TRUE BASIC

The Original BASIC

BTree Toolkit

Reference Guide

The B-Tree Library

Introduction

The B-Tree library provides you with a ready-to-use B-tree data structure which you can incorporate into your True BASIC programs through simple calls to the library. You can use these routines for any application that requires the storage and retrieval of data, including spreadsheets, databases, filing systems, and a host of other possibilities.

We also include True BASIC source code for these routines. You can use this code to learn more about the implementation of B-tree structures or modify it to more closely suit your specific needs.

Use the demo programs to experiment with the power and functionality of indexed record management. The program entitled **Compact.tru** is a useful utility for the maintenance of library data files.

The files supplied are:

BTreeLib.trc	The compiled library code
BTreeLib.tru	The library source code
BTManage.tru	A sample data base manager
Compact.tru	A file maintenance utility
BT_NewDb.tru	Creates a new Data Base called NameAdPh.
BT_Add.tru	Adds new data to the Data Base.
BT_AddCh.tru	Adds or changes data in the Data Base.
BT_Chng.tru	Changes data already in the Data Base.
BT_Del.tru	Removes data from the Data Base.
BT_Keys.tru	Displays the key of each entry in the Data Base.
BT_Find1.tru	Display selected keys contained in the Data Base.

BT_Find2.tru	Display selected keys contained in the Data Base.
BT_Find3.tru	Display selected keys contained in the Data Base.
BT_Tutor.tru	B-tree tutorial in text form.

The library file **BTreeLib.tru** and **BTreeLib.trc** are in the directory **TBLibs**. The remaining demo programs are in the directory

```
:tkdemos:B-tree
```

or

```
\b-tree toolkits.
```

B-trees

A B-tree is a method of storing data in, and, of course, retrieving data from, permanent storage on a diskette or some other device.

Although many methods have been devised for such processes, the use of B-trees is advantageous for a number of reasons. B-trees are fast, accurate, space efficient, and relatively easy to visualize. Thus, B-trees are useful to advanced and beginning programmers alike.

Because of their versatility and power, B-trees are widely used by applications developers for a wide variety of data handling. Similarly, because of their elegance and conceptual clarity, B-trees are widely studied by students of computer science. It was with these reasons in mind that the *True BASIC B-Tree Library* was developed.

However, the purpose of this chapter is to teach neither the applications of B-trees nor the theory of B-trees. Rather, the intent of this chapter is to teach the use of the routines which comprise the *B-Tree Library*. The various applications of these routines is left to your own imagination.

For a thorough discussion of the history and theory of B-tree structures, the interested user is referred to Chapter 9 of *File Structures: A Conceptual Toolkit* by Michael J. Folk and Bill Zoellick (Addison-Wesley, 1987). This highly readable chapter covers the evolution and implementation of B-tree structures and includes references to further reading on the subject.

Getting Started

When working with the *TB-Tree Library*, you may find it useful to use the True BASIC **load** command to install the **BTreeLib.trc** library in your computer's memory. This will reduce the time it takes a program to begin execution.

If, however, memory conservation is a consideration, you can, of course, reference the **BTreeLib.trc** library like any other, that is by using a **library** statement within your program.

Should you find it necessary to modify the library file, simply compile the altered source code, save it, and use that compiled file instead. Remember to always save a copy of the original source, in case your changes don't work as expected!

Using This Library

Before we discuss in detail the specific routines which comprise the *B-Tree Library*, we need to introduce a few terms fundamental to the functioning of a B-tree.

B-trees consist of an element, or *node*, for each item of information stored in them. In the implementation of a B-tree used by this Toolkit, each node consists of two items, a *key* and its *data*.

The *key* is a string value by which the node is identified. For this reason, it is important that the same key value not be used to identify two nodes. Keys may be up to 100 characters long and may contain any characters you choose [excluding *loval$* (or **chr$**(0)) and *hival$* (or **chr$**(127))], so creating unique names is not difficult.

The *data* is a string associated with the key. The data string is commonly used to hold the information to be stored. Since the data is identified by its associated key, there is no reason why two keys cannot identify identical data strings. There is no limit on the length of data strings, so you can store a large amount of data at each node. You will find that it is quite easy to store several data items in a single data string. Data strings may include any characters you choose to use.

The Toolkit stores B-trees in standard True BASIC files. Thus, the rules which concern standard True BASIC files also apply to the Toolkit's data files. If you are uncertain about these rules, refer to Chapters 9 & 18 for details. Note especially that in the parameter lists of the subroutines which follow, the #1 is used only as a parameter. It will be replaced by whatever channel number you specify in that position when you **call** that subroutine. Thus, it is possible to operate on several open data files at the same time.

The Routines

The following routines are all True BASIC subroutines. Thus, they are all invoked with the *call* statement.

BTOpen (dbasename$, fsize, #1)

BTOpen will first scan the proper directory for the existence of a B-Tree data file named *dbasename$*. If it exists, it will be opened. If it does not yet exist, it will be created first and then opened. Once opened it will be associated with the channel number specified by *#1*, and its current size in bytes is returned in *fsize*. A data file must be opened before its contents may be accessed.

When a database is created, it is initialized to have no key values, one index page (which is also the root page), and no data values. Notice that you are not required to set a maximum key or data record size, a maximum number of records, or a maximum file size; B-Tree data files will grow dynamically as records are added.

Note, however, that although you need never specifically create or initialize a data file, it is always possible to inadvertently create an unwanted data file with TBOpen by passing an incorrect *dbasename$*. See the routine **SUB IfExists** in the sample program BTManage for an easy way to avoid this problem.

BTClose (#1)

BTClose simply executes the standard True BASIC **CLOSE** statement for the channel number specified, and is provided solely as a mnemonic. Since B-Tree data files are essentially standard True BASIC files, they can be closed like any other file.

Add (key$, data$, #1, added)

Add adds the specified *key$* and its associated *data$* record to the open data file referenced by *#1*. If *key$* already exists in the specified file, no addition will occur. *Data$* may be as long as desired, and *key$* may be up to 100 characters long.

If an addition occurred and was successful, *added* will be returned equal to 1. If for some reason the addition failed, *added* will be returned equal to zero.

Update (key$, newdata$, #1, updated)

Update searches the open data file referenced by *#1* for the existence of *key$*. If that key value exists, *newdata$* replaces the data that was previously stored there. *Newdata$* may be any length desired.

If the update was successful, *updated* will be returned equal to 1, otherwise *updated* will equal zero.

Please note that, although *newdata$* is associated with *key$* and thereby retains its sequential position in the pointer list, the actual data contained in *newdata$* is added to the end of the entire data file, and no attempt is made by the Toolkit to reuse the storage occupied by the previous data item. This is important to remember for two reasons.

First, it means that a 'compactor' routine should be used to periodically reclaim space previously used by updated or deleted data items. This will ensure that the data file consists of only pertinent and current data. Such a routine is not difficult to write. For an example, see the sample program **Compact.tru** in the B-Tree directory.

Second, it means that original data which has been deleted or updated since the last 'compaction' can be recovered. It is, however, the programmer's responsibility to ensure that each data item in this case has enough information to identify it, since the key associated with it is lost.

Delete (key$, #1, deleted)

If *key$* exists in the file referenced by #*1*, that key is made into a 'tombstone.' If *key$* does not exist in the specified file, no action is taken. If a deletion occurred and was successful, *deleted* will be returned equal to 1, otherwise *deleted* will equal zero.

