数据结构与算法分析 学习笔记

罗聪

一月 30, 2005

[首页, 上一页, 下一页; 目录]

最后更新: 2005年1月30日 星期日 13点25分12秒
HTML 转换程序: TeX2page 2004-09-11

目录

[首页, 上一页, 下一页; 目录]

第一章

前言

1.1 所选教材

我所选择的教材是《数据结构与算法分析——C语言描述》（原书第2版），英文版的名称是《Data Structures and Algorithm Analysis in C》，作者是：(美)Mark Allen Weiss。原书曾被评为20世纪顶尖的30部计算机著作之一。之所以选这本书，还因为它的简体中文版翻译得相当不错，几乎没有给我的阅读带来什么障碍。^_^

这本教科书所使用的是C语言，也许很多人会说C语言已经过时了，但是，我认为在数据结构的学习中，应该用尽量简单的语言，以免进入了语言的细枝末节中，反而冲淡了主题。实际上在国外的许多大学中（甚至中学），数据结构和算法分析的课程是选用Scheme的，例如MIT麻省理工大学极其著名的SICP课程。呵呵，语言又能说明什么呢？

1.2 写作原因

数据结构与算法分析是计算机专业的必修课——但遗憾的是，我在大学阶段并不是计算机专业的学生，以至于没有系统地跟着老师学习过这门课程。现在我已经工作了，在实际的工作中，我经常感到自己的基础知识不够，有很多问题无法解决。在经历了一段痛苦的斗争后，我选择了自学的道路，想把这门课程扎扎实实地学好。

教科书中已经给出了大部分的代码，因此，我基本上也只是重复敲入了一次而已（或者是改写成C++），但这并不是没有意义的。我们在看书的时候经常会觉得自己已经懂了，但如果真的要亲自动手去做了，却会感到无法下手。我认为，亲自输入一次代码并调试通过，比任何空谈都有效。

在具体的代码实现上，我可能会参考MFC、STL……但也可能会进行一定的修改。

1.3 一些约定

我使用的是Visual C++ 6.0编译器，并将会用C/C++来撰写代码（我可能会用C++改写原书中的例子，以便能用在工作中，但一些地方还是会用C），不会使用任何与平台相关的特性（因此可以保证有比较好的移植性）。原书中的代码风格跟我平时的代码风格非常相近，但有一些地方我可能会进行一些改动。

我认为数据结构的代码不需要任何界面，因此，请您新建一个工程，类型为Win32 Console Application，即控制台工程。然后添加一个.h头文件和一个.c/.cpp文件。头文件中，我一般会写3行固定格式的预编译语句，如下：

```
#ifndef __LIST_H__
#define __LIST_H__

// TODO: Add header body code here

#endif  // __LIST_H__
```

表示这是一个list.h。

另外，C++操作符new的实现在不同的编译器中都不太一样，在VC6中，如果new失败，则会返回NULL，程序中我用检测返回值是否为NULL来判断new是否成功，但如果这个代码是用别的编译器编译的，则要特别注意别的编译器是否也是用NULL来表示new失败的，否则很可能会导致无法意料的结果。

为了方便调试内存泄漏，我会在一些地方写入这样的代码：

```
#include <assert.h>
#include <crtdbg.h>

#ifdef _DEBUG
#define DEBUG_NEW new (_NORMAL_BLOCK, THIS_FILE, __LINE__)
#endif

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#ifdef _DEBUG
#ifndef ASSERT
#define ASSERT  assert
#endif
#else   // not _DEBUG
#ifndef ASSERT
#define ASSERT
#endif
#endif  // _DEBUG
```

以及：

```
#ifdef _DEBUG
   _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif
```

在阅读时不用管它们，直接略过即可。

1.4 历史记录

[2005-01-30] 图的遍历一章。
[2005-01-28] 图的储存一章。
[2005-01-24] 排序一章。
[2005-01-21] AVL树一章。
[2005-01-18] 二叉搜索树一章。
[2005-01-14] 二叉树一章。
[2005-01-09] 递归一章。
[2005-01-08] 队列一章。
[2005-01-07] 栈一章。
[2005-01-05] 循环链表一章。
[2005-01-04] 双链表一章。
[2004-12-30] 单链表一章。

1.5 联系方式

作者：罗聪
主页：**http://www.luocong.com**
E-Mail：admin[AT]luocong.com

第二章

单链表

链表是最常用、最简单和最基本的数据结构之一。我们先来看看单链表的实现。

2.1 代码实现

单链表的实现如下：

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  slist.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2004-12-29 9:58:38
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#ifndef __SINGLE_LIST_H__
#define __SINGLE_LIST_H__

#include <assert.h>
#include <crtdbg.h>

#ifdef _DEBUG
#define DEBUG_NEW new (_NORMAL_BLOCK, THIS_FILE, __LINE__)
#endif

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#ifdef _DEBUG
#ifndef ASSERT
#define ASSERT  assert
#endif
#else   // not _DEBUG
#ifndef ASSERT
#define ASSERT
#endif
#endif  // _DEBUG

template<typename T>
class CNode
{
public:
    T data;
    CNode<T> *next;
    CNode() : data(T()), next(NULL) {}
```

```
    CNode(const T &initdata) : data(initdata), next(NULL) {}
    CNode(const T &initdata, CNode<T> *p) : data(initdata), next(p) {}
};

template<typename T>
class CSList
{
protected:
    int m_nCount;
    CNode<T> *m_pNodeHead;

public:
    CSList();
    CSList(const T &initdata);
    ~CSList();

public:
    int     IsEmpty() const;
    int     GetCount() const;
    int     InsertBefore(const int pos, const T data);
    int     InsertAfter(const int pos, const T data);
    int     AddHead(const T data);
    int     AddTail(const T data);
    void    RemoveAt(const int pos);
    void    RemoveHead();
    void    RemoveTail();
    void    RemoveAll();
    T&      GetTail();
    T       GetTail() const;
    T&      GetHead();
    T       GetHead() const;
    T&      GetAt(const int pos);
    T       GetAt(const int pos) const;
    void    SetAt(const int pos, T data);
    int     Find(const T data) const;
};

template<typename T>
inline CSList<T>::CSList() : m_nCount(0), m_pNodeHead(NULL)
{
}

template<typename T>
inline CSList<T>::CSList(const T &initdata) : m_nCount(0), m_pNodeHead(NULL)
{
    AddHead(initdata);
}

template<typename T>
inline CSList<T>::~CSList()
{
    RemoveAll();
}

template<typename T>
inline int CSList<T>::IsEmpty() const
{
    return 0 == m_nCount;
}

template<typename T>
inline int CSList<T>::AddHead(const T data)
{
    CNode<T> *pNewNode;

    pNewNode = new CNode<T>;
    if (NULL == pNewNode)
```

```
        return 0;

    pNewNode->data = data;
    pNewNode->next = m_pNodeHead;

    m_pNodeHead = pNewNode;
    ++m_nCount;

    return 1;
}

template<typename T>
inline int CSList<T>::AddTail(const T data)
{
    return InsertAfter(GetCount(), data);
}

// if success, return the position of the new node.
// if fail, return 0.
template<typename T>
inline int CSList<T>::InsertBefore(const int pos, const T data)
{
    int i;
    int nRetPos;
    CNode<T> *pTmpNode1;
    CNode<T> *pTmpNode2;
    CNode<T> *pNewNode;

    pNewNode = new CNode<T>;
    if (NULL == pNewNode)
    {
        nRetPos = 0;
        goto Exit0;
    }

    pNewNode->data = data;

    // if the list is empty, replace the head node with the new node.
    if (NULL == m_pNodeHead)
    {
        pNewNode->next = NULL;
        m_pNodeHead = pNewNode;
        nRetPos = 1;
        goto Exit1;
    }

    // is pos range valid?
    ASSERT(1 <= pos && pos <= m_nCount);

    // insert before head node?
    if (1 == pos)
    {
        pNewNode->next = m_pNodeHead;
        m_pNodeHead = pNewNode;
        nRetPos = 1;
        goto Exit1;
    }

    // if the list is not empty and is not inserted before head node,
    // seek to the pos of the list and insert the new node before it.
    pTmpNode1 = m_pNodeHead;
    for (i = 1; i < pos; ++i)
    {
        pTmpNode2 = pTmpNode1;
        pTmpNode1 = pTmpNode1->next;
    }
    pNewNode->next = pTmpNode1;
```

```
    pTmpNode2->next = pNewNode;

    nRetPos = pos;

Exit1:
    ++m_nCount;
Exit0:
    return nRetPos;
}

// if success, return the position of the new node.
// if fail, return 0.
template<typename T>
inline int CSList<T>::InsertAfter(const int pos, const T data)
{
    int i;
    int nRetPos;
    CNode<T> *pTmpNode;
    CNode<T> *pNewNode;

    pNewNode = new CNode<T>;
    if (NULL == pNewNode)
    {
        nRetPos = 0;
        goto Exit0;
    }

    pNewNode->data = data;

    // if the list is empty, replace the head node with the new node.
    if (NULL == m_pNodeHead)
    {
        pNewNode->next = NULL;
        m_pNodeHead = pNewNode;
        nRetPos = 1;
        goto Exit1;
    }

    // is pos range valid?
    ASSERT(1 <= pos && pos <= m_nCount);

    // if the list is not empty,
    // seek to the pos of the list and insert the new node after it.
    pTmpNode = m_pNodeHead;
    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }
    pNewNode->next = pTmpNode->next;
    pTmpNode->next = pNewNode;

    nRetPos = pos + 1;

Exit1:
    ++m_nCount;
Exit0:
    return nRetPos;
}

template<typename T>
inline int CSList<T>::GetCount() const
{
    return m_nCount;
}

template<typename T>
inline void CSList<T>::RemoveAt(const int pos)
```

```
{
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode1;
    CNode<T> *pTmpNode2;

    pTmpNode1 = m_pNodeHead;

    // head node?
    if (1 == pos)
    {
        m_pNodeHead = m_pNodeHead->next;
        goto Exit1;
    }

    for (i = 1; i < pos; ++i)
    {
        // we will get the previous node of the target node after
        // the for loop finished, and it would be stored into pTmpNode2
        pTmpNode2 = pTmpNode1;
        pTmpNode1 = pTmpNode1->next;
    }
    pTmpNode2->next = pTmpNode1->next;

Exit1:
    delete pTmpNode1;
    --m_nCount;
}

template<typename T>
inline void CSList<T>::RemoveHead()
{
    ASSERT(0 != m_nCount);
    RemoveAt(1);
}

template<typename T>
inline void CSList<T>::RemoveTail()
{
    ASSERT(0 != m_nCount);
    RemoveAt(m_nCount);
}

template<typename T>
inline void CSList<T>::RemoveAll()
{
    int i;
    int nCount;
    CNode<T> *pTmpNode;

    nCount = m_nCount;
    for (i = 0; i < nCount; ++i)
    {
        pTmpNode = m_pNodeHead->next;
        delete m_pNodeHead;
        m_pNodeHead = pTmpNode;
    }

    m_nCount = 0;
}

template<typename T>
inline T& CSList<T>::GetTail()
{
    ASSERT(0 != m_nCount);
```

```
    int i;
    int nCount;
    CNode<T> *pTmpNode = m_pNodeHead;

    nCount = m_nCount;
    for (i = 1; i < nCount; ++i)
    {
        pTmpNode = pTmpNode->next;
    }

    return pTmpNode->data;
}

template<typename T>
inline T CSList<T>::GetTail() const
{
    ASSERT(0 != m_nCount);

    int i;
    int nCount;
    CNode<T> *pTmpNode = m_pNodeHead;

    nCount = m_nCount;
    for (i = 1; i < nCount; ++i)
    {
        pTmpNode = pTmpNode->next;
    }

    return pTmpNode->data;
}

template<typename T>
inline T& CSList<T>::GetHead()
{
    ASSERT(0 != m_nCount);
    return m_pNodeHead->data;
}

template<typename T>
inline T CSList<T>::GetHead() const
{
    ASSERT(0 != m_nCount);
    return m_pNodeHead->data;
}

template<typename T>
inline T& CSList<T>::GetAt(const int pos)
{
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode = m_pNodeHead;

    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }

    return pTmpNode->data;
}

template<typename T>
inline T CSList<T>::GetAt(const int pos) const
{
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
```

```
    CNode<T> *pTmpNode = m_pNodeHead;

    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }

    return pTmpNode->data;
}

template<typename T>
inline void CSList<T>::SetAt(const int pos, T data)
{
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode = m_pNodeHead;

    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }
    pTmpNode->data = data;
}

template<typename T>
inline int CSList<T>::Find(const T data) const
{
    int i;
    int nCount;
    CNode<T> *pTmpNode = m_pNodeHead;

    nCount = m_nCount;
    for (i = 0; i < nCount; ++i)
    {
        if (data == pTmpNode->data)
            return i + 1;
        pTmpNode = pTmpNode->next;
    }

    return 0;
}

#endif  // __SINGLE_LIST_H__
```

调用如下：

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   slist.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2004-12-29 10:41:18
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include "slist.h"
using namespace std;

int main()
{
    int i;
    int nCount;
    CSList<int> slist;
```

```
#ifdef _DEBUG
   _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

   slist.InsertAfter(slist.InsertAfter(slist.AddHead(1), 2), 3);
   slist.InsertAfter(slist.InsertAfter(slist.GetCount(), 4), 5);
   slist.InsertAfter(slist.GetCount(), 6);
   slist.AddTail(10);
   slist.InsertAfter(slist.InsertBefore(slist.GetCount(), 7), 8);
   slist.SetAt(slist.GetCount(), 9);
   slist.RemoveHead();
   slist.RemoveTail();

   // print out elements
   nCount = slist.GetCount();
   for (i = 0; i < nCount; ++i)
      cout << slist.GetAt(i + 1) << endl;
}
```

代码比较简单，一看就明白，懒得解释了。如果有bug，请告诉我。

2.2 效率问题

考虑到效率的问题，代码中声明了一个成员变量：m_nCount，用它来记录链表的结点个数。这样有什么好处呢？在某些情况下就不用遍历链表了，例如，至少在GetCount()时能提高速度。

原书中提到了一个"表头"（header）或"哑结点"（dummy node）的概念，这个结点作为第一个结点，位置在0，它是不用的，我个人认为这样做有点浪费空间，所以并没有采用这种做法。

单链表在效率上最大的问题在于，如果要插入一个结点到链表的末端或者删除末端的一个结点，则需要遍历整个链表，时间复杂度是O(N)。平均来说，要访问一个结点，时间复杂度也有O(N/2)。这是链表本身的性质所造成的，没办法解决。不过我们可以采用双链表和循环链表来改善这种情况。

2.3 应用：一元多项式（加法和乘法）

2.3.1 基础知识

我们使用一元多项式来说明单链表的应用。假设有两个一元多项式：

```
P1(X) = X^2 + 2X + 3
```

以及

```
P2(X) = 3X^3 + 10X + 6
```

现在运用中学的基础知识，计算它们的和：

```
P1(X) + P2(X) = (X^2 + 2X + 3) + (3X^3 + 10X + 6)
          = 3X^3 + 1X^2 + 12X^1 + 9
```

以及计算它们的乘积：

```
P1(X) * P2(X) = (X^2 + 2X + 3) * (3X^3 + 10X + 6)
          = 3X^5 + 6X^4 + 19X^3 + 26X^2 + 42X^1 + 18
```

怎么样，很容易吧？:) 但我们是灵长类动物，这么繁琐的计算怎么能用手工来完成呢？（试想一下，如果多项式非常大的话……）我们的目标是用计算机来完成这些计算任务，代码就在下面。

2.3.2 代码实现

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   poly.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2004-12-30 17:32:54
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include "slist.h"

