Modelling of a temporal object-oriented geographic information system

David Edward Disalvio
University of Nevada, Las Vegas
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
Modelling of a
Temporal Object-Oriented
Geographic Information System

by

David E. DiSalvio

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
December 1994
The thesis of David E. DiSalvio, for the degree of Master of Science in Computer Science, is approved.

Chairperson, Kia Makki, Ph.D.

Examiner Committee Member, John Minor, Ph.D.

Examiner Committee Member, Yonina Cooper, Ph.D.

Graduate Faculty Representative, Ebrahim Salehi, Ph.D.

Interim Dean of Graduate College, Cheryl Bowles, Ed.D.

University of Nevada, Las Vegas
December 1994
ABSTRACT

Much research has been performed in recent years in object-oriented technologies. First, object-oriented programming languages were developed and more recently object-oriented databases. Geographic information systems have also been the subject of research during this time. While many models have been developed that add the concept of time to an object-oriented database, most of this work has been concentrated on efficient temporal queries. We feel that another important use of temporal databases is the ability to sequence through historical data efficiently. We will add this temporal feature to an object-oriented geographical information system.
TABLE OF CONTENTS

ABSTRACT .................................................. iii

LIST OF FIGURES ........................................... v

ACKNOWLEDGEMENTS ........................................ vi

CHAPTER 1 INTRODUCTION ................................. 1
  Previous Work ........................................... 2
  Database Evolution .................................... 3
  Temporal Data Structures ............................ 6
  The Object-Oriented Model ........................... 8
  Evolution of Geographic Information Systems ... 13

CHAPTER 2 DEFINING THE MODEL ......................... 21
  Defining the Object-Oriented Model ................. 22
  Defining the Temporal Object-Oriented Model ...... 26
  Addressing the Spacial Geographic
  Information System Issues .......................... 37

CHAPTER 3 MODELLING A TEMPORAL OBJECT-ORIENTED
  GEOGRAPHIC INFORMATION SYSTEM ............... 41

CHAPTER 4 DEVELOPMENT OF A PROTOTYPE ............ 48
  Prototype Specifics .................................. 48
  Implementation Issues .............................. 50
  Permanent Storage Medium ........................... 51
  Other Implementation Issues ....................... 52

CHAPTER 5 CONCLUSIONS AND REMARKS ............... 54

Appendix I PROTOTYPE SOURCE CODE .................. 69
Appendix II INIT. DAT .................................. 87

References ............................................... 92
LIST OF FIGURES

Figure 1 B+-Tree Leaf Node .................................. 56
Figure 2 B+-Tree .............................................. 57
Figure 3 B+-Tree Child Nodes ................................. 58
Figure 4 A Country Object ..................................... 59
Figure 5 A Country Object with Time ......................... 60
Figure 6 Attribute Tree ......................................... 61
Figure 7 Partitioned B+-tree buckets ......................... 62
Figure 8 A Location Object ..................................... 63
Figure 9 An Any Object ......................................... 64
Figure 10 Sample C++ Classes ................................. 65
Figure 11 Attrib_Tree C++ Class .............................. 66
Figure 12 Class Beacon C++ Code ............................. 67
Figure 13 Object of Type Beacon .............................. 68
I would like to express my appreciation and thanks to Dr. Kia Makki for offering his guidance and assistance in all aspects of this thesis. I would also like to thank Dr. Niki Pissinou for her review and critique of this work.

Special thanks to my family and friends who have given their support to my continued education.

Finally, I would like to thank the examining committee, Dr. John Minor, Dr. Yonina Cooper, and Dr. Ebrahim Salehi, for their time spent reviewing this thesis.
CHAPTER 1

INTRODUCTION

Object-oriented modeling, temporal database technology, and geographic information systems have been areas of widespread research in recent years. While numerous commercial products are available in the geographic information system area, object-oriented systems and temporal databases have yet to find widespread industry standards. It is the combination of these three areas where we feel there has been a lack of research. The importance of developing a temporal object-oriented geographic information system is stressed by Pissinou, Makki, and Park (1993). It is to this extent that where we will concentrate our efforts.

One area of concern is that most temporal models are developed around the historical query. While this is clearly important, we feel that there are other desirable uses for temporal databases. Sequencing through the historical data in an efficient manner is one feature we feel is important, especially when taken in context of a geographic information
system. For the purpose of this thesis, we will assume that the geographic information system we are developing is graphical in nature, and as such fast temporal retrieval will be critical in order for the temporal graphical display to meet the users expectations.

The remainder of this chapter will review some recent research in the areas of interest. Chapter 2 will give a brief overview of our model and the data structures involved. Chapter 3 will present a basic example of a temporal object-oriented geographic information system and present actual C++ code used to implement the data structures. In chapter 4 we will implement a simple working temporal object-oriented geographic information system. We will make our conclusions and final remarks in chapter 5.

Previous Work

It is essential to study the previous work relating to temporal object-oriented geographic information systems. We will break this historical overview into four sections. The first will deal with the evolution of databases. The second will review temporal data structures. The third will briefly describe the object-oriented model. The last will expound on the research performed on modern geographic information systems.
Database Evolution

Databases are an integral component to practically every major computer application. While computers are powerful in their own right, they would be considered useless without the ability to store, organize, and retrieve data. As computers have evolved from large behemoths with memory measured in kilobytes to small boxes containing many megabytes of power and ever increasing megahertz of speed, database technology has advanced at an almost equal pace.

Magnetic tapes, developed in 1945, were the first medium to allow searching (Elmasri & Navathe, 1989). Previous methods of storing data were limited to punch cards and paper tapes. Major improvements in database technology followed with the Hierarchical model, developed by International Business Machines in the late 1960's, the relational model, currently the most widespread database model, and more recently the object-oriented model.

The problem with early databases, and many products on the market today, is that they were one dimensional in the area of time. The fact that a database is only capable of storing the most recent occurrence of any particular data value, caused these products to be labeled snapshot databases. These databases literally contain a snapshot of the world at any one instant in time. If a data value changed, the old data was simply overwritten by the new
value. Unless backups were kept of the database, no historical queries were possible. A natural consequence of hardware becoming more powerful is that computer users demand higher performance from their software. This has led to the current interest in temporal databases. Temporal databases do not overwrite old data, but store it for future use. This allows the users to make historical queries to the database, make time line projections, and access a variety of other temporal functions.

The underlying model for database technology has taken several forms. The most prevalent in today's market is the relational model. Most approaches to temporal modelling have dealt with the relational database model. One of the earliest attempts to add time to the relational model was by Bubenko (1976). Bubenko simply added a field to each tuple relation, representing temporal validity. This method became known as tuple versioning. A significant problem with this method is the large degree of data redundancy that is required, since a complete copy of each record is need for each distinct version.

The heavy storage requirements for the tuple versioning method led to the development of the attribute versioning model. The attribute versioning model was formalized by Clifford and Tansel (1985). Attribute versioning significantly reduced the need for redundant data, by only saving the data which has changed since the last update.
Further research into attribute versioning was performed by Gadia and Yueng (1988). Even though attribute versioning reduces storage requirements, temporal database researchers still make use of methods similar to tuple versioning (Navathe & Ahmed, 1989; Snodgrass & Ahn, 1985).

The issue of time is even more complicated than whether to use tuple versioning or attribute versioning. In fact, there are many types of time to consider when developing a temporal database. The most obvious time value of interest is the time at which a data value becomes valid. This time is often referred to as world time. Another important time is the time the specific attribute was updated in the database. Because most systems are not real time systems, there are many administrative reasons for wanting to know when the database was updated. For obvious reasons this time is often referred to as update time. Usually the update time occurs after the world time, along the time line; but in some circumstances it may be possible to update the database, before the time a change is to take effect. World and update time are not the only time references that can apply to a temporal database.

Extensive research has been undertaken to determine which type of time is appropriate. Snodgrass points out that the concept of measured time is a manmade phenomenon. There are many methods of calculating time such as ephemeris, mean solar days, UTC, and TDT. The accuracy of these methods
depends on where one stands on the time line. While these methods point out the workings of the universe, they appear to be overkill for all but a small percentage of database applications.

Despite substantial activity in the temporal database area, there are no widely used commercial systems. Currently many terms are used to represent a specific meaning, thus causing confusion. At least two recent research efforts attempt to construct a common glossary of terminology on temporal databases (Jensen et al., 1994; Pissinou et al., 1994).

Temporal Data Structures

It is reasonable to expect that many of the data structures used to implement snapshot databases would be ineffective for use in a temporal database. One data structure, which appears to be very useful in the temporal model is the monotonic B+-tree (Elmasri, Jaseemuddin, & Kouramajian, 1992). The monotonic B+-tree, according to its creators, is an access structure suitable for implementing a time index for append-only temporal databases. As the name suggests, the monotonic B+-tree is an extension to the B+-tree, which is a widespread indexing scheme used to organize sorted data. This type of data structure makes sense for use
in a temporal database. In order to be of any practical use, temporal data must be sorted in chronological order, which directly suggests that a B+-tree, or its derivative, would be beneficial to a temporal database.

In order to fully understand the effects of the monotonic B+-tree it is necessary to review the B+-tree data structure. A B+-tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length. Each node in the tree has between \([n/2]\) and \(n\) children, where \(n\) is fixed for any particular tree (Korth & Silberschatz, 1986). Figure 1 shows a typical leaf node. The variable \(K\) represents key values, while the variable \(P\) represents pointers to buckets containing data associated with the corresponding key. Insertion and deletion routines maintain the balanced nature of the tree by allowing the node sizes to vary according to the predefined limits. Nodes are split or joined depending on the current status and function. A main benefit of the B+-tree derives from the use of the buckets in the leaf nodes. The most prevalent use of the buckets is to point to a linked list of records, which are related to the corresponding key value. A complete B+-tree is depicted in Figure 2.

The traditional B+-tree does not lend itself directly to a temporal database, however. Traditionally, the buckets contain information regarding data relating to the key value. If one assumes the key value represents time and the
buckets contain a list of objects in existence at that time, a large amount of data redundancy would be present in the database. For this reason it is beneficial to make use of the incremental bucket method. Children nodes for both methods are shown in Figure 3. Following the rules for the incremental method, only the first bucket for each child contains a list of all objects in existence at the applicable time. The remaining buckets in the node contain only objects which have been added or deleted, since the previous time instance. This method saves a tremendous amount of space, but has a corresponding time tradeoff. The tradeoff is that all time queries, which do not match a first key value in a child, will have to progress from the first bucket in the child node to the targeted time in order to determine if the object was indeed in existence at the queried time.

