AN OBJECT-ORIENTED GRAPHIC USER INTERFACE FOR VISUALIZATION OF B-TREES' ANIMATOR

By

BETTY H. LIN

Bachelor of Science

Harbin Medical University

Harbin, P. R. China

1984

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1997

AN OBJECT-ORIENTED GRAPHIC USER INTERFACE

FOR VISUALIZATION OF B-TREES' ANIMATOR

Thesis Approved:

J Chandler

Thesis Advisor

[illegible]

Jacques E. LaFrance

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. John P. Chandler for all his support, help, and invaluable comments to keep my thesis research on course. My most heartfelt thanks go to Dr. Jacques LaFrance. Without his guidance, creative ideas, and informative suggestions, this ultimate step would have been impossible. My special thanks go to Dr. George Hedrick for serving on my thesis committee as well as for his time, patience, and constructive comments throughout the time I have been working on this project I would also like to express my sincere thanks and gratitude to Dr. Nick Street for his time, help and support to make my thesis defense on time.

I extend my sincere thanks to my husband, Huiyao Lin, for his love, understanding, and support, and also to my parents, Zhenbing Hou and Shuhua Mao, my parents-in-law, Kuanpo and Changtsaimei Lin, my sister and brother-in-law, Lee and Ray Harvick, for their support, motivation, and continuous encouragement.

Finally, I would like to thank the Department of Computer Science at Oklahoma State University for providing the facilities and resources for my study at the past two years

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

B-trees were invented by Bayer and McCreight in 1972 [Bayer72]. A B-tree is a data structure that maintains an ordered set of data and allows efficient operations to find, delete, insert, and browse the data. There are widely uses of B-trees in a database system as pointed out by D. Comer [Comer79] : "While no single scheme can be optimum for all applications, the technique of organizing a file and its index called the B-tree is, de facto, the standard organization for indexes in a database system."

I. 1. Objective of the Thesis

This thesis project uses Microsoft Visual C++ to implement data structure animation. In this thesis project, a class hierarchy for members of the B-tree family is built. These members are B-tree, B*-tree and B^+-tree. The trees and the dynamic movements as they are being built or destroyed are shown graphically.

The goal of this thesis is:

- To use object-oriented programming to make effective use of classes organized into a hierarchical structure based on the concept of inheritance.
- To provide a GUI to enable the user to observe a graphical representation of the B-tree family's data structures as they are being built or destroyed.

Since this project is to present a visualization of the trees, Microsoft Windows 95 is chosen as the environment to develop the program.

I. 2. The organization of the thesis

The thesis provides the related topics and concepts of the research. such as The problem of accessing secondary storage, A quick review of B-trees and their variants, Object-oriented programming concepts, Windows 95 overview, and Basic technology of animation, etc. The thesis also gives the detail discussion of program design and implementation, and program testing and evaluation.

Associated with a large, randomly accessed file in a computer system is an index. The fundamental problem with keeping an index on secondary storage is that accessing secondary storage is slow. Chapter II gives the brief discussion about the problem of accessing secondary storage and the solutions which intend to solve the problem.

B-trees and B-tree variants have been used for maintaining large index files since their first presentation by Bayer and McCreight in 1972 [Bayer72]. The versatility of B-trees is the reason they are applications in database programs from mainframe packages (such as IBM's VSAM) to PC products (such as dBase and its competitors, or the database facility in OS/2 Extended Edition). There have been a number of variations in both data structures and algorithms ever since the invention of B-trees. They efficiently support the two types of access to data elements:

- Random access to an arbitrary data record
- Sequential processing of data records in key sequence.

Chapter III presents the literature review of several variations of the original idea of B-trees. A good survey is given by Douglas Comer [Comer79].

Object-oriented programming is a new way of approaching the job of programming, and is based on the concept of an object. Object-oriented programming is a technique that facilitates code reuse. Inheritance, polymorphism, and encapsulation are examples of its important features, which are defined in Chapter IV. Windows 95 is a very large, complex programming environment. It is part of the next generation of operating systems intended to operate PCs well into the next century. A brief picture of Windows 95 is also presented in Chapter IV. In the same Chapter, the basic technology of animation is discussed briefly.

For the rest of the thesis, the design and implementation issues are discussed in detail in Chapter V. Program testing is included in Chapter VI and Chapter VII contains the summary and some possible areas of future work.

II. THE PROBLEM OF ACCESSING SECONDARY STORAGE

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called file. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient [Comer79].

Associated with a large, randomly accessed file in a computer system is an index that speeds retrieval by directing the searcher to the small part of the file containing the desired item.

The fundamental problem with keeping an index on secondary storage, such as disk or tape, is that accessing secondary storage is slow because of its physical characteristics. If searching on a tape, the elements can only be accessed sequentially. And if it is on a disk, the delay is still required to spin the disk and move the disk head. Binary searching requires too many seeks. Searching for a key on a disk often involves seeking on different disk tracks. Since seeks are expensive, a search that must look in more than three or four locations before finding the key often requires more time than is desirable. For a completely balanced tree, the worst-case search to find a key, given N possible keys, looks at $\log_2 (N + 1)$ keys. So if we are using a binary search, an average of about 9.5 seeks is required to find a key in an index of 1,000 items. Although there is a potential solution to the searching problem, that is by dividing a binary tree into pages and then storing each page in a block of contiguous locations on disk. In that way, we should be able to reduce the number of seeks associated with any search. Figure 1 illustrates such a paged tree.

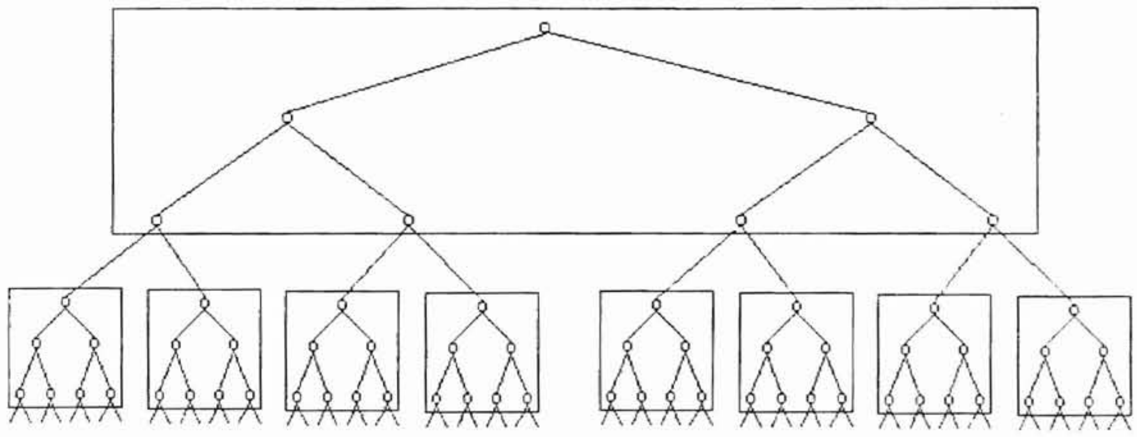


Figure 1. Paged binary tree.

In this tree we are able to locate any one of the 63 nodes in the tree with no more than two disk accesses. The number of seeks required for the paged versions of a completely full, balanced tree is $\log_{k+1}(N+1)$, where N is the number of keys, k is the number of keys held in a single page. It is the logarithmic effect of the page size that makes the impact of paging so dramatic:

$\log_{2}(134{,}217{,}727 + 1) = 27$ seeks

$\log_{511+1}(134{,}217{,}727 + 1) = 3$ seeks.

From the formulas showing above, we can see that breaking a tree into pages is a strategy that is well suited to the physical characteristics of secondary storage devices such as disk. The problem is how to build the tree once we decide to implement a paged tree. If we have the entire set of keys in hand before the tree is built, the solution is to sort the list of keys then build the tree from this sorted list. If we plan to start building the tree from the root, we know where to begin and are assured that this beginning point will divide the set of keys in a balanced manner. There will be a potential problem if we are receiving keys in random order and inserting them as soon as we receive them. When we start from the root, the initial keys must go into the root. For example, suppose we receive the following sequence of single-letter keys to build a paged tree:

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

Suppose we will build a paged binary tree that contains a maximum of three keys per page. As we insert the keys, we rotate them within a page as necessary to keep each page as balanced as possible. The resulting tree is illustrated in Figure 2. Since the keys are received in random order and inserted as soon as they arrive, so the initial keys, C, S and D will go into the root. But at least two of these keys, C and D, are not keys that we want in the root page. They are adjacent in sequence and tend toward the beginning of the total set of keys. Consequently, they force the tree out of balance. This is the problem caused by the top-down construction of paged trees.

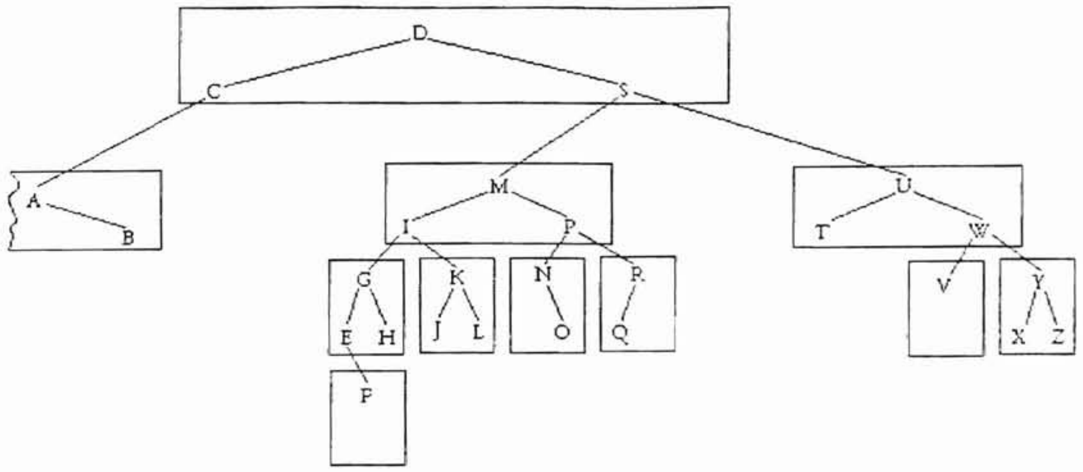


Figure 2. Paged tree constructed from keys arriving in random input sequence.

Bayer and McCreight's B-tree provides a solution directed precisely toward this problem. A B-tree is built upward from the bottom (leaves) instead of downward from the top (root). Bayer and McCreight recognized that the decision to work down from the root was, of itself, the problem. Rather than finding ways to undo a bad situation, they decided to avoid the difficulty altogether. With B-trees, we allow the root to emerge, rather than set it up and then find ways to change it [Folk92].

III. B-TREES AND THEIR VARIANTS

III. 1. B-trees

B-trees were discovered by Bayer and McCreight in 1972 [Bayer72] and are a natural evolution of earlier database designs.

A B-tree is a shallow tree structure that allows to store and retrieve a set of data records based on the keys. The shallow structure minimizes the number of disk seek required to access data records.

Unlike an arbitrary binary search tree, a B-tree is a balanced tree structure. Every leaf is at the same distance from the root. In a B-tree, one tree node can be made to correspond to a page. Because binary nodes are usually not large enough to take up one page, a B-tree node stores multiple keys and branches. A beneficial side effect of such multi-way nodes is that the height of the tree can be smaller than that of a binary tree. This property, of course, will lower the cost of find, insert, and delete operations.

III. 2. Properties of B-trees

B-trees are classified by their order, which refers to the maximum number of branches in a node. For example, in a B-tree of order 4, each node can have up to 4 branches. Corresponding to these branches are 3 keys that help determine which branch to take during a search. In general, a B-tree of order *m* has nodes with up to *m* branches and *m-1* keys. The order specifies the maximum number of branches. A node may have fewer branches than the maximum [Flamig93]. Figure 3 shows a B-tree of order 4, with all nodes full.

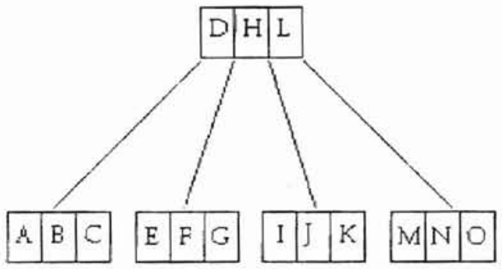


Figure 3. A B-tree of order 4.

Not all multi-way trees are B-trees. A multi-way tree is considered to be a B-tree only if it is balanced. In the classical definition of a B-tree of order *m*, the balance is achieved by maintaining the following properties:

- Except for the root node, all nodes must have at least $\lceil m/2 \rceil$ - 1keys and $\lceil m/2 \rceil$ branches. This means all nodes except the root are at least half full.
- All of the leaves of the tree are always on the same level.
- The root node has at least two children (unless it is a leaf).

III. 3. B-trees*

Knuth [Knuth73] defines a B-tree using a redistribution overflow technique to be a B*-tree. The insertion of a B*-tree employs a local redistribution scheme to delay splitting until two sibling nodes are full. Then the two nodes are divided into three, each 2/3 full (see Figure 6 for such a split) instead of just 1/2 full (see Figure 5). This scheme guarantees that storage utilization is at least 66%, while requiring only moderate adjustment of the maintenance algorithms. Note that increasing storage utilization has the side effect of speeding up the search since the height of the resulting tree is smaller.

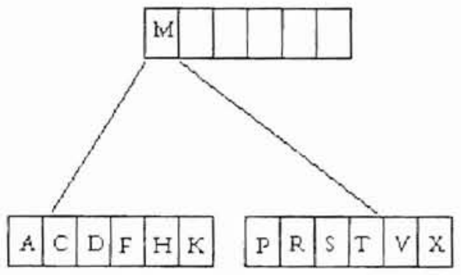


Figure 4. Original tree.