A tombstone is simply a key that has a negative, or invalid, pointer value, and tombstoned keys appear non-existent to the access routines. This means that that key can be reused by a subsequent addition. This approach also makes it easy for a compactor program or routine to reclaim the space used by old data items. The tombstone method is also useful because it is extremely fast and maintains the B-tree's balance. For a discussion of the internal handling of deleted keys, refer to the above discussion of the Update routine.

Find (key$, #1, found)

Find determines if the specified key currently exists in the specified data file. If it does exist *found* will be returned equal to 1, otherwise *found* will equal zero.

LoadKeys (keys$(), #1)

LoadKeys builds a sorted array containing the keys currently contained in the open data file referenced by #*1*.

InitNext

InitNext must be called before using GetNext to begin a sequential scan. It simply clears the list of pointers used internally for sequential access to the B-tree structure. Examine the sample programs for numerous examples.

Get (key$, data$, #1, found)

Get retrieves from the open data file referenced by #*1* the data item associated with the specified *key$* and returns it in *data$*. If the *key$* exists in the file, then *found* will be returned equal to 1, otherwise *found* will equal zero. If *found* does equal zero, the contents of *data$* should be ignored.

GetFirst (firstkey$, #1)

If the data file referenced by #*1* is not empty, then GetFirst returns the first key in sorted order in *firstkey$*. If the file is empty, *firstkey$* will equal the null string.

GetNext (next$, #1)

GetNext returns in *next$* the key value of the next key in sorted order in the file referenced by the channel number #*1*.

Since the keys in the data file are not physically stored in sorted order, a low-level recursive tree scanning routine builds a list of internal pointers, which is in sorted order. Since a file may become very long, however, this pointer list could also grow to be extremely long, and could therefore take a long time to build. To save space and time, the program maintains a list of only the next fifty keys in sorted order.

This pointer list is initially built by a call to InitNext and is periodically updated. The update is handled automatically by the Toolkit, but may cause a slight pause periodically in sequential processing. However, this automated updating means that InitNext need be called only once, at the beginning of the sequential scan, regardless of the size of the data file.

It is important to remember that whenever a new sequential scan begins from a non-contiguous location InitNext must be called. Otherwise, your scan may return keys left over from a previous scan.

GetLast (lastkey$, #1)

If the data file referenced by #*1* is not empty, then GetLast returns the last key in sorted order in *lastkey$*. If the file is empty, *lastkey$* will equal the null string.

GetNearest (partial_key$, key$, #1)

GetNearest returns in *key$* the value of the key in the file referenced by *#1* which is equal to *partial_key$*. If no such key value exists, GetNearest returns the next key from the file referenced by *#1* in sorted order which is greater than *partial_key$*.

This routine can be very useful for perusal of data files and listing intervals of key values. For example, refer to the sample program **BTManage.tru**, which offers the option to list an interval of keys in a file.

File Compaction

The *B-Tree Library* does not ever actually delete information from the data file. Instead it simply adds new data to the end of the existing data and remembers to ignore any old, or inactive, data.

This means that deletion operations can be done very quickly without any need for the time consuming task of reorganizing the data file. However, it also means that over time, as deletions and updates are made, the file grows with the retention of obsolete data items.

While this may at first sound somewhat inefficient, upon closer examination it turns out to offer two important benefits. As we have already mentioned, it means that any operation that requires a change of data can be performed as quickly as possible. Because no data is ever automatically erased from the file, it also means that no data can be irretrievably lost. In other words, deleted data is available, in case of error or emergency, until you as the programmer consciously delete it.

This, however, raises the question of how to delete this old data when it is not needed. There is a utility program on the Toolkit disk called **Compact.tru** which may be used to compact data files. While this program is designed to be used as an independent utility, the method it uses it easily adaptable to inclusion in a subroutine. In a simple data base manager, like **BTManage.tru**, such a subroutine could be used to implement a ‘compact’ command that the user can choose at any time. In a more complex system, it may be feasible to implement a feature which automatically compacts the data file after a set number of changes have been made. The best method of compacting the file will depend on the overall design of the system.

A TB-Tree Tutorial

The following group of programs will show you how to use the B-Tree Library to store and retrieve data.

The programs will also show some of the advantages of using B-Tree.

When the Sequential or Random access method is used to store data, the stored data is called a file. When the B-Tree access method is used to store data, the stored data is called a data base.

We will use the following programs to show how the B-Tree Access Method can be used to store and retrieve data.

BT_NewDb	will create a new Data Base called NameAdPh.
BT_Add	will add new data to the Data Base.
BT_Chng	will change data already in the Data Base.
BT_Del	will remove data from the Data Base.
BT_Keys	will display the key of each entry in the Data Base.
BT_Find	will display selected keys contained in the Data Base.

The first step in creating and using a data base is to decide what data we are going to put in the data base.

Lets create a data base to hold names, address, and phone numbers. A logical name for this data base would be **NameAdPh**.

We want each entry in the data base to be the name, address, and phone number of a particular person.

We want the name to consist of the person's last name followed by their first name, just as their name would appear in a phone book.

We want the address to consist of the person's street address, followed by the city, the state, and the zip code.

A typical entry in the data base might be the following:

```
Doe John
1234 Easy Street    Any Town    NY    10101
(000) 555-1234
```

The next step is to decide how to uniquely identify the entries in the data base. We need to be able to give each data base entry a unique name.

Since our data base is going to be small, it will not contain two people with the same name. We can therefore use the person's name to identify the entry. The identifier for the entry is called the entry's **key**. The key for the entry above would be the string "Doe John".

Now that we are done describing our data base, we are ready to create the data base. Program Bt-NewDb shows how to create a new data base.

Program BT_NewDb

A program to create a new data base called NameAdPh. We want this program to do the following:

First, check to see if data base NameAdPh already exists. If NameAdPh already exists, then display a message and stop.

Next, if NameAdPh does not exist, then create a new data base, giving it the name NameAdPh.

Lastly, we will display the size of the new data base. The size is the number of bytes of disk space used by the empty data base.

