ACM 程序设计常用算法与数据结构参考

Tomsdinary

目录

前言

如今的程序设计已不再是个人英雄时代了，程序的设计和开发实施需要靠团队成员的积极配合和合作。软件技术在当今时代已不是信息技术竞争核心，对于技术知识的获取变得不再重要。当今 IT 竞争，靠的是先进的观念，有效的沟通和合作，靠的是高瞻远瞩的预见能力，靠的是个人的想法而绝不是技能。当然掌握好众多技术是实现

我们独特创意的途径，但绝不是我们可以屹立 IT 界的根本法宝。

随着互联网发展的不断深入，技术知识的获取不再成为问题。程序员不能单靠通晓某一核心技术而获得核心竞争力。当今的 IT 界知识分享，知识交流，知识开放是主旋律，凡是开放的平台，开放的个人，开放的公司才是真正拥有竞争力的，凡是那些封闭的平台，封闭的个人，封闭的公司其发展的道路终会是艰难的。而这种开放的态度在中国程序员中更应该得到普及与遵守。其根本在于中国程序员中的高手充其量也只是一个高级用户，真正的技术掌握在技术公司那里。所以为什么还有保留一些使用技巧呢。不开放不分享不合作，优秀的程序员中会沦为平庸的人。

基于以上思考和论断，我将自己三年在算法设计和数据结构学习过程中可供借鉴和参考使用的代码总结如下。一方面作为我们队参加 ACM 的内部参考资料，另一方面分享出来供后来者学习参考。或许对诸位会有点帮助。同时也请您记住和接受以上观点，在一个分享交流的环境你，你的技术进步会更加迅速。也希望那些有好代码，好思想的高手将自己的智慧分享出来。整理出来。这不仅有利于个人学习总结，更有利于我们怀化学院 ACM 队伍健康发展。

排序算法

插入排序

```
/*
函数名:        InsertionSort
功能:          插入排序
```

```
模板参数说明：T必须支持小于操作
参数说明：      data待排序数组，size待排序数组大小
前置条件：      data!=NULL, size>0
后置条件：      data按非降序排列
用法：          int arr[]={10,9,8,4,5,7,6,3,1,4};
                 InsertionSort(arr, 10);
*/
template <typename T>
void InsertionSort (T data[], int size)
{
    int i,j;  T temp;
    for (i=1; i<size; ++i)
    {
        temp=data[i];
        for (j=i; j>0 && temp<data[j-1]; --j)
            data[j]=data[j-1];
        data[j]=temp;
    }
}

/*
函数名：        InsertionSort
功能：          插入排序
模板参数说明：T元素类型，Func函数对象或函数指针
参数说明：      data待排序数组，size待排序数组大小，f函数对象或地址
前置条件：      data!=NULL, size>0
后置条件：      data按f排列
用法：
bool cmp(int a, int b)
{ return a<b; }
int arr[]={10,9,8,4,5,7,6,3,1,4};
InsertionSort(arr, 10, cmp);
*/
template <typename T, typename Func >
void InsertionSort (T data[], int size, Func f)
{
    int i,j;  T temp;
    for (i=1; i<size; ++i)
    {
        temp=data[i];
        for (j=i; j>0 && f(temp,data[j-1]); --j)
            data[j]=data[j-1];
        data[j]=temp;
    }
}
```

选择排序

```
/*
函数名:       SelectionSort
功能:         选择排序
模板参数说明:T必须支持小于操作
参数说明:     data待排序数组, size待排序数组大小
前置条件:     data!=NULL, size>0
后置条件:     data按非降序排列
用法:
#include <algorithm>
int arr[]={10,9,8,4,5,7,6,3,1,4};
SelectionSort(arr, 10);
*/
template <typename T>
void SelectionSort (T data[], int size)
{
    int i,j,k;
    for (i=0; i<size-1; ++i)
    {
        k=i;
        for (j=i+1; j<size; ++j)
        {
            if (data[j]<data[k])
                k=j;
        }
        std::swap(data[i], data[k]);
    }
}

/*
函数名:       SelectionSort
功能:         选择排序
模板参数说明:T元素类型，Func函数对象或函数地址
参数说明:     data待排序数组, size待排序数组大小, f函数对象或地址
前置条件:     data!=NULL, size>0
后置条件:     data按f排列
用法:
#include <algorithm>
int arr[]={10,9,8,4,5,7,6,3,1,4};
bool cmp(int a, int b)
```

```
{ return a<b; }
SelectionSort(arr, 10, cmp);
*/
template <typename T, typename Func>
void SelectionSort (T data[], int size, Func f)
{
    int i,j,k;
    for (i=0; i<size-1; ++i)
    {
        k=i;
        for (j=i+1; j<size; ++j)
        {
            if (f(data[j],data[k]))
                k=j;
        }
        std::swap(data[i], data[k]);
    }
}
```

