Fundamentals of list structures and a pascal implementation of basic list processing techniques.

Mary J. Capece

Follow this and additional works at: http://preserve.lehigh.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

FUNDAMENTALS OF LIST STRUCTURES AND A PASCAL IMPLEMENTATION OF BASIC LIST PROCESSING TECHNIQUES

BY

Mary J. Capece

A Thesis
Presented to the Graduate Committee
of Lehigh University

in Candidacy for the Degree of
Master of Science

in
Computer Science

Lehigh University
1977
CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

MAY 6, 1977
(date)

Professor in Charge

Chairman of the Department
ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my advisor, Professor Samuel L. Gulden for providing his invaluable direction and inspiration in the preparation of this thesis.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>I. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>II. The List Concept</td>
<td>5</td>
</tr>
<tr>
<td>III. Representation of Lists</td>
<td>7</td>
</tr>
<tr>
<td>IV. Types of Lists</td>
<td>12</td>
</tr>
<tr>
<td>V. A List Processing System Embedded in PASCAL</td>
<td>17</td>
</tr>
<tr>
<td>VI. A LISP Interpreter</td>
<td>28</td>
</tr>
<tr>
<td>Bibliography</td>
<td>35</td>
</tr>
<tr>
<td>Appendix I: Program TEST</td>
<td>36</td>
</tr>
<tr>
<td>Appendix II: Program DEAL</td>
<td>49</td>
</tr>
<tr>
<td>Appendix III: Program HUFFMAN</td>
<td>60</td>
</tr>
<tr>
<td>Appendix IV: Program LISP</td>
<td>72</td>
</tr>
<tr>
<td>Vita</td>
<td>84</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure 1</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>29</td>
</tr>
</tbody>
</table>
Abstract

The construction of a program requires a well-designed algorithm as well as careful attention to the design of associated data structures. The linked list is a particularly useful structure type.

Let A be a nonempty set of objects called atoms. We distinguish one particular atom, called the NIL atom and designated by \( \Lambda \). Let \( L_0 = A \). We define the sets \( L_1, L_2, \ldots, L_n, \ldots \) as follows:

Suppose \( L_0, \ldots, L_k \) have been defined, \( K \geq 0 \). Define \( L_{k+1} \) to be the set of all sequences \( a_1, \ldots, a_m, m \geq 1 \), where \( a_1, \ldots, a_{m-1} \in (L_0 \cup L_1 \cup \ldots \cup L_k) \setminus \{\Lambda\} \) and \( a_m = \Lambda \). We call the members of \( L_n \) the linear lists of order less than or equal to \( n \).

In order to represent this list structure in computer memory, we utilize and maintain a set of nodes, each including a symbol field and a link to the next node. Since each node of the list contains a pointer to the next node, successive list elements are not required to be consecutive words in computer memory. This ability to utilize arbitrary, disjoint sections of memory is one of the powerful features of lists.

The operations on list structures normally consist of
accessing an element or series of elements, moving them to other lists, replacing them by other series, or processing the entities represented by them.

This paper describes the various types of list structures and explains the concepts behind list processing techniques. Several list processing methods are presented. The PASCAL programming language is used to implement a list processing system, in order that the reader may obtain a working knowledge of this beautifully simple and powerful aspect of programming.
I. Introduction

Computer programs consist of algorithms which transform informational structures. An informational structure consists of a collection of relations and properties on a basic set of elements or atoms. The construction of a program requires, in addition to a well-designed algorithm, careful attention to the design of associated informational structures.

Since computer programs are frequently designed to facilitate the processing of complicated situations, the informational structures required in such programs may be quite intricate. A particular structure type which has been used effectively in the development of informational structures is that of the list.

The use of lists and their manipulation has all too often been restricted to a few specialists in several narrow areas. Moreover, the most frequently used languages, FORTRAN and COBOL, do not permit the easy use of list processing techniques. Despite all this, list processing is capable of a wide area of application and should be known by more programmers.

It is still the case that many programmers feel that list processing techniques are quite complicated. We will see that there is nothing magic or mysterious about the methods of dealing with complex structures. List processing should be one of the
many techniques at the disposal of programmers, for use in those parts of programs which require it.

The purpose of this paper is to explain the concepts behind list processing techniques in order that the reader may obtain a working knowledge of this beautifully simple and powerful aspect of programming.

It will be shown how several list processing methods can be easily embedded and used in the language PASCAL.

The basic concepts of list processing may be found in [4], [7], [9], and [13].
II. The List Concept

Let $A$ be a non-empty set of objects. We distinguish one particular object and designate it by $\Lambda$. The objects of $A$ are called atoms and in particular $\Lambda$ is called the nil atom. Let $L_0 = A$. We define the sets $L_1, L_2, \ldots, L_n, \ldots$ as follows:

Suppose $L_0, \ldots, L_k$ have been defined, $k \geq 0$. Define $L_{k+1}$ to be the set of all sequences $a_1, \ldots, a_m, m \geq 1$, where $a_1, \ldots, a_{m-1} \in (L_0 \cup L_1 \cup \ldots \cup L_k) \cup \{\Lambda\}$ and $a_m = \Lambda$.

We call the members of $L_n$ the linear lists of order less than or equal to $n$. A list is said to be of order $n$ if and only if it has order less than or equal to $n$, but not order less than or equal to $n-1$.

For notational purposes, if $a_1, \ldots, a_p, \Lambda$ is a list, we write it as $(a_1, \ldots, a_p)$. Observe then that if $A = \{a, b, c, \Lambda\}$, then a list of order two might have the form $(a, ((a,b), c), (a,b), a)$.

Of course there are infinitely many lists of order $n$ for each $n \geq 1$. The latter is true even when $A$ is finite.

In order to realize the list structure in computer memory, we utilize and maintain a set of nodes. Each node consists of one or more consecutive words of computer memory, divided into named parts called fields. Every node includes a link field and
a symbol field. The link component contains the address of the node to be regarded as the successor of the node in question. The symbol component may represent any defined informational structure, e.g. a number, a string of characters, or other information. It may contain the address of another node and thus refer to another sequence of symbols.

Thus, since the items of a list may themselves be lists, the general structure obtained in this manner is called a list structure. Since a list element may contain a pointer to another list, it is possible to build up list structures of arbitrary complexity. Ordinarily these are tree structures, but it is possible to share sublists, build circular structures, etc.

Since each element of a list points to (that is, contains the address of) its successor, successive list elements are not required to be consecutive words in computer memory. This ability to utilize arbitrary, disjoint sections of memory is one of the powerful features of lists.
III. Representations of Lists

For simple programs, the space required for execution is known and allocated prior to execution. Suppose, however, we wish to store a set of numbers the size of which will not be known until the reading is completed. In order to make efficient use of space, the program should allocate space during execution. The techniques of list processing grew in solution to this type of problem.

Consider a program which is intended to read in a sentence, arrange the words in alphabetical order, and then print them in this order. Assume we store each word at a new address in memory. This might appear as follows:

```
100  FOUR
101  SCORE
102  AND
103  SEVEN
104  YEARS
105  AGO
```

where the column of numbers on the left indicates the storage location. Arranging the words such that they are ordered alphabetically yields the following:

```
200  AGO
201  AND
202  FOUR
203  SCORE
204  SEVEN
205  YEARS
```
An alternate approach eliminates the unnecessary duplication of words. We may create a vector which represents the alphabetical ordering by indicating the address of each of the words:

<table>
<thead>
<tr>
<th>300</th>
<th>105</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>102</td>
</tr>
<tr>
<td>302</td>
<td>100</td>
</tr>
<tr>
<td>303</td>
<td>101</td>
</tr>
<tr>
<td>304</td>
<td>103</td>
</tr>
<tr>
<td>305</td>
<td>104</td>
</tr>
</tbody>
</table>

where the numbers on the right are the contents of the locations numbered on the left. This has no effect upon the cells containing the actual characters. Additional words may be incorporated with this scheme without disturbing those currently present or the vectors already in existence. The idea is that it may be advantageous to manipulate the addresses of quantities rather than the quantities themselves. Such is the fundamental basis of list processing.

Consider the computer representation of a sentence in storage. The store for each word will also contain the address of the location of the next word in the sentence.

<table>
<thead>
<tr>
<th>100</th>
<th>THE, 101</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>BOY, 102</td>
</tr>
<tr>
<td>102</td>
<td>WALKED, 103</td>
</tr>
<tr>
<td>103</td>
<td>TO, 104</td>
</tr>
<tr>
<td>104</td>
<td>SCHOOL, Λ</td>
</tr>
</tbody>
</table>

Recall, the use of the greek letter lambda (Λ) denotes the end of a list. This structure may be represented
diagrammatically as follows:

THE → BOY → WALKED → TO → SCHOOL

Words can be added to or deleted from this sentence without moving the existing words, since the sequence of stores in which the words occur is insignificant.

100 THE 105
101 BOY, 102
102 WALKED, 103
103 TO, 104
104 SCHOOL, 106
105 LITTLE, 101
106 TODAY, A.

Linked storage representation allows the possibility of random insertions and deletions. These frequently used list operations are thus accomplished through simple manipulation of pointers. With sequential allocation of storage, insertion is particularly difficult since it may involve shifting a large number of elements. This also holds for deletion if we are to utilize deleted storage space. Insertion and deletion are much simpler with linked lists, as we need only alter the appropriate linkages.
The linked list lends itself immediately to more intricate structures. We can maintain a variable number of variable size lists; any node of the list may be a starting point for another list, the nodes may simultaneously be linked together in several orders, corresponding to different lists.

Suppose the items in the chain are addresses, for example addresses of strings of letters or perhaps addresses of other chains.

<table>
<thead>
<tr>
<th>Location</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>LIST</td>
</tr>
<tr>
<td>101</td>
<td>PROCESSING</td>
</tr>
<tr>
<td>102</td>
<td>COMPUTER</td>
</tr>
<tr>
<td>103</td>
<td>PROGRAMMING</td>
</tr>
<tr>
<td>200</td>
<td>101, 201</td>
</tr>
<tr>
<td>201</td>
<td>103, Λ</td>
</tr>
<tr>
<td>202</td>
<td>203, 205</td>
</tr>
<tr>
<td>203</td>
<td>100, 204</td>
</tr>
<tr>
<td>204</td>
<td>101, Λ</td>
</tr>
<tr>
<td>205</td>
<td>206, Λ</td>
</tr>
<tr>
<td>206</td>
<td>102, 207</td>
</tr>
<tr>
<td>207</td>
<td>103, Λ</td>
</tr>
</tbody>
</table>

A chain starts at location 200 which consists of two items - a pointer to the word "processing", and a pointer to the word "programming":

![Diagram](PROCESSING -> PROGRAMMING)

At location 202 begins another chain consisting of just two items. The first item is a chain of two items - a pointer to the word "list" and a pointer to the word "processing". The second item
is also a chain of two items - a pointer to the word "computer" and a pointer to the word "programming". This list structure is diagrammed as follows:

```
  L  →  P  →  C  →  A
  |    |    |    |
 LIST | PROCESSING | COMPUTER | PROGRAMMING
```

The objects which do not have the two-pointer nature represent the atoms. Their structure is not the concern of the particular program which is operating on the list in which they occur.
IV. Types of Lists

The conventions for the ending of lists may be altered. The space at the end of each sublist can indicate the place in the main list from which the sublist has been referenced. The cell at the end of a sublist must provide an indication that it is an end point, and not a continuation of the sublist. Extra space must be available for storing tag markers to imply this. When the procedure reaches a point in the list that is tagged as end of a sublist, then attention is transferred back to the main list. The following representation of the list \((A, (B,C), D)\) illustrates this concept:

![Diagram of list structure]

where * denotes the marker for the end of a sublist. Note that the sublist \((B,C)\) points to its referencing node in the main list and, therefore, cannot be a sublist anywhere else. This scheme has the serious disadvantage that a list can only be a sublist of one list, and if required as part of another list, then it must be duplicated. Some problems suffer
severely from shortage of store if common sublists do not exist.