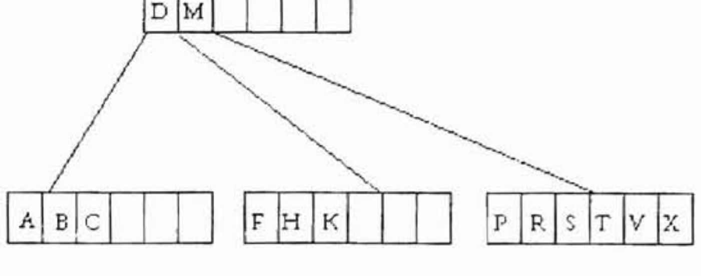


Figure 5. Two-way split: After the insertion of the key *B*.

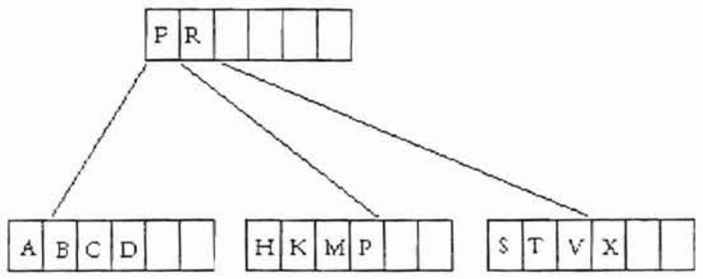


Figure 6. A two-to-three split: After the insertion of the key *B*.

III. 4. B^+-Trees

In a B^+-tree, all keys reside in the leaves. The upper levels, which are organized as a B-tree, consist only of an index, a roadmap to enable rapid location of the index and key parts [Comer79]. Figure 7 shows the logical separation of the index and key parts.

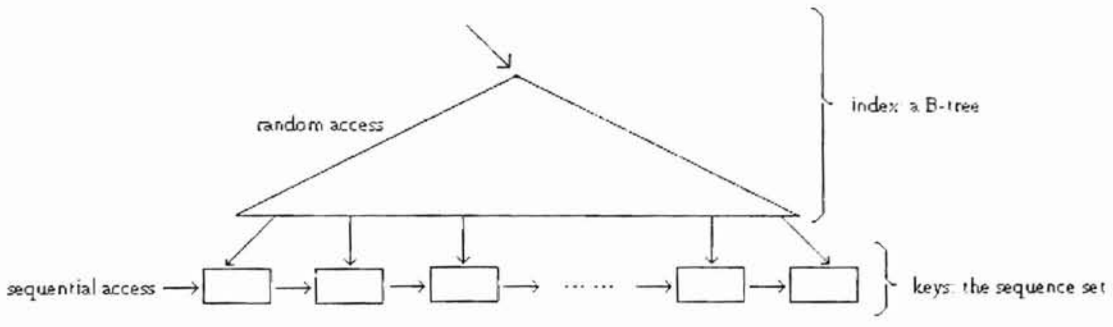


Figure 7. A B+-tree with separate index and key parts. Operations "by key" begin at the root as in a B-tree; sequential processing begins at the leftmost leaf.

Index nodes and leaf nodes may have different formats or even different sizes. In particular, leaf nodes usually are linked together left-to-right. The linked list of leaves is referred to as the sequence set. Sequence set links allow easy sequential processing [Comer79].

A beneficial side effect of having an independent index and sequence set is that it is well suited to applications that require both random and sequential processing.

III. 5. Other variants

III. 5. 1. Concurrent B-tree

Concurrent B-tree algorithms have been proposed for high-performance on-line transaction applications that allow concurrent accesses to B-trees. By concurrent accesses, we mean that many inserts/deletes/searches may occur during the same time interval. Many locking schemes are used in such cases, but all require exclusive locks on all nodes that are changed.

III. 5. 2. B^+-tree with partial expansion

A B^+-tree with partial expansion is based on the idea of gradually increasing the size of an overflowing bucket, instead of immediately splitting it. When the bucket reaches some maximum size, it is split in the normal way. The result showed by Baeza-Yates and Larson [Baeza-Yates89] research is that the storage utilization of B^+-trees with partial expansion is higher than standard B^+-trees.

IV. LITERATURE REVIEW

IV. 1. Object-Oriented Programming (OOP.)

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships [Booch94]. It is by the interaction of objects that computation proceeds [Budd91].

IV. 1.1 Objects

An *object* is an encapsulation of state (data values) and behavior (operations). An object is defined via its class, which determines everything about an object. Objects are individual instances of a class. All instances of the same class will behave in a similar fashion in response to a similar request. Terry Montlick gives the following definition for an object: "An object is a "black box" which receives and sends message*s*"[Montlick95]. As the user of an object, one should never need to peek inside the box. All communication to it is done via messages. An object will exhibit its behavior by invoking a method in response to a message. The interpretation of the message is decided by the object, and may differ from one class of objects to another.

IV. 1. 2. Classes

A *class* is a user-defined type which is just a collection of variables, often of different types, combined with a set of related functions. A class declaration specifies the representation of objects of the class and the set of operations that can be applied to such objects.

IV. 1. 3. Inheritance

Objects and classes extend the concept of abstract data types by adding the notion of *inheritance*. Classes can be organized into a hierarchical inheritance tree. Data and behavior associated with classes higher in the tree can also be accessed and used by classes lower in the tree. Such classes are said to inherit their behavior from the parent classes.

IV. 1. 4. Polymorphism

Another feature of Object-oriented programming is the ability of the objects to behave in different ways according to the message passed, and the ability to design operators to carry out object manipulation. This feature is called *polymorphism* and it can be either overriding or overloading.

IV. 1. 5. Encapsulation

Within an object, some of the code, functions, and/or data may be private to the object and inaccessible directly by anything outside the object. In this way, an object provides a significant level of protection against some other, unrelated part of the program accidentally modifying or incorrectly using the private parts of the object. The linkage of code and data in this way is often referred to as *encapsulation* [Schildt90].

IV. 2. Windows 95 Overview

Windows 95 was designed specifically to overcome several of the limitations imposed by its earlier incarnation: Windows 3.1 [Schildt95]. The most important characteristic of Windows 95 is that it is a 32-bit operating system. Unlike Windows 3.1, DOS, and other 8086-family operating systems, which use segmented memory, Windows 95 treats memory as though it were linear. In this new "Windows" world, each application has as much memory as it could possibly need.

Windows 95 was designed to be compatible with the large base of existing PC applications. Toward this end, Windows 95 can run four types of programs: those written for DOS, those written for Windows 3.1, those written for Windows NT, and those written specifically for Windows 95. Windows 95 automatically creates the right environment for the type of program people run. For example, when the user executes a

DOS program, Windows 95 automatically creates a windowed command prompt in which the program runs.

IV. 2. 1. The Graphical User Interface (GUI)

Windows is a graphical interface. A graphical interface is not only more attractive in appearance, but it can also impart a high level of information to the user.

Windows 95 is graphics-oriented, which means that it provides a Graphical User Interface (GUI). All graphical user interfaces make use of graphics on a bitmapped video display. Graphics provides better utilization of screen real estate, a visually rich environment for conveying information, and the possibility of a WYSIWYG (what you see is what you get) video display of graphics and formatted text prepared for a printed document [Petzold96].

In a graphical user interface, the video display shows various graphical objects in the form of icons and input devices such as buttons and scroll bars. Using the keyboard or mouse, the user can manipulate these objects directly on the screen. Graphics objects can be dragged, buttons can be pushed, and scroll bars can be scrolled.

With GUIs, the user can directly interact with the objects on the display instead of the one-way cycle of information from the keyboard to the program to the video display.

IV. 2. 2. The Multitasking Advantage

Under Windows, every program in effect becomes a RAM-resident popup. Several Windows programs can be displayed and running at the same time. Each program occupies a rectangular window on the screen. The user can move the windows around on the screen, change their size, switch among different programs, and transfer data from one program to another.

Earlier versions of Windows used a system of multitasking called "nonpreemptive". This meant that Windows did not use the system timer to allocate processing time among the various programs running under the system. The programs themselves had to give up control voluntarily so that other programs could run. Under Windows 95, multitasking is preemptive, and programs themselves can split into multiple threads of execution that seem to run concurrently.

IV. 2. 3. Memory Management

An operating system cannot implement multitasking without doing something about memory management. As new programs are started up and old ones terminate, memory can become fragmented. The system must be able to consolidate free memory space. This requires the system to move blocks of code and data in memory.

Even Windows 1, running on an 8088 microprocessor, was able to perform this type of memory management. Programs running under Windows can overcommit memory: a program can contain more code than can fit into memory at any one time. Windows can discard code from memory and later reload the code from the program's .EXE file. A user can run several copies, called "instances," of a program; all these instances share the same code in memory. Programs running in Windows can share routines located in other files called "dynamic link libraries." Windows includes a mechanism to link the program with the routines in the dynamic link libraries at run time. Windows itself is basically a set of dynamic link libraries.

Thus, even in Windows 1, the 640-kilobyte (KB) memory limit of the PC's architecture was effectively stretched without requiring any additional memory. Windows 2 gave the Windows applications access to expanded memory, and Windows 3 ran in protected mode to give Windows applications access to up to 16 MB of extended memory. And now Windows 95 blows these old restrictions away by being a full-fledged 32-bit operating system with a flat memory space [Petzold96].

IV. 2. 4. The Device-Independent Graphics Interface

Windows programs can make full use of graphics and formatted text on both the video display and the printer. Programs written for Windows do not directly access the hardware of graphics display devices such as the screen and printer. Instead, Windows includes a graphics programming language, called the Graphics Device Interface, or GDI, that makes it easy to display graphics and formatted text. Windows virtualizes display hardware. A program written for Windows will run with any video board or any printer for which a Windows device driver is available. The program does not need to determine what type of device is attached to the system.

Putting a device-independent graphics interface on the IBM PC was not an easy job for the developers of Windows. The PC design was based on the principle of open architecture. Third-party hardware manufacturers were encouraged to develop peripherals for the PC. Although several standards have emerged, conventional MS-DOS programs for the PC must individually support many different hardware configurations. For example, it is fairly common for an MS-DOS word-processing program to be sold with one or two disks of small files, each one supporting a particular printer. Windows 95 programs do not require these drivers because their support is part of Windows [Petzold96].

IV. 2. 5. Object-Oriented Programming

Windows are rectangular areas on the screen. A window receives user input from the keyboard or the mouse and displays graphical output on its surface. Windows programming is a type of object-oriented programming (OOP). In object-oriented programming, an "object" is a combinations of code and data. A window is an object. The code is the window procedure. The data is information retained by the window procedure and information retained by Windows for each window and window class that exists in the system.

An application window usually contains the program's title bar, menu, sizing border, and perhaps some scroll bars. Dialog boxes are additional windows. Moreover, the surface of a dialog box always contains several additional windows called "child windows." These child windows take the form of push buttons, radio buttons, check boxes, text entry fields, list boxes, and scroll bars.

The user sees these windows as objects on the screen and interacts directly with these objects by pushing a button or scrolling a scroll bar. Interestingly enough, the programmer's perspective is analogous to the user's perspective. The window receives this user input in the form of "messages" to the window. A window also uses messages to communicate with other windows.

IV. 2. 6. Message-Driven Architecture

In Windows, when a user resizes a window, Windows sends a message to the program indicating the new window size. The program can then adjust the contents of its window to reflect the new size. So "Windows sends a message to the program" means Windows calls a function within the program. The parameters to this function describe the particular message. This function located in the Windows program is known as the "window procedure." A window procedure processes messages to the window. Very often these messages inform a window of user input from the keyboard or the mouse. This is how a push-button window knows that it's being "pressed," for example. Other messages tell a window when it is being resized or when the surface of the window needs to be redrawn.

When a Windows program begins execution, Windows creates a "message queue" for the program. This message queue stores messages to all the various windows a program might create. The program includes a short chunk of code called the "message loop" to retrieve these messages from the queue and dispatch them to the appropriate window procedure. Other messages are sent directly to the window procedure without being placed in the message queue.

IV. 2. 7. The Window Procedure

Programs written in the traditional way call the operating system. However, Windows 95 generally works in the opposite way — the operating system making calls to the program. This is fundamental to Windows 95's object-oriented architecture.

Every window that a program creates has an associated window procedure. This window procedure is a function that could be either in the program itself or in a dynamic link library. Windows sends a message to a window by calling the window procedure. The window procedure does some processing based on the message and then returns control to Windows.

More precisely, a window is always created based on a "window class." The window class identifies the window procedure that processes messages to the window. The use of a window class allows multiple windows to be based on the same window class and hence use the same window procedure. For example, all buttons in all Windows programs are based on the same window class. This window class is associated with a window procedure (located in a Windows dynamic link library) that processes messages to all the button windows.

IV. 3. Basic Technology of Animation

Animation is effective way to communicate information. For example, it can illustrate the operation of a particular tool or reflect a particular state. It also can be used to include an element of fun in user's interface. One can use animation effects for objects within a window and interface elements, such as icons, buttons, and pointers.

Effective animation involves many of the same design considerations as other graphics elements, particularly with respect to color and sound. Fluid animation requires presenting images at 16 (or more) frames per second.

There are two basic types of animation on a computer: cast-based animation, which makes use of sprites, and frame-based animation, which operates more like a movie, consisting of a series of fixed images played in sequence. Cast-based animation is more interesting as a programming topic because it calls for dealing with concepts such as transparency and may involve interaction with the user. You can convert a cast-based animation into a frame-based animation by shooting each step in each scene and connecting the snapshots as a sequence of frames in a single file.

In this project, the image objects that we deal with are many identical rectangular boxes with different character strings printing inside the boxes. Whenever an animating operation occurs, the program moves only one node or one rectangular each time. It is not necessary to repaint the whole screen since it is very time-consuming. In order to make image displayed more effectively, only the moving node and few related nodes need to be repainted. Therefore, cast-based animation is selected to demonstrate the motion of moving B-tree nodes during insertion or deletion operations.

