hashtable.cm

```
/*
    ================================================================

    Copyright (c) 2012-2016 Seppo Laakko
    http://sourceforge.net/projects/cmajor/

    Distributed under the GNU General Public License, version 3 (GPLv3).
    (See accompanying LICENSE.txt or http://www.gnu.org/licenses/gpl.html
        )

    ================================================================

    */

// Copyright (c) 1994
// Hewlett-Packard Company
// Copyright (c) 1996
// Silicon Graphics Computer Systems, Inc.
// Copyright (c) 2009 Alexander Stepanov and Paul McJones

using System;
using System.Concepts;

namespace System.Collections
{
    public static class HashtablePrimes
    {
        static nothrow HashtablePrimes()
        {
            try
            {
                primes.Reserve(26);
                primes.Add(53);
                primes.Add(97);
                primes.Add(193);
                primes.Add(389);
                primes.Add(769);
                primes.Add(1543);
                primes.Add(3079);
                primes.Add(6151);
                primes.Add(12289);
                primes.Add(24593);
                primes.Add(49157);
                primes.Add(98317);
                primes.Add(196613);
                primes.Add(393241);
                primes.Add(786433);
                primes.Add(1572869);
                primes.Add(3145739);
                primes.Add(6291469);
                primes.Add(12582917);
                primes.Add(25165843);
```

```
            primes.Add(50331653);
            primes.Add(100663319);
            primes.Add(201326611);
            primes.Add(402653189);
            primes.Add(805306457);
            primes.Add(1610612741);
        }
        catch (const Exception& ex)
        {
            try
            {
                Console.Error() << "hashtable primes initialization
                    failed" << endl();
            }
            catch (const Exception&)
            {
            }
            exit(1);
        }
    }
    public static nothrow int GetNextPrime(int n)
    {
        List<int>.ConstIterator p = LowerBound(primes.CBegin(),
            primes.CEnd(), n);
        if (p < primes.CEnd())
        {
            return *p;
        }
        return primes.Back();
    }
    private static List<int> primes;
}

public nothrow inline ulong GetHashCode(long x)
{
    return cast<ulong>(x);
}

public nothrow inline ulong GetHashCode(ulong x)
{
    return x;
}

public nothrow inline ulong GetHashCode(char x)
{
    return cast<ulong>(x);
}

public nothrow inline ulong GetHashCode(wchar x)
{
    return cast<ulong>(x);
}
```

```
public nothrow inline ulong GetHashCode(uchar x)
{
    return cast<ulong>(x);
}

public nothrow inline ulong GetHashCode(void* x)
{
    return cast<ulong>(x);
}

public nothrow inline ulong GetHashCode(const string& s)
{
    ulong hashCode = 14695981039346656037u;
    for (char c : s)
    {
        hashCode = hashCode ^ cast<ulong>(c);
        hashCode = hashCode * 1099511628211u;
    }
    return hashCode;
}

public class Hasher<T>: UnaryFun<T, ulong>
{
    public nothrow inline ulong operator()(const T& x)
    {
        return GetHashCode(x);
    }
}

public class Bucket<T> where T is Semiregular
{
    public typedef T ValueType;

    public Bucket(const ValueType& value_, Bucket<ValueType>* next_):
        value(value_), next(next_)
    {
    }
    public nothrow inline const ValueType& Value() const
    {
        return value;
    }
    public nothrow inline ValueType& Value()
    {
        return value;
    }
    public nothrow inline Bucket<ValueType>* Next() const
    {
        return next;
    }
    public nothrow void SetNext(Bucket<ValueType>* next_)
    {
        next = next_;
    }
```

```
        private ValueType value;
        private Bucket<ValueType>* next;
    }

    public abstract class HashtableBase<T> where T is Semiregular
    {
        public typedef T ValueType;

        public default nothrow HashtableBase();
        public default nothrow HashtableBase(const HashtableBase<T>&);
        public default nothrow HashtableBase(HashtableBase<T>&&);
        public default nothrow void operator=(const HashtableBase<T>&);
        public default nothrow void operator=(HashtableBase<T>&&);
        public virtual nothrow ~HashtableBase()
        {
        }
        public abstract nothrow int GetBucketCount() const;
        public abstract nothrow int GetBucketIndex(const ValueType& value
            ) const;
        public abstract nothrow Bucket<ValueType>* GetBucket(int index)
            const;
    }

    public class HashtableIterator<T, R, P>
    {
        public typedef T ValueType;
        public typedef R ReferenceType;
        public typedef P PointerType;
        public typedef HashtableIterator<ValueType, ReferenceType,
            PointerType> Self;

        public nothrow HashtableIterator(): table(null), bucket(null)
        {
        }
        public nothrow HashtableIterator(HashtableBase<ValueType>* table_
            , Bucket<ValueType>* bucket_): table(table_), bucket(bucket_)
        {
        }
        public nothrow ReferenceType operator*()
        {
            #assert(bucket != null);
            return bucket->Value();
        }
        public nothrow PointerType operator->() const
        {
            #assert(bucket != null);
            return &(bucket->Value());
        }