A detailed explanation of how BT_NewDb works can be found in the comment blocks within the program as listed below:

```
!-----------------------------------------------------------------------
! Program .. BT_NewDb                                                  |
! This program will create a new Data Base called NameAdPh.           |
! The Data Base will be used to store Names, Addresses, & Phone numbers.|
! The B-Tree subroutines used to create the Data Base are stored in    |
! the BTREELIB library.                                                |
!-----------------------------------------------------------------------

  LIBRARY "BTREELIB"

!-----------------------------------------------------------------------
! The following lines of code check to see if file (NameAdPh) already  |
! exists.  If the file already exists, then the program will set       |
! variable (DataBaseExists$) to "Yes".                                 |
!-----------------------------------------------------------------------

  LET DataBaseExists$ = "Yes"

  WHEN ERROR IN
     OPEN #1: NAME "NameAdPh", ORG byte, CREATE old, ACCESS input
```

```
  USE
     IF EXTYPE = 9003 THEN LET DataBaseExists$ = "No"
  END WHEN

  CLOSE #1

!-------------------------------------------------------------------------
! If file (NameAdPh) already exists, then the program will display a    |
! message, and wait for you to respond by pressing a key.  Once a key   |
! is pressed, the program will stop.                                    |
!-------------------------------------------------------------------------

  IF DataBaseExists$ = "Yes" THEN
     CLEAR
     SET CURSOR 5, 29
     PRINT "NameAdPh already exists."
     SET CURSOR 8, 24
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CLEAR
     STOP
  END IF

!-------------------------------------------------------------------------
! The program is now ready to create a new Data Base called (NameAdPh). |
! This is done by calling the B-Tree subroutine (BtOpen).               |
! Subroutine (BtOpen) will create a new Data Base called (NameAdPh),    |
! and store the size of (NameAdPh) in variable (DataBaseSize).          |
! If (BtOpen) fails to create (NameAdPh), then it will set variable     |
! (ExType) to 2.                                                        |
!-------------------------------------------------------------------------

  CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-------------------------------------------------------------------------
! If file (NameAdPh) already exists, then the program will display a    |
! message, and wait for you to respond by pressing a key.  Once a key   |
! is pressed, the program will stop.                                    |
!-------------------------------------------------------------------------
```

```
!-------------------------------------------------------------------------
! If variable (ExType) is not 0, meaning (BtOpen) failed to create      |
! the (NameAdPh) Data Base, then the program will display a message     |
! and wait for you to respond by pressing a key.  Once a key is         |
! pressed, the program will stop.  Variable (ExText$) will contain an   |
! explanation of what caused the error.                                 |
!-------------------------------------------------------------------------

  IF ExType > 0 THEN
     CLEAR
     SET CURSOR 5, 12
     PRINT "Trying to create the data base ";
     PRINT "caused the following error:"
     SET CURSOR 7, 11
     PRINT ExType; ".. "; ExText$
     SET CURSOR 11, 12
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CLEAR
     STOP
     END IF
!-------------------------------------------------------------------------
! Getting to this point means that a new Data Base has been created.    |
! The following lines of code will display the size (number of bytes)   |
! of the empty Data Base, and wait for you to respond by pressing a key.|
! Once a key is pressed, the program will close the Data Base and stop. |
!-------------------------------------------------------------------------
  CLEAR
  SET CURSOR 5, 22
  PRINT "The empty data base takes"; DataBaseSize; "bytes."
  SET CURSOR 8, 22
  PRINT "Press any key to end the program .. ";
  DO WHILE KEY INPUT
     GET KEY KeyPressed
     LOOP
  GET KEY KeyPressed

  CALL BtClose (#1)
  CLEAR
  END
```

BT_NewDb uses two B-Tree commands, (BtOpen) and (BtClose). These commands actually call B-Tree subroutines. The (Library) command tells the program that the B-Tree subroutines can be found in the library called BTreeLib.

BTOpen

A B-Tree subroutine which Opens a data base and makes it available to the program.

```
Call BTOpen (DataBaseName$, DataBaseSize, #1)
```

If (DataBaseName$) already exists, then (BtOpen) will Open it and make it available to the program. If (DataBaseName$) does not exist, then (BtOpen) will create the structure for a new data base, and then Open it and make it available to the program. (BtOpen) will store the size of the data base in variable (DataBaseSize). If (BtOpen) is unable to Open the data base, it will cause a runtime error having an error number of 2.

BTClose

A TB-Tree subroutine which closes a previously opened data base.

```
Call BTClose (#1)
```

Now that we have created a data base called NameAdPh, we are ready to store data in the data base. Program Bt-Add shows how to add a new entry to the data base.

Program BT_Add

A program to add a new entry to data base NameAdPh. We want this program to do the following:

First, prompt for the new person's last name and first name.

Next, check to see if the new person is already in the data base. If they are, then display a message and stop.

Next, prompt for the new person's address and phone number.

Next, add the new person to the data base.

Lastly, display a message indicating whether or not the new entry was successfully added to the data base.

A detailed explanation of how BT_Add works can be found in the comment blocks within the program, as shown:

```
!-------------------------------------------------------------------------
! Program .. Bt-Add                                                      |
! This program adds a new entry to the (NameAdPh) Data Base.             |
! The entry is a person's Names, Addresses, & Phone number.              |
! The entry's identifier or Key is the person's name.                    |
! The B-Tree subroutines that work with the Data Base are stored in      |
! the BTREELIB library.                                                  |
!-------------------------------------------------------------------------

  LIBRARY "BTREELIB"

!-------------------------------------------------------------------------
! Prior to working with a Data Base, the program must (Open) it.         |
! This is done by calling the B-Tree subroutine (BtOpen).                |
! Since Data Base (NameAdPh) already exists, the (BtOpen) subroutine     |
! will not create a new Data Base as it did in the (Bt-NewDb) program.   |
! Instead, it will open the existing Data Base and make it available     |
! to the program.                                                        |
!-------------------------------------------------------------------------

  CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-------------------------------------------------------------------------
! The following lines of code will prompt for the person's last name     |
! and first name.                                                        |
! The program will then build the entry's identifier or key, by first    |
! converting the names to upper case, and then joining the two names     |
! together to form a single string.                                      |
! The names are converted to upper case so that (John Doe) & (JOHN DOE) |
! will produce the same key and thus be the same entry.                  |
!-------------------------------------------------------------------------
  CLEAR
  SET CURSOR 2, 2
  INPUT PROMPT "Enter the person's last name ..... ":
  LastName$
  SET CURSOR 4, 2
  INPUT PROMPT "Enter the person's first name ..... ":
  FirstName$

  LET DbKey$ = UCASE$(LastName$) & " " & UCASE$(FirstName$)
```

```
!-------------------------------------------------------------------------
! The program is now ready to see if this person is already in the      |
! Data Base.  This is done by calling the B-Tree subroutine (Find).     |
! If the person is already in the Data Base, then (Find) will set the   |
! variable (AlreadyExists) to 1.                                        |
!-------------------------------------------------------------------------

  CALL Find (DbKey$, #1, AlreadyExists)

!-------------------------------------------------------------------------
! If the person is already in the Data Base, then the program will      |
! display a message, and wait for you to respond by pressing a key.     |
! Once a key is pressed, the program will stop.                         |
!-------------------------------------------------------------------------

  IF AlreadyExists = 1 THEN
     SET CURSOR 6, 2
     PRINT "This person is already in the Data Base."
     SET CURSOR 8, 2
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CALL BtClose (#1)
     CLEAR
     STOP
     END IF

!-------------------------------------------------------------------------
! Getting to this point means that the person is not in the Data Base.  |
! The following lines of code will prompt for the person's address      |
! and phone number.                                                     |
!-------------------------------------------------------------------------
  SET CURSOR 6, 2
  INPUT PROMPT "Enter the street address or box number ... ":
  Street$
  SET CURSOR 8, 2
  INPUT PROMPT "Enter the city ........................... ": City$
  SET CURSOR 10, 2
  INPUT PROMPT "Enter the 2 character state abreviation .. ": State$
  SET CURSOR 12, 2
  INPUT PROMPT "Enter the zip code ....................... ": Zip$
  SET CURSOR 14, 2
  INPUT PROMPT "Enter the phone number ................... ": Phone$
```

```
!-------------------------------------------------------------------------
! The program is now ready to add this new entry to the Data Base.      |
! This is done by calling the B-Tree subroutine (Add).                  |
! Prior to adding a new entry to a Data Base, the data must be packed   |
! into a single data string.  When packing the data, if we separate     |
! the fields with a certain character (such as the \ character),        |
! then we can later unpack the data string and get the fields back.     |
! If subroutine (Add) is unable to add the new entry to the Data Base,  |
! then it will set variable (AddedOk) to 0.                             |
!-------------------------------------------------------------------------

  LET DataString$ = Street$ & "\" & City$ & "\" & State$ & "\" & Zip$ & "\"
  LET DataString$ = DataString$ & Phone$

  CALL Add (DbKey$, DataString$, #1, AddedOk)

!-------------------------------------------------------------------------
! The program now displays a message indicating whether or not the new  |
! entry was successfully added to the Data Base.                        |
! After the appropriate message is displayed, the program will wait     |
! for you to respond by pressing a key.  Once a key is pressed, the     |
! program will stop.                                                    |
!-------------------------------------------------------------------------

  SET CURSOR 16, 2
  IF AddedOk = 0 THEN
     PRINT "B-Tree subroutine (Add) was unable to add the new entry.";
     ELSE
     PRINT "The entry was successfully added to the Data Base.";
     END IF
  SET CURSOR 18, 2
  PRINT "Press any key to end the program .. ";
  DO WHILE KEY INPUT
     GET KEY KeyPressed
     LOOP
  GET KEY KeyPressed

  CALL BtClose (#1)
  CLEAR
  END
```

