BOTANIA - AN INTERACTIVE SOFTWARE SYSTEM FOR RANDOM ACCESS PLANT IDENTIFICATION

by

Anne-Lise Kathleen Laùx

A THESIS

Presented to the Department of Computer and
Information Science,
the Department of General Science,
and the Honors College of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Bachelor of Arts

June 1999

Approved: 

Professor Michal Young

An Abstract of the Thesis of

Anne-Lise Kathleen Laux for the degree of Bachelor of Arts

in the Department of Computer and Information Science and the Department of

General Science to be taken June 1999

Title: BOTANIA - AN INTERACTIVE SOFTWARE SYSTEM FOR RANDOM ACCESS PLANT IDENTIFICATION

Approved: [redacted]

In the field of botany, plant identification has been practiced for thousands of years. The current standard for plant identification involves the use of a dichotomous key. This form of identification is based on a binary tree structure, i.e., a sequence of decisions with only two possible outcomes. While relatively straightforward to use, this is not the most efficient method of identification because one must always start at the same place and answer questions in the prescribed order. Another method, called a random access key is available. With this form of key users can begin by identifying any characteristic of the plant that they wish, and can continue in any order. While difficult to implement on paper, this form of key is ideally suited for a software application. We created an interactive random access software application for plant identification for the Windows9x/NT platform. This thesis explores the issues involved with the creation of this application.

ACKNOWLEDGMENTS

The author expresses sincere appreciation to Professor Young, Professor Walkling, and Dr. Wagner for their assistance in the preparation of this manuscript. In addition, special thanks are due to Dr. Wagner who spent many hours writing the key to liverworts that was used in this thesis. This thesis would not have been possible without his efforts. I also thank Scott Herz for his technical advice and support throughout this project. I thank Minhoon Kim for his work on the production of the splash screen graphic as well as the icons. I thank Connie Wagner for her advice and support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Computer technology has had a major impact on the field of science in the past several decades, so the idea of creating a scientific software application is not new. Recently the field of *botany*[1] has emerged as an area extremely welcome to the technologies that the computer industry has to offer. Plant identification in particular lends itself well to being implemented in software applications.

Plant identification is the process of determining the identity of an organism by giving it a scientific name. For experts and novices alike, the process of identification most often involves the use of some type of key. A *key* is "an arrangement of the salient *characters* of a group of plants or animals or of *taxa* designed to facilitate identification"(WWWebster Dictionary). Although keys can be used to identify virtually anything, this thesis focuses only on the use of keys with regard to the identification of plants. Keying began as an attempt to make identification easier and evolved in many ways over the years. For the past twenty-five years, the standard keying format used for plant identification has been the *dichotomous* key. A dichotomous key is by definition, "a key for the identification of organisms based on a series of choices between two [mutually

[1] Italicized words can be found in the glossary.

exclusive] alternative characters"(WWWebster Dictionary). This form of identification is based on a *binary tree structure,* such that one must begin at the *root* and by making choices, travel down the *branches* until a single unique identity is reached. Each pair of opposite *character*s is called a *"couplet."* This method requires that the user must start at the beginning and answer questions in the prescribed order.

While popular and relatively straightforward to use, the dichotomous key is by no means the only method for identifying plants, nor is it the most efficient. Another method, called a random access key is also available. In this form of key users can begin by identifying any characteristic of the plant, and can continue in any order. While also difficult to implement on paper, this form of key is ideally suited for a software application. It is the goal of this thesis to explore and implement an alternative to a dichotomous key as a means for plant identification, namely a computerized interactive random access key.

Plant identification can be a complex process, often involving highly technical botanical terminology. It is also complex because plants are not *static.* Not only do they vary greatly from *species* to species, but individuals of the same species can also differ, and further, even the amount of variation can vary. In addition, plants vary in time as they mature and as the seasons change. Difficulties in identification can occur if the botanist is not familiar with the characters in which variance frequently occurs, or because the means of identification might not take into account this variance. For instance, to identify a plant using a dichotomous key, one must begin with the first set of characters given by the key and choose the most appropriate. With each choice, the botanist comes to a new couplet of characters to choose between. This process is repeated until a unique identification is reached. It could happen that the plant does not seem to fit

either of the characteristics given by the key at a particular stage. This occurs frequently when a specimen is incomplete or at a different stage of maturity. Regardless of the reason, the botanist is at a standstill. His/her only choice is to suppose that both paths are true and follow each out to an answer. This can double the time that must be spent to get an accurate answer. If this occurs more than once during the identification process, the time it takes to follow each of the paths to its end and compare the answers could increase drastically.

This thesis explores the issues with non-computerized dichotomous and random access keys and then presents a solution to these issues in the form of an interactive software-based random access key. The format of this thesis has the following structure: "Problem" presents the limitations of current non-computerized keys; "Approach" provides a discussion of our solution to these problems by detailing the internal structure of "Botania." Words appearing in italics are defined in the glossary at the end of the paper. Words in `Courier` font represent actual code used in the program.

CHAPTER II

PROBLEM

The process of following a dichotomous key is considered simple in theory because the process is unambiguous. The basic structure is always the same, i.e., choosing between two opposite characters. By definition the plant should be positive for one of the two choices.

A dichotomous key is not necessarily easy to write. There may be many ways to identify a plant, so the author must decide what he/she believes is the best order for dividing characters. For this reason, keys written by different people tend to vary greatly. Dichotomous keys are also difficult to write because it is not always possible to divide characteristics that have multiple options into only two choices. The way the choices are divided depends solely on the opinions of the key writer. This means that the quality and content of the key is entirely dependent on the author and his/her opinions on the relevant distinguishing characters. Because there are so many ways to divide the character choices, it is not unusual for authors to write more than one key to a group. For instance, many authors write one key based mainly on vegetative characters and another based mainly on reproductive characters. A widely recognized limitation with dichotomous keys is that they do not necessarily contain complete data

on each plant. For instance, a plant might have purple spots on the stem, but if that characteristic is not chosen by the author of the key as an identifying character, then it may not be present in the key.

In addition to the problems involved with authoring dichotomous keys, there are also many problems that occur with the use of these keys. Perhaps the most broadly recognized limitation is that the user must progress sequentially through the key. Every character can be used only where it occurs in the key and one must always start at the beginning. For example, a user can not simply look for a couplet where plants with red flowers are separated from those with blue flowers because there might be multiple places in the key where these two characters are separated. The only way to know which ones apply to the user's plant is for the user to start at the beginning and make correct choices every step of the way until finally reaching the correct couplet with red versus blue flowers. This limitation is another reason why authors frequently present more than one set of keys.

To bypass some of these problems, a random access keying system can be used. It is a different form of key that is based on the idea that any character can be accessed and specified at any time. A user can randomly choose any character and have access to specifying that character in the key. A truly random access key does not provide any prioritization of characteristics. In other words, each characteristic has equal weight and can be specified at any time. There are many different methods for implementing this type of key, two of which are discussed in the next section.

Random access keys can be unwieldy. Without a computer's ability to sort a large number of data, a large random access key is frustratingly cumbersome to use. One example of this is discussed below with the example of Clark

and Frye's book. This book has numerous separate tables often with individual tables spanning several pages. This can be difficult to use because of all the page flipping that invariably occurs.