The monotonic B+-tree formalizes these issues and takes advantage of the insert-only/no-deletion characteristic of temporal databases.

The Object-Oriented Model

It is widely recognized that relational models are inadequate for capturing the semantics of complex objects that arise in many applications (Atkinson, 1989). Some
problems which arise are caused by the fact that temporal and non-temporal data are not treated uniformly. Another short coming stems from the fact that some relational models deal with only a single notion of time, where as some applications may want to track transaction time and valid time (Wuu & Dayal, 1992).

The deficiencies in the relational model have led to an abundance of research in the object-oriented model. The Simula programming language was the first to introduce the idea of an "object" as a programming construct, in 1973 (Brathwaite, 1993). Many believe that the idea of using objects allows application developers to model solutions closely related to real world problems. Several object-oriented languages have been introduced since, with C++ seemingly taking the forefront. The problem with these programming languages, as far as incorporating them into database design, is that they lack support for persistence. In order for the database to function the data structures must have a continued existence over time rather than just during execution of the program. While C++ does not currently offer the persistence and other features needed to support an object-oriented database management system, it is a solid base for a object-oriented model and shall be used as such in this thesis.

The evolution of object-oriented languages has defined several requirements for an object-oriented database
management system (Brathwaite, 1992). These requirements include, object identity, object typing, encapsulation, and inheritance. It is important to understand the differences these requirements create between relational and object-oriented databases.

While the relational model is based on tables of records, with each record in a table having some relation to all the others, object-oriented models are based on individual objects, each having a series of attributes that describe the object. The notion of object identity states that objects must be distinguishable, even if two or more objects have identical attribute values. This is opposed to relational models that require a unique key value be present to distinguish records in a table.

Object typing is generally more flexible than the typing of records in the relational model. In the latter, each table is defined as to the types of values it may contain, and each record must adhere to that definition. The object-oriented model, on the other hand, uses a base of atomic values, which can be built up into more complex data types, if needed.

The notion of encapsulation is unique to object-oriented programming. In this system, not only does each object have a set of attributes associated with it, but also a set of functions that manipulate the values of the attributes. The functions are as much a part of the object
as the attributes. The significance of encapsulation can be seen when the notion of inheritance is explained.

A much touted feature of the object-oriented technology is the reusability of code and objects. This reusability of code leads to enhanced productivity, system quality, and modelling power (Brathwaite, 1992). Inheritance is a crucial component of the reusability feature. For example, if one were developing a system to display geometric objects on a computer display, a non-object-oriented system would require that a data type be developed for each geometric shape. An object-oriented system realizes that although different, each shape is derived from a common base. The fundamental unit of graphics is the single point on the screen (Turbo C++, 1992). As such, a POINT class (object) is created along with attributes (x y screen coordinates) and functions to manipulate these attribute values. At a minimum, a set and retrieve function is needed for each attribute in the class. A CIRCLE class can then be derived from the base class POINT. The CIRCLE class would inherit all attributes and functions associated with the POINT class. In most cases, the derived class will define additional attributes and functions needed to complete the new object. In our example, a circumference attribute would be the only additional attribute needed to provide our object with enough information to display a circle on a computer screen.

All geometric figures in the system would in fact be
derived from the POINT class. The derivations can continue theoretically indefinitely. If an object were defined as a sub-type of CIRCLE, this new object would inherit all the attributes and functions of POINT and CIRCLE. The actual structure of an object-oriented language is much more complex than the brief overview given, but a review of all the intricacies involved are beyond the scope of this paper.

Many models have been proposed for object-oriented databases. A major problem hampering the research in this area is the lack of acceptable standards. The standards are so ill defined that many products on the market that are labeled as object-oriented database management systems do not really support all aspects of the object-oriented model (Kim, 1994).

Several models have been developed where time has been added to the object-oriented model. Of these, the OODAPLEX model seems to be the most prominent (Wuu & Dayal, 1992). The OODAPLEX model supports the major components of an object-oriented model, such as object identity, encapsulation, complex object types, inheritance, and polymorphism. Polymorphism, the only component not yet described, involves using the same name or symbol within a derived class hierarchy. Each class in the hierarchy would then implement the function in a manner which is exclusive to itself. The model also includes a temporal query language. Recently the OODAPLEX model has been challenged by
the OOTempDBM (Cheng & Gadia, 1993). The creators of OOTempDBM incorporate many of the features of the OODAPLEX model and offer some enhancements. While many of the features presented in these data models are in fact features present in most object-oriented languages, they make significant contributions by coupling the temporal object-oriented model with temporal query languages.

Even though current research has made considerable contributions to the area of temporal object databases, a number of major objectives still remain (Pissinou, Makki & Yesha, 1993). We feel that current models are lacking in the area of specifying the data structures used to organize the temporal data, especially in a manner as to offer efficient sequencing through historical data. We will address these data structure and storage concerns in detail. We do not specify a query language with our model, as we feel that existing languages are more than adequate for such purposes, and could relatively easily be adapted for our model.

Evolution of Geographic Information Systems

There are many definitions that accurately describe a geographic information system. As with practically every developing technology, formal standards and definitions are not easily found. It is quite possible that in the area of
geographic information systems tight definitions will never be developed, simply because of the wide variety of applications for which the systems are useful. In fact, many systems are labeled as a geographic information system by their creators, but fail to qualify as such in the eyes of others (Carter, 1989). Even so, generic criteria can be used to define these systems. Regardless of the individual purpose, each system must address and perform the following functions: input, storage, manipulation, and display of geographical data (Ripple, 1989).

Geographic information systems are not by definition a computer based system, nor are they a recent invention (Parent & Church, 1989). As with most computer applications, manual procedures and applications were in existence long before the advent of computers. Computers did not create the need for the applications, but they do allow the user to obtain the needed results in a timely and efficient manner.

An obvious trait is that geographic information systems are spacial by nature (Parent & Church, 1989). Regardless of the specific application, a geographic information system concerns itself with attribute values pertaining to a specific area on the earth’s surface. The development of cartography and the production of the first accurate base maps can be traced back to the mid-eighteenth century. The idea of recording different layers of data on a series of base maps was an established cartographic convention by the
time of the American Revolutionary War (Harley & Petchenik, 1978). This task of overlaying maps is a critical function for most geographic information systems. The problems associated with the limitless amount of overlay combinations has largely been simplified by use of computer based systems. Users of these systems can create and study many layered maps, before ever producing a hard copy.

It is felt that the economics involved with the industrial revolution was the main catalyst in the evolution of geographic information systems. The need to construct an infrastructure to service the increased manufacturing and inevitable clustering of people in large cities was met by the utility provided by geographic information systems. As science and technology advanced to solve the problems of the day, with remarkably improved accuracy and speed, users were able to envision other problems where a geographic information system would be of use. This cycle of improving technology and creation of new applications will probably continue indefinitely.

To extend the general definition for a computer based geographic information system a bit further, it is not presumptuous to state that geographic information systems are actually a subset of information systems in general. Indeed, any information system would need to satisfy the four requirements input, storage, manipulation, and display, albeit not necessarily geographic information, that were
used to define a geographic information system. Not surprisingly, there are many systems which can be thought of as subsets to the generic geographic information systems. Urban, natural resource, weather forecasting, marine exploration, petroleum exploration, and navigation information systems are just several examples.

Graphical data representation is a critical component of most geographic information systems. At some point, the users of the system will wish to produce some sort of visual output, usually in the form of a map. As such, there are a variety of spatial representations that any geographic information system must handle (Peuquet, 1984). These geometric representations and a sample of their use are listed in Table 1.

<table>
<thead>
<tr>
<th>FORM</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isolated polygons</td>
<td>Selected metropolitan areas</td>
</tr>
<tr>
<td>Adjacent polygons</td>
<td>Map of state outlines</td>
</tr>
<tr>
<td>Nested polygons</td>
<td>Contour maps</td>
</tr>
<tr>
<td>Point data</td>
<td>Individual cities</td>
</tr>
<tr>
<td>Line data</td>
<td>Rivers</td>
</tr>
<tr>
<td>Disconnected lines</td>
<td>Faults</td>
</tr>
<tr>
<td>Network Structure</td>
<td>Highway system</td>
</tr>
</tbody>
</table>

Table 1
A handful of basic data models exist, for representing spacial data (Peuquet, 1984). The first, and most obvious is the analog model. This model is nothing more than a map. Drawbacks of this model are numerous. In order to update the data, an entire new map would have to be drawn. Also the accuracy of the information on the map could be suspect.

The second is a vector based model. The basic unit in this system is the line. To over simplify the problem, all objects are composed of a sequence of line segments. Several vector based methods exist, but we shall not detail them.

The third data model comes in several types. This group of models is called tessellation. Again, to over simplify the issue, tessellation involves dividing the area in question into a mesh of like shaped polygons. Each cell becomes the basic unit of space for which entity information is recorded. The three main types of mesh polygons are triangles, squares, and hexagons. Each polygon has an application in which it is preferable to the others, but the square representation is the most widely used. Regardless of the shape chosen, each individual cell can only contain one object (Medeiros & Pires, 1994).

The task of storing polygons in a geographic information system is not the only concern for these applications. Several methods are available that aid in the retrieval of geometric objects. These methods are particularly beneficial because the target area of a
graphical display may contain more than one object and possible portions of objects.

The R-tree, and its derivatives, are such data structures (Gutterman, 1984). In the R-tree, a geometric object is represented by its Minimum Bounding Rectangle (MBR). Each object is then represented as a node in the tree. Parents in the tree represent MBR's of all its children. The R-tree is an adaptation of the B-tree for representing point data that does not require regions covered by the nodes to be disjoint.

Several derivatives of the R-tree have been presented to improve on aspects of the R-tree. The packed R-tree (Roussopoulos & Leifker, 1985), the k-d tree (Ooi, McDonell & Sacks-Davis, 1987), R*-trees (Lin & Ozsoyoglu, 1994), and TI/R-trees (Kouramajian & Elmasri, 1994) are some of these derivatives.

There are two main methods of representing data using these structures: overlapping regions and clipping. Both methods attempt to organize geometric objects into MBR’s, with a major difference. The overlapping region method does not allow objects to be split among the leaves of the tree, while the clipping schemes allow splitting to occur.

The major problem with the overlapping regions method can be found when one examines an object such as a river. The MBR of a river could be so large that it encompasses an entire city, and all of the MBR’s of the objects in the
city. This can cause problems, especially when searching for objects which overlap or lie in close proximity to one another. The clipping schemes would break the river into a number of smaller MBR’s, which would allow for a more accurate approximation of the object. The R-tree data structure is an example of an overlapping region method. The R²-tree is an example of the clipping scheme.