V. DESIGN AND IMPLEMENTATION ISSUES

The main focus of this thesis is the implementation of data structure animation. The project, which is written in Microsoft Visual C++, graphically shows the trees and the motion of a page's splitting and concatenating due to insertion and deletion operations. In this thesis project, a class hierarchy for members of the B-tree family is built by using the object-oriented approach. These members are B-tree, B*-tree and B^+-tree. The project provides a GUI to enable the user to observe a graphical representation of the B-tree family's data structures as they are being built or destroyed. Since this project is to present a visualization of the trees, Microsoft Windows 95 is chosen as the environment to develop the program.

V. 1. Main Elements of the Project

V. 1. 1. The File-Based Objects — Fmgr Class

Since B-trees and B-tree variants have been used for maintaining large files, this project is designed to implement data structures that reside in files. At the heart of the file-based object design in this project is the *Fmgr* class that manages objects stored in files. It has two functions, Alloc() and Free(), which are analogous to the C functions malloc() and free(). In addition, the *Fmgr* class has Fetch() and Store() functions that read and write objects. The following example creates an *Fmgr* file and stores a Part object in it:

```
#include <iostream.h>
#include "fmgr.h"

struct Part {
int id;
float price;
Part(int I = 0,  float p = 0) {id = I,  price = p;}
};

main()
{
        Part part(17,  42.0);       // Memory buffer of part to be stored.
        FmgrPtr f(new Fmgr);  // Should always create dynamically.
        f-> Create("test.dat");    // Create and open file.

        long addr = f->Alloc(sizeof(Part));   // Allocate room for part.
        f->Store(&part,  sizeof(Part),  addr);

        // Close and reopen file for testing.
        f->Open("test.dat");   // Open() closes first.

        // See if we can get the part back.
        f->Fetch(&part,  sizeof(Part),  addr);
        cout << "Part: <" << part.id << ", " << part.price << ">\n";

        // Now delete the part.
        f->Free(sizeof(Part),  addr);

        return 0;  // File automatically closed by destructor.
}
```

Although the *Fmg*r class is fairly sophisticated, it is a low-level design. The evidence of the low-level nature of *Fmgr* can be found in functions like Alloc(), Fetch(), Store(), and Free(). These functions must be told the size of the objects being used. The detail information about the other member functions of the *Fmg*r class can be found in Appendix B.

V. 1. 2. Entry class

The *Entry* class contains a pair of key and RRN, Relative Record Number. along with its associated data which is ignored in this project implementation. Most functions for the *Entry* class, which are described in Appendix B, are straightforward. Only two of the functions, Compare() and Fullcompare(), need some explanation:

```
int Compare(const Entry &a,  const Entry &b)
{
        return strcmp(a.key.  b.key);
}


int FullCompare(const Entry &a.  const Entry &b)
{
        int rv = strcmp(a.key.  b.key);
        if (rv  >  0)   return  1:
        if (rv  <  0)   return  -1:
        if (a.data  >  b.data)   return  1;
        if (a.data  <  b.data)   return  -1:
        return  0;
}
```

The Compare() functions is used when searching for data based on a key. The idea here is that once an entry is found, the data field can be used to retrieve the data for the entry. However, when deleting an entry, the data field should already be known, so the FullCompare() function, which treats the data field as a secondary key, is used to find the specific entry to delete. Thus, duplicate keys are supported, but each key and data pair must be unique.

V. 1. 3. The Multi-Way Nodes — Page Class

The main feature of B-trees is the multi-way nodes that make up the trees. Multi-way nodes are essentially generalizations of binary nodes. Since there is always one more branch than keys in a multi-way node, the *Page* class is designed to have a left branch, a pointer to left most child, that leads to all nodes with keys smaller than the smallest key in the given node. The other *right* branches will be paired with a key. A key, along with its associated data and right branch field, is defined in the *Entry* class.

An array of entries is packaged into a multi-way node which is defined as *Page* class. In this class, ORDER is the maximum number of branches possible for the node. Since there is one less key than branch, ORDER - 1 entries are reserved. The *cnt* field indicates how may entries are actually in use. The extra branch, *left*, is placed immediately before the entries, as illustrated in Figure 8.

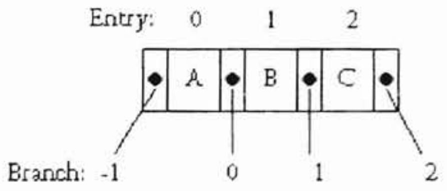


Figure 8. Layout of an Page node of order 4
(The cnt and data fields are not shown).

This arrangement allows us to index the branches from -1 to ORDER - 2, where the -1st branch represents the *left* branch, the 0th branch is the right branch of the first entry, and so on.

The *Page* class has functions to support searching for keys in a node, and for inserting entries into and deleting entries from a node. The functions of the *Page* class are also fairly straightforward, and they are described in Appendix B. Only the Search() function and FullSearch() function need to give some explanation. The Search() function sequentially scans the entries of the node, looking for a match. The FullSearch() function can be defined by simply replacing the call to Compare() with a call to FullCompare(). The code for these two functions are as following:

```
int Page::Search(const Entry &e, t &posn)
{
        posn = cnt - 1;

        while (posn >= 0)
                {
                int rv = Compare(e, entry[posn]);
                if (rv > 0) return 1;
                if (rv == 0) return 0;
                posn--;
                }
                return -1;
}


int Page::FullSearch(const Entry &e, int &posn)
// like the Search(), except we use FullCompare().
{
        posn = cnt - 1;

        while (posn >= 0)
                {
                int rv = FullCompare(e, entry[posn]);
                if (rv > 0) return 1;
                if (rv == 0) return 0;
                posn--;
                }
                return -1;
}
```

V. 1. 4. The Btree Class

In this project, the *Btree* class is the base class of the *Bstar* class and the *Bplus* class. Since the *Btree* class is file based, it has numerous file-management functions, such as Connect(), Disconnect(), Create(), Open(), Close(), and Flush(), among others. These functions allow multiple B-trees to be stored in a single file. Each B-tree has a header, defined in *BtreeHeader* which is a type of *struct*. The *BtreeHeade*r points to the root node of the tree, and stores some other pertinent data used mostly for testing. The headers are meant to be stored in the static data area of the file. The information for other member functions of this class is given in Appendix B.

V. 1. 5. The Bstar Class

The *Bstar* class is derived from the *Btree* class. This class inherits all the functions of the *Btree* class except Insert() function. The insertion into a B*-tree employs a local redistribution scheme to delay splitting until two sibling nodes are full, then the two nodes are divided into three, each 2/3 full. So the Insert() function here is overriding the Insert() function in the base class. The other two functions, FindParent() and SetParameters(), in the *Bstar* class are involved in redistribution of the tree structure. The *Bsta*r class and its associated data structures are used to set up a B^*-tree.

V. 1. 6. The Bplus Class

The *Bplus* class is derived from the *Btree* class too. Because of the nature of a B^+-tree, it inherits all the functions of the *Btree* class except the functions such as Insert(), Search(), FullSearch(), and Remove(). These functions overrides the member functions in the base class. The *Bplus* class also overloads the Delete() function of the base class. The *Bplus* class and its associated data structures are used to set up a B^+-tree.

V. 1. 7. The WNode Class

The primary purpose of creating this class is to link the B-tree data structure and the Windows 95 graphics data structure together. The *Wnode* class serves as the carrier that carries the data generated by the B-tree classes and gives the image output to the GUI window. Another purpose of this class is to provide animation mechanism to show the detail transition of B-trees' insertion and deletion dynamically. The image objects, RBoxes and KBoxes, of this class are declared in RECT structure which is defined in WINDOWS.H. These two data members provide coordinate information to Windows 95 for drawing the object image boundary. The other two important data members are chRRN, which is RRN of the child node, and Keys which are keys in the node. Both data members receive data from B-tree modules. This class provides InitеWNode() to initialize all the data members. The functions of MoveLeft(), MoveRight(), MoveUp(), and MoveDown() provide the basic animation mechanism and they are called by ShowNode() which is the function that animates image objects of this class. The

SetKey(), SetchRRN(), and Updated() are the methods to port B-trees' data into the *Wnode* class. The *Wnode* class also furnishs many output methods for program to achieve animation. They are Print(), EraseNode(), ShowNode(), PrintNode(), MoveParentSb(), and MoveTree(). There are many supporting functions that provide methods to manipulate private data members, such as GetChild(), DelChild(), FindChildren(), MoveChildren(), InsChild(), SetParent(), GetParent(), Set Position(), GetPosition(), AddSibling(), GetSibling(), CopySiblings(), SetLeftSb(),GetLeftSb(), and UpdateLSb(). The detail information on each class and its member functions is given in Appendix B.

V. 2. Object-Oriented Approach

Inheritance, encapsulation, and polymorphism are the important features of object-oriented programming. The encapsulation is also useful in conventional languages, such as using structures inside structures. But it is more important in object-oriented languages because of the natures of objects themselves. It is practical to built new objects from different simpler objects with different behaviors. The unique behavior of each new object will be partially based on the result of its components' behaviors.

As mentioned before, B*-tree and B^+-tree are the variants of the B-tree. A B-tree is a B*-tree if each its node is at least 2/3 full instead of just 1/2 full. This results from the B*-tree insertion operation. B*-tree insertion employs a local redistribution scheme to delay splitting until two sibling nodes are full. A B^+-tree is derived from B-tree. In a B^+-tree, the upper levels are organized as a B-tree which consists of an index. All the keys reside in the leaves. In other words, a B^+-tree maintains an independent index and sequence set. According to these characteristics of B-trees data structure, during the program design and implementation, we designed a class called Btree as a base class. And the other two classes, Bstar and Bplus, are derived from the Btree class. They inherit all the functions of the Btree class except the Insert(), Delete(), Search(), FullSearch(), and Remove() functions. These functions either override or overload the functions of the base class with different behaviors of the objects.

V. 3. Animation Approach

In this project, we are basically dealing with the tree nodes' movement to achieve the animation. There are only two events, split and merge, in B-trees' operations that will cause the tree nodes to rearrange and move to new locations. The analysis of tree nodes' behavior during the splitting and merging is the key step to see if the animation is achieved successfully. According to the behavior analysis, there are four different categories of the tree nodes in splitting and merging which are shown as following:

- Leaf nodes:

  Split: The leaf nodes that have the same parent node are arranged right under the parent node one by one with a fixed gap along the y - axis. In the other words, they have the same x coordinate value and it means that the movement of the leaf nodes will be only along the y - axis during leaf nodes splitting. During the splitting, the new node needs to be moved down along the y - axis with a fixed distance which is the node height plus the gap. Before the new node is moved, its right sibling nodes will be moved down first in the same distance. This is implemented by the member function, PrintNode(), of Wnode class.

Merge: This is done by reversing the procedure of the split operation.

Other movement: There are two situations that make leaf nodes move. Whenever the leaf nodes' parent is moved, leaf nodes have to be moved as well. In this case, we use the parent's new x and y coordinate values as references to determine the movement on both x and y axis direction. In the program design, we always move the node along the x direction first and then the y direction. This is implemented by the member function, MoveChildren(), of Wnode class.

- Leaf parent nodes (the nodes' children are leave):

Split: This kind of nodes will only move along the x - axis and they will affect all the nodes on their right hand side. During the splitting, the new node will be moved to the right in the distance of node length plus gap. Before the new node is moved, all its right sibling tree, its parents' right sibling tree, its grand parent's right sibling tree, its grand grand parent's right sibling tree, and so on will be moved first to the right in the same distance. These are done by the member functions, MoveParentSb() and MoveTree(), of Wnode class. After all the non-leaf nodes are moved, the MoveChildren() of Wnode class is called to move all the leaf nodes, whose parent nodes have been moved during this rearrangement, to the proper locations.

Merge: This is done by reversing the procedure of the split operation.

- Normal nodes (the nodes' children are non-leaf nodes):

  Split: This kind of nodes also only moves along the x - axis and they do not affect any other nodes during the movement. Since all the movements are done when the leaf parent node is being split. When the splitting propagates to the normal node, all the nodes are in the right positions and the only one that is not in the place is the new node. This is simply implemented by the member function, PrintNode(), of Wnode class.

  Merge: This is simply done by reversing the procedure of the split operation.

- Root node:

  Promotion: When the root node is split, a new root is created and it is moved up along the y - axis in the distance of node height plus gap with the same x coordinate value as the old root node. This operation will not affect any other nodes.

  Demotion: This is done by reversing the procedure of the promotion operation

  Split: It is the same as normal node's split.

VI. PROGRAM TESTING

VI. 1. Objective

There are *five phases* in the software life cycle. The *first phase* is the analysis which is to develop specifications describing the project and its requirements. The *second phase* is software design which is to construct a relatively detailed design plan according to the specifications. The *third phase* is coding that includes the writing of programs and the insertion of explanatory remarks into programs. The *fourth phase* is testing to ensure that programs are functional working well. The *fifth phase* is program maintenance that is to continue implementing the features that users request later or fixing problems. The purpose of this section is to map out the test plan and its strategy of the fourth phase, testing and design verification. The strategy can be divided into two parts, design verification and functional testing.

- Design Verification Objective

  The objective of design verification is to ensure that this program is implemented according to the specification that is defined in the phase II -- top_down design.

- Functional Testing Objective

  The objective of functional testing is to ensure that all the features of this software are working as intended.

VI. 2. Test Methodology

The test methodology for this software has two parts: part I relates to the design verification that is to go through each module and each class which are defined in the design phase; and part II is primarily concerned with functionality of each program feature and ease of use of the user interface.

VI. 2. 1. Part I : Design Verification

- Program Modules Verification

  In this stage, each program module is revisited and the job is to remove any unused variables, functions, and statements as well as add more comments to wherever is needed. This ensures that the module does not contain redundant codes and variables and also improves the readability of the program.

- Classes Verification

  This stage reviews the relationship among classes along with their inheritance hierarchy. The intention is to confirm the design with original definition.

VI. 2. 2. Part II : Functional Testing

- GUI User Interface Testing

1. File Menu

The file menu provides an interface for users to specify a file name to open the file and contains the data generated by the program. This test verifies if the file open dialog box is popping up when the user click the menu item "*File*". The test also checks if there any unusual data appears in the dialog box.