Bt-Add introduces two new TB-Tree commands, **Find** and **Add**.

Find is a TB-Tree subroutine which will check to see whether or not a certain entry is in the data base. Prior to calling the (Find) subroutine, store the entry's identifier in variable (Key$).

```
Call Find (Key$, #1, Found)
```

If the entry is in the data base, then (Find) will set variable (Found) equal to 1. If the entry is not in the data base, then (Find) will set (Found) equal to zero.

Add is a TB-Tree subroutine which will add a new entry to the data base. Prior to calling the (Add) subroutine, store the entry's identifier in variable (Key$). Store the entry's data (address & phone number) in variable (DataString$).

```
Call Add (Key$, DataString$, #1, AddedOk)
```

If the entry is not in the data base, then (Add) will add the new entry to the data base. If there were no problems with the addition process, then (Add) will set variable (AddedOk) equal to 1. If for some reason, the addition process failed, then (Add) will set (AddedOk) to zero.

If the entry was already in the data base, then (Add) will set (AddedOk) equal to zero, and return to the main program.

Notice that program Bt-Add displays a message and stops, if the entry is already in the data base. Suppose we want program Bt-Add to either add a new entry, or if the entry is already in the data base, then change the existing entry. A slight modification to Bt-Add could accomplish this.

Program BT_AddCh

A program to add or replace a data base entry. This time, we want the program to do the following:

First, prompt for the person's last name and first name.

Next, prompt for the person's address and phone number.

Next, if the person is not in the data base, then add the person to the data base. If the person is already in the data base, then replace the old entry with the new entry.

Lastly, display a message indicating whether or not the entry was successfully placed in the data base. If the entry was successfully placed, then have the message indicate whether it was a new entry or a change to an existing entry.

A detailed explanation of how Bt-AddCh works can be found in the comment blocks within the program as follows:

```
!-------------------------------------------------------------------------
! Program .. Bt-AddCh .. a modification of program Bt-Add.                |
! This program either adds a new entry to the (NameAdPh) Data Base,       |
! or if the entry is already in the Data Base, then changes the           |
! existing entry.                                                         |
!-------------------------------------------------------------------------

  LIBRARY "BTREELIB"

!-------------------------------------------------------------------------
! First, the program Opens the Data Base.                                 |
!-------------------------------------------------------------------------

  CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-------------------------------------------------------------------------
! Then prompts for the person's name, and builds the entry's key.         |
!-------------------------------------------------------------------------

  CLEAR
  SET CURSOR 2, 2
  INPUT PROMPT "Enter the person's last name ..... ": LastName$
  SET CURSOR 4, 2
  INPUT PROMPT "Enter the person's first name ..... ": FirstName$

  LET DbKey$ = UCASE$(LastName$) & " " & UCASE$(FirstName$)

!-------------------------------------------------------------------------
! Then checks to see if the person is already in the Data Base.           |
!-------------------------------------------------------------------------

  CALL Find (DbKey$, #1, AlreadyExists)

!-------------------------------------------------------------------------
! Unlike program Bt-Add, this program does not display a message and      |
! stop, if the entry is already in the Data Base.                         |
! Regardless of the outcome of the (Find) command, this program goes      |
! ahead and prompts for the person's address and phone number.            |
!-------------------------------------------------------------------------
```

```
  SET CURSOR 6, 2
  INPUT PROMPT "Enter the street address or box number ... ":
  Street$
  SET CURSOR 8, 2
  INPUT PROMPT "Enter the city ........................... ": City$
  SET CURSOR 10, 2
  INPUT PROMPT "Enter the 2 character state abreviation .. ": State$
  SET CURSOR 12, 2
  INPUT PROMPT "Enter the zip code ....................... ": Zip$
  SET CURSOR 14, 2
  INPUT PROMPT "Enter the phone number ................... ": Phone$

!-------------------------------------------------------------------------
! The program is now ready to add the entry to the Data Base.           |
! This time the program will use the B-Tree (Put) subroutine, instead   |
! of the B-Tree (Add) subroutine, to place the entry in the Data Base.  |
! If the person is a new entry, then (Put) will add the entry to the    |
! Data Base, just like the (Add) subroutine would do.                   |
! If however, the person is already in the Data Base, then (Put) will   |
! replace the existing entry with the new entry.                        |
!-------------------------------------------------------------------------

LET DataString$ = Street$ & "\" & City$ & "\" & State$ & "\" & Zip$ & "\"
LET DataString$ = DataString$ & Phone$

CALL Put (DbKey$, DataString$, #1, AddedOk)

!-------------------------------------------------------------------------
! The program is now ready to display a message indicating whether or   |
! not the entry was successfully placed in the Data Base.               |
! And if the entry was placed successfully, whether or not the person   |
! was already in the Data Base.                                         |
! After the appropriate message is displayed, the program will wait     |
! for you to respond by pressing a key.  Once a key is pressed, the     |
! program will stop.                                                    |
!-------------------------------------------------------------------------
```

```
SET CURSOR 16, 2
IF AddedOk = 0 THEN
   PRINT "B-Tree subroutine (Put) was unable to place the entry.";
   ELSE
   IF AlreadyExists = 1 THEN
      PRINT "The existing Data Base entry was successfully
replaced.";
      ELSE
      PRINT "The new entry was successfully added to the Data
Base.";
      END IF
   END IF
SET CURSOR 18, 2
PRINT "Press any key to end the program .. ";
DO WHILE KEY INPUT
   GET KEY KeyPressed
   LOOP
GET KEY KeyPressed

CALL BtClose (#1)
CLEAR
END
```

Bt-AddCh introduces one new B-Tree command, the **Put** command.

Put is a B-Tree subroutine which will either add a new entry to the Data Base, or if the entry is already in the data base, then replace the existing entry.

Prior to calling the (Put) subroutine, store the entry's identifier in variable (Key$). Store the entry's data (address & phone number) in variable (DataString$).

```
Call Put (Key$, DataString$, #1, AddedOk)
```

If the entry is not already in the data base, then (Put) works just like the (Add) subroutine outlined in the previous program.

If the entry was already in the data base, then (Put) will replace the old entry with the new entry.

If there were no problems with the addition or replacement process, then (Put) will set variable (AddedOk) to 1. If for some reason, the addition or replacement process failed, then (Put) will set (AddedOk) to zero.