#define Max(x,y) (((x)>(y)) ? (x) : (y))

typedef struct tagPOLYNOMIAL
{
    CSList<int> Coeff;
    int HighPower;
} * Polynomial;

static void AddPolynomial(
    Polynomial polysum,
    const Polynomial poly1,
    const Polynomial poly2
)
{
    int i;
    int sum;
    int tmp1;
    int tmp2;

    polysum->HighPower = Max(poly1->HighPower, poly2->HighPower);
    for (i = 1; i <= polysum->HighPower + 1; ++i)
    {
        tmp1 = poly1->Coeff.GetAt(i);
        tmp2 = poly2->Coeff.GetAt(i);
        sum = tmp1 + tmp2;
        polysum->Coeff.AddTail(sum);
    }
}

static void MulPolynomial(
    Polynomial polymul,
    const Polynomial poly1,
    const Polynomial poly2
)
{
    int i;
    int j;
    int tmp;
    int tmp1;
    int tmp2;

    polymul->HighPower = poly1->HighPower + poly2->HighPower;
```

```
    // initialize all elements to zero
    for (i = 0; i <= polymul->HighPower; ++i)
        polymul->Coeff.AddTail(0);

    for (i = 0; i <= poly1->HighPower; ++i)
    {
        tmp1 = poly1->Coeff.GetAt(i + 1);
        for (j = 0; j <= poly2->HighPower; ++j)
        {
            tmp = polymul->Coeff.GetAt(i + j + 1);
            tmp2 = poly2->Coeff.GetAt(j + 1);
            tmp += tmp1 * tmp2;
            polymul->Coeff.SetAt(i + j + 1, tmp);
        }
    }
}

static void PrintPoly(const Polynomial poly)
{
    int i;

    for (i = poly->HighPower; i > 0; i-- )
        printf( "%dX^%d + ", poly->Coeff.GetAt(i + 1), i);
    printf("%d\n", poly->Coeff.GetHead());
}

int main()
{
    Polynomial poly1 = NULL;
    Polynomial poly2 = NULL;
    Polynomial polyresult = NULL;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    poly1 = new (struct tagPOLYNOMIAL);
    if (NULL == poly1)
        goto Exit0;

    poly2 = new (struct tagPOLYNOMIAL);
    if (NULL == poly2)
        goto Exit0;

    polyresult = new (struct tagPOLYNOMIAL);
    if (NULL == polyresult)
        goto Exit0;

    // P1(X) = X^2 + 2X + 3
    poly1->HighPower = 2;
    poly1->Coeff.AddHead(0);
    poly1->Coeff.AddHead(1);
    poly1->Coeff.AddHead(2);
    poly1->Coeff.AddHead(3);

    // P2(X) = 3X^3 + 10X + 6
    poly2->HighPower = 3;
    poly2->Coeff.AddHead(3);
    poly2->Coeff.AddHead(0);
    poly2->Coeff.AddHead(10);
    poly2->Coeff.AddHead(6);

    // add result = 3X^3 + 1X^2 + 12X^1 + 9
    AddPolynomial(polyresult, poly1, poly2);
    PrintPoly(polyresult);
```

```
    // reset
    polyresult->Coeff.RemoveAll();

    // mul result = 3X^5 + 6X^4 + 19X^3 + 26X^2 + 42X^1 + 18
    MulPolynomial(polyresult, poly1, poly2);
    PrintPoly(polyresult);

Exit0:
    if (poly1)
    {
        delete poly1;
        poly1 = NULL;
    }
    if (poly2)
    {
        delete poly2;
        poly2 = NULL;
    }
    if (polyresult)
    {
        delete polyresult;
        polyresult = NULL;
    }
}
```

2.3.3 说明

原书中只给出了一元多项式的数组实现，而没有给出单链表的代码。实际上用单链表最大的好处在于多项式的项数可以为任意大。（当然只是理论上的。什么？你的内存是无限大的？好吧，当我没说……）

我没有实现减法操作，实际上减法可以转换成加法来完成，例如 a - b 可以换算成 a + (-b)，那么我们的目标就转变为做一个负号的运算了。至于除法，可以通过先换算“-”，然后再用原位加法来计算。（现在你明白加法有多重要了吧？^_^）有兴趣的话，不妨您试试完成它，我的目标只是掌握单链表的使用，因此不再继续深究。

[首页, 上一页, 下一页; 目录]

第三章

双链表

单链表学完后，理所当然的就是轮到双链表了。

3.1 代码实现

双链表的实现如下：

```
//////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   dlist.h
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-4 10:33:21
//  Comment     :
//
//////////////////////////////////////////////////////////////////////////////

#ifndef __DOUBLE_LIST_H__
#define __DOUBLE_LIST_H__

#include <assert.h>
#include <crtdbg.h>

#ifdef _DEBUG
#define DEBUG_NEW new (_NORMAL_BLOCK, THIS_FILE, __LINE__)
#endif

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#ifdef _DEBUG
#ifndef ASSERT
#define ASSERT  assert
#endif
#else   // not _DEBUG
#ifndef ASSERT
#define ASSERT
#endif
#endif  // _DEBUG

template<typename T>
class CNode
{
public:
    T data;
    CNode<T> *prior;
    CNode<T> *next;
```

```
    CNode() : data(T()), prior(NULL), next(NULL) {}
    CNode(const T &initdata) : data(initdata), prior(NULL), next(NULL) {}
};

template<typename T>
class CDList
{
protected:
    int m_nCount;
    CNode<T> *m_pNodeHead;
    CNode<T> *m_pNodeTail;

public:
    CDList();
    CDList(const T &initdata);
    ~CDList();

public:
    int     IsEmpty() const;
    int     GetCount() const;
    int     InsertBefore(const int pos, const T data);
    int     InsertAfter(const int pos, const T data);
    int     AddHead(const T data);
    int     AddTail(const T data);
    void    RemoveAt(const int pos);
    void    RemoveHead();
    void    RemoveTail();
    void    RemoveAll();
    T&      GetTail();
    T       GetTail() const;
    T&      GetHead();
    T       GetHead() const;
    T&      GetAt(const int pos);
    T       GetAt(const int pos) const;
    void    SetAt(const int pos, T data);
    int     Find(const T data) const;
    T&      GetPrev(int &pos);
    T&      GetNext(int &pos);
};

template<typename T>
inline CDList<T>::CDList() : m_nCount(0), m_pNodeHead(NULL), m_pNodeTail(NULL)
{
}

template<typename T>
inline CDList<T>::CDList(const T &initdata)
                : m_nCount(0), m_pNodeHead(NULL), m_pNodeTail(NULL)
{
    AddHead(initdata);
}

template<typename T>
inline CDList<T>::~CDList()
{
    RemoveAll();
}

template<typename T>
inline T& CDList<T>::GetNext(int &pos)
{
    ASSERT(0 != m_nCount);
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode = m_pNodeHead;
```

```
    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }

    ++pos;

    return pTmpNode->data;
}

template<typename T>
inline T& CDList<T>::GetPrev(int &pos)
{
    ASSERT(0 != m_nCount);
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode = m_pNodeHead;

    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }

    --pos;

    return pTmpNode->data;
}

template<typename T>
inline int CDList<T>::InsertBefore(const int pos, const T data)
{
    int i;
    int nRetPos;
    CNode<T> *pTmpNode;
    CNode<T> *pNewNode;

    pNewNode = new CNode<T>;
    if (NULL == pNewNode)
    {
        nRetPos = 0;
        goto Exit0;
    }

    pNewNode->data = data;

    // if the list is empty, replace the head node with the new node.
    if (NULL == m_pNodeHead)
    {
        pNewNode->prior = NULL;
        pNewNode->next = NULL;
        m_pNodeHead = pNewNode;
        m_pNodeTail = pNewNode;
        nRetPos = 1;
        goto Exit1;
    }

    // is pos range valid?
    ASSERT(1 <= pos && pos <= m_nCount);

    // insert before head node?
    if (1 == pos)
    {
        pNewNode->prior = NULL;
        pNewNode->next = m_pNodeHead;
        m_pNodeHead->prior = pNewNode;
        m_pNodeHead = pNewNode;
```

```
        nRetPos = 1;
        goto Exit1;
    }

    // if the list is not empty and is not inserted before head node,
    // seek to the pos of the list and insert the new node before it.
    pTmpNode = m_pNodeHead;
    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }
    pNewNode->next = pTmpNode;
    pNewNode->prior = pTmpNode->prior;

    pTmpNode->prior->next = pNewNode;
    pTmpNode->prior = pNewNode;

    // if tail node, must update m_pNodeTail
    if (NULL == pNewNode->next)
    {
        m_pNodeTail = pNewNode;
    }

    nRetPos = pos;

Exit1:
    ++m_nCount;
Exit0:
    return nRetPos;
}

template<typename T>
inline int CDList<T>::InsertAfter(const int pos, const T data)
{
    int i;
    int nRetPos;
    CNode<T> *pNewNode;
    CNode<T> *pTmpNode;

    pNewNode = new CNode<T>;
    if (NULL == pNewNode)
    {
        nRetPos = 0;
        goto Exit0;
    }

    pNewNode->data = data;

    // if the list is empty, replace the head node with the new node.
    if (NULL == m_pNodeHead)
    {
        pNewNode->prior = NULL;
        pNewNode->next = NULL;
        m_pNodeHead = pNewNode;
        m_pNodeTail = pNewNode;
        nRetPos = 1;
        goto Exit1;
    }

    // is pos range valid?
    ASSERT(1 <= pos && pos <= m_nCount);

    // if the list is not empty,
    // seek to the pos of the list and insert the new node after it.
    pTmpNode = m_pNodeHead;
    for (i = 1; i < pos; ++i)
    {
```

```
        pTmpNode = pTmpNode->next;
    }

    pNewNode->next = pTmpNode->next;
    pNewNode->prior = pTmpNode;

    // if NewNode's position is m_pNodeTail, update m_pNodeTail
    if (pTmpNode->next == m_pNodeTail)
    {
        m_pNodeTail->prior = pNewNode;
    }

    pTmpNode->next = pNewNode;

    // if tail node, must update m_pNodeTail
    if (NULL == pNewNode->next)
    {
        m_pNodeTail = pNewNode;
    }

    nRetPos = pos + 1;

Exit1:
    ++m_nCount;
Exit0:
    return nRetPos;
}

template<typename T>
inline T& CDList<T>::GetAt(const int pos)
{
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode = m_pNodeHead;

    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }

    return pTmpNode->data;
}

template<typename T>
inline T CDList<T>::GetAt(const int pos) const
{
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode = m_pNodeHead;

    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }

    return pTmpNode->data;
}

template<typename T>
inline int CDList<T>::AddHead(const T data)
{
    return InsertBefore(1, data);
}

template<typename T>
```

```
inline int CDList<T>::AddTail(const T data)
{
   return InsertAfter(GetCount(), data);
}

template<typename T>
inline CDList<T>::IsEmpty() const
{
   return 0 == m_nCount;
}

template<typename T>
inline CDList<T>::GetCount() const
{
   return m_nCount;
}

template<typename T>
inline T& CDList<T>::GetTail()
{
   ASSERT(0 != m_nCount);
   return m_pNodeTail->data;
}

template<typename T>
inline T CDList<T>::GetTail() const
{
   ASSERT(0 != m_nCount);
   return m_pNodeTail->data;
}

template<typename T>
inline T& CDList<T>::GetHead()
{
   ASSERT(0 != m_nCount);
   return m_pNodeHead->data;
}

template<typename T>
inline T CDList<T>::GetHead() const
{
   ASSERT(0 != m_nCount);
   return m_pNodeHead->data;
}

template<typename T>
inline void CDList<T>::RemoveAt(const int pos)
{
   ASSERT(1 <= pos && pos <= m_nCount);

   int i;
   CNode<T> *pTmpNode = m_pNodeHead;

   // head node?
   if (1 == pos)
   {
      m_pNodeHead = m_pNodeHead->next;
      goto Exit1;
   }

   for (i = 1; i < pos; ++i)
   {
      pTmpNode = pTmpNode->next;
   }
   pTmpNode->prior->next = pTmpNode->next;

Exit1:
```

```
    delete pTmpNode;
    --m_nCount;
    if (0 == m_nCount)
    {
        m_pNodeTail = NULL;
    }
}

template<typename T>
inline void CDList<T>::RemoveHead()
{
    ASSERT(0 != m_nCount);
    RemoveAt(1);
}

template<typename T>
inline void CDList<T>::RemoveTail()
{
    ASSERT(0 != m_nCount);
    RemoveAt(m_nCount);
}

template<typename T>
inline void CDList<T>::RemoveAll()
{
    int i;
    int nCount;
    CNode<T> *pTmpNode;

    nCount = m_nCount;
    for (i = 0; i < nCount; ++i)
    {
        pTmpNode = m_pNodeHead->next;
        delete m_pNodeHead;
        m_pNodeHead = pTmpNode;
    }

    m_nCount = 0;
}

template<typename T>
inline void CDList<T>::SetAt(const int pos, T data)
{
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode = m_pNodeHead;

    for (i = 1; i < pos; ++i)
    {
        pTmpNode = pTmpNode->next;
    }
    pTmpNode->data = data;
}

template<typename T>
inline int CDList<T>::Find(const T data) const
{
    int i;
    int nCount;
    CNode<T> *pTmpNode = m_pNodeHead;

    nCount = m_nCount;
    for (i = 0; i < nCount; ++i)
    {
        if (data == pTmpNode->data)
            return i + 1;
```

```
        pTmpNode = pTmpNode->next;
    }

    return 0;
}

#endif  // __DOUBLE_LIST_H__
```

调用如下：

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   dlist.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-4 10:58:22
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include "dlist.h"
using namespace std;

int main()
{
    int i;
    int nCount;
    CDList<int> dlist;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    dlist.AddTail(1);
    dlist.AddTail(3);
    dlist.InsertBefore(2, 2);
    dlist.AddHead(4);
    dlist.RemoveTail();

    nCount = dlist.GetCount();
    for (i = 1; i <= nCount;)
    {
        cout << dlist.GetNext(i) << endl;
    }
}
```

3.2 说明

单链表的结点中只有一个指向直接后继结点的指针，所以，从某个结点出发只能顺着指针往后查询其他的结点。靠，那如果我想访问某个结点的前一个结点，岂不只能重新从表头结点开始了？效率真低啊！换句话说，在单链表中，GetNext()的时间复杂度为O(1)，而GetPrev()的时间复杂度则为O(N)。为克服单链表这种单向性的缺点，我们可以利用——“当当当当”，Only you，就是——双链表。

顾名思义，在双链表的结点中有两个指针，一个指向直接后继，另一个指向直接前驱，在C++语言中表示如下：

```
struct Node
{
    struct Node *prior;
    struct Node *next;
    T data;
```

```
};
```

大部分对双链表的操作（只涉及到向后方向的指针的操作）都与单链表的相同，但在插入、删除时有很大的不同，在双链表中需同时修改两个方向上的指针。因此，可以直接继承单链表的类来完成双链表，然后改改不一样的函数就行了。但我没有这样做，别问为什么，人品问题而已。

如果你已经熟练掌握了单链表的指针域，那么双链表的这部分应该难不倒你了。不多说了，请看代码吧。如果有bug，请告诉我。^_^

[首页, 上一页, 下一页; 目录]

第四章

循环链表

4.1 基本概念

循环链表可以为单链表，也可以为双链表，但我不想把问题搞得那么复杂，姑且就做单链表的循环形式吧。

我们在实现了链表后，必然会提出一个问题：链表能不能首尾相连？怎样实现？

答案：能。其实实现的方法很简单，就是将表中最后一个结点的指针域指向头结点即可（P->next = head;）。这种形成环路的链表称为循环链表。

试想我们在学校的运动场上跑步锻炼身体（学校……好遥远的记忆啊），绕着400米跑道一直跑啊跑，好像永远没有尽头一样。这是因为跑道的首尾是相连的，跑完一圈后，“尾巴”突然就变成了“头”，这跟循环链表的原理是一样的。好了，明白了这个道理，实现起来就简单了，不过要注意的是，在循环链表里面如果要获得结点的个数，不能采用while()循环来遍历表，因为这个循环是永不会结束的，这就像无论有多长的长跑比赛都可以在400米的跑道上进行一样。我的做法还是通过增加一个m_nCount变量，每次新增或删除一个结点就对m_nCount进行相应的操作。