2. Option Menu

The option menu gives users a list of three choices to specify a type of B-tree variant. The testing of option menu includes menu selection and image output verification. When the user selects one of three tree types, a check mark (√) should appear in its left hand side. This can be confirmed visually by popping up the option menu again. The image output generated by the selected tree can be inspected by checking its splitting position for the choice of B-tree or B^*-tree to see if it splits in the correct position. For the choice of B^+-tree, the image output verification can be done by checking its separators to see if the separators appear in the expected position of the index set.

3. Quit Menu

The quit menu lets user to exit the program. This step examines if the program ends when the user click this menu item.

4. Help Menu

This menu item instructs WINHELP.EXE to open WBTREE.HLP when the user selects it. The testing focuses on the readability and organization of the help file.

5. Re-sizing and Moving

Since there may be some other programs coexisting with this program,so the window of this program can be minimized, maximized, moved, or re-sized. These operations potentially may result in the image lost in the client area. This test is to ensure that the client area is properly repainted whenever the above situations occur.

6. Animation

This step examines the moving of each object image to assure that the path is within the expectation.

- Data Handling Testing

In this stage, random number and size of data are input to the index file by using the selected tree type data structure. The insertion and deletion are also randomly performed to inspect the dynamic stability of this program. The verification of data handling is based on the specifications of three tree types.

1. B-Tree

Insertion: The testing verifies if the key inserted by the user goes to the proper object image as well as data integrity. This test also applies to the other two tree types.

Deletion: Redistribution and concatenation are key operations to evaluate deletion procedure. The testing examines these two operations to see if they perform the way as expected.

Splitting: This tests if the B-tree splits from the middle position.

2. B^*-tree

Insertion: The testing verifies if the key inserted by the user goes to the proper object image as well as data integrity. The result should be defferent from the one of B-tree insertion, due to the property of B^*-tree that it does not split until its siblings are also full. In this case, the redistribution occurs during the operation of insertion.

Deletion: This test is same as the test performed on the B-tree.

Splitting: This tests if the B^*-tree splits from the two third position.

3. B$^+$-tree

Insertion: One important property of B$^+$-tree is that it maintains an index set containing separators. The index set is in B-tree format. All the keys are inserted into the sequence set (or leaf). The test inspects if the separators are created properly.

Deletion: Since the key to be deleted must always reside in a leaf. As long as the leaf remains at least half full, the index need not be changed. even if a copy of the key (separator) had been propagated up into it. This test is to ensure that the B$^+$-tree's deletion preserves the separators.

Splitting: This test is same as the test performed on the B-tree.

VII. SUMMARY AND SUGGESTED FUTURE WORK

VII. 1. Summary

B-trees were invented by Bayer and McCreight in 1972. B-trees, or variations of them, have become the data structures of choice for database applications. B-trees allow fast database searching, due to the ability to optimally size the nodes to the paging requirements of a file system, and due to the relative flatness that results by using multi-way nodes [Flamig93]. Object-oriented programming is a new way of approaching the job of programming, and is based on the concept of an object. Object-oriented programming is a technique that facilitates code reuse. Inheritance, polymorphism, and encapsulation are examples of its important features. Animation is one effective way to communicate information. Effective animation involves many of the same design considerations as other graphics elements. There are two basic types of animation on a computer: cast-based animation, which makes use of sprites, and frame-based animation, which operates more like a movie, consisting of a series of fixed images played in sequence. Cast-based animation is more interesting as a programming topic because it uses concepts such as transparency and may involve interaction with the user.

This thesis project is implemented by using Microsoft Visual C++. In this project, an animator of B-trees is created by employing the cast-based animation technique to demonstrate the motion of moving B-trees' nodes during insertion or deletion operations. A class hierarchy for members of the B-tree family is built. These members are B-tree,

B^*-tree and B^+-tree. The trees and the dynamic movements as they are being built or destroyed are shown graphically. The thesis project is designed by using object-oriented approach to make effective use of classes organized into a hierarchical structure based on the concept of inheritance. Since B-trees and B-tree variants have been used for maintaining large files, this project is designed to implement data structures that reside in files. At the heart of the file-based object design in this project is the *Fmgr* class that manages objects stored in files. The *Btree* class is a base class of the *Bstar* class and the *Bplus* class. Because the *Btree* class is also file based, it has numerous file-management functions. The main feature of B-trees is the multi-way nodes that make up the trees. Since there is always one more branch than keys in a multi-way node, the *Page* class is designed to have a left branch that leads to all nodes with keys smaller than the smallest key in the given node. The other *right* branches will be paired with a key. So a key, along with its associated data and right branch field, is defined in the *Entry* class. The *Bstar* class is derived from the *Btree* class. This class inherits all the member functions of the *Btree* class except Insert() function. The *Bplus* class is derived from the *Btree* class too. Because of the nature of B^+-tree, it inherits all the member functions of the *Btree* class except the functions such as Insert(), Delete(), Search() and FullSearch(). Since the project also provides a GUI to enable the user to observe a graphical representation of the B-tree family's data structures as they are being built or destroyed, the *Wnode* class is created to link the B-tree data structures and the Windows 95 graphics data structures together. The *Wnode* class serves as the carrier that carries the data generated by the *B-tree* classes and gives the image output to the GUI window. Another purpose of this class is to provide animation mechanism to dynamically show the detail transition of B-trees' insertion and deletion. The program is designed to present a visualization of the trees and is, therefore, developed under Microsoft Windows 95 which is a very large, complex programming environment.

VII. 2. Future Work

The design that this project has presented for B-trees is a simple graphical demonstration with animation effects. There is ample space to extend this design to give more flexibility to the data structure, add more graphical technique, and improve the performance.

- Flexibility

There are several in the data structure are hard coded. In this design, the type of keys in the B-tree nodes is character string. This can be designed to accept various data type with flexible size. B-trees in this program has fixed order of five. And it can also be able to make it flexible to handle any order of the tree. All these can be implemented by utilizing template, function overloading, dynamically memory allocating techniques.

Another area that can be addressed for the future work is the key length. The key length in this design is fixed, and therefore, it could result in a lot of wasted space in the nodes when the keys are not in the maximum size. The approach is to allow variable-length keys in B-trees, where each key occupies only as much space as needed in a node. By using this strategy we can pack as many keys as possible into each node. This means the nodes can have a variable number of keys and branches. The result is a variable order of B-tree.

In such a tree, the size of the node is used as the criteria for minimum node size, rather than using the number of keys.

- Graphics

In this implementation, a node is formed in a set of two dimension rectangular boxes for which the program only draws the boundary. It will really impress users to create high-quality 3-D color images completed with shading, lighting, and other effects. Windows provides OpenGL API functions to deal with graphics primitives, matrix transformations, lighting, shading, coloring, texture mapping, and more. OpenGL is based on an industry standard that is maintained by an independent group called the Architectural Review Board (ARB) and is supported by a variety of platforms. The OpenGL API functions can be used to perform 3-D drawing and rendering for the future graphical implementation.

- Performance

Since a B-tree is a file-based data structure, it involves heavy file operations such as reading from and writing to a file. Furthermore, the file is usually stored on the disk or backup storage device which is always much slower than main storage in terms of the access time. To solve this problem, a cache mechanism can be added to the design to enhance performance. Rather than fetching pages from the file every time they are needed, the copies kept in main storage are accessed instead.

The animated images shown in this application rely solely on Windows graphical device interface (GDI) functions which could be too slow. To make the animated picture more smoothly, it is necessary to create its own set of bitmaps to improve performance and memory use.

BIBLIOGRAPHY

Baeza-Yates, R. A. & Larson, P. (1989). Performance of B^+-trees with Partial Expansions. IEEE Transactions on Knowledge and Data Engineering. Vol.1. No.2, 248-257.

Baeza-Yates, R. A. (1987). The Expected Behavior of B^+-trees. Technical Report CS-86-68. Dept. of Computer Science. University of Waterloo. Ontario. Canada.

Bayer, R. & McCreight, E. M. (1972). Organization and Maintenance of Large Ordered Indices. Acta Informatica. 1 (3), 173-189.

Budd, T. (1991). An Introduction to Object-Oriented Programming. Addison-Wesley.

Chu, J. H. & Knot, G. D. (1989). An Analysis of B-trees and Their Variants. Information systems, Vol.14, No.5, 359-370.

Comer, D. (1979). The Ubiquitous B-tree. Computing. Surveys, Vol. 11, No.2, 121-137.

Crotzer, A. D. (1975). Efficacy of B-trees in An Information Storage and Retrieval Environment. Unpublished Master's Thesis, OSU.

Davis, W. S. (1974). Empirical Behavior of B-trees. Unpublished Master's Thesis. OSU.

Eckel, G., Houlette, F., Stoddard, J. and Wagner, R. (1993). Inside Windows NT. New Riders Publishing.

Eisenbarth, B. Siviani, N. , Gonnet, G. H., Mehlhom. K. & Wood, D. (1982) The Theory of Fringe Analysis and Its Application to 2-3 Trees and B-trees. Inform. and Control 55 (1-3), 125-174.

Flaming, B. (1993). Practical Data Structures in C++. John Wiley & Sons, Inc.

Folk, M. J. & Soellick, B. (1992). File Structures. Second edition. Addison-Wesley.

Johnson, T. & Shasha, D. (1992). Reexamining B-trees. Dr. Dobb's Journal, Vol. 17, No. 1, 44-46.

Johnson, T. & Shasha. D. (1993). The Performance of Current B-tree Algorithms. ACM Transactions on Database Systems, Vol. 18, No. 1, 51-101.

Knuth, D. (1973). The Art of Computer Programming Vol. 3. Searching and Sorting. Reading, Mass.: Addison-Wesley.

Kuspert, K. (1983). Storage Utilization in B*+-trees With a Generalized Overflow Technique. Acta Informatica 19 (1), 35-55.

Leung, C. (1984). Approximate Storage Utilization of B-trees. A Simple Derivation and Generalizations. Inform. Process. Lett. 19, 199-201.

Lippman, S. (1991) C++ Primer. Second Edition. Addison-Wesley

Montlick, T. (1995). What Is Object-Oriented Software? Web site: http://www.soft-design.com/softinfo/objects.html.

Petzold, C. (1996). Programming Windows 95. Microsoft Press.

Quitzow, K. H. & Kloprogge, M. R. (1980). Space Utilization and Access Path Length in B-trees. Inform. Systems 5, 7-16.

Rodent, H. (1994). Animation in Win32. 1995 (January) - Microsoft Developer Network Library.

Rogerson, D. (1994). OpenGL I: Quick Start. Microsoft Developer Network Technology Group.
Schildt, H. (1995). Schildt's Windows 95 Programming in C and C++. McGraw-Hill, Inc.

Schildt, H. (1995). Using Turbo C++. McGraw-Hill, Inc.

Shasha, D., Lanin, V. & Schmidt, J. (1987). An Analytical Model for the Performance of Concurrent B-tree Algorithms. Ultracomputer Note 311, Dept. of Computer Science, New York Univ., New York.

Srinivasan,V. & Carey, M.(1991). Performance of B-tree Concurrency Control Algorithms. In proceedings of the 1991 ACM-SIGMoD International Conference on Management of Data, ACM, New York.425-461.

Stroustrup, B. (1995). The C++ Programming Language. Second Edition. Addison-Wesley.

Tamura, R., Belew, P., Blakely, J., Grantham, E., Griswold, R., Hall, W., Hipson, P., Kenner, B., Montemer, B., Parker, T., Thayer, J., Toth, V., Kottler, J., Woelfer, T., Harris, L., Laeremans, R., Lujan, P., Trujillo, S, Pietrocarlo, D., and Eman, O. (1995). Programming Windows 95. Sams Publishing.

Tello, E. (1991). Object-Oriented Programming for Windows. John Wiley & Sons, Inc.

Thompson, N. (1995). Animation Techniques in Win32. Microsoft Press

Wagner, W. (1992). The Windows Cartoon Engine Supplies High-Quality Animation to Any Application. 1995 (January) - Microsoft Developer Network Library.

Wright, W. E. (1985). Some Average Performance Measures for the B-tree. Acta Informatica 21 (1), 541-557.

APPENDIXES

Appendix A - Glossary

API API (Application Programming Interface) is a library of routines and services, each built from low-level operating system commands, which accomplish common tasks.

Cast-Based Animation Cast-based (or sprite-based) animation involves drawing an image on top of a background image. Sprites are generally used for game animation. To produce motion using sprites, one changes the location where the sprite is drawn.

DOS DOS is an acronym for Disk Operating System. It is a text-based, keyboard-oriented operating system. DOS was originally released in 1981 with the IBM PC, and it is still the de facto operating system for all IBM and compatible personal computers.

Frame-Based Animation Frame-based animation involves drawing a image on top of a background. Frame-based animation operates more like a movie, it is always used for cartoons. To produce motion using frame animation, one changes the foreground image.

GDI GDI (Graphic Device Interface) is the subsystem of Windows 95. It is responsible for displaying graphics (including text) on video displays and printers. The Windows GDI allows graphics output to be created by bitmaps, brushes, and pens. The GDI predefines three pens: Black, Null, and White. These can be used by calling the GetStockObject function. Seven brushes are predefined: Black, Dark-Gray, Gray, Hollow, Light-Gray, Null, and White. There are six hatched brush patterns. The GDI also supplies twelve text formatting styles.

GUI GUI (Graphical User Interface) is a metaphor representing the interaction between end user and computers. In particular, GUIs enable users to begin to think in terms of colors, icons, and graphics. With GUI, the system is more user-friendly, easier to learn and easier to use.

MS-DOS MS-DOS is the Microsoft implementation. IBM and Microsoft make efforts together to ensure that MS-DOS is functionally equivalent with PC-DOS which is the IBM implementation.

OpenGL The Microsoft implementation of OpenGL in Windows NT and Windows 95 is an implementation of the industry-standard OpenGL three-dimensional (3D) graphics software interface with which programmers create high-quality still and animated 3D color images.

