AN IMPLEMENTATION OF A DATA STRUCTURES DISPLAY SYSTEM

By

WILSON LEE

Bachelor of Science in Arts and Sciences

Oklahoma State University

Stillwater, Oklahoma

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1988

AN IMPLEMENTATION OF A

DATA STRUCTURES

DISPLAY SYSTEM

Thesis Approved:

Donald D Fisher
Thesis Adviser

J Chandler

Norman N. Durham
Dean of the Graduate College

PREFACE

The Data Structure Display System will be a helpful tool for students in an introductory level course in data structures. The system of programs is made up of many functions which I hope will be reusable code.

I wish to express sincere appreciation to all the people who assisted me in this study. I am especially grateful to my major adviser, Dr. D. D. Fisher, for his encouragement and guidance throughout my stay here at Oklahoma State University.

Appreciation is also extended to Dr. J. P. Chandler and Dr. K. M. George for serving on my graduate committee and providing suggestions and support.

I also wish to thank Dr. G. E. Hedrick for his guidance and for agreeing to be on the committee for my thesis defense.

Many thanks go to my parents, Soon Aik and Chin Sen San Lee, and my wife, Bee Geok, for their patience and never-ending support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Anyone who has written or studied computer programs has encountered data structures. Programs can be thought of as operations performed on some data structures. For example, a database program operates on the index structure to extract information. In a simple "Hello World!" program the operation is on the string of characters.

> A data structure is a structure whose elements are items of data, and whose organization is determined both by the relationships between the data items and by the access functions that are used to store and retrieve them [Baron, 1980].

Most operations on simple basic data structures can be visualized and understood easily by most students. Dynamic data structures may be built from these basic data structures. Operations on these dynamic data structures are not always visualized easily.

> Programming is a constructive activity. How can a constructive, inventive ability be taught? One method is to crystallize elementary composition principles out of many cases and exhibit them in a systematic manner. What remains in our arsenal of teaching methods is the careful selection and presentation of master examples [Wirth, 1986].

The study of data structures would be easier if the

student were to be able to view the graphical representation of the structure and test various operations associated with this structure. The keen learner would benefit even more if his algorithms could be tested easily.

The primary objective of this thesis is to design a system which :

1. graphically displays a variety of data structures,
2. allows the user to execute and study the immediate effects of each step of an operation on a particular data structure.

To be able to handle a variety of data structures, there must be a method of defining the characteristics of data structures so that the actual structure information can be bound at system runtime. This study introduces a method of defining the characteristics of simple dynamic data structures and allowing the system to function according to this definition.

This report first provides some background information on the areas related to the design of the data structure display system. After an overview of the display system is given, the syntax and semantics of the chosen algorithm specification language and the intermediate representation language are discussed. The actual implementation design of various parts of the data structure display system is discussed in Chapter V.

CHAPTER II

LITERATURE REVIEW

This study involves the design and implementation of a data structure display system. The areas related to the design considerations of this display system can be grouped into the following four topics: data structures, source language and translation, intermediate representation language, and user interface.

Data Structures

In the book, Data Structures, Reingold and Hansen [1983] define a data structure to have three components:

1. a set of function definitions: each function is an operation available to the rest of the program,
2. a storage structure specifies classes of values, collections of variables, and relations between variables as necessary to implement the functions,
3. a set of algorithms, one for each function in which each algorithm examines and modifies the storage structure to achieve the result defined for the corresponding function.

Using this definition as the basic concept, the data structure to be represented in the display system has a

definition file to define the storage structure. This data structure has its associated set of operations or functions. However, the third component can be extended to have a sequence of one or more algorithms achieve the result defined for the corresponding function. An example is to define a balanced tree insert operation to consist of an insert algorithm followed by a balancing algorithm.

There are at least two ways for implementing data structures: sequentially storing elements in contiguous memory locations and linking elements based on some relationship.

> Blocks are contiguous units of memory that are processed as structural entities. Linked structures are structures comprised of several blocks having some logical interconnection. Blocks may be divided into fields [Baron, 1980].

The data structures in this display system have the organization of linked structures. The prototype system does not support contiguous memory implementation data structures. Some examples of linked data structures are linked lists and trees.

> A tree is a collection of elements called nodes, one of which is distinguished as a root, along with a relation ("parenthood") that places a hierarchical structure on the nodes [Aho, 1985].

The general type of linked data structures supported in this display system has a single defined root and has well-defined links.

Source Language And Translation

Tremblay and Sorenson [1985] listed several goals to be important when designing a programming language. Some of these goals are human communication, the prevention and detection of errors, usability, program effectiveness, compilability, efficiency, machine independence, and simplicity.

Since the data structure display system is a learning tool, the language used to specify the algorithm should not be difficult. With this in mind, the three goals that the specification language must meet are simplicity, readability, and compilability. The syntax of each specification language instruction should reflect the semantics, or in other words be self-documenting. There should not be any side-effects to any instruction. There are parser generators available that produce a translator for this specification language. The idea of compilability would then be having the grammar of the specification language be of the type required by the parser generator used.

A grammar is a 4-tuple, $G = (N, \Sigma, P, S)$ where
1. N is a finite set of nonterminal symbols
2. Σ is a finite set of terminal symbols, disjoint from N
3. P is finite subset of $(N \cup \Sigma)^* N (N \cup \Sigma)^* X (N \cup \Sigma)^*$
   An element (α, β) in P will be written $\alpha \longrightarrow \beta$ and called a production
4. S is a distinguished symbol in N called the sentence (or start) symbol [Aho, 1972].

There are four general classes of grammars: unrestricted, context-sensitive, context-free, and right-linear. The last three classes, context-sensitive, context-free, and right-linear, are phrase-structured grammars.

For the sake of compilability, the specification language should be generated by a phrase-structured language, specifically, a context-free grammar.

> A grammar G is said to be context-free if each production in P is of the form $A \longrightarrow \beta$, where A is in N and β is in $(N \cup \Sigma)^*$ [Aho, 1972].

The specification language is discussed in Chapter IV.

Intermediate Representation Language

In the book, Compilers: Principles, Techniques, and Tools, Aho, Sethi, and Ullman [1985] listed several intermediate representation languages. They are "syntax trees", "postfix notation", and "three-address code" which can be implemented by quadruples, triples, and indirect triples. The representation suitable for this system is the three-address format.

The quadruples implementation has the following format.

<operator>, <operand1>, <operand2>, <result>

where <operand1> and <operand2> denote the first and second operands, respectively, and <result> specifies the result of the operation. The result is usually a temporary variable.

The quadruples implementation does not impede program optimization. One disadvantage of using quadruples is that the allocation of temporary names must be managed.

The triples notation has the following format.

<n> <operator>, <operand1>, <operand2>

The triples implementation has an advantage over quadruples because it does not require entering temporary names into the symbol table to handle compound arithmetic operations. Triples handle temporary results by pointing at the triple statement producing a particular intermediate result. Each triple has a number <n> and the result of a previous triple is specified by its number in parentheses. For example, an operand (5) means that the result of triple number five is used. A disadvantage of using triples notation is that performing code optimization can be difficult because all references to a triple must be updated when this triple is moved.

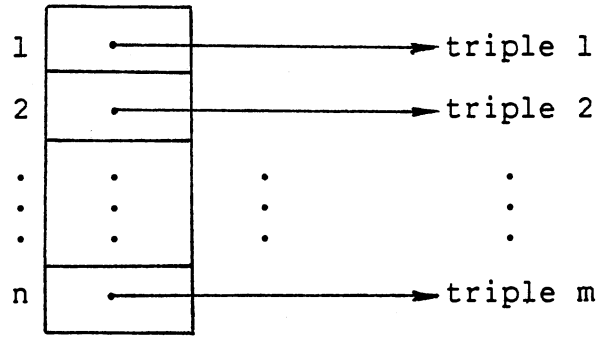


Figure 1. Indirect Triples Table

The indirect triples implementation uses a separate table which contains pointers to the triples. When code is moved during optimization, only the order of entries in the table is changed.

User Interface

The design of any system must always take into consideration the human factor. By considering how humans work better and what details in the system may make it difficult to learn and use, we can design a user-friendly system.

Rubinstein [1984] discussed several factors considered to be helpful in designing a user-friendly system. Some of these factors are listed below:

1. minimize conceptual load,
2. make states visible,
3. respond with an appropriate amount of information,
4. coordinate all system responses,
5. avoid multiple style modes,
6. acknowledge user actions quickly,
7. provide an easy way out,
8. allow people to work in real time,
9. announce long delays, and
10. avoid cluttering the display or overwhelming the user with visual attributes.

The general goals behind these factors are to promote ease of learning, ease of use, reliability, and

productivity.

The design of the data structure display system, to be an effective learning tool, has to abide by these and other rules. The main user control of the display system is through a series of menus. Two advantages of using menus are listed below:

1. explicit options are given, eliminating the possibility user typing mistakes, and
2. displayed options also serve as memory aid.

In the display system, the menus are kept short with a limited number of options in each menu so that the user can learn and remember how to use the data structure display system easily. The default option is always chosen to provide the user confidence. For example, most default options are either information requests, or exit to previous menu requests. All menus are also labeled so that the user can easily identify what state he is in.

Another interactive style of control in the user interface of the display system is direct manipulation. This style involves entering values for specific fields or questions. This style of interface is used only when a menu is not feasible.

CHAPTER III

SYSTEM OVERVIEW

The data structure display system allows the user to view and study an abstract data structure and its associated operations.

The prototype display system is implemented for VT100 type terminals. The reason for choosing VT100 terminals is that these terminals are available at many locations all over campus. These terminals also are the basis for Standard X3.64. However, the display system is designed so that it can be modified to handle other terminal types.

Currently, the display system can handle structures having up to four successors. This is a limitation due to the hardware problem of displaying graphics as characters rather than as a combination of pixels.

The data structure display system can be executed using one-screen or two-screen modes. The one-screen mode displays on one screen the graphical representation of part of the structure, the algorithm statement executed, and the values of the variables in that statement. In the two-screen mode, the second screen presents a larger part of the abstract data structure.

The figure below shows the basic flow of control in the display system.

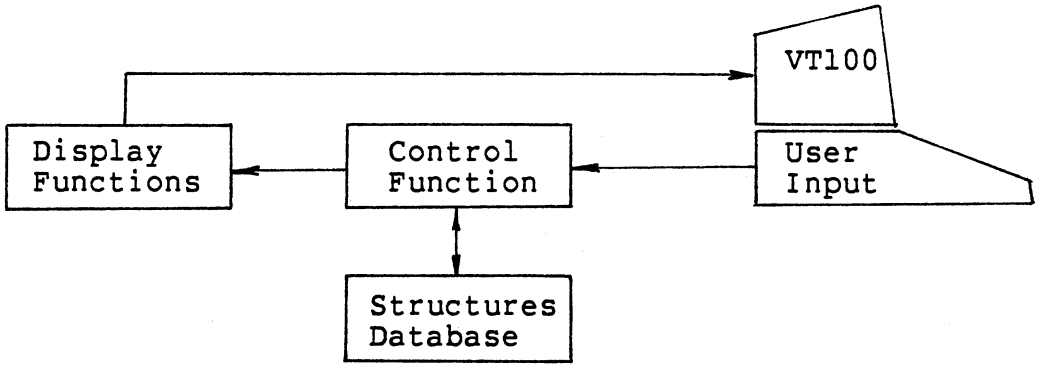


Figure 2. Overview of data structure display system.

The user is allowed the following operations:

1. select a data structure and any associated operation,
2. define new data structures and operations,
3. view the structure and algorithm of associated operations, and
4. step through an algorithm.

The control routines perform the following tasks:

1. handle user responses,
2. translate algorithm text into pseudo-code and an associated symbol table,
3. interpret the operation pseudo-code instructions,

4. access the structures database, and
5. supply the graphics driver with information.

The structure database contains the following parts:

1. algorithm text,
2. pseudo-code representation,
3. associated symbol table, and
4. specific information on structures.

During execution of the display system there are various menus and display screens. Screen dumps of these screens are in Appendix A. During execution of an algorithm the screen is divided into three areas: structure, values, and algorithm (see figure below).

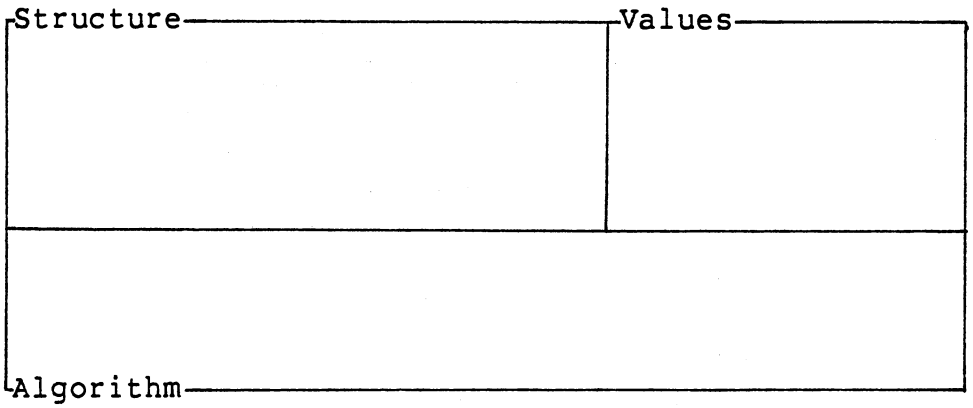


Figure 3. Algorithm Execution Screen Format.

The partial structure is displayed in the structure screen region. A maximum of four levels can be displayed at

one time. However, the user can browse up and down the whole structure. The graphics driver handles the portion of the structure to be displayed. The current node of the structure is indicated in the structure display region by an asterisk marker.

A maximum of nine lines of the algorithm is displayed in the algorithm region, with the appropriate line highlighted. The algorithm display driver handles the portion of the text to be displayed.

The values of variables in the algorithm can be seen in the values region. Other messages also are displayed in this area.

