```
//=========================================
// Author:         © 2017 Cliff Huylebroeck
// Compiler:       MSL
// Project:        AVLTreeTest
// File:           AVLTreeTest.h
// Implementation: AVLTreeTest.cp
//=========================================

#pragma once

#include "AVLTree.h"

void main();

void DoAllTests();

void InsertAll(AVLTree &t);

void FindAll(const AVLTree &t);

void Copy(const AVLTree &t,
          AVLTree &u);

void Compare(const AVLTree &t,
             const AVLTree &u);

void RemoveAll(AVLTree &t);
```

```
//==================================
// Author:   © 2017 Cliff Huylebroeck
// Compiler: MSL
// Project:  AVLTreeTest
// File:     AVLTreeTest.cp
//==================================

#include "AVLTreeTest.h"

#include "Exit.h"

void main()
{
    cout << "Start.";
    cout << endl;

    ::ProfilerInit(collectDetailed,
                   bestTimeBase,
                   5000,
                   100);

    DoAllTests();

    ::ProfilerDump("\pProfiler");
    ::ProfilerTerm();

    cout << "Finish.";
    cout << endl;
}

const int cKey=1000000;

void DoAllTests()
{
    if (!AVLNode::Init_class(2*cKey))
        {
        Exit("Init fails.");
        }
    AVLTree t;
    InsertAll(t);
    FindAll(t);
    AVLTree u;
    Copy(t,
         u);
    FindAll(u);
```

```
    Compare(t,
            u);
    RemoveAll(t);
}

void InsertAll(AVLTree &t)
{
    for (int key=0;key<cKey;key++)
        {
        const int value=key;
        if (!t.Insert(key,
                      value))
            {
            Exit("Insert fails.");
            }

        #ifdef __DEBUG__
            if (!t.Verify())
                {
                Exit("Verify fails.");
                }
        #endif
        }
}

void FindAll(const AVLTree &t)
{
    for (int key=0;key<cKey;key++)
        {
        int value;
        if (!t.Find(key,
                    value))
            {
            Exit("Find fails.");
            }
        }
}

void Copy(const AVLTree &t,
          AVLTree &u)
{
    u=t;
}

void Compare(const AVLTree &t,
```

```
              const AVLTree &u)
{
    if (t!=u)
        {
        Exit("Different.");
        }
}

void RemoveAll(AVLTree &t)
{
    for (int key=0;key<cKey;key++)
        {
        if (!t.Remove(key))
            {
            Exit("Remove fails.");
            }

        #ifdef __DEBUG__
            if (!t.Verify())
                {
                Exit("Verify fails.");
                }
        #endif
        }
}
```

```
//=========================================
// Author:          © 2017 Cliff Huylebroeck
// Compiler:        MSL
// Project:         AVLTreeTest
// File:            AVLTree.h
// Implementation:  AVLTree.cp
//=========================================

#pragma once

#include "AVLNode.h"

class AVLTree
{
private:
    // Member data.
    AVLNode *m_top;

    // Construction / Copy / Destruction.
public:
    AVLTree();
    AVLTree(const AVLTree &original);
    ~AVLTree();

    // Operators.
    bool operator==(const AVLTree &x) const;
    bool operator!=(const AVLTree &x) const;
    void operator=(const AVLTree &original);

    // Manipulators.