One issue that is common to all paper keys including dichotomous and random access keys is that of the user having to flip to the end of the book to look up a term in the glossary. With the power of computers, glossary definitions can be reached at the touch of a button.

CHAPTER III

APPROACH

Introduction to Botania

The goal of this thesis is to create a software application for the Windows 9x/NT operating systems that can be used to identify plants. In this thesis, leafy *liverworts* are used as an examplar group of plants, however any dataset can be substituted at any time. The application is interactive and allows the user to choose the relevant characters of his/her specimen from a complete list of characteristics. When the user submits the characteristics, the number of remaining species that fit those criteria is decreased. The user can choose to look at images and descriptions of the remaining species at any time. There is also a "Glossary" window where the user can find definitions for terms that are unfamiliar, and a "History" window where previous characteristic submissions are logged. The entire program is titled "Botania".

The possibilities for creating a software application that implements a random access key are numerous and can depend on several factors. One factor is the operating system that the application is being created for. The tools available for different operating systems vary widely in their capabilities and in the

options they provide. This is especially relevant regarding *user interface* items and *storage mechanisms*. Another factor is the prospective users of the program. It is important, especially with this kind of system, to determine whether the intended users are novices, experts or both. Finally, it is important to take into account the nature of the process that is being implemented. This can be done best by exploring the advantages and limitations of other implementations of random access keys.

A Study of Two Existing Implementations

We began the design process by looking at existing random access keys that are not software based. One key is in a book written in 1928 on the identification of liverworts (Clark and Frye 1928). This book uses tables of characteristics to distinguish between liverwort *genera* and liverwort species. For instance in Figure 1, each row of the table contains a distinct character and each column contains a *species*. Each *cell* of the table contains a symbol that represents the state of the character given on the row, for the genus given in that column. The most common symbols used are the "+" representing a positive occurrence of the characteristic or a "-" indicating a negative occurrence. However, other symbols are used in cases where more than one *character state* is present. For instance, often the species within a genus vary such that some individuals are positive for the character and some individuals are negative. The symbol "±" is used for this. Sometimes where measurements are part of the character, the range is given rather than a symbol. Often there are multiple possibilities for the state of a character. For instance, the leaf lobes of a liverwort could be acute, obtuse or

rounded, so the symbols correspond to the first letter of the state that is positive for that genus. Once the genus of the specimen is determined, the taxonomist can flip to the corresponding table in the book containing characters that distinguish between the species contained within that genus. In this way, a person can identify a liverwort using Clarke and Frye's book.

The other existing key system that we studied is "The Random Access Key to the Genera of Colorado Mosses"(Weber 1972). This approach makes use of a set of punchcards in which each card represents a character state, and the placement of the holes in the card corresponds to the genera that are positive for the character state of the card. The key comes with a booklet containing a numbered list of characters. The user selects a character from the list that applies to his/her specimen and then draws the corresponding punchcard from the deck. This process is repeated, and each time the user draws a new card, it is superimposed with the cards previously pulled. Holes that are still present after the superimposing represent the group of genera that fit all the characteristics the user has selected. Eventually a point will be reached where there is only one hole still open in the stack, at which time a unique identification has been reached. Next the user lays the final "overlay" card over the top. The "overlay" card has numbers next to each hole, so the user may select the number that lines up with the remaining hole and look up that number in the booklet to determine the genus name. At this point, the user should have a positive identification of his/her specimen. This system is random access because the user can select cards in any order and a unique identification may be reached with more than one combination of characters.

Although this is not an interactive computerized implementation, computers were used to generate the punchcards for this key. As punchcard comput-

ers were common at the time this key was written, the author saw the possibility of using automatic card sorting machines to do the selecting and refiling of the cards. Because most users did not have access to a punchcard machine, this became more commonly used by hand. Use of this key by hand brings up the issue of having to manually refile the character state cards. Because there are close to 200 cards in the deck, it is essential to keep the cards in their numerical order so that they can be found. This means that every time a user wants to identify a plant, he/she must afterwards spend a significant amount of time refiling the cards. This is a tedious and time-consuming process.

This punchcard implementation does not scale well. The data used for this particular implementation were the genera of Colorado mosses which had a relatively small number of characters. Using a punchcard system for a group of data like all flowering plants of North America would be literally impossible because of the enormous number of cards representing character states that would be needed to identify each of the plants in this set.

These two systems are interesting because they provide two very different implementations of the same basic idea. The system that uses the book tables allows several possible choices for each field. The punchcard system on the other hand calls for *binary* choices. This means that the descriptions of characters have to be written so that all choices require yes or no as a response rather than choosing from among multiple character states. One interesting characteristic of Example 1 is its separation of the characters useful for distinguishing genera from species. The key writers determined that it is valuable to distinguish between characteristics useful for distinguishing genera and those useful for the species within a genus. Example 2 excludes species altogether in its implementation because it states that for the data set (mosses of Colorado,)

many genera contain only one or two species and that narrowing down to a single genus is sufficient for identification.

There are some valuable time and space saving reasons why it is a worthwhile practice to distinguish between characters specific to genera and those more specific to species. If all characters needed to distinguish between a group of 100 species, for example, are placed in one large table, several things will happen. The characters listed will be so specific that each applies only to a few species. Each selection will only narrow a few species, thereby lengthening the amount of time needed to reach a unique identification. Although placing the genus and species characteristics together in one large table does constitute a truly random access key, the resulting table will be very large. `Characters` that are specific to only one species, will have only one positive cell, but many cells of negatives. This is a waste of space in terms of trying to put these keys to paper. Such a key might take up several pages in a book table. The same concept applies to computerized tables. Ideally in a software implementation the data input by the software should be split into multiple tables. Smaller groups of data can be managed and traversed more efficiently in memory than one large table. If the data are stored in multiple tables that make up a data *tree*, logical traversals can be made instead of random jumping from one table to another. By traversing the logical tree, search times are also decreased. The difference that these time and space savers make will not likely be noticed when using small groups of data, like Liverworts. On the other hand, with large sets of data, like for instance flowering plants is an enormous set of data, so these optimizations might mean the difference between the user having to wait two seconds to submit a character, or ten seconds.

The use of two types of tables such as was shown in Example 1 provides

an interesting solution to these time and space considerations. One table is given that distinguishes between genera. This allows general characteristics to narrow the search such that once the user has narrowed the search down to one genus, he/she can finish the search by looking at a much smaller table containing specific characteristics of each species in that genus. There is a limitation with this method. The user might notice a characteristic about a plant that is highly specific. He/she might believe that this characteristic would positively identify the plant if it could be found. The user would have to look at each table in the book until he/she finds that specific character and the species that exhibits it. This is clearly not ideal. One other issue with the tables in the books is that by definition of using separate tables, there is no way to organize the tables in an easily searchable manner.

It was by studying these two existing examples that we came up with the specifications for the random access software system we would implement.

Application Requirements

- It is important to be able to determine the identification of a plant down to its species, therefore the program must contain enough information to uniquely identify species.
- The program should be able to work for any set of plants (for instance, liverworts, flowering plants, mosses, trees). Some of these sets are small and some are enormous, so the method of data storage and look-up must conserve space and time. In addition, this means that the program must be general enough to accommodate any set of plants and still work correctly.