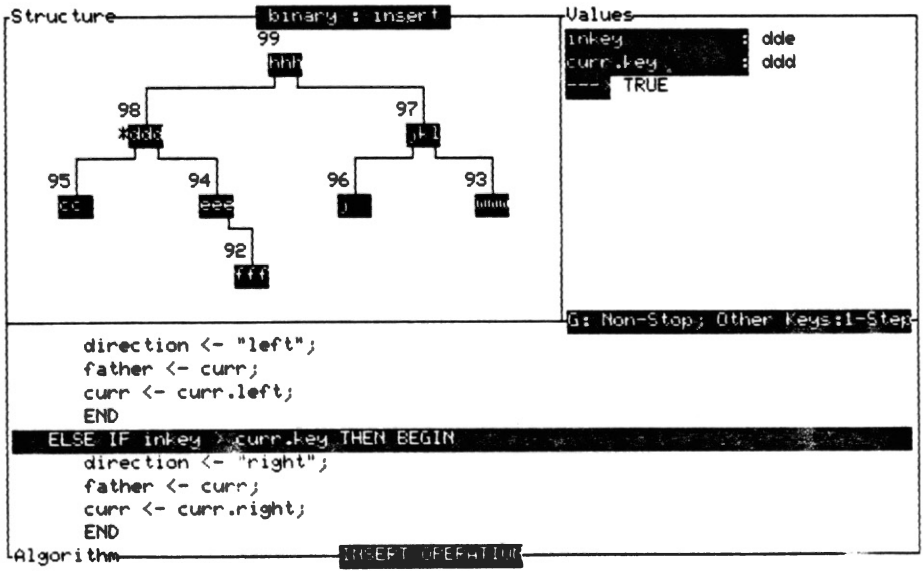


Figure 4. Sample Execution Screen.

In the figure above, the display system currently is performing the IF-statement. The values of the two variables, inkey and curr.key, considered in the boolean expression are displayed in the values region. The node that holds the curr.key is node number 98 and is indicated by an asterisk to its left. The user is given the choice of stepping through the algorithm one step at a time or to go non-stop.

Appendix A contains a sample run of the data structure display system. Details on the implementation design are discussed in Chapter V of this study. The next chapter discusses the syntax and semantics of the languages chosen for the Algorithm Specification Language (ASL) and the intermediate representation language.

CHAPTER IV

SYNTAX AND SEMANTICS

Algorithm Specification Language

The Algorithm Specification Language (ASL) for this system is designed with simplicity and readability in mind. It is meant to provide a minimal set of instructions sufficient for implementing simple non-recursive operations on dynamic data structures.

The keywords of ASL are listed in the figure below.

ALGORITHM	BEGIN	BOOLEAN	DECLARE
DEQUEUE	DO	ELSE	END
ENQUEUE	FALSE	FREENODE	HEAD
IF	IN	INPUT	INTEGER
NEWNODE	NIL	NODE	NOT
OUT	OUTPUT	POP	PUSH
STRING	THEN	TRUE	WHILE

Figure 5. ASL Keywords.

Although the list shown above uses all uppercase characters, it must be noted that the algorithm specification language is case insensitive. However, the user is encouraged to have keywords in uppercase characters to improve readability.

A valid ASL identifier is a string of letters or numbers which must begin with a letter. ASL keywords by itself cannot be used for an identifier name, although it may be embedded within other legal characters to create a valid identifier name.

The symbols used in ASL are listed in the figure below.

.	,	:	;
=	!=	<	>
(	)	<-	

Figure 6. ASL Reserved Symbols.

The basic structure of the ASL program is shown in the figure 7.

```
ALGORITHM
   <argument_sequence>
DECLARE
   <declaration_sequence>
BEGIN
   <statement_sequence>
END.
```

Figure 7. ASL Program Structure.

The first word in an ASL program is the keyword "ALGORITHM". The <argument_sequence> provides the program a means of receiving and returning values. The <declaration_sequence> follows the DECLARE keyword. Here, variables can be declared to be of any of the valid data types. The current system support only the data types HEAD, NODE, STRING, BOOLEAN, and INTEGER. The NODE data type is made up of several fields. These fields are defined in the structure definition file. The HEAD data type is similar to the NODE data type except that it indicates a special purpose as the header node.

ASL statements may be any of the nine types: assignment, if-then-else, loop, push, pop, enqueue, dequeue, input, and output. The logical end of the ASL program is indicated by the "END" keyword with a period immediately following it.

The detailed ASL grammar can be found in Appendix G. The context-free ASL grammar is designed to match the requirements of the language development tool, Yacc, a LALR(1) parser generator. The section on language translation in Chapter V discusses further the usage of this tool and other aspects of the translation process.

Intermediate Representation Language

The intermediate representation language for the data structure display system is a modified version of the quadruples format [Aho, 1985; Tremblay, 1985]. The problem of temporary variables is not faced in this system because there are no compound arithmetic operations.

The pseudo-code operation keywords are listed in the figure below.

LNO	MOV	TST	AND
OR	NOT	JMP	NEW
FRE	NOP	END	INP
OUT	PSH	POP	DEQ
ARG	RET		

Figure 8. Pseudo-code Operation Keywords.

These pseudo-code operations are sufficient to

implement all the statements available in the algorithm specification language. A special pseudo-code operation is "LNO" which is used to coordinate between the pseudo-code execution and the algorithm text display. The syntax for the pseudo-code operations is listed in the figure below.

```
LNO linenumber
MOV destination,source
TST operand1,operand2,condition
AND
OR
NOT
JMP location,condition
NEW operand
FRE operand
NOP
END
INP operand
OUT operand
PSH operand
POP operand
DEQ operand
ARG operand
RET operand
```

Figure 9. Syntax of Pseudo-Code Operations.

Most of the operations are self-explanatory. The "TST" operation compares operand1 and operand2 for the condition specified and pushes the boolean result on the interpreter stack. "AND" and "OR" operations pop two values off the stack and pushes the result back on the stack. The "JMP" operation jumps to the location specified if the value pop from the stack matches the condition specified.

CHAPTER V

IMPLEMENTATION DESIGN

The data structure display system is written using the 'C' programming language in a Unix environment. The 'C' programming language is chosen as the development language because it is available on all computer systems available in the department. Since 'C' is relatively portable, the data structure display system may be ported to another host system when necessary.

Although there are software development utilities in Unix, the "curses" package for example, that supports screen formats such as windows, the display system uses only specially written routines. There are several reasons for using these specially written routines. The first reason is that of speed and ease of control. Direct screen writing is faster than using the window environment in the "curses" package because there is no need for the window refresh operation. Another reason is that the data structure display system involves placing graphics characters on the screen. On VT100-type terminals, character graphics and special screen attributes are invoked by sending appropriate control sequences to the terminal.

However, for language translation operation, two Unix

software development tools, Lex and Yacc, are used to implement the translator for the Algorithm Specification Language. Lex and Yacc are generators for lexical analyzers and syntactic analyzers respectively.

The data structure display system can be divided into four main parts: Main Driver, Interpreter, Translator, and Graphics Driver. The hierarchy diagram of the display system is shown in the figure below.

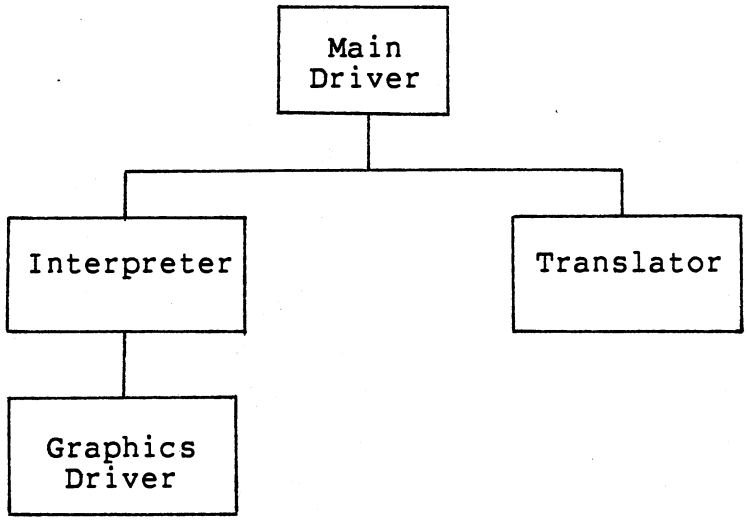


Figure 10. Hierarchy of Data Structure Display System Components.

In general, the main driver provides the user interface and calls the interpreter or translator at appropriate times. The interpreter, in turn, calls the graphics driver

to display the desired structure. These four components are discussed further in later sections.

Before discussing the design of the four main parts of the data structure display system in detail, an important overall design feature must be presented. This feature is the storage structure used to accommodate the various possible node formats.

General Storage Structure

The data structure display system requires a generic storage structure to be implemented in primary memory. Different structures may define nodes having different numbers of keys, pointers, and attributes, or different combinations of fields. This display system is bound to a node format only at runtime.

Contiguous bytes of primary memory are used as implementation storage for each node in a structure. A node format definition template is obtained from the structure definition file. This definition file, created when defining a new structure in the data structure display system, contains information on the size, offset, and name of fields in a node (see Figure 11).

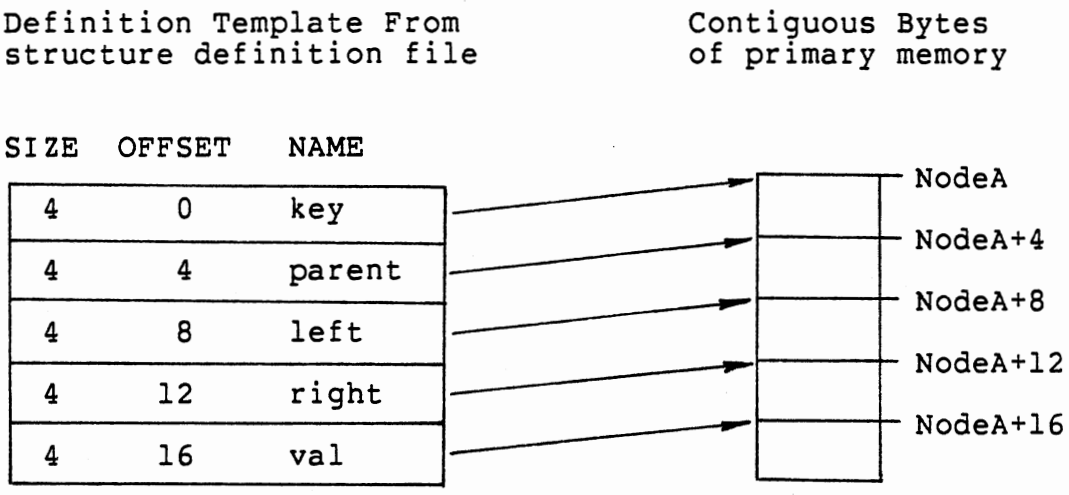


Figure 11. Node Definition Template.

This definition template is used extensively by the pseudo-code interpreter to handle all accesses to nodes of a structure. It is also used by the translator to set up the symbol table to reflect entries for a node declaration.

The structure definition file includes other information like the number of keys, parent pointers, child pointers, and attribute fields. It also states the location (field number) in the header node having the root pointer to the structure. Appendix B contains more information on this file.

Main Driver

The main driver functions as the controller for the data structure display system. To prevent uncontrolled interruption, the main driver places the terminal in raw

input/output processing mode. All input keys are handled solely by the data structure display system without the operating system checking for special control characters. Menus are used extensively to meet some of the criteria for a user-friendly system discussed in Chapter II. Menus allow the user to specify the desired action easily. To make this system easy to learn and use, there are only two menu formats and both formats use the same method of control. The first menu format (see Figure 12) is used for the main menu, abstract structure menu, and operation menu; the second format (see Figure 14) is used for the execution menu.

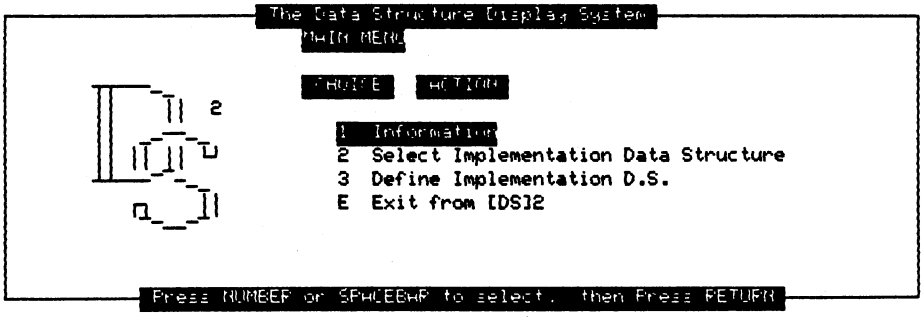


Figure 12. Menu Format One.

The first menu format uses the top half of the screen. Options are selected using the numbers, arrow keys, spacebar, or backspace, and then pressing the return key.

This method of selecting is common to all menus. When a selection other than "Exit" is made, the bottom half of the screen is used. Figure 13 shows an example of selecting an implementation structure.

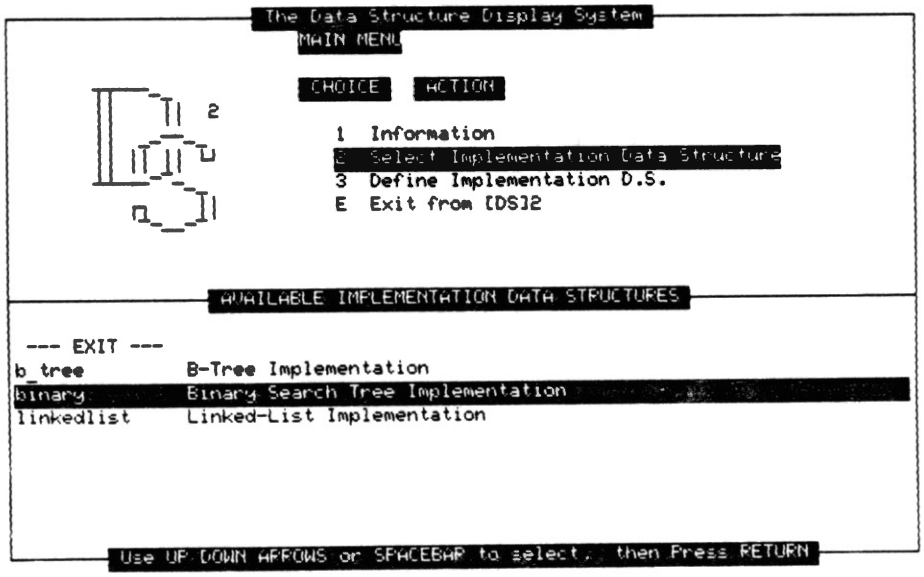


Figure 13. Selecting an Implementation Structure.

The second menu format is shown in Figure 14 below. The second menu format uses the top right corner of the screen. The method of selection is the same as in the first format. Both of these formats are driven by a screen function (screen.c) and a menu driver (menu.c).

