Generic Hash

```
//*****************************
// Written by Peter Golde
// Copyright (c) 2004-2007, Wintellect
//
// Use and restribution of this code is subject to the license agreement
// contained in the file “License.txt” accompanying this file.
//*****************************

using System;
using System.Diagnostics;
using System.Collections.Generic;
using System.Runtime.Serialization;

namespace Wintellect.PowerCollections
{
///

/// The base implementation for various collections classes that use hash tables
/// as part of their implementation. This class should not (and can not) be
/// used directly by end users; it's only for internal use by the collections package. The Hash
/// does not handle duplicate values.
///
///
/// The Hash manages items of type T, and uses a IComparer&lt;ItemTYpe&gt; that
/// hashes compares items to hash items into the table.
///
internal class Hash : IEnumerable//, ISerializable, IDeserializationCallback
{
// NOTE: If you add new member variables, you very well may
```

```
need to change the serialization
// code to serialize that member.
private IEqualityComparer equalityComparer; // interface for
comparing elements

private int count; // The count of elements in the table.
private int usedSlots; // Includes real elements and deleted
elements with the collision bit on. Used to determine
// when we need to resize.
private int totalSlots; // Size of the table. Always a power
of two.
private float loadFactor; // maximal load factor for the
table.
private int thresholdGrow; // floor(totalSlots * loadFactor);
private int thresholdShrink; // thresholdGrow / 3.
private int hashMask; // Mask to convert hash values to the
size of the table.
private int secondaryShift; // Shift to get the secondary skip
value.

private Slot[] table; // The hash table.

private int changeStamp; // An integer that is changed every
time the table structurally changes.
// Used so that enumerations throw an exception if the tree is
changed
// during enumeration.

private const int MINSIZE = 16; // minimum number of slots.

//private SerializationInfo serializationInfo; // Info used
during deserialization.

///

/// The structure that has each slot in the hash table. Each
slot has three parts:
/// 1. The collision bit. Indicates whether some item visited
this slot but had to
/// keep looking because the slot was full.
```

```
/// 2. 31-bit full hash value of the item. If zero, the slot
is empty.
/// 3. The item itself.
///
struct Slot
{
private uint hash_collision; // Lower 31 bits: the hash value.
Top bit: the collision bit.
public T item; // The item.

///

/// The full hash value associated with the value in this
slot, or zero
/// if the slot is empty.
///
public int HashValue
{
get
{
return (int)(hash_collision & 0x7FFFFFFF);
}
set
{
Debug.Assert((value & 0x80000000) == 0); // make sure sign bit
isn't set.
hash_collision = (uint)value | (hash_collision & 0x80000000);
}
}

///

/// Is this slot empty?
///
public bool Empty
{
get
{
return HashValue == 0;
```

```
}
}

///

/// Clear this slot, leaving the collision bit alone.
///
public void Clear()
{
HashValue = 0;
item = default(T); // Done to avoid keeping things alive that
shouldn't be.
}

///

/// The “Collision” bit indicates that some value hit this
slot and
/// collided, so had to try another slot.
///
public bool Collision
{
get
{
return (hash_collision & 0x80000000) != 0;
}
set
{
if (value)
hash_collision |= 0x80000000;
else
hash_collision &= 0x7FFFFFFF;
}
}
}

///

/// Constructor. Create a new hash table.
///
```

```
/// The comparer to use to compare items.  public
Hash(IEqualityComparer equalityComparer)
{
this.equalityComparer = equalityComparer;
this.loadFactor = 0.70F; // default load factor.
}

///

///  Gets the current enumeration stamp.  Call
CheckEnumerationStamp later
/// with this value to throw an exception if the hash table is
changed.
///
/// The current enumeration stamp.
internal int GetEnumerationStamp()
{
return changeStamp;
}

///

/// Must be called whenever there is a structural change in
the tree. Causes
/// changeStamp to be changed, which causes any in-progress
enumerations
/// to throw exceptions.
///
internal void StopEnumerations()
{
++changeStamp;
}

///

/// Checks the given stamp against the current change stamp.