OS/2 OS/2 is an operating system. In the mid - 1980s, Microsoft and IBM developed an operating system so superior to DOS that it was to become the dominant operating system of the 1980s and 1990s. They called it Operating System 2, or OS/2.

RAM-Resident Popup RAM-resident programs are designed to reside in memory while other programs run. Users can activate their services by clicking those programs' icons.

Sprite A sprite is an irregularly shaped picture that can be moved anywhere on the screen. A sprite can be in front of or behind another sprite. A character in an animation could be a sprite. To make the character move around in a scene, one moves the sprite for that character and change its image to simulate changes in the appearance of the character as it moves.

Transparency The simplest way to define transparency is to select a color not used elsewhere in the image and paint all the transparent areas with that color. When the sprite image is drawn, the transparent-colored pixels won't be copied to the screen.

VSAM VSAM (Virtual Sequential Access Method) is IBM's general purpose B-tree based access method. VSAM is designed to support sequential searching as well as logarithmic cost insertion, deletion, and find operations. VSAM supports two forms of sequential searching, one is key-sequenced, the other one is entry-sequenced. The entry-sequenced VSAM files allow efficient sequential processing when no key accompanies a record. Since entry-sequenced VSAM files require no index, they are less expensive to maintain.

Appendix B - User's Menu

Wbtree: A Windows Version of B-Trees Application

WBTree is an animator of B-tree and its variants (B-tree, B*-tree and B+-tree). This program illustrates the dynamic movements of B-trees while they are being built or destroyed.

The follows are the detail descriptions for each menu items and controls supported by this program.

Deletion

Each word can be deleted from an order five B-tree's family. The rearrangement of the tree structure will be animated to illustrate node redistribution and concatenation of the B-tree's family data structure. The leaf nodes are arranged vertically under the parent node. Initially click on the box beside "Delete". Then enter one to three letter words one at a time, clicking "Delete" to delete the word.

Insertion

Each word entered will be inserted into an order five B-tree's family. The tree expansion will be animated to illustrate node division and root promotion of the B-tree's family data structure. The leaf nodes are arranged vertically under the parent node. Initially click on the box beside "Insert". Then enter one to three letter words one at a time, clicking "Insert" to enter the word. The program is limited to 156 nodes, which is sufficient to illustrate the structure of a B-tree or its variants. Don't enter so many words as to exceed this limit.

Open File

"File" menu pops up a command dialog box to allow users to specify a file to be contained the B-tree index data.

Options

WBtree supports three types of B-tree variants, B_tree, B*_tree, and B+_tree. The tree type is selected by clicking the "Options" from the menu bar and marking one of three tree types.

Quit

Quit: To quit this application simply click the "Quit" from the menu bar.

Appendix C - Class Overview

Entry Class

Entry class contains a pair of key and RRN (Relative Record Number) along with data which is ignored in this project implementation. This class is the foundation of Page class which is the base class of Pobj class.

Data Member:	
char key[]	Character string serves as key
long data	Long integer
long right	RRN of right child
Construction:	
Entry();	
Entry(const char *c);	
Entry(const char *c, long d);	
Entry(const Entry &e);	
~Entry()	
Methods:	
void operator=(const Entry &e)	Assigns all data members
void operator=(const char *c)	Assigns key only
friend int Compare(const Entry &a const Entry &b)	Compares the key string only. Returns -1 if a < b, 0 if a = b, and 1 if a > b.
friend int FullCompare(const Entry &a, const Entry &b)	Compares the key string as well as the data. Returns -1 if a < b, 0 if a = b, and 1 if a > b.

Page Class

Page class forms the basic data model of an individual B-tree node which includes an Entry class array (it has order-1 cells), a key counter, and the RRN of the left most child. This class is the base class of Pobj class which combines the Fmgr class and Page class together and constructs a file based data structure.

Data Member:	
int cnt	Number of entries or keys
long left	RRN of the left most child.
Entry entry[ORDER-1]	
Construction:	
Page()	Set up an empty node
~Page()	
Methods	
int Search(const Entry &e, int &n)	Tries to match e's key with the keys in the entries of this node. Returns -1 if e's key is less than all keys in the node. returns 0 if there was a match, return 1 if e's key is greater than all keys in the node. Passes back n of matching entry if any, or the n of the entry containing the appropriate branch to keep searching down.
int FullSearch(const Entry &e, int &n)	Like the first Search(), except we use FullCompare().
void Split(Page &b, int n):	Moves right half of this node at n into empty b. Assumes n is in range.

void InsEntry(Entry &e, int n);	Inserts entry e into node at position n. Assumes there is room and assumes n <= cnt
void Concatenate(Entry &e);	Adds entry e to the end of the node
void Concatenate(Page &p);	Adds all entries of node n to the end of this node. Assumes there is room.
void DelEntry(int n);	Deletes the entry at position posn. Assumes n is in range.
long &Branch(int n);	Returns the branch for the given position. Due to the layout of the node, we'll get the left branch if n = -1.
int LastPosn();	Returns the position of the last entry in the node
int IsEmpty();	Returns 1 if cnt = 0.
int IsFull();	Returns 1 if cnt = ORDER - 1.
int IsPoor();	Returns 1 if node has fewer than the minimum entries.
int IsPlentiful();	Returns 1 if node has more than the minimum number of entries.

Pobj Class

Since B-tree is a file based data structure, Pobj is the class which is designed to meet this requirement. Pobj inherits Page class and includes a file object in its data member that makes file accessing more convenient.

Data Members:	
long addr	File address of Page data
FmgrPtr fp	File object pointer
Construction:	
Pobj()	
Pobj(Page &c, long p)	
Pobj(FmgrPtr f)	
Pobj(const Pobj &c)	Copy constructor.
~Pobj()	
Methods:	
void Copy(const Pobj &c)	Copies one Pobj into another
Pobj branch(int ps)	Loads the child page at position ps into memory.
void NewPage()	Calls fmgr to reserve a new page in the file.
void Updated()	Writes Page data to file
operator long() const	Returns RRN
void operator=(const Pobj &c)	
void operator=(long p)	

Fmgr Class

Fmgr is designed for file based data structure to fulfill the file operations such as opening files, closing files, reading bytes, writing bytes, and seeking. The Fmgr pointer is the data member of Pobj and Btree classes.

Data Members:	
enum Io_op	fetch, store, seek
enum AccessMode	read_write, read_only
enum CheckWord	
char name[]	Name of the file
long fs	Address to first block of "heap" free space
long fe	Address of byte after end of file
long hs	Address of the start of the "heap"
FILE *fp	Stream file handle
Io_op lastop	Last I/O operation
char status	
Construction:	
Fmgr()	
virtual ~Fmgr()	
Methods:	
void FetchFBlkHdr(FBlkHeader &h, long p);	Reads in free block header, and tests check word to make sure the file is still in sync.
void StoreFBlkHdr(const FBlkHeader &h, long p);	Writes out free block header.
long Reclaim(unsigned nbytes);	Finds the first block on the free space list that is big enough for nbytes of data. Puts back on the free list all those bytes that aren't needed. Note that the amount put back must be at least the size of the free block header, otherwise, the entire block is considered not an appropriate size and rejected. Returns address of the newly reclaimed data, or 0 if there wasn't a free space block that was appropriate.

virtual int Create(char *fname, long static_sz)	Creates and opens a new file named fname, truncating it if it already exists. The area at the front of the file of length static_sz + sizeof(FmgrHeader) is reserved. Returns 1 if the file was successfully created and opened, else 0.
virtual int Open(char *fname, AccessMode mode)	Opens the fname file. File must exist file or error occurs. First closes the current file if open. Returns 1 if file opened successfully, else 0.
virtual void Close(int flush)	Closes the file if not already closed. Does nothing if in the error state.
long Alloc(unsigned nbytes)	Allocates a block of at least nbytes of data, either from the free space list, or from the end of the file. The number of bytes allocated is adjusted to be large enough to hold a FBlkHeader. Nothing is written to the newly allocated space. Returns location of space allocated, or returns a 0 if some error occurred.
void Free(unsigned nbytes, long p)	Frees the block at location p assumed to be nbytes in size. Block is placed on the front of the free space list. If nbytes is < sizeof(FBlkHeader), it is forced to that size, since that's the minimum size allocated.
void Fetch(void *d, unsigned n, long p)	Fetches n bytes from address p into buffer d. The address is always interpreted to be from the beginning of the file, unless it's CURRADDR, which means from the

	current position.
void Store(const void *d, unsigned n, long p)	Stores n bytes from buffer d to addr p. The addr is always interpreted to be from the beginning of the file, unless it's CURRADDR, which means from the current position.
void Seek(long ofs, int seek_mode)	Moves the file pointer to the byte offset ofs, using seek_mode, (which should be either SEEK_SET, SEEK_CUR, or SEEK_END).
int IsOpen() const	Returns true open status bit is 1
int ReadOnly() const	
int ReadyForWriting() const	File is ready for writing if it is ok and not read-only
void ClearErr()	
int OK() const	
int operator!() const	
operator const int () const	

Btree Class

Btree class is the base class of Bstar and Bplus classes. This class defines all the operations and data structure used to set up a B-tree. Its data members include a file object pointer which points to a file to store B-tree index data, a Pobj to the root of the tree to ease access, a BtreeHeader data structure which contains B-tree's tree information such as address of root page, order of tree, height of tree, number of total entries, and number of total nodes.

Data Members:	
FmgrPtr f	File the Btree is connected to
long bh_addr	Address of the Btree header
BtreeHeader bh	Btree header
Pobj root	Root node
Construction:	
Btree()	
~Btree()	
Mothods:	
void ReadHdr()	
void WriteHdr()	
virtual int Insert(Entry &e, Pobj &t)	Recursive function that tries to insert entry e into subtree t. Returns SUCCESS, DUPLKEY, ALLOCERR, or NODE_OVERFLOW. If NODE_OVERFLOW, then e becomes the median_entry to pass back to t's parent.
int Delete(Entry &e, Pobj &t)	Recursive function that deletes entry e from the subtree with root p. Returns SUCCESS, or FAIL if we couldn't find the entry.
void RestoreBalance(Pobj &p, int posn)	Node down branch at position posn in node p has one too few entries. Give it an entry from either its left or right sibling, or perhaps just merge the node with a sibling.
void RotateRight(Pobj &p, int posn)	Does a "right rotation" using the entry at node p, position posn as the pivot point. Assumes p is not a leaf and that there is a

	left and right child. Also assumes right child isn't full, and that p and left child aren't empty.
void RotateLeft(Pobj &p, int posn)	Does a "left rotation" using the entry at node p, position posn as the pivot point. Assumes p is not a leaf and that there is a left and right child. Also assumes left child isn't full, and that p and right child aren't empty.
void Merge(Pobj &p, int posn)	Merges the node on the branch left of the entry at position posn of node p, with the entry of p and the node on the branch to the right of the entry of p. Assumes posn in range.
static void PrintNode(Pobj &n)	Prints only node n.
static void PrintTree(Pobj &t, int sp)	
void SetParameters()	Sets the split position.
int Connect(FmgrPtr &fptr, int create, long bh_addr)	Connect to an already open file. If create is 1, we're creating a new btree (but not necessarily a new file.) Returns SUCCESS or FAIL.
void Disconnect()	Disconnects the btree from the file.
int Create(char *fname, long bh_addr)	Creates a new file to hold the btree. Disconnects from any file that we may be connected to first. Returns SUCCESS or FAIL.
int Open(char *fname, Fmgr::AccessMode mode, long bh_addr)	Opens an existing file to hold the btree. Disconnect from any file that we may be connected to first. Returns SUCCESS or FAIL.

void Close()	
virtual int Search(Entry &e)	Search the tree for the first node having a matching entry (ie: keys must match). If found, the data field of e is filled in. Returns SUCCESS or FAIL.
virtual int FullSearch(const Entry &e)	Like Search(), except both keys and data must match. Returns SUCCESS or FAIL.
int Add(char *k, long d)	Creates a new entry with key k and data d, and attempts to add the entry to the tree. Returns SUCCESS, DUPLKEY, or ALLOCERR.
virtual int Remove(char *k, long d)	Deletes entry having key k and data d from the tree. Returns SUCCESS, or FAIL if we couldn't find the entry.
int IsEmpty() const	
int IsOpen() const	
int OK() const	
int operator!() const	
operator int() const	
void ClearErr()	
void Statistics(int full)	Display tree status
void PrintTree()	

Bstar Class

Bstar class inherits Btree class and overrides the Insert() function which makes the B*-tree different from the B-tree in terms of splitting. This class also defines additional data members and methods to support the changes in Insert().

Data Members:	
int path[20];	Records the search path top->down
int index	
Construction	
Bstar()	
~Bstar()	
Methods:	
int Insert(Entry &e, Pobj &t);	B*-tree version of Insert function.
Pobj FindParent(int in);	This function uses path[] to find the (in-1)th level of node which is the (in)th level's parent in a particular search.
void SetParameters();	Set the split position.

<u>Bplus Class</u>

Bplus class inherits Btree class and overrides the Insert(), Search(), FullSearch(), and Remove() functions from the base class. The Bplus class also overloades the Delete() function which is a member function of the base class. The changes of B^+-tree are used to maintain an index set as well as a sequence set.

Data Members:	
None	
Construction:	
Bplus()	
~Bplus()	

Methods:	
int Insert(Entry &e, Pobj &t);	B^+-tree version of Insert function.
int Delete(Entry &e, Pobj &t, Pobj &Parent);	This Delete() overloading Btree's Delete().
int Search(Entry &e);	B^+-tree version of Search function.
int FullSearch(const Entry &e);	B^+-tree version of FullSearch function.
int Remove(char *k, long d);	B^+-tree version of Remove function.

Wnode Class

Wnode class is created as an interface for B-trees to call Windows 95 API to display B-trees as graphical images to the screen. It creates several methods to accomplish the animation which occurs during tree splitting and merging. This class also defines many associated methods to assist an individual node to locate its relative nodes which would help to achieve animating the output image.