- Users cannot be expected to know every botanical term so there must be a glossary where they can look up unfamiliar terminology and view examples that illustrate the term (where possible).
- Users should be able to see how the character states they specify affect the number of species left in the search.
- Users should be able to view a list of the characters they specified previously as a record of their search.
- All characters should be present for the user to browse through. The characters should be organized in a fashion such that similar characters are grouped together, and characters that are mutually exclusive will cancel each other out.
- The interface should be easy to use and familiar to most users, where the users will be somewhat comfortable using computers and possibly the internet.
- The interface should provide visual cues to let the user know what action to take next and to show the user what characters have already been specified and which have not.
- Users should be able to pull up a list of all the genera and species contained in the entire data set if they choose.
- The user should not have to understand how the program stores and manipulates the data in order to use the program. In other words, the implementation should be screened from the user.

It was to these ends that we designed the program. The first step in the process was to design a preliminary user interface. Based on the requirements above, we made some rough sketches. Next we chose an operating system to design for and coding tools to use. This was the logical next decision because the limitations of the operating system and development tools would determine exactly how much of the interface would be implemented as designed and how

much would need to be altered. We chose to use Windows 98 as the operating system and Microsoft Visual C++/*MFC* as our coding environment.

CHAPTER IV

IMPLEMENTATION

Overall Architecture

When a user opens the program, characters and the plants that exhibit them are read from a text file into a hash table. These characters are displayed to the users in a tree. User interaction with the tree (e.g. submitting characters) thins out the hash table entries until only one entry is left. This final entry contains the unique identification of the user's plant. Figure 2 illustrates this process.

User Interface

The part of the program that the user sees is the "user interface." The interface of "Botania" is a Multiple Document Interface (MDI). "A Multiple Document Interface application can open many documents (typically files) at once. There is a Window menu and a Close item on the File menu"(Gregory 12). Using an MDI allowed us to place several windows next to each other in the application. These windows are the "`CharacterSelection`" window, the

"Glossary" window, and the "History" window. Each of these windows is based on a CFormView which inherits from CView. "A View object (which is just another kind of window) enables the user to see the data on-screen and to edit that data in a way that is appropriate to the application."(Gregory 115) In essence, the View contains the portion of the application that the user can see and interact with.

The Character Selection Window

The class `BotaniaView` implements the "`Character` Selection" window. This window is where the user will search through a list of characters and specify those that apply to his/her specimen. This `BotaniaView` class contains all the information needed to completely draw the "`Character` Selection" window and the interface *widgets* inside it. It also contains the information needed to react to user input by mapping actions to functions in the `BotaniaView` class. For instance, if the user clicks on an item in the tree that has not previously been selected and whose mode is "specifiable character", the character states for the user to select from will appear to the right of the tree. The text of the item's parent will appear as the question at the top and the character state choices appear as radio button items. Because there are not always the same number of choices under each heading, the program creates 8 radio buttons which can be shown and hidden as needed at each interval.

In the `BotaniaView` class, a CTreeCtrl is *declared* and *initialized.* CTreeCtrl is a tool provided by MFC that allows placement of a "Windows Explorer"-like tree in an application window. Once that tree is created, it is filled by

cycling through the `Characters` contained in `VisibleCharacterManager`. It begins by looking for elements that have a parent named "O". A parent of "O" indicates that there is no parent, and thus the question belongs at the root level of the tree. For each of these root level questions, the program *recursively* grabs a list of the children, those children's children and so on until a point is reached where one of the children has no children indicating the bottom of the tree has been reached. In this manner, the tree is filled with the information in `VisibleCharacterManager` and its `Characters`.

The Glossary Window

The "Glossary" window is intended as a visual aid. It has the ability to show the definitions of key botanical terms as well as illustrative images when available. This window is implemented as an embedded web page. This was done using a CHtmlView which is an Active X wrapper for Internet Explorer (IE) 4. We chose to use a web page in this window for a couple of reasons. First, we wanted the functionality available to view embedded images within the definitions. This is extremely important for terminology that is difficult to describe without pictures and diagrams. Associating definitions with illustrative images could allow the user to learn the terminology more quickly and therefore have an easier time knowing which character states apply to his/her plant. Second, often in botanical definitions, there are references to other technical botanical terms. We wanted the functionality available such that the user could see which words in the definition are also in the glossary. Then the user could simply click on the word and its definition would appear.

With the enormous number of people now connected to the internet, web pages are becoming a familiar interface for many users. Most users with internet access are familiar with clicking on blue underlined words in order to follow links. This fact may help make this section of the program more intuitive to many users. An embedded web page gives the added benefit of allowing different formatted image, e.g., GIF, JPEG, BMP. Using CHtmlView's HTML capabilities not only freed us from writing a parser and image viewer ourselves, but also opens the door for future versions of the program to use advanced content such as Java Applets or QuickTime VR animations. The main limitation of using an embedded web page is that the user must have IE4 installed on his/her computer in order to use this "Glossary" window. However, IE does come installed with Windows98 and NT (one of which is required to run this application). If the user does not have IE installed, it is free to download from Microsoft's web site.

A definition appears in the "Glossary" automatically when the user clicks on a question in the tree that contains a word that is "definable". Definability is determined by the key writer when the tree is written, and there is a *data member* in `Character` called `DefinableList` to hold these words. Each definition is in essence an html file, and the set of all definition files are stored in a "Glossary Data" folder inside the folder that contains the program.

The History Window

The "History" window is a visual record for the user. It records each character state that the user specifies and displays it as a text description. The implementation consists of a CFormView window that contains an edit box.

Each time the user specifies a character state, the "History" window recursively asks the `Character` the name of its parent, grandparent, and so on. Then it pieces the text descriptions of these ancestor `Characters` together into one long string showing the path in the tree that the user took to specify that character state. This window is very simple and is only there to provide a "memory".

The View Remaining Dialog

The "View Remaining" dialog bears a resemblance to the Glossary Window. This is because the "View Remaining" dialog contains an embedded web browser much like the "Glossary" window. However the "View Remaining" dialog is a modal dialog box rather than a CView. "A modal dialog is on top of all the other windows in the application: The user must deal with the dialog box and then close it before going on to other work"(Gregory 52). This dialog only appears when the user clicks the "View Remaining Species" button in the "`Character` Selection" window. This is in contrast to the "Glossary" window which is always visible to the user and can sit quietly in the background, or can be used at any time. The web page displayed in the "View Remaining" dialog has two frames. The frame on the left displays each genus that is still possible and the possible species within it in an outline format. Each species is a link that, when clicked, causes a page to appear in the right side frame with the detailed description of that species and any related images. The HTML files for each species detailed description are pre-created. The HTML file that appears in the left frame outlines the remaining species, and is created dynamically each time the user clicks the "View Remaining Species" button. The process to create this

page begins by creating a file called "leftside.html" which the program can write to. This file is created in the hard drive's "Temp" directory and is recreated each time a new search is begun. Once the file is created, the next step is to navigate the `DataSets` to look for genera and species that are still possible. It begins by looking for valid Level2 `DataSets` within the possibleList of the Level3 `DataSet`. For each of these valid Level2 `DataSets`, it creates the HTML tags for a heading of the genus name on the web page. Then it looks for valid Level1 `DataSetSpecies` within the possibleLists of the Level2 `DataSets`. For each of these valid Level1 `DataSetSpecies`, a link is created using standard HTML tags to a file of the species name plus the suffix ".html". These species description HTML files are stored in a directory called "SpeciesData" in the folder containing the program. Once the HTML file has been written to, it is closed by the program, and then loaded by the web browser. The file is closed before navigation to prevent any extraneous information being accidentally written to the file.

