Algorithms

7. Data Structures: Lists, Queues, Stacks and Arrays

R K Shyamasundar

This article continues with the exploration of common data structures. In particular, we shall discuss representation of common data structures such as lists, queues, stacks, trees, and two dimensional arrays.

R K Shyamasundar is Professor of Computer Science at TIFR, Mumbai and has done extensive research in various foundation areas of computer science.

Introduction

In the previous article, we introduced some common data-structures such as one dimensional arrays, trees and a common algorithm design technique called 'divide-and-conquer'. One of the characteristic features of the arrays introduced in the previous article is its ability to access any (randomly, i.e., accessing an element without necessarily going through other elements) element in the array. In this article, we introduce lists, in which accessing and removing elements cannot be done as randomly as in the case of arrays. First, we define the data-structure called *list* in which the i^{th} element can be accessed only after accessing all the elements prior to it. We also describe other variations of lists such as *queues* and *stacks*. The versatility of these data-structures in representing data abstractions is also illustrated. Further, we also show how to generalize single dimensional structures to two or more dimensions.

Previous articles of this series were:
1. Introduction to algorithms, January 1996.
2. The while-construct, March 1996.
3. Procedures and recursion, June 1996.
4. Turtle graphics, September 1996.
5. Data types and their representation in memory, December 1996.
6. Algorithms for Sorting and Searching, March 1997.

Lists, Queues, Stacks, and Arrays

In an earlier article, we studied one-dimensional arrays. In this section, we will introduce data-structures such as lists, queues, stacks, and two-dimensional arrays.

Lists: Let us consider a sequence of items of integers that are sorted and stored as a one dimensional array. Suppose we would like to

In a simple list structure, each item will have two components, the first one corresponds: to the actual value of the item and the second is a pointer to the next item in the list.

- *Insert an element in the list*: If the element to be inserted happens to be the element immediately after the last element then there is no problem. If the position of the new item falls somewhere in between, then we would have to copy the part of the array (assume that there are enough empty slots in the array) that would have to be shifted to the right and insert it.

- *Delete an element*: If there is no need to keep the array compact (having no garbage entry),then there is no problem; we can just set the entry to a known marker or sentinel value. However, if we have to store the elements in a compact form, we need to copy the part of the array that has to be shifted to the left.

Operations described above will be costly when the number of elements are large. In such situations data-structure referred to as *list* is used. In a simple list structure, each item (also referred to as a *cell*) will have two components, the first one corresponds to the actual value of the item and the second is a pointer to the next item in the list. A typical list is depicted in *Figure 1a* .The whole list named 'P0' is a pointer to the first element of the list; which is also called the *head* of the list. The length of the list is nothing but the number of elements in it that are accessible from the head of the list. For example in the list P0 , there are n elements. In the list P1 shown in *Figure 1b*, there are 3 elements before deletion and 2 elements after deletion. Similarly, in the list P2 shown in *Figure 1c*, there are 3 elements before insertion and 4 elements after insertion. It may be observed that, for accessing the i^{th} element, there is a need to go through all elements $j < i$ Note that in the array data structure, it was possible to access any element directly using the index. *Figure 1b* shows how an element can be deleted; note that even though the second element points to the third element it cannot be accessed from the pointer of the list P1. Such elements are called *garbage* elements. *Figure 1c* shows how an element can be inserted by re-orienting the pointers. Thus, the size of the lists can vary with the progress in computation. This is one of the distinct advantages of lists over arrays (in most programming languages, the size of