Operations on list structures normally consist of accessing an element or a series of elements, moving them to other lists, replacing them by other series, or processing the entities represented by them. Accessing an element in a list is usually restricted to the first element after a particular given element. Thus it is possible to access any list element, but only by traversing the list from the first element. This is the situation with simple linked lists.

Doubly linked lists. If each node of a list has two links, pointing to the nodes on either side of it, then a more flexible method of handling lists is obtained at the expense of extra storage space for links. This is intended to make movement about the list easily possible in both directions, as is illustrated in the following diagram:

```
LEFT → \[ A \] → \[ \] → \[ \] → \[ \] → \[ \] → \[ \] → \[ \] → \[ \] → \[ \] → \[ A \] ← RIGHT
```

Here, LEFT and RIGHT are pointer variables to the left and right of the doubly linked list. Each node includes two links, called, for example, LLINK and RLINK.

Manipulations of doubly linked lists almost always become much easier if a list head node is part of each list. We have
the following typical representation:

```
LIST HEAD
```

The LLINK and RLINK fields of the list head replace LEFT and RIGHT in the previous illustration. If the list is empty, then both link fields of the head point to the head itself.

This representation clearly satisfies the condition that

\[ \text{RLINK(LLINK(X))} = \text{LLINK(RLINK(X))} = X \]

where \( X \) is the location of any node in the list (including the head). It is for this reason that a list head is desirable.

In addition to the obvious advantage of the ability to move in either direction when examining a doubly linked list, one of the important new abilities is that we can delete a node \( X \) from the list containing it, given only the value of \( X \). In a simple linked list with only one-directional links, we cannot delete the node \( X \) without knowing its predecessor in the chain, since the link of the preceding node requires alteration in performing a deletion of the node \( X \).

Suppose we wish to write a particular routine to search a list of atoms to find the predecessor of a given atom \( A \). With singly linked list structures, it becomes necessary to keep track of two atoms at all times as we search the list. Everytime we compare an atom with \( A \), its predecessor must be known, in
the event of a match between the current atom and the atom A. With doubly linked list structures, however, this is not necessary. The desired result can be obtained by first locating A, and then ensue its predecessor pointer. A doubly linked list also permits easy insertion of a node adjacent on either side.

The obvious disadvantages of doubly linked lists are that more memory space is required, and more pointers need be manipulated than with singly linked lists.

**Circular lists.** A circular list is a list in which every element is the successor of exactly one other element of the list. A circular list possesses the property that its last node links back to the first node, instead of to A. There is no need to think of a first or last element. We require only one pointer to the list. The entire circular list may be accessed from any given node of that list.

The following diagram illustrates a (singly-linked) circular list. The nodes have just two fields: INFO and LINK:

![Circular list diagram](image)

Circular lists can represent not only inherently circular structures, but also linear structures. A circular list with one pointer to the rear node is essentially equivalent to a simple linked list with two pointers, one to the front and one
to the rear.

In view of the circular symmetry, and since there is no link to signal the end, how do we recognize the end of the list? We must record our starting point, process the list as desired, and stop when we encounter the starting node (assuming that node is still present in the list). An alternate solution is to include a special recognizable node in each circular list, as a convenient stopping point. This list head is quite convenient for applications. An obvious advantage is that the circular list will then never be empty.

Given only X, it is possible to delete the node X in a circular list. This is accomplished by progressing through the entire list in order to locate the predecessor of X. This operation may be inefficient. Some operations, however, become very efficient with circular lists. For example, it is very convenient to move an entire list to become part of another list, or to divide a circular list into two lists.
V. A List Processing System Embedded in PASCAL

While there may exist some programming tasks best solved entirely within some list processing system, most tasks facing the ordinary programmer require the application of a number of distinct techniques. Many programs contain sections which are suitable for list processing. The packaging of a variety of tools within a single tool box seems to be the best way to outfit a worker setting out to solve complex problems.

We shall use the PASCAL programming language and embed in it various procedures to implement a list processing system. Familiarity with the PASCAL language is assumed [6]. The task of understanding these new techniques, then, is that of adding to a vocabulary rather than that of learning an entirely new one. The ideas for the approach taken here come from [11].

PASCAL provides pointer variables as a simple tool for the construction of complicated and flexible data structures. The lists considered here (with the exception of the free list) are circular.

The declaration part of the main program will define the type PTR as a pointer to NODE, where NODE is defined as a record type including a LINK field of type PTR. Also, a variable identifier FREE of type PTR must be declared. Nodes are deleted by moving them to the list containing all freed nodes. FREE
will point to this list.

We shall first need a procedure to initialize the free list, which is initially empty. The free list is the only non-circular list being considered in this section. Procedure ORG performs the initialization:

```
PROCEDURE ORG;
BEGIN
    FREE := NIL;
END;
```

A very useful function is one which acts on a pointer variable P by finding its antecedent in the list. P remains unchanged. The value of the function is the pointer to the antecedent of P.

```
FUNCTION ANTE(P : PTR) : PTR;
VAR
    TEMP : PTR;
BEGIN
    TEMP := P;
    WHILE TEMP^.LINK ≠ P DO
        TEMP := TEMP^.LINK;
    ANTE := TEMP;
END;
```

A node, pointed to by P, which is to be "erased" is moved to the free list by Procedure RELEASE. P is changed to point to its successor in the original list, unless P was from a list of just one element. In this case, P is set to NIL.
PROCEDURE RELEASE (VAR P : PTR);
VAR
    TEMP, PTI : PTR;
BEGIN
    IF P↑.LINK = P THEN PTI := NIL
    ELSE
        BEGIN
            TEMP := ANTE(P);
            TEMP↑.LINK := P↑.LINK;
            PTI := P↑.LINK;
        END;
    P↑.LINK := FREE;
    FREE := P;
    P := PTI;
END;

Procedure RELIST (P,Q) may be used in the same way as
RELEASE, the only difference being that it will free the string
of nodes starting with that node pointed to by P and ending
with the node pointed to by Q. Q remains unchanged. P be-
comes what was Q↑.LINK unless P through Q was the entire
list. In that case, P is set to NIL.

PROCEDURE RELIST (VAR P:PTR; Q:PTR);
VAR
    TEMP, PTI:PTR;
BEGIN
    IF Q↑.LINK = P THEN PTI := NIL
    ELSE
        BEGIN
            TEMP := ANTE(P);
            TEMP↑.LINK := Q↑.LINK;
            PTI := Q↑.LINK;
        END;
    Q↑.LINK := FREE;
    FREE := P;
    P := PTI;
END;
Procedure ALLOCATE (P) allocates a variable of type PTR
and assigns its address to P. This is done utilizing nodes
from the free list, if there are any. Storage space is generated
dynamically by the procedure new if the free list is empty.

PROCEDURE ALLOCATE (VAR P : PTR);
BEGIN
  IF FREE = NIL THEN NEW (P)
  ELSE BEGIN
    P := FREE;
    FREE := FREE^.LINK;
  END;
END;

A new (circular) list of one element may be established by
means of the Procedure INIT (P). P then points to that one node.

PROCEDURE INIT (VAR P : PTR);
BEGIN
  ALLOCATE (P)
  P^.LINK := P;
END;

The Procedure INSERT (P) creates a variable of type PTR and
inserts it as the antecedent of the variable to which P points
in the list containing P. P becomes the pointer to this newly
created variable.
PROCEDURE INSERT (VAR P : PTR);
VAR
   TEMP, PTI : PTR;
BEGIN
   ALLOCATE (TEMP);
   PTI := ANTE (P);
   PTI ↑. LINK := TEMP;
   TEMP ↑. LINK := P;
   P := TEMP;
END;

One of the most important processes in list structuring is
the moving of list elements from one list to another. The
Procedure MOV (P,Q) moves the element to which P points such
that it becomes the antecedant of the element to which Q points.
Q is set equal to P and P becomes the pointer to what was
its successor, unless P is an entire list. In that case,
moving the node to which P points eliminates that list and
thus, P is set to NIL.

PROCEDURE MOV (VAR P,Q : PTR);
VAR
   TEMP, PTI, PT2 : PTR;
BEGIN
   IF P ↑. LINK = P THEN TEMP := NIL
   ELSE
      BEGIN
         PTI := ANTE (P);
         PTI ↑. LINK := P ↑. LINK;
         TEMP := P ↑. LINK;
      END;
   PT2 := ANTE (Q);
   PT2 ↑. LINK := P;
   P ↑. LINK := Q;
   Q := P;
   P := TEMP;
END;
The following example shows a simple use of MOV(P,Q).

Before:     (a,b,c,d,e)          (f,g,h,i)
        +                  +
        P                  Q

After:      (a,b,d,e)          (f,g,c,h,i)
        +                  +
        P                  Q

Suppose we now call MOV(P,Q) again:

Then we have     (a,b,e)        (f,g,d,c,h,i)
        +                  +
        P                  Q

Now a call of MOV(Q,P) will yield the following

(a,b,d,e)        (f,g,c,h,i)
        +                  +
        P                  Q

And a second call to MOV(Q,P) brings us back where we started:

(a,b,c,d,e)          (f,g,h,i)
        +                  +
        P                  Q

Procedure INCOR(,Q,R) given below may be used in the same way as MOV, the only difference being that it will move each of the nodes starting with that to which P points and ending with the node to which Q points. This string of elements is then inserted to precede the node to which R points. Q remains unchanged. R is set to P, and P becomes Q +. LINK unless
P through Q is an entire list. If this is so, then P is set to NIL.

```pascal
PROCEDURE INCOR (VAR P:PTR; Q:PTR; VAR R:PTR);
VAR
  TEMP, PTI,PT2 : PTR;
BEGIN
  IF Q^.LINK = P THEN TEMP := NIL
  ELSE
    BEGIN
      PTI = ANTE (P);
      PTI^.LINK := Q^.LINK;
      TEMP := Q^.LINK;
    END;
  PT2 := ANTE (R);
  PT2^.LINK := P;
  Q^.LINK := R;
  R := P;
  P := TEMP;
END;
```

The following illustrates the effect of Procedure INCOR (P,Q,R):

Before: 
```
  (a,b,c,d,e,f,g)   (h,i,j,k)
  +    +    +
  P    Q    R
```

After:
```
  (a,f,g)     (h,b,c,d,e,i,j,k)
  +    +    +
  P    R    Q
```

Function ELEM (P,N) will have as its value the pointer to the n-th element after the element to which P points. P remains unchanged.
FUNCTION   ELEM (P:PTR, N:INTEGER) : PTR;
VAR
  I : INTEGER;
BEGIN
  FOR I := 1 TO N DO
    P := P +. LINK;
    ELEM := P;
END;

Program TEST is included as the Appendix I so that the reader may inspect the performance of a few of these procedures.