循环链表的特点：

1. 从任一结点出发均可找到表中其他结点。
2. 操作仅有一点与单链表不同：循环条件。
    - 单链表：P = NULL 或 P->next = NULL
    - 循环链表：P = head 或 P->next = head

4.2 代码实现

循环链表的实现如下：

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  clist.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-5 10:43:17
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#ifndef __CIRC_LIST_H__
#define __CIRC_LIST_H__

#include "../../slist/src/slist.h"
```

```
template<typename T>
class CCList : public CSList<T>
{
protected:
    CNode<T> *m_pNodeCurr;

public:
    CCList();

public:
    T&      GetNext();
    void    RemoveAt(const int pos);
    int     GetCurrentIndex() const;
};

template<typename T>
inline T& CCList<T>::GetNext()
{
    ASSERT(0 != m_nCount);

    if ((NULL == m_pNodeCurr) || (NULL == m_pNodeCurr->next))
        m_pNodeCurr = m_pNodeHead;
    else
        m_pNodeCurr = m_pNodeCurr->next;

    return m_pNodeCurr->data;
}

template<typename T>
inline int CCList<T>::GetCurrentIndex() const
{
    ASSERT(0 != m_nCount);

    int i;
    CNode<T> *pTmpNode = m_pNodeHead;

    for (i = 1; i <= m_nCount; ++i)
    {
        if (pTmpNode == m_pNodeCurr)
            return i;
        else
            pTmpNode = pTmpNode->next;
    }

    return 0;
}

template<typename T>
inline void CCList<T>::RemoveAt(const int pos)
{
    ASSERT(1 <= pos && pos <= m_nCount);

    int i;
    CNode<T> *pTmpNode1;
    CNode<T> *pTmpNode2;

    pTmpNode1 = m_pNodeHead;

    // head node?
    if (1 == pos)
    {
        m_pNodeHead = m_pNodeHead->next;

        // added for loop list
        // m_pNodeCurr will be set to m_pNodeHead in function GetNext()
        m_pNodeCurr = NULL;
```

```
        goto Exit1;
    }

    for (i = 1; i < pos; ++i)
    {
        // we will get the previous node of the target node after
        // the for loop finished, and it would be stored into pTmpNode2
        pTmpNode2 = pTmpNode1;
        pTmpNode1 = pTmpNode1->next;
    }
    pTmpNode2->next = pTmpNode1->next;

    // added for loop list
    m_pNodeCurr = pTmpNode2;

Exit1:
    delete pTmpNode1;
    --m_nCount;
}

template<typename T>
inline CCList<T>::CCList() : m_pNodeCurr(NULL)
{
}

#endif  // __CIRC_LIST_H__
```

4.3 说明

由于循环链表的操作大部分是与非循环链表相同的，因此我的循环链表是直接从单链表继承来的，并且新增了表示当前结点的变量m_pNodeCurr，以及重载了几个函数。但还有两点是需要特别注意的：

1. 在GetNext()函数中，必须有判断当前结点应该如何指向下一个结点的条件。
2. 在RemoveAt()函数中，如果要删除一个结点，而该结点又恰好是头结点的话，那么当前结点必须指向NULL，这样才能在GetNext()中重新获得头结点的正确的值。

关于这两点应该毫无疑问吧？呵呵，那就让我们继续吧……什么？你不明白第二点是什么意思？我倒！

让我们来假定一下，如果当前结点指向了尾结点，然后这时我们调用了GetNext()，那么很显然，当前结点就应该指向头结点了。但问题是头结点已经被我们删除了，那么当前结点还能指向哪里呢？这时什么事情都可能发生，计算机可能会格式化了你的硬盘，也可能会把你的情书送给了班里的恐龙，更可能会告诉你的老板你愿意从此以后一分钱工资都不要一直做到over为止……但最有可能发生的事情是产生一个内存访问的异常，所以，咳咳，计算机是很笨的，必须由我们亲自告诉它："头结点已经完蛋啦，所以当前结点就指向NULL吧，你在GetNext()函数中自个儿给我解决好下一步的问题。"

明白了吗？还不明白的话……我……

4.4 应用：约瑟夫问题

约瑟夫问题几乎是最经典的用来讲解循环链表的案例了。为什么呢？我们来看看这个问题的描述就会明白了：

有一队由n个冒险家组成的探险队深入到热带雨林中，但他们遭遇到了食人族，食人族的游戏规则是让他们围成一圈，然后选定一个数字m，从第1个人开始报数，报到m时，这个人就要被吃掉了，然后从下一个人开始又重新从1报数，重复这个过程，直到剩下最后一个人，这个人是幸运者，可以离开而不被吃掉。那么问题是，谁是这个幸运者呢？

我们来举个例子：

假设这个探险队有6个探险家，食人族选定的数字m是5，那么在第一轮中，5号会被吃掉，剩下的就是：1, 2, 3, 4, 6总共5个人，然后从6号开始，重新从1开始报5个数：6, 1, 2, 3, 4，所以在第二轮里面被吃掉的就是4号……一直重复这个过程，按顺序应该是：5, 4, 6, 2, 3被吃掉，剩下1号活下来。

解决这个问题并不是只能用循环链表的，但使用循环链表应该是最方便的。我写的代码如下：

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  joseph.cpp
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-5 13:56:32
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include "clist.h"
using namespace std;

int main()
{
    int i;
    int n;
    int m;
    int nNumber;
    int nCurIndex;
    CCList<int> clist;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    cout << "请输入总的人数： ";
    cin >> n;

    cout << "请输入死亡号码： ";
    cin >> m;

    // 初始化序列号码列表：
    for (i = 1; i <= n; ++i)
    {
        clist.AddTail(i);
    }

    i = 0;
    do
    {
        ++i;
        nNumber = clist.GetNext();
        if (i == m)
        {
            cout << "第 " << nNumber << " 个人被吃掉了！" << endl;

            // 这个人倒霉了
            nCurIndex = clist.GetCurrentIndex();
            clist.RemoveAt(nCurIndex);
            --n;

            // 剩下的人重新开始报数
            i = 0;
```

```
        }
    } while (1 != n);

    cout << "最后活下来的是： " << clist.GetHead() << endl;
}
```

为了解决约瑟夫问题，我在循环链表中加入了GetCurrentIndex()函数，用来获得当前结点的索引值，以便删除当前结点。整个代码应该不难理解，实际动手做做就明白了。 :)

[首页, 上一页, 下一页; 目录]

第五章

栈

5.1 基本概念

栈（stack）是限制插入和删除只能在一个位置上进行的表，该位置是表的末端，叫做栈的顶（top），它是后进先出（LIFO）的。对栈的基本操作只有push（进栈）和pop（出栈）两种，前者相当于插入，后者相当于删除最后的元素。

由于栈在本质上是一种受限制的表，所以可以使用任何一种表的形式来实现它，我们最常使用的一般有两种：

1. 链表
2. 数组

它们在复杂度上的优缺点对比如下：

1. 新增和删除元素时的时间复杂度
    - 链表：在动态申请内存（new或者malloc）上的花销非常昂贵。
    - 数组：几乎没有花销，以常数O(1)时间运行，在带有自增和自减寻址功能的寄存器上操作时，编译器会把整数的push和pop操作编译成一条机器指令。
2. 空间复杂度
    - 链表：由于空间是动态申请、释放的，因此不会浪费空间，而且只要物理存储器允许，理论上能够满足最大范围未知的情况。
    - 数组：必须在初始化时指定栈的大小，有可能会浪费空间，也有可能不够空间用。

结论：

1. 如果对运行时的效率要求非常高，并且能够在初始化时预知栈的大小，那么应该首选数组形式；否则就应该选用链表形式。
2. 由于对栈的操作永远都是针对栈顶（top）进行的，因此数组的随机存取的优点就没有了，而且数组必须预先分配空间，空间大小也受到限制，所以一般情况下（对运行时效率的要求不是太高）链表应该是首选。

5.2 代码实现

栈的实现如下：

```
////////////////////////////////////////////////////////////////////////
//
// FileName    :  stack.h
```

```
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-6 11:42:17
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#ifndef __STACK_H__
#define __STACK_H__

#include "../../slist/src/slist.h"

template<typename T>
class CStack : public CSList<T>
{
public:
    int push(T data);
    int pop(T *data = NULL);
    int top(T *data) const;
};

template<typename T>
inline int CStack<T>::push(T data)
{
    return AddTail(data);
}

template<typename T>
inline int CStack<T>::pop(T *data)
{
    if (IsEmpty())
        return 0;

    if (data)
        top(data);

    RemoveTail();
    return 1;
}

template<typename T>
inline int CStack<T>::top(T *data) const
{
    ASSERT(data);

    if (IsEmpty())
        return 0;

    *data = GetTail();
    return 1;
}

#endif  // __STACK_H__
```

调用如下：

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  stack.cpp
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-6 11:42:28
// Comment     :
//
////////////////////////////////////////////////////////////////////////////
```

```
#include <iostream>
#include "stack.h"
using namespace std;

static void PrintValue(const int nRetCode, const int nValue)
{
    if (nRetCode)
        cout << nValue << endl;
    else
        cout << "Error occured!" << endl;
}

int main()
{
    CStack<int> stack;
    int nValue;
    int nRetCode;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    stack.push(1);
    stack.push(2);
    stack.push(3);

    nRetCode = stack.top(&nValue);
    PrintValue(nRetCode, nValue);

    nRetCode = stack.pop(&nValue);
    PrintValue(nRetCode, nValue);

    nRetCode = stack.pop(&nValue);
    PrintValue(nRetCode, nValue);

    nRetCode = stack.pop(&nValue);
    PrintValue(nRetCode, nValue);
}
```

5.3 说明

上面的代码就是在单链表的基础上实现的栈，您会看到，在C++的继承机制下，栈的实现简单得可怕。 :)

一个影响栈的运行效率的问题是错误检测。我的栈实现中是仔细地检查了错误的——对空栈进行top和pop操作，以及当存储空间不够时进行push操作是会引起异常的，显然，我们不愿意出现这种情况，但是，如果把对这些条件的检测放到代码中，那就很可能要花费像实际栈操作那样多的时间。由于这个原因，除非在错误处理极其重要的场合（例如在操作系统中），一般在栈中省去错误检测就成了普通的惯用手法。

但我认为，一个良好的程序首先应该是健壮的，这比效率还要重要，特别是对于栈这种最基本的数据结构，它很可能会被作为基本的元素而被别的地方大量地使用。所以我并没有因为效率的问题而省去了错误检查机制。

引入错误检查机制的代价是：

1. 对top和pop的操作变得有些繁琐。在代码中我是使用了返回值0或者1来表示成功或者失败，而实际的栈顶元素是通过参数来返回的。这样做必定会有人不满——太麻烦了！但这是我能想到的最好的解决方法，如果你有更好的方法，请告诉我。

2. 运行时效率会降低。如果确实耗费了太多的时间，你可以把错误检查去掉，但前提条件是你能确保整个运行过程中不会出错——其实还是要有错误检查的，只不过这些错误检查会放在外围来做而已。

好了，就说那么多，下面我们来看看栈的应用。

5.4 应用：中缀到后缀表达式的转换

对栈的应用实在是太广泛了（谁让栈是最基本的数据结构元素之一呢？），例如有平衡符号、表达式转换之类的，我们在这里就选择一个比较有实用价值的例子——中缀到后缀表达式的转换。（可以用在编译器等地方）

5.4.1 代码实现

```
//////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   postfix.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-6 16:00:54
//  Comment     :
//
//////////////////////////////////////////////////////////////////////////////

// 算法：
// 1)检查输入的下一元素。
// 2)假如是个操作数，输出。
// 3)假如是个开括号，将其压栈。
// 4)假如是个运算符，则
//   i) 假如栈为空，将此运算符压栈。
//   ii) 假如栈顶是开括号，将此运算符压栈。
//   iii) 假如此运算符比栈顶运算符优先级高，将此运算符压入栈中。
//   iv) 否则栈顶运算符出栈并输出，重复步骤4。
// 5)假如是个闭括号，栈中运算符逐个出栈并输出，直到遇到开括号。开括号出栈并丢弃。
// 6)假如输入还未完毕，跳转到步骤1。
// 7)假如输入完毕，栈中剩余的所有操作符出栈并输出它们。

#include <stdio.h>
#include "stack.h"

// 返回操作符的优先级
// +和-的优先级是一样的，*和/的优先级也是一样的，但+和-的优先级要比*和/的低。
static int GetPRI(const char optr)
{
    switch (optr)
    {
    case '+': return 1;
    case '-': return 1;
    case '*': return 2;
    case '/': return 2;
    default : return 0;
    }
}

// 在这个函数中完成对栈顶的操作符和当前操作符的优先级对比，
// 并决定是输出当前的操作符还是对当前的操作符进行入栈处理。
static void ProcessStackPRI(
    CStack<char> &stack,
    const char optr,
    char **szPostfix
)
{
    ASSERT(*szPostfix);

    int i;
    int nRetCode;
    char chStackOptr;
```

```
    int nCount = stack.GetCount();

    for (i = 0; i <= nCount; ++i)
    {
        nRetCode = stack.top(&chStackOptr);
        if (
            (0 == nRetCode) ||                  // 栈顶为空，新操作符添加到栈顶
            (GetPRI(chStackOptr) < GetPRI(optr))// 栈顶操作符优先级比当前的要低
        )
        {
            stack.push(optr);
            break;
        }
        else
        {
            // 如果栈顶操作符优先级不低于当前的，则栈顶元素出栈并输出：
            stack.pop();
            *(*szPostfix)++ = chStackOptr;
        }
    }
}

static void Infix2Postfix(
    const char *szInfix,
    char *szPostfix
)
{
    ASSERT(szPostfix);

    char chOptr;
    int nRetCode;
    CStack<char> stack;

    while (*szInfix)
    {
        switch (*szInfix)
        {
        // 忽略空格和TAB：
        case ' ':
        case '\t':
            break;

        // 对操作符进行优先级判断，以便决定是入栈还是输出：
        case '+':
        case '-':
        case '*':
        case '/':
            nRetCode = stack.IsEmpty();
            if (!nRetCode)
                ProcessStackPRI(stack, *szInfix, &szPostfix);
            else
                stack.push(*szInfix);   // 当栈为空时，毫无疑问操作符应该入栈
            break;

        // 遇到左括号时，无条件入栈，因为它的优先级是最高的
        case '(':
            stack.push(*szInfix);
            break;

        // 遇到右括号时，逐个把栈中的操作符出栈，直到遇到左括号为止
        case ')':
            do
            {
                nRetCode = stack.pop(&chOptr);
                if (nRetCode && ('(' != chOptr))            // 左括号本身不输出
                    *szPostfix++ = chOptr;
            } while (!stack.IsEmpty() && ('(' != chOptr));  // 遇到左括号为止
```

```
            break;

        // 其余的情况，直接输出即可
        default:
            *szPostfix++ = *szInfix;
            break;
        }
        ++szInfix;
    }
    // 如果输入的内容已经分析完毕，那么就把栈中剩余的操作符全部出栈
    while (!stack.IsEmpty())
    {
        nRetCode = stack.pop(&chOptr);
        *szPostfix++ = chOptr;
    }
    *szPostfix = '\0';
}

int main()
{
    char *szInfix = "a+b*c+(d*e+f)*g";
    char szPostfix[255];

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    Infix2Postfix(szInfix, szPostfix);

    printf("Infix   : %s\n", szInfix);
    printf("Postfix : %s\n", szPostfix);
}
```

5.4.2 说明

源代码里面已经有了详细的注释，我就不再罗嗦了。我只做了+、-、*、/四种操作符的转换，另外，如果括号不匹配，例如有左括号但是没有右括号，或者反过来，程序就可能会运行不正确，但这不是我写这个例子的重点，我写它只是为了掌握栈的用法，如果您有兴趣，可以试着完善它。

下面给出两个例子：