冒泡排序

```
/*
函数名:        BubbleSort
功能:          冒泡排序
模板参数说明:T必须支持小于操作
参数说明:      data待排序数组, size待排序数组大小
前置条件:      data!=NULL, size>0
后置条件:      data按非降序排列
用法:
#include <algorithm>
int arr[]={10,9,8,4,5,7,6,3,1,4};
BubbleSort(arr, 10);
*/
template <typename T>
void BubbleSort (T data[], int size)
{
    int i,j;
    for (i=0; i<size-1; ++i)
    {
        for (j=size-1; j>i; --j)
        {
            if (data[j]<data[j-1])
                std::swap(data[j], data[j-1]);
        }
    }
}

/*
函数名:       BubbleSort
功能:         冒泡排序
模板参数说明:T元素类型,Func函数对象或函数地址
参数说明:     data待排序数组, size待排序数组大小,f函数对象或地址
前置条件:     data!=NULL, size>0
后置条件:     data按f序排列
用法:
#include <algorithm>
bool cmp(int a, int b)
{ return a<b; }
int arr[]={10,9,8,4,5,7,6,3,1,4};
BubbleSort(arr, 10, cmp);
*/
template <typename T, typename Func>
void BubbleSort (T data[], int size, Func f)
{
    int i,j;
    for (i=0; i<size-1; ++i)
    {
        for (j=size-1; j>i; --j)
        {
            if (f(data[j],data[j-1]))
                std::swap(data[j], data[j-1]);
        }
    }
}
```

希尔排序

```
/*
函数名:       ShellSort
功能:         希尔排序
模板参数说明:T必须支持小于操作
参数说明:     data待排序数组, size待排序数组大小
前置条件:     data!=NULL, size>0
后置条件:     data按非降序序排列
用法:
int arr[]={10,9,8,4,5,7,6,3,1,4};
```

```
ShellSort(arr, 10);
*/
template <typename T>
void ShellSort (T data[], int size)
{
    int i, j, hCnt, h;
    int array[20], k;
    T temp;
    for (h=1, i=0; h<size; ++i)
    {
        array[i]=h;
        h=3*h+1;
    }
    for (i--; i>=0; --i)
    {
        h=array[i];
        for (hCnt=h; hCnt<2*h; ++hCnt)
        {
            for (j=hCnt; j<size; )
            {
                temp=data[j];
                k=j;
                while (k-h>=0 && temp<data[k-h])
                {
                    data[k]=data[k-h];
                    k-=h;
                }
                data[k]=temp;
                j+=h;
            }
        }
    }
}

/*
函数名:      ShellSort
功能:        希尔排序
模板参数说明:T元素类型，Func函数对象或指针
参数说明:    data待排序数组, size待排序数组大小,f函数对象或指针
前置条件:    data!=NULL, size>0
后置条件:    data按f排列
用法:
bool cmp(int a, int b)
{ return a<b; }
```

```
int arr[]={10,9,8,4,5,7,6,3,1,4};
ShellSort(arr, 10, cmp);
*/
template <typename T, typename Func>
void ShellSort (T data[], int size, Func f)
{
    int i, j, hCnt, h;
    int array[20], k;
    T temp;
    for (h=1, i=0; h<size; ++i)
    {
        array[i]=h;
        h=3*h+1;
    }
    for (i--; i>=0; --i)
    {
        h=array[i];
        for (hCnt=h; hCnt<2*h; ++hCnt)
        {
            for (j=hCnt; j<size; )
            {
                temp=data[j];
                k=j;
                while (k-h>=0 && f(temp,data[k-h]))
                {
                    data[k]=data[k-h];
                    k-=h;
                }
                data[k]=temp;
                j+=h;
            }
        }
    }
}
```