Program Bt-Chng

Now that we have entries in our data base, we need a program which will allow us to change one of the entries. Maybe John Doe has moved from Any Town to New Town, and we want to correct his entry in the data base.

Program Bt-Chng shows how to change one of the data base entries.

We want this program to do the following:

First, prompt for the person's last name and first name.

Next, check to see if the person is in the data base. If they are not, then display a message and stop.

Next, retrieve the entry from the data base.

Next, verify that the entry was retrieved successfully. If there was a problem, then display a message and stop.

Next, display the entry.

Next, prompt for the person's new address and phone number.

Next, change the data base entry by replacing the old address and phone number with the new address and phone number.

Lastly, display a message indicating whether or not the Data Base entry was successfully changed.

A detailed explanation of how Bt-Chng works can be found in the comment blocks within the program as listed below:

```
!-----------------------------------------------------------------------
! Program .. Bt-Chng                                                   |
! This program changes the data in one of the Data Base entries.       |
!-----------------------------------------------------------------------

  LIBRARY "BTREELIB"

!-----------------------------------------------------------------------
! First, the program Opens the Data Base.                              |
!-----------------------------------------------------------------------

  CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-----------------------------------------------------------------------
! Then prompts for the person's name, and builds the entry's key.      |
!-----------------------------------------------------------------------
```

```
  CLEAR
  SET CURSOR 2, 2
  INPUT PROMPT "Enter the person's last name ..... ": LastName$
  SET CURSOR 4, 2
  INPUT PROMPT "Enter the person's first name ..... ": FirstName$

  LET DbKey$ = UCASE$(LastName$) & " " & UCASE$(FirstName$)

!-------------------------------------------------------------------------
! Then checks to see if the person is already in the Data Base.          |
!-------------------------------------------------------------------------

  CALL Find (DbKey$, #1, AlreadyExists)

!-------------------------------------------------------------------------
! If the person is not in the Data Base, then the program will display  |
! a message and wait for you to respond by pressing a key.  Once a key  |
! is pressed, the program will stop.                                    |
!-------------------------------------------------------------------------

  IF AlreadyExists = 0 THEN
     SET CURSOR 6, 2
     PRINT "This person is not in the Data Base."
     SET CURSOR 8, 2
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CALL BtClose (#1)
     CLEAR
     STOP
     END IF

!-------------------------------------------------------------------------
! Getting to this point means that the person is in the Data Base.      |
! The following line of code will retrieve the entry from the Data Base |
! and store the entry's data (address and phone number) in variable     |
! (DataString$).  If the (Get) subroutine successfully retrieves the    |
! entry from the Data Base, then it will set variable (GetOk) to 1.     |
! If the (Get) subroutine fails to retrieve the entry, then it will     |
! set variable (GetOk) to 0.                                            |
!-------------------------------------------------------------------------
```

```
  Call Get (DbKey$, DataString$, #1, GetOk)

!-------------------------------------------------------------------------
! If the (Get) subroutine failed to retrieve the entry from the Data    |
! Base, then the program will display a message and wait for you to     |
! respond by pressing a key.  Once a key is pressed, the program will   |
! stop.                                                                 |
!-------------------------------------------------------------------------

  IF GetOk = 0 THEN
     SET CURSOR 6, 2
     PRINT "B-Tree subroutine (Get) was unable to retrieve the entry."
     SET CURSOR 8, 2
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CALL BtClose (#1)
     CLEAR
     STOP
     END IF

!-------------------------------------------------------------------------
! Getting to this point means the (Get) subroutine was able to retrieve |
! the Data Base entry, and store the data in variable (DataString$).    |
! When program (Bt-Add) added the entry to the Data Base, it packed the |
! address and phone number fields into a single string.  It used the (\)|
! character to separate the Street, City, State, Zip, and Phone number. |
! The following lines of code will use those (\) characters to unpack   |
! (DataString$) and extract the individual fields.                      |
!-------------------------------------------------------------------------
  LET PositionSlash1 = POS (DataString$, "\", 1)
  LET FirstChar = 1
  LET LastChar = PositionSlash1 - 1
  LET Street$ = DataString$ [FirstChar:LastChar]

  LET FirstChar = PositionSlash1 + 1
  LET PositionSlash2 = POS (DataString$, "\", FirstChar)
  LET LastChar = PositionSlash2 - 1
  LET City$ = DataString$ [FirstChar:LastChar]
```

```
  LET FirstChar = PositionSlash2 + 1
  LET PositionSlash3 = POS (DataString$, "\", FirstChar)
  LET LastChar = PositionSlash3 - 1
  LET State$ = DataString$ [FirstChar:LastChar]

  LET FirstChar = PositionSlash3 + 1
  LET PositionSlash4 = POS (DataString$, "\", FirstChar)
  LET LastChar = PositionSlash4 - 1
  LET Zip$ = DataString$ [FirstChar:LastChar]

  LET FirstChar = PositionSlash4 + 1
  LET LastChar = LEN (DataString$)
  LET Phone$ = DataString$ [FirstChar:LastChar]

!-------------------------------------------------------------------------
! The program can now display the person's name, address, and phone.    |
!-------------------------------------------------------------------------

  CLEAR
  PRINT TAB (2, 2); "Name .......... "; DbKey$
  PRINT TAB (3, 2); "address ....... "; Street$
  PRINT TAB (4, 2); "city .......... "; City$
  PRINT TAB (5, 2); "state ......... "; State$
  PRINT TAB (6, 2); "zip code ...... "; Zip$
  PRINT TAB (7, 2); "phone number .. "; Phone$

!-------------------------------------------------------------------------
! The following lines of code will prompt for the person's new address  |
! and phone number.  If a single character (*) is entered for one of    |
! the fields, then the program will NOT change that field in the entry. |
!-------------------------------------------------------------------------

  SET CURSOR 9, 2
  INPUT PROMPT "Enter the street address or box number ... ": Entry$
  IF Entry$ <> "*" THEN LET Street$ = Entry$
  SET CURSOR 11, 2
  INPUT PROMPT "Enter the city ........................... ": Entry$
  IF Entry$ <> "*" THEN LET City$ = Entry$
  SET CURSOR 13, 2
  INPUT PROMPT "Enter the 2 character state abreviation .. ": Entry$
  IF Entry$ <> "*" THEN LET State$ = Entry$
  SET CURSOR 15, 2
  INPUT PROMPT "Enter the zip code ....................... ": Entry$
```

```
  IF Entry$ <> "*" THEN LET Zip$ = Entry$
  SET CURSOR 17, 2
  INPUT PROMPT "Enter the phone number ..................... ": Entry$
  IF Entry$ <> "*" THEN LET Phone$ = Entry$

!-------------------------------------------------------------------------
! The program is now ready to change the entry in the Data Base.         |
! This is done by calling the B-Tree subroutine (Update).                |
! Just as in program (Bt-Add), the data must be packed into a single     |
! data string prior to calling the B-Tree subroutine.                    |
! If the (Update) subroutine is unable to change the entry in the        |
! Data Base, then it will set variable (ChangedOk) to 0.                 |
!-------------------------------------------------------------------------

  LET DataString$ = Street$ & "\" & City$ & "\" & State$ & "\" & Zip$ & "\"
  LET DataString$ = DataString$ & Phone$

  CALL UpDate (DbKey$, DataString$, #1, ChangedOk)

!-------------------------------------------------------------------------
! The program is now ready to display a message indicating whether or    |
! not the Data Base entry was successfully changed.                      |
! After the appropriate message is displayed, the program will wait      |
! for you to respond by pressing a key.  Once a key is pressed, the      |
! program will stop.                                                     |
!-------------------------------------------------------------------------

  SET CURSOR 21, 2
  IF ChangedOk = 0 THEN
     PRINT "B-Tree subroutine (UpDate) was unable to change the entry.";
     ELSE
     PRINT "The Data Base entry was successfully changed.";
     END IF
  SET CURSOR 23, 2
  PRINT "Press any key to end the program .. ";
  DO WHILE KEY INPUT
     GET KEY KeyPressed
     LOOP
  GET KEY KeyPressed

  CALL BtClose (#1)
  CLEAR
  END
```

Bt-Chng uses two new B-Tree commands, **Get** and **UpDate**.