If different, the
/// collection has changed during enumeration and an
InvalidOperationException
/// must be thrown
///
```

```
/// changeStamp at the start of the enumeration. internal void CheckEnumerationStamp(int startStamp)
{
if (startStamp != changeStamp)
{
throw new InvalidOperationException(“Change During Enumeration”);
}
}
///

/// Gets the hash code for an object using a comparer. Correctly handles
/// null.
///
/// Item to get hash code for. Can be null. /// The comparer to use. /// The hash code for the item.
public static int GetHashCode(T item, IEqualityComparer equalityComparer)
{
if (item == null)
return 0x1786E23C;
else
return equalityComparer.GetHashCode(item);
}
///

/// Gets the full hash code for an item.
///
/// Item to get hash code for. /// The full hash code. It is never zero.
private int GetFullHash(T item)
{
uint hash;

hash = (uint)GetHashCode(item, equalityComparer);

// The .NET framework tends to produce pretty bad hash codes.
// Scramble them up to be much more random!
```

```
hash += ~(hash <> 10);
hash += (hash <> 6);
hash += ~(hash <> 16);
hash &= 0x7FFFFFFF;
if (hash == 0)
hash = 0x7FFFFFFF; // Make sure it isn't zero.
return (int)hash;
}

///

/// Get the initial bucket number and skip amount from the full hash value.
///
/// The full hash value. /// Returns the initial bucket. Always in the range 0..(totalSlots – 1). /// Returns the skip values. Always odd in the range 0..(totalSlots – 1). private void GetHashValuesFromFullHash(int hash, out int initialBucket, out int skip)
{
initialBucket = hash & hashMask;

// The skip value must be relatively prime to the table size. Since the table size is a
// power of two, any odd number is relatively prime, so oring in 1 will do it.
skip = ((hash >> secondaryShift) & hashMask) | 1;
}

///

/// Gets the full hash value, initial bucket number, and skip amount for an item.
///
/// Item to get hash value of. /// Returns the initial bucket. Always in the range 0..(totalSlots – 1). /// Returns the skip values. Always odd in the range 0..(totalSlots – 1). /// The full hash value. This is never zero.
private int GetHashValues(T item, out int initialBucket, out
```

```
int skip)
{
int hash = GetFullHash(item);
GetHashValuesFromFullHash(hash, out initialBucket, out skip);
return hash;
}

///

/// Make sure there are enough slots in the hash table that
/// items can be inserted into the table.
///
/// Number of additional items we are inserting. private void
EnsureEnoughSlots(int additionalItems)
{
StopEnumerations();

if (usedSlots + additionalItems > thresholdGrow)
{
// We need to expand the table. Figure out to what size.
int newSize;

newSize = Math.Max(totalSlots, MINSIZE);
while ((int)(newSize * loadFactor) < usedSlots +
additionalItems) { newSize *= 2; if (newSize <= 0) { // Must
have overflowed the size of an int. Hard to believe we didn't
run out of memory first. throw new
InvalidOperationException("CollectionTooLarge"); } }
ResizeTable(newSize); } } ///
/// Check if the number of items in the table is small enough
that
/// we should shrink the table again.
///

private void ShrinkIfNeeded()
{
if (count < thresholdShrink) { int newSize; if (count > 0)
{
newSize = MINSIZE;
```

```
while ((int)(newSize * loadFactor) < count) newSize *= 2; } else { // We've removed all the elements. Shrink to zero. newSize = 0; } ResizeTable(newSize); } } ///
/// Given the size of a hash table, compute the “secondary shift” value – the shift
/// that is used to determine the skip amount for collision resolution.