```
中缀表达式：a + b * c + (d * e + f) * g
后缀表达式：abc*+de*f+g*+
```

```
中缀表达式：2 * (x + y) / (1 - x)
后缀表达式：2xy+*1x-/
```

第六章

队列

6.1 基本概念

像栈一样，队列（queue）也是表。然而，使用队列时插入在一端进行而删除则在另一端进行，也就是先进先出（FIFO）。队列的基本操作是EnQueue（入队），它是在表的末端（叫做队尾（rear））插入一个元素；还有DeQueue（出队），它是删除（或返回）在表的开头（叫做队头（front））的元素。

队列一般有链式队列和循环队列两种。链式队列相当于我们在银行中排队，后来的人排到队伍的最后，前面的人办理完业务后就会离开，让下一个人进去；循环队列则跟循环链表很相似。

我在此只写出链式队列的代码，循环队列其实也可以继承自循环链表，就不多罗嗦了。可以看到，队列的实现也是惊人的简单。

6.2 代码实现

队列的实现如下：

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  lqueue.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-8 16:49:54
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#ifndef __LIST_QUEUE_H__
#define __LIST_QUEUE_H__

#include "../../slist/src/slist.h"

template<typename T>
class CLQueue : public CSList<T>
{
public:
    int EnQueue(const T data);
    T   DeQueue();
    T&  GetFront();
    T   GetFront() const;
    T&  GetRear();
    T   GetRear() const;
};

template<typename T>
inline int CLQueue<T>::EnQueue(const T data)
```

```
{
    return AddTail(data);
}

template<typename T>
inline T CLQueue<T>::DeQueue()
{
    T data = GetHead();
    RemoveHead();
    return data;
}

template<typename T>
inline T& CLQueue<T>::GetFront()
{
    return GetHead();
}

template<typename T>
inline T CLQueue<T>::GetFront() const
{
    return GetHead();
}

template<typename T>
inline T& CLQueue<T>::GetRear()
{
    return GetTail();
}

template<typename T>
inline T CLQueue<T>::GetRear() const
{
    return GetTail();
}

#endif  // __LIST_QUEUE_H__
```

调用如下：

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   queue.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-8 17:00:40
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include "lqueue.h"
using namespace std;

int main()
{
    CLQueue<int> queue;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    queue.EnQueue(1);
    queue.EnQueue(2);
    queue.EnQueue(3);
```

```
    while (!queue.IsEmpty())
        cout << queue.DeQueue() << endl;
}
```

6.3 应用

队列的应用一般来说是模拟现实生活中的一些离散现象，例如银行排队、打印机任务、接线员工作等等。还有的就是使用队列来提高运行效率的算法，这些一般是在图算法中使用到。考虑到队列的应用要么是比较简单，要么是在特定的环境中进行，因此我就不给出应用的例子了，如果您有兴趣的话可以自行试试。

[*首页*, *上一页*, *下一页*; *目录*]

第七章

递归

7.1 基本概念

按照原书的流程，现在应该讲到递归了。递归是一种有力的数学工具。不知道各位学过Lisp或者它的方言没有（例如Scheme），如果学过的话，一定会对递归非常熟悉，因为在Lisp和它的方言中，是没有循环语句的，如果您要构造一个循环，必须通过递归的形式来实现。当时我的脑袋怎么也转不过弯来，因为我已经习惯了在C/C++里面使用for、while等语句来循环了，在Lisp里面刚开始我几乎没有办法写出一个不出错的循环来。

例如，下面的代码：

```
for (int i = 0; i <= 10; ++i)
{
}
```

可以被转换成递归：

```
void recursion_loop(int i)
{
    if (i == 10)
        return;
    else
        recursion_loop(i + 1);
}

// 调用 :
recursion_loop(0);
```

递归具有以下的性质：

1. 递归就是在某个过程中重复调用它本身。例如在上面的例子中，就是在recursion_loop()这个函数中再调用它本身。

2. 必须有停止条件。这很容易理解，因为如果没有停止条件的话，那么这个递归就会子子孙孙无穷溃也。例如在上面的例子中，if (i == 10) 就是停止的条件。

3. 递归会受到现实中的限制，例如栈的大小不够而导致失败。这是因为在计算机中，栈的大小是有上限的，而每次递归调用函数本身，都需要在栈中保存返回地址、参数等信息，在经过N次递归之后，栈很可能就会满了，这样就会导致无法进行第(N+1)次递归。

根据上面的性质3我们可以知道，并不是所有的语言都支持递归的——如果某种语言能够支持递归，那么它必须是支持"栈"这种结构的。目前就我所知道的对递归的使用发挥得最淋漓尽致的语言，Lisp和它的方言是当之无愧的王者。

7.2 应用

唉，本来都不想写递归的例子了，因为这些例子已经被写过无数次。提到递归，就一定会说到阶乘、斐波那契数列和汉诺塔这三个例子，但本着把教科书过一遍的目的，我还是再进行一次重复劳动吧（但不再对这三个例子进行讲解了，随便找一本数据结构的书都会有这方面的内容）。最后增加一个帕斯卡三角形，在我国也就是著名的杨辉三角。

7.2.1 阶乘

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  factorial.c
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-8 21:23:16
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>

static long factorial(const long n)
{
    return 0 == n || 1 == n ? 1  : n * factorial(n - 1);
}

int main()
{
    long lResult = factorial(10);
    printf("%ld\n", lResult);
}
```

7.2.2 斐波那契数列

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  fib.c
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-8 21:28:56
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>

static long fib(const long n)
{
    return 0 == n || 1 == n ? 1 : fib(n - 1) + fib(n - 2);
}

int main()
{
    long lResult = fib(10);
    printf("%ld\n", lResult);
}
```

7.2.3 汉诺塔

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  hanoi.c
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-8 21:40:44
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>

static void move(const char x, const int n, const char z)
{
    printf("把圆盘 %d 从柱子 %c 移动到 %c 上\n", n, x, z);
}

static void hanoi(const int n, const char x, const char y, const char z)
{
    if (1 == n)
        move(x, 1, z);          // 如果只有一个盘，则直接将它从x移动到z
    else
    {
        hanoi(n - 1, x, z, y);  // 把1 ~ n - 1个盘从x移动到y，用z作为中转
        move(x, n, z);          // 把第n个盘从x移动到z
        hanoi(n - 1, y, x, z);  // 把1 ~ n - 1个盘从y移动到z，用x作为中转
    }
}

int main()
{
    hanoi(1, 'X', 'Y', 'Z');
}
```

7.2.4 帕斯卡三角形（杨辉三角）

下面的数值被称为帕斯卡三角形，在我国则是著名的杨辉三角：

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
```

三角形边界上的数都是1，内部的每个数是位于它上面的两个数之和。

利用递归我们可以很容易地把问题转换为这个性质：

假设f(row, col)表示杨辉三角的第row行的第col个元素，那么：

1. f(row, col) = 1 (col = 1 或者 row = col)，也就是递归的停止条件。
2. f(row, col) = f(row - 1, col - 1) + f(row - 1, col)，也就是上一行的两个相邻元素的和。

有了这个性质，我们的递归程序就容易写了。^_^

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  pascaltriangle.c
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-9 14:53:57
```

```
// Comment    :
//
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>

static long GetElement(const long row, const long col)
{
    // 每行的外围两个元素为1
    if ((1 == col) || (row == col))
        return 1;
    else
        // 其余的部分为上一行的(col - 1)和(col)元素之和
        return GetElement(row - 1, col - 1) + GetElement(row - 1, col);
}

static long PascalTriangle(const long n)
{
    int row;
    int col;

    for (row = 1; row <= n; ++row)
    {
        for (col = 1; col <= row; ++col)
            printf(" %4ld", GetElement(row, col));
        printf("\n");
    }
}

int main()
{
    PascalTriangle(5);
}
```

[首页, 上一页, 下一页; 目录]

第八章

二叉树

8.1 基本概念

树是一种非线性的数据结构，它在客观世界中广泛存在，例如人类社会的族谱和各种社会组织机构都可以用树来表示。我们最常用到的是树和二叉树，其中又以二叉树更为实用。为什么这样说呢？因为大部分的操作都可以转变为一个父亲、一个左儿子和一个右儿子来实现，而且对二叉树的操作更为简单。

8.2 代码实现

二叉树的代码实现如下：

```
//////////////////////////////////////////////////////////////////////////
//
// FileName    :  btree.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-12 12:22:40
// Comment     :
//
//////////////////////////////////////////////////////////////////////////

#ifndef __BINARY_TREE_H__
#define __BINARY_TREE_H__

#include <assert.h>
#include <crtdbg.h>

#ifdef _DEBUG
#define DEBUG_NEW new (_NORMAL_BLOCK, THIS_FILE, __LINE__)
#endif

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#ifdef _DEBUG
#ifndef ASSERT
#define ASSERT  assert
#endif
#else   // not _DEBUG
#ifndef ASSERT
#define ASSERT
#endif
#endif  // _DEBUG
```

```cpp
template<typename T>
class CBTNode
{
public:
    T data;
    CBTNode<T> *parent;
    CBTNode<T> *left;
    CBTNode<T> *right;
    CBTNode(
        T data = T(),
        CBTNode<T> *parent = NULL,
        CBTNode<T> *left = NULL,
        CBTNode<T> *right = NULL
    ) : data(data), parent(parent), left(left), right(right) {}
};

template<typename T>
class CBTree
{
protected:
    CBTNode<T> *m_pNodeRoot;

public:
    CBTree(CBTNode<T> *initroot = NULL);
    ~CBTree();
    void AssignTo(CBTNode<T> *p);
    void Copy(CBTree<T> &p);

private:
    CBTNode<T>* Copy(CBTNode<T> *p);

    void DestroyNode(CBTNode<T> *p);

    void PreOrderTraverse(
        const CBTNode<T> *p,
        void (*Visit)(const T &data)
    ) const;

    void InOrderTraverse(
        const CBTNode<T> *p,
        void (*Visit)(const T &data)
    ) const;

    void PostOrderTraverse(
        const CBTNode<T> *p,
        void (*Visit)(const T &data)
    ) const;

    void GetNodesCount(const CBTNode<T> *p, unsigned int *unCount) const;

    void GetLeafCount(const CBTNode<T> *p, unsigned int *unCount) const;

public:
    T&            GetNodeData(CBTNode<T> *p);
    T             GetNodeData(const CBTNode<T> *p) const;
    void          SetNodeData(CBTNode<T> *p, const T &data);
    CBTNode<T>*&   GetRoot();
    CBTNode<T>*    GetRoot() const;
    CBTNode<T>*&   GetParent(CBTNode<T> *p);
    CBTNode<T>*    GetParent(const CBTNode<T> *p) const;
    CBTNode<T>*&   GetLeftChild(CBTNode<T> *p);
    CBTNode<T>*    GetLeftChild(const CBTNode<T> *p) const;
    CBTNode<T>*&   GetRightChild(CBTNode<T> *p);
    CBTNode<T>*    GetRightChild(const CBTNode<T> *p) const;
    CBTNode<T>*&   GetLeftSibling(CBTNode<T> *p);
    CBTNode<T>*    GetLeftSiblig(const CBTNode<T> *p) const;
    CBTNode<T>*&   GetRightSibling(CBTNode<T> *p);
```

```
    CBTNode<T>*     GetRightSibling(const CBTNode<T> *p) const;

public:
    int             IsEmpty() const;
    void            Destroy();
    void            PreOrderTraverse(void (*Visit)(const T &data)) const;
    void            InOrderTraverse(void (*Visit)(const T &data)) const;
    void            PostOrderTraverse(void (*Visit)(const T &data)) const;
    unsigned int    GetNodesCount() const; // Get how many nodes
    unsigned int    GetLeafCount() const;
    unsigned int    GetDepth() const;
    unsigned int    GetDepth(const CBTNode<T> *p) const;
};

template<typename T>
inline CBTree<T>::CBTree(CBTNode<T> *initroot) : m_pNodeRoot(initroot)
{
}

template<typename T>
inline CBTree<T>::~CBTree()
{
    Destroy();
}

template<typename T>
inline void CBTree<T>::AssignTo(CBTNode<T> *p)
{
    ASSERT(p);
    m_pNodeRoot = p;
}

template<typename T>
inline void CBTree<T>::Copy(CBTree<T> &p)
{
    if (NULL != p.m_pNodeRoot)
        m_pNodeRoot = Copy(p.m_pNodeRoot);
    else
        m_pNodeRoot = NULL;
}

template<typename T>
inline CBTNode<T>* CBTree<T>::Copy(CBTNode<T> *p)
{
    if (p)
    {
        CBTNode<T> *pNewNode = new CBTNode<T>;
        if (NULL == pNewNode)
            return NULL;
        pNewNode->data = p->data;
        pNewNode->parent = p->parent;
        pNewNode->left = Copy(p->left);
        pNewNode->right = Copy(p->right);
        return pNewNode;
    }
    else
        return NULL;
}

template<typename T>
inline CBTNode<T>*& CBTree<T>::GetLeftChild(CBTNode<T> *p)
{
    ASSERT(p);
    return *(&(p->left));
}

template<typename T>
```

```
inline CBTNode<T>* CBTree<T>::GetLeftChild(const CBTNode<T> *p) const
{
    ASSERT(p);
    return p->left;
}

template<typename T>
inline CBTNode<T>*& CBTree<T>::GetRightChild(CBTNode<T> *p)
{
    ASSERT(p);
    return *(&(p->right));
}

template<typename T>
inline CBTNode<T>* CBTree<T>::GetRightChild(const CBTNode<T> *p) const
{
    ASSERT(p);
    return p->right;
}

template<typename T>
inline CBTNode<T>*& CBTree<T>::GetLeftSibling(CBTNode<T> *p)
{
    ASSERT(p);

    if (p->parent)
        return *(&(p->parent->left));
    else
        return *(&(p->parent)); // return NULL;
}

template<typename T>
inline CBTNode<T>* CBTree<T>::GetLeftSiblig(const CBTNode<T> *p) const
{
    ASSERT(p);

    if (p->parent)
        return p->parent->left;
    else
        return p->parent;       // return NULL;
}

template<typename T>
inline CBTNode<T>*& CBTree<T>::GetRightSibling(CBTNode<T> *p)
{
    ASSERT(p);

    if (p->parent)
        return *(&(p->parent->right));
    else
        return *(&(p->parent)); // return NULL;
}

template<typename T>
inline CBTNode<T>* CBTree<T>::GetRightSibling(const CBTNode<T> *p) const
{
    ASSERT(p);

    if (p->parent)
        return p->parent->right;
    else
        return p->parent;       // return NULL;
}

template<typename T>
inline CBTNode<T>*& CBTree<T>::GetParent(CBTNode<T> *p)
{
```

```
    ASSERT(p);
    return *(&(p->parent));
}

template<typename T>
inline CBTNode<T>* CBTree<T>::GetParent(const CBTNode<T> *p) const
{
    ASSERT(p);
    return p->parent;
}

template<typename T>
inline T& CBTree<T>::GetNodeData(CBTNode<T> *p)
{
    ASSERT(p);
    return p->data;
}

template<typename T>
inline T CBTree<T>::GetNodeData(const CBTNode<T> *p) const
{
    ASSERT(p);
    return p->data;
}

template<typename T>
inline void CBTree<T>::SetNodeData(CBTNode<T> *p, const T &data)
{
    ASSERT(p);
    p->data = data;
}

template<typename T>
inline int CBTree<T>::IsEmpty() const
{
    return NULL == m_pNodeRoot;
}

template<typename T>
inline CBTNode<T>*& CBTree<T>::GetRoot()
{
    return *(&(m_pNodeRoot));
}

template<typename T>
inline CBTNode<T>* CBTree<T>::GetRoot() const
{
    return m_pNodeRoot;
}

template<typename T>
inline void CBTree<T>::DestroyNode(CBTNode<T> *p)
{
    if (p)
    {
        DestroyNode(p->left);
        DestroyNode(p->right);
        delete p;
    }
}

template<typename T>
inline void CBTree<T>::Destroy()
{
    DestroyNode(m_pNodeRoot);
    m_pNodeRoot = NULL;
}
```

```cpp
template<typename T>
inline void CBTree<T>::PreOrderTraverse(void (*Visit)(const T &data)) const
{
    PreOrderTraverse(m_pNodeRoot, Visit);
}

template<typename T>
inline void CBTree<T>::PreOrderTraverse(
    const CBTNode<T> *p,
    void (*Visit)(const T &data)
) const
{
    if (p)
    {
        Visit(p->data);
        PreOrderTraverse(p->left, Visit);
        PreOrderTraverse(p->right, Visit);
    }
}

template<typename T>
inline void CBTree<T>::InOrderTraverse(void (*Visit)(const T &data)) const
{
    InOrderTraverse(m_pNodeRoot, Visit);
}

template<typename T>
inline void CBTree<T>::InOrderTraverse(
    const CBTNode<T> *p,
    void (*Visit)(const T &data)
) const
{
    if (p)
    {
        InOrderTraverse(p->left, Visit);
        Visit(p->data);
        InOrderTraverse(p->right, Visit);
    }
}

template<typename T>
inline void CBTree<T>::PostOrderTraverse(void (*Visit)(const T &data)) const
{
    PostOrderTraverse(m_pNodeRoot, Visit);
}

template<typename T>
inline void CBTree<T>::PostOrderTraverse(
    const CBTNode<T> *p,
    void (*Visit)(const T &data)
) const
{
    if (p)
    {
        PostOrderTraverse(p->left, Visit);
        PostOrderTraverse(p->right, Visit);
        Visit(p->data);
    }
}

template<typename T>
inline unsigned int CBTree<T>::GetNodesCount() const
{
    unsigned int unCount;
    GetNodesCount(m_pNodeRoot, &unCount);
    return unCount;
```

```
}

template<typename T>
inline void CBTree<T>::GetNodesCount(
    const CBTNode<T> *p,
    unsigned int *unCount
) const
{
    ASSERT(unCount);

    unsigned int unLeftCount;
    unsigned int unRightCount;

    if (NULL == p)
        *unCount = 0;
    else if ((NULL == p->left) && (NULL == p->right))
        *unCount = 1;
    else
    {
        GetNodesCount(p->left, &unLeftCount);
        GetNodesCount(p->right, &unRightCount);
        *unCount = 1 + unLeftCount + unRightCount;
    }
}

template<typename T>
inline unsigned int CBTree<T>::GetLeafCount() const
{
    unsigned int unCount = 0;
    GetLeafCount(m_pNodeRoot, &unCount);
    return unCount;
}

template<typename T>
inline void CBTree<T>::GetLeafCount(
    const CBTNode<T> *p,
    unsigned int *unCount
) const
{
    ASSERT(unCount);

    if (p)
    {
        // if the node's left & right children are both NULL, it must be a leaf
        if ((NULL == p->left) && (NULL == p->right))
            ++(*unCount);
        GetLeafCount(p->left, unCount);
        GetLeafCount(p->right, unCount);
    }
}

template<typename T>
inline unsigned int CBTree<T>::GetDepth() const
{
    // Minus 1 here because I think the root node's depth should be 0.
    // So, don't do it if u think the root node's depth should be 1.
    return GetDepth(m_pNodeRoot) - 1;
}

template<typename T>
inline unsigned int CBTree<T>::GetDepth(const CBTNode<T> *p) const
{
    unsigned int unDepthLeft;
    unsigned int unDepthRight;

    if (p)
    {
```

```
        unDepthLeft = GetDepth(p->left);
        unDepthRight = GetDepth(p->right);
        return 1 +  // if don't plus 1 here, the tree's depth will be always 0
            (unDepthLeft > unDepthRight ? unDepthLeft : unDepthRight);
    }
    else
        return 0;
}

#endif  // __BINARY_TREE_H__
```