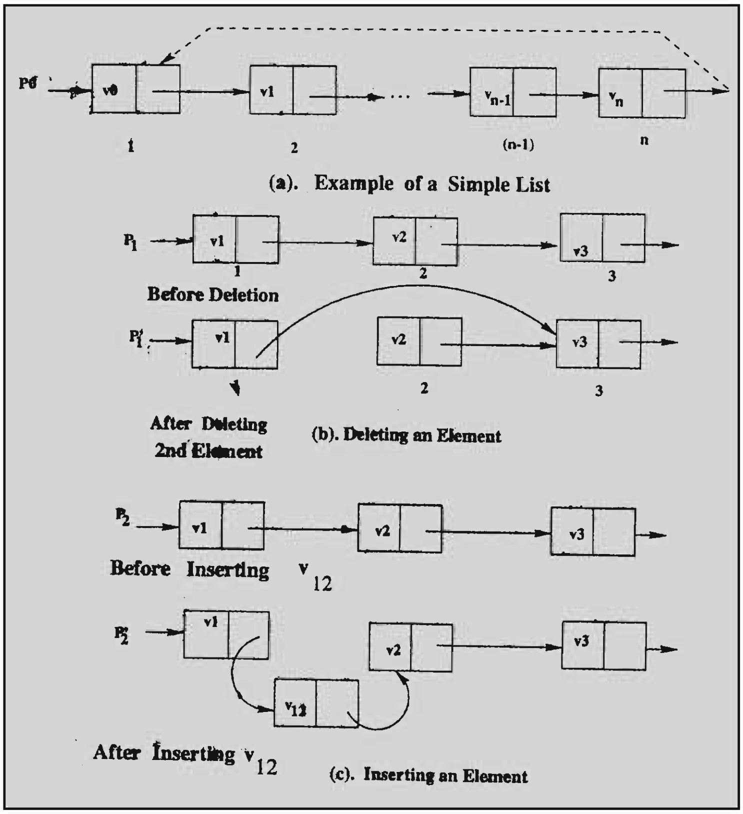
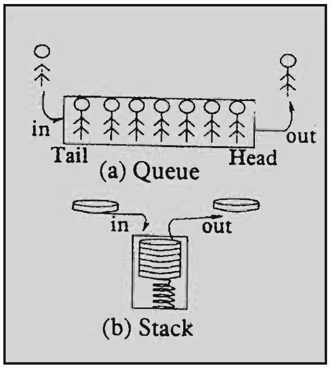


Figure 1 Lists: Insertion and Deletion.

the arrays has to be fixed *a priori*). In other words, the list representation becomes handy when it is not possible to fix the size (or even bound) of the sequence *a priori*.

In case the last element of a list points to the first element then it is referred to as a *circular list*; the dotted arrow in *Figure 1a* shows a pointer from the last element to the first element. The advantage of such a list is the possibility of going from any element to another. This is not always possible in a simple list unless there is a pointer from an element connected from the head to some element already accessed in the list(referred to as *cycles* in the graph theoretic terminology).

Figure 2 A Typical Queue and a Stack.



Queues and Stacks: Abstractly, a *queue* is a list with the restriction: an element can only be removed from the *head* (front) of the queue and an element can be added only to the *tail* (end) of the queue. In other words, a queue is a list that

A queue is a list that follows the rule: the first item to enter the queue is the first to exit.

follows the rule: the first item to enter the queue is the first to exit. This is often referred to as FIFO (First-In-First-Out). *Figure 2a* depicts a queue; the representation will remain as a list described above. However, insertions can only be done at the tail and deletions done only from the head.

A *stack* is a list wherein elements are added at the head of the list and also deleted from there. This data structure is typical of a stack of plates kept in a restaurant, as shown in *Figure 2b*. Only the topmost plate can be 'popped' and another plate can be 'pushed' on top of the stack. Further, if the stack of plates is empty obviously a plate cannot be popped. Thus, a stack is a list that follows the rule: the last element to enter the stack is the first element to be removed; therefore, stacks are referred to as LIFO (Last-In-First-Out). The operations on a stack are usually referred to as push (adding a new element to the top), pop (removing the topmost element) or top (reading the topmost element without deleting it). Again the representation of a list can be used with the restriction that elements can be inserted or deleted only from the head of the list.

It is also possible to realize queues and stacks through the use of array structures by appropriately defining the operations of accessing the elements of the array; for example, one can define procedures *head* and *tail* through which deletion/addition of elements is done.

Trees: In the last article, we saw the utility of 'trees' in arriving at an efficient sorting algorithm. Here, we show how a binary tree can be represented through a generalization of the list.

A stack is a list that follows the rule: the last element to enter the stack is the first element to be removed.