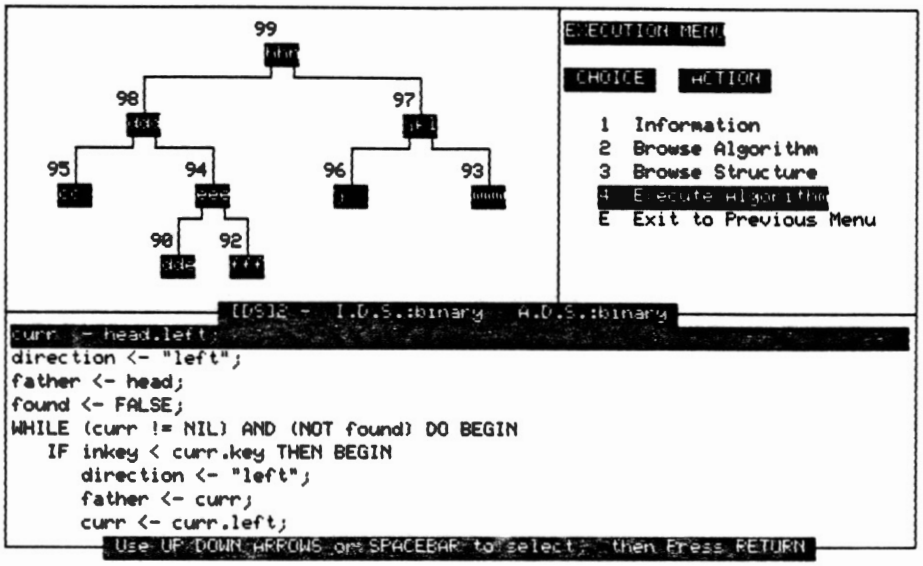


Figure 14. Menu Format Two.

Using these menus, the data structure display system moves to the correct level in the system to perform an operation. The directory hierarchy layout of the display system is shown in the figure 15.

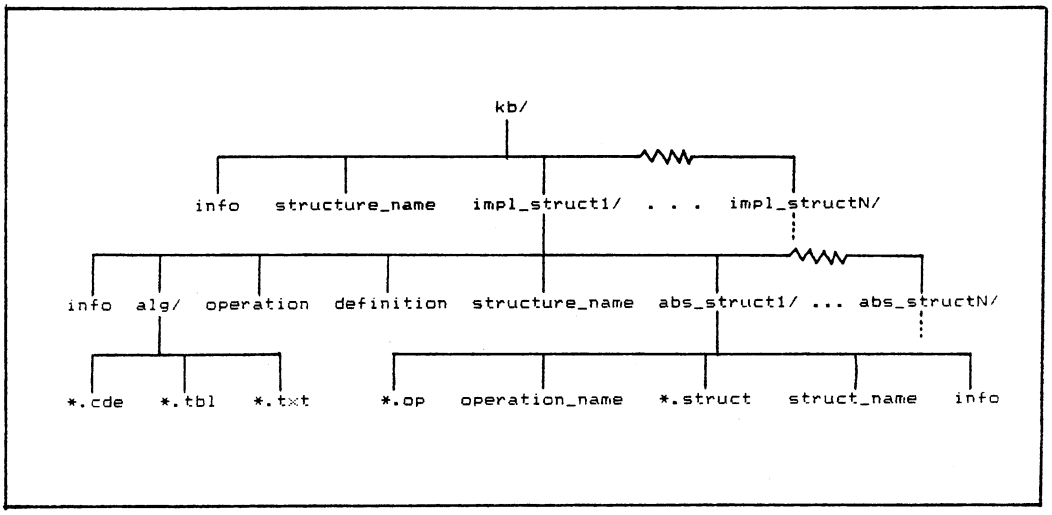


Figure 15. Hierarchy of System Database.

The information stored by the data structure display system is organized into a hierarchy of directories. In the "kb/" directory there is a file, "structure_name", containing the names of all implementation structures defined. Each name item consists of a word (10 characters maximum) and a short description. The main driver uses the selected name to move to the directory associated with the selected implementation structure. The interpreter works at the implementation directory level. In the implementation directory level, there is a structure definition file, "definition". This definition file contains the node description template discussed in the earlier section on storage structure. This directory level also has a file, "structure_name", which has the name entries for all defined abstract data structures. There is a subdirectory for

algorithm, "alg/", and subdirectories for abstract structures. A subdirectory exists for each abstract structure containing the definition of operations associated with this abstract structure. Created structures are also saved in this subdirectory.

Each operation definition is a sequence of various algorithms and system utilities needed to perform the required operation. For example, the insertion operation for a HB[1] Tree consists of the following sequence:

1. use system utility to ask and extract the correct tree,
2. perform general insertion algorithm,
3. perform balancing algorithm, and
4. use system utility to return tree to database.

The communication between one algorithm and another is through a communication file, "comm". The algorithm subdirectory, "alg/", contains all algorithm text, pseudo-code, and symbol table files for all operations for an implementation structure.

Language Translation

The data structure display system allows users to enter new algorithms for a defined implementation data structure. These algorithm texts must be translated before the interpreter can work on an algorithm. The translator for this system is developed using the software tools, Lex [Lesk, 1975] and Yacc [Johnson, 1975]. In brief, Lex

generates a lexical analysis function and Yacc generates a syntax analysis function.

The lexical analyzer recognizes words in the Algorithm Specification Language program and returns the appropriate tokens to the syntax analyzer. The syntax analyzer is a pushdown automaton. Based on the grammar rules, the syntax analyzer performs one of the four actions: shift, reduce, accept, or reject.

The input to the translator is the algorithm text written in the Algorithm Specification Language (ASL). The context-free ASL grammar is listed in Appendix G. The translator generates an intermediate representation (pseudo-code) program, and a symbol table. To be able to work for various types of node structures, the translator obtains information from the node definition template. This information allows the translator to install correct symbol table entries for fields of a node when a node variable is declared. Appendices D and E contain examples of a pseudo-code program and a symbol table respectively.

Pseudo-Code Interpreter

The intermediate language (pseudo-code) interpreter is a major component of the data structure display system. It performs the algorithm associated with an abstract data structure by interpreting the pseudo-code instructions.

The interpreter first reads in the node definition template. This template determines the correct memory

location for each field in the storage structure employed for the data structure display system (see earlier section on storage structure). Then the interpreter reads the algorithm text, the pseudo-code program, and the associated symbol table.

The actions of the interpreter is shown to the user through three main display functions. One function (algorithm.c) displays the algorithm text and highlights the currently executed statement. Another (display_value.c) displays the values of variables in that algorithm statement. The third function (draw.c) is part of the graphics driver that displays the graphical representation of the structure being studied. This graphics driver is discussed in the next section.

The first task of the interpreter is to handle the algorithm's input arguments, if any. The interpreter attempts to get the values for these arguments from the communication file, "comm". If it fails then the user is prompted for the argument's values. Once all input arguments are assigned values, the interpreter proceeds with the rest of the instructions. At the end of the pseudo-code program, return results are written to the communication file, "comm". The values in the communication file, "comm", are used by the next algorithm.

Since the data structure display system must handle various structures, the interface between the interpreter and the graphics driver must use a general format. This

general format is the sequential representation of the partial structure to be displayed. This sequential representation is discussed further in the next section on the graphics driver.

Graphics Driver

There are three main functions that make up the graphics driver. The first is the structure drawing function (draw.c). This function draws the structure using the information stored in the sequentially represented structure string. The sequential representation format contains the nodes to be displayed arranged sequentially in order of positions on the screen.

The second function (struct_mark.c) places an asterisk to the left of the current node being accessed. This function also uses the sequential representation of the structure. If the node to be marked is not in the string, that is the node is not displayed, then the marking function creates a new sequential representation string containing the desired node and invokes the drawing function.

The third function is the conversion routine that extracts information from the partial structure to be displayed to create the sequential representation string. All the graphics driver functions have built-in checks to avoid redrawing unchanged graphics on the screen.

The graphics driver draws to the screen by using macros defined in a VT100 screen control header file (screen.h).

These macros are print statements that send screen control sequences to the terminal to set a desired mode or to position the cursor. Screen control sequences and information to be displayed are directed through the standard error file stream, "stderr". The standard error stream is different from the standard output stream, "stdout", in that it is not buffered. Although the display system places the terminal in raw input/output processing mode, which means that "stdout" stream is also unbuffered, the standard error stream is still used because the data structure display system may be ported to a computer system that does not fully support the raw input/output mode.

Two-screen Mode

The data structure display system can operate in either one-screen or two-screen modes. In the two-screen mode, a larger partial structure is displayed on the second screen.

The main driver of the data structure display system is the master process. After displaying the title screen, the main driver prompts the user to select either one-screen or two-screen mode. If the two-screen mode is selected, the main driver provides additional instructions on how to use a second terminal. The user is instructed to log on at another terminal and type "slave" at the Unix prompt. The program "slave" is a slave process. This slave process checks for its own process identifier (pid), terminal device used, and terminal type. These information are written to a

communication file, "comminfo" and the user is prompted to press the return key at the main terminal.

The master process reads the information in the communication file. If this file is absent, then the display system assumes one-screen mode. Using the slave pid, the master process signals the slave process using the "kill" function call. The "SIGUSR2" signal places the slave process into a suspended state. The slave process stays in this suspended state for the duration of the session. Before the main process terminates, it signals the slave process again. The "SIGUSR1" signal brings the slave process out of the suspended state to perform cleaning-up operation. The slave process removes the communication file and exits.

CHAPTER VI

SUMMARY, CONCLUSIONS, AND FUTURE WORK

There are numerous operations on dynamic data structures that are difficult to visualize. The data structure display system provides the user with a means to study an operation by graphically displaying the structure while stepping through an algorithm applying to the structure. A user can define new structures and operations, or use existing ones. The current prototype display system handles structures having up to four successor pointers. During the execution of an algorithm, the screen is divided into three display areas: structure, values, and algorithm.

The data structure display system's user interface is mostly menu-driven to help make it a user-friendly system. The users' manual in Appendix H is written to guide the user in creating new structures and using existing structures and operations on the display system. It can be used as a standalone manual.

One major strong point of the design of the data structure display system is that most parts of the system are generalized functions. The data structure display system uses a generic storage structure implemented in primary memory. Each node consists of contiguous bytes of primary

memory. By using node structure templates, this display system is able to handle different node structures. In this way, the display system is bound to a node format only at runtime.

The data structures display system is limited to managing linked data structures with one header node and having up to a maximum of four successor nodes. This display system cannot handle data structures with multiple header nodes, such as the orthogonal linked list.

Some areas where future work can be done are the graphics driver and the algorithm specification language. Firstly, the graphics capability could be extended to handle more varied structures. It would also be nice to handle many different types of terminals using the information found in files such as the Unix "termcap" and "terminfo" terminal description files. The data structure display system could be modified to look up terminal description files for codes to handle the graphics display.

The algorithm specification language used in this thesis provides a minimal set of instructions. This language can be extended to provide other instructions and to support other data types. An immediate choice would be adding the array data type. Another possible way, but more drastic in its changes, is to use an ML-like language. Spooner [1986] discussed and gave examples of applying the language ML. He stated that this approach would allow the user to define both the syntax and the semantics of the

source language.

Another possible area to work on is to implement a knowledge-based editor (KBE) for the algorithm specification language. A KBE helps the user in writing algorithms for the system by reminding him of the syntax the system expects and in some cases the semantics as well. The KBE also reminds the user of the names given to the fields of the defined node structure.

A SELECTED BIBLIOGRAPHY

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. Data Structures and Algorithms. Reading, Massachusetts : Addison-Wesley, 1985.

Aho, A. V. and Ullman, J. D. The Theory of Parsing, Translation, and Compiling, Vol. I : Parsing. Englewood Cliffs, New Jersey : Prentice-Hall, 1972.

Aho, A. V., Sethi, R., and Ullman, J. D. Compilers : Principles, Techniques, and Tools. Reading, Massachusetts : Addison-Wesley, 1985.

Baron, R. J. and Shapiro, L. G. Data Structures and their Implementations. New York, New York : Van Nostrand Reinhold, 1980.

Barrett, W. A., Bates, R. M., Gustafson, D. A., and Couch, J. D. Compiler Construction : Theory and Practice. Chicago, Illinois : Science Research Associates, 1986.

Digital Equipment Corporation Rainbow Owner's Manual. Bedford, Massachusetts : Digital Equipment Corporation, 1983.

Johnson, S. C. "Yacc - Yet Another Compiler-Compiler." Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975).

Lesk, M. E. "Lex - A Lexical Analyzer Generator." Comp. Sci. Tech. Rep. No. 39, Bell Laboratories (October 1975).

Reingold, E. M. and Hansen, W. J. Data Structures. Boston, Massachusetts : Little, Brown and Company, 1983.

Rubinstein, R. and Hersh, H. M. The Human Factor : Designing Computer Systems for People. Burlington, Massachusetts : Digital Press, 1984.

Spooner, C. R. "The ML Approach to the Readable All-Purpose Language." ACM Transaction on Programming Languages and Systems, Vol. 8, No. 2 (April 1986), pp. 215 - 243.

Tremblay, J. and Sorenson, P. G. An Implementation Guide to Compiler Writing. New York, New York : McGraw-Hill, 1982.

Tremblay, J. and Sorenson, P. G. The Theory and Practice of Compiler Writing. New York, New York : McGraw-Hill, 1985.

Wirth, N. Algorithms and Data Structures. Englewood Cliffs, New Jersey : Prentice-Hall, 1986.

APPENDIX A

SAMPLE RUN OF THE SYSTEM

The Data Structures Display System is designed to be almost fully menu-driven. There are four major menus:

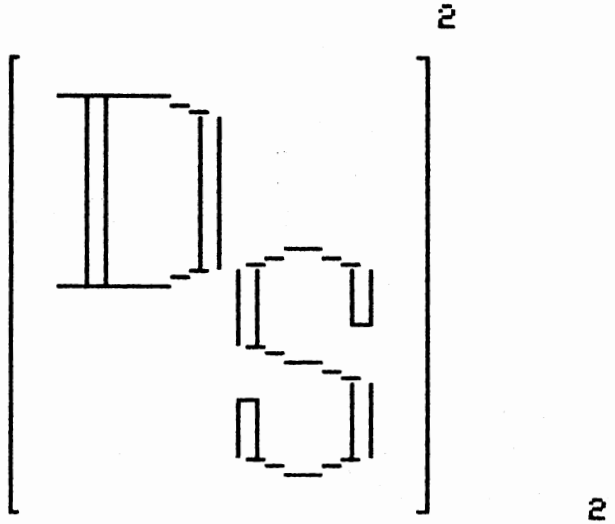
1. Main Menu
    a. Information
    b. Select Implementation Data Structure
    c. Define Implementation Data Structure
2. Abstract Data Structure Menu
    a. Information
    b. Select Abstract Data Structure
    c. Define Abstract Data Structure
3. Operation Menu
    a. Information
    b. Select Operation
    c. Define Operation
4. Execution Menu
    a. Information
    b. Browse Algorithm
    c. Browse Structure
    d. Execute Algorithm

Since the data structure display system is a visual learning tool, the best way to describe this system is to show what the user sees in a sample run. The following pages show the screens as a user works with the data structure display system. This sample run involves using the already defined insertion operation for a binary search tree.