Design and Implementation

In essence, this application is a search engine. The user provides criteria in the form of character states and the program searches for species that match those criteria. The searching works on a similar principle to the two non-computerized examples mentioned above. When the program begins, all genera and species are possible. The user observes a list of characters and selects those that apply to his/her plant. The program takes each state and looks for genera and species that exhibit that character state. This process is called "specifying a character". It marks the species and genera that do exhibit the character state as

still possible, and it marks the species and genera that do not exhibit the character state as no longer possible. This idea of "specifying" is implemented in the *classes* `Character`, `VisibleCharacterManager`, `DataSet`, and `DataSetManager`.

Data Abstraction

An essential characteristic of this application is abstraction of information. The program sets a clear division between the way information is presented to the user, and the way information is stored in and used by the program. In other words, the amount and type of information the user needs to see in order to be able to use the program is different from the amount and type of information needed to make the application function. The user should only see a list of characters that he/she can select from. On the other hand, the program itself needs to know more detailed information in order to perform its "searching" function and display the information correctly to the user.

This abstraction of data is intended in part to make information as easy as possible for users to deal with, without taking away from the functionality of the program. One demonstration of this is seen in the way character choices appear to the user. For example, suppose the user sees a character titled "Color" with the state choices as "red," "blue," "green". The user might see this as three possible answers to the question "What is the color?" But, the program sees this same information as these three individual yes or no questions: "is the color red?" "is the color blue?" and "is the color green?"

Abstraction occurs with the idea of `Characters` versus `DataSets`. A

`Character` holds a single character state and is contained by the `VisibleCharacterManager`. The `VisibleCharacterManager` holds all of the information the application needs to display the characters to the user in an organized tree format. It also contains information to give the user feedback on which character states have already been specified. A `DataSet` contains information about characters that occur in a single genus or species. A `DataSetManager` contains `DataSets` and is primarily responsible for the process of "specifying characters".

The VisibleCharacterManager and its Characters

A `VisibleCharacterManager` is a container and manipulator of `Characters`. Its tasks include opening the input file, reading information from the file and creating and filling `Character` *objects* with this information, and taking part in the process of "specifying" `Characters`. Each `Character` contains an identification number, the text that the user will see, the tree relationships (parent and children) so that the tree knows how to draw itself correctly with each item in its correct place, a list of the words present in the text that are definable by the glossary, a valid bit, and a mode bit (see Figure 3).

The identification (ID) number for each `Character` is unique. When the `Character` is placed in the tree, its ID is stored in the MFC *CTreeCtrl*. When events are generated by the tree (such as mouse clicks, double clicks, or selections), we can use this ID to quickly reference the selected character and thereafter its parent and siblings. The ID is also used by `DataSets` as a way of knowing which characters it is able to specify. This will be explained further later.

The text portion of `Character` is what the user sees when he/she opens the program and browses the characters. The text is simply a description of a character state. The text is only for the user, as the program uses the identification number for its manipulations. It does this because a number is smaller and faster to compare for equality than a piece of text.

The list of definable words holds each word in the text of the character that can be found in the glossary. This information is held in the `Character` class so that when the user clicks on a character in the tree all definable words in the text of that character automatically appear in the glossary window.

The mode string defines the `Character` type. There are two different types of `Characters`; "specifiable characters," and "headings". A mode of "specifiable character" means the user can select and specify the character in the "`Character` Selection" window. This means when the user clicks on the tree item of the desired character, it will appear to the right of the tree as a *radio button*. The user now has the option to select it. A `Character` with mode "specifiable character" can be applied to a `DataSet` to rule out species. The second mode is "heading". A `Character` with mode "heading" is not a specifiable character, but is rather a space holder. It is used to group the "specifiable characters" in the tree into characters that are related (see Figure 4). A "heading" cannot be specified by the user and it cannot be applied to a `DataSet`.

In order for the program to draw the tree on the screen, it needs to know where to draw each `Character`. This is accomplished by storing tree relationships in each `Character`. The tree relationships come in the form of identification numbers as well as *pointers*. Looking at a tree, one can see that each element has one element above it, and zero to many elements below it. Each `Character` knows the ID of the item that contains it. This item is its parent. The elements

contained by a `Character` are its children. Each `Character` knows the ID numbers of its children. Each `Character` is also aware of its siblings, which are Characters that have the same parent. After this information is read in from the input file, pointers are created from each `Character` to its parent and children to facilitate navigation between elements at different levels in the tree.

The valid bit is a true/false *flag* that is true when a `Character` has not yet been specified and none of its mutually exclusive siblings have been specified either. It is false when the `Character` has already been specified or one of its siblings has been specified instead.

The DataSetManager and its Datasets

Data groups contain all the information about a given group of genera, a single genus, or a single species depending on the type of data group. All data groups are managed by a `DataSetManager`. (This is analogous to how the `VisibleCharacterManager` manages all `Characters`.) There are two types of data groups: `DataSet` and `DataSetSpecies`. A `DataSet` is used to represent either a single genus, groups of genera or other higher *taxonomic* classifications. It is essentially equivalent to the tables in Example 1 mentioned above. A `DataSet` knows its name (ie: name of genus) and it has a valid bit, a possible list, and a character table (see Figure 5).

The valid bit is a true/false flag that represents whether or not that `DataSet` has been ruled out as a possible match to the search. A `DataSet` is ruled out when the user specifies a characteristic which that `DataSet` does not exhibit.

Each DataSet has a set of possible sub-level DataSets that are stored in a *hash table* called the PossibleList. For instance, suppose GenusA has 3 species. The PossibleList of GenusA would be a list of pointers to these three species. In other words, the PossibleList is a pathway to the data groups below it (see Figure 6). One could think of DataSets as having parents and children much like the elements in the VisibleCharacterManager. A genus is the parent of a group of species in the sense that the genus encapsulates those species. The PossibleList is used as a way to navigate through DataSets beginning at one level and moving down to its children, grandchildren, etc... The need for this functionality will be discussed further in the section "Specifying Characters".

Each DataSet has a group of characters that it can specify, as well as a list of the sub-level DataSets contained by it that exhibit those character states. For instance, if a particular genus has leaves, then it may be useful to specify characters about leaf characters in order to distinguish between the species within that genus. However if the plant does not have leaves, it would not make sense to be able to specify characters about leaves as its DataSets would not exhibit that character. In addition, different genera contain different species, so they will have different lists of species that exhibit the exact same character. For this reason, each DataSet must know which Characters it is able to specify, and how to specify them. This information is contained in the CharacterTable which is implemented as a hash table of *lists*. The hash table contains the ID numbers of Characters the genus can specify. For each of these Characters, there is a list that contains the names of sub-level DataSets within the current DataSet that exhibit that character state. This is the YesList. Every element in each of the YesLists will appear once in the PossibleList, and therefore

each of these lists is a *subset* of the `PossibleList`. The purpose of the `CharacterTable` is essentially to answer this question: "Which `Characters` are useful in distinguishing between my sub-level `DataSets`?" Similarly, the purpose of the `YesList` is essentially to answer this question: "Of all the possible sub-level `DataSets` contained within this `DataSet`, which exhibit this character state?"