It is also interesting to note the TI/R-tree. This data structure adds the temporal feature by embedding incremental R-trees into the buckets of the conventional time index tree. Using this method a complete R-tree can be constructed for any time instance.

While we are not interested in specific existing geographic information systems, we will give a brief overview of the most prevalent commercial package simply because it will bring to light several specific areas which will influence the development of our model. ARC/INFO has been an industry leader in the commercial geographic information system market since the 1980’s. The system, like many other commercial systems, organizes data in two formats, relational and topological. The topological data structures are utilized to handle the locational data in the system. The topological data structure has several advantages over previous formats (Peuquet & Marble, 1990). The main advantage is that polygon boundary data may be more efficiently stored as structured networks of line segments.
or arcs, rather than as closed polygon loops. This arc-node data structure significantly reduces storage costs, as opposed to other systems. Other advantages, such as speed of retrieval, can be traced to the improved storage technique. ARC/INFO utilizes a relation database management system to manage all attribute data.

The areas of geometric involvement discussed can be greatly expanded, but these specifics are not within the scope of this thesis. It is also not within the scope of this thesis to concern ourselves with the intricacies entailed with individual geographic information system applications. It is our desire, however, to create a model, which will allow for the use of all these applications in a temporal object-oriented environment. The temporality of the system is where we would like to place the most emphasis. It is to this goal where we will concentrate our efforts. We hope the examples presented will show that our model can be used as a basis for any temporal geographic information system.
CHAPTER 2

DEFINING THE MODEL

Our model of a temporal object-oriented geographic information system has been derived from the current research in the area. The subject of object-oriented databases has received much research attention in recent years. It is the feeling of many that object-oriented technology is superior to the widely available and used relational model. Temporal database design has also been the subject of recent research, but it seems that the bulk of the research has been directed toward adding time to the relational model. One obvious reason for this is that relational databases are utilized far more often than the relatively new object-oriented products. Naturally some of this research has been focused on temporal geographic information systems. In our opinion, the subject of temporal object-oriented geographic information systems has not been fully examined and many useful applications for such have seemingly been neglected. This, of course, provides the opportunity for us to model the following temporal object-
oriented geographic information system.

This chapter will give an overview of our model and the data structures involved. Since geographic information systems are a subset of databases in general, it makes sense that the first step needed in order to develop our model is to understand the basics of an object-oriented database. We will do this in the following section. We will then determine the most practical method of adding time to the model. In the final section of this chapter we will tailor our model for geographic applications.

In order to present our model in a clear fashion, we will present all proper name objects and classes in capital letters.

Defining the Object-Oriented Model

A brief overview of the object-oriented model was presented in the introduction. The construct will now be expanded, as the basics for an object-oriented database are defined. To this extent one can assume that there are two atomic objects: a single numeric character and a single alpha character. A character string is an object derived from the base class of a single alpha character. In most instances the terms 'object' and 'class' are interchangeable; 'class' being a C++ term used to define an
object. Floating point, integers, and other forms of numeric data are objects derived from the base class of a single numeric object. In similar fashion, an alpha numeric string is an object derived from both the alpha and numeric atomic classes. For most practical purposes defining objects at this level is not necessary, as most object-oriented languages have predefined object types for character and numeric representation.

A general object-oriented modeling technique will be used to define the basis for our system. Figure 4 shows an object of type COUNTRY and its five attributes. It is important to realize that everything in an object-oriented database is itself an object, possibly derived from a combination of one or more base or atomic classes. Even the attributes of an object are objects derived from some primitive base. The five attributes defined for COUNTRY are date, name, population, political leader, and boundaries. The object-oriented model does not place restrictions on the number of attributes that can be assigned to a particular object. Obviously, in practice, a country object would have many more attributes associated with it. At first glance, one might assume that the attribute name would be the key to indexing the COUNTRY object; but this is not the case, as there are no key values similar to those of a relational model. In an object-oriented model each object stands alone as a separate entity. It is possible to have two objects
whose attributes are identical in value, yet the objects will still be distinguishable to the system.

In the COUNTRY example, the date attribute contains a value which represents the date when the country became a sovereign state. The date attribute is conceptually derived from the atomic numeric class. The atomic objects and base classes are not shown for every attribute, due to cluttering, space restrictions, and the fact that in practice these object types are defined by the system. The name attribute has the obvious function of storing the name of the country. Unlike the relational model, which would use the name as a key value, our object-oriented model allows for the name of the country to change, while still representing the same object. The population attribute is derived from the integer class, which has been derived from the atomic numeric class. The political leader attribute is the first complex attribute in our example. This attribute of the COUNTRY object actually has attributes of its own. In actuality the political leader attribute would be implemented as a stand-alone object, which is referenced from the COUNTRY object via pointer. The reasons for doing so will be explained later in this chapter. The attribute boundaries is actually a multi-occurring attribute. It is possible to represent these multi-occurrences with an array or linked list data structure. The boundaries attribute is also a complex attribute, it consists of the boundary
coordinates and a bordering COUNTRY object. The coordinates portion stores geographic information pertaining to the country’s borders. While it may appear that the boundaries attribute is actually derived from a country class, this is not the case. In reality, the boundaries object has an attribute which points to the country object, which shares the common boundary with the host country. The flexibility of whether or not to incorporate a data value as an attribute or as an object should become clearer as we develop our model.

There are several other attributes that belong to the COUNTRY object, which are not shown in Figure 4. These attributes include date and time stamps for when the object was created, last updated, and ceased to exist. These attributes are located in a base class called EXISTENCE, which is used to derive all temporal objects in our system. These unseen attributes are identical in function to the attributes discussed, however they are not temporal.

The functionality of object-oriented models will not be discussed further at this time. In this section, we have defined the basics of a primitive object, with several attributes, derived from a base class. In the next section we will add the time dimension to COUNTRY and make it a temporal object.
It is in this area of adding time where we feel our model has made the most significant contribution. In this section, we will add temporal capabilities to the object-oriented model previously described, and detail primary and secondary data structures that will allow easy and efficient retrieval of historical data. As stated earlier, our primary concern is with the sequential access to historical data, rather than a one time query. The geographic information system envisioned is graphical in nature and will be used to display time sequences of the graphical data.

The first question we need to answer is, what type of versioning to use for our model. We felt it desirable for each attribute related to an object be independently variable over time. Therefore, attribute versioning was chosen over object versioning (the equivalent to the relational tuple versioning). When all objects in the database are taken into consideration, it becomes clear why the storage benefits of attribute versioning is preferable.

The COUNTRY object introduced in the previous section will now be expanded to include the time dimension. Figure 5 shows the inclusion of a time square attached to almost every attribute of the object. Technically, every attribute in the system could have a temporal component attached to
it, but in practice this would never happen. For instance, it would be possible to add temporal capabilities to the 'of type' attribute of the population object; but this would only be utilized if population's type were to change from integer to real, for example. Clearly, changes like this in a database are not practical or desired. On the other hand, it is very desirable to add temporality to the population attribute of the COUNTRY object. It is expected that the population of a country will change over time.

At this point we would like to address the modelling issue of objects and attributes mentioned in the previous section. The question is whether an object's attribute is an object in its own right, simply referenced by the object via pointer; or is it actually part of the object, derived from a base class or defined in the object itself. While it is possible for every circle and oval in the Figure 5 to be an object, this point of view is only helpful when designing the conceptual model. As alluded to earlier, an object derived from other objects will inherit all attributes and functions related to the base object. Referring again to Figure 5, it is not desirable to derive the COUNTRY object from the five objects, which represent its attributes.

Consider the political leader attribute. It is perfectly legal to incorporate this information into the COUNTRY object via the derivation process. In this case all functions and attributes encapsulated in a POLITICAL LEADER
object would be inherited into the COUNTRY object. The political leader data would then belong to this derived object and only this object. On the other hand, if the political leader attribute of the COUNTRY object were just a pointer to a stand-alone POLITICAL LEADER object, this inheritance would not occur, but access to the information would still be available. What is the difference?

The main purpose of object-oriented programming is to simplify development by reducing redundancy and allowing for the reusability of code. While both methods described supply the same functionality, one would force the system to use redundant data. If the POLITICAL LEADER object was encapsulated into the COUNTRY object, we would have to populate the COUNTRY object with the political leader information. Suppose that another object in the database wished to access the information regarding the political leader in question. This second area would then have to access the country for which this person is attached by making function calls to the COUNTRY object. The alternative would be for the second object to contain within itself a person attribute and populate itself with redundant data. Needless to say this method is cumbersome and not at all desirable.

Using a pointer to reference the POLITICAL LEADER object is the method to use in this case. In this instance, any number of objects can reference the same person, without
data redundancy. This also has the desirable effect of only requiring one update to the database, if the person's data changes. A similar case can be made for using a pointer to connect a boundary to its associated countries. Although this boundary sharing technique may cause problems if the two countries in question are involved in a boundary dispute. Ultimately, these decisions will be made by the database developer, after weighing the advantages and disadvantages.

Of the many types of time to consider, when developing a temporal database, one obvious time value of interest is the time represented by the boxes in Figure 5. This time describes when the value of the attribute became valid. This time is often referred to as world time.

It is clear that each object in the database will have a creation time, a time for which the object becomes an entity. Some objects will also have a termination time, a time at which the object ceases to exist. Both these times are defined in the generic base class we shall call the EXISTENCE class. While it is possible for an individual object to die and come back to life, we will not address that phenomenon, except to say that our model will allow for extensions to allow such occurrences simply by adding extra time attributes to the EXISTENCE class or by making these time values temporal in their own right.

It is also desirable to know at what time an attribute
value of the object was updated. This brings us back to the issue of whether or not to derive an object from a base object or use a pointer to reference the information. Changes to attributes in a base object will imply that the derived object has been changed. Changes to attributes in an object pointed to by another does not imply that the pointing object has changed. This is just another design decision which must be addressed when developing the database.

The toughest decisions to make, when adding time to a database, is how to handle the versioning of attributes. Specifically, what data structures will offer the best results for temporal queries and chronological sequencing. We have realized that most research into temporal database design has concentrated on developing data structures that allow for efficient single temporal queries. While this issue is also important to us, we are also interested in efficient chronological progression once the object of the initial temporal query has been found.

