linkedlist.cm

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

    Copyright (c) 2012-2015 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 LinkedListNodeBase
    {
        public nothrow LinkedListNodeBase(LinkedListNodeBase* prev_,
            LinkedListNodeBase* next_): prev(prev_), next(next_)
        {
        }
        public virtual nothrow ~LinkedListNodeBase()
        {
        }
        public nothrow inline LinkedListNodeBase* Prev() const
        {
            return prev;
        }
        public nothrow void SetPrev(LinkedListNodeBase* prev_)
        {
            prev = prev_;
        }
        public nothrow inline LinkedListNodeBase* Next() const
        {
            return next;
        }
        public nothrow void SetNext(LinkedListNodeBase* next_)
        {
            next = next_;
        }
        private LinkedListNodeBase* prev;
        private LinkedListNodeBase* next;
```

```
    }

    public class LinkedListNode<T> : LinkedListNodeBase
    {
        public typedef T ValueType;

        public LinkedListNode(const ValueType& value_, LinkedListNodeBase
            * prev_, LinkedListNodeBase* next_): base(prev_, next_), value
            (value_)
        {
        }
        public nothrow const ValueType& Value() const
        {
            return value;
        }
        public nothrow ValueType& Value()
        {
            return value;
        }
        private ValueType value;
    }

    public abstract class LinkedListBase
    {
        public default nothrow LinkedListBase();
        public default nothrow LinkedListBase(const LinkedListBase& that)
            ;
        public default nothrow void operator=(const LinkedListBase& that)
            ;
        public default nothrow LinkedListBase(LinkedListBase&& that);
        public default nothrow void operator=(LinkedListBase&& that);
        public virtual ~LinkedListBase()
        {
        }
        public abstract nothrow LinkedListNodeBase* GetTail();
    }

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

        public nothrow LinkedListNodeIterator(): list(null), node(null)
        {
        }
        public nothrow LinkedListNodeIterator(LinkedListBase* list_,
            LinkedListNode<ValueType>* node_): list(list_), 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 nothrow Self& operator++()
        {
            #assert(node != null);
            node = cast<LinkedListNode<ValueType>*>(node->Next());
            return *this;
        }
        public nothrow Self& operator--()
        {
            if (node == null)
            {
                node = cast<LinkedListNode<ValueType>*>(list->GetTail());
            }
            else
            {
                node = cast<LinkedListNode<ValueType>*>(node->Prev());
            }
            return *this;
        }
        public nothrow inline LinkedListNode<ValueType>* GetNode() const
        {
            return node;
        }
        private LinkedListBase* list;
        private LinkedListNode<ValueType>* node;
    }

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

    public class LinkedList<T> : LinkedListBase where T is Regular
    {
        public typedef T ValueType;
        public typedef LinkedListNodeIterator<ValueType, ValueType&,
            ValueType*> Iterator;
        public typedef LinkedListNodeIterator<ValueType, const ValueType
            &, const ValueType*> ConstIterator;

        public nothrow LinkedList(): base(), head(null), tail(null),
            count(0)
        {
        }
```

```
public LinkedList(const LinkedList<ValueType>& that): base(),
    head(null), tail(null), count(0)
{
    CopyFrom(that);
}
public nothrow LinkedList(LinkedList<ValueType>&& that): base(),
    head(that.head), tail(that.tail), count(that.count)
{
    that.head = null;
    that.tail = null;
    that.count = 0;
}
public void operator=(const LinkedList<ValueType>& that)
{
    Clear();
    CopyFrom(that);
}
public default nothrow void operator=(LinkedList<ValueType>&&
    that);
public nothrow override ~LinkedList()
{
    Clear();
}
public nothrow Iterator Begin()
{
    return Iterator(this, head);
}
public nothrow ConstIterator Begin() const
{
    return ConstIterator(this, head);
}
public nothrow ConstIterator CBegin() const
{
    return ConstIterator(this, head);
}
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 inline int Count() const
{
    return count;
}
public nothrow inline bool IsEmpty() const
{
```