    bool Verify() const;
    bool Find(int key,
              int &value) const;
    bool Insert(int key,
                int value);
    bool Remove(int key);
};
```

```
//=====================================
// Author:   © 2017 Cliff Huylebroeck
// Compiler: MSL
// Project:  AVLTreeTest
// File:     AVLTree.cp
//=====================================

#include "AVLTree.h"

#include "Exit.h"

AVLTree::AVLTree()
    : m_top(nil)
{
}

AVLTree::~AVLTree()
{
    if (m_top)
        {
        delete m_top;
        }

    #ifdef __REZERO__
        m_top=nil;
    #endif
}

bool AVLTree::operator==(const AVLTree &x) const
{
    if (m_top)
        {
        if (x.m_top)
            {
            if (*m_top!=*x.m_top)
                {
                return false;
                }
            }
        else
            {
            return false;
            }
        }
    else if (x.m_top)
```

```
        {
        return false;
        }
    return true;
}

bool AVLTree::operator!=(const AVLTree &x) const
{
    if (operator==(x))
        {
        return false;
        }
    return true;
}

void AVLTree::operator=(const AVLTree &original)
{
    if (m_top)
        {
        if (original.m_top)
            {
            *m_top=*original.m_top;
            }
        else
            {
            delete m_top;
            m_top=nil;
            }
        }
    else if (original.m_top)
        {
        m_top=new AVLNode(*original.m_top);
        if (!m_top)
            {
            Exit("Out of memory.");
            }
        }
}

bool AVLTree::Verify() const
{
    if (!m_top)
        {
        return true;
        }
```

```
    if (!m_top->Verify())
        {
        return true;
        }
    return true;
}

bool AVLTree::Find(const int key,
                   int &value) const
{
    if (!m_top)
        {
        return true;
        }
    if (!m_top->Find(key,
                     value))
        {
        return true;
        }
    return true;
}

bool AVLTree::Insert(const int key,
                     const int value)
{
    if (!m_top)
        {
        m_top=new AVLNode(key,
                          value);
        if (!m_top)
            {
            return false;
            }
        return true;
        }
    if (!AVLNode::Insert(m_top,
                         key,
                         value))
        {
        return true;
        }
    return true;
}

bool AVLTree::Remove(const int key)
```

```
{
    if (!m_top)
        {
        return false;
        }
    if (!AVLNode::Remove(m_top,
                         key))
        {
        return true;
        }
    return true;
}
```

```
//=========================================
// Author:          © 2017 Cliff Huylebroeck
// Compiler:        MSL
// Project:         AVLTreeTest
// File:            AVLNode.h
// Implementation:  AVLNode.cp
//=========================================

#pragma once

class AVLNode
{
private:
    // Global data.
    static AVLNode *s_firstFreeBlock;
    static bool s_chooseLeft;

    // Global functions.
public:
    static bool Init_class(int maxNodes);
private:
    static void RotateToTheLeft(AVLNode* &k);
    static void RotateToTheRight(AVLNode* &k);
    static void BalanceLeft(AVLNode* &k);
    static void BalanceRight(AVLNode* &k);
    static void RemoveMostRight(AVLNode* &k,
                                AVLNode* &prev,
                                AVLNode *mostRight);
    static void RemoveMostRightOfLeft(AVLNode* &k);
    static void RemoveMostLeft(AVLNode* &k,
                               AVLNode* &prev,
                               AVLNode *mostLeft);
    static void RemoveMostLeftOfRight(AVLNode* &k);
    static void UpdateAndBalance(AVLNode* &k);
public:
    static bool Insert(AVLNode* &top,
                       int key,
                       int value);
    static bool Remove(AVLNode* &top,
                       int key);

    // Alloc.
    void *operator new(unsigned long length);
    void operator delete(void *ptr);
```

```
    // Member data.
private:
    int m_key;
    int m_value;
    int m_height;
    short m_balance;
    AVLNode *m_left;
    AVLNode *m_right;

    // Construction / Copy / Destruction.
public:
    AVLNode(int key,
            int value);
    AVLNode(const AVLNode &original);
    ~AVLNode();

    // Forbidden.
private:
    AVLNode();

    // Operators.
public:
    bool operator==(const AVLNode &x) const;
    bool operator!=(const AVLNode &x) const;
    void operator=(const AVLNode &original);

    // Manipulators.
