

Figure 3: Example binary search tree.

Variants of these data structures are used in many geographic information systems as a means of speeding up search. In particular, the search can either be for a location or a set of locations where a specified object or set of objects are found, or for an object which is located at a given location. There are many such data structures, with variants of the quadtree, which is a multidimensional binary search tree, and R-tree being the most prominent. They are distinguished in part by being either space hierarchies, as are many variants of the former, or object hierarchies, as are the latter. This means that, in the former, the hierarchy is in terms of the space occupied by the objects, while in the latter, the hierarchy is in terms of groups of objects. For more details, see the book by Samet.

Further Readings:

1. D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, vol. 1. Addison-Wesley, Reading, MA, third edition, 1997.

2. D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison-Wesley, Reading, MA, second edition, 1998.

3. H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.