///

/// The new size of the table. /// The secondary skip amount.
private static int GetSecondaryShift(int newSize)
{
int x = newSize – 2; // x is of the form 0000111110 – a single string of 1's followed by a single zero.
int secondaryShift = 0;

// Keep shifting x until it is the set of bits we want to extract: it be the highest bits possible,
// but can't overflow into the sign bit.
while ((x & 0x40000000) == 0)
{
x < 0)
table = new Slot[totalSlots];
else
table = null;

if (oldTable != null && table != null)
{
foreach (Slot oldSlot in oldTable)
{
int hash, bucket, skip;

hash = oldSlot.HashValue;
GetHashValuesFromFullHash(hash, out bucket, out skip);

// Find an empty bucket.
while (!table[bucket].Empty)
{
```

```
// The slot is used, but isn't our item. Set the collision bit
and keep looking.
table[bucket].Collision = true;
bucket = (bucket + skip) & hashMask;
}

// We found an empty bucket.
table[bucket].HashValue = hash;
table[bucket].item = oldSlot.item;
}
}

usedSlots = count; // no deleted elements have the collision
bit on now.
}

///

/// Get the number of items in the hash table.
///
/// The number of items stored in the hash table.
public int ElementCount
{
get
{
return count;
}
}

///

/// Get the number of slots in the hash table. Exposed
internally
/// for testing purposes.
///
/// The number of slots in the hash table.
internal int SlotCount
{
get
```

```
{
return totalSlots;
}
}

///

/// Get or change the load factor. Changing the load factor may cause
/// the size of the table to grow or shrink accordingly.
///
///
public float LoadFactor
{
get
{
return loadFactor;
}
set
{
// Don't allow hopelessly inefficient load factors.
if (value < 0.25 || value > 0.95)
throw new ArgumentOutOfRangeException(“value”, value.ToString());
//throw new ArgumentOutOfRangeException(“value”, value, Strings.InvalidLoadFactor);

StopEnumerations();

bool maybeExpand = value < loadFactor; // May need to expand or shrink the table -- which? // Update loadFactor and thresholds. loadFactor = value; thresholdGrow = (int)(totalSlots * loadFactor); thresholdShrink = thresholdGrow / 3; if (thresholdShrink <= MINSIZE) thresholdShrink = 1; // Possibly expand or shrink the table. if (maybeExpand) EnsureEnoughSlots(0); else ShrinkIfNeeded(); } } ///
/// Insert a new item into the hash table. If a duplicate item
```

```
exists, can replace or
/// do nothing.
///

/// The item to insert. /// If true, duplicate items are replaced. If false, nothing
/// is done if a duplicate already exists. /// If a duplicate was found, returns it (whether replaced or not). /// True if no duplicate existed, false if a duplicate was found.
public bool Insert(T item, bool replaceOnDuplicate, out T previous)
{
int hash, bucket, skip;
int emptyBucket = -1; // If >= 0, an empty bucket we can use for a true insert
bool duplicateMightExist = true; // If true, still the possibility that a duplicate exists.

EnsureEnoughSlots(1); // Ensure enough room to insert. Also stops enumerations.

hash = GetHashValues(item, out bucket, out skip);

for (; ; )
{
if (table[bucket].Empty)
{
// Record the location of the first empty bucket seen. This is where the item will
// go if no duplicate exists.
if (emptyBucket == -1)
emptyBucket = bucket;

if (!duplicateMightExist || !table[bucket].Collision)
{
// There can't be a duplicate further on, because a bucket with the collision bit
// clear was found (here or earlier). We have the place to
```

```
insert.
break;
}
}
else if (table[bucket].HashValue == hash &&
equalityComparer.Equals(table[bucket].item, item))
{
// We found a duplicate item. Replace it if requested to.
previous = table[bucket].item;
if (replaceOnDuplicate)
table[bucket].item = item;
return false;
}
else
{
// The slot is used, but isn't our item.
if (!table[bucket].Collision)
{
// Since the collision bit is off, we can't have a duplicate.
if (emptyBucket >= 0)
{
// We already have an empty bucket to use.
break;
}
else
{
// Keep searching for an empty bucket to place the item.
table[bucket].Collision = true;
duplicateMightExist = false;
}
}
}

bucket = (bucket + skip) & hashMask;
}

// We found an empty bucket. Insert the new item.
```

```
table[emptyBucket].HashValue = hash;
table[emptyBucket].item = item;

++count;
if (!table[emptyBucket].Collision)
++usedSlots;
previous = default(T);
return true;
}

///

/// Deletes an item from the hash table.