A brief summary of the sequence of operation is given

below.

1. Title Page -- This is the first screen displayed by the system and stays on for two seconds.
2. Main Menu -- The user uses the number or SPACEBAR to highlight the desired option and press RETURN to select. In this case, the user selects option 2, "Select Implementation Data Structure."
3. Available Implementation Data Structures -- The user selects the desired implementation structure of a binary tree.

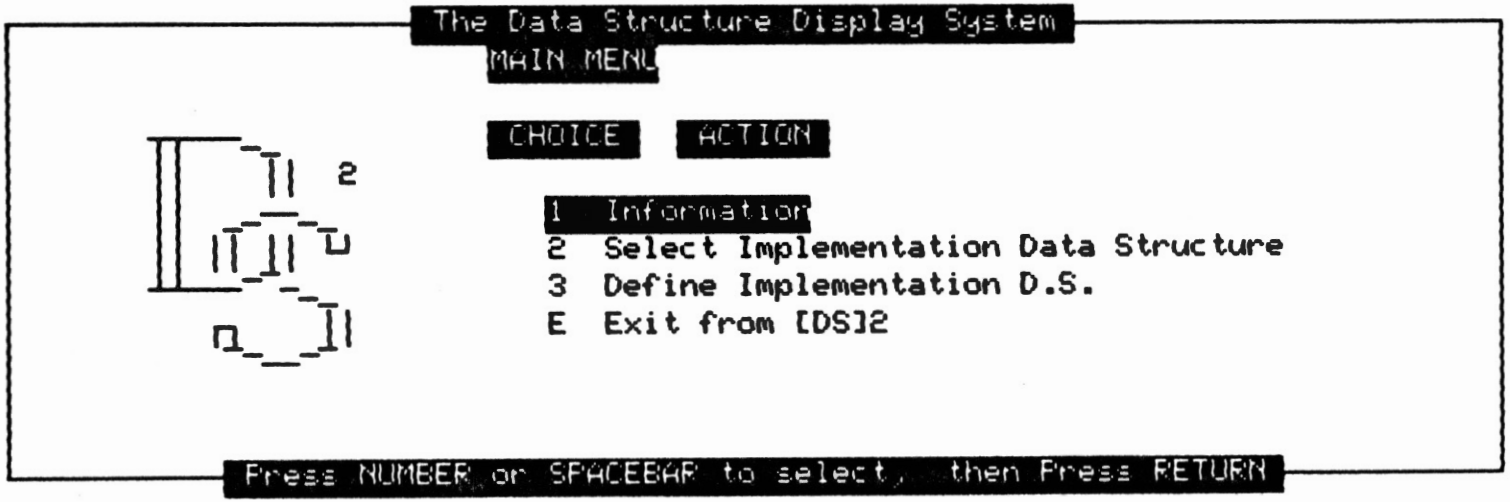


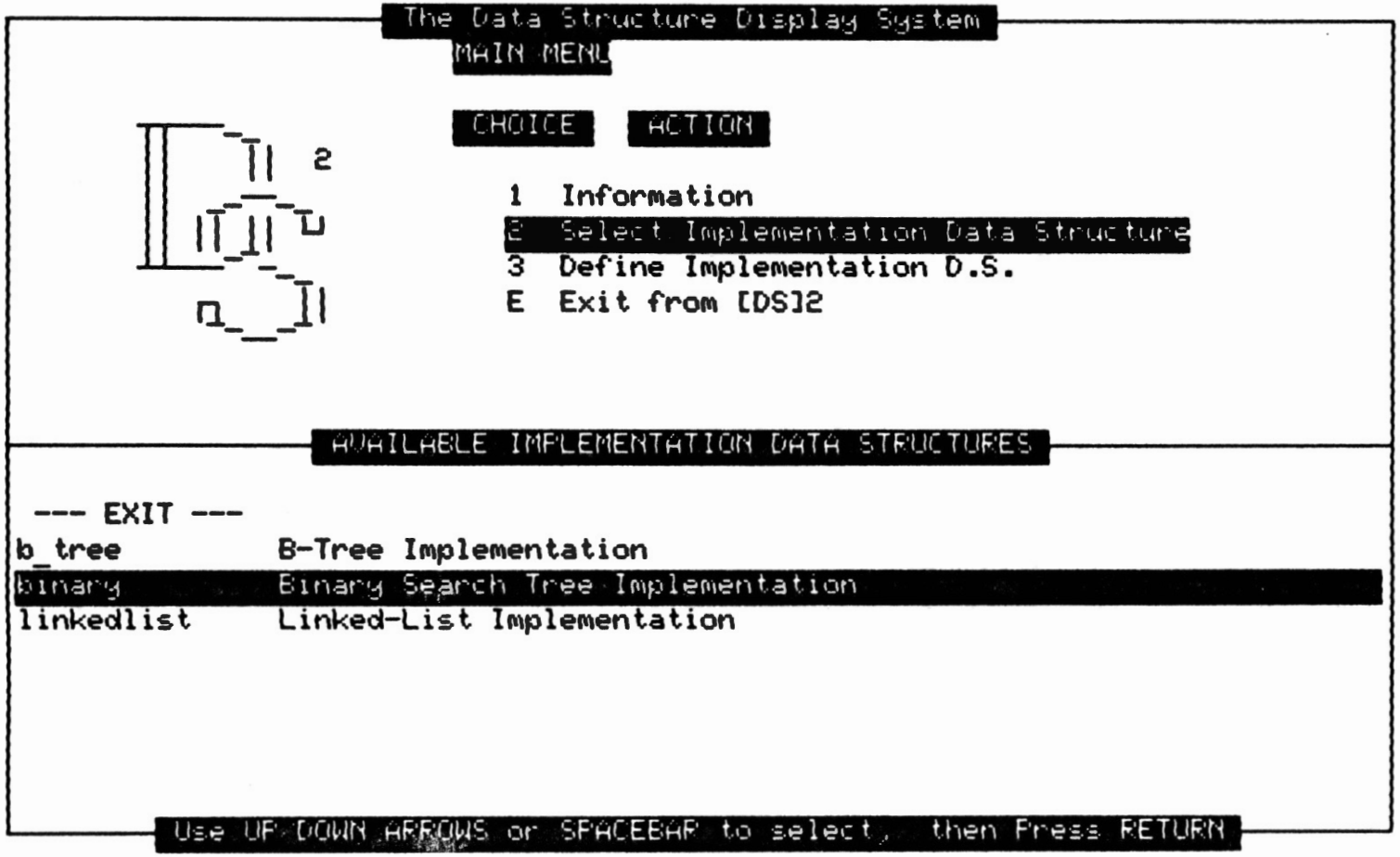
The Data Structure Display System ([DS])

A Thesis Project
by
Wilson Lee

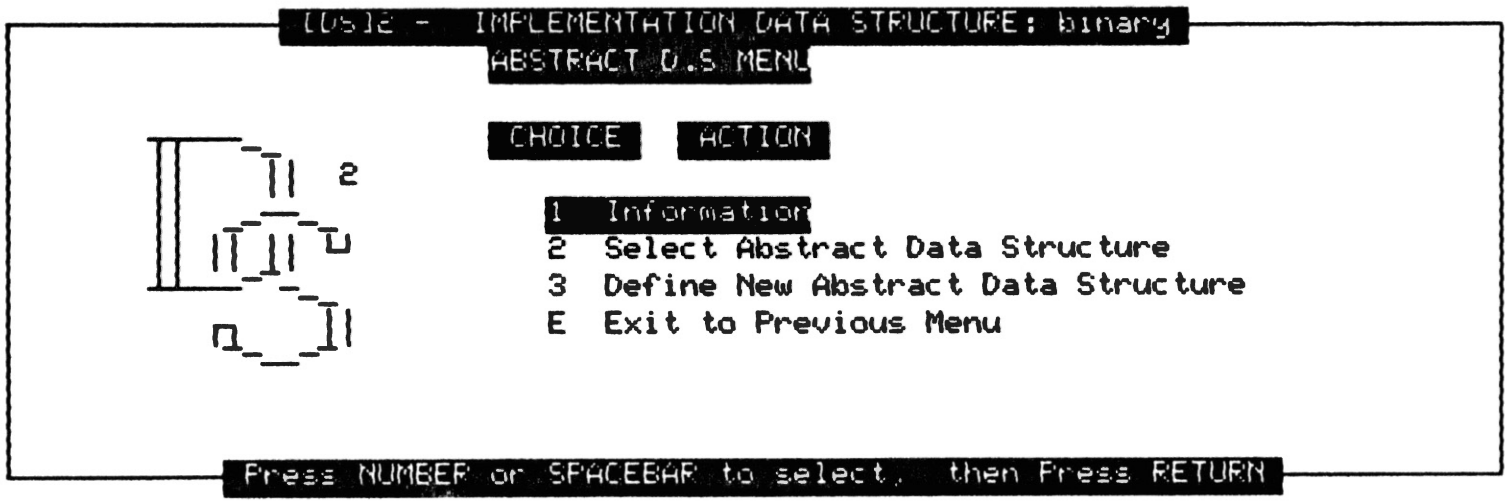
Welcome to the data structure program.... Written by Wilson Lee (1988)..

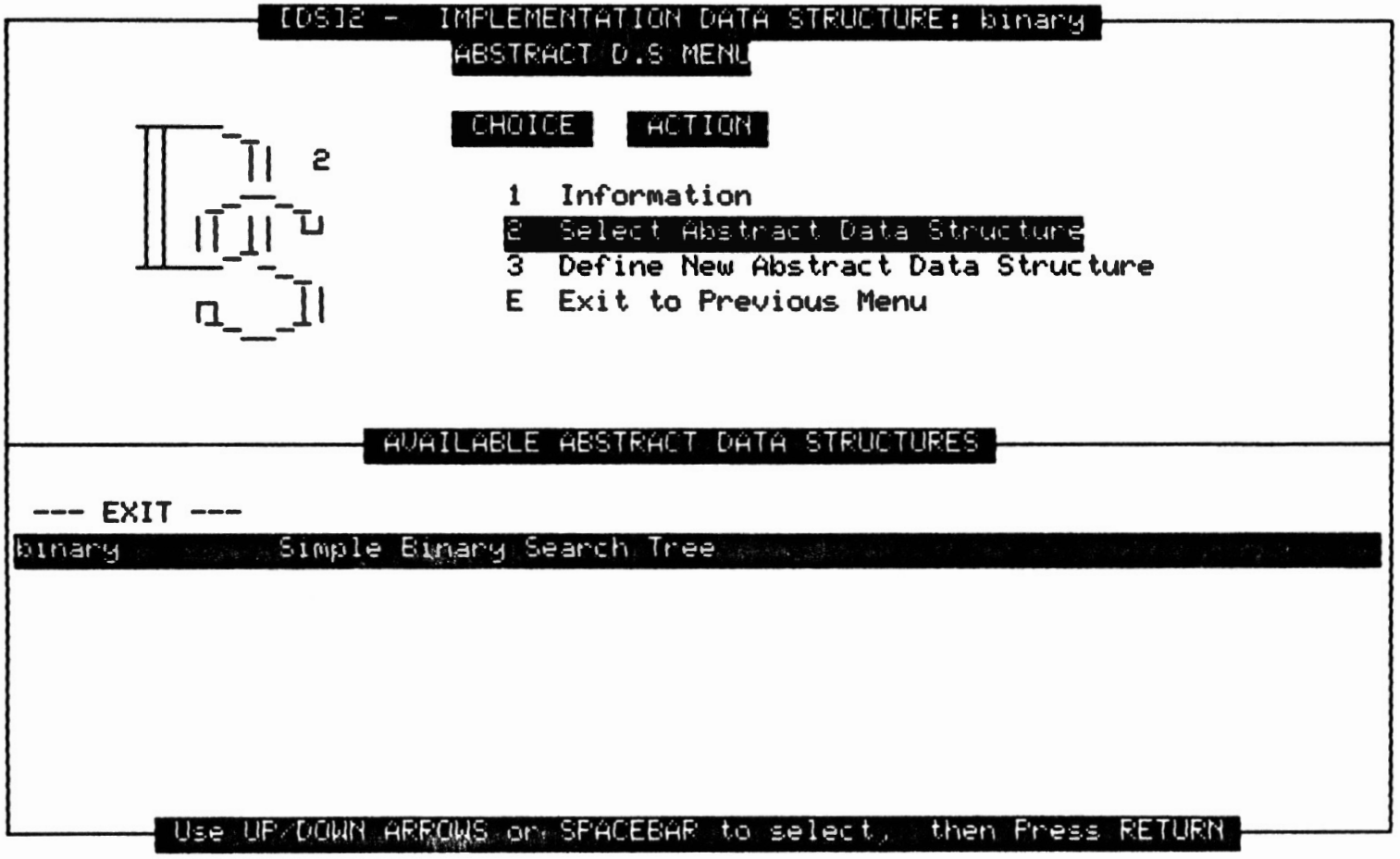
Department of Computing & Information Sciences
Oklahoma State University

