HashTable

Inherits From: Object

Declared In: objc/HashTable.h

Class Description

The HashTable class defines objects that store associations of keys and values. You use a HashTable object when you need a convenient and efficient way to retrieve the data associated with an arbitrary key. Internally, a hash table locates the key and associated object according to the value returned by applying a *hashing* function to the key. However, the hashing operation is provided automatically by the HashTable's methods, so that the methods that add an association to a HashTable (or return an association, given its key) accept and return the key values directly, not their hashed forms.

In a HashTable object, keys must be of the same type (so that the same hashing function can be applied to each of them), and associated values must be of the same type. The types of the keys and the values are established when the HashTable is initialized. The **initKeyDesc:valueDesc:…** methods take arguments that let you specify key type and value type independently. The **initKeyDesc:** method specifies the type of the keys but assumes that the associated values are **id**s. The **init** method assumes that both keys and associated values are of type **id** (object pointers). The following characters are used as HashTable descriptions (that is, as arguments to the **initKeyDesc:** or **initKeyDesc:valueDesc:** methods):

Character	Type
@	id
*	char *
%	NXAtom
i	int
!	other

Hashing Algorithm and Tests for Equality

The class uses three different algorithms, selected according to the description of the keys. For keys that are of type "object", the HashTable sends itself a **hash** message (inherited from Object). For keys that are strings, it uses a string hashing function. For all other cases, it uses a generic integer hashing function.

To test whether a proposed key is equal to a key already included in the HashTable, keys that are objects are compared using an **isEqual:** message. If two keys are equal in the sense of **isEqual:**, then their hashed values must be equal.

Keys that are strings are compared using a string comparison. Note that the HashTable object keeps only a pointer to the string used as a key (without making a copy of the string), so the string to which it points must never change as long as the association remains in the table.

If you're creating a HashTable whose keys are List or Storage objects, note that these classes have an **isEqual:** method but no **hash** method; you can either subclass or define a **hash** method.

When freeing a HashTable, only object keys or object values are freed.

When a HashTable is archived, data is archived according to its type description. For keys or values whose description is "**%**", upon reading to reconstitute an archived HashTable, each such string is reconstructed by again calling the **NXUniqueString()** function to assure that it is unique in the new context.

Function Interface to Hash Tables

When even greater efficiency of storage and access is required, consider using the C function interface to hash tables rather than the HashTable class (see **NXCreateHashTable()**).

Related Classes

Two other classes for storage and retrieval are NXStringTable and List. An NXStringTable object is a hash table specialized for the situation in which both keys and values are character strings. A List stores a sequential collection of objects; however, it stores the objects (that is, the pointers to them) without keys, so the time required to find a particular element in a List grows linearly with the number of elements.)

Instance Variables

unsigned int **count**;
const char ***keyDesc**;
const char ***valueDesc**;

count	Current number of associations
keyDesc	Description (character representing the type) of keys
valueDesc	Description (character representing the type) of values

Method Types

Initializing and freeing a HashTable	– init – initKeyDesc: – initKeyDesc:valueDesc: – initKeyDesc:valueDesc:capacity: – free – freeObjects – freeKeys:values: – empty
Copying a HashTable	– copyFromZone:
Manipulating table associations	– count – isKey: – valueForKey: – insertKey:value: – removeKey:
Iterating over all associations	– initState – nextState:key:value:
Archiving	– read: – write:

Instance Methods

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new HashTable of the same size as the receiving object. Memory for the new HashTable is allocated from *zone*. Keys and values are copied.

count

– (unsigned int)**count**

Returns the number of objects in the table.

empty

– **empty**

Empties the HashTable but retains its capacity.

free

– **free**

Deallocates the HashTable (but not the objects that its associations point to).

freeKeys:values:

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

Conditionally deallocates the HashTable's associations but does not deallocate the table itself.

freeObjects

– **freeObjects**

Deallocates every object in the HashTable, but not the HashTable itself. Strings are not recovered.

init

– **init**

Initializes a new HashTable to map keys of type “object” to values of type “object.” Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:

– **initKeyDesc:**(const char *)*aKeyDesc*

Initializes a new HashTable to map keys as described by *aKeyDesc* to object values. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:valueDesc:

– **initKeyDesc:**(const char *)*aKeyDesc* **valueDesc:**(const char *)*aValueDesc*

Initializes a new HashTable to map keys and values as described by *aKeyDesc* and *aValueDesc*. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:valueDesc:capacity:

– **initKeyDesc:**(const char *)*aKeyDesc*
valueDesc:(const char *)*aValueDesc*
capacity:(unsigned int)*aCapacity*

Initializes a new HashTable. This is the designated initializer for HashTable objects: If you subclass HashTable, your subclass’s designated initializer must maintain the initializer chain by sending a message to **super** to invoke this method. See the introduction to the class specifications for more information.

A HashTable initialized by this method maps keys and values as described by *aKeyDesc* and *aValueDesc*. The argument *aCapacity* is given only as a hint; you can use 0 to create a table of minimal size. As more space is needed, it will be allocated automatically, each time doubling the table’s capacity. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initState

– (NXHashState)**initState**

Returns an NXHashState structure that's required when iterating through the HashTable. Iterating through all of a HashTable's associations involves setting up an iteration state, conceptually private to HashTable, and then progressing until all entries have been visited. Here's an example of visiting all the associations in a HashTable called **table** (and just counting them):

```
unsigned int count = 0;
const   void  *key;
        void  *value;
NXHashState  state = [table initState];
while ([table nextState: &state key: &key value: &value])
    count++;
```

See also: – **nextState:key:value:**

insertKey:value:

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

Adds or updates a key and value pair, as specified by *aKey* and *aValue*. If *aKey* is already in the hash table, it's associated with *aValue* and its previously associated value is returned. Otherwise, **insertKey:value:** returns **nil**.

See also: – **removeKey:**

isKey:

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

Returns YES if *aKey* is in the table, otherwise NO.

See also: – **valueForKey:**

nextState:key:value:

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

Moves to the next entry in the HashTable and provides the addresses of pointers to its key/value pair. No **insertKey:** or **removeKey:** should be done while iterating through the table. Returns NO when there are no more entries in the table; otherwise, returns YES. If there are no more entries, *aKey* and *aValue* are set to NULL.

See also: – **initState**

read:

– **read:**(NXTypedStream *)*stream*

Reads the HashTable from the typed stream *stream*. Returns **self**.

See also: – **write:**

removeKey:

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

Removes the hash table entry identified by *aKey*. Always returns **nil**.

See also: – **insertKey:value:**

valueForKey:

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

Returns the value mapped to *aKey*. Returns **nil** if *aKey* is not in the table.

See also: – **isKey:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the HashTable to the typed stream *stream*. Returns **self**.

See also: – **read:**