```
        return count == 0;
    }
    public nothrow void Clear()
    {
        LinkedListNode<ValueType>* n = head;
        while (n != null)
        {
            LinkedListNode<ValueType>* next = cast<LinkedListNode<
                ValueType>*>(n->Next());
            delete n;
            n = next;
        }
        head = null;
        tail = null;
        count = 0;
    }
    public Iterator InsertFront(const ValueType& value)
    {
        if (head == null)
        {
            head = new LinkedListNode<ValueType>(value, null, null);
            tail = head;
        }
        else
        {
            head = new LinkedListNode<ValueType>(value, null, head);
            head->Next()->SetPrev(head);
        }
        ++count;
        return Iterator(this, head);
    }
    public Iterator Insert(Iterator pos, const ValueType& value)
    {
        LinkedListNode<ValueType>* next = pos.GetNode();
        if (next != null)
        {
            LinkedListNode<ValueType>* prev = cast<LinkedListNode<
                ValueType>*>(next->Prev());
            LinkedListNode<ValueType>* n = new LinkedListNode<
                ValueType>(value, prev, next);
            next->SetPrev(n);
            if (prev != null)
            {
                prev->SetNext(n);
            }
            else
            {
                head = n;
            }
            ++count;
            return Iterator(this, n);
        }
        else
```

```
            {
                Add(value);
                return Iterator(this, tail);
            }
        }
        public void Add(const ValueType& value)
        {
            if (tail == null)
            {
                tail = new LinkedListNode<ValueType>(value, null, null);
                head = tail;
            }
            else
            {
                tail = new LinkedListNode<ValueType>(value, tail, null);
                tail->Prev()->SetNext(tail);
            }
            ++count;
        }
        public nothrow void RemoveFirst()
        {
            #assert(head != null);
            LinkedListNode<ValueType>* n = head;
            head = cast<LinkedListNode<ValueType>*>(head->Next());
            if (head != null)
            {
                head->SetPrev(null);
            }
            else
            {
                tail = null;
            }
            delete n;
            --count;
        }
        public nothrow void RemoveLast()
        {
            #assert(tail != null);
            LinkedListNode<ValueType>* n = tail;
            tail = cast<LinkedListNode<ValueType>*>(tail->Prev());
            if (tail != null)
            {
                tail->SetNext(null);
            }
            else
            {
                head = null;
            }
            delete n;
            --count;
        }
        public nothrow void Remove(Iterator pos)
        {
```

```
        LinkedListNode<ValueType>* n = pos.GetNode();
        #assert(n != null);
        LinkedListNode<ValueType>* prev = cast<LinkedListNode<
            ValueType>*>(n->Prev());
        LinkedListNode<ValueType>* next = cast<LinkedListNode<
            ValueType>*>(n->Next());
        if (prev != null)
        {
            prev->SetNext(next);
        }
        else
        {
            head = next;
        }
        if (next != null)
        {
            next->SetPrev(prev);
        }
        else
        {
            tail = prev;
        }
        delete n;
        --count;
    }
    public nothrow void Remove(const ValueType& value)
    {
        Iterator i = Begin();
        Iterator e = End();
        while (i != e)
        {
            if (*i == value)
            {
                Iterator r = i;
                ++i;
                Remove(r);
            }
            else
            {
                ++i;
            }
        }
    }
    public nothrow const ValueType& Front() const
    {
        #assert(head != null);
        return head->Value();
    }
    public nothrow const ValueType& Back() const
    {
        #assert(tail != null);
        return tail->Value();
    }
```

```
        public nothrow override LinkedListNode<ValueType>* GetTail()
        {
            return tail;
        }
        private void CopyFrom(const LinkedList<ValueType>& that)
        {
            ConstIterator e = that.CEnd();
            for (ConstIterator i = that.CBegin(); i != e; ++i)
            {
                Add(*i);
            }
        }
        private LinkedListNode<ValueType>* head;
        private LinkedListNode<ValueType>* tail;
        private int count;
    }

    public nothrow bool operator==<T>(const LinkedList<T>& left, const
        LinkedList<T>& right) where T is Regular
    {
        if (left.Count() != right.Count())
        {
            return false;
        }
        return Equal(left.CBegin(), left.CEnd(), right.CBegin(), right.
            CEnd());
    }

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