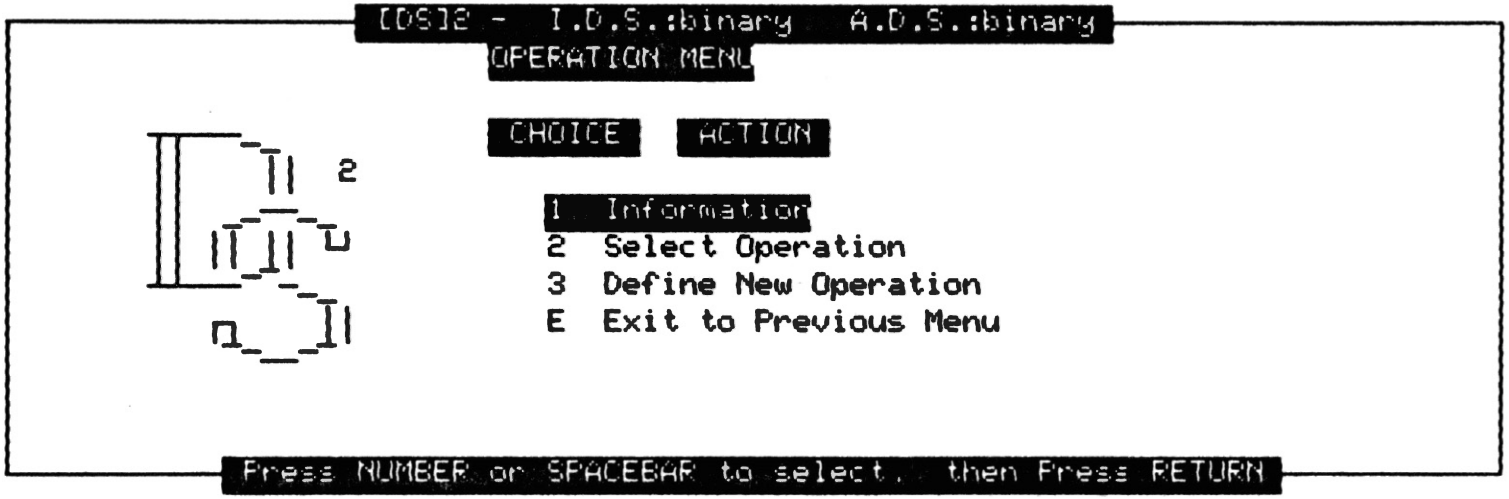

The Data Structure Display System
MAIN MENU
CHOICE
ACTION
1 Information
2 Select Implementation Data Structure
3 Define Implementation D.S.
E Exit from [DS]2
Press NUMBER or SPACEBAR to select, then Press RETURN

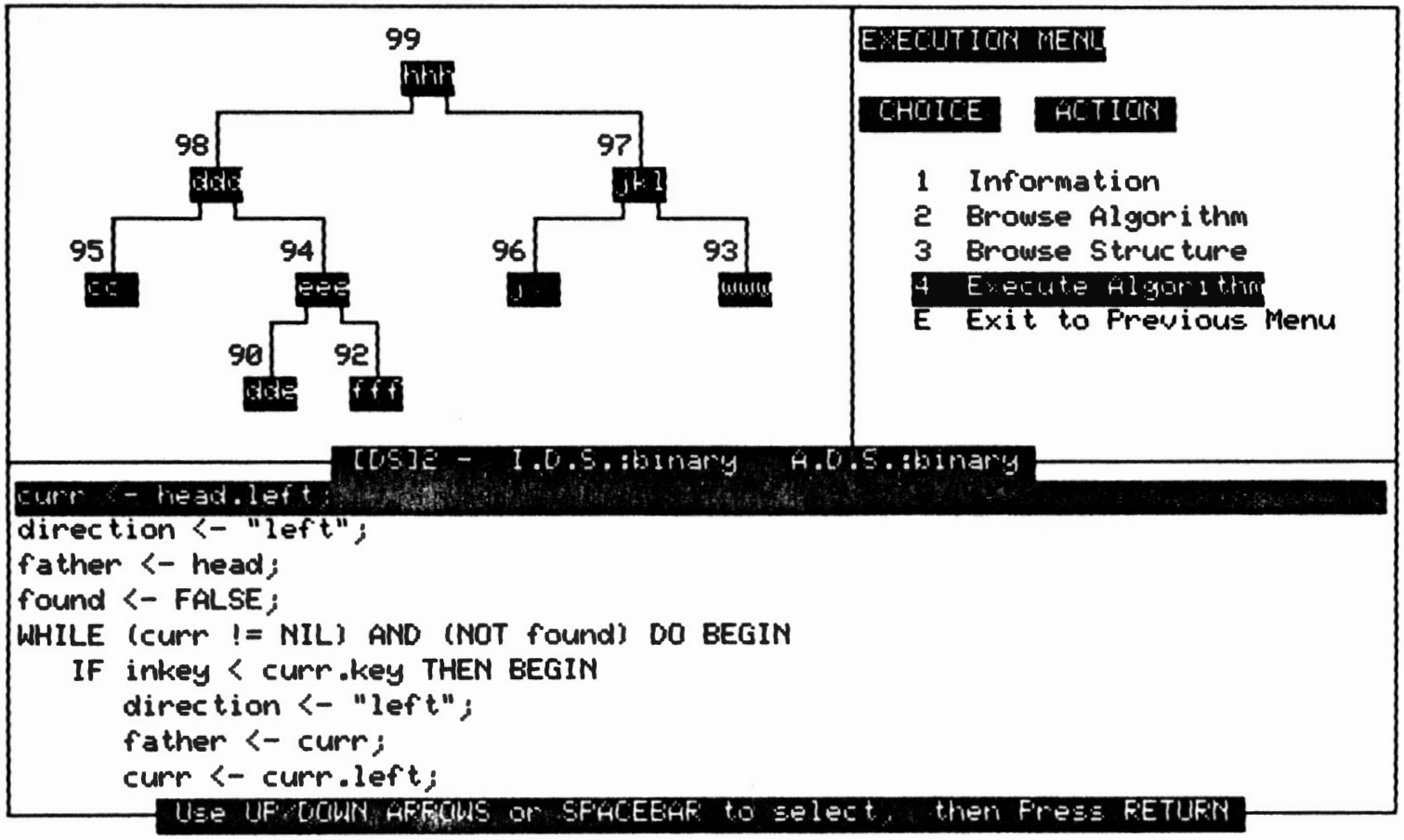

The Data Structure Display System
MAIN MENU
CHOICE
ACTION
1 Information
2 Select Implementation Data Structure
3 Define Implementation D.S.
E Exit from [DS]2
AVAILABLE IMPLEMENTATION DATA STRUCTURES
--- EXIT ---
b_tree B-Tree Implementation
binary Binary Search Tree Implementation
linkedlist Linked-List Implementation
Use UP/DOWN ARROWS or SPACEBAR to select, then Press RETURN

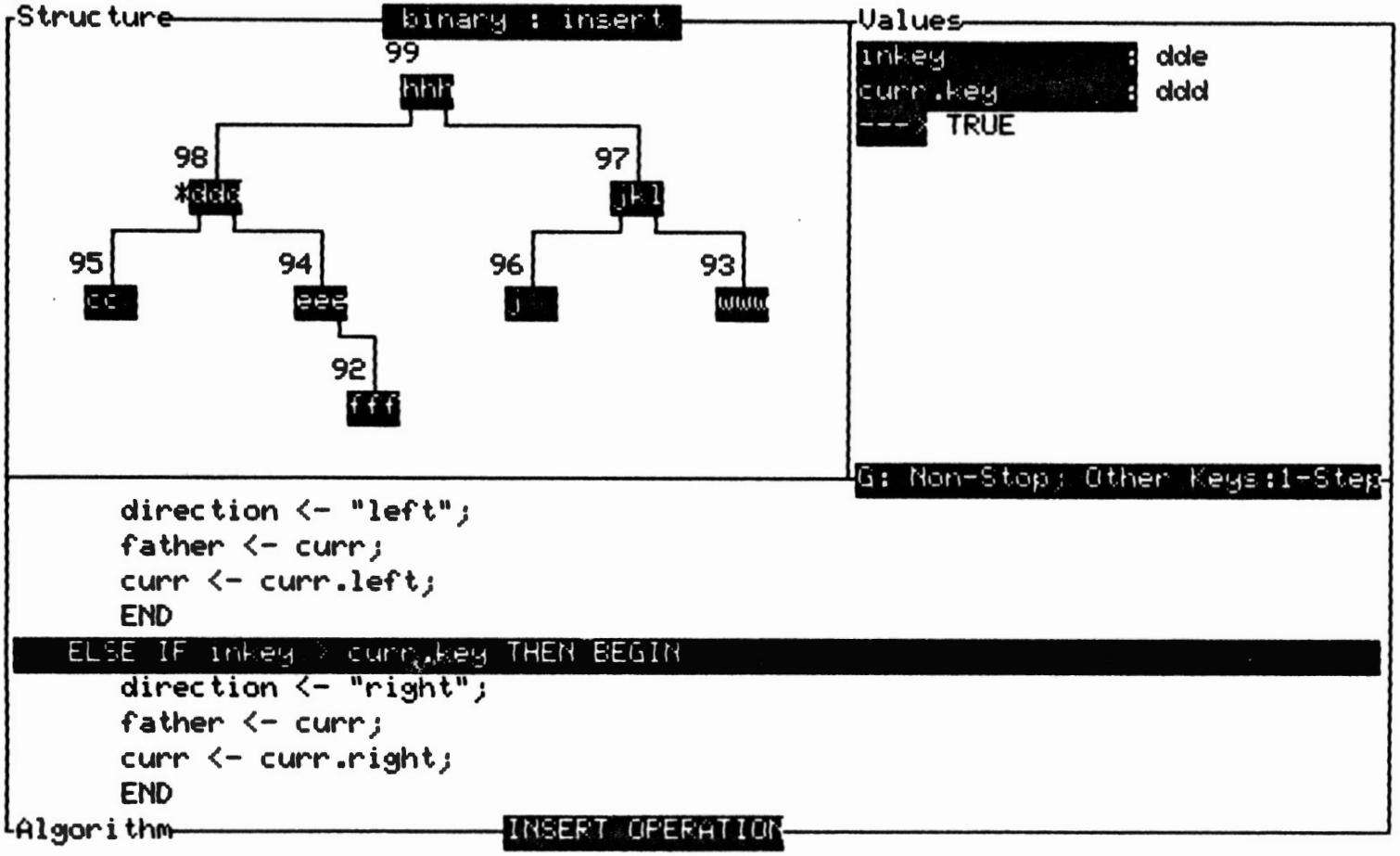
4. Abstract Structures Menu -- Having selected the implementation structure, the user now selects the option 2, "Select Abstract Data Structure."
5. Available Abstract Data Structures -- The user selects the binary search tree as the abstract data structure.
6. Operation Menu -- Having selected the implementation and abstract structures, the user chooses option 2 to select an operation. In this case the insertion operation is chosen.
7. Execution Menu -- After entering the name of the tree to be used, the execution menu is displayed. Option 4 is chosen to execute the insertion algorithm.
8. Execution Screen -- This is the screen displayed during the execution of an algorithm.


[DS]2 - IMPLEMENTATION DATA STRUCTURE: binary
ABSTRACT D.S MENU
CHOICE
ACTION
1 Information
2 Select Abstract Data Structure
3 Define New Abstract Data Structure
E Exit to Previous Menu
Press NUMBER or SPACEBAR to select, then Press RETURN


[DS]2 - IMPLEMENTATION DATA STRUCTURE: binary
ABSTRACT D.S MENU
CHOICE
ACTION
2
1 Information
2 Select Abstract Data Structure
3 Define New Abstract Data Structure
E Exit to Previous Menu
AVAILABLE ABSTRACT DATA STRUCTURES
--- EXIT ---
binary Simple Binary Search Tree
Use UP/DOWN ARROWS or SPACEBAR to select, then Press RETURN


[DS12 - I.D.S.:binary A.D.S.:binary
OPERATION MENU
CHOICE
ACTION
1 Information
2 Select Operation
3 Define New Operation
E Exit to Previous Menu
Press NUMBER or SPACEBAR to select, then Press RETURN


99
hhh
98
ddd
97
jkl
95
cc
94
eee
96
j
93
wwww
90
ddd
92
fff
EXECUTION MENU
CHOICE
ACTION
1 Information
2 Browse Algorithm
3 Browse Structure
4 Execute Algorithm
E Exit to Previous Menu
[DS]2 - I.D.S.:binary A.D.S.:binary
curr <- head.left;
direction <- "left";
father <- head;
found <- FALSE;
WHILE (curr != NIL) AND (NOT found) DO BEGIN
IF inkey < curr.key THEN BEGIN
direction <- "left";
father <- curr;
curr <- curr.left;
Use UP/DOWN ARROWS or SPACEBAR to select, then Press RETURN


Structure
binary : insert
99
hhh
98
*ddd
97
jkl
95
cc
94
eee
96
j
93
www
92
fff
Values
inkey : dde
curr.key : ddd
---> TRUE
G: Non-Stop; Other Keys:1-Step
direction <- "left";
father <- curr;
curr <- curr.left;
END
ELSE IF inkey > curr.key THEN BEGIN
direction <- "right";
father <- curr;
curr <- curr.right;
END
Algorithm
INSERT OPERATION

After the execution of the algorithm has completed then the execution menu is displayed again. From here the user can choose other operations, other structures, or exit from the system by selecting appropriate choices.

APPENDIX B

SAMPLE IMPLEMENTATION STRUCTURE DEFINITION

The Data Structure Display System interpreter and translator work according to the implementation structure definitions given in a file. Each implementation data structure owns a definition file. This file contains information on the components of a structure node. The first line of the structure definition file contains the following information:

1. Number of keys,
2. Number of Parent Pointers,
3. Number of Successor Pointers,
4. Number of Attributes, and
5. Field Number of Root Node holding Root Pointer.

The remaining lines in the file provide information on each individual field. The first number is the length of the field measured in bytes. The second number is the byte offset of that field from the start of the node. The third value on the line is the name of the field. This field name is used in when writing the Algorithm Specification Language Program.