Data Members	
RECT *RBoxes	Provides data structures to draw boxes to hold chRRN.
long *chRRN	Child nodes' RRN
RECT *KBoxes	Provides data structures to draw boxes to hold key strings.
char Keys[][]	Key strings
int Parent	Index of parent block
int *RSibling	Index of the right siblings
int LSibling	Index of the left sibling

int *Children	Index of children
int Entrys	Number of entries
long x	x is the coordinate of up left corner of wnode.
long y	y is the coordinate of up left corner of wnode.
long RRN	RRN of itself
int id	Index of itself
int Order	Order of the tree
BYTE direct	Move direction
int dist	Move distance
Construction:	
WNode()	
WNode(int n)	
~WNode()	
Methods:	
void InitWNode(int n,int ID)	All the data members are initialized in this function.
void ResetWNode()	When the node is deleted, this function is called to reset all the data members back to initial state except the id.
void MoveLeft(long dx)	Moves the whole object image left in dx units.
void MoveRight(long dx)	Moves the whole object image right in dx units.
void MoveUp(long dy)	Moves the whole object image up in dy units.
void MoveDown(long dy)	Moves the whole object image down in dy units.

void SetID(int i)	Assigns i to the data member id.
void SetKey(char *k,int i)	Copys k to Keys[i].
void SetchRRN(long cr, int i)	Assigns cr to chRRN[i].
void InsChild(int id, int posn)	Assigns id to Children[posn] and shift out all the children after the (posn)th children one position right.
int GetChild(int n)	Returns the index of the nth child
void DelChildren(int posn)	Resets the content of Children[] into -1 starting from the nth child (n=posn).
void FindChildren(WNode *wn)	This function locates several WNodes that have the RRNs match chRRN. It also sets up the sibling relation among children.
void MoveChildren(HDC hdc,WNode *wn)	If the children nodes are the leaf, this function moves them accroding to this wnode's position.
void SetParent(int p)	Assigns p to Parent.
int GetParent()	Returns Parent to caller.
void SetPosition(long X,long Y)	Sets the new location on all the RECT data members to make them reference to (X,Y).
void GetPosition(long &dX, long &dY)	Returns x and y into dX and dY.
void AddSibling(int s)	Assigns s to RSibling[0] and pushes the rest of right siblings accordingly.
int* GetSiblings()	Returns an integer array containing right sibling index
void CopySiblings(int *S)	Copys S[] to RSibling[].
void SetLeftSb(int ls)	Assigns ls to LSibling.
int GetLeftSb()	Returns Lsibling.
void UpdateLSb(WNode *wn)	Recursive function. It updates each child's

	right sibling information. The call starts from the rightmost child and stop at the left most child.
void Print(HDC hdc)	Simply prints to Windows
void EraseNode(HDC hdc)	This function prints wnode using background color. The effect is same as the eraser
void ShowNode(HDC hdc, WNode *wn)	
void PrintNode(HDC hdc, WNode *wn)	The function determines the move algorithm and calls ShowNode() to display the animation of nodes.
void MoveParentSb(HDC hdc, WNode *wn,BYTE dirt,int dst)	This is recursive function to move parent's siblings.
void MoveTree(HDC hdc, WNode *wn,BYTE dirt,int dst)	Recursive function to move nodes under it. This function also call ShowNode to display nodes.
int GetEntrys()	Returns data member of Entrys
void Updated(Page p)	Gets Page structure to update certain data members
void Updated(Pobj P)	Passes Pobj structure to update certain data members
void SetDirect(int d)	Sets the move direction. 0 - up, 1 - down, 2 - right, 3 - left, and 4 - not move.
int GetDirect()	Returns the move direction.
void SetDist(int d)	Sets the move distance.
int GetDist()	Returns the move distance.
BOOL IsLeaf()	
void operator=(long r)	

void operator=(int I)	
operator long() const	

Appendix D - Makefile

```
# Microsoft Developer Studio Generated NMAKE File, Format Version 4.00
# ** DO NOT EDIT **

# TARGTYPE "Win32 (x86) Application" 0x0101

!IF "$(CFG)" == ""
CFG=wbtree - Win32 Debug
!MESSAGE No configuration specified.  Defaulting to wbtree - Win32 Debug.
!ENDIF

!IF "$(CFG)" != "wbtree - Win32 Release" && "$(CFG)" != "wbtree - Win32 Debug"
!MESSAGE Invalid configuration "$(CFG)" specified.
!MESSAGE You can specify a configuration when running NMAKE on this makefile
!MESSAGE by defining the macro CFG on the command line.  For example:
!MESSAGE
!MESSAGE NMAKE /f "wbtree.mak" CFG="wbtree - Win32 Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "wbtree - Win32 Release" (based on "Win32 (x86) Application")
!MESSAGE "wbtree - Win32 Debug" (based on "Win32 (x86) Application")
!MESSAGE
!ERROR An invalid configuration is specified.
!ENDIF

!IF "$(OS)" == "Windows_NT"
NULL=
!ELSE
NULL=nul
!ENDIF
################################################################################
# Begin Project
# PROP Target_Last_Scanned "wbtree - Win32 Debug"
MTL=mktyplib.exe
CPP=cl.exe
RSC=rc.exe

!IF  "$(CFG)" == "wbtree - Win32 Release"

# PROP BASE Use_MFC 0
# PROP BASE Use_Debug_Libraries 0
# PROP BASE Output_Dir "Release"
# PROP BASE Intermediate_Dir "Release"
# PROP BASE Target_Dir ""
# PROP Use_MFC 0
# PROP Use_Debug_Libraries 0
# PROP Output_Dir "Release"
# PROP Intermediate_Dir "Release"
# PROP Target_Dir ""
OUTDIR=.\Release
INTDIR=.\Release

ALL : "$(OUTDIR)\wbtree.exe"

CLEAN :
	-@erase ".\Release\wbtree.exe"
	-@erase ".\Release\page.obj"
	-@erase ".\Release\wmain.obj"
	-@erase ".\Release\WBT.obj"
	-@erase ".\Release\Bstar.obj"
	-@erase ".\Release\wnode.obj"
	-@erase ".\Release\Exchdlr.obj"
	-@erase ".\Release\Fmgr.obj"
	-@erase ".\Release\Btree.obj"
	-@erase ".\Release\pobj.obj"
	-@erase ".\Release\bplus.obj"
	-@erase ".\Release\wbt.res"

"$(OUTDIR)" :
   if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"

# ADD BASE CPP /nologo /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /YX /c
# ADD CPP /nologo /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /YX /c
CPP_PROJ=/nologo /ML /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS"\
 /Fp"$(INTDIR)/wbtree.pch" /YX /Fo"$(INTDIR)/" /c
CPP_OBJS=.\Release/
CPP_SBRS=
# ADD BASE MTL /nologo /D "NDEBUG" /win32
# ADD MTL /nologo /D "NDEBUG" /win32
MTL_PROJ=/nologo /D "NDEBUG" /win32
# ADD BASE RSC /l 0x409 /d "NDEBUG"
# ADD RSC /l 0x409 /d "NDEBUG"
RSC_PROJ=/l 0x409 /fo"$(INTDIR)/wbt.res" /d "NDEBUG"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
BSC32_FLAGS=/nologo /o"$(OUTDIR)/wbtree.bsc"
BSC32_SBRS=
LINK32=link.exe
# ADD BASE LINK32 kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib /nologo /subsystem:windows /machine:I386
# ADD LINK32 kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib /nologo /subsystem:windows /machine:I386
LINK32_FLAGS=kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib\
 advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib\
 odbccp32.lib /nologo /subsystem:windows /incremental:no\
 /pdb:"$(OUTDIR)/wbtree.pdb" /machine:I386 /out:"$(OUTDIR)/wbtree.exe"
LINK32_OBJS= \
	"$(INTDIR)/page.obj" \
	"$(INTDIR)/wmain.obj" \
	"$(INTDIR)/WBT.obj" \
	"$(INTDIR)/Bstar.obj" \
	"$(INTDIR)/wnode.obj" \
	"$(INTDIR)/Exchdlr.obj" \
	"$(INTDIR)/Fmgr.obj" \
	"$(INTDIR)/Btree.obj" \
	"$(INTDIR)/pobj.obj" \
	"$(INTDIR)/bplus.obj" \
	"$(INTDIR)/wbt.res"

"$(OUTDIR)\wbtree.exe" : "$(OUTDIR)" $(DEF_FILE) $(LINK32_OBJS)
  $(LINK32) @<<
 $(LINK32_FLAGS) $(LINK32_OBJS)
<<

!ELSEIF  "$(CFG)" == "wbtree - Win32 Debug"

# PROP BASE Use_MFC 0
# PROP BASE Use_Debug_Libraries 1
# PROP BASE Output_Dir "Debug"
# PROP BASE Intermediate_Dir "Debug"
# PROP BASE Target_Dir ""
# PROP Use_MFC 0
# PROP Use_Debug_Libraries 1
# PROP Output_Dir "Debug"
# PROP Intermediate_Dir "Debug"
# PROP Target_Dir ""
OUTDIR=.\Debug
INTDIR=.\Debug

ALL : "$(OUTDIR)\wbtree.exe"

CLEAN :
	-@erase ".\Debug\vc40.pdb"
	-@erase ".\Debug\vc40.idb"
	-@erase ".\Debug\wbtree.exe"
	-@erase ".\Debug\WBT.obj"
	-@erase ".\Debug\Fmgr.obj"
	-@erase ".\Debug\bplus.obj"
	-@erase ".\Debug\wnode.obj"
	-@erase ".\Debug\pobj.obj"
	-@erase ".\Debug\Bstar.obj"
	-@erase ".\Debug\page.obj"
	-@erase ".\Debug\wmain.obj"
	-@erase ".\Debug\Btree.obj"
	-@erase ".\Debug\Exchdlr.obj"
	-@erase ".\Debug\wbt.res"
	-@erase ".\Debug\wbtree.ilk"
	-@erase ".\Debug\wbtree.pdb"

"$(OUTDIR)" :
   if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"

# ADD BASE CPP /nologo /W3 /Gm /GX /Zi /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /YX /c
# ADD CPP /nologo /W3 /Gm /GX /Zi /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /YX /c
CPP_PROJ=/nologo /MLd /W3 /Gm /GX /Zi /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS"\
 /Fp"$(INTDIR)/wbtree.pch" /YX /Fo"$(INTDIR)/" /Fd"$(INTDIR)/" /c
```

```
CPP_OBJS=.\Debug/
CPP_SBRS=
# ADD BASE MTL /nologo /D "_DEBUG" /win32
# ADD MTL /nologo /D "_DEBUG" /win32
MTL_PROJ=/nologo /D "_DEBUG" /win32
# ADD BASE RSC /l 0x409 /d "_DEBUG"
# ADD RSC /l 0x409 /d "_DEBUG"
RSC_PROJ=/l 0x409 /fo"$(INTDIR)/wbt.res" /d "_DEBUG"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
BSC32_FLAGS=/nologo /o"$(OUTDIR)/wbtree.bsc"
BSC32_SBRS=
LINK32=link.exe
# ADD BASE LINK32 kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib /nologo /subsystem:windows /debug /machine:I386
# ADD LINK32 kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib /nologo /subsystem:windows /debug /machine:I386
LINK32_FLAGS=kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib\
 advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib\
 odbccp32.lib /nologo /subsystem:windows /incremental:yes\
 /pdb:"$(OUTDIR)/wbtree.pdb" /debug /machine:I386 /out:"$(OUTDIR)/wbtree.exe"
LINK32_OBJS= \
	"$(INTDIR)/WBT.obj" \
	"$(INTDIR)/Fmgr.obj" \
	"$(INTDIR)/bplus.obj" \
	"$(INTDIR)/wnode.obj" \
	"$(INTDIR)/pobj.obj" \
	"$(INTDIR)/Bstar.obj" \
	"$(INTDIR)/page.obj" \
	"$(INTDIR)/wmain.obj" \
	"$(INTDIR)/Btree.obj" \
	"$(INTDIR)/Exchdlr.obj" \
	"$(INTDIR)/wbt.res"

"$(OUTDIR)\wbtree.exe" : "$(OUTDIR)" $(DEF_FILE) $(LINK32_OBJS)
   $(LINK32) @<<
 $(LINK32_FLAGS) $(LINK32_OBJS)
<<

!ENDIF

.c{$(CPP_OBJS)}.obj:
  $(CPP) $(CPP_PROJ) $<

.cpp{$(CPP_OBJS)}.obj:
  $(CPP) $(CPP_PROJ) $<

.cxx{$(CPP_OBJS)}.obj:
  $(CPP) $(CPP_PROJ) $<

.c{$(CPP_SBRS)}.sbr:
  $(CPP) $(CPP_PROJ) $<

.cpp{$(CPP_SBRS)}.sbr:
  $(CPP) $(CPP_PROJ) $<

.cxx{$(CPP_SBRS)}.sbr:
  $(CPP) $(CPP_PROJ) $<

################################################################################
# Begin Target

# Name "wbtree - Win32 Release"
# Name "wbtree - Win32 Debug"

!IF  "$(CFG)" == "wbtree - Win32 Release"

!ELSEIF  "$(CFG)" == "wbtree - Win32 Debug"

!ENDIF

################################################################################
# Begin Source File

SOURCE=.\wnode.cpp

!IF  "$(CFG)" == "wbtree - Win32 Release"

DEP_CPP_WNODE=\
	".\wnode.h"\
	".\wbt.h"\
	".\pobj.h"\
	".\page.h"\
	".\Fmgr.h"\
	".\Exchdlr.h"\


"$(INTDIR)\wnode.obj" : $(SOURCE) $(DEP_CPP_WNODE) "$(INTDIR)"


!ELSEIF  "$(CFG)" == "wbtree - Win32 Debug"

DEP_CPP_WNODE=\
	".\wnode.h"\
	".\wbt.h"\
	".\pobj.h"\
	".\page.h"\
	".\Fmgr.h"\
	".\Exchdlr.h"\

NODEP_CPP_WNODE=\
	".\wn"\
	".\i"\


"$(INTDIR)\wnode.obj" : $(SOURCE) $(DEP_CPP_WNODE) "$(INTDIR)"


!ENDIF
```

```
# End Source File
################################################################################
# Begin Source File

SOURCE=.\Exchdlr.cpp
DEP_CPP_EXCHD=\
	".\Exchdlr.h"\
	

"$(INTDIR)\Exchdlr.obj" : $(SOURCE) $(DEP_CPP_EXCHD) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\Fmgr.cpp
DEP_CPP_FMGR_=\
	".\Fmgr.h"\
	".\Exchdlr.h"\
	

"$(INTDIR)\Fmgr.obj" : $(SOURCE) $(DEP_CPP_FMGR_) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\page.CPP
DEP_CPP_PAGE_=\
	".\page.h"\
	

"$(INTDIR)\page.obj" : $(SOURCE) $(DEP_CPP_PAGE_) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\pobj.CPP
DEP_CPP_POBJ_=\
	".\pobj.h"\
	".\page.h"\
	".\Fmgr.h"\
	".\Exchdlr.h"\
	

"$(INTDIR)\pobj.obj" : $(SOURCE) $(DEP_CPP_POBJ_) "$(INTDIR)"


# End Source File
################################################################################
```

```
# Begin Source File

SOURCE=.\WBT.cpp
DEP_CPP_WBT_C=\
	".\wbt.h"\
	".\Btree.h"\
	".\Bstar.h"\
	".\bplus.h"\
	".\wnode.h"\
	".\Fmgr.h"\
	".\pobj.h"\
	".\Exchdlr.h"\
	".\page.h"\


"$(INTDIR)\WBT.obj" : $(SOURCE) $(DEP_CPP_WBT_C) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\wbt.rc
DEP_RSC_WBT_R=\
	".\wbt.h"\


"$(INTDIR)\wbt.res" : $(SOURCE) $(DEP_RSC_WBT_R) "$(INTDIR)"
  $(RSC) $(RSC_PROJ) $(SOURCE)


# End Source File
################################################################################
# Begin Source File

SOURCE=.\wmain.cpp
DEP_CPP_WMAIN=\
	".\wbt.h"\


"$(INTDIR)\wmain.obj" : $(SOURCE) $(DEP_CPP_WMAIN) "$(INTDIR)"


# End Source File



################################################################################
# Begin Source File

SOURCE=.\Btree.cpp
DEP_CPP_BTREE=\
	".\Btree.h"\
	".\Fmgr.h"\
	".\pobj.h"\
	".\wnode.h"\
	".\Exchdlr.h"\
	".\page.h"\
	".\wbt.h"\


"$(INTDIR)\Btree.obj" : $(SOURCE) $(DEP_CPP_BTREE) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\Bstar.cpp
DEP_CPP_BSTAR=\
	".\Bstar.h"\
	".\Btree.h"\
	".\Fmgr.h"\
	".\pobj.h"\
	".\wnode.h"\
	".\Exchdlr.h"\
	".\page.h"\
	".\wbt.h"\


"$(INTDIR)\Bstar.obj" : $(SOURCE) $(DEP_CPP_BSTAR) "$(INTDIR)"


# End Source File
################################################################################
# Begin Source File

SOURCE=.\bplus.cpp
DEP_CPP_BPLUS=\
	".\bplus.h"\
	".\Btree.h"\
	".\Fmgr.h"\
	".\pobj.h"\
	".\wnode.h"\
	".\Exchdlr.h"\
	".\page.h"\
	".\wbt.h"\


"$(INTDIR)\bplus.obj" : $(SOURCE) $(DEP_CPP_BPLUS) "$(INTDIR)"


# End Source File
# End Target
# End Project
################################################################################
```