As a simple example of the use of these list processing techniques, we consider the dealing of a deck of cards in a bridge game [11]. Declare a node to be a record with three fields: card value, card suit, and a link to the next card in the list. A program to simulate the deal has been written in five sections.

1. INITIALIZE. In this procedure, we seed a random number generator, call Procedure ORG to initialize the free list, and set the symbols J,Q,K,A to represent the jack, queen, king, and ace of each suit.

2. GENDECK. This procedure generates a circular list containing a node for each of the fifty-two different combinations of card values and suits. The pointer variable, DECK, will designate the list by pointing to an arbitrary node.

3. STARTLISTS. Sixteen lists are initialized - four per player (one for each of the four suits). M is a four by four
array containing pointers to the first element for each of the sixteen lists.

4. DEALDECK. In this procedure, a card is randomly chosen from the remainder of the deck. The card is removed from the deck list and placed in one of the sixteen lists initialized in STARTLISTS (which list depends upon the suit of the card drawn, and which player is to receive the card).

5. PRINT. This procedure prints to output the hands of the four players, with cards listed according to suit. That is, the sixteen lists of STARTLISTS are printed. As each card is printed, it is moved from its current list to the deck list. At the conclusion of this procedure, the deck is reconstructed.

The program listing and an actual run are presented in Appendix II in Program DEAL. One deal requires about 0.5 CP seconds.

As another example of the use of list processing techniques, we consider the construction of a Binary Huffman Code [5]. Given a message source of N possible messages, \( N \geq 1 \), each with its own probability of occurrence, the process is as follows:

1. Organize the possible messages according to probability of occurrence, in descending order.

2. Combine the two messages with lowest probabilities by drawing lines from each to a single point. Label the line
of the more frequent message with a "1" and the line of the other message with "0".

3. Combine the next two messages with lowest probabilities, and label them.

4. Continue this process until all messages are merged at one point.

5. Read the labels along the path from the unique point to each symbol for its code.

For example, suppose we have four messages with probabilities of occurrence as follows:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>PROBABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.45</td>
</tr>
<tr>
<td>B</td>
<td>.40</td>
</tr>
<tr>
<td>C</td>
<td>.10</td>
</tr>
<tr>
<td>D</td>
<td>.05</td>
</tr>
</tbody>
</table>

Then the following diagram illustrates the construction of a Huffman Code for this message source.

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>PROBABILITY</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.45</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>.40</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>.10</td>
<td>101</td>
</tr>
<tr>
<td>D</td>
<td>.05</td>
<td>100</td>
</tr>
</tbody>
</table>
The computer program to construct a Huffman Code first reads the symbols and their corresponding probabilities of occurrence. Each symbol and corresponding probability is represented in a node of a circular list. This list is then sorted according to the probability field, in descending order. Then, the two nodes of lowest probability are removed from this list to form a new circular list with head. They are substituted by a single element with a probability equal to the combined probabilities of the other two nodes. This substitute element points at, and is pointed at by the head of the circular list formed by the nodes with low probabilities. Now the original list is sorted again, and the two nodes with lowest probabilities are combined as before. This process is repeated until just one element exists in the list.

Now the list structure is complete, and we need only traverse it properly to obtain a Huffman Code. See Program HUFFMAN in Appendix III for the program listing and sample run.
VI. A LISP Interpreter

LISP lists are simple singly linked structures. Each list element contains a pointer to a data item and a pointer to the following list element. The last element of a list points to the special atom NIL. The two pointers in a list element are termed the CAR pointer and the CDR pointer. The CAR pointer indicates the data item while the CDR pointer indicates the successor to that list item. The CAR value of a list item may be a pointer to an atom, or to another list.

Of the elementary LISP operations, CAR and CDR dissect a list, giving as values the left and right pointers, respectively. Suppose X is the list ((A),B,C,(D,E,((F))),G).

Diagrammatically, this list may be represented as in figure 1. Then CAR(X) = (A) and CDR(X) = (B,C,(D,E,((F))),G). These operations may be applied successively so, for example, CAR(CDR(X)) = B and CDR(CDR(X)) = (C,(D,E,((F))),G). The functions CAR and CDR are undefined on atomic objects. Note that successive elements of a list X are given by CAR(X), CAR(CDR(X)), CAR(CDR(CDR(X))), CAR(CDR(CDR(CDR(X)))),...

To construct a list, the operator CONS is used. If X is an atom or a list and Y is a list, then CONS(X,Y) has as its value a new list cell whose left pointer indicates X and whose right
Figure 1
pointer indicates Y. To form a list of one element, say A, we have that CONS(A,NIL) = (A). Although it would be possible to allow the second parameter of CONS to be an atom, we shall not do so but shall preserve the convention that items of a list are shown by the left pointer and that the right pointer (the second parameter) links the remaining list cells. The only exception to this is that the special atom NIL may appear as the second parameter.

Note that CAR(CONS(A,B)) = A and CDR(CONS(A,B)) = B. But CONS(CAR(X), CDR(X)) is a new cell in storage which contains the same pointers as did X. It is a copy of X, not the cell X itself.

The function ATOM(X) has the value *T* (representing "true") if X is an atom and the value NIL otherwise. Function EQ(X,Y) has the value *T* if the two atoms, X and Y, are identical. Otherwise, its value is NIL. EQ(X,Y) is undefined if either X or Y is not an atom.

To implement this system in PASCAL, we create a circular list with head, to keep track of all atoms. This list is initialized to contain nodes with name NIL and *T*. Two types of nodes are considered: atomic and nonatomic. Atomic nodes contain two fields - one for NAME and another for a LINK to other elements of the atomlist. Nonatomic nodes contain two
fields, a HEAD and a TAIL, both pointers.

When a list containing atoms is input to the program, the names of the atoms are inserted in the atomlist, unless they already appear there.

TREW is the name of the pointer to the atom whose name is "*T*" and NILL is the name of the pointer to the atom whose name is "NIL". The Function ATOM(L1) assumes either the value TREW or the value NILL, depending upon whether L1 is an atom or not:

```pascal
FUNCTION ATOM(L1 : PTR) : PTR;
BEGIN
  IF L1 +. STATE = ATOMIC THEN ATOM := TREW
  ELSE ATOM := NILL;
END;
```

The value of the Function CONS(L1, L2) is a pointer to the cell whose head is L1 and whose tail is L2:

```pascal
FUNCTION CONS (L1, L2 : PTR) : PTR;
VAR
  Q : PTR;
BEGIN
  NEW(Q):
  WITH Q + DO
  BEGIN
    STATE := NONATOMIC;
    HEAD := L1;
    TAIL := L2;
  END;
  CONS := Q :
END;
```

The Function CAR(L1) assumes as its value a pointer to the
head of the list \( L_1 \). If \( L_1 \) is an atom, the function is undefined.

FUNCTION \textsc{Car}(L1 : PTR) : PTR;
BEGIN
IF ATOM(L1) = TRUE THEN ERROR(1)
ELSE CAR := L1 +. HEAD;
END;

The Function \textsc{Cdr}(L1) assumes as its value a pointer to the tail of the list \( L_1 \). If \( L_1 \) is an atom, then the function is undefined.

FUNCTION \textsc{Cdr}(L1 : PTR) : PTR;
BEGIN
IF ATOM(L1) = TRUE THEN ERROR(2)
ELSE CDR := L1 +. TAIL;
END;

Function \textsc{Eq}(L1,L2) takes on the value \textsc{True} or \textsc{Nil}, depend-
upon whether the atoms \( L_1 \) and \( L_2 \) are identical. If either \( L_1 \) or \( L_2 \) is not an atom then the function is undefined.

FUNCTION \textsc{Eq}(L1, L2 : PTR) : PTR;
BEGIN
IF (ATOM(L1) = NIL) or (ATOM(L2) = NIL)
THEN ERROR (4)
ELSE IF L1 = L2 THEN EQ := TRUE
ELSE EQ := NIL;
END;

Function \textsc{Equal}(L1, L2) performs exactly as does Function \textsc{Eq}(L1, L2), except that \( L_1, L_2 \) need not be atoms, and the general list structures, \( L_1 \) and \( L_2 \), are tested for equality.
FUNCTION EQUAL (L1, L2 : PTR) : PTR;
BEGIN
IF (ATOM(L1) = TREW) AND (ATOM(L2) = TREW)
THEN EQUAL := EQ(L1, L2)
ELSE
IF EQUAL (CAR(L1), CAR(L2)) = TREW
THEN EQUAL := EQUAL (CDR(L1), CDR(L2))
ELSE EQUAL := NILL;
END;

Thus, we have the basic LISP operations. Let us discover what can be accomplished with these functions.

Consider the Function FLAT(L1, L2) which accepts the general list L1 and creates another list containing the same atoms in the identical order as in L1, but with all atoms on the same level. This flattened version of L1 is placed in front of L2 for the final result. For example, let
X = ((A),B,(C,D,(E,F),G)). Then FLAT(X,NILL) is a pointer to the list structure (A,B,C,D,E,F,G). With the use of the techniques defined in this section, Function FLAT is defined with only one program statement.

Another usage of these techniques occurs in Function REV(L1,L2). This function reverses the top level of the list L1, and places it in front of L2. Suppose L1 is the list ((A),B,C,(D,E,F),G). Then REV(L1,NILL) indicates the list (G,(D,E,F),C,B,(A)). Also, the programming for this function requires only one statement.
Function EVALUATE (L1) takes the list L1 to be the Polish notation of an arithmetic expression, and creates the corresponding infix notation for its evaluation. This effort is greatly simplified by the list processing techniques presented here.

Procedure PRINT (L1) performs a preorder traversal of the list L1 (also, a tree) in order to write to output the list corresponding to the internal computer representation.

