fwdlist.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 class ForwardListNode<T> where T is Semiregular
    {
        private typedef T ValueType;
        private typedef ForwardListNode<ValueType> Self;
        private typedef Self* NodePtr;

        public ForwardListNode(NodePtr next_, const ValueType& value_):
            next(next_), value(value_)
        {
        }
        public inline nothrow NodePtr Next() const
        {
            return next;
        }
        public inline nothrow void SetNext(NodePtr next_)
        {
            next = next_;
        }
        public inline nothrow const ValueType& Value() const
        {
            return value;
        }
        public inline nothrow ValueType& Value()
        {
            return value;
        }
        private ValueType value;
```

```
        private NodePtr next;
    }

    public class ForwardListNodeIterator<T, R, P>
    {
        public typedef T ValueType;
        public typedef R ReferenceType;
        public typedef P PointerType;
        private typedef ForwardListNodeIterator<ValueType, ReferenceType,
            PointerType> Self;
        private typedef ForwardListNode<ValueType> NodeType;
        private typedef NodeType* NodePtr;

        public nothrow ForwardListNodeIterator(): node(null)
        {
        }
        public nothrow ForwardListNodeIterator(NodePtr node_): node(node_
            )
        {
        }
        public nothrow ReferenceType operator*() const
        {
            #assert(node != null);
            return node->Value();
        }
        public nothrow PointerType operator->() const
        {
            #assert(node != null);
            return &(node->Value());
        }
        public inline nothrow Self& operator++()
        {
            #assert(node != null);
            node = node->Next();
            return *this;
        }
        public inline nothrow NodePtr GetNode() const
        {
            return node;
        }
        private NodePtr node;
    }

    public nothrow bool operator==<T, R, P>(ForwardListNodeIterator<T, R,
        P> left, ForwardListNodeIterator<T, R, P> right)
    {
        return left.GetNode() == right.GetNode();
    }

    public class ForwardList<T> where T is Regular
    {
        public typedef T ValueType;
        private typedef ForwardList<ValueType> Self;
```

```
public typedef ForwardListNodeIterator<ValueType, const ValueType
    &, const ValueType*> ConstIterator;
public typedef ForwardListNodeIterator<ValueType, ValueType&,
    ValueType*> Iterator;
private typedef ForwardListNode<ValueType> NodeType;
private typedef NodeType* NodePtr;

public nothrow ForwardList(): head(null)
{
}
public nothrow ~ForwardList()
{
    Clear();
}
public ForwardList(const Self& that): head(null)
{
    CopyFrom(that);
}
public nothrow ForwardList(Self&& that): head(that.head)
{
    that.head = null;
}
public void operator=(const Self& that)
{
    Clear();
    CopyFrom(that);
}
public nothrow void operator=(Self&& that)
{
    Clear();
    Swap(head, that.head);
}
public nothrow inline bool IsEmpty() const
{
    return head == null;
}
public nothrow int Count() const
{
    int count = 0;
    NodePtr node = head;
    while (node != null)
    {
        node = node->Next();
        ++count;
    }
    return count;
}
public nothrow void Clear()
{
    while (head != null)
    {
        NodePtr toRemove = head;
        head = head->Next();
```

```
            delete toRemove;
        }
    }
    public nothrow Iterator Begin()
    {
        return Iterator(head);
    }
    public nothrow Iterator End()
    {
        return Iterator(null);
    }
    public nothrow ConstIterator Begin() const
    {
        return ConstIterator(head);
    }
    public nothrow ConstIterator End() const
    {
        return ConstIterator(null);
    }
    public nothrow ConstIterator CBegin() const
    {
        return ConstIterator(head);
    }
    public nothrow ConstIterator CEnd() const
    {
        return ConstIterator(null);
    }
    public const ValueType& Front() const
    {
        #assert(head != null);
        return head->Value();
    }
    public Iterator InsertFront(const ValueType& value)
    {
        head = new NodeType(head, value);
        return Iterator(head);
    }
    public Iterator InsertAfter(Iterator pos, const ValueType& value)
    {
        NodePtr node = pos.GetNode();
        if (node == null)
        {
            return InsertFront(value);
        }
        node->SetNext(new NodeType(node->Next(), value));
        return Iterator(node->Next());
    }
    public void RemoveFront()
    {
        #assert(head != null);
        NodePtr node = head;
        head = head->Next();
        delete node;
```

```
    }
    public void RemoveAfter(Iterator pos)
    {
        NodePtr node = pos.GetNode();
        #assert(node != null);
        NodePtr toRemove = node->Next();
        #assert(toRemove != null);
        node->SetNext(toRemove->Next());
        delete toRemove;
    }
    public void Remove(const ValueType& value)
    {
        NodePtr prev = null;
        NodePtr node = head;
        while (node != null)
        {
            if (node->Value() == value)
            {
                if (node == head)
                {
                    head = head->Next();
                    delete node;
                    node = head;
                }
                else
                {
                    #assert(prev != null);
                    prev->SetNext(node->Next());
                    delete node;
                    node = prev->Next();
                }
            }
            else
            {
                prev = node;
                node = node->Next();
            }
        }
    }
    private void CopyFrom(const Self& that)
    {
        NodePtr n = that.head;
        NodePtr last = null;
        while (n != null)
        {
            if (head == null)
            {
                InsertFront(n->Value());
                last = head;
            }
            else
            {
                last->SetNext(new NodeType(null, n->Value()));
```

```
                    last = last->Next();
                }
                n = n->Next();
            }
        }
        private NodePtr head;
    }

    public bool operator==<T>(const ForwardList<T>& left, const
        ForwardList<T>& right) where T is Regular
    {
        return Equal(left.CBegin(), left.CEnd(), right.CBegin(), right.
            CEnd());
    }

    public bool operator<<T>(const ForwardList<T>& left, const
        ForwardList<T>& right) where T is TotallyOrdered
    {
        return LexicographicalCompare(left.CBegin(), left.CEnd(), right.
            CBegin(), right.CEnd());
    }
}
```