`DataSetSpecies` are similar to `DataSets`, but because individual species are the most specific classification that can be determined by this program, they do not contain `PossibleLists` or `CharacterTables`. Individual species have no need to keep a list of possible sub-level `DataSets`, because they have none. `DataSetSpecies` do contain a valid bit, because it is important to know whether a species has been ruled out of the search.

There are two ways to retrieve information from `DataSets` contained by `DataSetManager`. The first way is to navigate down the logical tree from the `DataSet` at the root of the tree by way of `PossibleLists`. This is represented by thin black lines with arrows in Figure 6. This method of navigation is used in the process of "specifying a question" and in gathering the data to display in the "View Remaining" window. The second way to retrieve `DataSet` information is by searching for the name of the `DataSet` in its "level map". This method of navigation is illustrated using thick blue lines with arrows. The "level maps" are hash tables that provide an alternative way to access `DataSets` or `DataSetSpecies` at a given level in the taxonomic hierarchy. These hash tables essentially constitute a "back door" to the data. This is only used when information about an entire Level is needed. This is used when the program starts up and reads the `DataSet` information in from files. It simply provides a fast easy way to group the data.

Specifying Characters

When a new search begins, all species and genera are possible. As character states are specified, the `DataSetManager` applies them to the possible `DataSets`. An example of this process is as follows.

1. The user selects a character state and submits it in the "`Character` Selection" window. This ID number of the character is received by `BotaniaView::OnSubmit()` which is called when the user clicks the submit button.
2. `BotaniaView::OnSubmit()` passes this ID number to `VisibleCharacterManager::SpecifyCharacter()`.
3. `VisibleCharacterManager` grabs the `Character` object associated with this ID number. Then it calls `VisibleCharacterManager::FillCharOrderStack()` who creates a *stack* by asking that `Character` the name of its parent, grandparent, great-grandparent, and so on until the top of the tree is reached (in other words, a question with no parent). This is called the `CharOrderStack`. This `CharOrderStack` does not contain `Characters`, but rather the ID number associated with each `Character`. Suppose the user specifies "red rhizoids" but has not yet specified "Plant has rhizoids". Both of these are `Characters` that are specifiable. By specifying red rhizoids, it is implied that the plant has rhizoids. In order to make sure the plants who have no rhizoids are thinned out of the search, we must explicitly specify this `Character` in the program (See Figure 7). This is done "quietly" because the user does not need to be aware of this process. The `CharOrderStack` ensures that when a character state is specified, each character that is implied by it is also specified.

4. `VisibleCharacterManager::ApplyCharOrderStack()` is then called to pop the top off the stack one ID number at a time and pass that ID number to `DataSetManager::SpecifyCharacter()`.

5. `DataSetManager::SpecifyCharacter()` receives the ID number of the `Character` to be specified.

6. Begin with the `Level3Map`. There should be only one `DataSet` in this table. This `DataSet` contains characteristics that distinguish between genera. The `PossibleList` of this `DataSet` points to all genera. The `CharacterTable` of this `DataSet` contains the ID numbers of `Characters` that are useful in distinguishing between genera. For each ID number, the `CharacterTable` points to the `YesList` of the names of genera that exhibit the `Character` represented by that number. With the ID number in hand, `DataSetManager` looks at this one Dataset from the `Level3Map`.

7. If the ID number is present in the `CharacterTable` of this `DataSet`, then this `DataSet` is able to specify this character. Extract the `YesList` from the `CharacterTable` for that particular `Character`'s ID.

Apply this `YesList` to the `PossibleList`. This is done as follows. For each `DataSet` pointed to by the `PossibleList` look for only those `DataSets` that are still valid. (`DataSets` that are invalid by the time this character is specified have already been ruled out as matches to the search. Once a `DataSet` is ruled out for a search, it cannot become valid again.) For each valid `DataSet`, ask for its name and look for that name in the `YesList`.

If the `DataSet's` name is present in the `YesList`, this means that the `DataSet` was possible before this character was specified and does exhibit the currently specified character state, so it remains possible.

However, if the `DataSet's` name is not present in the `YesList`, then we

have specified a character that is no longer exhibited by this `DataSet`. This means that we had a possible `DataSet`, but it did not exhibit the currently specified characteristic and therefore is no longer possible. In this case, we change its valid bit from valid to invalid. We also set any sub-level `DataSets` to invalid. For instance if a genus is not a possible solution to the search, then any of its species are also not possible.

8. Once the above process has been performed on the one `DataSet` in the `Level3Map`, it is then performed again on each valid element pointed to by the `PossibleList` of this `DataSet`. Each element at this level (pointed to also by the `Level2Map`) is a genus. Each of these genus `DataSets` contains characteristics that can distinguish between the species within that genus.

CHAPTER IV

CONCLUSIONS AND FUTURE WORK

By using the processing power of computers, random access keys have become a viable and accessible means for plant identification. The method we have chosen allows the user to focus on the identification process rather than the computation details of the implementation. Computationally, it conserves on the space required to hold information by using multiple tables.

In the future, we will create versions of this program for the Macintosh operating system and perhaps the Palm operating system for use on hand-held Palm Pilots. Many features will be added as well. A new window called the "Priority Window" will be aimed at helping users who have not yet found a unique identification, but do not know what character to specify next. This window will tell the user which character specification will divide the number of possibilities in half. In addition, multiple undo features will be added.

APPENDIX A

FIGURES

Comparison of species of Nardia and Gyrothyra	Nardia					
	1. obovata	2. rubra	3. scalaris	4. geoscyphe	5. breidleri	Gyrothyra underwoodiana P. 65
Valves of the capsule spirally coiled.....	–	–	–	–	–	+
Underleaves when present 2-toothed....	–	×	–	–	+	+
Diameter of leaf cells in mu.....	32-40	20-50	36-50	20-40	10-16	25-70
Leaf apex *r*ounded, *e*marginate or *b*ilobed.....	r	r	r	e	b	r
Rhizoids *p*urplish, *r*eddish, *c*olorless or *y*ellowish.....	p-r	c	c	c	p	c-y
Trigones bulging into the cells in old leaves.....	–	+	+	+	–	+
Trigones *n*one, *s*mall or *l*arge.....	s	l	l	sl	n	l
Marginal row of leaf cells *l*arger, or *s*ame as others.....	s	l	s	s	s	l
Underleaves *n*umerous, fairly *c*ommon, only near *t*ip, or *w*anting.....	t	w	n	t	c	c
Underleaves *l*arge, *s*mall, *m*inute, or *n*one	s-n	n	l	m	s	l
Leaf cuticle *s*mooth, *p*unctate, *s*triate...	pr	p	m	m	m	m
Perianth *i*mmersed, about *e*ven with *b*racts, *e*xserted.....	e	x	i	i	i	i-e
Perianth, *c*renulate, *d*enticulate, *l*aciniate, *l*acerate.....	it	t-e	c-t	c	c	c
Diameter of spores in mu	16-21	13-16	15-17	14-16	9-12	±12
*D*ioicous or *p*aroicous.....	p	d	d	p	d	d
Length of plants in mm.....	1–30	3–12	2–30	1–10	2-3	10-30
Oil bodies present.....		+	+	±	–	
Length of mature elaters in mu.....		90–130			63-84	210-240

Figure 1. Illustration of a page from Clark and Frye's "Liverworts of the Northwest".