Appendix E - Sample Output Screens

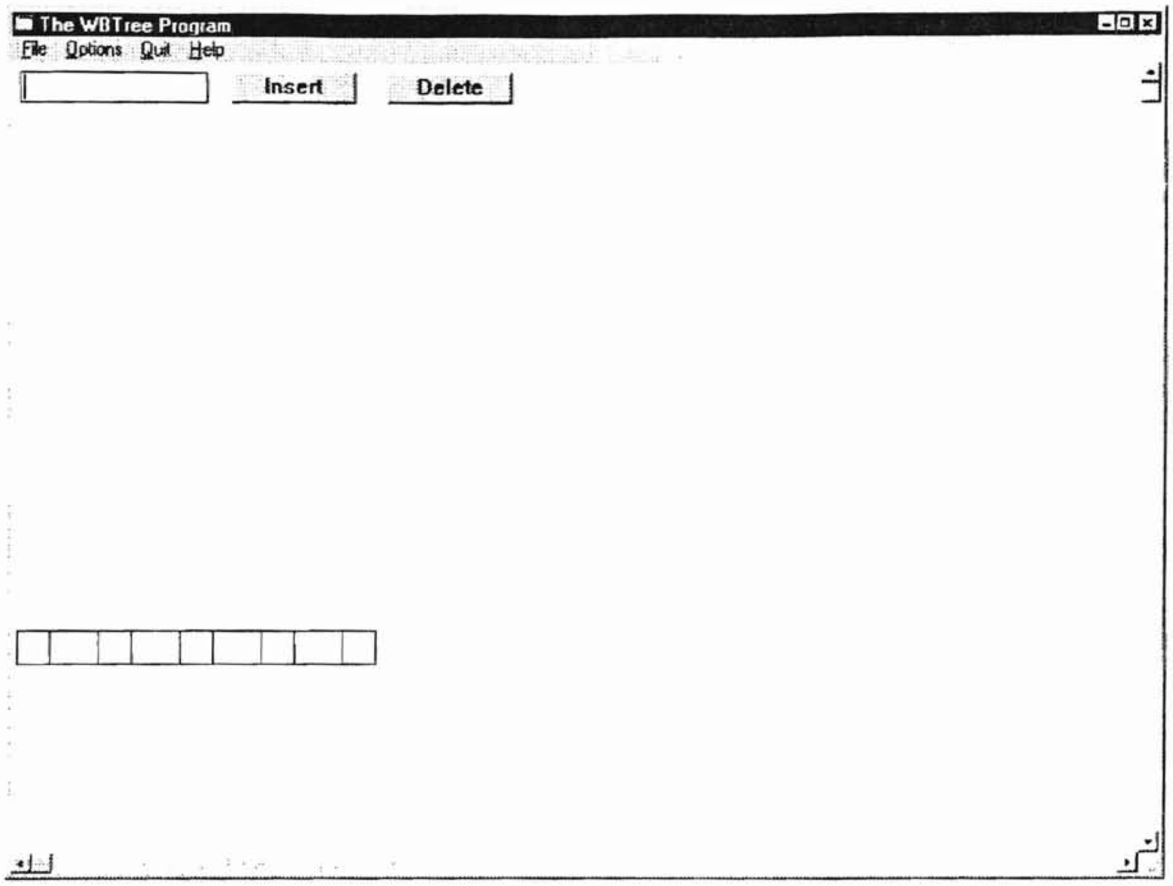


Figure 9 This is the initial screen of this program.

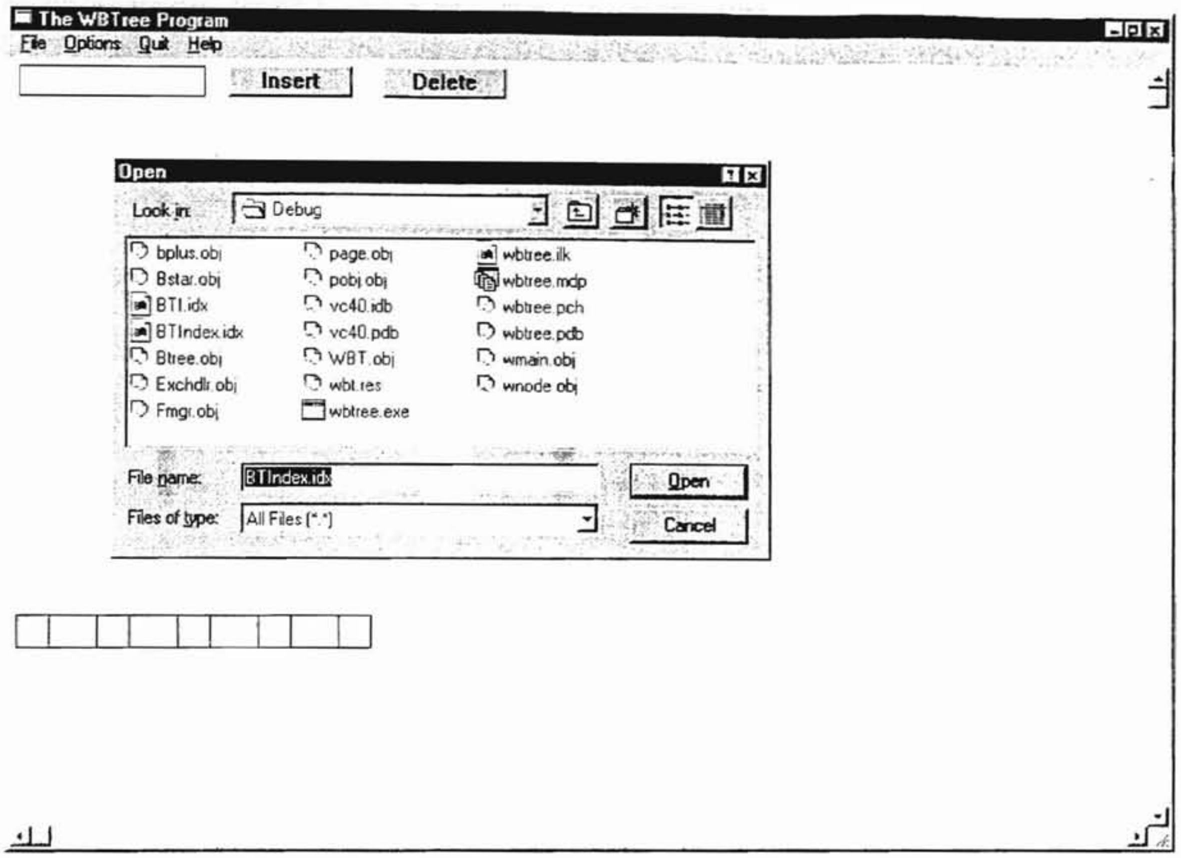


Figure 10 This screen shows a open file dialog box which is popped up by clicking the "File" in the menu bar.

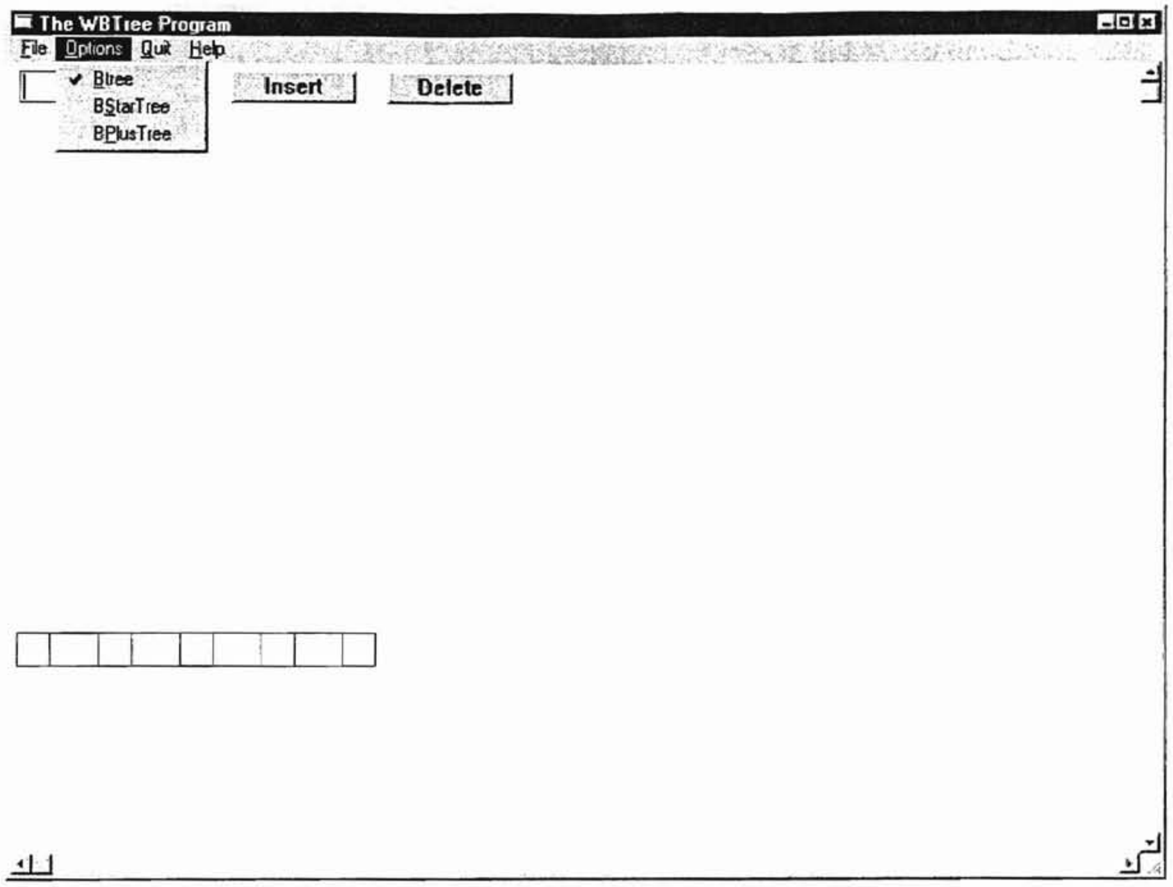
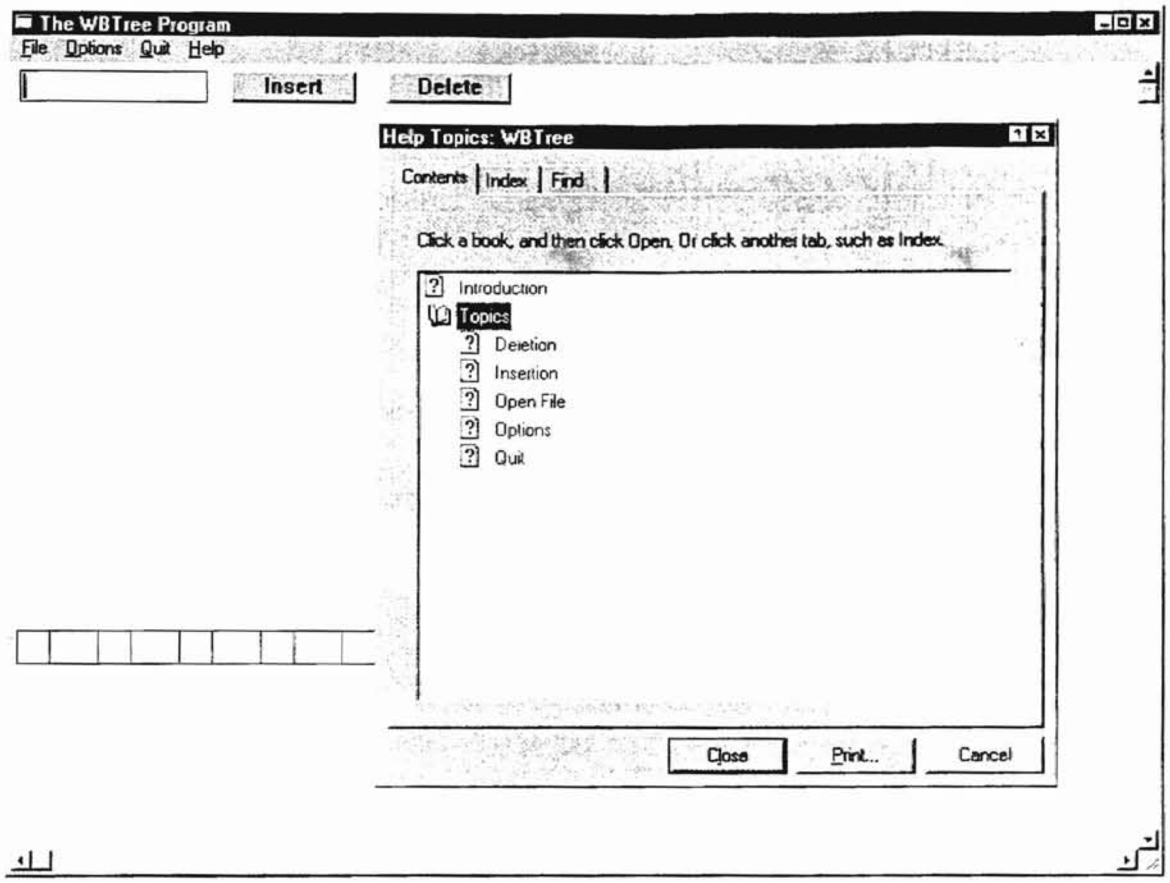


Figure 11 This screen shows the options menu.



Click a book, and then click Open. Or click another tab, such as Index.

- Introduction
- Topics
  - Deletion
  - Insertion
  - Open File
  - Options
  - Quit

Figure 12 This is the help screen when it is initially popped up by users.

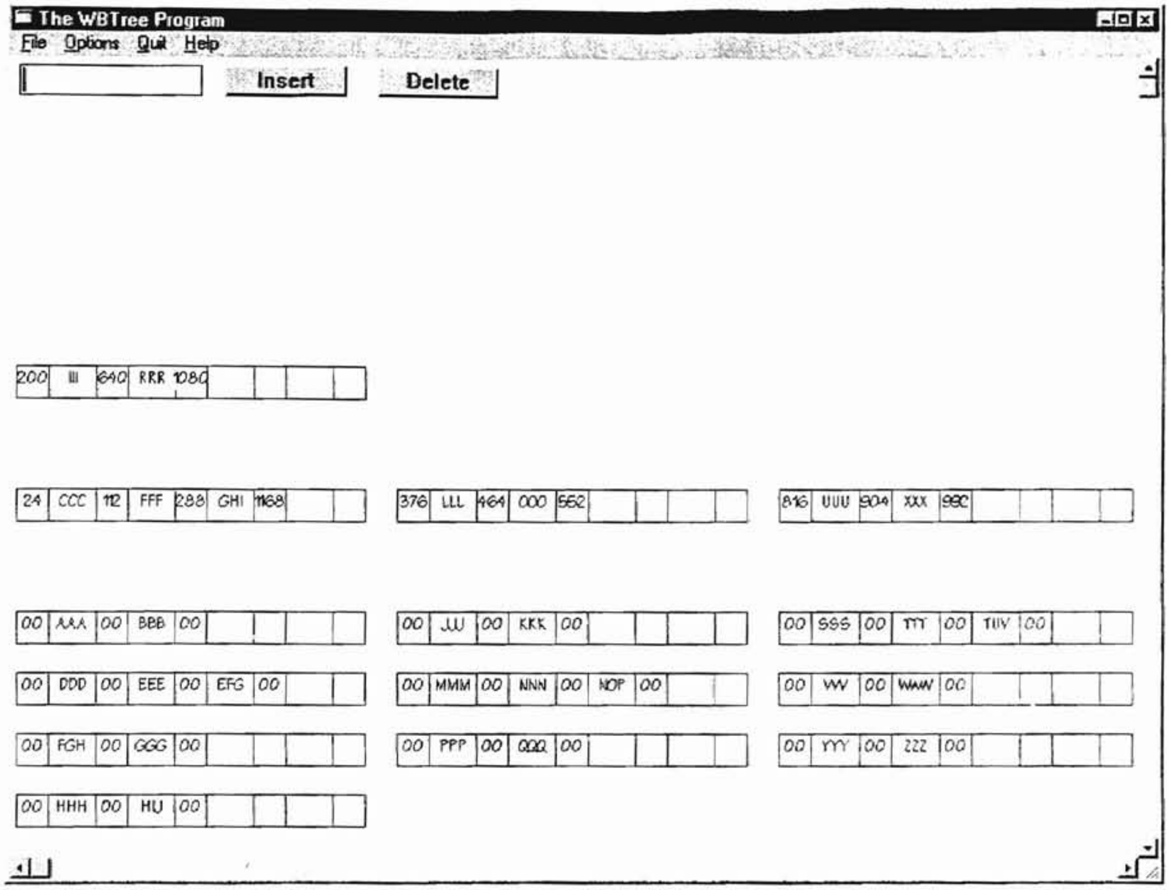


Figure 13 This screen shows the program is building a B-tree.

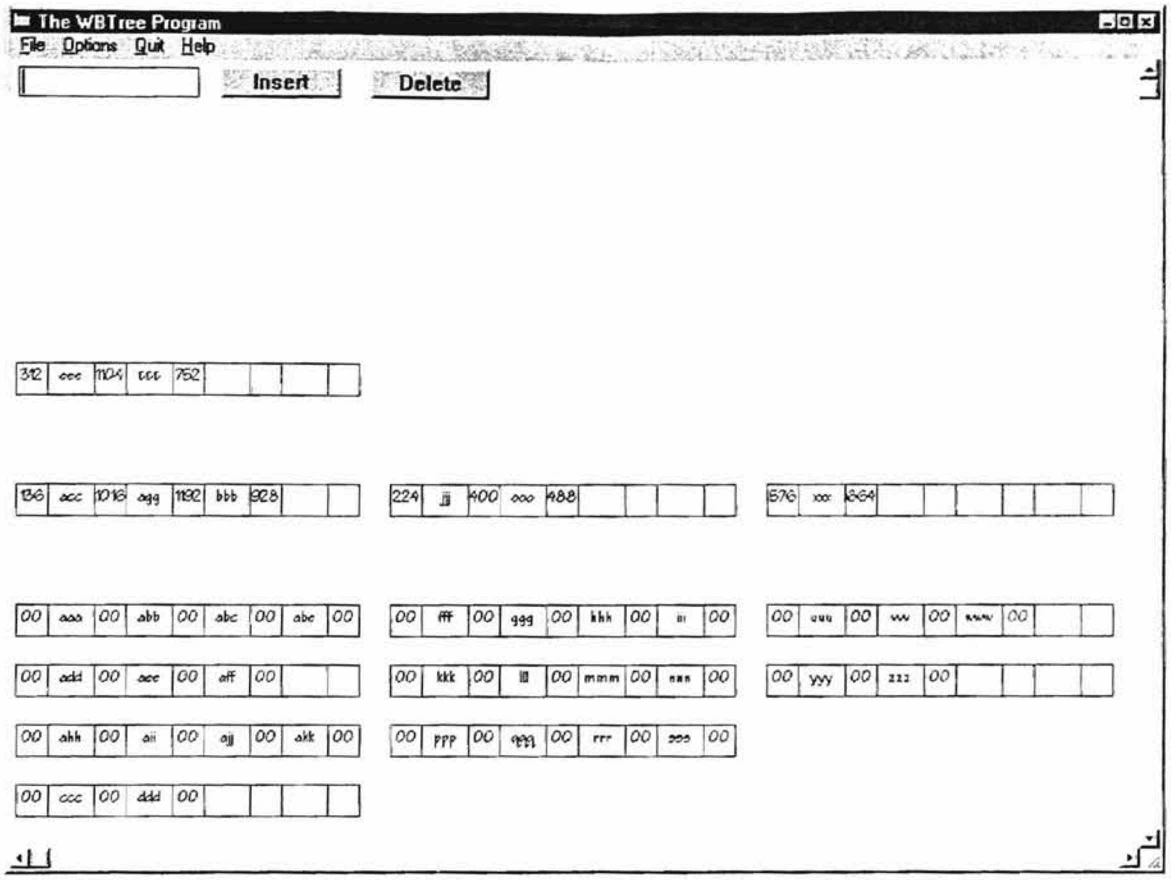


Figure 14. Screen of a B*-tree while it is being built

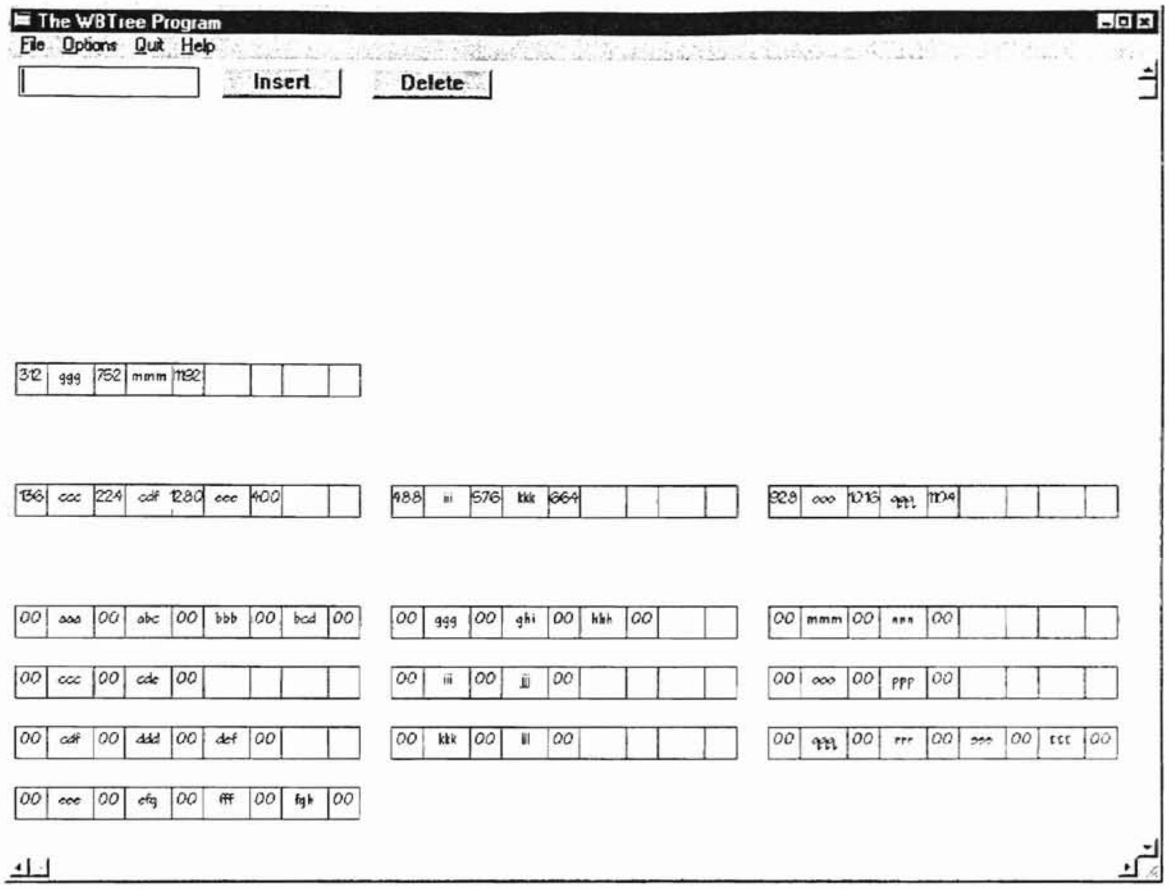


Figure 15. Screen of a B$^+$-tree while it is being built

VITA

Betty H.Lin

Candidate for the Degree of

Master of Science

Thesis: AN OBJECT-ORIENTED GRAPHIC USER INTERFACE FOR VISUALIZATION OF B-TREES' ANIMATOR

Major Field: Computer Science

Biographical:

Education: Graduated from Harbin #35 High School, Harbin, P. R. China; received Bachelor of Science degree in Public Health from Harbin Medical University in July 1984. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 1997.

Professional Experience: Employed as a newspaper reporter by Shanghai Public Health Newspaper, Shanghai, P. R. China from 1984 to 1987; attended an English and Communication Skill Training Program at World Health Organization (WHO) of United Nations, Manila, Philippines from 1987 to 1988; employed as a health education coordinator by WHO - Shanghai Collaborating Center for Health Education, Shanghai, P. R. China from 1988 to 1989.