测试代码：

```
//////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   btree.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-12 13:17:07
//  Comment     :
//
//////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include "btree.h"
using namespace std;

// 结点的数据类型
typedef char ElementType;

// 回调函数 : Visit() = PrintElement()
static void PrintElement(const ElementType &data)
{
    cout << data;
}

int main()
{
    CBTNode<ElementType> *pRoot;
    CBTNode<ElementType> *pLeftChild;
    CBTNode<ElementType> *pRightChild;
    CBTree<ElementType> btree;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    pRoot = new CBTNode<ElementType>;
    if (NULL == pRoot)
        return EXIT_FAILURE;

    pLeftChild = new CBTNode<ElementType>;
    if (NULL == pLeftChild)
        return EXIT_FAILURE;

    pRightChild = new CBTNode<ElementType>;
    if (NULL == pRightChild)
        return EXIT_FAILURE;

    // 创建父亲结点
    pRoot->data = '+';
    pRoot->parent = NULL;
    pRoot->left = pLeftChild;
    pRoot->right = pRightChild;

    // 创建左儿子结点
```

```
    pLeftChild->data = 'a';
    pLeftChild->parent = pRoot;
    pLeftChild->left = NULL;
    pLeftChild->right = NULL;

    // 创建右儿子结点
    pRightChild->data = 'b';
    pRightChild->parent = pRoot;
    pRightChild->left = NULL;
    pRightChild->right = NULL;

    // 创建二叉树
    btree.AssignTo(pRoot);

    // 输出这棵二叉树
    cout << "   (" << btree.GetNodeData(btree.GetRoot()) << ")  " << endl;
    cout << "  /   \\ " << endl;
    cout << "(" << btree.GetNodeData(btree.GetLeftChild(btree.GetRoot()))
        << ")    (" << btree.GetNodeData(btree.GetRightChild(btree.GetRoot()))
        << ")" << endl << endl;

    cout << "这棵树的叶子数：" << btree.GetLeafCount() << endl;

    cout << "这棵树的深度是：" << btree.GetDepth() << endl;

    cout << "先序遍历：";
    btree.PreOrderTraverse(PrintElement);

    cout << endl << "中序遍历：";
    btree.InOrderTraverse(PrintElement);

    cout << endl << "后序遍历：";
    btree.PostOrderTraverse(PrintElement);

    cout << endl;

    return EXIT_SUCCESS;
}
```

8.3 说明

您也许已经注意到了一个"奇怪"的现象：在我的二叉树实现中，有各种对结点的访问操作（例如计算树的高、各种遍历），但就是没有插入和删除这两个操作的函数。其实这并不值得奇怪。因为二叉树基本上是一个最"底层"的类，将来我们在写二叉搜索树等更高级的类时，是要从二叉树开始继承的，而对于树这种非线性的数据结构来说，插入和删除是要根据它所处的环境来具体问题具体分析的——也就是说，没有一个特定的法则（这点不像链表，链表无论怎么变，它都是线性的）。所以，在具体的应用中，我才会给出具体的插入和删除代码。在这里，我用了一种很拙劣的方式来创建了一棵二叉树，请读者在这个问题上不要深究。

在结点类CBTNode中，我定义了4个成员变量：data、parent、left和right。data表示该结点的数据域，parent表示该结点的父亲结点，left和right分别表示该结点的左右儿子结点。这里要说明的是：

1. parent指针并不是必需的，但有了它之后，就会大大简化许多对父亲结点的操作。因此，在资源并不十分紧张的情况下应该考虑加入它。

2. 二叉树的根结点（root）的parent应该赋值为NULL。

在二叉树中还大量运用了前面所说的一个强大的工具——递归。例如对二叉树的遍历操作就都是通过递归来实现的（不递归也行，可以用栈来模拟，但速度会比较慢，同时也多占用了很多空间，也就是说，非递归的算法无论是时间复杂度还是空间复杂度都比递归要高——非递归的唯一好处只是节省了堆栈。因此到底选用哪个，就要看具体的应用环境

了）。另外，我在先序、中序和后序遍历中用了Visit()这个回调函数，这是为了增加处理的自由度。除此之外，我还写了几个要使用到遍历技术的子函数，如：GetLeafCount()，就是用先序遍历来获得二叉树的叶子个数。在此不一一而足，如有不清楚的地方，请联系我。

8.4 应用

基本的二叉树还谈不上有什么应用，因此我的示例程序只是做了一个对表达式的转换……您是不是想说，对表达式的转换不是在栈那里已经做过了吗？

是的！但实际上二叉树这种数据结构才是对表达式的最直观的储存和表达方式，甚至可以说，它天生就是一棵表达式！我的例子代码是用一棵二叉树来表示一个表达式：a + b，执行完后，会得到这样的输出结果：

```
  (+)
 /   \
(a)   (b)

这棵树的叶子数：2
这棵树的深度是：1
先序遍历：+ab
中序遍历：a+b
后序遍历：ab+
```

[首页, 上一页, 下一页; 目录]

第九章

二叉搜索树

9.1 基本概念

二叉树的一个重要的应用是它们在查找中的使用。二叉搜索树的概念相当容易理解，即：对于树中的每个结点X，它的左子树中所有关键字的值都小于X的关键字值，而它的右子树中的所有关键字值都大于X的关键字值。这意味着该树所有的元素都可以用某种统一的方式排序。

例如下面就是一棵合法的二叉搜索树：

```
   6
  /\
 2  8
/\
1  4
  /
 3
```

二叉搜索树的性质决定了它在搜索方面有着非常出色的表现：要找到一棵树的最小结点，只需要从根结点开始，只要有左儿子就向左进行，终止结点就是最小的结点。找最大的结点则是往右进行。例如上面的例子中，最小的结点是1，在最左边；最大的结点是8，在最右边。

9.2 代码实现

二叉树的代码实现如下：

```
//////////////////////////////////////////////////////////////////////////////
//
// FileName    :  bstree.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-17 22:53:52
// Comment     :
//
//////////////////////////////////////////////////////////////////////////////

#ifndef __BINARY_SEARCH_TREE_H__
#define __BINARY_SEARCH_TREE_H__

#include "../../btree/src/btree.h"

template<typename T>
class CBSTree : public CBTree<T>
{
private:
    CBTNode<T>* Find(const T &data, CBTNode<T> *p) const;
```

```
    CBTNode<T>* FindMin(CBTNode<T> *p) const;
    CBTNode<T>* FindMax(CBTNode<T> *p) const;
    CBTNode<T>* Insert(const T &data, CBTNode<T> *p);
    CBTNode<T>* Delete(const T &data, CBTNode<T> *p);

public:
    CBTNode<T>* Find(const T &data) const;
    CBTNode<T>* FindMin() const;
    CBTNode<T>* FindMax() const;
    CBTNode<T>* Insert(const T &data);
    CBTNode<T>* Delete(const T &data);
};

template<typename T>
inline CBTNode<T>* CBSTree<T>::Find(const T &data) const
{
    return Find(data, m_pNodeRoot);
}

template<typename T>
inline CBTNode<T>* CBSTree<T>::Find(const T &data, CBTNode<T> *p) const
{
    if (NULL == p)
        return NULL;
    if (data < p->data)
        return Find(data, p->left);
    else if (data > p->data)
        return Find(data, p->right);
    else
        return p;
}

template<typename T>
inline CBTNode<T>* CBSTree<T>::FindMin() const
{
    return FindMin(m_pNodeRoot);
}

template<typename T>
inline CBTNode<T>* CBSTree<T>::FindMin(CBTNode<T> *p) const
{
    if (NULL == p)
        return NULL;
    else if (NULL == p->left)
        return p;
    else
        return FindMin(p->left);
}

template<typename T>
inline CBTNode<T>* CBSTree<T>::FindMax() const
{
    return FindMax(m_pNodeRoot);
}

template<typename T>
inline CBTNode<T>* CBSTree<T>::FindMax(CBTNode<T> *p) const
{
    if (NULL == p)
        return NULL;
    else if (NULL == p->right)
        return p;
    else
        return FindMax(p->right);
}

template<typename T>
```

```
inline CBTNode<T>* CBSTree<T>::Insert(const T &data)
{
    return Insert(data, m_pNodeRoot);
}

template<typename T>
inline CBTNode<T>* CBSTree<T>::Insert(const T &data, CBTNode<T> *p)
{
    if (NULL == p)
    {
        p = new CBTNode<T>;
        if (NULL == p)
            return NULL;
        else
        {
            p->data = data;
            p->left = NULL;
            p->right = NULL;
            if (NULL == m_pNodeRoot)
            {
                m_pNodeRoot = p;
                m_pNodeRoot->parent = NULL;
            }
        }
    }
    else if (data < p->data)
    {
        p->left = Insert(data, p->left);
        if (p->left)
            p->left->parent = p;
    }
    else if (data > p->data)
    {
        p->right = Insert(data, p->right);
        if (p->right)
            p->right->parent = p;
    }
    // else data is in the tree already, we'll do nothing!

    return p;
}

template<typename T>
inline CBTNode<T>* CBSTree<T>::Delete(const T &data)
{
    return Delete(data, m_pNodeRoot);
}

template<typename T>
inline CBTNode<T>* CBSTree<T>::Delete(const T &data, CBTNode<T> *p)
{
    if (NULL == p)
    {
        // Error! data not found!
    }
    else if (data < p->data)
    {
        p->left = Delete(data, p->left);
    }
    else if (data > p->data)
    {
        p->right = Delete(data, p->right);
    }
    else if (p->left && p->right)   // found it, and it has two children
    {
        CBTNode<T> *pTmp = FindMin(p->right);
        p->data = pTmp->data;
```

```
        p->right = Delete(p->data, p->right);
    }
    else    // found it, and it has one or zero children
    {
        CBTNode<T> *pTmp = p;
        if (NULL == p->left)
            p = p->right;
        else if (NULL == p->right)
            p = p->left;

        if (p)
            p->parent = pTmp->parent;

        if (m_pNodeRoot == pTmp)
            m_pNodeRoot = p;

        delete pTmp;
    }

    return p;
}

#endif  // __BINARY_SEARCH_TREE_H__
```