First, we must address storage problems associated with temporal databases. The fact that a temporal database never deletes records, causes one to realize that the size of the database will grow to an unlimited size. In all probability, it would not take very long before the entire database overflows conventional magnetic storage devices. Because of this, and the fact that we want our model to be independent
from the hardware chosen, we will model our data structures to be compatible with any type of random access storage medium.

The first data structure we need to incorporate into our model is one to maintain and organize the temporal attribute data. The function of keeping track of temporal attribute values is dynamic in nature, although the collection of values will only grow, as deletions are not allowed in a temporal database. We will model our database so that each object will only contain attribute values representing the current state of the object. In other words all temporal data will reside outside the structure of the object. A desirable characteristic of the data structure chosen is to be able to perform fast searches for specific temporal data. We have chosen a balanced binary tree to perform this function. To allow this structure to efficiently access data during a chronological search, we have added chronological pointers. These pointers will be doubly linked in order to allow forward or backward traversal. Since these trees are dynamic and as such must be stored on a medium which allows updating, they will contain no attribute data, but rather pointers to where the data is stored, most likely on a permanent medium. Without loss of generality we will assume this permanent medium is an optical disk. An example of this data structure is depicted in Figure 6.
It is fair to ask why a new data structure, the attribute tree, is needed when other data structures have been developed to incorporate temporal data, namely the monotonic B+-tree. It is our feeling that the monotonic B+-tree does not fully lend itself to the object-oriented paradigm. We feel that all data pertaining to an object should be contained in the object itself, or at least referenced from the object. While the monotonic B+-tree may be adapted to contain only pointers to the data in an object, internal object data structures will still be needed to attach the data to the object. Also, since we are concerned with temporal progressions, the attributes of each object would have to be reconstructed at the beginning of each leaf node, if using the monotonic B+-tree. No reconstruction is ever needed using our attribute tree. The attribute tree also allows for the next node of a particular attribute to be found in constant time. If one were to use a monotonic B+-tree, this function can not be performed in constant time, and there is no way to tell if there is even a next node without scanning through the entire database.

In our first version of the attribute tree we indexed the structure according to the time when the attribute value changed. Since the target time of the search will most likely not equal the time value used in the index, we quickly found a flaw with this method in that every search will require a traversal of the binary tree until reaching a
leaf. One additional move may be necessary, using the chronological index, in order to find the desired key value. This indexing scheme requires a search time of \( \log n + 1 \) for every search. To reduce the average search time we reverted to a more common method, by adding a second time value in each node, signifying the time when that attribute value ceased to be valid. Instead of each node containing only a single time value, it is now possible to determine the interval of time the values in the node are valid. As a result, the search will not need to reach a leaf, before finding the desired node. This also aids performance during chronological searches, since the time at which the data becomes invalid is readily available for forward and backward progressions.

Although only the five pointers are shown in Figure 6, the beginning and ending time values for the interval are also stored in each node. The root of the tree is chosen in such a manner as to keep the tree balanced. The thick lines represent the binary tree pointers, while the thin lines represent the chronological pointers. The nature of the data structure forces the oldest version of the attribute to the far left of the figure and the most recent to the far right. The optical pointer, not shown, contains the address on the optical disk where the data associated with the node is stored. It is not necessary to store any information regarding the length or type of the attribute data stored on
the optical disk or in the tree, as the system will be able
to determine the characteristics by referencing the object
to which the attribute is attached.

The fact that only pointer and time values are stored
in the tree reduces the dynamic storage requirements needed
to maintain the temporal data. It is possible to find
instances where the attribute actually requires less memory
than the pointer and time stamp components. However, we do
not anticipate many attribute trees where this will be
applicable, especially considering the complex geometric
figures associated with geographic information systems.

Since each temporal attribute of each object in the
database will require an attribute tree, there will be a
significant amount of overhead required to maintain the
large number of pointers. This overhead is the trade-off
between the attribute and object versioning models.

While the attribute tree allows for the maintenance of
and access to the temporal attribute data, it does nothing
to organize the various objects in the system. At any one
instance in time, depending on the system, a large portion
of the objects in the database may not even be in existence.
If there were ten thousand objects in the system, but at
time \( n \) only one thousand were in existence, we would not
want to waste system resources by accessing each object in
the system. The question then arises, how can we efficiently
find only those objects that are in existence for the
desired query time? In addition, if after finding those objects a chronological progression is desired, how can we determine when a live object dies, and just as important when a new object is created, without searching through the entire database at every iteration?

The monotonic B+-tree is the solution to these problems. Instead of using the monotonic B+-tree to keep track of when attribute values change, we will implement it to keep track of when objects are in existence. The monotonic B+-tree will be updated every time an object is created or marked for death. The question then arises, what objects should we place in the monotonic B+-tree? Although tracking every object in the database would be feasible, it may not be necessary. For example, the POLITICAL LEADER object discussed earlier, would not need to be placed into this data structure, since the person referenced will never be displayed temporally in our database. The only time that the political leader object would need to be validated as being in existence is when the COUNTRY object, or other objects in the database, is updated to reference a particular person. This validation will be done at the time of the update and consists of a single query to the database.

While the monotonic B+-tree solves the problem at hand, its efficiency is in question for active databases with many object types during chronological queries where only a
fraction of the database objects are of concern. For example, assume a database with one hundred different objects which may be of interest during a chronological progression. In this example, the data values change frequently, which will lead to many updates of the database and a large number of entries in the buckets of the monotonic B+-tree. When a chronological progression is requested, the system will have to check all of the data in the buckets of the monotonic B+-tree at each iteration. On progressions where only a fraction of the objects are of concern, much time is wasted checking for changes in unwanted objects.

We offer a solution to this problem by partitioning the buckets of the monotonic B+-tree. By partitioning the buckets we allow for the objects in the database to be divided into logical groups. These groups would be chosen by the designer of the database in a manner which would be deemed beneficial to the performance of the system. Under these circumstances, a temporal sequence query to display only one of the objects in the database could be performed efficiently, without the loss of performance expected by searching through an entire unsegregated bucket.

We show an example of this partitioning in Figure 7. Each bucket has been partitioned into four groups; 'a', 'b', 'c', 'd'. If a chronological progression is run for objects in group 'a' only, 17 entries must be checked. If the
buckets weren't partitioned, 65 entries would have to be checked. This differential is magnified as the size of the database increases and the duration of the progression increases.

It also must be noted that we have added an incremental bucket to the end of each leaf node in the monotonic B+-tree. This additional bucket is needed to accommodate temporal progressions which span leaf nodes. Since complete buckets are only available at the beginning of each leaf node, a current list of objects must be recreated whenever a progression jumps to a new node. In the backward direction the system would have to jump to the beginning of the new leaf and scan the entire node to construct the current information for the last key in the node before proceeding. The addition of the incremental bucket eliminates this time consuming process and provides for a seamless progression among leaf nodes.

Addressing the Spacial Geographic Information System Issues

We have stated that there are many different types of geographic information systems. It would be impossible for us to define one model that will satisfy all of the requirements for every possible system. Instead, we will
address several issues which may be of concern to many geographic information systems.

The previous section used a very simplistic country object as the basis for a geographic information system. However, we may not always be interested in a country per se, but rather what is located at a particular coordinate at a given time. Figure 8 shows the model of a LOCATION object. This figure depicts an object with three attributes. In this instance the coordinates in question are not an attribute of the object. Since the object itself is actually a coordinate, the name of the object will reflect the coordinate values for which we wish to gather data. This type of modeling is possible because the coordinates of the object will not change over time. This is opposed to the name attribute for our COUNTRY object. A country is an abstract object, which can have varying borders, population, and name. Thus, the name is not a defining attribute of the object. Assuming that the method used to determine coordinate values does not change, a LOCATION object will be associated with one and only one coordinate value throughout its existence.

Another object a geographic information system may be interested in tracking is a moveable object. Figure 9 shows this type of object, which we have labeled as ANY object. The ANY object is the base class for any moveable object we may wish to include in our database. We have defined four
attributes, but clearly many more possibilities exist, depending on the type of object we are tracking. It is interesting to note the circularity involved between Figure 8 and Figure 9. The LOCATION object has an attribute OCCUPIED BY, which is actually a pointer to an ANY object. This ANY object occupies the coordinated in question. The ANY object contains an attribute coordinates, which can be used to reference the coordinate object representing the location of the ANY object. This phenomenon points out the necessity for strong data integrity in the database. Since all of the attributes in question are of the temporal variety, failure to update the attribute pointers properly will result in the corruption of the temporal feature. Erroneous temporal data values can cause drastic errors in these circular references.

It must be noted that in practice the disjoint time attribute, included only in Figure 9, is just a pictorial representation of the EXISTENCE class, which we have introduced earlier. This attribute is not temporal.

We also feel that the R-tree, or any of its derivatives, can be successfully incorporated into our model. We will address the issues concerning the use of the TI/R-tree. Again, our feeling that all data in the object-oriented model should be contained in the object, would suggest that the data in the TI/R-tree would have to be references to the object rather than the data itself. Since
we find it unrealistic to actually divide an object, as in a clipping scheme, any request to the object for data concerning an MBR would have to be accompanied with actual coordinates of the MBR. The object would use these coordinates to return the appropriate information. These minor adaptations of the TI/R-tree will reduce the need for redundant data, as well as reduce the storage requirements for the TI/R-tree. These storage savings are realized since each node in the tree need only contain the coordinates of the MBR it represents and a list of objects that are in the MBR at the time in question. No actual spacial data is present in the node.
CHAPTER 3

MODELLING A TEMPORAL OBJECT-ORIENTED GEOGRAPHIC INFORMATION SYSTEM

This chapter will present a simple example of a temporal object-oriented geographic information system. The example will make use of stationary and moveable objects for which we wish to record temporal data. Because common standards and definitions do not exist, development of a system will be language dependent. All specifics outlined in this paper will pertain to Borland's Turbo C++ version 3.0.

We will now use our model to structure a simple geographic information system. In this problem we wish to monitor the movements of an ice glacier, as it slowly slides down the face of a mountain into the ocean. Data collecting devices, which we shall call beacons, will be placed into the ice of the glacier at varying locations and depths. The beacons have the ability to monitor the temperature and pressure of the immediate area and transmit the data to a remote receiver. The data will then be loaded into our object-oriented geographic information system. The compiled
data will be used to monitor movement of the glacier and allow researchers to accurately predict when pieces of the glacier will break off and fall into the ocean.

This example will show what we feel is an important feature of our model. Most databases are concerned with single queries. For this reason the data is stored in data structures that lend themselves to a fast search time. While the data structures employed in our model also lend themselves to fast query time, they were designed for the efficient chronological retrieval of data, once an initial search has been performed. Chronological retrievals can be performed in constant time. This constant retrieval time for the next chronological data element is important for systems which rely heavily on graphical representation of temporal data.

Our example assumes that chronological retrieval is required. The complete history of beacon data is needed to allow for accurate forecasting. Not only will the data collected by our geographic information system be used to formulate mathematical equations, but it will be viewed graphically in forward and backward time sequences. While a case can be made that real-time access to temporal data is not essential for monitoring slow moving glaciers, geographic information systems may be used in areas such as warfare, where fast response time and accurate forecasting are crucial.
As stated in the overview of our model, the base class for our database is an existence class, called EXISTENCE. We will also create a class called BEACON, which is derived from the EXISTENCE class, and will inherit all attributes associated with the EXISTENCE class. We may also wish to monitor a particular location, possibly near the edge of the glacier, to determine if the boundary is expanding, contracting, or oscillating. For this purpose we have created a class called LOCATION, also derived from the EXISTENCE class. Figure 10 shows pseudo-C++ code, which describes the classes.

The code presented is clearly just the basic structure of the model. We have allocated enough memory to store fifty beacons. Since the beacons are not related to one another, a simple array can be used to store the beacon data. The variable B will be used to identify this array. Conventionally, B[i] will refer to the i\textsuperscript{th} beacon in the array. It can be seen from the previous code that each instance of B has seven attributes, two from the Existence class and five from the BEACON class. The data stored in B is the most recent recorded value for each attribute. The temporal data, actually pointers to the attribute trees, have purposefully been omitted in order to show the effects of adding temporal data to the database.

In order to incorporate historical data, we need to create a new class. This new class will be used to form a
tree data structure, which will point to the desired data located on secondary storage. It must be noted that no historical data is stored in this tree, only pointers to the data. The trees, which are dynamic in nature, will reside in main memory or on an updatable medium.

Our first inclination was to create a new class for each type of temporal data to be stored (integer, floating point, etc.). Upon further consideration, we discovered that this is not necessary. The tree structure will point to the starting location of the data in secondary storage. Since a separate tree structure will be needed for each attribute, the system will be able to determine the type of data the pointer is pointing to, and determine the number of bytes to retrieve. This is significant because each attribute value may not be a simple data value. In fact, there will be a minimum of two data values stored for any attribute. The update time, the time at which the database was modified for this attribute, and the value of the attribute itself. The world time, the time where the attribute value actually changed, is found in the attribute tree. It is possible for the pointer to be pointing to a far larger and more complex group of information, namely topological data structures used to represent geometric shapes. Pseudo-C++ code for the Attrib_tree class can be found in Figure 11. The Attrib_tree class is used to construct all of the attribute trees.

As stated earlier, two types of tree pointers are used
in our attribute trees. Left_Tree_Ptr and Right_Tree_Ptr are the pointers used to construct the balanced binary tree. These pointers will be utilized to search the tree as a result of a temporal query. Prev_Chrono_Ptr and Next_Chrono_Ptr are pointers used to organize the temporal attribute values in chronological order. They will be utilized to perform chronological retrievals, after a temporal query has been performed. The chronological pointers provide for a constant search time for chronological retrievals.

With the Attrib_tree class created, we can now update the BEACON class to include necessary pointers to the historical data. Figure 12 shows the updated temporal BEACON class.

With the addition of temporal pointers and values, the BEACON class becomes very large. It can be seen that as a minimum each attribute requires five components, two pointers and three data values, in order to acquire temporal characteristics. The purpose of each component, excluding the attribute’s data value, will now be explained.

Root_Pointer - this points to the root of the balanced binary tree containing attribute information. The temporal data for each attribute is stored in separate trees. The root is chosen as to minimize search time for temporal queries. The root pointer will contain a
value of NULL, if there are no historical values for
the attribute.

Chrono_Ptr - this pointer can have two uses depending on
the desired use of the database. One option calls for
this value to point to the immediate predecessor to the
current attribute value. The second option calls for
the value to point to the initial value of the
attribute in the tree. If necessary an additional
pointer can be added to the class as to accommodate
both purposes. The chronological pointer will contain a
value of NULL, if there are no historical values for
the attribute.

Update_time - this value represents the time that the
current attribute value was entered into the database.
The historical update times are stored with the
historical data on secondary or optical storage.

World_time - this value represents the actual time the
change to the attribute takes place. The world time is
the time value used to index the attribute tree. The
historical world time is stored in the attribute tree.

Figure 13 depicts a graphical representation of the
temporal BEACON class. The object-oriented characteristics
of our model allows this class to be reused, if necessary to build a more complex object. One example would be if a new model of a beacon were produced that contained more features (attributes). This reusability of objects is a major benefit of object-oriented systems. In our example we will only be monitoring one glacier with one type of beacon. Therefore, this class is only used in the B array, which contains all the beacons in the example.
CHAPTER 4

DEVELOPMENT OF A PROTOTYPE

In this chapter we will develop and implement a simple geographic information system. The code will be written in Borland’s C++ version 3.0 and will operate on IBM compatible personal computers. A complete listing of the prototype source code along with all header files can be found in Appendix I.

Prototype Specifics

We will now develop a partial prototype to emphasize the temporal sequencing method of our model. The geographic information system we will develop will be limited to ten COUNTRY objects. We will also limit the number of attributes for each COUNTRY to three: name, population, and boundary.

In order to keep the prototype at a manageable size and to accommodate the prototype environment, it is necessary to place other restrictions on our model. First, user
interaction as far as adds, updates, and deletes to the database will not be allowed. The database will be populated using a sequential data file. The routines used to populate the database, however, are equivalent to routines which would be used in an interactive environment. None of the rules of object-orientation or of our model are broken by using this method.

Upon execution our prototype will populate the database from a sequential data file. The data file used in our simulation can be found in Appendix II. Since the model allows for an unlimited amount of line segments to form a border, two input record formats are needed. In both formats, the first data value represents the number of the country to which the data applies. The second number indicates the time at which the data became valid. The standard record has three remaining fields representing; country name, population, and number of border coordinate pairs. These fields are only populated if the data has changed. A number sign (#) is used to indicate that no change has been made to an alphanumeric field and a zero (0) to indicate no change to a numeric field.

If the border field indicates that there are \( n \) border coordinate pairs, then the next \( n \) records will be of the special border format. The coordinate pairs in these records are a sequential list outlining the boundary of the country. The first and last pairs in the sequence must be identical.
in order to close the polygon.

The program then queries the user for a starting time, restricted for our simulation to 0 through 10, and a sequence direction, -1 for backwards and 1 for forwards. The prototype will search all of the COUNTRY objects and determine which are in existence at the start time. The country's name, population, and border will then be displayed. The program will automatically progress through the chronological time series, modifying only the data that has changed from the previous display. Our test data, if run in its entirety, includes the creation and extinction of countries and properly displays the data in both forward and reverse time sequences.

Implementation Issues

As with implementing any model, problems arise which were not foreseen during the conceptual phase of the project. Some of these problems arise from the particular workings of various hardware and software products. While none of the problems we ran into implementing this prototype invalidated any of our model constructs, they did bring to light several issues which may warrant more attention in the future.
Our model calls for the use of a permanent storage medium for the historical data in the database. Since technology in this area has and will continue to change, we tried to develop our model to interface with any storage medium.

The development of optical disks has alleviated some of the problems associated with massive data storage, but several problems exist with this platform. Before we consider these, we will accentuate the positive features of optical disks.

The average floppy disk has the capacity to store roughly one and a half million bytes of information. While hard disk capacities are widely varied, two to three hundred million bytes are common. The optical disk is capable of storing over six hundred million bytes of data (Williams, 1993). Besides the increased storage capacity, optical disks offer a higher degree of data integrity. While magnetic mediums suffer loss of performance with age, optical disks theoretically suffer no degradation over time.

One of the major problems with optical disks is that current technology only allows for write once optical disks. The dynamic data structures, critical to temporal database design, such as B-trees and various forms of hashing are rendered virtually useless in a write once environment.
While we assume in our model that historical data is written to permanent storage whenever data values change, this is not the case in practice. Optical disks, for instance, use a sector as the smallest writable unit. Various partitioning research has been performed in order to efficiently handle the storage of historical data on a permanent medium (Ahn & Snodgrass, 1988; Elmasri, Jaseemuddin, & Kouramajian, 1992). We feel that our model will be compatible with any partitioning method, since the model does not care what type of medium is used to store the data; only an address is needed for retrieval.

Our prototype simulates an optical disk by using a linked list. Each time a data value is changed in the database, a new node is added to the linked list containing the values of the attribute that has changed. For simplicity, we have used a generic data structure for this linked list. Each node contains a storage location for each of the three possible attribute values, although only one will be populated in any one node. The address of the new node is then stored in the appropriate attribute tree.

Other Implementation Issues

While coding this prototype we were made aware of circumstances which we did not address while defining our
model. The first concerns the time sequence display. While a display is in progress, it is necessary to keep track of the current pointer to the historical data tree for each attribute. In our model we did not define where this pointer should be stored. For our prototype we stored it in the COUNTRY object, which seems to be the logical choice. The only problem with this choice is if for some reason there is a need to simultaneously traverse through more than one time sequence. In this instance two or more pointers would be needed. We surmise that a dynamic data structure within the host object would be able to handle this variable need, however we will leave this as an open problem.

The second problem we ran into, while developing the prototype, is the need to erase the previously displayed data prior to displaying the current data. While this procedure is definitely needed in order to operate properly within Borland Turbo C++, it may not necessarily be required for all languages. As such we added another pointer to each attribute within the COUNTRY object, which points to the location on the simulated optical disk where the currently display data is stored. This value will be used to erase the desired area of the display, before the new data is presented. Since this problem may be language dependent, we will not alter our model, but leave it as a problem to be addressed depending of the language and hardware used to implement the database.
CONCLUSIONS AND REMARKS

It was the goal of this thesis to develop a model for an operational temporal object-oriented geographic information system. The emphasis was placed on the ability to efficiently retrieve historical data in forward and backward sequences in order to display graphical time series on a video display.

To this extent, the model successfully added the time component to a temporal object-oriented database. A balanced binary tree was modified to include chronological pointers to facilitate the sequential retrieval of historical data in constant time.

The monotonic B+-tree was extended to include partitioned buckets. By partitioning the buckets, a considerable amount of processor time is saved during temporal time progressions. This benefit increases as the size of the database and the number of objects tracked increase.