Plant Characters

Character	Plants that exhibit the character
has more than 10 petals per flower	rose, azalea
has less than 10 petals per flower	tulip
has thorns	rose
no thorns	azalea, tulip

Possible Plants

Initial:

tulip	rose	azalea

User selects that his/her plant has more than 10 petals per flower:

~~tulip~~	rose	azalea

User selects that his/her plant has thorns:

~~tulip~~	rose	~~azalea~~

The only plant left is a rose, therefore the user has identified a rose.

Figure 2. Simple example of Botania's overall architecture.

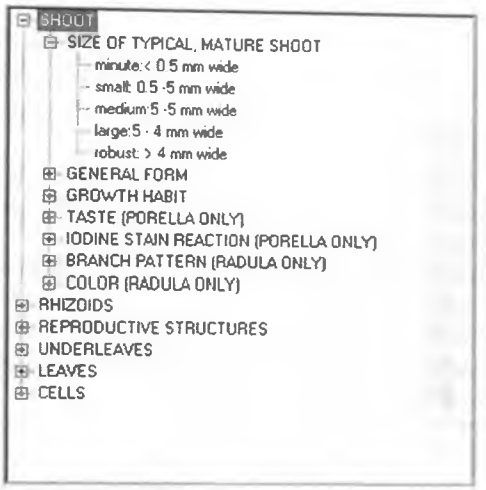

Figure 4. Illustration of mode. The mode of "SHOOT" is "heading". All headings are typed in capital letters. Similarly "SIZE OF TYPICAL MATURE SHOOT" is also a heading. It would not make sense to ask a DataSet "What is the size of your typical mature shoot?" because in this program character states are binary. However, "minute" is a specifiable character because it does make sense to ask a DataSet, "Is your typical mature shoot minute in size?" This question is binary because its answer is either yes or no.

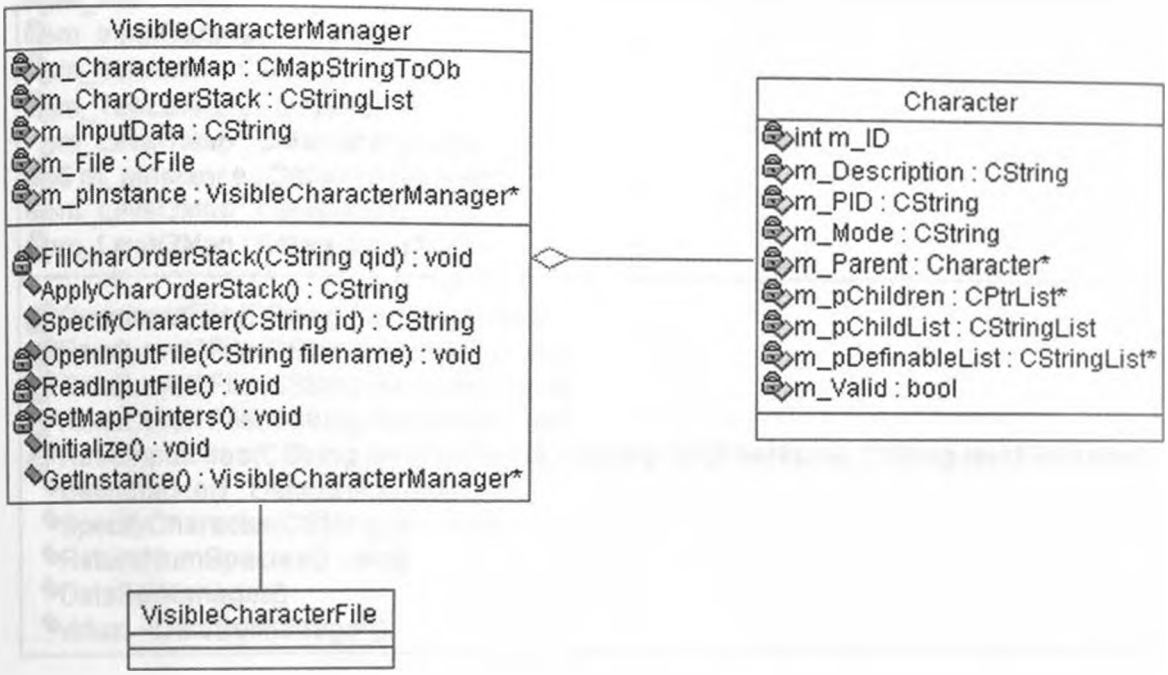

Figure3. UML Class Diagram of VisibleCharacterManager and Character.

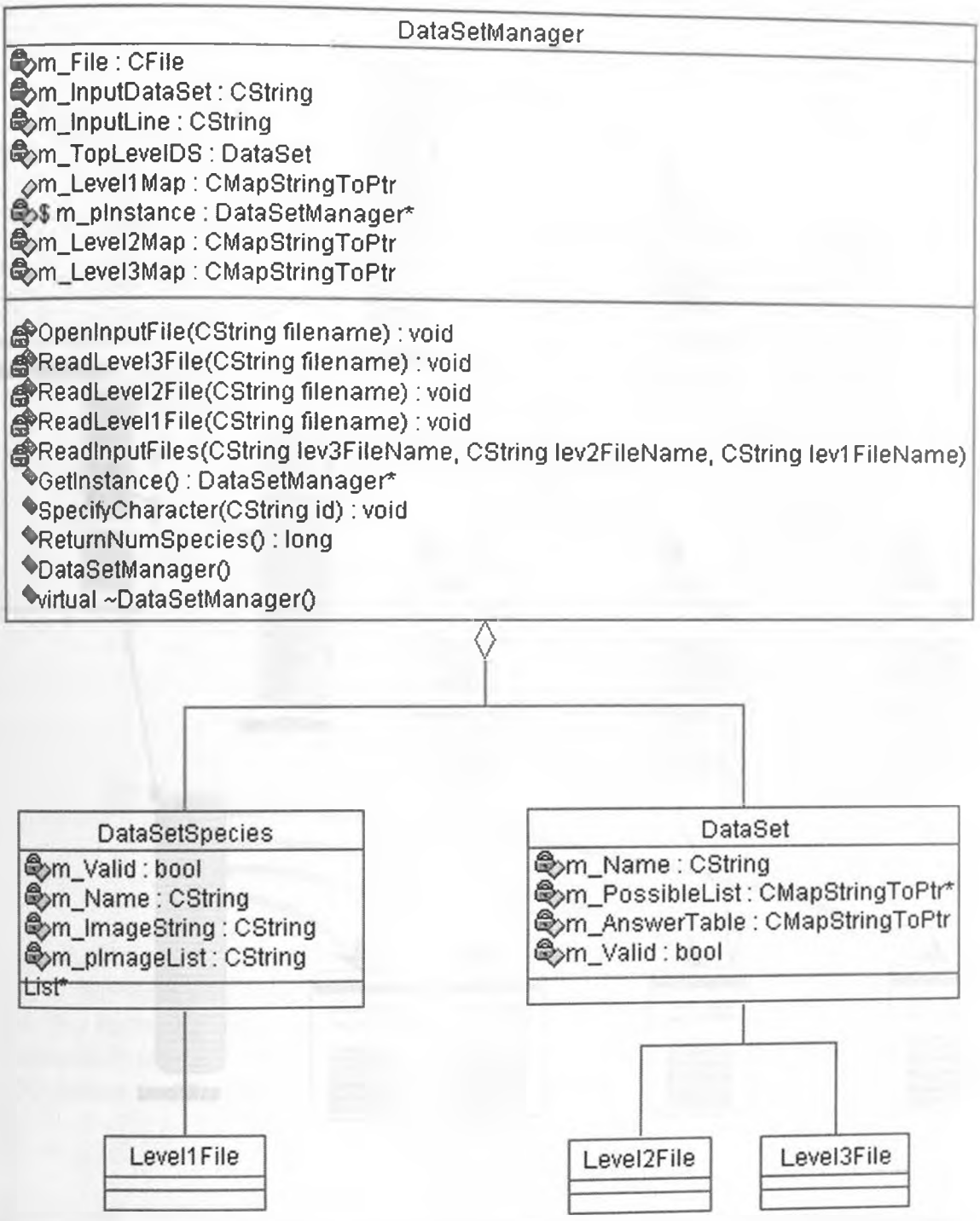

Figure 5. UML class diagram of DataSetManager, DataSet, and DataSetSpecies.