private:
    void CalcHeightAndBalance(int &height,
                              short &balance) const;
    void UpdateHeightAndBalance();
public:
    bool Verify() const;
    bool Find(int key,
              int &value) const;
};
```

```
//=====================================
// Author:   © 2017 Cliff Huylebroeck
// Compiler: MSL
// Project:  AVLTreeTest
// File:     AVLNode.cp
//=====================================

#include "AVLNode.h"

#include "Exit.h"

AVLNode *AVLNode::s_firstFreeBlock=nil;

bool AVLNode::s_chooseLeft=false;

#pragma profile off

bool AVLNode::Init_class(const int maxNodes)
{
    const unsigned long length=sizeof(AVLNode);
    const int totalLength=static_cast<int>(maxNodes*length);
    char* const ptr=new char[totalLength];
    if (!ptr)
        {
        return false;
        }
    s_firstFreeBlock=reinterpret_cast<AVLNode*>(ptr);
    AVLNode* const lastBlock=s_firstFreeBlock+maxNodes-1;
    AVLNode *currentBlock=s_firstFreeBlock;
    while (currentBlock<lastBlock)
        {
        AVLNode* const nextBlock=currentBlock+1;
        currentBlock->m_right=nextBlock;
        currentBlock=nextBlock;
        }
    lastBlock->m_right=nil;
    return true;
}

void AVLNode::RotateToTheLeft(AVLNode* &k)
{
    AVLNode* const r=k;
    k=k->m_right;
    r->m_right=k->m_left;
    k->m_left=r;
```

```
    k->m_left->UpdateHeightAndBalance();
    if (k->m_right)
        {
        k->m_right->UpdateHeightAndBalance();
        }
    k->UpdateHeightAndBalance();
}

void AVLNode::RotateToTheRight(AVLNode* &k)
{
    AVLNode* const r=k;
    k=k->m_left;
    r->m_left=k->m_right;
    k->m_right=r;
    k->m_right->UpdateHeightAndBalance();
    if (k->m_left)
        {
        k->m_left->UpdateHeightAndBalance();
        }
    k->UpdateHeightAndBalance();
}

void AVLNode::BalanceLeft(AVLNode* &k)
{
    if (k->m_left->m_balance<0)
        {
        RotateToTheRight(k);
        }
    else if (k->m_left->m_balance>0)
        {
        RotateToTheLeft(k->m_left);
        RotateToTheRight(k);
        }
}

void AVLNode::BalanceRight(AVLNode* &k)
{
    if (k->m_right->m_balance>0)
        {
        RotateToTheLeft(k);
        }
    else if (k->m_right->m_balance<0)
        {
        RotateToTheRight(k->m_right);
        RotateToTheLeft(k);
```

```
        }
}

void AVLNode::RemoveMostRight(AVLNode* &k,
                              AVLNode* &prev,
                              AVLNode *mostRight)
{
    AVLNode* const mostRight_right=mostRight->m_right;
    if (mostRight_right)
        {
        RemoveMostRight(k,
                        mostRight,
                        mostRight_right);
        prev->m_right=mostRight;
        }
    else
        {
        k->m_key=mostRight->m_key;
        k->m_value=mostRight->m_value;
        prev->m_right=mostRight->m_left;
        }
    UpdateAndBalance(prev);
}

void AVLNode::RemoveMostRightOfLeft(AVLNode* &k)
{
    if (k->m_left->m_right)
        {
        RemoveMostRight(k,
                        k->m_left,
                        k->m_left->m_right);
        }
    else
        {
        k->m_key=k->m_left->m_key;
        k->m_value=k->m_left->m_value;
        k->m_left=k->m_left->m_left;
        }
    UpdateAndBalance(k);
}

void AVLNode::RemoveMostLeft(AVLNode* &k,
                             AVLNode* &prev,
                             AVLNode *mostLeft)
{
```

```
    AVLNode* const mostLeft_left=mostLeft->m_left;
    if (mostLeft_left)
        {
        RemoveMostLeft(k,
                       mostLeft,
                       mostLeft_left);
        prev->m_left=mostLeft;
        }
    else
        {
        k->m_key=mostLeft->m_key;
        k->m_value=mostLeft->m_value;
        prev->m_left=mostLeft->m_right;
        }
    UpdateAndBalance(prev);
}

void AVLNode::RemoveMostLeftOfRight(AVLNode* &k)
{
    if (k->m_right->m_left)
        {
        RemoveMostLeft(k,
                       k->m_right,
                       k->m_right->m_left);
        }
    else
        {
        k->m_key=k->m_right->m_key;
        k->m_value=k->m_right->m_value;
        k->m_right=k->m_right->m_right;
        }
    UpdateAndBalance(k);
}

void AVLNode::UpdateAndBalance(AVLNode* &k)
{
    k->UpdateHeightAndBalance();
    if (k->m_balance<-1)
        {
        if (k->m_left->m_balance<=0)
            {
            RotateToTheRight(k);
            }
        else if (k->m_left->m_balance>0)
            {
```

```
            RotateToTheLeft(k->m_left);
            RotateToTheRight(k);
            }
        }
    else if (k->m_balance>1)
        {
        if (k->m_right->m_balance>=0)
            {
            RotateToTheLeft(k);
            }
        else if (k->m_right->m_balance<0)
            {
            RotateToTheRight(k->m_right);
            RotateToTheLeft(k);
            }
        }
}

bool AVLNode::Insert(AVLNode* &top,
                     const int key,
                     const int value)
{
    if (key==top->m_key)