The model was developed around a hypothetical
geographic information system, which tracks beacons imbedded in an ice glacier. A prototype of the model was developed using Turbo C++. The limited prototype stores temporal data for ten fictional countries and displays time progressions graphically.

Future work on this model may include, but is not limited to, development of parallel data structures and algorithms. Because of the large number of objects stored in a temporal object-oriented database, parallel processing would significantly increase response time during temporal sequencing. We will leave the specifics of this as an open problem.
B+-TREE LEAF NODE

\[ \begin{array}{ccccccc} P_1 & K_1 & P_2 & K_2 & \cdots & P_{n-1} & K_{n-1} & P_n \end{array} \]
B⁺-TREE

FIGURE 2
B+-TREE CHILD NODES

non-incremental

\[
\begin{array}{c|c|c|c}
33 & 48 & 60 & 90 \\
\end{array}
\]

{a,b,c,d} \rightarrow {a,b,e,f}

{a,b,d} \rightarrow {a,b,e,f,g}

incremental

\[
\begin{array}{c|c|c|c}
33 & 48 & 60 & 90 \\
\end{array}
\]

{a,b,c,d} \rightarrow \{-d,+e,+f\}

{-c} \rightarrow \{+g\}

FIGURE 3
A COUNTRY OBJECT

has sovereignty date

has name

has population

has political leader

has party

has birth date

has coordinates

has boundaries

FIGURE 4
A COUNTRY OBJECT WITH TIME

- country
  - at time has sovereign date
  - at time has name of type character string
  - at time has population of type integer
  - at time has political leader
    - at time has name
    - at time has political party
    - has birth date
    - at time has coordinates
  - at time has boundaries
  - at time with country

FIGURE 5
PARTITIONED B+-TREE BUCKETS

FIGURE 7
A LOCATION OBJECT

FIGURE 8
class Existence {
    time creation_time;
    time last_updated_time;
    time extinction_time;
};

class Beacon : Existence {
    float x_coord;
    float y_coord;
    float z_coord;
    float temp;
    float pressure;
};

class Location : Existence {
    string occupied_by;
    float temp;
    float pressure;
};

Beacon B[50]; // Declares 50 dimension
    // array B of type Beacon
class Attrib_tree
{
    Attrib_tree *Left_Tree_Ptr;
    Attrib_tree *Prev_Chrono_Ptr;
    time begin_interval_time;
    Optical_addr *Optical_Ptr;
    time end_interval_time;
    Attrib_tree *Next_Chrono_Ptr;
    Attrib_tree *Right_Tree_Ptr;
}
BEACON CLASS C++ CODE

class Beacon : Existence
{
    float x_coord;
    Attrib_tree *X_Root_Pointer;
    Attrib_tree *X_Chrono_Ptr;
    Time X_update-time;
    Time X_world_time;
    float y_coord;
    Attrib_tree *Y_Root_Pointer;
    Attrib_tree *Y_Chrono_Ptr;
    Time Y_update-time;
    Time Y_world_time;
    float z_coord;
    Attrib_tree *Z_Root_Pointer;
    Attrib_tree *Z_Chrono_Ptr;
    Time Z_update-time;
    Time Z_world_time;
    float temp;
    Attrib_tree *T_Root_Pointer;
    Attrib_tree *T_Chrono_Ptr;
    Time T_update-time;
    Time T_world_time;
    float pressure;
    Attrib_tree *P_Root_Pointer;
    Attrib_tree *P_Chrono_Ptr;
    Time P_update-time;
    Time P_world_time;
};
Appendix I

PROTOTYPE SOURCE CODE

Main Program

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <graphics.h>
#include <fstream.h>
#include <strstream.h>
#include "boundary.h" // declarations for graphics library
#include <fstream.h>
#include <strstream.h>
#include "boundary.h" // boundary linked list structure
#include "attribute.h" // attribute tree data structure
#include "country.h" // a country object

enum boolean {false, true};

// function prototypes
void load_data();
void get_input();
void display_time_series();
void search_for_time();
void display_header();
void clear_boarder(int);
void clear_population(int);
void clear_name(int);
void draw_boarder(int);
void display_population(int);
void display_name(int);
void wait(int);

// global variables

69
boolean end_of_program = true;
Country country[10]; // limit of 10 countries
opt_disk *opt_start = NULL; // first available location on
    // optical disk
opt_disk *opt_last = NULL; // next available location on
    // optical disk
int init_time; // initial time for time
    // series display
int init_dir; // -1 for reverse
    +1 for forward

//=== === = === =================== ====== ==== ========::= = = === = ===
//main program//==========================================================
int main()
{
    int graphdriver = DETECT, graphmode; // initialize the
    // graphics system
    if (registerbgidriver(EGAVGA_driver) < 0) exit(1);
    initgraph(&graphdriver, &graphmode, "");

    if (! (opt_start = new opt_disk))
    {
        cout << ("insufficient memory for opt_disk");
        exit;
    }
    opt_last = opt_start;
load_data(); // load data from flat file
get_input(); // query user for initial time and
    // chronological order
do
    {
        cleardevice();
        display_time_series();
        wait(100);
    } while (! end_of_program); // allow only one query per
    // run for now

    closegraph();
    return 0;
};

////////////////////////////////////////////////////////////////////////
//global functions
////////////////////////////////////////////////////////////////////////
// load data from disk file
////////////////////////////////////////////////////////////////////////
void load_data()
{
    cout << "\nLoading data from file ....\n";
    ifstream f1("init.dat");
if (fl) {
    int c_no; // number of country
    int e_time; // effective_time
    int d_type; // type of record 1-country-name
    // 2-population
    // 3-boundary
    char in_string[20]; // input string
    int in_int1; // input integer 1
    int in_int2; // input integer 2
    int bound_count; // how many boundary segments
    int is_new; // is it a new boundary 1-yes 0-no

    char inbuf[81];
    while (fl.getline(inbuf,81)) {
        istringstream ins(inbuf,strlen(inbuf));
        ins >> c_no >> e_time >> in_string >> in_int1 >> in_int2;
        if ((c_no > 10) || (c_no < 0))
            cout << "invalid c_no. record skipped\n" << c_no;
        else {
            if (strchr(in_string, '#') == NULL)
                country[c_no].set_narae(e_time,in_string,opt_last);
            if (in_int1 != -1)
                country[c_no].set_pop(e_time,in_int1,opt_last);
            bound_count = in_int2; // how many boundary coordinates to read
            is_new = 1;
            while ((bound_count-- > 0) &&
                   (fl.getline(inbuf,81))) {
                istringstream ins(inbuf,strlen(inbuf));
                ins >> c_no >> e_time >> in_string >> in_int1 >> in_int2;
                country[c_no].set_boundary
                (e_time,in_int1,in_int2,opt_last,is_new);
                is_new = 0;
            }
        }
    }
}

//================================
// ask user for initial query data
//================================
void get_input()
{
    cleardevice();
}
char buffer[41] =
   "Enter an initial time and direction F/B \0";
outtextxy(10,10,buffer);
gotoxy(10,54);
cin >> init_time >> init_dir;
}

//==========================================================
// display the time series
//==========================================================
void display_time_series()
{
  int ws_count = 0;
  int i;

cleardevice();
do
{
  setcolor(15);
  outtextxy(200,8,"AT TIME");
  outtextxy(400,8,"POPULATION");
  for (i = 0; i < 10; i++)
  {
    display_header();
    if (((country[i].return_creation_time() < init_time)
        &&
        (country[i].return_creation_time() != -1))
        &&
        ((country[i].return_extinction_time() > init_time) ||
        (country[i].return_extinction_time() == -1))
        &&
        (country[i].return_extinction_time() == init_time) &&
        (init_dir == -1))
    {
      if (ws_count == 0)
        country[i].search(init_time); // search for initial
        // time in tree
      else
        country[i].next_in_seq(init_time); // check if any
        // data changed
      if (country[i].return_n_changed() == 1)
      {
        clear_name(i);
        display_name(i);
      }
      if (country[i].return_b_changed() == 1)
      {
        clear_boarder(i);
        draw_boarder(i);
      }
      if (country[i].return_p_changed() == 1)
      {
        clear_population(i);
        display_population(i);
      }
  }
  ws_count++;
}
}
else
{
    if (((country[i].return_creation_time() == 
        (init_time + 1)) &&
        (init_dir == -1)) ||
        ((country[i].return_extinction_time() == init_time) &&
        (init_dir == 1))) // country not in
    { // existence at current
        // time. Remove all data
        // from display.
        country[i].prepare_clear();
        clear_name(i);
        clear_boarder(i);
        clear_population(i);
    }
    else
    {
        if (((country[i].return_creation_time() == 
            init_time) &&
            (init_dir == 1)) ||
            ((country[i].return_extinction_time() == 
                (init_time - 1)) &&
            (init_dir == -1) && (init_time != 0)))
        { // new country add to display
            country[i].prepare_new(init_time);
            display_name(i);
            draw_boarder(i);
            display_population(i);
        }
    }
}
}
ws_count = 1;
wait(60);
    init_time += init_dir;
} while ((init_time >= 0) && (init_time <= 10));

//========================================================================
// display header information
//========================================================================
void display_header()
{
    char buffer[6];
    setcolor(0); // blackout old data
    sprintf(buffer, "%d", (init_time - init_dir));
    outtextxy(265, 8, buffer);
    setcolor(15); // display new data
    sprintf(buffer, "%d", (init_time));
    outtextxy(265, 8, buffer);
void clear_boarder(int c)
{
   int xy[30];    // array of boundary points
   int i = 0;
   boundary *curr_ptr;

   setcolor(0);    // erase previous boundary
   curr_ptr = country[c].return_boundary(1);
   while (curr_ptr != NULL)
   {
      xy[i++] = curr_ptr->return_x();
      xy[i++] = curr_ptr->return_y();
      curr_ptr = curr_ptr->return_next();
   }
   i = i / 2;
   setfillstyle(SOLID_FILL,0);
   fillpoly(i,xy);
}

void draw_boarder(int c)
{
   int xy[30];      // array of boundary points
   int i = 0;
   boundary *curr_ptr;

   setcolor(0);    // display current boundary
   i = 0;
   curr_ptr = country[c].return_boundary(0);
   while (curr_ptr != NULL)
   {
      xy[i++] = curr_ptr->return_x();
      xy[i++] = curr_ptr->return_y();
      curr_ptr = curr_ptr->return_next();
   }
   i = i / 2;
   setfillstyle(SOLID_FILL,c+1);
   fillpoly(i,xy);
}

void clear_name(int c)
{
   char buffer[6];
setcolor(0); // erase previous name
outtextxy(400, 18+(10*c), country[c].return_name(1));
}

