HashFile

INHERITS FROM HashTable

REQUIRES HEADER FILES HashFile.h

DEFINED BY C.D. Lane (lane@sumex-aim.stanford.edu)

CLASS DESCRIPTION

HashFile implements a bridge between the hashed database library functions, db(3) (used to implement the 'defaults' database and Digital Librarian index files), and the HashTable object. It can be viewed as an alternate interface to the database library functions or as a type of HashTable that survives program execution.

Except for initial object creation, methods are a superset of those provided by HashTable and a HashFile can be substituted for a HashTable in most code. In order to maintain this similarity, the database file is closed when the **free** method is called on the HashFile object, rather than having a separate method.

Database files created using the HashFile object can be manipulated using the utilities dbCatenate, dbCompare, dbDescribe etc. on /usr/lib/database (see dbCatenate(1)).

The HashFile object uses a HashTable object as a buffer to optimize multiple access to keys/values. Keys and values can be of type **id**, **int**, **void ***, **char ***, or any other 32-bit quantity that can be described by a type string and handed to **NXReadType()**. Descriptions must be invariant strings and are restricted to encode 32-bit quantities, typically the following: "@" (id), "*" (char *), "i" (int)

Note that the "!" (other) type that HashTable accepts doesn't really make sense here as it has a different meaning to **NXReadType()**. Also, when using type **id**, "@", for keys, objects that have the same content, though still distinct, are still mapped into the same database file entry, thus not making for very useful keys. For hash files, the printed representations of the keys, not their pointers, are compared for equality.

When the description is "%", hashing and equality are same as for "*". On reading, however, the string is uniqued, using the **NXUniqueString()** function.

HashFiles must generally meet the same restrictions that a HashTable must satisfy.

Since the **newFromFile:** method will create a database file if one doesn't exist, the factory method **isHashFile:** can be used to determine if a file exists before opening it.

INSTANCE VARIABLES

Inherited from HashTable	const char	*keyDesc;
	const char	*valueDesc;
Declared in HashFile	Database	*db;
	Data	d;
	const char	*filename;
	BOOL	readOnly;

db	open Database descriptor
d	Data structure for accessing database file
filename	Database name
readOnly	Value indicating if database compressed or readonly

METHOD TYPES

Initializing and freeing a HashFile	- initFromFile: - initFromFile:keyDesc: - initFromFile:keyDesc:valueDesc: + isHashFile: - free: - freeKeys:values: - freeObjects
Emptying and copying a HashFile	- copy - empty
Manipulating table elements	- count - insertKey:value: - isKey: - removeKey: - valueForKey:
Iterating over all elements	- initState - nextState:key:value:
Archiving	- read: - write:

CLASS METHODS

isHashFile:

+ (BOOL) **isHashFile:**(const char *)*name*

True if database *name* exists (i.e., if directory and leaf files *name.[DL]* exist).

newFromFile:

+ **newFromFile:**(const char *)*name*

Returns a HashFile object that maps objects to objects, creating the directory and leaf files *name.[DL]* if they don't already exist. Returns **nil** on failure.

newFromFile:keyDesc:

+ **newFromFile:**(const char *)*name* **keyDesc:**(const char *)*aKeyDesc*

Returns a HashFile object that maps keys as described with *aKeyDesc* to objects.

newFromFile:keyDesc:valueDesc:

+ **newFromFile:**(const char *)*name* **keyDesc:**(const char *)*aKeyDesc*
valueDesc:(const char *)*aValueDesc*

Returns a HashFile object that maps keys and values as described with *aKeyDesc* and *aValueDesc*.

INSTANCE METHODS

copy

– **copy**

Returns a new HashFile. Neither keys nor values are copied. ***Not implemented.***

count

– (unsigned)**count**

Returns the number of objects in the database (>= the HashTable buffer's count).

empty

– **empty**

Empties the database file and HashTable buffer.

free

– **free**

Closes the database, closing the directory and leaf files *name.[DL]* and flushing any unwritten data to disk. Deallocates the HashTable buffer, but not the objects in it.

freeKeys:values:

– **freeKeys:**(void (*)(void *))*keyFunc* **values:**(void (*)(void *))*valueFunc*

Conditionally deallocates the database file's elements but does not deallocate the file itself. ***Not implemented.***

freeObjects

– **freeObjects**

Deallocates every object in the HashTable buffer, but not the HashTable itself. Strings are not recovered. Has no effect on the database file. To empty the database file, use the **empty** method.

initState

– (NXHashState)**initState**

Iterating over all elements of a database file involves setting up an iteration state, conceptually private to HashFile, and then progressing until all entries have been visited. An example of counting elements in a table follows:

```
unsigned count = 0;
const void *key, *value;
NXHashState state = [hashfile initState];

while([hashfile nextState:&state key:&key value:&value])
   count++;
```

initState begins the process of iteration through the database. You cannot have multiple **NXHashState** instances active on the same HashFile at the same time.

See also: **nextState:key:value:**

insertKey:value:

– (void *)**insertKey:**(const void *)*aKey* **value:**(void *)*aValue*

Adds or updates *akey*/*avalue* pair. Returns **(void *) nil** on failure (eg. if database is read-only).

isKey:

– (BOOL)**isKey:**(const void *)*aKey*

Indicates whether *aKey* is in the database.

nextState:key:value:

– (BOOL)**nextState:**(NXHashState *)*aState*
key:(const void **)*aKey*
value:(void **)*aValue*

Moves to the next entry in the database. No **count**, **insertKey:**, **isKey:**, **removeKey:** nor **valueForKey:** should be done while iterating through the file. The order followed is uninteresting, being determined by a hash function. Returns **NO** when the end of the database is reached. The values and keys are not entered into the HashTable buffer.

See also: **initState**

read:

– **read:**(NXTypedStream *)*stream*

Reads the HashFile from the typed stream *stream*. ***Not implemented.***

removeKey:

– (void *)**removeKey:**(const void *)*aKey*

Removes *akey*/*avalue* pair. Always returns **(void *) nil** (unfortunately).

valueForKey:

– (void *)**valueForKey:**(const void *)*aKey*

Returns the value mapped to *aKey*. Returns **nil** on failure (eg. if *aKey* is not in the database file).

write:

– **write:**(NXTypedStream *)*stream*

Writes the HashFile to the typed stream *stream*. ***Not implemented.***

REFERENCES

This document is derived from the following NeXT documents in /NextLibrary/Documentation:

NeXT/SysRefMan/22_ClassSpecs/CommonClasses/HashTable.wn

Unix/ManPages/man3/db.3

Unix/ManPages/man1/dbCatenate.1