These programs illustrate the utility of general list processing techniques, and are included in Program LISP in Appendix IV for the reader's inspection.
BIBLIOGRAPHY


PROGRAM TEST(INPUT, OUTPUT);

TYPE
  PTR = * NODE;
  NODE = RECORD
    VAL: INTEGER;
    NAME: ALFA;
    LINK: PTR;
  END;

VAR
  x: REAL;
  FREE: PTR;
  i: INTEGER;
  j: INTEGER;
  el: PTR;

PROCEDURE ORG;
(* INITIALIZE THE FREE LIST *)
BEGIN
  FREE := NIL;
END (*ORG*);

FUNCTION ANTE(P: PTR): PTR;
(* POINTS TO THE ANTECEDENT OF P *)
(* P REMAINS UNCHANGED *)

VAR
  TEMP: PTR;

BEGIN
  TEMP := P;
  WHILE TEMP^.LINK <> P DO
    TEMP := TEMP^.LINK;
  ANTE := TEMP;
END (*ANTE*);

PROCEDURE RELIST(VAR P: PTR; Q: PTR);
(* MOVES THE STRING OF ELEMENTS, BEGINNING WITH THE ELEMENT TO WHICH P POINTS,*)
AND ENDING WITH THE ELEMENT TO WHICH Q POINTS *)
(* INSERTS IT IN THE FREE LIST *)
(* IF P THRU Q IS AN ENTIRE LIST, THEN P := NIL ELSE P := Q^.LINK; *)
(* Q REMAINS UNCHANGED *)

VAR
    TEMP: PTR;
    PT1: PTR;
BEGIN
    IF Q^.LINK = P
    THEN
        PT1 := NIL
    ELSE
        BEGIN
            TEMP := ANTE(P);
            TEMP^.LINK := Q^.LINK;
            PT1 := Q^.LINK;
        END;
        Q^.LINK := FREE;
        FREE := P;
        P := PT1;
    END (*RELIST*);
END

PROCEDURE ALLOCATE(VAR P: PTR);
(* CREATES A VARIABLE OF TYPE PTR, POINTED TO BY P *)
(* UTILIZES SPACE FROM THE FREE LIST, IF THERE IS ANY *)
BEGIN
    IF FREE = NIL
    THEN
        NEW(P)
    ELSE
        BEGIN
            P := FREE;
            FREE := FREE^.LINK;
        END;
END (*ALLOCATE*);

PROCEDURE INIT(VAR P: PTR);
(* ESTABLISHES A NEW CIRCULAR LIST OF ONE ELEMENT *)
(* P POINTS TO THAT ELEMENT *)
BEGIN
ALLOCATE(P);
P+.LINK := P;
END (*INIT*);

PROCEDURE INSERT(VAR P: PTR);
(* CREATES A VARIABLE OF TYPE PTR *)
(* INSERTS IT AS THE ANTECEDENT OF THE VARIABLE TO WHICH
P POINTS *)
(* P BECOMES THE POINTER TO THIS NEWLY CREATED VARIABLE *)

VAR
TEMP: PTR;
PTI: PTR;
BEGIN
ALLOCATE(TEMP);
PTI := ANTE(P);
PTI+.LINK := TEMP;
TEMP+.LINK := P;
P := TEMP;
END (*INSERT*);

FUNCTION ELEM(P: PTR; N: INTEGER): PTR;
(* POINTS TO THE N-TH ELEMENT AFTER THE ELEMENT TO WHICH
P POINTS *)
(* P REMAINS UNCHANGED *)

VAR
I: INTEGER;
BEGIN
FOR I := 1 TO N DO
P := P+.LINK;
ELEM := P;
END (*ELEM*);

FUNCTION RANDOM: REAL;
EXTERN;
PROCEDURE SKIP(N: INTEGER);

VAR
  I: INTEGER;

BEGIN
  FOR I := 1 TO N DO
    WRITELN;
END (*SKIP*);

PROCEDURE WRITEPTR(PT: PTR);

BEGIN
  WRITELN(PT#.NAME: 7, PT+.NAME: 10,
           PT#.VAL: 2, PT+.LINK#.ORD: 7);
END (*WRITEPTR*);

PROCEDURE WRIEFORELIST(LIST: PTR);

VAR
  EL: PTR;
  I: INTEGER;

BEGIN
  EL := LIST;
  IF EL = NIL THEN
    WRITELN(EL#.PT = NIL);
  ELSE
    BEGIN
      I := 1;
      REPEAT
        WRITE(EL#.E, I: 3, EL#.E);
        WRITEPTR(EL);
        I := I + 1;
        EL := EL#.LINK;
      UNTIL (EL = LIST) OR (EL = NIL);
    END;
  END (*WRIEFORELIST*);
PROCEDURE TESTELEM;

VAR
  PT: PTR;
  I, N: INTEGER;

BEGIN
  WRITELN(\textasciitilde TESTING FUNCTION ELE\textasciitilde);  
  SKIP(1);
  WRITE(\textasciitilde EL: \textasciitilde);
  WRITEPTR(EL);
  SKIP(2);
  FOR I := 1 TO 3 DO
    BEGIN
      N := TRUNC(RANDOM * 7);
      PT := ELEM(EL, N);
      WRITE(\textasciitilde N=\textasciitilde, N\# 2, \textasciitilde \textasciitilde);
      WRITEPTR(PT);
      SKIP(1);
    END;
  END (*TESTELEM*);

PROCEDURE TESTANTE;

VAR
  N: INTEGER;
  PT: PTR;

BEGIN
  WRITELN(\textasciitilde TESTING FUNCTION ANTE\textasciitilde);  
  SKIP(1);
  WRITE(\textasciitilde EL: \textasciitilde);
  WRITEPTR(EL);
  SKIP(2);
  FOR I := 1 TO 3 DO
    BEGIN
      N := TRUNC(RANDOM * 7);
      PT := ELEM(EL, N);
      WRITE(\textasciitilde N=\textasciitilde, N\# 2, \textasciitilde \textasciitilde);
      WRITEPTR(PT);
      WRITE(\textasciitilde ANTE\textasciitilde);
      WRITEPTR(ANTE(PT));
      SKIP(1);
    END;
  END (*TESTANTE*);
PROCEDURE TESTRELIST;

VAR
    TEMPI PTR;
    I, NI INTEGER;

BEGIN
    WRITELN("TESTING PROCEDURE RELIST");
    SKIP(1);
    WRITELN("EL - LIST");
    WRITELIST(EL);
    SKIP(1);
    WRITELN("FREE - LIST");
    WRITELIST(FREE);
    SKIP(3);
    I := 7;
    WRITE("EL: ");
    WRITEPTR(EL);
    N := TRUNC(RANDOM * I);
    TEMP := ELEM(EL, N);
    WRITE("ELEM(EL, N: ");
    WRITEPTR(TEMP);
    SKIP(1);
    RELIST(EL, TEMP);
    WRITELN("EL - LIST");
    WRITELIST(EL);
    SKIP(1);
    WRITELN("FREE - LIST");
    WRITELIST(FREE);
    SKIP(3);
    I := I - (N + 1);
    IF I > 0 THEN
        BEGIN
            RELIST(EL, ANTE(EL));
            WRITELN("EL - LIST");
            WRITELIST(EL);
            SKIP(1);
            WRITELN("FREE - LIST");
            WRITELIST(FREE);
        END;
    END (*TESTRELIST*);

PROCEDURE TESTALLOCATE;
VAR

I: INTEGER;

BEGIN

WRITELN('TESTING PROCEDURE ALLOCATE');
SKIP(1);
WRITELN('FREE - LIST');
WRITELIST(FREE);
SKIP(3);
FOR I := 1 TO 10 DO
BEGIN
ALLOCATE(EL);
WRITE(EL); WRITE(EL);
WRITELN(EL, ' ', ORD(EL) + 7);
SKIP(1);
WRITELN('FREE - LIST');
WRITELIST(FREE);
SKIP(3);
END;
END (*TESTALLOCATE*);

BEGIN (*TEST*)

FOR I := 1 TO CLOCK MOD 750 DO
X := RANDOM;
ORG;
SKIP(4);
FOR I := 1 TO 7 DO
BEGIN
IF I = 1
THEN
INIT(EL)
ELSE
INSERT(EL);
FOR J := 1 TO 10 DO
EL+.NAME[J] := CHR(I);
EL+.VAL := I;
END;
WRITELN('EL - LIST');
WRITELIST(EL);
SKIP(4);
TESTELEM;
SKIP(4);
TESTANTE;
SKIP(4);
TESTRELIST;
SKIP(4);

43
TESTALLOCATE;
SKIP(4);
END (*TEST*).
**EL - LIST**

<table>
<thead>
<tr>
<th>No.</th>
<th>Pt</th>
<th>Name</th>
<th>Val</th>
<th>Pt-&gt;Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2100709</td>
<td>GGGGGGGGGGG</td>
<td>7</td>
<td>1838569</td>
</tr>
<tr>
<td>2</td>
<td>1838569</td>
<td>FFFFFFFFFFF</td>
<td>6</td>
<td>1576429</td>
</tr>
<tr>
<td>3</td>
<td>1576429</td>
<td>EEEEEEEEEEE</td>
<td>5</td>
<td>1314289</td>
</tr>
<tr>
<td>4</td>
<td>1314289</td>
<td>DDDDDDDDDD</td>
<td>4</td>
<td>1052149</td>
</tr>
<tr>
<td>5</td>
<td>1052149</td>
<td>CCCCCCCCCCC</td>
<td>3</td>
<td>790009</td>
</tr>
<tr>
<td>6</td>
<td>790009</td>
<td>BBBBBBBBBBB</td>
<td>2</td>
<td>527869</td>
</tr>
<tr>
<td>7</td>
<td>527869</td>
<td>AAAAAAAAAAA</td>
<td>1</td>
<td>2100709</td>
</tr>
</tbody>
</table>

**TESTING FUNCTION ELEM**

<table>
<thead>
<tr>
<th>El:</th>
<th>Pt</th>
<th>Name</th>
<th>Val</th>
<th>Pt-&gt;Link</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2100709</td>
<td>GGGGGGGGGGG</td>
<td>7</td>
<td>1838569</td>
</tr>
<tr>
<td>N=5</td>
<td>790009</td>
<td>BBBBBBBBBBB</td>
<td>2</td>
<td>527869</td>
</tr>
<tr>
<td>N=4</td>
<td>1052149</td>
<td>CCCCCCCCCC</td>
<td>3</td>
<td>790009</td>
</tr>
<tr>
<td>N=0</td>
<td>2100709</td>
<td>GGGGGGGGGGG</td>
<td>7</td>
<td>1838569</td>
</tr>
</tbody>
</table>

**TESTING FUNCTION ANTE**

<table>
<thead>
<tr>
<th>El:</th>
<th>Pt</th>
<th>Name</th>
<th>Val</th>
<th>Pt-&gt;Link</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2100709</td>
<td>GGGGGGGGGGG</td>
<td>7</td>
<td>1838569</td>
</tr>
<tr>
<td>N=6</td>
<td>527869</td>
<td>AAAAAAAAAAAAA</td>
<td>1</td>
<td>2100709</td>
</tr>
<tr>
<td>ANTE:Pt</td>
<td>790009</td>
<td>BBBBBBBBBBB</td>
<td>2</td>
<td>527869</td>
</tr>
<tr>
<td>N=3</td>
<td>1314289</td>
<td>DDDDDDDDDDD</td>
<td>4</td>
<td>1052149</td>
</tr>
<tr>
<td>ANTE:Pt</td>
<td>1576429</td>
<td>EEEEEEEEEEE</td>
<td>5</td>
<td>1314289</td>
</tr>
<tr>
<td>N=6</td>
<td>527869</td>
<td>AAAAAAAAAAAAA</td>
<td>1</td>
<td>2100709</td>
</tr>
<tr>
<td>ANTE:Pt</td>
<td>790009</td>
<td>BBBBBBBBBBB</td>
<td>2</td>
<td>527869</td>
</tr>
<tr>
<td></td>
<td>PT</td>
<td>NAME</td>
<td>VAL</td>
<td>PT*.LINK</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>------------</td>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>1</td>
<td>2100709</td>
<td>GGGGGGGGGG</td>
<td>7</td>
<td>1838569</td>
</tr>
<tr>
<td>2</td>
<td>1838569</td>
<td>FFFFFFFFFF</td>
<td>6</td>
<td>1576429</td>
</tr>
<tr>
<td>3</td>
<td>1576429</td>
<td>EEEEEEEEE</td>
<td>5</td>
<td>1314289</td>
</tr>
<tr>
<td>4</td>
<td>1314289</td>
<td>DDDDDDDDD</td>
<td>4</td>
<td>1052149</td>
</tr>
<tr>
<td>5</td>
<td>1052149</td>
<td>CCCCCCCCCC</td>
<td>3</td>
<td>790009</td>
</tr>
<tr>
<td>6</td>
<td>790009</td>
<td>BBBBBBBBBB</td>
<td>2</td>
<td>527869</td>
</tr>
<tr>
<td>7</td>
<td>527869</td>
<td>AAAAAAAAAA</td>
<td>1</td>
<td>2100709</td>
</tr>
</tbody>
</table>