The binary tree node structure definition file, "definition", is listed below. The last line in the file is for the unused bytes in the node (default node size of 40 bytes).

```
1    1    2    1    2
4         0         key
4         4         parent
4         8         left
4         12        right
4         16        attr
4         20        info
16        24        misc
```

APPENDIX C

SAMPLE ASL PROGRAM

The Data Structures Display System coordinates the execution of the algorithm with the display and the highlighting of the particular algorithm text line. The algorithm text is stored in "name.txt", where name is the short-form (10 characters maximum) name of the operation. This algorithm text is written using the Algorithm Specification Language (ASL) discussed in the body of this report. This algorithm text is translated to produce the pseudo-code instructions and the associated symbol table. The algorithm text, "insert.txt", for the binary search tree insert operation is listed below. The associated pseudo-code program, "insert.cde", and symbol table, "insert.tbl", are in Appendices D and E respectively.

```
ALGORITHM
   inkey : IN;
   found : OUT;
   curr  : OUT;
DECLARE
   head : HEAD;
   curr : NODE;
   father : NODE;
   direction : STRING;
   found : BOOLEAN;
BEGIN
   curr <- head.left;
   direction <- "left";
   father <- head;
   found <- FALSE;
   WHILE ((curr != NIL) AND (NOT found)) DO BEGIN
      IF (inkey < curr.key) THEN BEGIN
         direction <- "left";
         father <- curr;
         curr <- curr.left;
         END
      ELSE IF (inkey > curr.key) THEN BEGIN
         direction <- "right";
```

```
            father <- curr;
            curr <- curr.right;
            END
         ELSE found <- TRUE;
         END;
      IF (NOT found) THEN BEGIN
         curr <- NEWNODE();
         curr.parent <- father;
         curr.left <- NIL;
         curr.right <- NIL;
         curr.key <- inkey;
         IF (direction = "left") THEN
            father.left <- curr;
         ELSE father.right <- curr;
         END;
END.
```

APPENDIX D

SAMPLE PSEUDO-CODE PROGRAM

The Data Structures Display System interpreter executes the instructions in a specific pseudo-code file. This pseudo-code file is given the name "name.cde", where name is the short-form (10 characters maximum) name of the operation. A listing of the binary search tree insert operation pseudo-code program, "insert.cde", is given below. This program is translated from the algorithm text "insert.txt" given in Appendix C. The corresponding symbol table, "insert.tbl", is given in Appendix E.

```
ARG   31
LNO   1
MOV   11,26
LNO   2
MOV   29,32
LNO   3
MOV   17,23
LNO   4
MOV   30,2
LNO   5
TST   11,3,-4
TST   30,1,-4
AND
JMP   40,-2
LNO   6
TST   31,12,-6
JMP   25,-2
LNO   7
MOV   29, 32
LNO   8
MOV   17,11
LNO   9
MOV   11,14
LNO   10
JMP   38,-8
LNO   11
TST   31,12,-7
JMP   36,-2
LNO   12
MOV   29,33
LNO   13
```

```
MOV   17,11
LNO   14
MOV   11,15
LNO   15
JMP   38,-8
LNO   16
MOV   30,1
LNO   17
JMP   9,-8
LNO   18
TST   30,1,-4
JMP   61,-2
LNO   19
NEW   11
LNO   20
MOV   13,17
LNO   21
MOV   14,3
LNO   22
MOV   15,3
LNO   23
MOV   12,31
LNO   24
TST   29,32,-5
JMP   59,-2
LNO   25
MOV   20,11
JMP   61,-8
LNO   26
MOV   21,11
LNO   27
NOP
RET   30
RET   11
END
```

APPENDIX E

SAMPLE SYMBOL TABLE

The Data Structures Display System interpreter executes the pseudo-code instructions using the associated symbol table. The symbol table is stored in a file in the following format, each terminated by a newline character:

1. Name of Symbol,
2. Value of Symbol,
3. Type of Symbol,
4. Parent Symbol Location, and
5. Offset from Parent Symbol Location.

The symbol table file, "insert.tbl", for the binary search tree insert operation is listed below. The corresponding pseudo-code file, "insert.cde", is in Appendix D and the algorithm text, "insert.txt", is in Appendix C.

INSERT OPERATION

```
0
_TRUE
TRUE
9
1
1
_FALSE
FALSE
9
2
2
_NIL
NIL
9
3
3
_NEQ
NEQ
9
```

```
4
4
_EQ
EQ
9
5
5
_LESS
LESS
9
6
6
_GTR
GTR
9
7
7
_ALL
ALL
9
8
8
_LEQ
LEQ
9
9
9
_GEQ
GEQ
9
10
10
curr
undefined
6
11
11
curr.key
undefined
2
11
0
curr.parent
undefined
1
11
4
curr.left
undefined
1
11
8
curr.right
undefined
```

```
1
11
12
curr.val
undefined
2
11
16
father
undefined
6
17
17
father.key
undefined
2
17
0
father.parent
undefined
1
17
4
father.left
undefined
1
17
8
father.right
undefined
1
17
12
father.val
undefined
2
17
16
head
100
7
23
23
head.key
NIL
12
23
0
head.parent
NIL
11
23
4
head.left
```

```
NIL
11
23
8
head.right
NIL
11
23
12
head.val
NIL
12
23
16
direction
undefined
5
29
29
found
undefined
3
30
30
inkey
undefined
8
31
31

"left"
10
32
32

"right"
10
33
33
```

APPENDIX F

LIST OF IMPLEMENTATION PROGRAMS

The following list contains the names of the 'C' Language Programs and supporting files written to implement the Data Structures Display System. A listing of these programs is available at the Department of Computing and Information Sciences, Oklahoma State University.

Header Files

Directory: include/

instruction.h	- definitions for the pseudo-code instructions interpreter
screen.h	- macro definitions for VT100-type terminal screen control sequences including line-drawing characters

Main Driver Programs

Directory: src/

browsetext.c	- browse a named text file by using arrow keys or the space bar
get_term.c	- get terminal type from environment
indexmenu.c	- allow scrolling menu selection from items listed in a file
logo_at.c	- display [DS]2 logo at the screen location specified
main.c	- main driver for the display system and is the user-interface
menu.c	_ menu driver for both format one and two menus
screen.c	_ screen frames
title.c	- title screen
wrdcpy.c	- copy word from string

Display Driver Programs

Directory: src/

algorithm.c	- display algorithm text and highlighting a specific line
box_line.c	_ draw line box
display_val.c	_ display value of variables
draw.c	_ main graphics drawing routine for the graphical representation of the data structure
keyhit.c	- keyboard handler returns the key hit
line_to.c	- draw line from a starting point to an ending point
put_text.c	_ display text with screen character attribute at a given location
raw_gets.c	- raw mode gets()

Language Translation Programs

Directory: src/

translate.c	- main translation driver
lex.yy.c	- lexical analyzer generated by Lex
y.tab.c	- LALR(1) parser generated by Yacc

Pseudo-code Interpretation Programs

Directory: src/

struct_heap.c	- make structure into sequential representation
struct_mark.c	- mark named node with asterisk and calls draw() to redraw the local portion of the structure if necessary
interpret.c	- main interpretation driver
power.c	- x to power of y
read_code.c	- read pseudo-code from file
read_def.c	_ read implementation structure definition from file

read_table.c	- read symbol table from file
read_text.c	- read algorithm text from file
readsave.c	- read/save structure nodes from/to file

APPENDIX G

CONTEXT-FREE ASL GRAMMAR

The Algorithm Specification Language (ASL) has a context-free grammar, G_{ASL}. A grammar for any language has several properties: a set of terminal and nonterminal symbols, a starting symbol, and a set of rules. Aho [1972] states the following formal definition of a grammar.

> A grammar is a 4-tuple, $G = (N, \Sigma, P, S)$ where
> 1. N is a finite set of nonterminal symbols,
> 2. Σ is a finite set of terminal symbols, disjoint from N,
> 3. P is finite subset of $(N \cup \Sigma)^* \; N \; (N \cup \Sigma)^* \; X \; (N \cup \Sigma)^*$ where an element (α, β) in P is written $\alpha \longrightarrow \beta$ and called a production,
> 4. S is a distinguished symbol in N called the sentence (or start) symbol.

The grammar G is context-free if every production rule in P has the form $\alpha \longrightarrow \beta$, where α is a nonterminal symbol and β consists of zero or more terminal and nonterminal symbols.

The set of terminal symbols (or alphabet set Σ_{ASL}) consists of ASL keywords, identifiers, string constants, and integers. The ASL grammar alphabet set Σ_{ASL} is listed below.

AND	ALGORITHM	BEGIN	BOOLEAN
DECLARE	DEQUEUE	DO	ELSE
END	ENQUEUE	FALSE	FREENODE
HEAD	IF	IN	INPUT
INTEGER	NEWNODE	NIL	NODE
NOT	OR	OUT	OUTPUT
POP	PUSH	STRING	THEN
TRUE	WHILE	PERIOD	COMMA
COLON	SEMICOLON	EQUAL	NOTEQUAL
LESSTHAN	GREATERTHAN	LEFTBRACE	RIGHTBRACE
ASSIGN	PLUS	MINUS	GREATEREQUAL
LESSEQUAL	QUOTE	IDENTIFIER	NUMBER

During the language translation phase, these grammar alphabet members, σ_{ASL}, are actually tokens returned by the lexical analyzer. The lexical analyzer matches strings of input characters to the set of definitions for ASL keywords, identifiers, string constants, and integers. The ASL identifier consists of a letter and followed by zero or more letters or digits. A period is allowed in positions other than the first and the last positions to indicate a node field variable.

The Unix-style regular expressions for describing an ASL identifier are listed below.

```
letter          [a-zA-Z]
digit           [0-9]
identifier      {letter}(({letter}|{digit})*\.?
                        ({letter}|{digit})+)*
```

The symbols "[" and "]" indicate that one element from the set is chosen. The symbol "|" represents the boolean "or"

operation and the "?" indicates that the preceding symbol may or may not be present. The symbols "*" and "+" mean zero or more and one or more occurrences, respectively.

The set of nonterminals, N_{ASL}, consists of the symbols appearing on the left-hand side of the production rules, P_{ASL}. These nonterminals are listed below.

program	define_sect	parameter_sect
parameter	para_type	declare_sect
declare	data_type	body_sect
block	stmt_seq	stmt
assign_stmt	while_stmt	if_stmt
pop_stmt	push_stmt	deq_stmt
enq_stmt	new_stmt	free_stmt
input_stmt	output_stmt	control_expr
comparison	boolean_op	

The set of production rules, P_{ASL}, is listed below. The terminal symbols are in upper-case and the nonterminal symbols are in lower-case.

```
program             : define_sect body_sect
                    ;

define_sect         : algorithm parameter_sect DECLARE
                         declare_sect
                    | algorithm DECLARE declare_sect
                    ;

parameter_sect      : parameter_sect parameter
                    | parameter
                    ;

parameter           : IDENTIFIER COLON para_type SEMICOLON
                    ;

para_type           : IN
                    | OUT
                    ;

declare_sect        : declare_sect declare
                    | declare
                    ;
```

```
declare             : IDENTIFIER COLON data_type SEMICOLON
                    ;

data_type           : BOOLEAN
                    | HEAD
                    | INTEGER
                    | NODE
                    | STRING
                    ;

body_sect           : block PERIOD
                    ;

block               : BEGIN stmt_seq END
                    ;

stmt_seq            : stmt_seq stmt
                    | stmt
                    ;

stmt                : assign_stmt SEMICOLON
                    | while_stmt SEMICOLON
                    | if_stmt SEMICOLON
                    | pop_stmt SEMICOLON
                    | push_stmt SEMICOLON
                    | deq_stmt SEMICOLON
                    | enq_stmt SEMICOLON
                    | new_stmt SEMICOLON
                    | free_stmt SEMICOLON
                    | input_stmt SEMICOLON
                    | output_stmt SEMICOLON
                    ;

assign_stmt         : IDENTIFIER ASSIGN IDENTIFIER
                    | IDENTIFIER ASSIGN NUMBER
                    | IDENTIFIER ASSIGN QUOTE
                    | IDENTIFIER ASSIGN FALSE
                    | IDENTIFIER ASSIGN TRUE
                    | IDENTIFIER ASSIGN NIL
                    ;

while_stmt          : WHILE control_expr DO block
                    | WHILE control_expr DO stmt
                    ;

if_stmt             : IF control_expr THEN block
                    | IF control_expr THEN stmt
                    | IF control_expr THEN block ELSE
                        block
                    | IF control_expr THEN block ELSE stmt
                    | IF control_expr THEN stmt ELSE stmt
                    | IF control_expr THEN stmt ELSE
                        block
                    ;
```

```
pop_stmt            : IDENTIFIER ASSIGN POP LEFTBRACE
                        RIGHTBRACE
                    ;

push_stmt           : PUSH LEFTBRACE IDENTIFIER
                        RIGHTBRACE
                    ;

deq_stmt            : IDENTIFIER ASSIGN DEQUEUE LEFTBRACE
                        RIGHTBRACE
                    ;

enq_stmt            : ENQUEUE LEFTBRACE IDENTIFIER
                        RIGHTBRACE
                    ;

new_stmt            : IDENTIFIER ASSIGN NEWNODE LEFTBRACE
                        RIGHTBRACE
                    ;


free_stmt           : FREENODE LEFTBRACE IDENTIFIER
                        RIGHTBRACE
                    ;

input_stmt          : INPUT LEFTBRACE IDENTIFIER
                        RIGHTBRACE
                    ;

output_stmt         : OUTPUT LEFTBRACE IDENTIFIER
                        RIGHTBRACE
                    | OUTPUT LEFTBRACE QUOTE RIGHTBRACE
                    ;

control_expr        : LEFTBRACE control_expr AND
                        comparison RIGHTBRACE
                    | LEFTBRACE control_expr OR
                        comparison RIGHTBRACE
                    | comparison
                    ;

comparison          : LEFTBRACE IDENTIFIER boolean_op
                        IDENTIFIER RIGHTBRACE
                    | LEFTBRACE IDENTIFIER boolean_op
                        NUMBER RIGHTBRACE
                    | LEFTBRACE IDENTIFIER boolean_op
                        QUOTE RIGHTBRACE
                    | LEFTBRACE IDENTIFIER boolean_op
                        NIL RIGHTBRACE
                    | LEFTBRACE IDENTIFIER boolean_op
                        TRUE RIGHTBRACE
                    | LEFTBRACE IDENTIFIER boolean_op
                        FALSE RIGHTBRACE
                    | LEFTBRACE NOT IDENTIFIER RIGHTBRACE
```

```
                    ;

boolean_op          : EQUAL
                    | NOTEQUAL
                    | LESSTHAN
                    | GREATERTHAN
                    | LESSEQUAL
                    | GREATEREQUAL
                    ;
```

Each production rule has a nonterminal symbol on the left-hand side of the rule. This nonterminal symbol defines the partial language structure on the right-hand side. The language structure is described with terminal and nonterminal symbols.

A production rule can be written to define a partial language structure in terms of other partial language structures. In this way, the result is a production rule defining the overall structure of an ASL program. The distinguished nonterminal symbol, S_{ASL}, which defines the overall structure of the ASL is "program".