        public nothrow Self& operator++()
        {
            #assert(bucket != null && table != null);
            Bucket<ValueType>* old = bucket;
            bucket = bucket->Next();
```

```
            if (bucket == null)
            {
                int index = table->GetBucketIndex(old->Value());
                #assert(index != -1);
                ++index;
                int n = table->GetBucketCount();
                while (bucket == null && index < n)
                {
                    bucket = table->GetBucket(index);
                    ++index;
                }
            }
            return *this;
        }
        public nothrow inline Bucket<ValueType>* GetBucket() const
        {
            return bucket;
        }
        private HashtableBase<ValueType>* table;
        private Bucket<ValueType>* bucket;
    }

    public nothrow bool operator==<T, R, P>(const HashtableIterator<T, R,
        P>& left, const HashtableIterator<T, R, P>& right)
    {
        return left.GetBucket() == right.GetBucket();
    }

    public class Hashtable<KeyType, ValueType, KeyOfValue, HashFun =
        Hasher<KeyType>, Compare = EqualTo<KeyType>> : HashtableBase<
        ValueType>
        where KeyType is Semiregular and ValueType is Semiregular and
            KeySelectionFunction<KeyOfValue, KeyType, ValueType> and
            HashFunction<HashFun, KeyType> and Compare is Relation and
            Compare.Domain is KeyType
    {
        public typedef Hashtable<KeyType, ValueType, KeyOfValue, HashFun,
            Compare> Self;
        public typedef HashtableIterator<ValueType, ValueType&, ValueType
            *> Iterator;
        public typedef HashtableIterator<ValueType, const ValueType&,
            const ValueType*> ConstIterator;

        public nothrow Hashtable(): base(), buckets(), count(0),
            loadFactor(0.0), maxLoadFactor(0.8), keyOf(), hash(), equal()
        {
        }
        public Hashtable(const Self& that)
        {
            CopyFrom(that);
        }
        public default nothrow Hashtable(Self&&);
        public void operator=(const Self& that)
```

```
        {
            Clear();
            CopyFrom(that);
        }
        public default nothrow void operator=(Self&&);
        public nothrow override ~Hashtable()
        {
            Clear();
        }
        public nothrow inline int Count() const
        {
            return count;
        }
        public nothrow inline bool IsEmpty() const
        {
            return count == 0;
        }
        public nothrow void Clear()
        {
            int n = buckets.Count();
            for (int i = 0; i < n; ++i)
            {
                Bucket<ValueType>* bucket = buckets[i];
                while (bucket != null)
                {
                    Bucket<ValueType>* next = bucket->Next();
                    delete bucket;
                    bucket = next;
                }
                buckets[i] = null;
            }
            count = 0;
        }
        public nothrow Iterator Begin()
        {
            return Iterator(this, GetFirstBucket());
        }
        public nothrow ConstIterator Begin() const
        {
            return ConstIterator(this, GetFirstBucket());
        }
        public nothrow ConstIterator CBegin() const
        {
            return ConstIterator(this, GetFirstBucket());
        }
        public nothrow Iterator End()
        {
            return Iterator(this, null);
        }
        public nothrow ConstIterator End() const
        {
            return ConstIterator(this, null);
        }
```

```
public nothrow ConstIterator CEnd() const
{
    return ConstIterator(this, null);
}
public nothrow void SetMaxLoadFactor(double maxLoadFactor_)
{
    maxLoadFactor = maxLoadFactor_;
}
public Pair<Iterator, bool> Insert(const ValueType& value)
{
    if (buckets.Count() == 0)
    {
        buckets.Resize(HashtablePrimes.GetNextPrime(0));
    }
    const KeyType& key = KeyOf(value);
    int index = Hash(key);
    #assert(index != -1);
    Bucket<ValueType>* bucket = buckets[index];
    while (bucket != null)
    {
        if (Equal(KeyOf(bucket->Value()), key))
        {
            return MakePair(Iterator(this, bucket), false);
        }
        bucket = bucket->Next();
    }
    bucket = new Bucket<ValueType>(value, buckets[index]);
    buckets[index] = bucket;
    ++count;
    SetLoadFactor();
    CheckForRehash();
    return MakePair(Iterator(this, bucket), true);
}
public nothrow void Remove(const KeyType& key)
{
    int index = Hash(key);
    if (index == -1) return;
    Bucket<ValueType>* bucket = buckets[index];
    Bucket<ValueType>* prev = null;
    while (bucket != null)
    {
        if (Equal(KeyOf(bucket->Value()), key))
        {
            if (prev != null)
            {
                prev->SetNext(bucket->Next());
            }
            else
            {
                buckets[index] = bucket->Next();
            }
            delete bucket;
            --count;
```

```
                SetLoadFactor();
                return;
            }
            prev = bucket;
            bucket = bucket->Next();
        }
    }
    public nothrow void Remove(Iterator pos)
    {
        Bucket<ValueType>* bucket = pos.GetBucket();
        if (bucket != null)
        {
            int index = Hash(KeyOf(bucket->Value()));
            if (index == -1) return;
            Bucket<ValueType>* b = buckets[index];
