

rbtree.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.Support;  
using System.Concepts;  
  
namespace System.Collections  
{  
    public class RedBlackTreeNodeBase  
    {  
        public enum Color { red, black }  
        public typedef RedBlackTreeNodeBase Node;  
  
        public nothrow RedBlackTreeNodeBase(Node* parent_): color(Color.  
            black), parent(parent_), left(null), right(null)  
        {  
        }  
        public virtual nothrow ~RedBlackTreeNodeBase()  
        {  
            if (left != null && left != this)  
            {  
                delete left;  
            }  
            if (right != null && right != this)  
            {  
                delete right;  
            }  
        }  
        public nothrow inline Color GetColor() const  
        {  
            return color;  
        }  
        public nothrow inline void SetColor(Color color_)  
        {  

```

```

        color = color_;
    }
    public nothrow inline Node* Parent() const
    {
        return parent;
    }
    public nothrow inline Node*& ParentRef()
    {
        return parent;
    }
    public nothrow inline void SetParent(Node* parent_)
    {
        parent = parent_;
    }
    public nothrow inline Node* Left() const
    {
        return left;
    }
    public nothrow inline Node*& LeftRef()
    {
        return left;
    }
    public nothrow inline void SetLeft(Node* left_)
    {
        left = left_;
    }
    public nothrow inline Node* Right() const
    {
        return right;
    }
    public nothrow inline Node*& RightRef()
    {
        return right;
    }
    public nothrow inline void SetRight(Node* right_)
    {
        right = right_;
    }
    public nothrow inline bool IsHeaderNode() const
    {
        return color == Color.red && parent != null && parent->parent
            == this;
    }
    public static nothrow inline Node* Min(Node* n)
    {
        #assert(n != null);
        while (n->Left() != null)
        {
            n = n->Left();
        }
        return n;
    }
    public static nothrow inline Node* Max(Node* n)

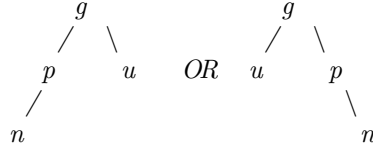
```

```

{
    #assert (n != null);
    while (n->Right() != null)
    {
        n = n->Right();
    }
    return n;
}
public static nothrow inline Node* Prev(Node* n)
{
    #assert (n != null);
    if (n->IsHeaderNode())
    {
        return n->Right(); // rightmost
    }
    else if (n->Left() != null)
    {
        return Max(n->Left());
    }
    else
    {
        Node* u = n->Parent();
        while (n == u->Left())
        {
            n = u;
            u = u->Parent();
        }
        return u;
    }
}
public static nothrow inline Node* Next(Node* n)
{
    #assert (n != null);
    if (n->Right() != null)
    {
        return Min(n->Right());
    }
    else
    {
        Node* u = n->Parent();
        while (n == u->Right())
        {
            n = u;
            u = u->Parent();
        }
        if (n->Right() != u)
        {
            return u;
        }
        return n;
    }
}
}
/*

```

REBALANCE-AFTER-INSERT:



```

*/
public static nothrow void RebalanceAfterInsert(Node* n, Node*&
root)
{
    #assert(n != null);
    n->SetColor(Color.red);
    while (n != root && n->Parent()->GetColor() == Color.red)
    {
        if (n->Parent() == n->Parent()->Parent()->Left())
        {
            Node* u = n->Parent()->Parent()->Right();
            if (u != null && u->GetColor() == Color.red)
            {
                n->Parent()->SetColor(Color.black);
                u->SetColor(Color.black);
                n->Parent()->Parent()->SetColor(Color.red);
                n = n->Parent()->Parent();
            }
            else
            {
                if (n == n->Parent()->Right())
                {
                    n = n->Parent();
                    RotateLeft(n, root);
                }
                n->Parent()->SetColor(Color.black);
                n->Parent()->Parent()->SetColor(Color.red);
                RotateRight(n->Parent()->Parent(), root);
            }
        }
        else
        {
            Node* u = n->Parent()->Parent()->Left();
            if (u != null && u->GetColor() == Color.red)
            {
                n->Parent()->SetColor(Color.black);
                u->SetColor(Color.black);
                n->Parent()->Parent()->SetColor(Color.red);
                n = n->Parent()->Parent();
            }
            else
            {
                if (n == n->Parent()->Left())
                {
                    n = n->Parent();
                }
            }
        }
    }
}

```