        {
        return false;
        }

    if (key<top->m_key)
        {
        if (top->m_left)
            {
            if (!top->m_left->Insert(top->m_left,
                                     key,
                                     value))
                {
                return false;
                }
            }
        else
            {
            top->m_left=new AVLNode(key,
                                    value);
            if (!top->m_left)
                {
                return false;
```

```
                }
            }
        top->UpdateHeightAndBalance();
        if (top->m_balance<-1)
            {
            BalanceLeft(top);
            }
        }
    else
        {
        if (top->m_right)
            {
            if (!top->m_right->Insert(top->m_right,
                                      key,
                                      value))
                {
                return false;
                }
            }
        else
            {
            top->m_right=new AVLNode(key,
                                     value);
            if (!top->m_right)
                {
                return false;
                }
            }
        top->UpdateHeightAndBalance();
        if (top->m_balance>1)
            {
            BalanceRight(top);
            }
        }

    return true;
}

bool AVLNode::Remove(AVLNode* &top,
                     const int key)
{
    if (top->m_key==key)
        {
        if (!top->m_right)
            {
```

```
            top=top->m_left;
            }
        else if (!top->m_left)
            {
            top=top->m_right;
            }
        else if (top->m_balance<0)
            {
            RemoveMostRightOfLeft(top);
            }
        else if (top->m_balance>0)
            {
            RemoveMostLeftOfRight(top);
            }
        else if (s_chooseLeft)
            {
            RemoveMostRightOfLeft(top);
            s_chooseLeft=false;
            }
        else
            {
            RemoveMostLeftOfRight(top);
            s_chooseLeft=true;
            }
        }
    else if (top->m_key<key)
        {
        if (!top->m_right)
            {
            return false;
            }
        if (!Remove(top->m_right,
                    key))
            {
            return false;
            }
        UpdateAndBalance(top);
        }
    else
        {
        if (!top->m_left)
            {
            return false;
            }
        if (!Remove(top->m_left,
```

```
                    key))
            {
            return false;
            }
        UpdateAndBalance(top);
        }
    return true;
}

void *AVLNode::operator new(unsigned long)
{
    if (!s_firstFreeBlock)
        {
        Exit("Out of memory.");
        }
    AVLNode* const foundBlock=s_firstFreeBlock;
    s_firstFreeBlock=s_firstFreeBlock->m_right;
    return foundBlock;
}

void AVLNode::operator delete(void *ptr)
{
    AVLNode* const unwantedBlock=reinterpret_cast<AVLNode*>(ptr);
    unwantedBlock->m_right=s_firstFreeBlock;
    s_firstFreeBlock=unwantedBlock;
}

AVLNode::AVLNode(const int key,
                 const int value)
    : m_key(key),
      m_value(value),
      m_height(1),
      m_balance(0),
      m_left(nil),
      m_right(nil)
{
}

AVLNode::AVLNode(const AVLNode &original)
    : m_key(original.m_key),
      m_value(original.m_value),
      m_height(original.m_height),
      m_balance(original.m_balance)
{
    if (original.m_left)
```

```
        {
        m_left=new AVLNode(*original.m_left);
        if (!m_left)
            {
            Exit("Out of memory.");
            }
        }
    else
        {
        m_left=nil;
        }

    if (original.m_right)
        {
        m_right=new AVLNode(*original.m_right);
        if (!m_right)
            {
            Exit("Out of memory.");
            }
        }
    else
        {
        m_right=nil;
        }
}

AVLNode::~AVLNode()
{
    #ifdef __REZERO__
        m_key=0;
        m_value=0;
        m_height=0;
        m_balance=0;
    #endif

    if (m_left)
        {
        delete m_left;

        #ifdef __REZERO__
            m_left=nil;
        #endif
        }

    if (m_right)
```

```
        {
        delete m_right;

        #ifdef __REZERO__
            m_right=nil;
        #endif
        }
}

bool AVLNode::operator==(const AVLNode &x) const
{
    if (m_key!=x.m_key)
        {
        return false;
        }

    if (m_value!=x.m_value)
        {
        return false;
        }

    if (m_height!=x.m_height)
        {
        return false;
        }

    if (m_balance!=x.m_balance)
        {
        return false;
        }

    if (m_left)
        {
        if (x.m_left)
            {
            if (*m_left!=*x.m_left)
                {
                return false;
                }
            }
        else
            {
            return false;
            }
        }
```