测试代码：

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   bstree.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-17 22:55:12
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include "bstree.h"

int main()
{
    CBSTree<int> bstree;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    bstree.Insert(1);
    bstree.Insert(2);
    bstree.Insert(3);

    bstree.Delete(1);
}
```

9.3 说明

我的二叉搜索树是从二叉树继承而来的，我写了Find()、FindMin()、FindMax()、Insert()和Delete()一共5个成员函数。这里要说的是，对非线性数据结构的操作总是特别的不直观，因为一般来说我们会选择使用递归——而人脑一般不太容易"调试"递归的程序——如果递归的层数比较少（例如只有1、2次）那还好点，但一旦超过5、6次，恐怕人脑的"堆栈"就要溢出了。

好了，牢骚完毕，来解释一下：

1. Find()：如果树为空，则返回NULL；如果根结点比它的左儿子要小，就往左进行，否则如果比右儿子小就往右进行，一直到既不大于也不小于它的儿子为止，那么这个结点就一定是我们要找的了。

2. FindMin()：从根结点开始，只要有左儿子就向左进行，直到遇到终止结点为止。

3. FindMax()：除分支朝右儿子进行外，其余过程与FindMin()相同。

4. Insert()：如果找到了相同的元素，则什么都不做；否则，递归查找到遍历路径的最后一点上，然后执行Insert操作。

5. Delete()：正如许多数据结构一样，最困难的操作是删除。删除的操作可以分成下面几种情况：

    - 如果结点是一片树叶，那么它可以被立即删除。
    - 如果结点有一个儿子，那么该结点可以在其父结点调整指针绕过该结点后被删除。
    - 最复杂的情况是处理具有两个儿子的结点。我们可以用其右子树的最小的数据（很容易找到）代替该结点的数据，并递归地删除那个结点。为什么？因为一个结点肯定比它的右子树的所有结点都小，同时又比它的左子树的所有结点都大，所以我们只要在其右子树中找到最小的那个结点来代替它，就能满足二叉树的性质了。（根据这个规则，我们还可以用其左子树的最大的数据来代替该结点的数据，道理是一样的，不再叙述）

说了那么多，估计我还是没有讲清楚（主要是有点抽象），请读者编译我的代码并亲自动手调试一下吧。我的测试代码没有输出结果，因为要写个打印二叉树的函数我觉得有点烦，您可以在相应的函数中下断点，我个人认为只要能弄懂Delete()函数，那别的应该都没问题了。:)

[首页, 上一页, 下一页; 目录]

第十章

AVL树

10.1 基本概念

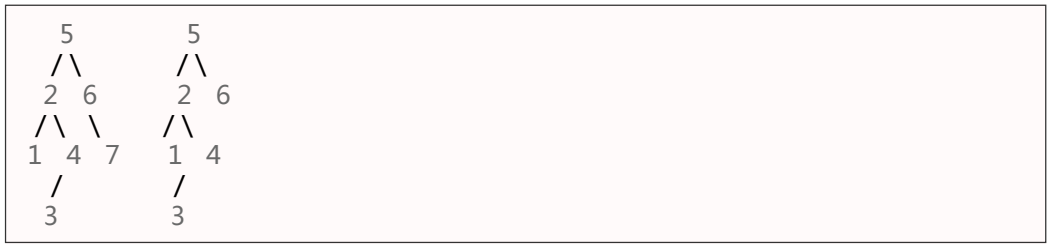
AVL树的复杂程度真是比二叉搜索树高了整整一个数量级——它的原理并不难弄懂，但要把它用代码实现出来还真的有点费脑筋。下面我们来看看：

10.1.1 AVL树是什么？

AVL树本质上还是一棵二叉搜索树（因此读者可以看到我后面的代码是继承自二叉搜索树的），它的特点是：

1. 本身首先是一棵二叉搜索树。
2. 带有平衡条件：每个结点的左右子树的高度之差的绝对值（平衡因子）最多为1。

例如：

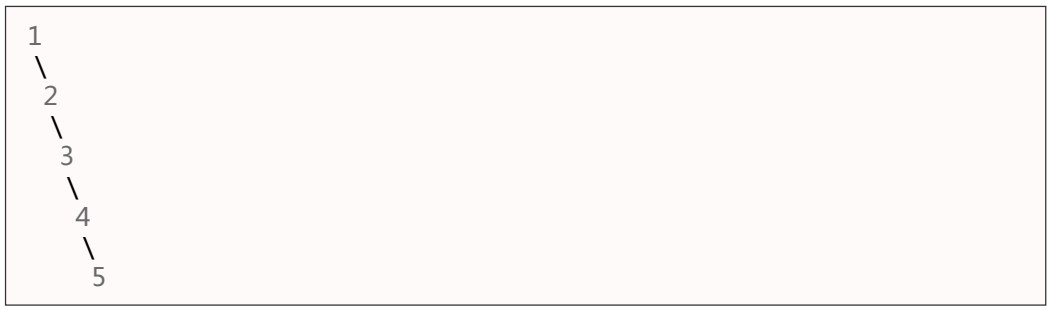


上图中，左边的是AVL树，而右边的不是。因为左边的树的每个结点的左右子树的高度之差的绝对值都最多为1，而右边的树由于结点6没有子树，导致根结点5的平衡因子为2。

10.1.2 为什么要用AVL树？

有人也许要问：为什么要有AVL树呢？它有什么作用呢？

我们先来看看二叉搜索树吧（因为AVL树本质上是一棵二叉搜索树），假设有这么一种极端的情况：二叉搜索树的结点为1、2、3、4、5，也就是：



聪明的你是不是发现什么了呢？呵呵，显而易见——这棵二叉搜索树其实等同于一个链表了，也就是说，它在查找上的优势已经全无了——在这种情况下，查找一个结点的时间复杂度是O(N)！

好，那么假如是AVL树（别忘了AVL树还是二叉搜索树），则会是：

```
 2
/\
1 4
 /\
 3 5
```

可以看出，AVL树的查找平均时间复杂度要比二叉搜索树低——它是O(logN)。也就是说，在大量的随机数据中AVL树的表现要好得多。

10.1.3 旋转

假设有一个结点的平衡因子为2（在AVL树中，最大就是2，因为结点是一个一个地插入到树中的，一旦出现不平衡的状态就会立即进行调整，因此平衡因子最大不可能超过2），那么就需要进行调整。由于任意一个结点最多只有两个儿子，所以当高度不平衡时，只可能是以下四种情况造成的：

1. 对该结点的左儿子的左子树进行了一次插入。
2. 对该结点的左儿子的右子树进行了一次插入。
3. 对该结点的右儿子的左子树进行了一次插入。
4. 对该结点的右儿子的右子树进行了一次插入。

情况1和4是关于该点的镜像对称，同样，情况2和3也是一对镜像对称。因此，理论上只有两种情况，当然了，从编程的角度来看还是四种情况。

第一种情况是插入发生在"外边"的情况（即左-左的情况或右-右的情况），该情况可以通过对树的一次单旋转来完成调整。第二种情况是插入发生在"内部"的情况（即左-右的情况或右-左的情况），该情况要通过稍微复杂些的双旋转来处理。

关于旋转的具体理论分析和例子请参阅教科书，我实在不想在这里重新打一次了.....就此省略65535个字，原谅我吧，出来混，迟早要还的。

10.2 代码实现

二叉树的代码实现如下：

```
//////////////////////////////////////////////////////////////////////////
//
// FileName    :  avltree.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-20 17:04:31
// Comment     :
//
//////////////////////////////////////////////////////////////////////////

#ifndef __AVL_TREE_H__
#define __AVL_TREE_H__

#include "../../bstree/src/bstree.h"

template<typename T>
class CAVLTree : public CBSTree<T>
```

```
{
private:
    CBTNode<T>* Insert(const T &data, CBTNode<T> *p);

public:
    CBTNode<T>* SingleRotateWithLeft(CBTNode<T> *p);
    CBTNode<T>* DoubleRotateWithLeft(CBTNode<T> *p);
    CBTNode<T>* SingleRotateWithRight(CBTNode<T> *p);
    CBTNode<T>* DoubleRotateWithRight(CBTNode<T> *p);
    CBTNode<T>* Insert(const T &data);
    CBTNode<T>* Delete(const T &data);
};

template<typename T>
inline CBTNode<T>* CAVLTree<T>::SingleRotateWithLeft(CBTNode<T> *p)
{
    CBTNode<T> *p2;

    // rotate
    p2 = p->left;
    p->left = p2->right;
    p2->right = p;

    // update parent relationship
    p2->parent = p->parent;
    p->parent = p2;
    if (p->left)
        p->left->parent = p;

    // update root node if necessary
    if (p == m_pNodeRoot)
        m_pNodeRoot = p2;

    return p2;  // New root
}

template<typename T>
inline CBTNode<T>* CAVLTree<T>::DoubleRotateWithLeft(CBTNode<T> *p)
{
    p->left = SingleRotateWithLeft(p->left);
    return SingleRotateWithLeft(p);
}

template<typename T>
inline CBTNode<T>* CAVLTree<T>::SingleRotateWithRight(CBTNode<T> *p)
{
    CBTNode<T> *p2;

    // rotate
    p2 = p->right;
    p->right = p2->left;
    p2->left = p;

    // update parent relationship
    p2->parent = p->parent;
    p->parent = p2;
    if (p->right)
        p->right->parent = p;

    // update root node if necessary
    if (p == m_pNodeRoot)
        m_pNodeRoot = p2;

    return p2;  // New root
}

template<typename T>
```

```
inline CBTNode<T>* CAVLTree<T>::DoubleRotateWithRight(CBTNode<T> *p)
{
    p->right = SingleRotateWithLeft(p->right);
    return SingleRotateWithRight(p);
}

template<typename T>
inline CBTNode<T>* CAVLTree<T>::Insert(const T &data)
{
    return Insert(data, m_pNodeRoot);
}

template<typename T>
inline CBTNode<T>* CAVLTree<T>::Insert(const T &data, CBTNode<T> *p)
{
    if (NULL == p)
    {
        // Create and return a one-node tree
        p = new CBTNode<T>;
        if (NULL == p)
            return NULL;
        else
        {
            p->data = data;
            p->left = NULL;
            p->right = NULL;
            if (NULL == m_pNodeRoot)
            {
                m_pNodeRoot = p;
                m_pNodeRoot->parent = NULL;
            }
        }
    }
    // left child
    else if (data < p->data)
    {
        p->left = Insert(data, p->left);
        if (p->left)
            p->left->parent = p;

        if (2 == (GetDepth(p->left) - GetDepth(p->right)))
        {
            // left tree, need to do single rotation
            if (data < p->left->data)
                p = SingleRotateWithLeft(p);
            // right tree, need to do double rotation
            else
                p = DoubleRotateWithLeft(p);
        }
    }
    // right child
    else if (data > p->data)
    {
        p->right = Insert(data, p->right);
        if (p->right)
            p->right->parent = p;

        if (2 == (GetDepth(p->right) - GetDepth(p->left)))
        {
            // right tree, need to do single rotation
            if (data > p->right->data)
                p = SingleRotateWithRight(p);
            // left tree, need to do double rotation
            else
                p = DoubleRotateWithRight(p);
        }
    }
```

```
    // else data is in the tree already, we'll do nothing!

    return p;
}

template<typename T>
inline CBTNode<T>* CAVLTree<T>::Delete(const T &data)
{
    // not completed yet.
    return NULL;
}

#endif  // __AVL_TREE_H__
```

测试代码：

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   avltree.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-20 17:06:50
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include "avltree.h"

int main()
{
    CAVLTree<int> avltree;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    avltree.Insert(1);
    avltree.Insert(2);
    avltree.Insert(3);
    avltree.Insert(4);
}
```

10.3 说明

我的AVL树是从二叉搜索树继承而来的（还记得我的二叉搜索树又是从二叉树继承来的吗？^_^）。另外，由于水平的关系，我没有写出Delete()函数来，因此如果您使用了Delete()，那是不会有效果的。-_-b...

插入的核心思路是通过递归（又是递归！）找到合适的位置，插入新结点，然后看新结点是否平衡（平衡因子是否为2），如果不平衡的话，就分成两种大情况以及两种小情况：

1. 在结点的左儿子(data < p->data)
    - 在左儿子的左子树((data < p->data) AND (data < p->left->data))，“外边”，要做单旋转。
    - 在左儿子的右子树((data < p->data) AND (data > p->left->data0))，“内部”，要做双旋转。
2. 在结点的右儿子(data > p->data)
    - 在右儿子的左子树((data > p->data) AND (data < p->right->data))，“内部”，要做双旋转。

- 在右儿子的右子树((data > p->data) AND (data > p->right->data))，“外边”，要做单旋转。

代码已经写得很清楚了，我就不多废话了，关键在于动手去调试，这样才能弄明白。值得说明的是，当进行了旋转之后，必定会有结点的“父结点”是需要更新的，例如：

```
 2
/\
1 4
 /\
 3 5
   \
    6
```

上图是调整前的，下图是调整后的：

```
  4
 /\
 2 5
/\ \
1 3 6
```

可以看出，根结点2不平衡，是由于它的右儿子的右子树插入了新的结点6造成的。因此，这属于“外边”的情况，要进行一次单旋转。于是我们就把结点4调整上来作为根结点，再把结点2作为4的左儿子，最后把结点2的右儿子修改为原来的结点4的左儿子。

调整后的parent指针变化规律如下：

1. 调整前的右儿子（调整后它就变为父亲了）的parent指针应该指向调整前的父亲（调整后它就变成左儿子了）的parent指针。
2. 调整前的父亲（调整后它就变成左儿子了）的parent指针应该指向调整前的右儿子（调整后它就变成父亲了）。
3. 调整前的父亲的右儿子的parent指针应该指向调整前的右儿子的左儿子。

很难理解是吗？我来联系到上图说明：

1. 调整前的右儿子是结点4，调整后，它的parent指针应该指向调整前它的父亲的parent指针，也就是NULL，因为调整前结点4的父亲是结点2，而结点2是根结点，其parent指针为NULL。
2. 调整前的父亲是结点2，调整后，它的parent指针应该指向调整前的右儿子（结点4）。
3. 调整前的父亲的右儿子（也就是调整后的结点2的右儿子）应该指向调整前的右儿子（结点4）的左儿子（结点3）。

这是SingleRotateWithRight()里面对parent指针的处理，SingleRotateWithLeft()里面的道理也是相通的，只是顺序有点不同。

呵呵，希望没把您弄晕。^_^

AVL树就讲到这里了，如果您有兴趣，可以把Delete()函数写完，并请给我一份以便我学习。

[首页, 上一页, 下一页; 目录]

第十一章

排序

11.1 基本概念

排序可以说是百花齐放，据算法宗师高德纳爷爷的《TAOCP》第三卷所记载的至少就有20多种。从大的方面来说，排序可以分成内排序和外排序——内排序是外排序的基础。我们常用的内排序又可以粗略分成下面的类型：

1. 插入排序
2. 交换排序
3. 堆排序
4. 归并排序

别看排序有那么多种类型，但它们都离不开这样的核心思想：

```
|有序序列区|无序序列区|
```

一个待排序列总是被不断从无序序列转变为有序序列。

从效率来说，目前已知最快的排序方法是"快速排序（QuickSort）"。牛B吧？呵呵，连名字都起得那么牛。（别的排序方法的名称要么是表示出它的本质（例如"插入排序"），要么是以其发明者命名的（例如"ShellSort"），只有QuickSort是直言不讳地用"Quick"来命名，这或许就是在排序上的最高荣誉吧！）

但要注意的是，没有一种排序方法的效率是在任何情况下都能独占鳌头的，具体采取哪种方法要根据实际情况而定（有的人喜欢用快速排序通吃各种情况，这有点像是赌博了，呵呵）。我举个例子，假设要在10000个随机的数据中找出最大的10个数，那么采用堆排序应该是最合适的，因为：第一，经验指出堆排序是一个非常稳定的算法，在各种环境中其效率变化不会太大；第二，堆排序的特性决定了只要构建一棵根节点为最大数的优先队列树，然后取其前10个根节点就行了。

该说的基本上就说完了，我不想重复敲入书上的话，各种排序算法的具体解释请参阅教科书，下面给出代码。

11.2 代码实现

各种排序的代码实现如下：

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  sort.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-23 16:49:42
```

```
//  Comment    :
//
////////////////////////////////////////////////////////////////////////////

#ifndef __SORT_H__
#define __SORT_H__

template<typename T>
class CSort
{
private:
    // the following three functions are for HeapSort():
    int  LeftChild(const int i);
    void PercDown(T x[], int i, const int n);
    void Swap(T *l, T *r);
    // the following two functions are for MergeSort():
    void MSort(T x[], T tmp[], int left, int right);
    void Merge(T x[], T tmp[], int lpos, int rpos, int rightend);
    // for QuickSort():
    T Median3(T x[], const int left, const int right);

public:
    void InsertSort(T x[], const int n);
    void ShellSort(T x[], const int n);
    void HeapSort(T x[], const int n);
    void MergeSort(T x[], int n);
    void QuickSort(T x[], int left, int right);
};

template<typename T>
inline void CSort<T>::InsertSort(T x[], const int n)
{
    int i;
    int j;
    T tmp;

    for (i = 0; i < n; ++i)
    {
        tmp = x[i];            // copy it first
        for (j = i; j > 0; --j) // unsorted region; (0 ~ (i - 1)) is sorted
            if (x[j - 1] > tmp)
                x[j] = x[j - 1];// move back elements to empty a right position
            else
                break;         // we got it! x[j] is the right position
        x[j] = tmp;             // place it to the right position
    }
}

template<typename T>
inline void CSort<T>::ShellSort(T x[], const int n)
{
    int i;
    int j;
    int nIncrement;
    T tmp;

    for (nIncrement = n / 2; nIncrement > 0; nIncrement /= 2)
    {
        for (i = nIncrement; i < n; ++i)
        {
            tmp = x[i];
            for (j = i; j >= nIncrement; j -= nIncrement)
            {
                if (tmp < x[j - nIncrement])
                    x[j] = x[j - nIncrement];
                else
                    break;
```