```

        RotateRight(n, root);
    }
    n->Parent()->SetColor(Color.black);
    n->Parent()->Parent()->SetColor(Color.red);
    RotateLeft(n->Parent()->Parent(), root);
}
}
root->SetColor(Color.black);
}
/*
REBALANCE-FOR-REMOVE:
*/
public static nothrow Node* RebalanceForRemove(Node* z, Node*&
root, Node*& leftmost, Node*& rightmost)
{
    #assert(z != null);
    Node* y = z;
    Node* x = null;
    Node* p = null;
    if (y->Left() == null) // z has at most one non-null child.
        y == z
    {
        x = y->Right(); // x might be null
    }
    else
    {
        if (y->Right() == null) // z has exactly one non-null
            child. y == z
        {
            x = y->Left(); // x is not null
        }
        else // z has two non-null children.
            set y to z's successor. x might be null.
        {
            y = y->Right();
            while (y->Left() != null)
            {
                y = y->Left();
            }
            x = y->Right();
        }
    }
    if (y != z) // relink y in place of z. y is z's successor
    {
        z->Left()->SetParent(y);
        y->SetLeft(z->Left());
        if (y != z->Right())
        {
            p = y->Parent();
            if (x != null)
            {
                x->SetParent(y->Parent());
            }
        }
    }
}

```

```

    }
    y->Parent()->SetLeft(x);    // y must be child of
        left
    y->SetRight(z->Right());
    z->Right()->SetParent(y);
}
else
{
    p = y;
}
if (root == z)
{
    root = y;
}
else if (z->Parent()->Left() == z)
{
    z->Parent()->SetLeft(y);
}
else
{
    z->Parent()->SetRight(y);
}
y->SetParent(z->Parent());
Color c = y->GetColor();
y->SetColor(z->GetColor());
z->SetColor(c);
y = z;
// y now points to node to be actually deleted
}
else    // y == z
{
    p = y->Parent();
    if (x != null)
    {
        x->SetParent(y->Parent());
    }
    if (root == z)
    {
        root = x;
    }
    else
    {
        if (z->Parent()->Left() == z)
        {
            z->Parent()->SetLeft(x);
        }
        else
        {
            z->Parent()->SetRight(x);
        }
    }
}
if (leftmost == z)
{

```

```

        if (z->Right() == null) // z->Left() must be null
            also
        {
            leftmost = z->Parent();
        }
        else
        {
            leftmost = Min(x);
        }
    }
    if (rightmost == z)
    {
        if (z->Left() == null)
        {
            rightmost = z->Parent();
        }
        else
        {
            rightmost = Max(x);
        }
    }
}
if (y->GetColor() != Color.red)
{
    while (x != root && (x == null || x->GetColor() == Color.
        black))
    {
        if (x == p->Left())
        {
            Node* w = p->Right();
            if (w->GetColor() == Color.red)
            {
                w->SetColor(Color.black);
                p->SetColor(Color.red);
                RotateLeft(p, root);
                w = p->Right();
            }
            if ((w->Left() == null || w->Left()->GetColor()
                == Color.black) &&
                (w->Right() == null || w->Right()->GetColor()
                == Color.black))
            {
                w->SetColor(Color.red);
                x = p;
                p = p->Parent();
            }
            else
            {
                if (w->Right() == null || w->Right()->
                    GetColor() == Color.black)
                {
                    if (w->Left() != null)
                    {

```