Get is a TB-Tree subroutine which will retrieve a data base entry. Prior to calling the (Get) subroutine, store the entry's identifier in variable (Key$).

```
Call Get (Key$, DataString$, #1, GetOk)
```

If the entry is in the data base, then (Get) will retrieve the entry and store the data in variable (DataString$). If there were no problems with the retrieval process, then (Get) will set variable (GetOk) to 1. If for some reason, the retrieval process failed, then (Get) will set (GetOk) to zero.

If the entry is not in the data base, then (Get) will set (GetOk) to zero and return to the main program.

If the entry was retrieved, then the program will have to unpack (DataString$) before the individual data items (address & phone number) can be used.

UpDate is a TB-Tree subroutine which will change a data base entry.

Prior to calling the (UpDate) subroutine, store the entry's identifier in variable (Key$). Store the entry's new data (address & phone number) in variable (DataString$).

```
Call UpDate (Key$, DataString$, #1, ChangedOk)
```

If the entry is in the data base, then (UpDate) will replace the old entry with the new entry. If there were no problems with the updating process, then (UpDate) will set variable (ChangedOk) to 1. If for some reason, the updating process failed, then (UpDate) will set (ChangedOk) to zero.

If the entry is not in the data base, then (UpDate) will set (ChangedOk) to zero and return to the main program.

We can now add and change data base entries, but we do not have a way to remove an unwanted entry. Program Bt-Del shows how to delete one of the data base entries.

Program Bt-Del

A program to delete a data base entry. We want this program to do the following:

First, prompt for the person's last name and first name.

Next, check to see if the person is in the data base. If they are not, then display a message and stop.

Next, retrieve the entry from the data base.

Next, verify that the entry was retrieved successfully. If there was a problem, then display a message and stop.

Next, display the entry.

Next, prompt whether or not the entry is to be deleted.

Lastly, if the answer is No, then stop the program. If the answer is Yes, then delete the entry from the data base. Display a message indicating whether or not the entry was successfully deleted.

A detailed explanation of how Bt-Del works can be found in the comment blocks within the program as listed below:

```
!-----------------------------------------------------------------------
! Program .. Bt-Del                                                    |
! This program deletes one of the Data Base entries.                   |
! This program is similar to program Bt-Chng, except it removes the    |
! the Data Base entry instead of changing it.                          |
!-----------------------------------------------------------------------

  LIBRARY "BTREELIB"

!-----------------------------------------------------------------------
! First, the program Opens the Data Base.                              |
!-----------------------------------------------------------------------

CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-----------------------------------------------------------------------
! Then prompts for the person's name, and builds the entry's key.      |
!-----------------------------------------------------------------------
  CLEAR
  SET CURSOR 2, 2
  INPUT PROMPT "Enter the person's last name ..... ": LastName$
  SET CURSOR 4, 2
  INPUT PROMPT "Enter the person's first name ..... ":
  FirstName$

  LET DbKey$ = UCASE$(LastName$) & " " & UCASE$(FirstName$)

!-----------------------------------------------------------------------
! Then checks to see if the person is already in the Data Base.        |
!-----------------------------------------------------------------------

  CALL Find (DbKey$, #1, AlreadyExists)
```

```
!-------------------------------------------------------------------------
! If the person is not in the Data Base, then the program displays a    |
! message and stops.                                                     |
!-------------------------------------------------------------------------

  IF AlreadyExists = 0 THEN
     SET CURSOR 6, 2
     PRINT "This person is not in the Data Base."
     SET CURSOR 8, 2
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CALL BtClose (#1)
     CLEAR
     STOP
     END IF

!-------------------------------------------------------------------------
! At this point, the program knows that the person is in the Data Base, |
! so it retrieves the entry from the Data Base and stores the entry's   |
! data in variable (DataString$).                                        |
!-------------------------------------------------------------------------

  Call Get (DbKey$, DataString$, #1, GetOk)

!-------------------------------------------------------------------------
! If (Get) failed to retrieve the entry, then the program displays a    |
! message and stops.                                                     |
!-------------------------------------------------------------------------

  IF GetOk = 0 THEN
     SET CURSOR 6, 2
     PRINT "B-Tree subroutine (Get) was unable to retrieve the entry."
     SET CURSOR 8, 2
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CALL BtClose (#1)
     CLEAR
     STOP
     END IF
```

```
!-------------------------------------------------------------------------
! Knowing that the entry was retrieved successfully, the program        |
! unpacks (DataString$) and extract the individual fields.              |
!-------------------------------------------------------------------------

  LET PositionSlash1 = POS (DataString$, "\", 1)
  LET FirstChar = 1
  LET LastChar = PositionSlash1 - 1
  LET Street$ = DataString$ [FirstChar:LastChar]

  LET FirstChar = PositionSlash1 + 1
  LET PositionSlash2 = POS (DataString$, "\", FirstChar)
  LET LastChar = PositionSlash2 - 1
  LET City$ = DataString$ [FirstChar:LastChar]

  LET FirstChar = PositionSlash2 + 1
  LET PositionSlash3 = POS (DataString$, "\", FirstChar)
  LET LastChar = PositionSlash3 - 1
  LET State$ = DataString$ [FirstChar:LastChar]

  LET FirstChar = PositionSlash3 + 1
  LET PositionSlash4 = POS (DataString$, "\", FirstChar)
  LET LastChar = PositionSlash4 - 1
  LET Zip$ = DataString$ [FirstChar:LastChar]

  LET FirstChar = PositionSlash4 + 1
  LET LastChar = LEN (DataString$)
  LET Phone$ = DataString$ [FirstChar:LastChar]

!-------------------------------------------------------------------------
! The program then displays the person's name, address, and phone.      |
!-------------------------------------------------------------------------

  CLEAR
  PRINT TAB (2, 2); "Name ......... "; DbKey$
  PRINT TAB (3, 2); "address ....... "; Street$
  PRINT TAB (4, 2); "city ......... "; City$
  PRINT TAB (5, 2); "state ......... "; State$
  PRINT TAB (6, 2); "zip code ...... "; Zip$
  PRINT TAB (7, 2); "phone number .. "; Phone$
```

```
!-------------------------------------------------------------------------
! This is the part which is different from program Bt-Chng.              |
! The following lines of code will prompt whether or not the entry is    |
! to be deleted from the Data Base.                                      |
!-------------------------------------------------------------------------

  SET CURSOR 9, 2
  PRINT TAB (9, 2); "Do you want to delete this entry from the Data
  Base ";
  PRINT ".. Press Y or N .. ";

  LET Choice$ = ""
  DO UNTIL Choice$ = "Y" OR Choice$ = "N"
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     LET Choice$ = UCASE$(CHR$(KeyPressed))
     LOOP

!-------------------------------------------------------------------------
! If the (N) key was pressed, then the program will stop without         |
! deleting the entry from the Data Base.                                 |
!-------------------------------------------------------------------------
  IF Choice$ = "N" THEN
     CALL BtClose (#1)
     CLEAR
     STOP
     END IF

!-------------------------------------------------------------------------
! Getting to this point means that the (Y) key was pressed.              |
! The following line of code will delete the entry from the Data Base.   |
! If the (Delete) subroutine is unable to remove the entry from the      |
! Data Base, then it will set variable (DeletedOk) to 0.                 |
!-------------------------------------------------------------------------

  CALL Delete (DbKey$, #1, DeletedOk)
```

```
!-------------------------------------------------------------------------
! The program is now ready to display a message indicating whether or   |
! not the Data Base entry was successfully deleted.                     |
! After the appropriate message is displayed, the program will wait     |
! for you to respond by pressing a key.  Once a key is pressed, the     |
! program will stop.                                                    |
!-------------------------------------------------------------------------

  SET CURSOR 13, 2
  IF DeletedOk = 0 THEN
     PRINT "B-Tree subroutine (Delete) was unable to remove the entry.";
     ELSE
     PRINT "The Data Base entry was successfully deleted.";
     END IF
  SET CURSOR 15, 2
  PRINT "Press any key to end the program .. ";
  DO WHILE KEY INPUT
     GET KEY KeyPressed
     LOOP
  GET KEY KeyPressed

  CALL BtClose (#1)
  CLEAR
  END
```