FREE - LIST
PT = NIL

EL - LIST
1 PT 2100709 NAME GGGGGGGGGG VAL 7 PT*.LINK 1838569
2 PT 1838569 NAME FFFFFFFFFF VAL 6 PT*.LINK 1576429
3 PT 1576429 NAME EEEEEEEEEE VAL 5 PT*.LINK 1314289
4 PT 1314289 NAME DDDDDDDDDD VAL 4 PT*.LINK 1052149
5 PT 1052149 NAME CCCCCCCCCC VAL 3 PT*.LINK 790009
6 PT 790009 NAME BBBBBBBBBB VAL 2 PT*.LINK 527869
7 PT 527869 NAME AAAAAAAAAA VAL 1 PT*.LINK 2100709

FREE - LIST
PT = NIL

EL - LIST
1 PT 527869 NAME AAAAAAAAAA VAL 1 PT*.LINK 527869

FREE - LIST
1 PT 2100709 NAME GGGGGGGGGG VAL 7 PT*.LINK 1838569
2 PT 1838569 NAME FFFFFFFFFF VAL 6 PT*.LINK 1576429
3 PT 1576429 NAME EEEEEEEEEE VAL 5 PT*.LINK 1314289
4 PT 1314289 NAME DDDDDDDDDD VAL 4 PT*.LINK 1052149
5 PT 1052149 NAME CCCCCCCCCC VAL 3 PT*.LINK 790009
6 PT 790009 NAME BBBBBBBBBB VAL 2 PT*.LINK 527869
7 PT 527869 NAME AAAAAAAAAA VAL 1 PT*.LINK 2100709

FREE - LIST
PT = NIL

EL - LIST
1 PT 527869 NAME AAAAAAAAAA VAL 1 PT*.LINK 2100709
2 PT 2100709 NAME GGGGGGGGGG VAL 7 PT*.LINK 1838569
3 PT 1838569 NAME FFFFFFFFFF VAL 6 PT*.LINK 1576429
4 PT 1576429 NAME EEEEEEEEEE VAL 5 PT*.LINK 1314289
5 PT 1314289 NAME DDDDDDDDDD VAL 4 PT*.LINK 1052149
6 PT 1052149 NAME CCCCCCCCCC VAL 3 PT*.LINK 790009
7 PT 790009 NAME BBBBBBBBBB VAL 2 PT*.LINK 131071

FREE - LIST
PT = NIL

EL - LIST
1 PT 527869 NAME AAAAAAAAAA VAL 1 PT*.LINK 2100709
2 PT 2100709 NAME GGGGGGGGGG VAL 7 PT*.LINK 1838569
3 PT 1838569 NAME FFFFFFFFFF VAL 6 PT*.LINK 1576429
4 PT 1576429 NAME EEEEEEEEEE VAL 5 PT*.LINK 1314289
5 PT 1314289 NAME DDDDDDDDDD VAL 4 PT*.LINK 1052149
6 PT 1052149 NAME CCCCCCCCCC VAL 3 PT*.LINK 790009
7 PT 790009 NAME BBBBBBBBBB VAL 2 PT*.LINK 131071
TESTING PROCEDURE ALLOCATE

FREE - LIST
1  PT  527869  NAME  AAAAAAAAAA  VAL  1  PT+.LINK  2100709
2  PT  2100709  NAME  GGGGGGGGGG  VAL  7  PT+.LINK  1838569
3  PT  1838569  NAME  FFFFFFFFFFF  VAL  6  PT+.LINK  1576429
4  PT  1576429  NAME  EEEEEEEEE  VAL  5  PT+.LINK  1314289
5  PT  1314289  NAME  DDDDDDDDDD  VAL  4  PT+.LINK  1052149
6  PT  1052149  NAME  CCCCCCCCCC  VAL  3  PT+.LINK  790009
7  PT  790009  NAME  BBBBBBBBBBB  VAL  2  PT+.LINK  131071

EL:  PT  527869

FREE - LIST
1  PT  2100709  NAME  GGGGGGGGGG  VAL  7  PT+.LINK  1838569
2  PT  1838569  NAME  FFFFFFFFFFF  VAL  6  PT+.LINK  1576429
3  PT  1576429  NAME  EEEEEEEEE  VAL  5  PT+.LINK  1314289
4  PT  1314289  NAME  DDDDDDDDDD  VAL  4  PT+.LINK  1052149
5  PT  1052149  NAME  CCCCCCCCCC  VAL  3  PT+.LINK  790009
6  PT  790009  NAME  BBBBBBBBBBB  VAL  2  PT+.LINK  131071

EL:  PT  2100709

FREE - LIST
1  PT  1838569  NAME  FFFFFFFFFFF  VAL  6  PT+.LINK  1576429
2  PT  1576429  NAME  EEEEEEEEE  VAL  5  PT+.LINK  1314289
3  PT  1314289  NAME  DDDDDDDDDD  VAL  4  PT+.LINK  1052149
4  PT  1052149  NAME  CCCCCCCCCC  VAL  3  PT+.LINK  790009
5  PT  790009  NAME  BBBBBBBBBBB  VAL  2  PT+.LINK  131071

EL:  PT  1838569

FREE - LIST
1  PT  1576429  NAME  EEEEEEEEE  VAL  5  PT+.LINK  1314289
2  PT  1314289  NAME  DDDDDDDDDD  VAL  4  PT+.LINK  1052149
3  PT  1052149  NAME  CCCCCCCCCC  VAL  3  PT+.LINK  790009
4  PT  790009  NAME  BBBBBBBBBBB  VAL  2  PT+.LINK  131071
EL*: PT 1576429

FREE - LIST
1 PT 1314289 NAME DDDDDDDDDDD VAL 4 PT+.LINK 1052149
2 PT 1052149 NAME CCCCCCCCCC VAL 3 PT+.LINK 790009
3 PT 790009 NAME BBBBBBBBBBB VAL 2 PT+.LINK 131071

EL*: PT 1314289

FREE - LIST
1 PT 1052149 NAME CCCCCCCCCC VAL 3 PT+.LINK 790009
2 PT 790009 NAME BBBBBBBBBBB VAL 2 PT+.LINK 131071

EL*: PT 1052149

FREE - LIST
1 PT 790009 NAME BBBBBBBBBBB VAL 2 PT+.LINK 131071

EL*: PT 790009
FREE - LIST
PT = NIL

EL*: PT 2362849
FREE - LIST
PT = NIL

EL*: PT 2624989
FREE - LIST
PT = NIL

EL*: PT 2887129
FREE - LIST
PT = NIL
Appendix II: Program DEAL
PROGRAM DEAL(OUTPUT);

TYPE
  COLOR = (SPADES, HEARTS, DIAMONDS, CLUBS);
  PTR = ^NODE;
  NODE = RECORD
    VAL: 0..14;
    SUIT: COLOR;
    LINK: PTR;
  END;

VAR
  FREE: PTR;
  DECK: PTR;
  M: ARRAY [1..4, COLOR] OF PTR;
  X1, X2: INTEGER;

PROCEDURE ORG;
(* INITIALIZE THE FREE LIST *)
BEGIN
  FREE := NIL;
END (*ORG*);

FUNCTION ANTE(P: PTR): PTR;
(* POINTS TO THE ANTECEDENT OF P *)
(* P REMAINS UNCHANGED *)

VAR
  TEMP: PTR;

BEGIN
  TEMP := P;
  WHILE TEMP^.LINK <> P DO
    TEMP := TEMP^.LINK;
  ANTE := TEMP;
END (*ANTE*);

PROCEDURE RELEASE(VAR P: PTR);
(* MOVES THE ELEMENT TO WHICH P POINTS, TO THE FREE LIST *)
(* IF P IS AN ENTIRE LIST, THEN P := NIL ELSE P := P^.LINK *)

VAR
   TEMP: PTR;
   PT1: PTR;

BEGIN
   IF P^.LINK = P
   THEN
      PT1 := NIL
   ELSE
      BEGIN
         TEMP := ANTE(P);
         TEMP^.LINK := P^.LINK;
         PT1 := P^.LINK;
      END;
   P^.LINK := FREE;
   FREE := P;
   P := PT1;
END (*RELEASE*);

PROCEDURE RELIST(VAR P: PTR; Q: PTR);
(* MOVES THE STRING OF ELEMENTS, BEGINNING WITH THE ELEMENT TO WHICH P POINTS, AND ENDING WITH THE ELEMENT TO WHICH Q POINTS *)
(* INSERTS IT IN THE FREE LIST *)
(* IF P THRU Q IS AN ENTIRE LIST, THEN P := NIL ELSE P := Q^.LINK; *)
(* Q REMAINS UNCHANGED *)

VAR
   TEMP: PTR;
   PT1: PTR;

BEGIN
   IF Q^.LINK = P
   THEN
      PT1 := NIL
   ELSE
      BEGIN
         TEMP := ANTE(P);
         TEMP^.LINK := Q^.LINK;
         PT1 := Q^.LINK;
      END;
   FREE := P;
   P := PT1;
END (*RELIST*);
PROCEDURE ALLOCATE(VAR P: PTR);
(* CREATE A VARIABLE OF TYPE PTR, POINTED TO BY P *)
(* UTILIZES SPACE FROM THE FREE LIST, IF THERE IS ANY *)
BEGIN
  IF FREE = NIL
  THEN
    NEW(P)
  ELSE
    BEGIN
      P := FREE;
      FREE := FREE*.LINK;
    END;
END (*ALLOCATE*);

PROCEDURE INIT(VAR P: PTR);
(* ESTABLISHES A NEW CIRCULAR LIST OF ONE ELEMENT *)
(* P POINTS TO THAT ELEMENT *)
BEGIN
  ALLOCATE(P);
  P*.LINK := P;
END (*INIT*);