```

        w->Left ()->SetColor (Color . black);
    }
    w->SetColor (Color . red);
    RotateRight (w, root);
    w = p->Right ();
}
w->SetColor (p->GetColor ());
p->SetColor (Color . black);
if (w->Right () != null)
{
    w->Right ()->SetColor (Color . black);
}
RotateLeft (p, root);
break;
}
}
else    // same as above, with right <-> left
{
    Node* w = p->Left ();
    if (w->GetColor () == Color . red)
    {
        w->SetColor (Color . black);
        p->SetColor (Color . red);
        RotateRight (p, root);
        w = p->Left ();
    }
    if ((w->Right () == null || w->Right ()->GetColor ()
        == Color . black) &&
        (w->Left () == null || w->Left ()->GetColor ()
        == Color . black))
    {
        w->SetColor (Color . red);
        x = p;
        p = p->Parent ();
    }
    else
    {
        if (w->Left () == null || w->Left ()->GetColor
            () == Color . black)
        {
            if (w->Right () != null)
            {
                w->Right ()->SetColor (Color . black);
            }
            w->SetColor (Color . red);
            RotateLeft (w, root);
            w = p->Left ();
        }
        w->SetColor (p->GetColor ());
        p->SetColor (Color . black);
        if (w->Left () != null)
        {
            w->Left ()->SetColor (Color . black);

```



```

    }
    RotateRight(p, root);
    break;
    }
    }
}
if (x != null)
{
    x->SetColor(Color.black);
}
return y;
}
/* ROTATE LEFT:
    n
   / \
  a   u
     / \
    b   c
    =>
    u
   / \
  n   c
 / \
a   b
*/
private static nothrow void RotateLeft(Node* n, Node*& root)
{
    #assert(n != null);
    Node* u = n->Right();
    #assert(u != null);
    n->SetRight(u->Left());
    if (u->Left() != null)
    {
        u->Left()->SetParent(n);
    }
    u->SetParent(n->Parent());
    if (n == root)
    {
        root = u;
    }
    else if (n == n->Parent()->Left())
    {
        n->Parent()->SetLeft(u);
    }
    else
    {
        n->Parent()->SetRight(u);
    }
    u->SetLeft(n);
    n->SetParent(u);
}
/* ROTATE RIGHT:
    n
   / \
  u   c
 / \
a   b
    =>
    u
   / \
  a   n
     / \
    b   c
*/

```

```

private static nothrow void RotateRight(Node* n, Node*& root)
{
    #assert(n != null);
    Node* u = n->Left();
    #assert(u != null);
    n->SetLeft(u->Right());
    if (u->Right() != null)
    {
        u->Right()->SetParent(n);
    }
    u->SetParent(n->Parent());
    if (n == root)
    {
        root = u;
    }
    else if (n == n->Parent()->Right())
    {
        n->Parent()->SetRight(u);
    }
    else
    {
        n->Parent()->SetLeft(u);
    }
    u->SetRight(n);
    n->SetParent(u);
}

private Color color;
private Node* parent;
private Node* left;
private Node* right;
}

public class RedBlackTreeNode<T>: RedBlackTreeNodeBase
{
    public typedef T ValueType;

    public RedBlackTreeNode(const ValueType& value_, Node* parent_):
        base(parent_), value(value_) {}
    public nothrow const ValueType& Value() const { return value; }
    public nothrow ValueType& Value() { return value; }
    private ValueType value;
}

public class RedBlackTreeNodeIterator<T, R, P>
{
    public typedef T ValueType;
    public typedef R ReferenceType;
    public typedef P PointerType;
    private typedef RedBlackTreeNodeIterator<ValueType, ReferenceType,
        PointerType> Self;
    private typedef RedBlackTreeNode<ValueType> Node;

    public nothrow RedBlackTreeNodeIterator(): node(null)

```

```

{
}
public nothrow RedBlackTreeNodeIterator(Node* 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 nothrow Self& operator++()
{
    #assert(node != null);
    node = cast<Node*>(Node.Next(node));
    return *this;
}
public nothrow Self& operator--()
{
    #assert(node != null);
    node = cast<Node*>(Node.Prev(node));
    return *this;
}
public nothrow inline Node* GetNode() const
{
    return node;
}
private Node* node;
}

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

public class RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>
    where KeyType is Semiregular and ValueType is Semiregular and
        KeySelectionFunction<KeyOfValue, KeyType, ValueType> and
        Compare is Relation and Compare.Domain is KeyType
{
    private typedef RedBlackTree<KeyType, ValueType, KeyOfValue,
        Compare> Self;
    public typedef RedBlackTreeNodeIterator<ValueType, const
        ValueType&, const ValueType*> ConstIterator;
    public typedef RedBlackTreeNodeIterator<ValueType, ValueType&,
        ValueType*> Iterator;
    private typedef RedBlackTreeNode<ValueType> Node;

```