//========================================================
// display name
//========================================================
void display_name(int c)
{
    char buffer[6];

    setcolor(15); // display current name
    outtextxy(400, 18+(10*c), country[c].return_name(0));
}

//========================================================
// clear population
//========================================================
void clear_population(int c)
{
    char buffer[6];

    setcolor(0); // erase previous population
    sprintf(buffer, "%d", country[c].return_pop(1));
    outtextxy(500, 18+(10*c), buffer);
}

//========================================================
// display population
//========================================================
void display_population(int c)
{
    char buffer[6];

    setcolor(c+1); // display current population
    sprintf(buffer, "%d", country[c].return_pop(0));
    outtextxy(500, 18+(10*c), buffer);
}

//========================================================
// used to slow processing to accommodate human visualization
//========================================================
void wait(int zz)
{
    int i, j;

    for (i = 0; i < zz; i++)
        for (j = 0; j < 32000; j++)
            i = i + 0;
}
# country.h

// Existence class member functions
// ALL VITAL DATABASE OBJECTS WILL DERIVED FROM THIS CLASS

class Existence{
    protected:
        int creation_time;
        int extinction_time;

    public:
        int return_creation_time() {return creation_time;};
        int return_extinction_time() {return extinction_time;};
        void set_creation_time(int new_time)
            {creation_time = new_time;};
        void set_extinction_time(int new_time)
            {extinction_time = new_time;};
};

class Country : public Existence{

    char name[20]; // current name
    int n_changed; // 0 not changed : 1 changed
    Attr_tree *n_Root_ptr; // root of balanced tree
    Attr_tree *n_chrono_ptr; // previous name
    Attr_tree *n_curr_ptr; // used to keep place in time
    Attr_tree *n_prev_ptr; // used to erase previous
                          // display
    int pop; // current population
    int p_changed; // 0 not changed : 1 changed
    Attr_tree *p_Root_ptr; // root of balanced tree
    Attr_tree *p_chrono_ptr; // previous population
    Attr_tree *p_curr_ptr; // used to keep place in time
                          // lapse
    Attr_tree *p_prev_ptr; // used to erase previous
                          // display

    // boundary stored as linked list of coordinated pairs
    // forming a closed polygon
    boundary *boundary_ptr; // current boundary
    int b_changed; // 0 not changed : 1 changed
    Attr_tree *b_Root_ptr; // root of balanced tree
    Attr_tree *b_chrono_ptr; // previous boundary
    Attr_tree *b_curr_ptr; // used to keep place in time
                          // lapse
Attr_tree *b_prev_ptr; // used to erase previous
    // display

public:
    Country();
    char  *return_name(int);
    int   return_pop(int);
    boundary* return_boundary(int);
    int   return_n_changed() {return n_changed;};
    int   return_p_changed() {return p_changed;};
    int   return_b_changed() {return b_changed;};
    void  set_name(int, char *new_name, opt_disk* &);
    void  set_pop(int, int new_pop, opt_disk* &);
    void  set_boundary
    (int, int, int, opt_disk* &,int);
    void  search(int);
    void  next_in_seq(int);
    void  prepare_clear();
    void  prepare_new(int);
    void  extinct(int);

private:
    Attr_tree* add_old_to_tree();

    //***************************************************************************
    char *Country::return_name(int x)
    {
        if (x == 0)  // return current name
        {
            if (n_curr_ptr == NULL)
                return name;
            else
                return (n_curr_ptr->return_opt())->return_name();
        }
        else
        {
            return (n_prev_ptr->return_opt())->return_name();
        }
    }
    //***************************************************************************
return (n_prev_ptr->return_opt())->return_name();
}

/*********************************************
int Country::return_pop(int x)
{
    if (x == 0) // return current population
    {
        if (p_curr_ptr == NULL)
            return pop;
        else
            return (p_curr_ptr->return_opt())->return_pop();
    }
    else // return previous name in time series
    {
        if (p_prev_ptr == NULL)
            return pop;
        else
            return (p_prev_ptr->return_opt())->return_pop();
    }
}

/*********************************************
boundary* Country::return_boundary(int x)
{
    if (x == 0) // return ptr to current boundary list
    {
        if (b_curr_ptr == NULL)
            return boundary_ptr;
        else
            return (b_curr_ptr->return_opt())->return_bound();
    }
    else // return previous ptr in time series
    {
        if (b_prev_ptr == NULL)
            return boundary_ptr;
        else
            return (b_prev_ptr->return_opt())->return_bound();
    }
}

/*********************************************
void Country::set_name(int e_time, char *new_name,
                        opt_disk* &opt_last)
{
    Attr_tree *new_ptr;
    if (creation_time == -1) // the birth of a new country
    {
        creation_time = e_time;
        strcpy(name, new_name);
    }
    else
    {
opt_last->get_next(opt_last); // update optical disk
opt_last->set_name(name); // " " "
new_ptr = add_old_to_tree(); // get ptr to new // attrtree
if (n_chrono_ptr != NULL)
{
    new_ptr->set_pcp(n_chrono_ptr);
    n_chrono_ptr->set_ncp(new_ptr);
    new_ptr->set_bit(n_chrono_ptr->return_eit());
}
else
    new_ptr->set_bit(creation_time);
new_ptr->set_opt(opt_last);
new_ptr->set_eit(e_time);
    n_chrono_ptr = new_ptr;
}
strcpy(name,new_name); // set new name

//*********************************************
void Country::set_pop(int e_time, int in_int1,
    opt_disk* &opt_last)
{
    Attr_tree *new_ptr;
    if (creation_time == -1)  // the birth of a new country
    {
        creation_time = e_time;
        pop = in_int1;
    }
    else
    {
        if (pop != -1)
        {
            opt_last->get_next(opt_last); // update optical disk
            opt_last->set_pop(pop); // " " "
            new_ptr = add_old_to_tree(); // get ptr to new // attrtree
            if (p_chrono_ptr != NULL)
            {
                new_ptr->set_pcp(p_chrono_ptr);
                p_chrono_ptr->set_ncp(new_ptr);
                new_ptr->set_bit(p_chrono_ptr->return_eit());
            }
            else
                new_ptr->set_bit(creation_time);
            new_ptr->set_opt(opt_last);
            new_ptr->set_eit(e_time);
            p_chrono_ptr = new_ptr;
        }
        pop = in_int1; // set new pop
    }
void Country::set_boundary(int e_time, int point_x, int point_y, opt_disk* &opt_last, int is_new)
{
    Attr_tree *new_attr_ptr;
    boundary *new_bound_ptr;

    if (!(new_bound_ptr = new boundary))
    {
        cout << ("insufficient memory for boundary");
        exit;
    }

    if (creation_time == -1) // the birth of a new country
    {
        creation_time = e_time;
        if (boundary_ptr != NULL)
            new_bound_ptr->set_ptr(boundary_ptr);
        new_bound_ptr->set_x(point_x);
        new_bound_ptr->set_y(point_y);
        boundary_ptr = new_bound_ptr;
    }
    else
    {
        if ((boundary_ptr != NULL) && (is_new == 1))
        {
            opt_last->get_next(opt_last); // update optical disk
            opt_last->set_boundary(boundary_ptr); // "    "
            new_attr_ptr = add_old_to_tree(); //ptr to new attrtree
            if (b_chrono_ptr != NULL)
            {
                new_attr_ptr->set_pcp(b_chrono_ptr);
                b_chrono_ptr->set_ncp(new_attr_ptr);
                new_attr_ptr->set_bit(b_chrono_ptr->return_eit());
            }
            else
                new_attr_ptr->set_bit(creation_time);
            new_attr_ptr->set_opt(opt_last);
            new_attr_ptr->set_eit(e_time);
            b_chrono_ptr = new_attr_ptr;
            boundary_ptr = NULL;
        }
        new_bound_ptr->set_x(point_x);
        new_bound_ptr->set_y(point_y);
        new_bound_ptr->set_ptr(boundary_ptr);
        boundary_ptr = new_bound_ptr;
    }
}

void Country::search(int i_time)

{
    Attr_tree *work_ptr;

    if (((i_time <= extinction_time) ||
         (extinction_time == -1)) &&
        ((i_time >= creation_time) && (creation_time != -1)))
    {
        if ((n_chrono_ptr != NULL) &&
            (n_chrono_ptr->return_eit() > i_time))
        {
            work_ptr = n_chrono_ptr;
            while (i_time < work_ptr->return_bit())
                work_ptr = work_ptr->return_pcp();
            n_curr_ptr = work_ptr;
        }
        if ((p_chrono_ptr != NULL) &&
            (p_chrono_ptr->return_eit() > i_time))
        {
            work_ptr = p_chrono_ptr;
            while (i_time < work_ptr->return_bit())
                work_ptr = work_ptr->return_pcp();
            p_curr_ptr = work_ptr;
        }
        if ((b_chrono_ptr != NULL) &&
            (b_chrono_ptr->return_eit() > i_time))
        {
            work_ptr = b_chrono_ptr;
            while (i_time < work_ptr->return_bit())
                work_ptr = work_ptr->return_pcp();
            b_curr_ptr = work_ptr;
        }
    }
}

//=================================================================================================================
// check if current country’s data has changed since previous time in time series
//=================================================================================================================
void Country::next_in_seq(int i_time)
{
    n_changed = p_changed = b_changed = 0;
    if (n_curr_ptr != NULL)
    {
        if (n_curr_ptr->return_bit() > i_time)
        {
            n_changed = 1;
            n_prev_ptr = n_curr_ptr;
            n_curr_ptr = n_curr_ptr->return_pcp();
        }
        if (n_curr_ptr->return_eit() <= i_time)
        {
            n_changed = 1;
            n_prev_ptr = n_curr_ptr;
        }
    }
}
n_curr_ptr = n_curr_ptr->return_ncp();
}
else
{
    if (((n_chrono_ptr != NULL) &&
         (n_chrono_ptr->return_eit() > i_time))
    {
        n_changed = 1;
        n_prev_ptr = n_curr_ptr;
        n_curr_ptr = n_chrono_ptr;
    }
}

if (p_curr_ptr != NULL)
{
    if (p_curr_ptr->return_bit() > i_time)
    {
        p_changed = 1;
        p_prev_ptr = p_curr_ptr;
        p_curr_ptr = p_curr_ptr->return_pcp();
    }
    if (p_curr_ptr->return_eit() <= i_time)
    {
        p_changed = 1;
        p_prev_ptr = p_curr_ptr;
        p_curr_ptr = p_curr_ptr->return_ncp();
    }
}
else
{
    if (((p_chrono_ptr != NULL) &&
         (p_chrono_ptr->return_eit() > i_time))
    {
        p_changed = 1;
        p_prev_ptr = p_curr_ptr;
        p_curr_ptr = p_chrono_ptr;
    }
}

if (b_curr_ptr != NULL)
{
    if (b_curr_ptr->return_bit() > i_time)
    {
        b_changed = 1;
        b_prev_ptr = b_curr_ptr;
        b_curr_ptr = b_curr_ptr->return_pcp();
    }
    if (b_curr_ptr->return_eit() <= i_time)
    {
        b_changed = 1;
        b_prev_ptr = b_curr_ptr;
    }
}
b_curr_ptr = b_curr_ptr->return_ncp();
}
else
{
  if ((b_chrono_ptr != NULL) &&
      (b_chrono_ptr->return_eit() > i_time))
  {
    b_changed = 1;
    b_prev_ptr = b_curr_ptr;
    b_curr_ptr = b_chrono_ptr;
  }
}