Consider a complete binary tree, where there are either two children from a node or there is no successor. In the list discussed earlier, we had a pointer from one element to its successor. Now, consider an element (or cell) of a list having the following structure: a pointer to the left successor, the value and a pointer to the right successor. A cell with such a structure is a *generalization* of the cell having one value and a single pointer.

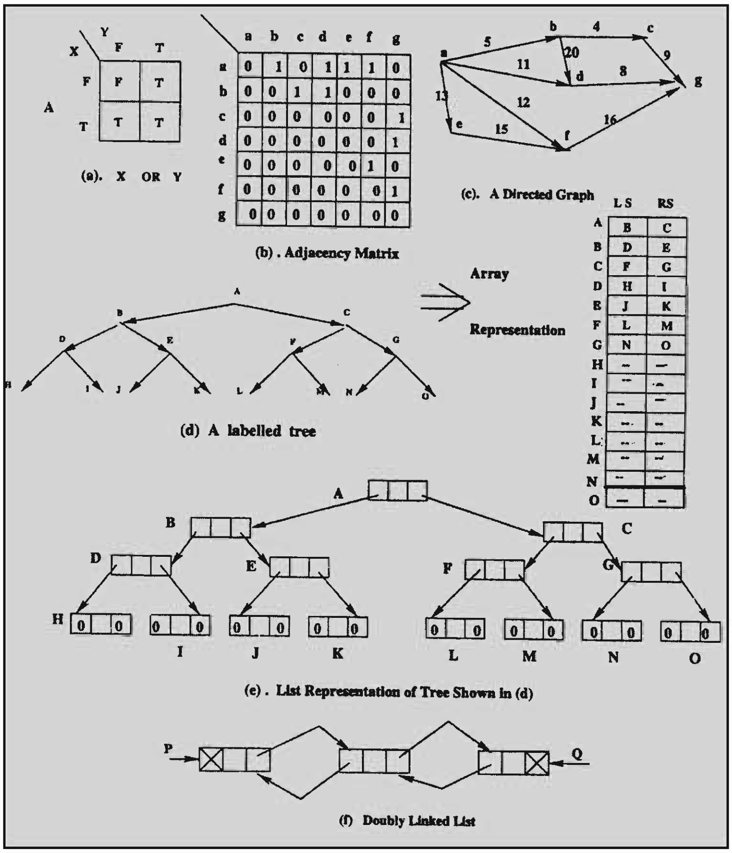


Figure 3 Arrays, doubly linked lists and trees.

Figure 3e is a representation of the abstract tree shown in *Figure 3d.* The nodes H, I, J, K, L, M, N and O have no successors and so, the pointer items of the cells have been initialized to '0'.

We can further generalize the list structure to have a finite set of value components and a finite set of pointer components to indicate the successors. In fact, in a list, we can keep track of the immediate successor as well as the immediate predecessor explicitly. Such a list is referred to as a *doubly linked list.* A typical doubly linked list is shown in *Figure 3f.* The ability to get to either the successor or predecessor not only makes access easy but also enables one to *backtrack* in a search.

The ability to get to either the successor or predecessor in a doubly linked list not only makes access easy but also enables one to backtrack in a search.

Two Dimensional Arrays: It is often useful to arrange data in the form of tables (which have two dimensions). The corresponding data-structure is referred to as a *matrix* or a *two-dimensional array.* Using programming notations, a two-dimensional array is declared as follows:

```
var A: array [1..N, 1..M] of integer;
```

The above declaration denotes that A is an array having N rows and M columns. Applications for arrays are innumerable; the simplest being the classical multiplication table. A table can also be used to store hostel room numbers and codes of the persons staying in the respective rooms. The notation A[8,9] stands for the value of the element located at the 8th row and 9th column in array A. Thus, in the case of two dimensional arrays, two indices are used to access an element instead of a single index as in the case of one-dimensional arrays.

In the case of two dimensional arrays, two indices are used to access an element instead of a single index as in the case of one-dimensional arrays.

In *Figure 3a*, we have described the logical 'OR' through array A; **X OR Y** is true when either **X** or **Y** is true. In the table, **T** denotes true and **F** denotes false. Now A[X,Y] actually represents such an operation.