```

private typedef RedBlackTreeNodeBase NodeBase;

public RedBlackTree(): header(), count(0), keyOf(), comp()
{
    Init();
}
public RedBlackTree(const Self& that): header(), count(0), keyOf(
    ), comp() where ValueType is Copyable
{
    Init();
    CopyFrom(that);
}
public default nothrow RedBlackTree(Self&& that) where ValueType
    is Movable;
public void operator=(const Self& that) where ValueType is
    Copyable
{
    Clear();
    CopyFrom(that);
}
public default nothrow void operator=(Self&& that) where
    ValueType is Movable;
private void CopyFrom(const Self& that) where ValueType is
    Copyable
{
    if (that.Root() != null)
    {
        SetRoot(Copy(that.Root(), header.GetPtr()));
        SetLeftmost(cast<Node*>(Node.Min(Root())));
        SetRightmost(cast<Node*>(Node.Max(Root())));
        count = that.Count();
    }
}
public nothrow ~RedBlackTree()
{
    Clear();
}
public nothrow ConstIterator Begin() const
{
    return ConstIterator(Leftmost());
}
public nothrow Iterator Begin()
{
    return Iterator(Leftmost());
}
public nothrow ConstIterator CBegin() const
{
    return ConstIterator(Leftmost());
}
public nothrow ConstIterator End() const
{
    return ConstIterator(header.GetPtr());
}

```

```

public nothrow Iterator End()
{
    return Iterator(header.GetPtr());
}
public nothrow ConstIterator CEnd() const
{
    return ConstIterator(header.GetPtr());
}
public nothrow inline int Count() const
{
    #assert(count >= 0);
    return count;
}
public nothrow inline bool IsEmpty() const
{
    #assert(count >= 0);
    return count == 0;
}
public nothrow void Clear()
{
    if (header.IsNull())
    {
        return;
    }
    Node* root = Root();
    if (root != null)
    {
        delete root;
        SetRoot(null);
    }
    SetLeftmost(header.GetPtr());
    SetRightmost(header.GetPtr());
    count = 0;
}
public nothrow Iterator Find(const KeyType& key)
{
    Node* x = Root();
    while (x != null)
    {
        if (Comp(key, KeyOf(x->Value())))
        {
            x = cast<Node*>(x->Left());
        }
        else if (Comp(KeyOf(x->Value()), key))
        {
            x = cast<Node*>(x->Right());
        }
        else
        {
            return Iterator(x);
        }
    }
    return End();
}

```

```

}
public nothrow ConstIterator Find(const KeyType& key) const
{
    Node* x = Root();
    while (x != null)
    {
        if (Comp(key, KeyOf(x->Value())))
        {
            x = cast<Node*>(x->Left());
        }
        else if (Comp(KeyOf(x->Value()), key))
        {
            x = cast<Node*>(x->Right());
        }
        else
        {
            return ConstIterator(x);
        }
    }
    return CEnd();
}

public nothrow ConstIterator CFind(const KeyType& key) const
{
    Node* x = Root();
    while (x != null)
    {
        if (Comp(key, KeyOf(x->Value())))
        {
            x = cast<Node*>(x->Left());
        }
        else if (Comp(KeyOf(x->Value()), key))
        {
            x = cast<Node*>(x->Right());
        }
        else
        {
            return ConstIterator(x);
        }
    }
    return CEnd();
}

public Pair<Iterator, bool> Insert(const ValueType& value) where
    ValueType is Copyable
{
    Node* x = Root();
    Node* p = header.GetPtr();
    bool comp = true;
    while (x != null)
    {
        p = x;
        comp = Comp(KeyOf(value), KeyOf(x->Value()));
        if (comp)
        {

```