Bt-Del uses one new B-Tree command, the **Delete** command.

Delete is a TB-Tree subroutine which will remove a data base entry.

Prior to calling the (Delete) subroutine, store the entry's identifier in variable (Key$).

```
  Call Delete (Key$, #1, DeletedOk)
```

If the entry is in the data base, then (Delete) will delete the entry from the data base. If there were no problems with the deletion process, then (Delete) will set variable (DeletedOk) to 1. If for some reason, the deletion process failed, then (Delete) will set (DeletedOk) to zero.

If the entry is not in the data base, then (Delete) will set (DeletedOk) to zero and return to the main program.

Now that we can Add, Change and Delete entries, we need a program that can show us which entries are still in the data base. Program Bt-Keys shows how to display the Key of each entry in the data base.

Program Bt-Keys

A program to display the data base keys. We want this program to display the key of each entry in the data base.

A detailed explanation of how Bt-Keys works can be found in the comment blocks within the program as listed below:

```
!-------------------------------------------------------------------------
! Program .. Bt-Keys                                                     |
! This program displays the Key of each entry in the Data Base.         |
!-------------------------------------------------------------------------

  LIBRARY "BTREELIB"

  DIM KeyArray$(0)

!-------------------------------------------------------------------------
! First, the program Opens the Data Base.                                |
!-------------------------------------------------------------------------

  CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-------------------------------------------------------------------------
! The next line of code stores the Data Base keys in array (KeyArray$)  |
! The (LoadKeys) subroutine stores the keys in alphabetically sequence. |
!-------------------------------------------------------------------------
  CALL LoadKeys (KeyArray$(), #1)

!-------------------------------------------------------------------------
! The following lines of code will display the keys.                     |
!-------------------------------------------------------------------------

  CLEAR
  PRINT
  FOR i = 1 TO SIZE (KeyArray$)
      IF KeyArray$(i) <> CHR$(127) THEN PRINT " "; KeyArray$(i)
      NEXT i
  PRINT
  PRINT " Press any key to end the program .. ";
  DO WHILE KEY INPUT
     GET KEY KeyPressed
     LOOP
  GET KEY KeyPressed
```

```
CALL BtClose (#1)
CLEAR
END
```

Bt-Keys uses one new B-Tree command, the **LoadKeys** command.

LoadKeys is a B-Tree subroutine which will store the data base keys in an array.

```
Call LoadKeys (KeyArray$(), #1)
```

Subroutine (LoadKeys) not only stores the data base keys in array (KeyArray$), it stores them in sorted order.

The previous program showed us which entries are in the data base, by displaying the Key of each entry. Suppose we want to display the Key of each entry which has a last name of Doe? We still need a way to display selected keys. The last series of programs show how to display selected keys from a data base.

Programs Bt-Find

These programs display selected keys from a data base. We want these programs to do the following:

Bt-Find1, displays the first Key and last Key of the data base. The first Key is the Key which would come first, if the Keys in the data base were sorted alphabetically. The last Key is the Key which would come last, if the Keys in the data base were sorted alphabetically.

Bt-Find2, prompts for part of a person's name, such as the person's last name. Then display the Key which is nearest to the name specified. Nearest Key means the Key which would come next, if the Keys in the data base were sorted alphabetically.

Bt-Find3, prompts for a person's last name. Then display the Key of each entry which has a last name equal to the name specified.