// prepare to remove country from display
void Country::prepare_clear()
{
  n_prev_ptr = n_curr_ptr;
  p_prev_ptr = p_curr_ptr;
  b_prev_ptr = b_curr_ptr;
  n_curr_ptr = p_curr_ptr = b_curr_ptr = NULL;
}

// prepare to add country to display
void Country::prepare_new(int i_time)
{
  n_changed = p_changed = b_changed = 1;
  search(i_time);
}

void Country::extinct(int i_time)
{
  extinction_time = i_time;
}

Attr_tree* Country::add_old_to_tree()
{
  Attr_tree *new_ptr;

  if (!(new_ptr = new Attr_tree))
  {
    cout << ("insufficient memory for Attr_tree");
    exit;
  }
  return new_ptr;
}
// = = === = = = = = = = = = = = = = = = = = === = = = = = === = = = = = = ™  = = = = = = = = = = =
// class opt_disk and member functions
// used to simulate an optical disk
//====================================::==================
class opt_disk
{
    char    name[20];
    int     pop;
    boundary *bound;
    opt_disk *next_ptr;
public:
    opt_disk();
    void set_name(char *new_name)
    {strcpy(name,new_name);};
    void set_pop(int new_pop) {pop = new_pop;};
    void set_boundary(boundary *new_bound)
    {bound = new_bound;};
    void set_ptr(opt_disk *new_ptr)
    {next_ptr = new_ptr;};
    void get_next(opt_disk* & )
    {return name;};
    char *return_name() {return name;};
    int    return_pop() {return pop;};
    boundary* return_bound() {return bound;};
};

//******************************
void opt_disk::opt_disk()
{
    strcpy(name," ");
    pop = 0;
    bound = NULL;
    next_ptr = NULL;
};

//******************************
void opt_disk::get_next(opt_disk* &opt_last)
{
    opt_disk *new_ptr;
    if (!(new_ptr = new opt_disk))
    {
        cout << ("insufficient memory for opt_disk");
        exit;
    }
    opt_last->next_ptr = new_ptr;
    opt_last = new_ptr;
}

//******************************

//=*=*=//class attribute_tree and member functions
//******************************
class Attr_tree
{
    Attr_tree *left_tree_ptr;
    Attr_tree *prev_chrono_ptr;
    int begin_interval_time;
    opt_disk *optical_ptr;
    int end_interval_time;
    Attr_tree *next_chrono_ptr;
    Attr_tree *right_tree_ptr;
public:
    Attr_tree();
    Attr_tree* return_ltp() {return left_tree_ptr;};
    Attr_tree* return_pcp() {return prev_chrono_ptr;};
    int return_bit()
    {return begin_interval_time;};
    opt_disk* return_opt() {return optical_ptr;};
    int return_eit()
    {return end_interval_time;};
    Attr_tree* return_ncp() {return next_chrono_ptr;};
    Attr_tree* return_rtp() {return right_tree_ptr;};
    void set_ltp(Attr_tree *new_ptr)
    {left_tree_ptr = new_ptr;};
    void set_pcp(Attr_tree *new_ptr)
    {prev_chrono_ptr = new_ptr;};
    void set_bit(int new_time)
    {begin_interval_time = new_time;};
    void set_opt(opt_disk *new_ptr)
    {optical_ptr = new_ptr;};
    void set_eit(int new_time)
    {end_interval_time = new_time;};
    void set_ncp(Attr_tree *new_ptr)
    {next_chrono_ptr = new_ptr;};
    void set_rtp(Attr_tree *new_ptr)
    {right_tree_ptr = new_ptr;};
};

void Attr_tree::Attr_tree()
{
    left_tree_ptr = prev_chrono_ptr = NULL;
    optical_ptr = NULL;
    next_chrono_ptr = right_tree_ptr = NULL;
    begin_interval_time = end_interval_time = 0;
};
```cpp
//====================================
//class boundary and member functions
//====================================

class boundary
{
    int x_coord;
    int y_coord;
    boundary *next_ptr; // pointer to next in linked list

public:
    int return_x() { return x_coord; }
    int return_y() { return y_coord; }
    boundary* return_next() { return next_ptr; }
    void set_x(int new_x) { x_coord = new_x; }
    void set_y(int new_y) { y_coord = new_y; }
    void set_ptr(boundary *new_ptr) { next_ptr = new_ptr; }
};
```
Appendix II

INIT.DAT

0 0 country.a 1 5
0 0 # 100 100
0 0 # 150 100
0 0 # 150 150
0 0 # 100 150
0 0 # 100 100
0 3 country.a3 31 0
0 4 # -1 7
0 4 # 100 100
0 4 # 150 100
0 4 # 150 150
0 4 # 100 150
0 4 # 90 140
0 4 # 95 120
0 4 # 100 100
0 5 # 51 0
0 6 # 61 8
0 6 # 100 100
0 6 # 150 100
0 6 # 150 150
0 6 # 100 150
0 6 # 90 140
0 6 # 60 130
0 6 # 95 120
0 6 # 100 100
0 7 # 71 0
0 9 | -1 0
1 0 country.b 2 7
1 0 # 150 150
1 0 # 200 250
1 0 # 225 267
1 0 # 250 300
1 0 # 250 215
1 0 # 150 125
1 0 # 150 150
1 1 # -1 7
1 1 # 150 150
1 1 # 180 250
1 1 # 225 267
1 1 # 250 300
1 1 #   250 215
1 1 #   150 125
1 1 #   150 150
1 2 #   -1 7
1 2 #   150 150
1 2 #   160 250
1 2 #   225 267
1 2 #   250 300
1 2 #   250 215
1 2 #   150 125
1 2 #   150 150
1 3 #   -1 7
1 3 #   150 150
1 3 #   140 250
1 3 #   225 267
1 3 #   250 300
1 3 #   250 215
1 3 #   150 125
1 3 #   150 150
1 4 #   -1 7
1 4 #   150 150
1 4 #   140 250
1 4 #   240 310
1 4 #   250 300
1 4 #   250 215
1 4 #   150 125
1 4 #   150 150
1 5 #   52 0
1 6 #   -1 7
1 6 #   150 150
1 6 #   130 260
1 6 #   245 315
1 6 #   250 305
1 6 #   250 235
1 6 #   150 125
1 6 #   150 150
2 0 country.c 3 7
2 0 #   150 125
2 0 #   150 90
2 0 #   165 80
2 0 #   170 85
2 0 #   180 110
2 0 #   190 125
2 0 #   150 125
2 6 country.c6 63 0
2 8 #   83 0
2 9 #   93 0
3 0 country.d 4 5
3 0 #   150 125
3 0 #   200 125
3 0 #   270 175
3 0 #   250 215
3 0 # 150 125
3 7 country.d7 74 7
3 7 # 150 125
3 7 # 200 125
3 7 # 250 150
3 7 # 255 200
3 7 # 280 235
3 7 # 250 235
3 7 # 150 125
4 4 country.e4 44 7
4 4 # 100 350
4 4 # 120 360
4 4 # 140 350
4 4 # 140 320
4 4 # 120 300
4 4 # 100 320
4 4 # 100 350
4 7 # 74 7
4 7 # 100 350
4 7 # 120 360
4 7 # 160 350
4 7 # 160 320
4 7 # 120 300
4 7 # 100 320
4 7 # 100 350
5 0 country.f 6 12
5 0 # 300 300
5 0 # 310 302
5 0 # 320 305
5 0 # 330 309
5 0 # 340 315
5 0 # 340 321
5 0 # 350 328
5 0 # 348 200
5 0 # 340 210
5 0 # 330 230
5 0 # 320 260
5 0 # 300 300
5 2 country.f2 26 0
5 4 # 46 12
5 4 # 300 300
5 4 # 310 305
5 4 # 320 308
5 4 # 330 312
5 4 # 340 318
5 4 # 340 324
5 4 # 350 331
5 4 # 348 197
5 4 # 340 207
5 4 # 330 227
5 4 # 320 257
5 4 # 300 300
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>66</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>300</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>310</td>
<td>309</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>320</td>
<td>312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>330</td>
<td>316</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>340</td>
<td>322</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>340</td>
<td>328</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>350</td>
<td>335</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>348</td>
<td>193</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>340</td>
<td>203</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>330</td>
<td>223</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>320</td>
<td>253</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>#</td>
<td>300</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>country.g4</td>
<td>44</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>#</td>
<td>400</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>#</td>
<td>420</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>#</td>
<td>420</td>
<td>320</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>#</td>
<td>430</td>
<td>330</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>#</td>
<td>430</td>
<td>290</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>#</td>
<td>400</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>country.g6</td>
<td>64</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>#</td>
<td>390</td>
<td>290</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>#</td>
<td>420</td>
<td>295</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>#</td>
<td>420</td>
<td>320</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>#</td>
<td>440</td>
<td>330</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>#</td>
<td>420</td>
<td>290</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>#</td>
<td>390</td>
<td>290</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td></td>
<td>-1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>country.h5</td>
<td>58</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>348</td>
<td>196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>300</td>
<td>205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>300</td>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>304</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>308</td>
<td>130</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>311</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>313</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>315</td>
<td>126</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>317</td>
<td>138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>333</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>333</td>
<td>185</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>#</td>
<td>348</td>
<td>196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>country.h5</td>
<td>58</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>348</td>
<td>196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>300</td>
<td>205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>256</td>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>251</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>262</td>
<td>130</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>278</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>313</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>315</td>
<td>126</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>317</td>
<td>138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td>333</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>country.il</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>country.i3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
References