```

        x = cast<Node*>(x->Left());
    }
    else
    {
        x = cast<Node*>(x->Right());
    }
}
Iterator j = p;
if (comp)
{
    if (j == Begin())
    {
        return MakePair(Insert(x, p, value), true);
    }
    else
    {
        --j;
    }
}
if (Comp(KeyOf(j.GetNode()->Value()), KeyOf(value)))
{
    return MakePair(Insert(x, p, value), true);
}
return MakePair(j, false);
}
private Iterator Insert(Node* x, Node* p, const ValueType& value)
    where ValueType is Copyable
{
    Node* n = new Node(value, p);
    if (p == header.GetPtr() || x != null || Comp(KeyOf(value),
        KeyOf(p->Value())))
    {
        p->SetLeft(n);
        if (p == header.GetPtr())
        {
            SetRoot(n);
            SetRightmost(n);
        }
        else if (p == Leftmost())
        {
            SetLeftmost(n);
        }
    }
    else
    {
        p->SetRight(n);
        if (p == Rightmost())
        {
            SetRightmost(n);
        }
    }
    Node.RebalanceAfterInsert(n, RootRef());
    ++count;
}

```

```

        return Iterator(n);
    }
    public nothrow bool Remove(const KeyType& key)
    {
        Node* n = Root();
        while (n != null)
        {
            if (Comp(key, KeyOf(n->Value())))
            {
                n = cast<Node*>(n->Left());
            }
            else if (Comp(KeyOf(n->Value()), key))
            {
                n = cast<Node*>(n->Right());
            }
            else
            {
                break;
            }
        }
        if (n != null)
        {
            if (count == 1)
            {
                Clear();
            }
            else
            {
                Remove(Iterator(n));
            }
            return true;
        }
        return false;
    }
    public nothrow void Remove(Iterator pos)
    {
        Node* toRemove = cast<Node*>(Node.RebalanceForRemove(pos.
            GetNode(), RootRef(), LeftmostRef(), RightmostRef()));
        toRemove->SetLeft(null);
        toRemove->SetRight(null);
        delete toRemove;
        --count;
    }
    private nothrow inline const KeyType& KeyOf(const ValueType&
        value) const
    {
        return keyOf(value);
    }
    private nothrow inline bool Comp(const KeyType& left, const
        KeyType& right) const
    {
        return comp(left, right);
    }
}

```



```

private nothrow inline Node* Root()
{
    return cast<Node*>(header->Parent());
}
private nothrow inline NodeBase*& RootRef()
{
    return header->ParentRef();
}
private nothrow inline void SetRoot(Node* root)
{
    header->SetParent(root);
}
private nothrow inline Node* Leftmost()
{
    return cast<Node*>(header->Left());
}
private nothrow inline NodeBase*& LeftmostRef()
{
    return header->LeftRef();
}
private nothrow inline void SetLeftmost(Node* lm)
{
    header->SetLeft(lm);
}
private nothrow inline Node* Rightmost()
{
    return cast<Node*>(header->Right());
}
private nothrow inline NodeBase*& RightmostRef()
{
    return header->RightRef();
}
private nothrow inline void SetRightmost(Node* rm)
{
    header->SetRight(rm);
}
private void Init()
{
    header.Reset(new Node(ValueType(), null));
    header->SetColor(Node.Color.red);
    SetLeftmost(header.GetPtr());
    SetRightmost(header.GetPtr());
}
private Node* Copy(Node* x, Node* p)
{
    #assert(x != null && p != null);
    Node* top = CloneNode(x, p);
    if (x->Right() != null)
    {
        top->SetRight(Copy(cast<Node*>(x->Right()), top));
    }
    p = top;
    x = cast<Node*>(x->Left());
}

```

```

        while (x != null)
        {
            Node* y = CloneNode(x, p);
            p->SetLeft(y);
            if (x->Right() != null)
            {
                y->SetRight(Copy(cast<Node*>(x->Right()), y));
            }
            p = y;
            x = cast<Node*>(x->Left());
        }
        return top;
    }
    private Node* CloneNode(Node* x, Node* p) const
    {
        #assert(x != null && p != null);
        Node* clone = new Node(x->Value(), p);
        clone->SetColor(x->GetColor());
        return clone;
    }
    private UniquePtr<Node> header;
    private int count;
    private KeyOfValue keyOf;
    private Compare comp;
}
}

```