The array structure is convenient to keep track of several bookkeeping functions. For example, consider keeping track of the students and the courses they have taken. One can take the list of students as one dimension (say rows) and the list of courses as another dimension (say, columns). It can also be used to represent a graph to indicate the adjacencies of nodes or vertices in it. For example, *Figure 3b* is the adjacency matrix of the directed graph shown in *Figure 3c.* In the adjacency graph, $A[i,j]$ will have the entry 1 if there is a directed edge from node i to node j; if there is no directed edge from i to j then the entry is 0. For example, $A[a,e]$ has entry 1 as there is a directed edge from node a to node e. Similarly, $A[d,d]$ has entry 0 as there is no direct edge from node d onto itself.

A complete binary tree representation through two-dimensional arrays is shown in *Figure 3d* where for each node one keeps track of the left-child (denoted LS) and right-child (denoted RS). Absence of a LS or a RS is denoted by '—'. A two-dimensional array can also be seen as an array of arrays. That is, we have a single dimensional array wherein each element of the array is itself an array. A typical specification takes the form:

A: array [1..100] of array of [1..50] of integer.

That is, A is an one dimensional array of 100 elements which are themselves arrays of 50 elements (which are of type integer). Thus, pictorially, the data would look like a rectangular matrix. Now, consider a singie dimensional array whose elements themselves are lists. Such a data structure can be specified by

A: array [1..100] of LIST,

which denotes an array A of 100 elements, each array element being of type list. Since the length of a list need not be fixed beforehand, the above structure represented pictorially need not necessarily be a rectangular matrix. Thus, in the first representation, we should choose the length of the array to be the maximum of all the arrays and leave those entries not required empty; in the latter, the length of the list (i.e., the number of elements) can be varied and thus, we will use only the requisite number of elements.

We shall illustrate the widely used matrix multiplication algorithm using the two dimensional arrays in the following.

Consider two matrices A and B of integer type with dimensions $m \times n$ and $n \times p$ respectively. Then, multiplication of A by B denoted, $A \times B$, is defined by matrix C of dimension $m \times p$ where

$$C[i,j] = \Sigma_{k=1}^{n} A[i,k] \times B[k,j]. \qquad (1)$$

The algorithm for matrix multiplication is described in *Table 1*. After initializing each element of C to zero, the new value is

Suggested Reading

- N Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall. Englewood Cliffs. New Jersey, 1976.
- D Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley Publishing Co. Inc., 1987.

Table 1 Program for Multiplying Two Matrices

```
(* Program for  computing matrices X and Y and placing the result in C *)
var A: array [1..M, 1..N] of integer; (* M and N are given constants *)
  B: array [1..N, 1..P] of integer; (* N and P are given constants *)
  C: array [1..M, 1..P] of integer;
  i, j, k : integers;       (* Used as Indices *)
begin
  for i:= 1, M do
         for j:= 1, P do
                C[i, j] := 0;  (* Initializes the element *)
                for k:= 1, N do
                       C[i, j] := C[i, j] + A[i, k] * B[k, j]
                endfor;
         endfor;
     endfor;
end
```

computed by the body of the innermost for-loop (which corresponds to (1) above). For computing $C[i, j]$, N executions of the body of the inner most for-loop are performed. Thus, the total number of operations required is proportional to $M \times N \times P$ (follows from the three for-loops).

The two-dimensional arrays can be generalized in a natural way. A k-dimensional array that has M_i elements in the i^{th} dimension is specified below:

var A: array [1.. M_1, 1..M_2,..., 1.. M_k] of integer;

Such a structure can also be treated as an array of arrays. For example, A can be treated as an one dimensional array whose elements are arrays of $k-1$ dimensional arrays.

Address for Correspondence
R K Shyamasundar
Computer Science Group
Tata Institute of Fundamental Research
Homi Bhabha Road
Mumbai 400 005, India
email:shyam@tcs.tifr.res.in
Fax:022-215 2110

Discussion

In this article, we have seen various common data-structure abstractions, and their representations. In the next article, we shall discuss algorithm design techniques such as balancing, greedy technique, dynamic programming strategy and backtracking.