Inter-Office Memorandum

To	Alto Bcpl Programmers	Date	October 12, 1977
From	Ed McCreight	Location	Palo Alto
Subject	B-Tree Package	Organization	CSL

XEROX

The file <ALTOSOURCE>BTREE.DM contains the five BCPL code modules BTreeRead.bcpl, BTreeWrtMS0.bcpl, BTreeWrtMS1.bcpl, BTreeWrtMS2.bcpl, and BTreeDel.bcpl, plus a declarations module BTree.decl (plus BTreeCheck.bcpl and some other model Bcpl code). These modules, with considerable support from user-supplied routines, implement B-Trees with variable-length records for the Alto. (If you don't know what a B-Tree is, put this memo down for a while and read section 6.2.4 of Knuth's *The Art of Computer Programming*, volume 3.) Subroutines are assigned to modules in such a way that if one only *reads* from the B-Tree, only one modules is required; *writing* requires up to four; *deleting* may require all five.

The B-Tree package infers the lengths of B-Tree records and order among B-Tree records by asking questions of user-supplied routines. It also assigns responsibility for storing and recalling B-Tree pages to/from a disk (or whatever) to user-supplied routines. These routines are provided to the B-Tree procedures via a *tree handle*, a block of memory that is supplied as a parameter in all calls to the B-Tree package. This tree handle must be initialized before calling the B-Tree package. The package is re-entrant (if the user-supplied routines are), and can be working on several tree handles at once.

What the Package Needs

The user must supply the following routines in the tree handle (see BTree.decl for the exact format of the tree handle: it's the structure called TREE):

ReadBTreePage(BTreeHandle, pageNumber) = core address
The routine must read the desired page into core somewhere and return the address of where it was put. The page may be flushed from core at any later time (within reason) except if it is locked (see below).

WriteBTreePage(BTreeHandle, pageNumber) = core address
The routine must read the desired page into core somewhere and returns the address of where it was put. In addition, the B-Tree routines will immediately alter the page in core, so that it must subsequently be written out before being flushed from core.

LockBTreePtr(BTreeHandle, lv pointer)
This call notifies that **pointer** must be checked before flushing any B-Tree page from core. If **pointer** points into a B-Tree page, that page may not be moved or removed.

UnlockBTreePtr(BTreeHandle, lv pointer)
Notification that **pointer** need not be checked any more.

AllocateBTreePage(BTreeHandle) = pageNumber
This routine must return the number of a page not currently being used in the B-Tree.

FreeBTreePage(BTreeHandle, pageNumber)
Notification that the numbered page is now no longer being used in the B-Tree.

CompareKeyRtn(Key, Record) = {-1, 0, 1}
This routine must compare an isolated key (whatever that is) with a record and say whether the key is less than, equal to, or greater than the record.

LengthRtn(Record) = length
This routine must return the length of the record in words.

It is no coincidence that the first four routines mesh hand in glove with four nearly identical routines in the VMEM package. I shall eventually make a nice stand-alone **OpenTree** procedure that uses the ISF and VMEM packages in the simplest possible ways and creates a proper tree handle. For the moment, alas, users must follow the models in IFSBTREERES and IFSBTREESWAP, which are also included in the dump file for inspirational purposes only. I know that this puts a pretty high potential barrier in front of somebody who wants to use B-Trees in a simple application, but the only applications to appear thus far have been far from simple, and ultimately needed the full generality of the IFS environment.

What The Package Will Do
The following routines are part of the BTREEREAD module. In these routines, **CompareKeyRtn** is an optional comparison routine which, if present, is used in preference to the comparison routine specified in the tree handle. The ordering relation R' used to search the tree must be a *weakening* of the relation R used to create the tree. That is, if a<b in R', then a<b in R. Similarly, if a>b in R', then a>b in R. This feature can be useful, for example, when one wishes to store capitalizations distinctly while sometimes having a search match any capitalization. In this case R would sort first on letter content, and then (if all letters were the same) on capitalization. R' would simply sort on letter content and call all keys containing the same letters in the same order equal.

ReadRecLE(TreeHandle, Key, CompareKeyRtn) = record copy
This returns either 0 or a pointer to a copy of the tree record with the greatest key less than or equal to **Key**. When the user is finished with the record copy, he must FREE it into **TreeHandle**>>TREE.Zone.

MapTree(TreeHandle, StartKey, Function, Param, CompareKeyRtn, dontCopy)
This function is similar to ReadRecLE. However, instead of returning a pointer to a copy of the tree record, it passes that pointer to the user-supplied **Function** which should take three arguments. The first one will be a pointer to a copy of the record, or to the live record itself if **dontCopy** is true, the second one will be **Param**, and the third will be uninteresting. When it is finished, **Function** is responsible for FREEing the record copy into **TreeHandle**>>TREE.Zone. If **Function** returns the value *true*, then it is called again on the next larger record in the tree, and so on until either **Function** returns the value *false* or the largest record in the tree has been processed. **MapTree** itself returns the value *false* if **Function** returned *false*, and *true* if there are no more records to process.

The following external subroutine is contained in the BTREEWRTMS0 module:

UpdateRecord(TreeHandle, Key, RecordGenerator, Param, CompareKeyRtn)
The user-supplied function **RecordGenerator** is called with two parameters. The first parameter is either 0 or a pointer to a copy of a record in the tree whose key is equal (according to **CompareKeyRtn** or the key-comparison routine specified by the tree handle) to **Key**. The second parameter is **Param**. **RecordGenerator** is expected to produce a record whose key is equal to **Key** (this is checked). That new record will replace the original one (if any) in the tree. Finally the record produced by **RecordGenerator** will be FREEd into **TreeHandle**>>TREE.Zone.

The following external subroutine is contained in the BTREEDEL module:

DeleteKey(TreeHandle, Key, CompareKeyRtn)
The record whose key is equal to **Key** is deleted from the tree. The value returned is *true* if this was done, and *false* if no such record was contained in the tree.