APPENDIX H

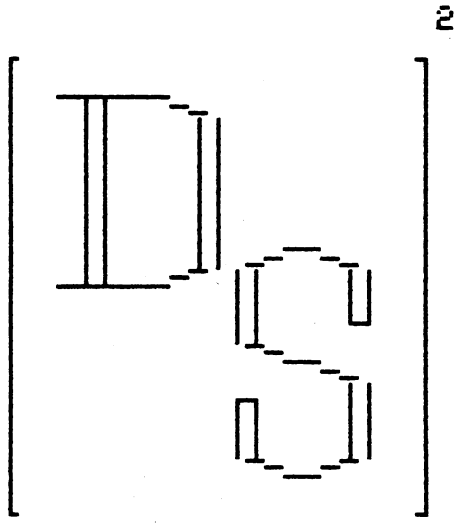
USERS' MANUAL

The Data Structures Display System Users' Manual provides information on the general use of this system. This document, "user_man", is kept in the directory "doc/". It can be printed by sending the formatted output to a printer. The following command accomplishes this:

```
pr66 user_man | lp -d<dest>
```

where <dest> is the name of the printer queue.
The data structure display system user's manual starts on the next page.

DATA STRUCTURE DISPLAY SYSTEM USERS' MANUAL



The Data Structure Display System ([DS])

The data structure display system allows the user to view and study an abstract data structure and its associated operations. This system displays the graphical representation of the structure. At the same time, the display system also displays the algorithm statement executed and the values of all variables in that statement.

This users' manual is written to guide the user in defining implementation and abstract data structures and their associated operations and algorithms. This manual also explains how the user applies an operation to an abstract data structure. Detailed discussion on the display system implementation design is not included in this manual but

it can be found in the Master of Science in Computer Science thesis report by Wilson Lee, 1988.

The data structure display system is covered in this users' manual in the following sequence.

1. System Overview.
2. Algorithm Specification Language (ASL) Program.
3. ASL Syntax Summary.
4. General Usage Instructions.
5. Screen Mode Selection.
6. Implementation Structure Definition.
7. Abstract Structure Definition.
8. Operation Definition.
9. Operation Execution.
10. Notes.

First, the users' manual provides an overview of the data structure display system. The language used to specify an algorithm is covered in sections two and three. The method of defining new structures and operations is discussed in sections six through eight. Section nine covers using an existing operation applied on a data structure. The user is assumed to have some experience programming in a high-level language and some knowledge of data structures.

This users' manual is kept in the data structure display system directory. It can be printed by sending the formatted output to a printer. The following command accomplishes this:

```
pr66 user_man | lp -d<dest>
```

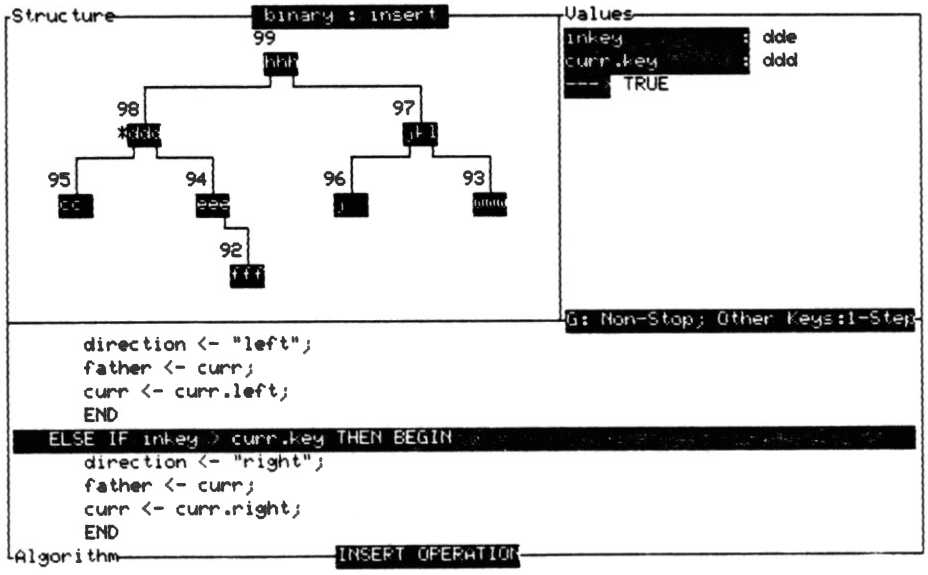
where <dest> is the name of the printer queue.

Please note any questions and suggestions and direct them to Wilson Lee through the Department of Computing and Information Sciences.

System Overview

The prototype data structure display system is implemented for VT100-type terminals. Any terminal that supports VT100-type screen controls and line drawing graphics characters can be used.

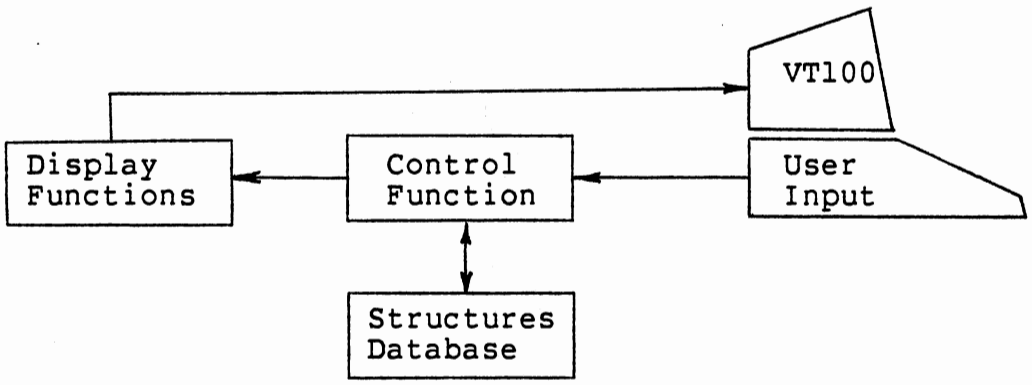
The data structure display system can operate in one-screen and two-screen modes. In the one-screen mode, the display system displays on one screen the graphical representation of part of the structure, the algorithm text, and the values of variables.



In addition to the one-screen display, the two-screen mode allows the user to see a larger partial graphical

representation of the data structure on the second screen. The method of mode selection is explained in a later section.

The data structures that can be handled by this system has a header node and a maximum of four successors. The limitation of four successors maximum is due to the hardware problem of displaying graphics as characters and not as a combination of pixels. The figure below shows the basic flow of control in the data structure display system.



The user is allowed the following operations:

1. select a data structure and any associated operation,
2. define new data structures and operations,
3. view the structure and algorithm of associated operations, and
4. step through an algorithm.

The control routines perform the following tasks:

1. handle user responses,
2. translate algorithm text into pseudo-code and an associated symbol table,
3. interpret the operation pseudo-code instructions,
4. access the structures database, and
5. supply the graphics driver with information.

The structure database contains the following parts:

1. algorithm text,
2. pseudo-code representation,
3. associated symbol table, and
4. specific information on structures.

Algorithm Specification Language (ASL) Program

The Algorithm Specification Language (ASL) for this system is designed both with simplicity and with readibility in mind. However, the user is assumed to be reasonably familiar with programming languages and to have some knowledge of linked data structures. It provides a minimal set of instructions sufficient for implementing simple non-recursive operations on dynamic linked data structures.

The ASL keywords are listed below.

ALGORITHM	BEGIN	BOOLEAN	DECLARE
DEQUEUE	DO	ELSE	END
ENQUEUE	FALSE	FREENODE	HEAD
IF	IN	INPUT	INTEGER
NEWNODE	NIL	NODE	NOT
OUT	OUTPUT	POP	PUSH
STRING	THEN	TRUE	WHILE

Although the list shown above uses all upper-case characters, it must be noted that the Algorithm Specification Language is case insensitive. However, the user is encouraged to have keywords in upper-case characters to improve readability.

A valid ASL identifier must begin with a letter and followed by zero or more letters or digits. An ASL keyword by itself cannot be used for an identifier name, although it may be embedded within other legal characters to create a valid identifier name. A period is embedded in a identifier name in the case of node field name. The maximum length of an identifier name is twenty characters.

The reserved symbols of ASL are listed below:

```
.        ,        :        ;
=        !=       <        >
(        )        <-
```

The basic structure of the ASL program is shown below.

```
ALGORITHM
   <parameter_sequence>
DECLARE
   <declaration_sequence>
BEGIN
   <statement_sequence>
END.
```

The first word in an ASL program is the keyword "ALGORITHM". The <parameter_sequence> provides the program a means of receiving and returning values. One point to note is that algorithms are executed and the values returned only at the end of the algorithm program. The <declaration_sequence> follows the "DECLARE" keyword. Here, variables can be declared to be of any of the valid data types. The current display system supports the data types HEAD, NODE, STRING, BOOLEAN, and INTEGER. The NODE data type is made up of several fields. These fields are defined in the structure definition file. The structure definition file contains information the user provides when he creates a new structure. The method of creating new structures is covered in a later section. The HEAD data type is similar to the

NODE data type except that it indicates a special purpose as the header node of a structure.

ASL statements may be any of the nine types: assignment, if-then-else, while-loop, push, pop, enqueue, dequeue, input, and output. These statements are placed between the "BEGIN" and "END" keywords. The logical end of the ASL program is indicated by the period following the "END" keyword.

The Algorithm Specification Language syntax is defined in the next section. An example of the assignment statement is shown below. The assign operator is "<-".

```
curr <- curr.left
```

In the above assignment statement, the variable "curr" is assigned the value in the node field "left" of the node pointed to by "curr". The left-hand side of the assignment statement is always an identifier. The data_type for the right-hand operand must match that of the left operand. See the syntax definition for more details.

The if-then-else statement has the following format.

```
IF  boolean-expression THEN  sequence-of-statements-1
                       ELSE  sequence-of-statements-2
```

The boolean-expression is evaluated and the appropriate sequence of statements is executed by an if-then-else statement. The else part of this statement format may be omitted if an if-then type of statement is desired.

The while-loop statement is similar to that used in the

Pascal language.

WHILE boolean-expression DO sequence-of-statements

The sequence of statements are executed while the boolean expression is tested to be true.

The push statement places the value of the identifier on top of the stack. There is only one stack available to the user algorithm. The pop statement returns the value on top of the stack.

identifier <- POP()

The format of the pop statement is similar to that of the assignment. The difference is that the value on top of the stack is assigned.

The enqueue and dequeue statement formats are shown below.

identifier <- DEQUEUE()

ENQUEUE(identifier)

These statements are used for managing a queue. On the current system the stack and the queue refer to the same data structure. Therefore, avoid using the queue statements when the stack is used.

The input and output statements are used for bringing values into the algorithm and displaying values of identifiers. The formats of these statements are shown below.

INPUT(identifier)

OUTPUT(identifier or string constant)

The output statement may be used to display an identifier's value or a string constant.

ASL Syntax Summary

The Algorithm Specification Language syntax is covered in this section. The following notation is used in the syntax definition of the Algorithm Specification Language (ASL).

[xxx]	xxx is optional
{xxx}*	xxx may be repeated 0 or more times
{xxx}+	xxx may be repeated 1 or more times
xxx|yyy	choose either xxx or yyy
xxx or yyy	choose either xxx or yyy

The ASL keywords are represented in this manual in upper-case letters and the program items in lower-case letters. The grouping of items is indicated by enclosing the items between backslashes "/". The item being defined is shown enclosed between the "<" and ">" characters, and the definition of this item follows indented on the next line.

The following definitions are for the basic items.

```
<digit>
     0|1|2|...|9

<letter>
     a|b|c|...|z|A|B|C|...|Z

<integer>
     \-|[+]\{digit}+

<identifier>
     letter{letter|digit}*[.{letter|digit}+]

<string>
     " {letter|digit|.| }* "
```

The following definitions are the program items.

```
<program>
     define_sect body_sect

<define_sect>
     ALGORITHM parameter_sect DECLARE declare_sect

<parameter_sect>
     { identifier : para_type ; }*

<para_type>
     IN | OUT

<declare_sect>
     { identifier : data_type ; }*

<data_type>
     BOOLEAN | HEAD | INTEGER | NODE | STRING

<body_sect>
     block .

<block>
     BEGIN END
  or BEGIN stmt_seq END

<stmt_seq>
     {stmt}+

<stmt>
     assign_stmt ;
  or while_stmt ;
  or if_stmt ;
  or pop_stmt ;
  or push_stmt ;
  or deq_stmt ;
  or enq_stmt ;
  or new_stmt ;
  or free_stmt ;
  or input_stmt ;
  or output_stmt ;

<assign_stmt>
     identifier <- value

<value>
     identifier
  or integer
  or string
  or TRUE
  or FALSE
  or NIL
```

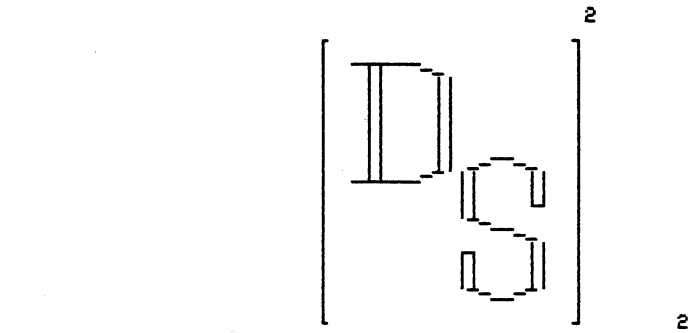
```
<while_stmt>
     WHILE control_expr DO stmt|block

<if_stmt>
     IF control_expr THEN stmt|block
                     [ELSE stmt|block ]

<pop_stmt>
     identifier <- POP ( )

<push_stmt>
     PUSH ( identifier )

<deq_stmt>
     identifier <- DEQUEUE ( )

<enq_stmt>
     ENQUEUE ( identifier )

<new_stmt>
     identifier <- NEWNODE ( )

<free_stmt>
     FREENODE ( identifier )

<input_stmt>
     INPUT ( identifier )

<output_stmt>
     OUTPUT ( identifier|string )

<control_expr>
     ( control_expr AND|OR comparison )
  or comparison

<comparison>
     ( identifier boolean_op
       identifier|integer|string|NIL|TRUE|FALSE )
  or ( NOT identifier )

<boolean_op>
     EQUAL | NOTEQUAL | LESSTHAN
  or GREATERTHAN | LESSEQUAL | GREATEREQUAL
```

General Usage Instructions

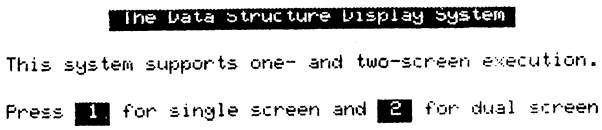
The user must have a valid data structure display system account to use this system. The data structure display system must be run on a terminal that supports VT100-type screen controls and line drawing graphics characters. Log onto the data structure display system account and type "ds2" at the Unix prompt. This command executes the "ds2" program which is the main driver for the data structure display system. If you are using a Tandy DT100 terminal, set the Unix environment variable "TERM" by typing "setenv TERM dt100" before running the program.