PROCEDURE INSERT(VAR P: PTR);
(* CREATE A VARIABLE OF TYPE PTR *)
(* INSERTS IT AS THE ANTECEDENT OF THE VARIABLE TO WHICH P POINTS *)
(* P BECOMES THE POINTER TO THIS NEWLY CREATED VARIABLE *)
VAR
  TEMP: PTR;
  PT1: PTR;
BEGIN
ALLOCATE(TEMP);
PT1 := ANTE(P);
PT1^.LINK := TEMP;
TEMP^.LINK := P;
P := TEMP;
END (*INSERT*);

PROCEDURE MOV(VAR P, Q: PTR);
(* MOVES THE ELEMENT TO WHICH P POINTS, SUCH THAT IT IS
THE ANTECEDENT
OF THE ELEMENT TO WHICH Q POINTS *)
(* IF P IS AN ENTIRE LIST THEN P:=NIL ELSE P:=P^.LINK *)
(* Q NOW POINTS TO WHAT WAS ORIGINALLY POINTED TO BY P *)
VAR
PT1, PT2: PTR;
TEMP: PTR;
BEGIN
IF P^.LINK = P
THEN
TEMP := NIL
ELSE
BEGIN
PT1 := ANTE(P);
PT1^.LINK := P^.LINK;
TEMP := P^.LINK;
END;
PT2 := ANTE(Q);
PT2^.LINK := P;
P^.LINK := Q;
Q := P;
P := TEMP;
END (*MOV*);

PROCEDURE INCOR(VAR P: PTR; Q: PTR; VAR R: PTR);
(* MOVES THE STRING OF ELEMENTS, BEGINNING WITH THE ELEM-
ENT TO WHICH
P POINTS, AND ENDING WITH THE ELEMENT TO WHICH Q POINTS *)
(* INSERTS IT TO PRECEDE THE ELEMENT TO WHICH R POINTS *)
(* IF P THRU Q IS AN ENTIRE LIST, THEN P:=NIL ELSE P:=Q*
VAR
  PT1, PT2: PTR;
  TEMP: PTR;
BEGIN
  IF Q+.LINK = P
  THEN
    TEMP := NIL
  ELSE
    BEGIN
      PT1 := ANTE(P);
      PT1+.LINK := Q+.LINK;
      TEMP := Q+.LINK;
    END;
    PT2 := ANTE(R);
    PT2+.LINK := P;
    Q+.LINK := R;
    R := P;
    P := TEMP;
END (*INCOR*);

FUNCTION ELEM(P: PTR; N: INTEGER): PTR;
(* POINTS TO THE N-TH ELEMENT AFTER THE ELEMENT TO WHICH
P POINTS *)
(* P REMAINS UNCHANGED *)
VAR
  I: INTEGER;
BEGIN
  FOR I := 1 TO N DO
    P := P+.LINK;
    ELEM := P;
END (*ELEM*);

FUNCTION RANDOM: REAL;
EXTERN;
PROCEDURE WRITELIST(LIST: PTR);

VAR
PT1: PTR;
I: INTEGER;

BEGIN
PT1 := LIST;
I := 1;
REPEAT
  WRITELN(\$ \$, I := 2, PT1\+.VAL, ORD(PT1\+.SUIT));
  I := I + 1;
  PT1 := PT1\+.LINK;
UNTIL PT1 = LIST;
END (*WRITELIST*);

PROCEDURE INITIALIZE;

VAR
I: INTEGER;
X: REAL;

BEGIN
  FOR I := 1 TO CLOCK MOD 750 DO
    X := RANDOM;
  ORG;
  SYM[12] := EQE;
  SYM[14] := EAE;
END (*INITIALIZE*);

PROCEDURE GEND ECK;

VAR
I: INTEGER;
J: COLOR;

BEGIN
  INIT(D ECK);
  FOR J := SPADES TO CLUBS DO
    FOR I := 2 TO 14 DO
      BEGIN
IF (J <> SPADES) OR (I <> 2) THEN
   INSERT(DECK);
   DECK*.VAL := I;
   DECK*.SUIT := J;
END;
END (*GENDECK*);

PROCEDURE STARTLISTS;

VAR
   I: INTEGER;
   J: COLOR;

BEGIN
   FOR I := 1 TO 4 DO
      FOR J := SPADES TO CLUBS DO
         BEGIN
            INIT(M[I, J]);
            M[I, J] *.VAL := 0;
         END;
   END (*STARTLISTS*);

PROCEDURE DEALDECK;

VAR
   I: INTEGER;
   P: PTR;

BEGIN
   FOR I := 52 DOWNTO 1 DO
      BEGIN
         DECK := ELEM(DECK, TRUNC(RANDOM * I));
         P := M[(I MOD 4) + 1, DECK*.SUIT];
         REPEAT
            P := P*.LINK;
         UNTIL P*.
         VAL := DECK*.VAL;
         MOV(DECK, P);
      END;
   END (*DEALDECK*);

PROCEDURE PRINT;

56
VAR
    ST, NAME: ALFA;
    V, I: INTEGER;
    J: COLOR;
    P: PTR;
BEGIN
    FOR I := 1 TO 4 DO
        BEGIN
            CASE I OF
                1:
                    NAME := ENORTH;
                2:
                    NAME := EEAST;
                3:
                    NAME := ESOUTH;
                4:
                    NAME := EWEST;
            END;
            WRITELN;
            WRITELN(EN, NAME);
            FOR J := SPADES TO CLUBS DO
                BEGIN
                    CASE J OF
                        SPADES:
                            ST := ESPADES;
                        HEARTS:
                            ST := EHEARTS;
                        DIAMONDS:
                            ST := EDIAMONDS;
                        CLUBS:
                            ST := ECLUBS;
                    END;
                    WRITE(EN, ST);
                    P := MI[I, J] +.LINK;
                    V := P+.VAL;
                    WHILE V > 0 DO
                        BEGIN
                            IF V < 11
                                THEN
                                    WRITE(EN, V: 3)
                                ELSE
                                    WRITE(EN, SYM[V]: 3);
                            IF DECK = NIL
                                THEN
                                    BEGIN
                                        INIT(DECK);
                                    END;
                END;
        END;
END;
BEGIN (*DEAL*)
  INITIALIZE;
  GENDECK;
  STARTLISTS;
  WRITELN;
  WRITELN;
  WRITELN;
  WRITELN;
  WRITELN;
  DEALDECK;
  PRINT;
END (*DEAL*).
<table>
<thead>
<tr>
<th>NORTH</th>
<th>SPADES</th>
<th>HEARTS</th>
<th>DIAMONDS</th>
<th>CLUBS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A 8 7 3</td>
<td>Q 9 3</td>
<td>K Q 4 3</td>
<td>K J</td>
</tr>
<tr>
<td>EAST</td>
<td>SPADES</td>
<td>HEARTS</td>
<td>DIAMONDS</td>
<td>CLUBS</td>
</tr>
<tr>
<td></td>
<td>Q 10 6</td>
<td>K</td>
<td>J 8 7 6 5</td>
<td>A Q 7 6</td>
</tr>
<tr>
<td>SOUTH</td>
<td>SPADES</td>
<td>HEARTS</td>
<td>DIAMONDS</td>
<td>CLUBS</td>
</tr>
<tr>
<td></td>
<td>4 2</td>
<td>A 10 8 7 6</td>
<td>A 10 9 2</td>
<td>5 2</td>
</tr>
<tr>
<td>WEST</td>
<td>SPADES</td>
<td>HEARTS</td>
<td>DIAMONDS</td>
<td>CLUBS</td>
</tr>
<tr>
<td></td>
<td>K J 9 5</td>
<td>J 5 4 2</td>
<td>10 9 8 4 3</td>
<td>59</td>
</tr>
</tbody>
</table>
Appendix III: Program HUFFMAN
(*$U*[W1,56] MARY CAPECE *)

PROGRAM HUFFMAN(INPUT, OUTPUT);

TYPE
  PTR = * NODE;
  NODE = RECORD
    LINK: PTR;
    CODE: INTEGER;
    FREQ: REAL;
    CASE CONTINUE: BOOLEAN OF
      TRUE: (MORE: PTR);
      FALSE: (SYMBOL: ALFA);
    END;
  END;

VAR
  FREE: PTR;
  FIRSTREAD: BOOLEAN;
  HEAD: PTR;
  P, Q, R: PTR;

PROCEDURE WRITELIST(LIST: PTR);

VAR
  P: PTR;

BEGIN
  P := LIST^.LINK;
  REPEAT
    WRITE('$ P = ', ORD(P), $ CODE$ = P^.CODE, $ FREQ$ = P^.FREQ);
    CASE P^.CONTINUE OF
      TRUE:
        WRITE('$ MORE = ', ORD(P^.MORE));
      FALSE:
        WRITE('$ SYMBOL = ', P^.SYMBOL);
      END;
    WRITE('$ P^.LINK = ', ORD(P^.LINK));
    WRITELN;
    P := P^.LINK;
  UNTIL P = LIST;
  WRITELN;
END (*WRITELIST*);

PROCEDURE ORG;
(* INITIALIZE THE FREE LIST *)
BEGIN
  FREE := NIL;
END (*ORG*);

FUNCTION ANTE(P: PTR): PTR;
(* POINTS TO THE ANTECEDENT OF P *)
(* P REMAINS UNCHANGED *)
VAR
  TEMP: PTR;
BEGIN
  TEMP := P;
  WHILE TEMP.LINK <> P DO
    TEMP := TEMP.LINK;
    ANTE := TEMP;
  END (*ANTE*);

PROCEDURE RELEASE(VAR P: PTR);
(* MOVES THE ELEMENT TO WHICH P POINTS, TO THE FREE LIST *)
(* IF P IS AN ENTIRE LIST, THEN P := NIL ELSE P := P.LINK *)
VAR
  TEMP: PTR;
  PT1: PTR;
BEGIN
  IF P.LINK = P
  THEN
    PT1 := NIL
  ELSE
    BEGIN
      TEMP := ANTE(P);
      TEMP.LINK := P.LINK;
      PT1 := P.LINK;
    END;
  P.LINK := FREE;
  FREE := P;
  P := PT1;
END (*RELEASE*);
PROCEDURE RELIST(VAR P: PTR; Q: PTR);
(* MOVES THE STRING OF ELEMENTS,
BEGINNING WITH THE ELEMENT TO WHICH P POINTS,
AND ENDING WITH THE ELEMENT TO WHICH Q POINTS *)
(* INSERTS IT IN THE FREE LIST *)
(* IF P THRU Q IS AN ENTIRE LIST, THEN P := NIL ELSE P := Q*.LINK; *)
(* Q REMAINS UNCHANGED *)

VAR
  TEMP: PTR;
  PT1: PTR;

BEGIN
  IF Q*.LINK = P
  THEN
    PT1 := NIL
  ELSE
    BEGIN
      TEMP := ANTE(P);
      TEMP*.LINK := Q*.LINK;
      PT1 := Q*.LINK;
    END;
  Q*.LINK := FREE;
  FREE := P;
  P := PT1;
  END (*RELIST*);

PROCEDURE ALLOCATE(VAR P: PTR);
(* CREATES A VARIABLE OF TYPE PTR, POINTED TO BY P *)
(* UTILIZES SPACE FROM THE FREE LIST, IF THERE IS ANY *)