A detailed explanation of how the Bt-Find programs work can be found in the comment blocks within the programs as listed below:

```
!-----------------------------------------------------------------------
! Program .. Bt-Find1                                                  |
! This program displays the first Key and last Key in the Data Base.   |
! The first Key is the Key which would come first, if the keys in      |
! the Data Base were sorted alphabetically.                            |
! The last Key is the Key which would come last, if the keys in        |
! the Data Base were sorted alphabetically.                            |
!-----------------------------------------------------------------------
```

```
  LIBRARY "BTREELIB"

!-------------------------------------------------------------------------
! First, the program Opens the Data Base.                                 |
!-------------------------------------------------------------------------

  CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-------------------------------------------------------------------------
! The following B-Tree subroutine stores in variable (FirstKey$),        |
! the first Key found in the Data Base.                                   |
! If there are no entries in the Data Base, then subroutine (GetFirst)    |
! sets variable (FirstKey$) to the Null string.                           |
!-------------------------------------------------------------------------

  CALL GetFirst (FirstKey$, #1)

!-------------------------------------------------------------------------
! If the Data Base is empty, then the program displays a message and      |
! stops.                                                                  |
!-------------------------------------------------------------------------

  IF FirstKey$ = "" THEN
     CLEAR
     SET CURSOR 2, 2
     PRINT "There are no entries in the Data Base."
     SET CURSOR 4, 2
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CALL BtClose (#1)
     CLEAR
     STOP
     END IF

!-------------------------------------------------------------------------
! Getting to this point means that the Data Base is not empty.           |
! The following lines of code display the first Data Base Key.           |
!-------------------------------------------------------------------------
```

```
  CLEAR
  SET CURSOR 2, 2
  PRINT "The first Key in the Data Base is .. "; FirstKey$;

!-------------------------------------------------------------------------
! The following B-Tree subroutine stores in variable (LastKey$),        |
! the last Key found in the Data Base.                                   |
!-------------------------------------------------------------------------

  CALL GetLast (LastKey$, #1)

!-------------------------------------------------------------------------
! The following lines of code display the last Data Base Key.            |
!-------------------------------------------------------------------------

  SET CURSOR 4, 2
  PRINT "The last Key in the Data Base is ... "; LastKey$;
  SET CURSOR 6, 2
  PRINT "Press any key to end the program ... ";
  DO WHILE KEY INPUT
     GET KEY KeyPressed
     LOOP
  GET KEY KeyPressed

  CALL BtClose (#1)
  CLEAR
  END

!-------------------------------------------------------------------------
! Program .. Bt-Find2                                                    |
! This program displays the Key of the the Data Base entry nearest to    |
! the name you specify.  Nearest Key means the Key which would come      |
! next, if the keys in the Data Base were sorted alphabetically.         |
!-------------------------------------------------------------------------

  LIBRARY "BTREELIB"

!-------------------------------------------------------------------------
! First, the program Opens the Data Base.                                |
!-------------------------------------------------------------------------
```

```
  CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-------------------------------------------------------------------------
! The following lines of code prompt for part of a person's name.       |
! The program uses this name to build the Data Base search Key (DbKey$).|
!-------------------------------------------------------------------------

  CLEAR
  SET CURSOR 2, 2
  INPUT PROMPT "Enter part of a person's name .. ": LastName$

  LET DbKey$ = UCASE$(LastName$)

!-------------------------------------------------------------------------
! The following B-Tree subroutine stores in variable (NearestKey$),     |
! the Data Base Key which is nearest to the name you specified.         |
! Nearest Key means the Key which would come next, if the keys were     |
! sorted alphabetically.                                                |
! If the Data Base is empty, or if there is no Key which comes after    |
! the name you specified, then the (GetNearest) subroutine sets         |
! variable (NearestKey$) equal to the Null string.                      |
!-------------------------------------------------------------------------

  CALL GetNearest (DbKey$, NearestKey$, #1)
!-------------------------------------------------------------------------
! If the Data Base is empty, or the nearest Key was not found,          |
! then the program displays a message and stops.                        |
!-------------------------------------------------------------------------

  IF NearestKey$ = "" THEN
     SET CURSOR 4, 2
     PRINT "Either the Data Base is empty, "
     SET CURSOR 6, 2
     PRINT "or there is no entry which comes after the name you specified."
     SET CURSOR 8, 2
     PRINT "Press any key to end the program .. ";
     DO WHILE KEY INPUT
        GET KEY KeyPressed
        LOOP
     GET KEY KeyPressed
     CALL BtClose (#1)
     CLEAR
     STOP
     END IF
```

```
!-------------------------------------------------------------------------
! Getting to this point means that the (GetNearest) subroutine found    |
! the Key nearest to the name you specified.                            |
! The following lines of code will display the nearest Key and stop.    |
!-------------------------------------------------------------------------

  SET CURSOR 4, 2
  PRINT "The nearest or next name is .... "; NearestKey$;
  SET CURSOR 6, 2
  PRINT "Press any key to end the program .. ";
  DO WHILE KEY INPUT
     GET KEY KeyPressed
     LOOP
  GET KEY KeyPressed

  CALL BtClose (#1)
  CLEAR
  END

!-------------------------------------------------------------------------
! Program .. Bt-Find3                                                   |
! This program displays the Key of each Data Base entry which has a     |
! last name equal to the name you specify.  The program will display    |
! the keys in alphabetical order.                                       |
!-------------------------------------------------------------------------

  LIBRARY "BTREELIB"

!-------------------------------------------------------------------------
! First, the program Opens the Data Base.                               |
!-------------------------------------------------------------------------

  CALL BtOpen ("NameAdPh", DataBaseSize, #1)

!-------------------------------------------------------------------------
! The following lines of code prompt for a person's (last) name.        |
! The program uses this name to build the Data Base search Key.         |
!-------------------------------------------------------------------------

  CLEAR
  SET CURSOR 2, 2
  INPUT PROMPT "Enter the person's last name .. ": LastName$

  LET LastName$ = UCASE$ (LastName$)
```

```
!-------------------------------------------------------------------------
! The following B-Tree subroutine builds a list of Data Base keys.      |
! The subroutine stores the keys in alphabetical order.                 |
! InitNext must be called if the program is going to be using the       |
! B-Tree subroutine (GetNext).                                          |
!-------------------------------------------------------------------------

  CALL InitNext

!-------------------------------------------------------------------------
! The following lines of code cycle through the list of keys created    |
! by the (InitNext) subroutine.                                         |
!                                                                       |
!    The B-Tree subroutine (GetNext) gets the next Key from the         |
!    list, and stores it in variable (NextKey$).                        |
!                                                                       |
!    If variable (NextKey$) is equal to CHR$(127), then either the      |
!    Data Base is empty, or there are no more Keys in the list.         |
!    If this happens, then the program is finished.                     |
!                                                                       |
!    The (LET) statements extract the last name field from the          |
!    (NextKey$) string.                                                 |
!                                                                       |
!    If (NextKey$) has the same last name as the name you specified     |
!    then the Key is displayed.                                         |
!                                                                       |
!    Once (NextKey$) is past the name you specified, then the program   |
!    is finished.                                                       |
!-------------------------------------------------------------------------

  PRINT

  DO UNTIL KeyLastName$ > LastName$

     CALL GetNext (NextKey$, #1)

     IF NextKey$ = CHR$(127) THEN EXIT DO
     LET PositionSpace = POS (NextKey$, " ", 1)
     LET LastChar = PositionSpace - 1
     LET KeyLastName$ = NextKey$ [1: LastChar]
     IF KeyLastName$ = LastName$ THEN PRINT " "; NextKey$
     LOOP
```

```
PRINT
PRINT " Press any key to end the program .. ";
DO WHILE KEY INPUT
   GET KEY KeyPressed
   LOOP
GET KEY KeyPressed

CALL BtClose (#1)
CLEAR
END
```

The Bt-Find programs use five new B-Tree command, **GetFirs**t, **GetLast**, **GetNearest**, **InitNext**, and **GetNext**.

GetFirst is a B-Tree subroutine which will store in variable Key$, the first key in the data base.

```
Call GetFirst (Key$, #1)
```

The first Key is the Key which would come first, if the Keys in the data base were sorted alphabetically.

GetLast is a B-Tree subroutine which will store in variable Key$, the last key in the data base.

```
Call GetLast (Key$, #1)
```

The last Key is the Key which would come last, if the Keys in the data base were sorted alphabetically.

GetNearest is a B-Tree subroutine which will store in variable NextKey$, the data base Key which is nearest to the value stored in variable Key$.

```
Call GetNearest (Key$, NextKey$, #1)
```

The nearest Key is the Key which would come next, if the Keys in the data base were sorted alphabetically.

InitNext is a B-Tree subroutine which builds a list of data base Keys.

```
Call InitNext
```

The keys are stored in the list in alphabetical order. The keys are used by the B-Tree subroutine (GetNext).

GetNext is a B-Tree subroutine which will store in variable Key$, the next key in the data base.

```
Call GetNext (Key$, #1)
```

The next Key is the Key which would come next, if the Keys in the data base were sorted alphabetically.

The B-Tree subroutine (InitNext) must be called prior to the first use of the (GetNext) subroutine.

Another Demo Program

Your *TB-Tree Library* disk contains another sample program to demonstrate the use of the routines contained in this Library. The sample program, BTManage, is a simple, but complete, command-driven utility for creating and maintaining a B-tree file. All of the fundamental operations you will need to include in any program utilizing the *TB-Tree Library* are demonstrated in this rather simple program. In fact, you may later find that it offers a nice framework upon which to build your own application. For the present, you may wish simply to examine the listing of this file to acquaint yourself with its structure and logic.