After displaying the title screen for approximately two seconds, the data structure display system asks the user to select the screen mode. Selection of screen mode is covered in the next section.

Selecting Screen Mode



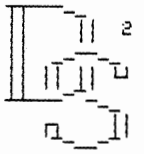
The data structure display system has two modes of operation: one-screen and two-screen modes. The display system prints a message after displaying the title screen asking the user to select either one-screen or two-screen mode.



The user must press "1" or "2" in response to that question. In the case of pressing "2" to indicate the two-screen mode, the display system provides instructions to prepare the second terminal.

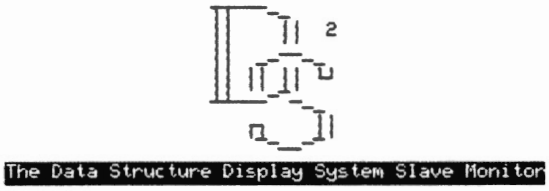
```
The Data Structure Display System

This system supports one- and two-screen execution.

To use two screens, log on to this account on another
terminal and type 'slave' at the unix prompt.

The system displays a title screen.
Wait for a 'message to continue' on the
second screen.

Press RETURN when the message appears.
```

The user should now log onto the data structure display system account at a second terminal and type "slave" at the Unix prompt.



```
The Data Structure Display System Slave Monitor

Setting up communications; one moment please . .
```

The second screen shows the slave monitor title screen and displays the following message: "Setting up communications; one moment please ".



When communication has been set up with the first terminal this second terminal displays the message: "This terminal is ready; press RETURN at the other terminal". Now the user hits the RETURN key and the display system shows the main menu on the first terminal.

All user input is through the first terminal. The second terminal is only used as a data structure display.

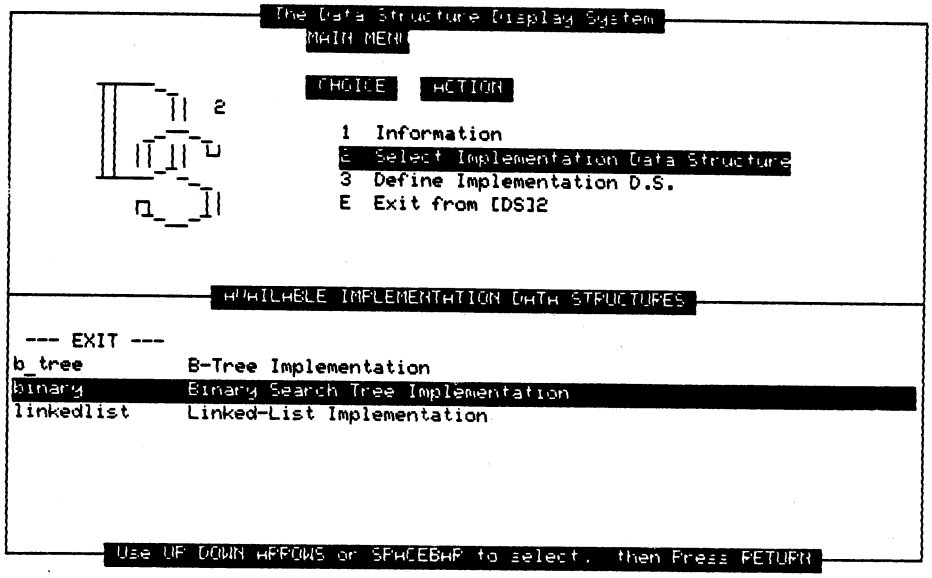
Defining An Implementation Structure

Before the data structure display system can be used, the user must define implementation and abstract data structures. The operations and algorithms must also be defined.

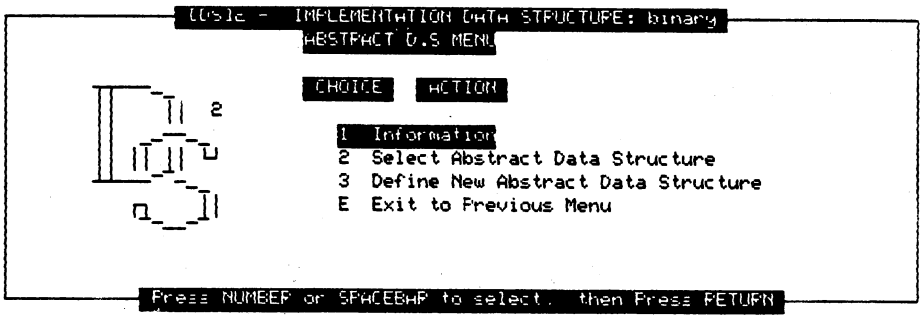
The first step is to define the implementation data structure. The user selects the definition option at the main menu. The display system displays the information it needs to know, namely: number of keys, parent pointers, child pointers, and attributes in a node in the implementation structure. The display system also asks for the name for each field of the node structure.

The display system returns to the main menu after the user has answered all questions.

Defining An Abstract Structure



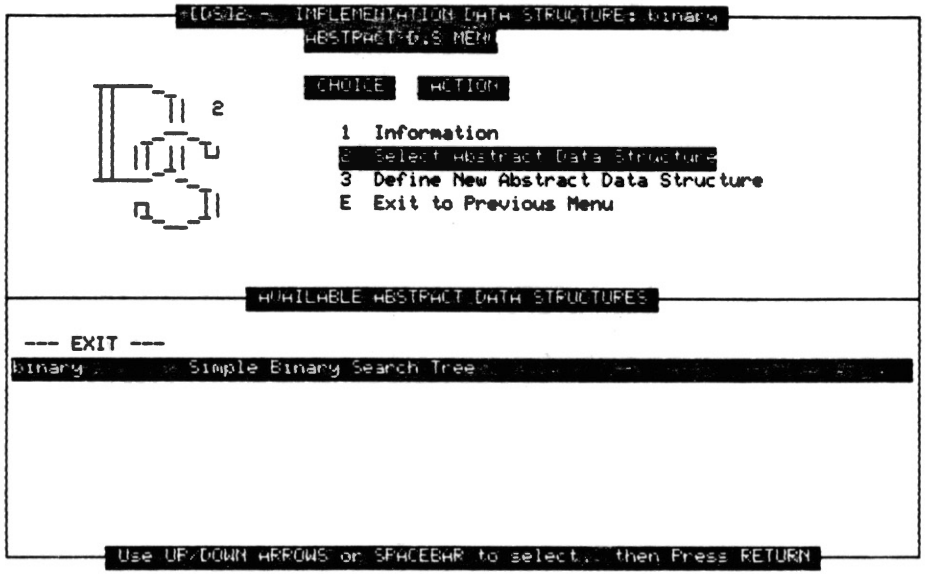
After defining the implementation structure, the user selects the "Select Implementation Data Structure" option at the main menu. After selecting the desired implementation data structure, the display system now displays the abstract structure menu.



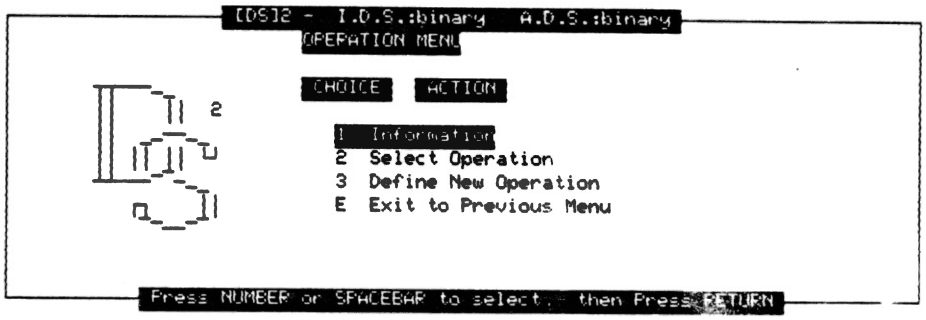
The user selects the definition option at the abstract structure menu. The display system displays a message and asks the user for a name for the new abstract structure. If the name is non-existent then the data structure display system creates a new abstract structure.

The display system returns to the main menu if the user's answers are valid.

Defining An Operation



An operation associated with an existing abstract structure can be defined by first selecting the "Select Abstract Data Structure" option at the abstract structure menu. After selecting the desired abstract data structure, the display system displays the operations menu. Now the user selects the definition option.

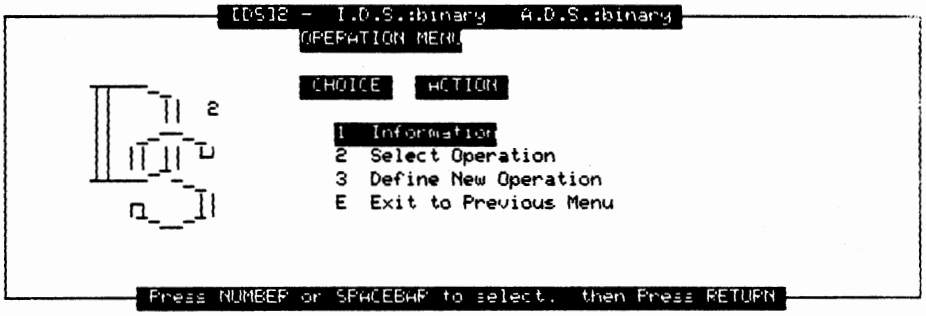


An operation consists of a number of system utilities and algorithms. The display system shows the available system utilities and the existing algorithms for this abstract structure. The user types in the correct sequence for the system utilities and algorithms. The system then prepares for the user to type in all the non-existent algorithms. The user enters the algorithm text using the Unix "vi" editor. The user can learn more about the "vi" editor by reading any Unix users' guide.

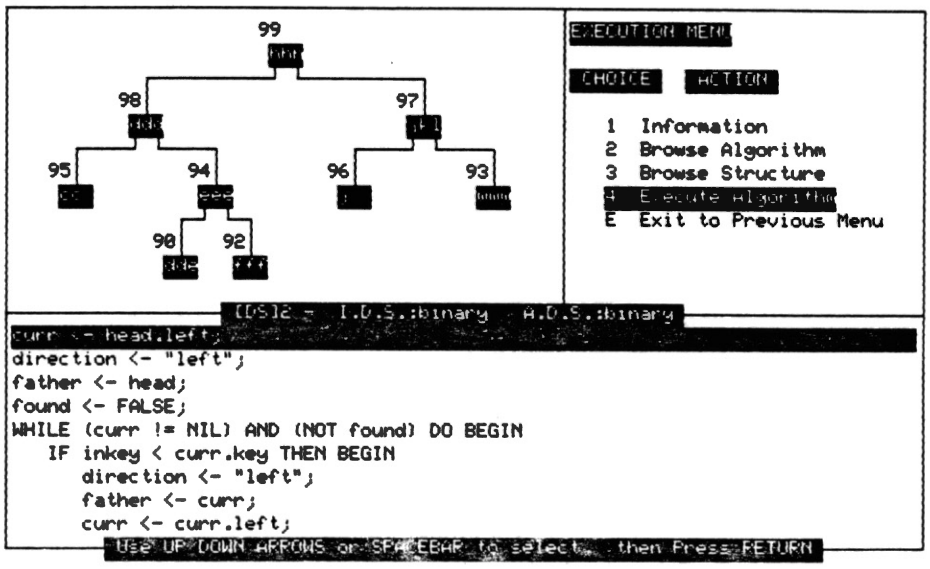
The system returns to the operations menu after the user completes entering all required information.

Execution Of An Operation

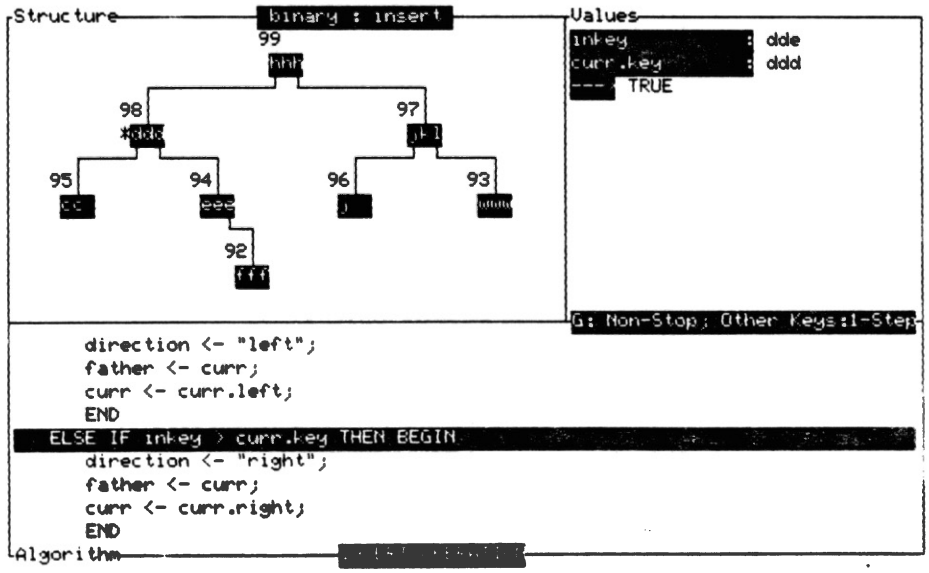
The execution of an operation on an abstract data structure can only be done after the implementation structure, abstract structure, and the associated operation have been defined.



The user selects the "Select Operation" option at the operations menu. After an operation is chosen, the display system shows a three-window screen with the execution menu. The user executes the operation by selecting the "Execute Algorithm" option at this menu.



The display system now leads the user through the algorithm execution step by step.



After execution has completed, the execution menu is displayed again.

VITA 2

Wilson Lee

Candidate for the Degree of

Master of Science

Thesis: AN IMPLEMENTATION OF A DATA STRUCTURES DISPLAY SYSTEM

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Singapore, September 19, 1957, the son of Mr. and Mrs. Soon Aik Lee.

Education: Graduated from Raffles Institution in December 1975; received Technician Diploma in Electrical Engineering from Singapore Polytechnic in May 1983; received Bachelor of Science Degree in Computing and Information Sciences from Oklahoma State University in May 1986; completed requirements for the Master of Science Degree at Oklahoma State University in December 1988.

Professional Experience: Computer Programmer, State 4-H Programs Office, March 1986 to July 1986; Teaching Assistant, Department of Computing and Information Sciences, Oklahoma State University, August 1986 to May 1988.