            Bucket<ValueType>* prev = null;
            while (b != bucket && b != null)
            {
                prev = b;
                b = b->Next();
            }
            if (b == bucket)
            {
                if (prev != null)
                {
                    prev->SetNext(b->Next());
                }
                else
                {
                    buckets[index] = b->Next();
                }
                delete bucket;
                --count;
                SetLoadFactor();
            }
        }
    }
    public nothrow Iterator Find(const KeyType& key)
    {
        int index = Hash(key);
        if (index >= 0)
        {
            Bucket<ValueType>* bucket = buckets[index];
            while (bucket != null)
            {
                if (Equal(KeyOf(bucket->Value()), key))
                {
                    return Iterator(this, bucket);
                }
                bucket = bucket->Next();
            }
        }
        return Iterator(this, null);
```

```
        }
        public nothrow ConstIterator Find(const KeyType& key) const
        {
            int index = Hash(key);
            if (index >= 0)
            {
                Bucket<ValueType>* bucket = buckets[index];
                while (bucket != null)
                {
                    if (Equal(KeyOf(bucket->Value()), key))
                    {
                        return ConstIterator(this, bucket);
                    }
                    bucket = bucket->Next();
                }
            }
            return ConstIterator(this, null);
        }
        public nothrow ConstIterator CFind(const KeyType& key) const
        {
            int index = Hash(key);
            if (index >= 0)
            {
                Bucket<ValueType>* bucket = buckets[index];
                while (bucket != null)
                {
                    if (Equal(KeyOf(bucket->Value()), key))
                    {
                        return ConstIterator(this, bucket);
                    }
                    bucket = bucket->Next();
                }
            }
            return ConstIterator(this, null);
        }
        public override nothrow int GetBucketCount() const
        {
            return buckets.Count();
        }
        public override nothrow int GetBucketIndex(const ValueType& value
            ) const
        {
            return Hash(KeyOf(value));
        }
        public override nothrow Bucket<ValueType>* GetBucket(int index)
            const
        {
            return buckets[index];
        }
        private nothrow inline void SetLoadFactor()
        {
            int bc = buckets.Count();
            if (bc == 0)
```

```
        {
            loadFactor = 1.0;
        }
        else
        {
            double c = count;
            loadFactor = c / bc;
        }
    }
    private void CheckForRehash()
    {
        if (loadFactor > maxLoadFactor)
        {
            Rehash();
        }
    }
    private void Rehash()
    {
        List<Bucket<ValueType>*> b;
        Swap(buckets, b);
        int n = b.Count();
        buckets.Resize(HashtablePrimes.GetNextPrime(n + 1));
        for (int i = 0; i < n; ++i)
        {
            Bucket<ValueType>* bucket = b[i];
            while (bucket != null)
            {
                const KeyType& key = KeyOf(bucket->Value());
                int index = Hash(key);
                #assert(index != -1);
                Bucket<ValueType>* next = bucket->Next();
                bucket->SetNext(buckets[index]);
                buckets[index] = bucket;
                bucket = next;
            }
        }
    }
    private nothrow Bucket<ValueType>* GetFirstBucket() const
    {
        Bucket<ValueType>* bucket = null;
        int n = buckets.Count();
        for (int i = 0; i < n; ++i)
        {
            bucket = buckets[i];
            if (bucket != null)
            {
                break;
            }
        }
        return bucket;
    }
    private void CopyFrom(const Self& that)
    {
```

```
            int n = that.buckets.Count();
            buckets.Resize(n);
            for (int i = 0; i < n; ++i)
            {
                Bucket<ValueType>* bucket = that.buckets[i];
                while (bucket != null)
                {
                    buckets[i] = new Bucket<ValueType>(bucket->Value(),
                        buckets[i]);
                    bucket = bucket->Next();
                }
            }
            count = that.count;
            loadFactor = that.loadFactor;
            maxLoadFactor = that.maxLoadFactor;
        }
        private nothrow inline const KeyType& KeyOf(const ValueType&
            value) const
        {
            return keyOf(value);
        }
        private nothrow inline int Hash(const KeyType& key) const
        {
            if (buckets.IsEmpty()) return -1;
            return cast<int>(hash(key) % cast<ulong>(buckets.Count()));
        }
        private nothrow inline bool Equal(const KeyType& left, const
            KeyType& right) const
        {
            return equal(left, right);
        }
        private List<Bucket<ValueType>*> buckets;
        private int count;
        private double loadFactor;
        private double maxLoadFactor;
        private KeyOfValue keyOf;
        private HashFun hash;
        private Compare equal;
    }
}
```