```
    else if (x.m_left)
        {
        return false;
        }

    if (m_right)
        {
        if (x.m_right)
            {
            if (*m_right!=*x.m_right)
                {
                return false;
                }
            }
        else
            {
            return false;
            }
        }
    else if (x.m_right)
        {
        return false;
        }

    return true;
}

bool AVLNode::operator!=(const AVLNode &x) const
{
    if (operator==(x))
        {
        return false;
        }
    return true;
}

void AVLNode::operator=(const AVLNode &original)
{
    m_key=original.m_key;
    m_value=original.m_value;
    m_height=original.m_height;
    m_balance=original.m_balance;

    if (m_left)
        {
```

```
        if (original.m_left)
            {
            *m_left=*original.m_left;
            }
        else
            {
            delete m_left;
            m_left=nil;
            }
        }
    else if (original.m_left)
        {
        m_left=new AVLNode(*original.m_left);
        if (!m_left)
            {
            Exit("Out of memory.");
            }
        }

    if (m_right)
        {
        if (original.m_right)
            {
            *m_right=*original.m_right;
            }
        else
            {
            delete m_right;
            m_right=nil;
            }
        }
    else if (original.m_right)
        {
        m_right=new AVLNode(*original.m_right);
        if (!m_right)
            {
            Exit("Out of memory.");
            }
        }
}

void AVLNode::CalcHeightAndBalance(int &height,
                                   short &balance) const
{
    const int left_height=m_left ? m_left->m_height : 0;
```

```
    const int right_height=m_right ? m_right->m_height : 0;
    height=1+((left_height>right_height) ? left_height : right_height);
    balance=static_cast<short>(right_height-left_height);
}

void AVLNode::UpdateHeightAndBalance()
{
    CalcHeightAndBalance(m_height,
                         m_balance);
}

bool AVLNode::Verify() const
{
    int height;
    short balance;
    CalcHeightAndBalance(height,
                         balance);
    if (height!=m_height)
        {
        return false;
        }

    if (balance!=m_balance)
        {
        return false;
        }
    if (balance<-1)
        {
        return false;
        }
    if (balance>1)
        {
        return false;
        }

    if (m_left)
        {
        if (m_key<=m_left->m_key)
            {
            return false;
            }
        if (!m_left->Verify())
            {
            return false;
            }
```

```
        }

    if (m_right)
        {
        if (m_key<=m_right->m_key)
            {
            return false;
            }
        if (!m_right->Verify())
            {
            return false;
            }
        }

    return false;
}

bool AVLNode::Find(const int key,
                   int &value) const
{
    if (key==m_key)
        {
        value=m_value;
        return true;
        }

    if (key<m_key)
        {
        if (m_left)
            {
            if (m_left->Find(key,
                             value))
                {
                return true;
                }
            }
        }
    else
        {
        if (m_right)
            {
            if (m_right->Find(key,
                              value))
                {
                return true;
```

```
                }
            }
        }

    return false;
}
```

```
//=========================================
// Author:          © 2017 Cliff Huylebroeck
// Compiler:        MSL
// Project:         AVLTreeTest
// File:            Exit.h
// Implementation:  Exit.cp
//=========================================

#pragma once

void Exit(const char *message);
```

```
//====================================
// Author:   © 2017 Cliff Huylebroeck
// Compiler: MSL
// Project:  AVLTreeTest
// File:     Exit.cp
//====================================

#include "Exit.h"

void Exit(const char* const message)
{
    cout << "Error: ";
    cout << message;
    cout << endl;
    exit(0);
}
```

```
//=====================================
// Author:   © 2017 Cliff Huylebroeck
// Compiler: MSL
// Project:  AVLTreeTest
// File:     Debug.pch++
//=====================================

#pragma precompile_target "Debug"

#include "Both.h"

#define __DEBUG__
#define __REZERO__
```

```
//===================================
// Author:   © 2017 Cliff Huylebroeck
// Compiler: MSL
// Project:  AVLTreeTest
// File:     Release.pch++
//===================================

#pragma precompile_target "Release"

#include "Both.h"
```

```
//====================================
// Author:   © 2017 Cliff Huylebroeck
// Compiler: MSL
// Project:  AVLTreeTest
// File:     Both.h
//====================================

#pragma once

#include <iostream>
using namespace std;

#include "Profiler.h"
```