init
initKeyDesc:
initKeyDesc:valueDesc:
initKeyDesc:valueDesc:capacity:
free
freeObjects
freeKeys:values:
empty

Copying a HashTable copyFromZone:

copyFromZone:(NXZone *)zone

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

(unsigned int)count

Returns the number of objects in the table.

empty

Empties the HashTable but retains its capacity.

free

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

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

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

freeObjects

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

init

Initializes a new HashTable to map keys of type ªobjectº to values of type ªobject.º Returns self.
initKeyDesc:key:value:capacity:

initKeyDesc:(const char *)aKeyDesc

Initializes a new HashTable to map keys as described by aKeyDesc to object values. Returns self.
initKeyDesc:key:value: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 subcl
subclass's designated initializer must maintain the initializer chain by sending a message to super t
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 aValu
aCapacity is given only as a hint you can use 0 to create a table of minimal size. As more space is
allocated automatically, each time doubling the table's capacity. Returns self.

initKeyDesc:key:value:capacity:

(NXHashState)initState

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

nextState:key: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 h
associated with aValue and its previously associated value is returned. Otherwise, insertKey:value

removeKey:

(BOOL)isKey:(const void *)aKey

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

valueForKey:

(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 pa
removeKey: should be done while iterating through the table. Returns NO when there are no more
otherwise, returns YES. If there are no more entries, aKey and aValue are set to NULL.

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

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

insertKey:value:

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

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

isKey:

write:(NXTypedStream *)stream

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

read: