NDX -- BTREE INDEXING ROUTINES
Version 3.03

INTRODUCTION

This package implements yet another variant of the balanced tree indexing method (Btree) described in section 6.2.4 of D. E. Knuth's "Sorting and Searching", volume 3 of "The Art of Programming." Its features are:

1. Files with variable length records can be indexed. Data file format is arbitrary, since an index is stored as a separate file.

2. Keys are variable length ASCIZ strings. The package allows retrieval on heads of index keys: when retrieving on the search key "FOO", records with keys "FOO", "FOO1", "FOOP" and "FOOPU" may be retrieved in that order. That is, a search key may be matched by any key of which it is a head. Matching can be case sensitive or case-insensitive. Search behavior is specified when the index is opened.

3. ISAM access to index entries is available; it may be first-in-first-out or last-in-last-out, as specified when the index file is created.

4. Index file records are cached. A single cache is used when more than one index file is open.

5. The source file is supplied. The package can be compiled using Turbo C version 1.5+ or Microsoft C version 5.0+.

INDEX FILE FORMAT

The index file begins with a short header, padded with enough bytes to fill a 512 byte disk sector. (See NDX.H for the format of the header and other data structures.) Nodes in the tree are 1024 bytes in length on the disk. Each starts with a value -- that of the node if it is currently in use, or that of the next node in the chain of free nodes on the disk. The number of bytes occupied by keys in the node follows, and then the keys.

Each Key has three fields. First is the file offset, if not zero, of a lower level node which has keys less than or equal to the current key, followed by the offset (or record number) in the data file which corresponds to the current key. Finally, the ASCIZ key itself is stored. Only as many bytes as are necessary are stored.

Immediately following the last key in the node, a pointer to the next lower level may be stored; the field end_keys of NODE does not include the length of this pointer. Starting at the root node, if all pointers to lower level nodes were replaced by the content of those nodes, the result would be all keys in the index in order.

The index structure Index is allocated and initialized when the index file is opened. It contains:

1. A copy of the file header

2. A Key structure which holds the key resulting from index file operations.

3. A stack holding the path from the root of the index to the current key entry.

4. The index file name and handle.

5. The string comparison function to be used during key retrieval.

USAGE

The constructor

Index::Index(char *indexname, unsigned mode, CFN compfn)

creates an empty index file. The comparison function, if NULL, defaults to strcmp. The mode is one of:

NODUPS No duplicate keys allowed

FIFO Duplicate keys are retrieved first-in-first-out.

LIFO Duplicate keys are retrieved last-in-last out.

The mode is stored in the index file header. A NULL second argument defaults to NODUPS. The third argument is the routine to be used for key matching -- if NULL, strcmp is used.

Index::Index(char *indexname, CFN compfn)

is used to open an existing index file, while the destructor closes an index file. Using the type cast (CFN), string comparison functions such as strcmp, strncmp, etc. may be used.

DWORD Index::insert(const char *key, DWORD value);

adds a key to the index, except when the mode is NODUPS and the key is already present. In this case, the value specified in the call replaces the previous value. If successful, insert returns the specified DWORD and otherwise it returns zero.

The value can be a long unsigned seek address used to reference the data file, or just a serial record number when the data file has fixed length records. A valid value may not be zero. The comparison function used for insertion is always strcmp.

Index::remove(const char *key, DWORD value);

deletes the key from the index. If this leaves an empty node, the node is placed in the freelist. The second argument is required due to the possibility of multiple occurrences of the same key. Returns 0 for failure and 1 for success.

DWORD Index::find(const char *key);

attempts to find the specified key, according to the comparison function specified when the file was opened and the mode specified when the file was created. The value part of the key found is returned on success; zero is returned on failure.

In order to find all occurrences of a key when FIFO or LIFO operation is in effect, use Index::find with a NULL second argument after the first occurrence has been found.

DWORD Index::first(unsigned last);

positions the index file pointer to the first (last) key in the index. The "file pointer" is, strictly speaking, the top stack entry and contains the seek address of the node and the offset of the key in that node. The remainder of the stack records information required for traveling over the index tree starting from the file pointer, as if the index were flat rather than a tree. Zero is returned if the index is empty, else the value part of the key located is returned.

DWORD Index::next();

DWORD Index::prev();

These routines change the file pointer to the next or previous key and return the value part of that key, or zero if the current file pointer was at the end or beginning of the index.

Note that Index::find, Index::insert and Index::remove position the file pointer. In the latter case, the file pointer is set to the key following the deleted key. Note further that each successful key retrieval stores a copy of the current key structure in the index structure. This key structure may be accessed using

```
const Key Index::key();
```

For examples of the use of the routines, see the test program.

IMPROVEMENTS

Knuth suggests ways in which the Btree algorithms may be improved. If you decide to experiment, be prepared for an interesting [sic] experience. Be particularly wary of any changes you attempt to make in Index::remove -- updating the index to properly reflect a deletion is an extremely tricky affair. Even xnext and xprev, which look deceptively simple, are easy to upset if you try to change the order in which things are done.

CHANGES

My address is:

George H. Mealy

38 Gilson Road
Scituate, MA 02066
USA

(617) 545-1727

Version 2.02 corrected an egregious blunder in the Index::insert routine -- thanks to John A. Matzen. I also fixed a bug in Index::find, thanks to strong type checking and use of the typedef CFN.

The package has been recompiled using Borland C++ v3.1. Version 3.01 corrects another bug in Index::insert. NDX.H has been changed to make a few more items accessible to user programs. The routine Index::key is supplied to allow safe access to the current key.

All previous versions contained a serious bug in Index::newnode. Since the order in which nodes are first written out to the index file is unpredictable, it is vital that chsize() is called when the length of the file is extended! Version 3.02 includes this fix together with a few minor cosmetic changes.

Prior to version 3.03, the state of the index following a call to Index::remove was unpredictable. The index file position is now defined to be at the key which immediately followed the removed key. In addition, the index key count is stored in the index and can be verified by use of the routine Index::isvalid. Index::count may be used to retrieve the index key count.