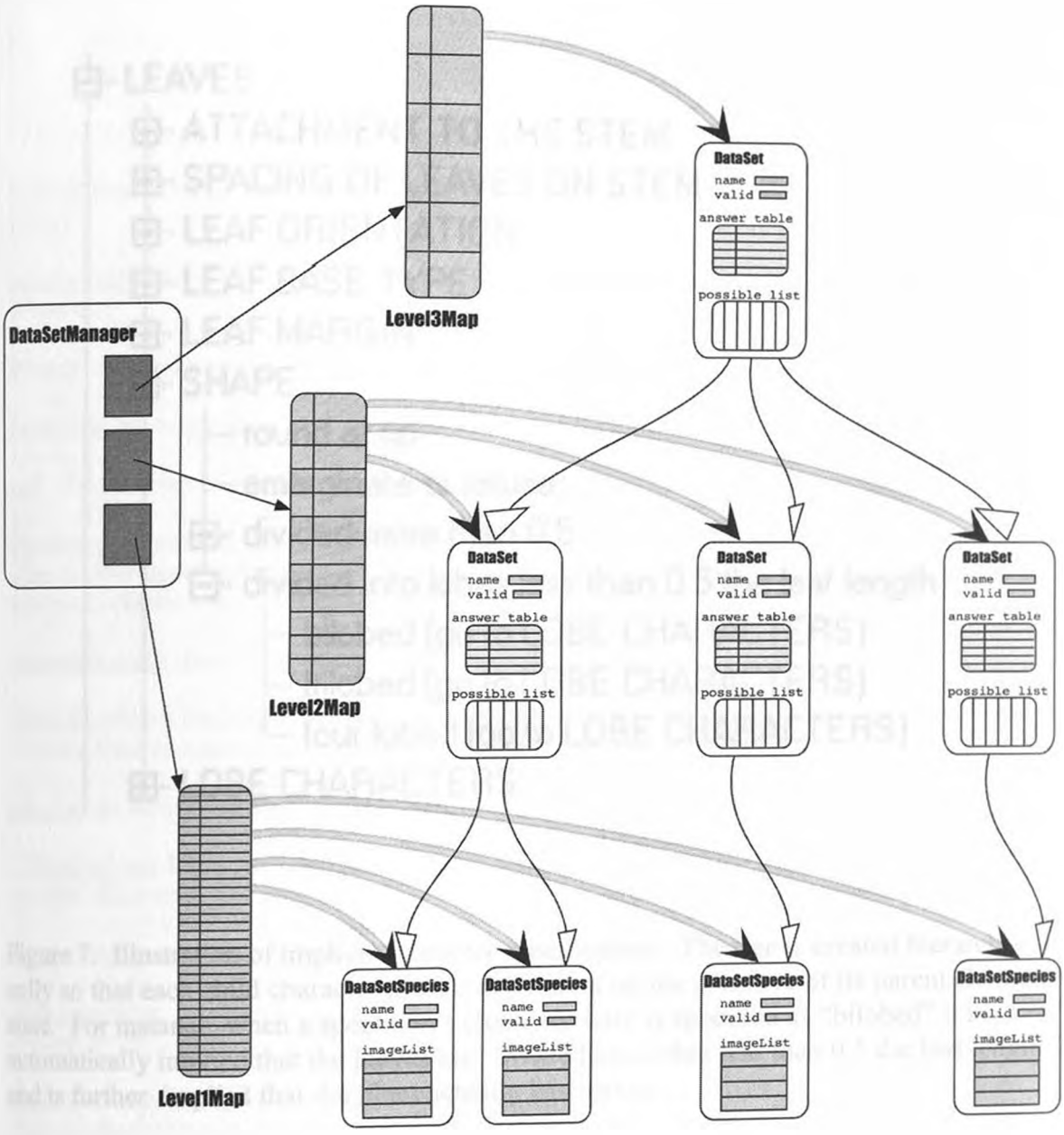

Figure 6. Illustration of DataSet tree relationships and navigation. Thick blue lines with black arrows indicate one means of traversing the information while thin black lines with hollow arrows illustrate another means of traversing the data. This figure illustrates the tree data structure that the DataSets constitute. The DataSet pointed to by Level3Map contains each element in the Level2Map. This can be seen as the thin black lines pointing from the parent DataSet to its children.

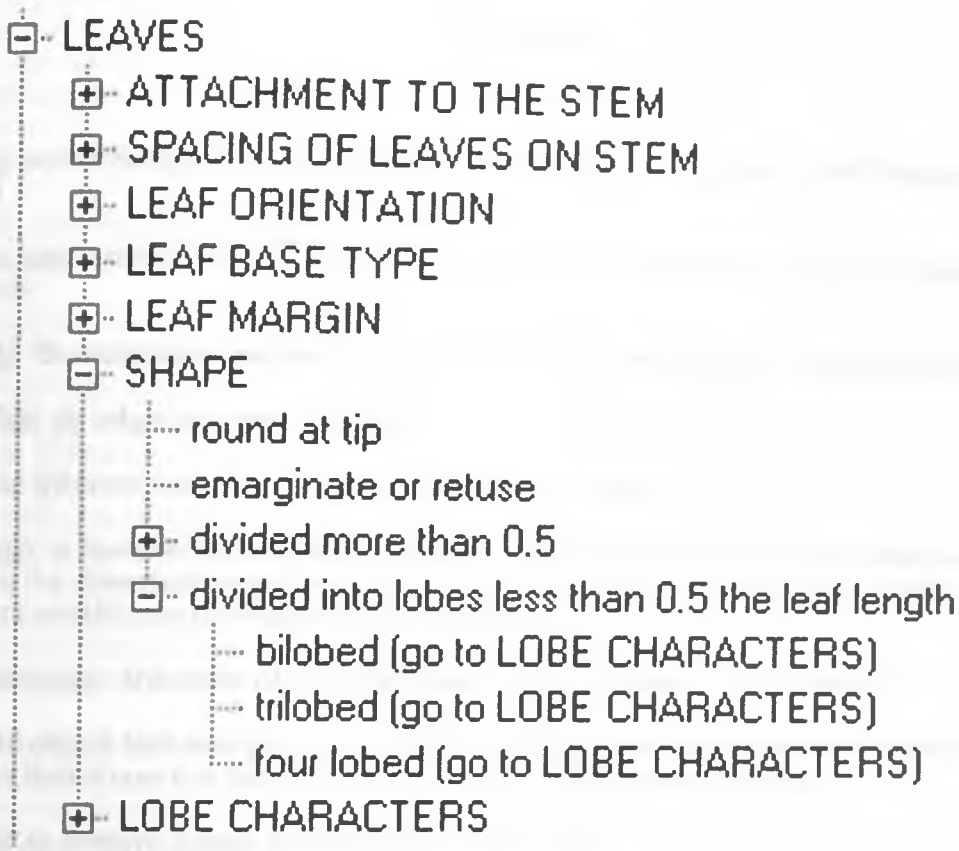

Figure 7. Illustration of implicit character specification. The tree is created hierarchically so that each child character state is dependent on the presence of its parent character state. For instance, when a specimen's character state is specified as "bilobed" it is automatically implied that the leaves are "divided into lobes less than 0.5 the leaf length" and is further implied that the plant actually has leaves.