///
/// Item to search for and delete. /// If true returned, the actual item stored in the hash table (must be
/// equal to , but may not be identical. /// True if item was found and deleted, false if item wasn't found.
public bool Delete(T item, out T itemDeleted)
{
int hash, bucket, skip;

StopEnumerations();

if (count == 0)
{
itemDeleted = default(T);
return false;
}

hash = GetHashValues(item, out bucket, out skip);

for (; ; )
{
if (table[bucket].HashValue == hash && equalityComparer.Equals(table[bucket].item, item))
{
// Found the item. Remove it.
itemDeleted = table[bucket].item;
```

```
table[bucket].Clear();
–count;
if (!table[bucket].Collision)
–usedSlots;
ShrinkIfNeeded();
return true;
}
else if (!table[bucket].Collision)
{
// No collision bit, so we can stop searching. No such element.
itemDeleted = default(T);
return false;
}

bucket = (bucket + skip) & hashMask;
}
}

///

/// Find an item in the hash table. If found, optionally replace it with the
/// finding item.
///
/// Item to find. /// If true, replaces the equal item in the hash table
/// with . /// Returns the equal item found in the table, if true was returned. /// True if the item was found, false otherwise.
public bool Find(T find, bool replace, out T item)
{
int hash, bucket, skip;

if (count == 0)
{
item = default(T);
return false;
```

```
}

hash = GetHashValues(find, out bucket, out skip);

for (; ; )
{
if     (table[bucket].HashValue     ==     hash     &&
equalityComparer.Equals(table[bucket].item, find))
{
// Found the item.
item = table[bucket].item;
if (replace)
table[bucket].item = find;
return true;
}
else if (!table[bucket].Collision)
{
// No collision bit, so we can stop searching. No such
element.
item = default(T);
return false;
}

bucket = (bucket + skip) & hashMask;
}
}

///

/// Enumerate all of the items in the hash table. The items
/// are enumerated in a haphazard, unpredictable order.
///
/// An IEnumerator&lt;T&gt; that enumerates the items
/// in the hash table.
public IEnumerator GetEnumerator()
{
if (count > 0)
{
int startStamp = changeStamp;
```

```
foreach (Slot slot in table)
{
if (!slot.Empty)
{
yield return slot.item;
CheckEnumerationStamp(startStamp);
}
}
}
}

///

/// Enumerate all of the items in the hash table. The items
/// are enumerated in a haphazard, unpredictable order.
///
/// An IEnumerator that enumerates the items
/// in the hash table.
System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
{
return GetEnumerator();
}

///

/// Creates a clone of this hash table.
///
/// If non-null, this function is applied to each item when
cloning. It must be the
/// case that this function does not modify the hash code or
equality function. /// A shallow clone that contains the same
items.
public Hash Clone(Converter cloneItem)
{
Hash clone = new Hash(equalityComparer);
clone.count = this.count;
clone.usedSlots = this.usedSlots;
clone.totalSlots = this.totalSlots;
```

```
clone.loadFactor = this.loadFactor;
clone.thresholdGrow = this.thresholdGrow;
clone.thresholdShrink = this.thresholdShrink;
clone.hashMask = this.hashMask;
clone.secondaryShift = this.secondaryShift;
if (table != null)
{
clone.table = (Slot[])table.Clone();

if (cloneItem != null)
{
for (int i = 0; i < table.Length; ++i) { if (!table[i].Empty)
table[i].item = cloneItem(table[i].item); } } } return clone;
} #region Serialization /* ///
/// Serialize the hash table. Called from the serialization
infrastructure.
///

void  ISerializable.GetObjectData(SerializationInfo  info,
StreamingContext context)
{
if (info == null)
throw new ArgumentNullException(“info”);

info.AddValue(“equalityComparer”,     equalityComparer,
typeof(IEqualityComparer));
info.AddValue(“loadFactor”, loadFactor, typeof(float));
T[] items = new T[count];
int i = 0;
foreach (Slot slot in table)
if (! slot.Empty)
items[i++] = slot.item;
info.AddValue(“items”, items, typeof(T[]));
}

///

/// Called on deserialization. We cannot deserialize now,
```

```
because hash codes
/// might not be correct now. We do real deserialization in
the OnDeserialization call.