BEGIN
  IF FREE = NIL
  THEN
    NEW(P)
  ELSE
    BEGIN
      P := FREE;
      FREE := FREE*.LINK;
    END;
  END (*ALLOCATE*);
PROCEDURE INIT(VAR P: PTR);
(* ESTABLISHES A NEW CIRCULAR LIST OF ONE ELEMENT *)
(* P POINTS TO THAT ELEMENT *)
BEGIN
    ALLOCATE(P);
    P+.LINK := P;
END (*INIT*);

PROCEDURE INSERT(VAR P: PTR);
(* CREATES A VARIABLE OF TYPE PTR *)
(* INSERTS IT AS THE ANTECEDENT OF THE VARIABLE TO WHICH P POINTS *)
(* P BECOMES THE POINTER TO THIS NEWLY CREATED VARIABLE *)
VAR
    TEMP: PTR;
    PT1: PTR;
BEGIN
    ALLOCATE(TEMP);
    PT1 := ANTE(P);
    PT1+.LINK := TEMP;
    TEMP+.LINK := P;
    P := TEMP;
END (*INSERT*);

PROCEDURE MOV(VAR P, Q: PTR);
(* MOVES THE ELEMENT TO WHICH P POINTS, SUCH THAT IT IS THE ANTECEDENT
OF THE ELEMENT TO WHICH Q POINTS *)
(* IF P IS AN ENTIRE LIST THEN P:=NIL ELSE P:=P+.LINK *)
(* Q NOW POINTS TO WHAT WAS ORIGINALLY POINTED TO BY P *)
VAR
    PT1: PTR;
    PT2: PTR;
    TEMP: PTR;
BEGIN

64
IF \( P+.\text{LINK} = P \)
THEN
  \( \text{TEMP} := \text{NIL} \)
ELSE
  BEGIN
    \( \text{PT1} := \text{ANTE}(P); \)
    \( \text{PT1+.\text{LINK}} := P+.\text{LINK}; \)
    \( \text{TEMP} := P+.\text{LINK}; \)
  END;
  \( \text{PT2} := \text{ANTE}(Q); \)
  \( \text{PT2+.\text{LINK}} := P; \)
  \( P+.\text{LINK} := Q; \)
  \( Q := P; \)
  \( P := \text{TEMP}; \)
END (*MOV*);

PROCEDURE INCOR(VAR P: PTR; Q: PTR; VAR R: PTR);
(* MOVES THE STRING OF ELEMENTS, BEGINNING WITH THE ELEMENT TO WHICH P POINTS, AND ENDING WITH THE ELEMENT TO WHICH Q POINTS *)
(* INSERTS IT TO PRECEDE THE ELEMENT TO WHICH R POINTS *)
(* IF P THRU Q IS AN ENTIRE LIST, THEN P:=NIL ELSE P:=Q+.\text{LINK} *)
(* Q REMAINS UNCHANGED *)
(* R NOW POINTS TO WHAT WAS ORIGINALLY POINTED TO BY P *)

VAR
  PT1: PTR;
  PT2: PTR;
  TEMP: PTR;
BEGIN
  IF \( Q+.\text{LINK} = P \)
  THEN
    \( \text{TEMP} := \text{NIL} \)
  ELSE
    BEGIN
      \( \text{PT1} := \text{ANTE}(P); \)
      \( \text{PT1+.\text{LINK}} := Q+.\text{LINK}; \)
      \( \text{TEMP} := Q+.\text{LINK}; \)
    END;
    \( \text{PT2} := \text{ANTE}(R); \)
    \( \text{PT2+.\text{LINK}} := P; \)
END (*INCOR*);
Q+.LINK := R;
R := P;
P := TEMP;
END (*INCOR*);

FUNCTION ELEM(P: PTR; N: INTEGER): PTR;
(* POINTS TO THE N-TH ELEMENT AFTER THE ELEMENT TO WHICH
P POINTS *)
(* P REMAINS UNCHANGED *)

VAR
I: INTEGER;
BEGIN
FOR I := 1 TO N DO
  P := P+.LINK;
  ELEM := P;
END (*ELEM*);

PROCEDURE READLINE;

VAR
CH: CHAR;
NAME: ALFA;

PROCEDURE NEXTCH;

BEGIN
IF FIRSTREAD THEN
  FIRSTREAD := FALSE
ELSE
  GET(INPUT);
  CH := INPUT+;
END (*NEXTCH*);

PROCEDURE GETNONBLANK;

BEGIN
  NEXTCH;

  66
WHILE (CH = = =) AND (NOT EOLN(INPUT)) DO
  NEXTCH;
END (*GETNONBLANK*);

PROCEDURE GETALFA(VAR NAME: ALFA);

VAR
  I: INTEGER;

BEGIN
  FOR I := 1 TO 10 DO
    BEGIN
      NAME[I] := CH;
      NEXTCH;
    END;
END (*GETALFA*);

BEGIN (*READLINE*)
  P := HEAD;
  GETNONBLANK;
  WHILE NOT EOLN(INPUT) DO
    BEGIN
      INSERT(P);
      P^.CONTINUE := FALSE;
      GETALFA(NAME);
      P^.SYMBOL := NAME;
      READ(P^.FREQ);
      WritelN(" E, P^.SYMBOL, P^.FREQ);
      FIRSTREAD := TRUE;
      GETNONBLANK;
    END;
END (*READLINE*);

PROCEDURE SORT;

VAR
  VI: REAL;
  LAST, NEXT: PTR;

BEGIN
  LAST := HEAD^.LINK;
  NEXT := LAST^.LINK;

  WHILE (CH = = =) AND (NOT EOLN(INPUT)) DO
    NEXTCH;
END (*GETNONBLANK*);
WHILE NEXT <> HEAD DO
IF LAST+.FREQ < NEXT+.FREQ THEN
BEGIN
LAST := NEXT;
NEXT := LAST+.LINK;
END ELSE
BEGIN
P := HEAD;
V := NEXT+.FREQ;
REPEAT
P := P+.LINK;
UNTIL P+.FREQ ≥ V;
MOV(NEXT, P);
END;
END (*SORT*));

PROCEDURE COMBINE;
VAR
I: INTEGER;
F: REAL;
TEMP: PTR;
BEGIN
REPEAT
SORT;
P := HEAD;
F := 0;
FOR I := 0 TO 1 DO
BEGIN
P := P+.LINK;
P+.CODE := I;
F := F + P+.FREQ;
END;
INIT(Q);
TEMP := Q;
R := HEAD+.LINK;
INCOR(R, P, Q);
INSERT(R);
R+.CONTINUE := TRUE;
R+.MORE := TEMP;
R+.FREQ := F;
TEMP+.CONTINUE := TRUE;
END
TEMP*.MORE := R;
UNTIL HEAD*.
LINK*.LINK = HEAD;
END (*COMBINE*);

PROCEDURE ANSWER;
BEGIN
WHILE HEAD*.LINK <> HEAD DO
BEGIN
WRITE(£ £);
Q := HEAD*.LINK;
REPEAT
P := Q*.MORE;
Q := P*.LINK;
WRITE(Q*.CODE: 2);
UNTIL NOT Q*.
CONTINUE;
WRITE(£ £);
WRITE(Q*.SYMBOL);
WRITELN;
RELEASE(Q);
WHILE (P = P*.LINK) AND (P <> HEAD) DO
BEGIN
Q := P*.MORE;
RELEASE(P);
P := Q*.LINK;
RELEASE(Q);
END;
END (*ANSWER*);
BEGIN (*HUFFMAN*)
WRITELN;
WRITELN;
ORG;
FIRSTREAD := TRUE;
INIT(HEAD);
WHILE NOT EOF(INPUT) DO
  READLINE;
COMBINE;
ANSWER;
WRITELN;
WRITELN;

69
END (*HUFFMAN*).
1	2.00000000000000E-001
2	1.80000000000000E-001
3	1.00000000000000E-001
4	1.00000000000000E-001
5	1.00000000000000E-001
6	6.00000000000000E-002
7	6.00000000000000E-002
8	4.00000000000000E-002
9	4.00000000000000E-002
10	4.00000000000000E-002
11	4.00000000000000E-002
12	3.00000000000000E-002
13	1.00000000000000E-002

0 0: 1
0 1 0: 3
0 1 1: 4
1 0 0: 5
1 0 1 0: 6
1 0 1 1 0 0: 13
1 0 1 1 0 1: 12
1 0 1 1 1: 8
1 1 0: 2
1 1 1 0 0: 11
1 1 1 0 1: 10
1 1 1 1 0: 9
1 1 1 1 1: 7
Appendix IV: Program LISP
(*$U*IWi,56) MARY CAPECE *)
PROGRAM LISPCINPUT, OUTPUT);

TYPE
  SYMBOL = 
    (ATOMSYM, LPAREN, RPAREN, COMMA, DOL);
  KIND = 
    (ATOMIC, NONATOMIC);
  PTR = * NODE;
  NODE = RECORD
    CASE STATE: KIND OF
    ATOMIC: (NAME: ALFA;
     LINK: PTR);
    NONATOMIC: (HEAD, TAIL: PTR);
  END;

VAR
  L1, L2: PTR;
  NILL, TREW, ATOMLIST: PTR;
  SYM: SYMBOL;
  IDENT: ALFA;
  XX, YY, ZZ: PTR;

PROCEDURE ERROR(N: INTEGER);
BEGIN
  WRITELN;
  WRITE(E = )
  CASE N OF
    1:
      WRITE(E*** CAR OF ATOM IS UNDEFINED ***E);
    2:
      WRITE(E*** CDR OF ATOM IS UNDEFINED ***E);
    3:
      WRITE(E*** INPUT TO PROCEDURE WRITEATOM MUST BE ATOMIC ***E);
    4:
      WRITE(E*** EQ DEFINED ONLY ON TWO ATOMS ***E);
    5:
      WRITE(E*** ERROR IN INPUT ***E);
    6:
      WRITE(E*** ERROR IN LIST ***E);
  END;
  WRITELN;

73
HALT;
END (*ERROR*);

FUNCTION ATOM(L1: PTR): PTR;
BEGIN
  IF L1.STATE = ATOMIC THEN
    ATOM := TRUE
  ELSE
    ATOM := NIL;
  END (*ATOM*);

FUNCTION CONS(L1, L2: PTR): PTR;
VAR
  Q: PTR;
BEGIN
  NEW(Q);
  WITH Q DO
    BEGIN
      STATE := NONATOMIC;
      HEAD := L1;
      TAIL := L2;
    END;
  CONS := Q;
END (*CONS*);

FUNCTION CAR(L1: PTR): PTR;
BEGIN
  IF ATOM(L1) = TRUE THEN
    ERROR(1)
  ELSE
    CAR := L1.HEAD;
  END (*CAR*);

FUNCTION CDR(L1: PTR): PTR;
74
BEGIN
  IF ATOM(L1) = TREW
  THEN
      ERROR(2)
  ELSE
      CDR := L1.TAIL;
  END (*CDR*);