APPENDIX B

GLOSSARY

binary something made of or based on two things or parts (WWWebster Dictionary)

binary tree a tree in which each node has at most two successors or child nodes (Howe)

botany the science of plants, their classification and study (GardenWeb)

branches an edge in a tree (Howe)

cell the intersection of a row and column in a table

character a feature, trait, habit, function, structure, or visual mark that contributes to the description and or diagnosis of a species (or any other rank in a taxonomic classification) (Wagner, pers. comn.)

character state the state of the character; often "present" or "absent"

class in object technology, a user-defined data type that defines a collection of objects that share the same characteristics (TechEncyclopedia)

couplet in botany, a pair of opposite characters

CTreeCtrl an MFC control that contains the functionality of a "Windows Explorer"-like tree

data member a variable associated with a particular class

data structure a software-based container where data is stored and can be accessed from

declare the process of associating a name with a data object, a function, or a data type so that the programmer can refer to that item by name (Dale, Weems, and Headington 56)

dichotomous a key for the identification of organisms based on a series of choices between two [mutually exclusive] alternative characters (WWWebster Dictionary)

genus a group of closely related species. The taxonomic category ranking above a species and below a family. (GardenWeb)

flag a variable or quantity that can take on one of two values; a bit, particularly one that is used to indicate one of two outcomes or is used to control which of two things is to be done. (Howe)

hash table a data structure for storing items where each item has a key, which is looked up in the hash table and a value that is returned. An example of a hash table would be a dictionary where the key is the word to look up and the value is the definition.

initialize the process of assigning an initial value to an element of a C++ program

key an arrangement of the salient characters of a group of plants or animals or of taxa designed to facilitate identification (WWWebster Dictionary)

liverwort a division (phylum) of non-vascular, spore producing plants directly descended from the earliest land plants, with a free-living gametophyte and a sporophyte dependent on the gametophyte; the capsule opening by (usually) four slits and containing elaters mixed in with the spores. (Wagner, pers. comn.)

list a data structure holding many values, possibly of different types, which is usually accessed sequentially, working from the head to the end of the tail (Howe)

MFC (Microsoft Foundation Classes) a C++ framework provided by Microsoft to access functions in the Windows operating system.

object an instance of a class

pointer pointers are variables that tell where to find something else; that is, pointers contain the addresses or locations of other variables.

radio buttons a series of on-screen buttons that allow only one selection. If a button is currently selected, it will de-select when another button is selected. This type of selection is used whenever there is only one choice out of several. Choosing one deselects the others. Pressing a preset station button on old radios pushes out the others. (TechEncylcopedia)

recursive in programming, the ability of a subroutine or program module to call itself. It is helpful for writing routines that solve problems by repeatedly processing the output of the same process. (TechEncylcopedia)

root an element of a tree that has no parents but typically does have children.

species a category in plant classification, the rank below genus, containing closely related, very similar individual plants. Species are ordinarily grouped to form genera. The following are examples of species within a genus (group). Sequoia (Redwood tree) is the genus name and the species (or varieties) that belong to that group are sempervirens, gracilis, pendula, etc. These are written along with Sequoia. (E.g. Sequoia sempervirens is one species, Sequoia gracilis is another, and so on.) (GardenWeb)

stack a data structure for storing items which are to be accessed in last-in first-out order. The operations on a stack are to create a new stack, to "push" a new item onto the top of a stack and to "pop" the top item off. (Howe)

static standing or fixed in one place; characterized by a lack of movement, animation, or progression; showing little change (WWWebster Dictionary)

storage mechanism a means for holding data within a program. Common examples include databases, hash tables, stacks, lists, and arrays.

string In programming, a contiguous set of alphanumeric characters that does not contain numbers used for calculations. Names, addresses and error messages are examples of strings (TechEncylcopedia)

subset a set each of whose elements is an element of an inclusive set (WWWebster Dictionary)

taxon a group of genetically similar organisms that are classified together as a species, genus, family, etc. (pl. Taxa) (GardenWeb)

taxonomy the classification of organisms based on genetic similarities (GardenWeb)

tree a graph data structure wherein there is only one route between any pair of nodes, and there is a notion of "toward top of the tree" (i.e. the root node), and its opposite direction, toward the leaves. A tree with n nodes has n-1 edges. (Howe)

user interface the combination of menus, screen design, keyboard commands, command language and online help, which creates the way a user interacts with a computer. If input devices other than a keyboard and mouse are required, this is also included. (TechEncylcopedia)

widget in graphical user interfaces, a combination of a graphic symbol and some program code to perform a specific function. E.g. a scroll-bar or button. Windowing systems usually provide widget libraries containing commonly used widgets drawn in a certain style and with consistent behavior. (Howe)

WORKS CITED

Anonymous. "Garden Web" Authors Links and Info. The Virtual Mirror, Inc. Internet. Available: http://www.gardenweb.com/glossary (1999)

Anonymous. "TechEncylopedia" Authors Links and Info. Internet. Available: http://www.techweb.com/encyclopedia (1999)

Anonymous. "WWWebster Dictionary" Authors Links and Info. Merriam-Webster, Incorporated. Internet. Available: http://www.m-w.com (1999)

Clark, L. and T.C. Frye. The Liverworts of the Northwest. Publications of the Puget Sound Biological Station Vol. 6, 1928

Dale, Weems, and Headington. Programming and Problem Solving with C++. Boston: Jones and Bartlett Publishers, 1997.

Dattatri, Kayshav. C++ Effective Object-Oriented Software Construction. New Jersey: Prentice Hall PTR, 1997.

Davis, Stephen. C++ For Dummies. Foster City: IDG Books, 1994.

Gregory, Kate. Using Visual C++ 5 Special Edition. Indianapolis: Que Corp., 1997.

Horton, Ivor. Ivor Horton's Introduction to Visual C++ 6.0 Standard Edition. Wrox Press Ltd, 1998.

Howe, Denis. "The Free On-Line Dictionary of Computing" Authors Links and Info. Internet. Available: http://wombat.doc.ic.ac.uk (1999)

Oualline, Steve. Practical C++ Programming. Cambrige: O'Reilly & Associates, 1997.

Weber, William. Random Access Key to the Genera of Colorado Mosses. Boulder, University of Colorado, 1972

Weiss, Mark. Algorithms, Datastructures and Problem Solving with C++. Reading, Mass: Addison-Wesley, 1996.