///
protected Hash(SerializationInfo serInfo, StreamingContext
context)
{
// Save away the serialization info for use later. We can't be
sure of hash codes
// being stable until the entire object graph is deserialized,
so we wait until then
// to deserialize.
serializationInfo = serInfo;
}

///

/// Deserialize the hash table. Called from the serialization
infrastructure when
/// the object graph has finished deserializing.
///
void IDeserializationCallback.OnDeserialization(object sender)
{
if (serializationInfo == null)
return;

loadFactor = serializationInfo.GetSingle(“loadFactor”);
equalityComparer = (IEqualityComparer)
serializationInfo.GetValue(“equalityComparer”,
typeof(IEqualityComparer));

T[] items = (T[])serializationInfo.GetValue(“items”,
typeof(T[]));
T dummy;

EnsureEnoughSlots(items.Length);
foreach (T item in items)
Insert(item, true, out dummy);

serializationInfo = null;
```

```
}
*/
#endregion Serialization

#if DEBUG
///

/// Print out basic stats about the hash table.
///
internal void PrintStats()
{
Console.WriteLine(“count={0} usedSlots={1} totalSlots={2}”, count, usedSlots, totalSlots);
Console.WriteLine(“loadFactor={0} thresholdGrow={1} thresholdShrink={2}”, loadFactor, thresholdGrow, thresholdShrink);
Console.WriteLine(“hashMask={0:X} secondaryShift={1}”, hashMask, secondaryShift);
Console.WriteLine();
}

///

/// Print out the state of the hash table and each of the slots. Each slot looks like:
/// Slot 4: C 4513e41e hello
/// where the “C” indicates the collision bit is on
/// the next hex number is the hash value
/// followed by ToString() on the item.
///
internal void Print()
{
PrintStats();
for (int i = 0; i < totalSlots; ++i) Console.WriteLine("Slot {0,4:X}: {1} {2,8:X} {3}", i, table[i].Collision ? "C" : " ", table[i].HashValue, table[i].Empty ? "” : table[i].item.ToString());
Console.WriteLine();
}
```

```
///

/// Check that everything appears to be OK in the hash table.
///
internal void Validate()
{
Debug.Assert(count <= usedSlots); Debug.Assert(count <=
totalSlots); Debug.Assert(usedSlots <= totalSlots);
Debug.Assert(usedSlots <= thresholdGrow);
Debug.Assert((int)(totalSlots * loadFactor) == thresholdGrow);
if (thresholdShrink > 1)
Debug.Assert(thresholdGrow / 3 == thresholdShrink);
else
Debug.Assert(thresholdGrow / 3 <= MINSIZE); if (totalSlots >
0)
{
Debug.Assert((totalSlots & (totalSlots – 1)) == 0); //
totalSlots is a power of two.
Debug.Assert(totalSlots – 1 == hashMask);
Debug.Assert(GetSecondaryShift(totalSlots) == secondaryShift);
Debug.Assert(totalSlots == table.Length);
}

// Traverse the table. Make sure that count and usedSlots are
right, and that
// each slot looks reasonable.
int expectedCount = 0, expectedUsed = 0, initialBucket, skip;
if (table != null)
{
for (int i = 0; i < totalSlots; ++i) { Slot slot = table[i];
if (slot.Empty) { // Empty slot if (slot.Collision)
++expectedUsed; Debug.Assert(object.Equals(default(T),
slot.item)); } else { // not empty. ++expectedCount;
++expectedUsed; Debug.Assert(slot.HashValue != 0);
Debug.Assert(GetHashValues(slot.item, out initialBucket, out
skip) == slot.HashValue); if (initialBucket != i)
Debug.Assert(table[initialBucket].Collision); } } }
```

```
Debug.Assert(expectedCount == count);
Debug.Assert(expectedUsed == usedSlots); } #endif //DEBUG } }
```
[/csharp]