FUNCTION EQ (L1, L2: PTR): PTR;
BEGIN
  IF (ATOM(L1) = NILL) OR (ATOM(L2) = NILL)
  THEN
      ERROR(4)
  ELSE
      IF L1 = L2
      THEN
          EQ := TREW
      ELSE
          EQ := NIL;
  END (*EQ*);
END (*EQ*);

FUNCTION EQUAL(L1, L2: PTR): PTR;
BEGIN
  IF (ATOM(L1) = TREW) AND (ATOM(L2) = TREW)
  THEN
      EQUAL := EQ (L1, L2)
  ELSE
      IF EQUAL(CAR(L1), CAR(L2)) = TREW
      THEN
          EQUAL := EQUAL(CDR(L1), CDR(L2))
      ELSE
          EQUAL := NIL;
  END (*EQUAL*);
END (*EQUAL*);

PROCEDURE ENTER(WRD: ALFA; VAR S: PTR);
VAR
  Q: PTR;

75
BEGIN
ATOMLIST*.NAME := WRD;
Q := ATOMLIST*.LINK;
WHILE Q*.NAME <> WRD DO 
  Q := Q*.LINK;
IF Q = ATOMLIST THEN
  BEGIN
    NEW(ATOMLIST); 
    ATOMLIST*.STATE := ATOMIC; 
    ATOMLIST*.LINK := Q*.LINK; 
    Q*.LINK := ATOMLIST;
  END;
S := Q;
END (*ENTER*);

PROCEDURE GETSYM;
VAR
  I: INTEGER;
BEGIN
  WHILE INPUT* = E E DO
    GET(INPUT);
  IF INPUT* = E(= THEN
    BEGIN
      SYM := LPAREN;
      GETCINPUT;
    END
  ELSE
    IF INPUT* IN [EAE .. E9E] THEN
      BEGIN
        SYM := ATOMSYM;
        I := 0;
        REPEAT
          I := I + 1;
          IF I <= 10 THEN
            IDENT[I] := INPUT*;
            GET(INPUT);
          UNTIL NOT (INPUT* IN [EAE .. E9E]);
        FOR I := I + 1 TO 10 DO
          IDENT[I] := E E;
      END
    ELSE
      IF INPUT* = E,=
THEN
BEGIN
SYM := COMMA;
GET(INPUT);
END
ELSE
IF INPUT* = =)
THEN
BEGIN
SYM := RPAREN;
GET(INPUT);
END
ELSE
ELSE
IF INPUT* = $=
THEN
BEGIN
SYM := DOL;
GET(INPUT);
END
ELSE
ERROR(5);
END (*GETSYM*);

PROCEDURE GETLIST(VAR L: PTR);

VAR
PTSTACK: ARRAY [1..100] OF PTR;
OPSTACK: ARRAY [1..100] OF SYMBOL;
PTTOP, OPTOP: INTEGER;
BEGIN
GETSYM;
PTTOP := 1;
OPTOP := 1;
WHILE SYM <> DOL DO
BEGIN
IF SYM = ATOMSYM
THEN
BEGIN
PTTOP := PTTOP + 1;
ENTER(IDENT, PTSTACK[PTTOP]);
GETSYM;
END
ELSE
IF SYM = LPAREN
THEN
BEGIN
  OPTOP := OPTOP + 1;
  OPSTACK[OPTOP] := LPAREN;
  GETSYM;
END
ELSE
  IF SYM = COMMA
  THEN
    BEGIN
      OPTOP := OPTOP + 1;
      OPSTACK[OPTOP] := COMMA;
      GETSYM;
    END
  ELSE
    IF SYM = RPAREN
    THEN
      BEGIN
        PTTOP := PTTOP + 1;
        ENTER(ENIL, PTSTACK[PTTOP]);
        PTSTACK[PTTOP - 1] := CONS(PTSTACK[PTTOP - 1], PTSTACK[PTTOP]);
        PTTOP := PTTOP - 1;
        WHILE OPSTACK[OPTOP] = COMMA DO
          BEGIN
            PTSTACK[PTTOP - 1] := CONS(PTSTACK[PTTOP - 1], PTSTACK[PTTOP]);
            PTTOP := PTTOP - 1;
            OPTOP := OPTOP - 1;
          END;
        IF OPSTACK[OPTOP] <> LPAREN
        THEN
          ERROR(6)
        ELSE
          BEGIN
            OPTOP := OPTOP - 1;
            GETSYM
          END;
      END;
    L := PTSTACK[PTTOP];
END (*GETLIST*);

78
PROCEDURE PRINT(L1: PTR);

PROCEDURE WRITEATOM(L1: PTR);

VAR
    WRD: ALFA;
    I: INTEGER;
BEGIN
    IF ATOM(L1) = NILL
    THEN
        ERROR(3)
    ELSE
        BEGIN
            WRITE("=");
            WRD := L1^\$.NAME;
            I := 1;
            REPEAT
                WRITE(WRD[I]);
                I := I + 1;
            UNTIL (I > 10) OR (WRD[I] = "= ");
            WRITE("=");
        END;
    END (*WRITEATOM*);

PROCEDURE TRAVERSE(L1: PTR);

BEGIN
    IF L1 = NILL
    THEN
        WRITE("= ");
    ELSE
        WITH L1 DO
            BEGIN
                IF ATOM(HEAD) = NILL
                THEN
                    BEGIN
                        WRITE("= ");
                        TRAVERSE(HEAD);
                    END
                ELSE
                    WRITEATOM(HEAD);
                    TRAVERSE(TAIL);
            END;
    END;

79
BEGIN (*PRINT*)
  WRITELN;
  IF ATOM(L1) = TREW THEN
    WRITEATOM(L1);
  ELSE
    BEGIN
      WRITE('ε (ε);
      TRAVERSE(L1);
      END;
      WRITELN;
      WRITELN;
  END (*PRINT*);

PROCEDURE INITIALIZEATOMLIST;

BEGIN
  NEW(ATOMLIST);
  NEW(NILL);
  NEW(TREW);
  ATOMLIST*.STATE := ATOMIC;
  NILL*.STATE := ATOMIC;
  TREW*.STATE := ATOMIC;
  ATOMLIST*.LINK := NULL;
  NILL*.LINK := TREW;
  TREW*.LINK := ATOMLIST;
  NILL*.NAME := ENIL;
  TREW*.NAME := ET*;
END (*INITIALIZEATOMLIST*);

FUNCTION FLAT(L1, L2: PTR): PTR;

BEGIN
  IF L1 = NILL THEN
    FLAT := L2
  ELSE
    IF ATOM(L1) = TREW THEN
      FLAT := CONS(L1, L2)
ELSE
FLAT := FLAT(CAR(L1), FLAT(CDR(L1), L2));
END (*FLAT*);

FUNCTION REV(L1, L2: PTR): PTR;
BEGIN
IF L1 = NIL
THEN
REV := L2
ELSE
REV := REV(CDR(L1), CONS(CAR(L1), L2));
END (*REV*);

FUNCTION EVALUATE(L1: PTR): PTR;
VAR
M, N: PTR;
BEGIN
IF ATOM(L1) = TRUE
THEN
EVALUATE := L1
ELSE
BEGIN
M := CAR(CAR(CDR(L1)));
N := CAR(CAR(CDR(CDR(L1))));
EVALUATE := CONS(EVALUATE(M), CONS(CAR(L1), CONS(EVALUATE(N), NILL)));
END;
END (*EVALUATE*);

BEGIN (*LISP*)
INITIALIZEATOMLIST;
WRITELN;
GETLIST(XX);
WRITE(XXE);
PRINT(XX);
GETLIST(YY);
WRITE(YYE);
PRINT(YY);
WRITE(CAR(YYE));
PRINT(CAR(YY));
WRITE(COR(YY));
PRINT(COR(YY));
WRITE(ATOM(xx));
PRINT(ATOM(xx));
WRITE(ATOM(yy));
PRINT(ATOM(yy));
WRITE(EQ(NILL,NILL));
PRINT(EQ(NILL,NILL));
WRITE(EQUAL(xx,yy));
PRINT(EQUAL(xx,yy));
WRITE(CONS(xx,yy));
PRINT(CONS(xx,yy));
WRITE(CONS(yy,xx));
PRINT(CONS(yy,xx));
WRITE(FLAT(yy,NILL));
PRINT(FLAT(yy,NILL));
WRITE(REV(yy,NILL));
PRINT(REV(yy,NILL));
GETLIST(ZZ);
WRITE(ZZ);
PRINT(ZZ);
WRITE(EVALUATE(ZZ));
PRINT(EVALUATE(ZZ));
END (*LISP*).
XX
((A)B)

YY
((E)(F(GH))I)

CAR(YY)
(E)

CDR(YY)
((F(GH))I)

ATOM(XX)
NIL

ATOM(YY)
NIL

EQ(NILL,NILL)
*T*

EQUAL(XX,YY)
NIL

CONS(XX,YY)
(((A)B)(E)(F(GH))I)

CONS(YY,XX)
(((E)(F(GH))I)(A)B)

FLAT(YY,NILL)
(EF GH I)

REV(YY,NILL)
(I(F(GH))(E))

ZZ
(DIV((PLUS(A(MINUS(BG)))))
(TIMES((DIV(D E)F))))

EVALUATE(ZZ)
((A PLUS(B MINUS C)) DIV
((D DIV E) TIMES F))
XX
(( A ) B )

YY
(( E ) ( F ( G H )) I )

CAR(YY)
( E )

CDR(YY)
(( F ( G H )) I )

ATOM(XX)
NIL

ATOM(YY)
NIL

EQ(NILL, NILL)
*T*

EQUAL(XX, YY)
NIL

CONS(XX, YY)
((( A ) B ) ( E ) ( F ( G H )) I )

CONS(YY, XX)
((( E ) ( F ( G H )) I ) ( A ) B )

FLAT(YY, NILL)
( E F G H I )

REV(YY, NILL)
( I ( F ( G H )) ( E ) )

ZZ
( DIV ( ( PLUS ( A ( MINUS ( B G ))) )
( TIMES ( ( DIV ( D E ) F ))) )

EVALUATE(ZZ)
(( A PLUS ( B MINUS C )) DIV
( ( D DIV E ) TIMES F ))

83
VITA

Mary J. Capece, daughter of Mr. and Mrs. David S. Capece, was born in Philadelphia, Pennsylvania on February 10, 1954. She attended Chestnut Hill College in Philadelphia and graduated Magna Cum Laude, receiving the degree of Bachelor of Science in Mathematics. Ms. Capece is a member of Delta Epsilon Sigma National Scholastic Honor Society. In September 1975, she began working toward the degree of Master of Science in Computer Science at Lehigh University in Bethlehem, Pennsylvania. During that time, Ms. Capece held a teaching assistantship in the Department of Mathematics at Lehigh University.
May 6, 1977

Dean Robert D. Stout
Graduate School
Whitaker Laboratory

Dear Dean Stout:

The computer programs included in the master's thesis of Mary Capece do not have any commercial value nor are there any copyright problems. The programs have purely educational content.

Yours truly,

Samuel L. Gulden
Professor of Mathematics

SL:gb