```
            }
            x[j] = tmp;
        }
    }
}

template<typename T>
inline int CSort<T>::LeftChild(const int i)
{
    return (2 * i + 1);
}

template<typename T>
inline void CSort<T>::PercDown(T x[], int i, const int n)
{
    int nChild;
    T tmp;

    for (tmp = x[i]; LeftChild(i) < n; i = nChild)
    {
        nChild = LeftChild(i);
        if ((nChild != n - 1) && (x[nChild + 1] > x[nChild]))
            ++nChild;
        if (tmp < x[nChild])
            x[i] = x[nChild];
        else
            break;
    }
    x[i] = tmp;
}

template<typename T>
inline void CSort<T>::Swap(T *l, T *r)
{
    T tmp = *l;
    *l = *r;
    *r = tmp;
}

template<typename T>
inline void CSort<T>::HeapSort(T x[], const int n)
{
    int i;

    for (i = n / 2; i >= 0; --i)    // build heap
        PercDown(x, i, n);
    for (i = n - 1; i > 0; --i)
    {
        Swap(&x[0], &x[i]);         // delete max
        PercDown(x, 0, i);
    }
}

template<typename T>
inline void CSort<T>::Merge(T x[], T tmp[], int lpos, int rpos, int rightend)
{
    int i;
    int leftend;
    int numelements;
    int tmppos;

    leftend = rpos - 1;
    tmppos = lpos;
    numelements = rightend - lpos + 1;

    // main loop
    while ((lpos <= leftend) && (rpos <= rightend))
```

```
    {
        if (x[lpos] <= x[rpos])
            tmp[tmppos++] = x[lpos++];
        else
            tmp[tmppos++] = x[rpos++];
    }

    while (lpos <= leftend)     // copy rest of first half
        tmp[tmppos++] = x[lpos++];
    while (rpos <= rightend)    // copy rest of second half
        tmp[tmppos++] = x[rpos++];

    // copy tmp back
    for (i = 0; i < numelements; ++i, --rightend)
        x[rightend] = tmp[rightend];
}

template<typename T>
inline void CSort<T>::MSort(T x[], T tmp[], int left, int right)
{
    int center;

    if (left < right)
    {
        center = (left + right) / 2;
        MSort(x, tmp, left, center);
        MSort(x, tmp, center + 1, right);
        Merge(x, tmp, left, center + 1, right);
    }
}

template<typename T>
inline void CSort<T>::MergeSort(T x[], int n)
{
    T *tmp;

    tmp = new (T[n * sizeof(T)]);
    if (NULL != tmp)
    {
        MSort(x, tmp, 0, n - 1);
        delete tmp;
    }
}

template<typename T>
inline T CSort<T>::Median3(T x[], const int left, const int right)
{
    int center = (left + right) / 2;

    if (x[left] > x[center])
        Swap(&x[left], &x[center]);
    if (x[left] > x[right])
        Swap(&x[left], &x[right]);
    if (x[center] > x[right])
        Swap(&x[center], &x[right]);

    // invariant: x[left] <= x[center] <= x[right]

    Swap(&x[center], &x[right - 1]);    // hide pivot
    return x[right - 1];                // return pivot
}

template<typename T>
inline void CSort<T>::QuickSort(T x[], int left, int right)
{
    int i;
    int j;
```

```
    int cutoff = 3;
    T pivot;

    if (left + cutoff <= right)
    {
        pivot = Median3(x, left, right);
        i = left;
        j = right - 1;
        for (;;)
        {
            while (x[++i] < pivot) {}
            while (x[--j] > pivot) {}
            if (i < j)
                Swap(&x[i], &x[j]);
            else
                break;
        }
        Swap(&x[i], &x[right - 1]); // restore pivot
        QuickSort(x, left, i - 1);
        QuickSort(x, i + 1, right);
    }
    else    // do an insertion sort on the subarray
        InsertSort(x + left, right - left + 1);
}

#endif  // __SORT_H__
```

测试代码：

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   sort.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-23 16:49:39
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include "sort.h"

int main()
{
    int x[] = {2, 9, 1, 6, 4, 8, 10, 7, 3, 5};
    CSort<int> sort;

    sort.QuickSort(x, 0, 9);
    // sort.ShellSort(x, 10);
}
```

[首页, 上一页, 下一页; 目录]

第十二章

图的储存

12.1 基本概念

图（Graph）是数据结构中的最后一个"堡垒"——攻下它，数据结构就结束了。但就像在打游戏的最终BOSS一样，BOSS肯定是最强的，图也一样，它比线性表和树都更为复杂。在线性表中，数据元素间仅有线性关系，每个数据元素只有一个直接前驱和一个直接后继，也就是"一对一"的关系；在树形结构中，数据元素之间有着比较明显的层次关系，并且每一层上的数据元素可能和下一层中多个元素（儿子）相关，但只能和上一层中的一个元素（父亲）相关，也就是它是"一对多"的关系。到了图形结构中，数据元素之间的关系就可以是任意的，图中任意两个数据元素之间都可能相关，即"多对多"的关系。因此，图在200多年的发展中，应用极其广泛。这也造成了大部分高级的算法分析都不可避免地要用到图的知识。

不管图形结构有多复杂，我们要做的第一步必定是要先把它用某种结构储存起来。关于这一点，我们在树里面已经有了体会——对树的学习，关键是学习如何建树以及排序。我们要透过现象看本质，别看书里唧歪了半天，列了好多种储存图的方法，但其核心其实只有一个，那就是邻接矩阵（Adjacency Matrix）。但邻接矩阵的缺点是它对空间的耗费比较大，因为它是用一个二维数组来储存图的顶点和边的信息的——如果有N个顶点，则需要有N ^ 2的空间来储存。因此，如果图是稀疏的，那么我们就可以用邻接表（Adjacency List）来储存它，充分发挥链表的动态规划空间的优点。

下面我就分别给出这两种结构的代码实现。其中，邻接表使用了前面所写的单链表的类，因此在具体的实现上并不会太困难。另外，由于图结构本身比较复杂的原因，我无法把基类写得十分具有通用性，但它们应该已经可以基本满足后面的学习的需要了。

12.2 邻接矩阵

```
//////////////////////////////////////////////////////////////////////////
//
// FileName    :  MatrixGraph.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-27 0:01:12
// Comment     :
//
//////////////////////////////////////////////////////////////////////////

#ifndef __MATRIX_GRAPH_H__
#define __MATRIX_GRAPH_H__

#include <iostream>
using namespace std;

#include <assert.h>
#include <crtdbg.h>

#ifdef _DEBUG
```

```
#define DEBUG_NEW new (_NORMAL_BLOCK, THIS_FILE, __LINE__)
#endif

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#ifdef _DEBUG
#ifndef ASSERT
#define ASSERT  assert
#endif
#else   // not _DEBUG
#ifndef ASSERT
#define ASSERT
#endif
#endif  // _DEBUG

template<typename T_Vertex, typename T_Edge>
class CMatrixGraph
{
    friend ostream& operator<<(ostream &os, CMatrixGraph<T_Vertex, T_Edge> &g);

private:
    int m_nVertexNum;
    int m_nEdgeNum;
    int m_nMaxVertexNum;
    T_Vertex *m_Vertex;
    T_Edge **m_Edge;
    T_Vertex m_NoVertex;
    T_Edge m_NoEdge;

public:
    CMatrixGraph(const int nMaxVertexNum);
    ~CMatrixGraph();

public:
    int GetVertexNum() const;
    int GetEdgeNum() const;
    T_Vertex& GetVertexAt(const int n);
    T_Vertex  GetVertexAt(const int n) const;
    T_Edge& GetEdgeAt(const int nVertexIndex, const int n);
    T_Edge  GetEdgeAt(const int nVertexIndex, const int n) const;
    int Find(const T_Vertex &v, int *nIndex = NULL) const;
    int InsertVertex(const T_Vertex &v);
    int InsertEdge(const T_Vertex &v1, const T_Vertex &v2, const T_Edge &e);
    int GetFirstAdjVertexIndex(const int n) const;
    int GetNextAdjVertexIndex(const int n, const int nn) const;
};

template<typename T_Vertex, typename T_Edge>
inline CMatrixGraph<T_Vertex, T_Edge>::CMatrixGraph(const int nMaxVertexNum)
                                : m_nVertexNum(0),
                                  m_nEdgeNum(0),
                                  m_nMaxVertexNum(nMaxVertexNum),
                                  m_NoVertex(0), // this can be customized
                                  m_NoEdge(0)    // this can be customized
{
    int i;

    m_Edge = new T_Edge*[nMaxVertexNum];
    if (NULL == m_Edge)
        return;

    for (i = 0; i < nMaxVertexNum; ++i)
    {
```

```
        m_Edge[i] = new T_Edge[nMaxVertexNum];
    }

    m_Vertex = new T_Vertex[nMaxVertexNum];
}

template<typename T_Vertex, typename T_Edge>
inline CMatrixGraph<T_Vertex, T_Edge>::~CMatrixGraph()
{
    int i;

    delete[] m_Vertex;

    for (i = 0; i < m_nMaxVertexNum; ++i)
    {
        delete[] m_Edge[i];
    }
    delete[] m_Edge;
}

template<typename T_Vertex, typename T_Edge>
inline int CMatrixGraph<T_Vertex, T_Edge>::Find(
    const T_Vertex &v,
    int *nIndex
) const
{
    int i;
    int nVertexNum = m_nVertexNum;

    for (i = 0; i < nVertexNum; ++i)
    {
        if (v == m_Vertex[i])
        {
            if (nIndex)
                *nIndex = i;
            return 1;
        }
    }
    return 0;
}

template<typename T_Vertex, typename T_Edge>
inline int CMatrixGraph<T_Vertex, T_Edge>::InsertVertex(const T_Vertex &v)
{
    int i;

    if ((m_nVertexNum >= m_nMaxVertexNum) || Find(v))
        return 0;

    m_Vertex[m_nVertexNum] = v;

    for (i = 0; i < m_nMaxVertexNum; ++i)
        m_Edge[m_nVertexNum][i] = m_NoEdge;

    ++m_nVertexNum;

    return 1;
}

template<typename T_Vertex, typename T_Edge>
inline int CMatrixGraph<T_Vertex, T_Edge>::InsertEdge(
    const T_Vertex &v1,
    const T_Vertex &v2,
    const T_Edge &e
)
{
    int nIndexV1;
```

```
    int nIndexV2;

    if (
        (v1 == v2) ||
        (!Find(v1, &nIndexV1)) ||
        (!Find(v2, &nIndexV2)) ||
        (m_Edge[nIndexV1][nIndexV2] != m_NoEdge)
    )
        return 0;

    m_Edge[nIndexV1][nIndexV2] = e;
    ++m_nEdgeNum;

    return 1;
}

template<typename T_Vertex, typename T_Edge>
inline T_Edge& CMatrixGraph<T_Vertex, T_Edge>::GetEdgeAt(
    const int nVertexIndex,
    const int n
)
{
    if ((0 > nVertexIndex) || (nVertexIndex >= m_nMaxVertexNum))
        return m_NoEdge;

    if ((0 > n) || (n >= m_nMaxVertexNum))
        return m_NoEdge;

    return *(&m_Edge[nVertexIndex][n]);
}

template<typename T_Vertex, typename T_Edge>
inline T_Edge CMatrixGraph<T_Vertex, T_Edge>::GetEdgeAt(
    const int nVertexIndex,
    const int n
) const
{
    if ((0 > nVertexIndex) || (nVertexIndex >= m_nMaxVertexNum))
        return m_NoEdge;

    if ((0 > n) || (n >= m_nMaxVertexNum))
        return m_NoEdge;

    return m_Edge[nVertexIndex][n];
}

template<typename T_Vertex, typename T_Edge>
inline T_Vertex& CMatrixGraph<T_Vertex, T_Edge>::GetVertexAt(const int n)
{
    if ((0 > n) || (n >= m_nMaxVertexNum))
        return m_NoVertex;
    else
        return *(&m_Vertex[n]);
}

template<typename T_Vertex, typename T_Edge>
inline T_Vertex CMatrixGraph<T_Vertex, T_Edge>::GetVertexAt(const int n) const
{
    if ((0 > n) || (n >= m_nMaxVertexNum))
        return m_NoVertex;
    else
        return m_Vertex[n];
}

template<typename T_Vertex, typename T_Edge>
inline int CMatrixGraph<T_Vertex, T_Edge>::GetVertexNum() const
{
```

```
    return m_nVertexNum;
}

template<typename T_Vertex, typename T_Edge>
inline int CMatrixGraph<T_Vertex, T_Edge>::GetEdgeNum() const
{
    return m_nEdgeNum;
}

template<typename T_Vertex, typename T_Edge>
inline int CMatrixGraph<T_Vertex, T_Edge>::GetFirstAdjVertexIndex(
    const int n
) const
{
    int i;

    for (i = 0; i < m_nVertexNum; ++i)
    {
        if (m_Edge[n][i] != m_NoEdge)
            return i;
    }
    return -1;
}

template<typename T_Vertex, typename T_Edge>
inline int CMatrixGraph<T_Vertex, T_Edge>::GetNextAdjVertexIndex(
    const int n,
    const int nn
) const
{
    int i;

    for (i = nn + 1; i < m_nVertexNum; ++i)
    {
        if (m_Edge[n][i] != m_NoEdge)
            return i;
    }
    return -1;
}

template<typename T_Vertex, typename T_Edge>
inline ostream &operator<<(ostream &os, CMatrixGraph<T_Vertex, T_Edge> &g)
{
    int i;
    int j;
    int nVertexNum;

    nVertexNum = g.GetVertexNum();
    for (i = 0; i < nVertexNum; ++i)
    {
        for (j = 0; j < nVertexNum; ++j)
        {
            os << g.GetEdgeAt(i, j) << ' ';
        }
        os << endl;
    }

    return os;
}

#endif  // __MATRIX_GRAPH_H__
```

测试代码：

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  MatrixGraph.cpp
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-27 0:02:03
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include "MatrixGraph.h"

int main()
{
    CMatrixGraph<int, int> mgraph(4);

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    // (1) - - >(2)
    // | ↖
    // |   \
    // ↓    \
    // (3) - - >(4)
    mgraph.InsertVertex(1);
    mgraph.InsertVertex(2);
    mgraph.InsertVertex(3);
    mgraph.InsertVertex(4);
    mgraph.InsertEdge(1, 2, 1);
    mgraph.InsertEdge(1, 3, 1);
    mgraph.InsertEdge(3, 4, 1);
    mgraph.InsertEdge(4, 1, 1);

    cout << mgraph << endl;
}
```

12.3 邻接链表

```
////////////////////////////////////////////////////////////////////////////
//
// FileName    :  ListGraph.h
// Version     :  0.10
// Author      :  Luo Cong
// Date        :  2005-1-27 10:26:20
// Comment     :
//
////////////////////////////////////////////////////////////////////////////

#ifndef __LIST_GRAPH_H__
#define __LIST_GRAPH_H__

#include <iostream>
using namespace std;

#include "../../slist/src/slist.h"

template<typename T_Vertex, typename T_Edge>
class CListGraph
{
    friend ostream& operator<<(ostream &os, CListGraph<T_Vertex, T_Edge> &g);

private:
```

```
    typedef struct tagLGEdge
    {
        int nextvertexindex;
        T_Edge edata;
    } LGEdge;

    typedef struct tagLGVertex
    {
        T_Vertex vdata;
        CSList<LGEdge> *edgelist;
    } LGVertex;

    int m_nVertexNum;
    int m_nEdgeNum;
    CSList<LGVertex> m_Vertex;

public:
    CListGraph();
    ~CListGraph();
    void Output(ostream &os) const;

private:
    int GetVertexAt(const int n, LGVertex *vertex) const;
    int GetEdgeAt(const int nVertexIndex, const int n, LGEdge *edge) const;

public:
    int GetVertexNum() const;
    int GetEdgeNum() const;
    int GetVertexAt(const int n, T_Vertex *v) const;
    int GetEdgeAt(const int nVertexIndex, const int n, T_Edge *e) const;
    T_Edge GetEdgeAt(const int nVertexIndex, const int n) const;
    int Find(const T_Vertex &v, int *nIndex = NULL) const;
    int InsertVertex(const T_Vertex &v);
    int InsertEdge(const T_Vertex &v1, const T_Vertex &v2, const T_Edge &e);
    int GetFirstAdjVertexIndex(const int n) const;
    int GetNextAdjVertexIndex(const int n, const int nn) const;
};

template<typename T_Vertex, typename T_Edge>
inline CListGraph<T_Vertex, T_Edge>::CListGraph()
                                : m_nVertexNum(0),
                                  m_nEdgeNum(0)
{
}

template<typename T_Vertex, typename T_Edge>
inline CListGraph<T_Vertex, T_Edge>::~CListGraph()
{
    int i;
    int nVertexNum = m_nVertexNum;
    CSList<LGEdge> *edgelist;

    for (i = 0; i < nVertexNum; ++i)
    {
        edgelist = m_Vertex.GetAt(i + 1).edgelist;
        if (edgelist)
        {
            edgelist->RemoveAll();
            delete edgelist;
        }
    }
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::GetVertexNum() const
{
    return m_nVertexNum;
```

```
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::GetEdgeNum() const
{
    return m_nEdgeNum;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::GetVertexAt(
    const int n,
    LGVertex *vertex
) const
{
    ASSERT(vertex);

    if ((0 > n) || (n >= m_Vertex.GetCount()))
        return 0;

    *vertex = m_Vertex.GetAt(n + 1);
    return 1;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::GetVertexAt(
    const int n,
    T_Vertex *v
) const
{
    ASSERT(v);

    LGVertex vertex;

    if (GetVertexAt(n, &vertex))
    {
        *v = vertex.vdata;
        return 1;
    }
    else
        return 0;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::GetEdgeAt(
    const int nVertexIndex,
    const int n,
    LGEdge *edge
) const
{
    ASSERT(edge);

    LGVertex vertex;
    int nVertexEdgelistCount;

    if (0 == GetVertexAt(nVertexIndex, &vertex))
        return 0;

    if (vertex.edgelist)
        nVertexEdgelistCount = vertex.edgelist->GetCount();
    else
        return 0;

    if (
        (0 > n) ||
        (n >= nVertexEdgelistCount)
    )
        return 0;
```

```
    *edge = vertex.edgelist->GetAt(n + 1);

    return 1;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::GetEdgeAt(
    const int nVertexIndex,
    const int n,
    T_Edge *e
) const
{
    ASSERT(e);

    LGEdge edge;

    if (GetEdgeAt(nVertexIndex, n, &edge))
    {
        *e = edge.edata;
        return 1;
    }
    else
        return 0;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::Find(
    const T_Vertex &v,
    int *nIndex
) const
{
    int i;
    int nVertexNum = m_nVertexNum;
    LGVertex vertex;

    for (i = 0; i < nVertexNum; ++i)
    {
        vertex = m_Vertex.GetAt(i + 1);
        if (v == vertex.vdata)
        {
            if (nIndex)
                *nIndex = i;
            return 1;
        }
    }
    return 0;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::InsertVertex(const T_Vertex &v)
{
    LGVertex vertex;

    if (Find(v))
        return 0;

    vertex.vdata = v;
    vertex.edgelist = NULL;
    m_Vertex.AddTail(vertex);
    ++m_nVertexNum;

    return 1;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::InsertEdge(
```

```
    const T_Vertex &v1,
    const T_Vertex &v2,
    const T_Edge &e
)
{
    int i;
    int nIndexV1;
    int nIndexV2;
    LGEdge edge;
    CSList<LGEdge> *edgelist;
    int nVertexEdgelistCount;

    if (
        (v1 == v2) ||
        (!Find(v1, &nIndexV1)) ||
        (!Find(v2, &nIndexV2))
    )
        return 0;

    // if there's no edges, let's create it first
    edgelist = m_Vertex.GetAt(nIndexV1 + 1).edgelist;
    if (NULL == edgelist)
    {
        edgelist = new CSList<LGEdge>;
        m_Vertex.GetAt(nIndexV1 + 1).edgelist = edgelist;
    }

    // is there an edge between v1 and v2 already?
    nVertexEdgelistCount = edgelist->GetCount();
    for (i = 0; i < nVertexEdgelistCount; ++i)
    {
        edge = edgelist->GetAt(i + 1);
        if (
            (edge.edata == e) &&
            (edge.nextvertexindex == nIndexV2)
        )
            return 0;
    }

    // new edge's data
    edge.edata = e;
    edge.nextvertexindex = nIndexV2;

    edgelist->AddTail(edge);

    ++m_nEdgeNum;

    return 1;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::GetFirstAdjVertexIndex(
    const int n
) const
{
    LGVertex vertex;

    if (0 == GetVertexAt(n, &vertex))
        return -1;
    if (vertex.edgelist)
        return vertex.edgelist->GetHead().nextvertexindex;
    return -1;
}

template<typename T_Vertex, typename T_Edge>
inline int CListGraph<T_Vertex, T_Edge>::GetNextAdjVertexIndex(
    const int n,
```

```
    const int nn
) const
{
    LGEdge edge;
    LGVertex vertex;
    int nVertexEdgelistCount;

    if (0 == GetVertexAt(n, &vertex))
        return -1;

    if (vertex.edgelist)
        nVertexEdgelistCount = vertex.edgelist->GetCount();
    else
        return -1;

    if (
        (0 > nn) ||
        ((nn + 1) >= nVertexEdgelistCount)
    )
        return -1;

    edge = vertex.edgelist->GetAt((nn + 1) + 1);

    return edge.nextvertexindex;
}

template<typename T_Vertex, typename T_Edge>
inline void CListGraph<T_Vertex, T_Edge>::Output(ostream &os) const
{
    int i;
    int j;
    LGEdge edge;
    LGVertex vertex;
    int nVertexNum;
    int nVertexEdgelistCount;

    nVertexNum = GetVertexNum();
    for (i = 0; i < nVertexNum; ++i)
    {
        if (0 == GetVertexAt(i, &vertex))
            return ;
        os << "(V" << i + 1 << ") ";
        os << 'V' << vertex.vdata;
        if (vertex.edgelist)
            nVertexEdgelistCount = vertex.edgelist->GetCount();
        else
            nVertexEdgelistCount = 0;
        for (j = 0; j < nVertexEdgelistCount; ++j)
        {
            os << " --> ";
            edge = vertex.edgelist->GetAt(j + 1);
            os << 'V' << edge.nextvertexindex + 1;
        }
        os << endl;
    }
}

template<typename T_Vertex, typename T_Edge>
inline ostream& operator<<(ostream &os, CListGraph<T_Vertex, T_Edge> &g)
{
    g.Output(os);
    return os;
}

#endif  // __LIST_GRAPH_H__
```

测试代码：

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   ListGraph.cpp
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-27 10:27:55
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#include "ListGraph.h"

int main()
{
    CListGraph<int, int> lgraph;

#ifdef _DEBUG
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    // (1) - - >(2)
    // | ↖
    // |   ＼
    // ↓     ＼
    // (3) - - >(4)
    lgraph.InsertVertex(1);
    lgraph.InsertVertex(2);
    lgraph.InsertVertex(3);
    lgraph.InsertVertex(4);
    lgraph.InsertEdge(1, 2, 1);
    lgraph.InsertEdge(1, 3, 1);
    lgraph.InsertEdge(3, 4, 1);
    lgraph.InsertEdge(4, 1, 1);

    cout << lgraph << endl;
}
```

[首页, 上一页, 下一页; 目录]

第十三章

图的遍历

13.1 基本概念

解决了图的储存问题后，接下来的肯定就是解决如何去访问图上面的元素的问题了，也就是图的遍历。书里面对图的遍历是用深度优先搜索算法（Depth First Search，简称DFS）和广度优先搜索算法（Breadth First Search，简称BFS），其实说白了就是按照"分层"的思想来进行。深度优先就是先访问完最深层次的数据元素，广度优先就是先访问完同一层次的数据元素，它们的时间复杂度都是一样的，只是访问元素的顺序不同而已。

13.2 代码实现

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   TraverseGraph.h
//  Version     :   0.10
//  Author      :   Luo Cong
//  Date        :   2005-1-29 16:28:44
//  Comment     :
//
////////////////////////////////////////////////////////////////////////////

#ifndef __TRAVERSE_GRAPH_H__
#define __TRAVERSE_GRAPH_H__

#include "../../ListGraph/src/ListGraph.h"
#include "../../queue/src/lqueue.h"

template<typename T_Vertex, typename T_Edge>
class CTraverseGraph : public CListGraph<T_Vertex, T_Edge>
{
protected:
    int *m_nVisited;

private:
    int InitializeVisited(const int n);
    void FinalizeVisited();
    void DFS(const int n, void (*Visit)(const T_Vertex &vertex)) const;

public:
    void DFS(void (*Visit)(const T_Vertex &vertex));
    void BFS(void (*Visit)(const T_Vertex &vertex));
};

template<typename T_Vertex, typename T_Edge>
inline int CTraverseGraph<T_Vertex, T_Edge>::InitializeVisited(const int n)
{
    int i;
```

```
    m_nVisited = new int[n];
    if (NULL == m_nVisited)
        return 0;

    for (i = 0; i < n; ++i)
        m_nVisited[i] = 0;

    return 1;
}

template<typename T_Vertex, typename T_Edge>
inline void CTraverseGraph<T_Vertex, T_Edge>::FinalizeVisited()
{
    if (m_nVisited)
    {
        delete[] m_nVisited;
        m_nVisited = NULL;
    }
}

template<typename T_Vertex, typename T_Edge>
inline void CTraverseGraph<T_Vertex, T_Edge>::DFS(
    const int n,
    void (*Visit)(const T_Vertex &vertex)
) const
{
    int i;
    int j = 0;
    int nRetCode;
    T_Vertex vertex;

    m_nVisited[n] = 1;

    nRetCode = GetVertexAt(n, &vertex);
    if (0 == nRetCode)
        return ;
    Visit(vertex);

    i = GetFirstAdjVertexIndex(n);
    for (; i != -1; i = GetNextAdjVertexIndex(n, j++))
    {
        if (!m_nVisited[i])
            DFS(i, Visit);
    }
}

template<typename T_Vertex, typename T_Edge>
inline void CTraverseGraph<T_Vertex, T_Edge>::DFS(
    void (*Visit)(const T_Vertex &vertex)
)
{
    int i;
    int nVertexNum;

    nVertexNum = GetVertexNum();

    if (!InitializeVisited(nVertexNum))
        return ;

    for (i = 0; i < nVertexNum; ++i)
    {
        if (!m_nVisited[i])
            DFS(i, Visit);
    }

    FinalizeVisited();
}
```

```
template<typename T_Vertex, typename T_Edge>
inline void CTraverseGraph<T_Vertex, T_Edge>::BFS(
    void (*Visit)(const T_Vertex &vertex)
)
{
    int i;
    int j;
    int k;
    int l;
    int nRetCode;
    int nVertexNum;
    T_Vertex vertex;
    CLQueue<int> queue;

    nVertexNum = GetVertexNum();

    if (!InitializeVisited(nVertexNum))
        return ;

    for (i = 0; i < nVertexNum; ++i)
    {
        if (m_nVisited[i])
            continue;

        // visit vertex[i]
        m_nVisited[i] = 1;
        nRetCode = GetVertexAt(i, &vertex);
        if (0 == nRetCode)
            return ;
        Visit(vertex);
        queue.EnQueue(i);

        // visit vertex[i]'s adjacency vertex(s)
        while (!queue.IsEmpty())
        {
            j = queue.DeQueue();    // equal to i above
            k = GetFirstAdjVertexIndex(j);
            l = 0;
            // adjacency vertex(s):
            for (; k != -1; k = GetNextAdjVertexIndex(j, l++))
            {
                if (!m_nVisited[k])
                {
                    m_nVisited[k] = 1;
                    nRetCode = GetVertexAt(k, &vertex);
                    if (0 == nRetCode)
                        return ;
                    Visit(vertex);
                    queue.EnQueue(k);
                }
            }
        }
    }

    FinalizeVisited();
}

#endif  // __TRAVERSE_GRAPH_H__
```

测试代码：

```
////////////////////////////////////////////////////////////////////////////
//
//  FileName    :   TraverseGraph.cpp
//  Version     :   0.10
```

```
// Author     :  Luo Cong
// Date       :  2005-1-29 16:30:34
// Comment    :
//
////////////////////////////////////////////////////////////////////////////

#include "TraverseGraph.h"

typedef int ElementType;

static void PrintVertex(const ElementType &vertex)
{
   cout << 'V' << vertex << " --> ";
}

int main()
{
   CTraverseGraph<ElementType, ElementType> tgraph;

#ifdef _DEBUG
   _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

   //        (1)
   //        /  \
   //       /    \
   //      /      \
   //    (2)      (3)
   //    / \      / \
   //   /   \    /   \
   // (4)   (5) (6)--(7)
   //   \   /
   //    \ /
   //    (8)
   tgraph.InsertVertex(1);
   tgraph.InsertVertex(2);
   tgraph.InsertVertex(3);
   tgraph.InsertVertex(4);
   tgraph.InsertVertex(5);
   tgraph.InsertVertex(6);
   tgraph.InsertVertex(7);
   tgraph.InsertVertex(8);
   // 因为CListGraph是一个有向图类，所以这里为了创建一个无向图，
   // 必须把每条边从入边和出边两个方向分别创建一次：
   tgraph.InsertEdge(1, 2, 1);
   tgraph.InsertEdge(2, 1, 1);
   tgraph.InsertEdge(1, 3, 1);
   tgraph.InsertEdge(3, 1, 1);
   tgraph.InsertEdge(2, 4, 1);
   tgraph.InsertEdge(4, 2, 1);
   tgraph.InsertEdge(2, 5, 1);
   tgraph.InsertEdge(5, 2, 1);
   tgraph.InsertEdge(3, 6, 1);
   tgraph.InsertEdge(6, 3, 1);
   tgraph.InsertEdge(3, 7, 1);
   tgraph.InsertEdge(7, 3, 1);
   tgraph.InsertEdge(6, 7, 1);
   tgraph.InsertEdge(7, 6, 1);
   tgraph.InsertEdge(4, 8, 1);
   tgraph.InsertEdge(8, 4, 1);
   tgraph.InsertEdge(5, 8, 1);
   tgraph.InsertEdge(8, 5, 1);

   cout << "Graph is:" << endl;
   cout << tgraph << endl;

   cout << "DFS Traverse:" << endl;
```

```
    tgraph.DFS(PrintVertex);
    cout << "NULL" << endl << endl;

    cout << "BFS Traverse:" << endl;
    tgraph.BFS(PrintVertex);
    cout << "NULL" << endl;
}
```

13.3 说明

为了说明DFS和BFS，我另外写了一个类：CTraverseGraph，它继承于前面的邻接链表图类：CListGraph。呵呵，用C++的继承机制真是舒服啊，不用每次都重写一堆功能重复的代码了，而且它能更直观地表现出数据结构的ADT来，实在是居家旅行、跳槽单干的必备良器……嗯，扯远了，咳咳。另外，我之所以不把DFS和BFS这两个函数直接写到图的基类里面，是因为对图的遍历很难做到高度抽象的通用性，所以在这里就只写了一个试验级别的CTraverseGraph了，但是说实话，对于试验来说，它已经足够了。

[首页, 上一页, 下一页; 目录]