随机化快速排序

```
/*
函数名:        quick_sort
功能:          快速排序辅助过程
*/
template <typename T>
void quick_sort (T data[], int frist, int last)
```

```
{
    int lower=frist+1;
    int upper=last;
    int t=rand()%(last-frist)+frist;
    std::swap(data[frist], data[t]);
    T& bound=data[frist];
    while (lower<=upper)
    {
        while (data[lower]<bound)
        {
            ++lower;
        }
        while (bound<data[upper])
        {
            --upper;
        }
        if (lower<upper)
        {
            std::swap(data[lower], data[upper]);
            ++lower;
            --upper;
        }
        else
            ++lower;
    }
    std::swap(data[upper], data[frist]);
    if (frist<upper-1)
        quick_sort(data, frist, upper-1);
    if (upper+1<last)
        quick_sort(data, upper+1, last);
}

/*
函数名:       QuickSort
功能:         快速排
模板参数说明：T必须支持小于操作
参数说明:     data待排序数组, size待排序数组大小
前置条件:     data!=NULL, size>0
后置条件:     data按f排列
用法:
#include <algorithm>
#include <stdlib.h>
#include <time.h>
int arr[]={10,9,8,4,5,7,6,3,1,4};
```

```
QuickSort(arr, 10);
*/
template <typename T>
void QuickSort (T data[], int size)
{
    int i, max;
    if (size<2)
        return;
    for (i=1, max=0; i<size; ++i)
    {
        if (data[max]<data[i])
            max=i;
    }
    std::swap(data[size-1], data[max]);
    srand(time(0));
    quick_sort(data, 0, size-2);
}



/*
函数名:      quick_sort
功能:        快速排序辅助过程
*/
template <typename T, typename Func>
void quick_sort (T data[], int frist, int last, Func& f)
{
    int lower=frist+1;
    int upper=last;
    int t=rand()%(last-frist)+frist;
    std::swap(data[frist], data[t]);
    T& bound=data[frist];
    while (lower<=upper)
    {
        while (f(data[lower],bound))
            ++lower;
        while (f(bound,data[upper]))
            --upper;
        if (lower<upper)
        {
            std::swap(data[lower], data[upper]);
            ++lower;
            --upper;
        }
        else
            ++lower;
    }
    std::swap(data[upper], data[frist]);
    if (frist<upper-1)
        quick_sort(data, frist, upper-1, f);
    if (upper+1<last)
        quick_sort(data, upper+1, last, f);
}

/*
函数名:       QuickSort
功能:         快速排
模板参数说明: T元素类型，Func函数对象或指针
参数说明:     data待排序数组, size待排序数组大小,f函数对象或指针
前置条件:     data!=NULL, size>0
后置条件:     data按f排列
用法:
#include <algorithm>
#include <stdlib.h>
#include <time.h>
bool cmp(int a, int b)
{ return a<b; }
int arr[]={10,9,8,4,5,7,6,3,1,4};
QuickSort(arr, 10, cmp);
*/
template <typename T, typename Func>
void QuickSort (T data[], int size, Func f)
{
    int i, max;
    if (size<2)
        return;
    for (i=1, max=0; i<size; ++i)
    {
        if (f(data[max],data[i]))
            max=i;
    }
    std::swap(data[size-1], data[max]);
    srand(time(0));
    quick_sort(data, 0, size-2, f);
}
```

归并排序

```
/*
函数名:	   MergeSort
功能:	   归并排序
模板参数说明:T必须支持小于操作
参数说明:	   data待排序数组, size待排序数组大小
前置条件:	   data!=NULL, size>0
后置条件:	   data按非降序排列
用法:
#include <algorithm>
int arr[]={10,9,8,4,5,7,6,3,1,4};
MergeSort(arr, 10);
*/
template <typename T>
void MergeSort (T data[], int size)
{
    if ( size>1 )
    {
        //预处理
        int mid=size/2;
        int numOfleft=mid;
        int numOfright=size-mid;
        T* left=new T[numOfleft];
        T* right=new T[numOfright];
        //分
        std::copy(data, data+numOfleft, left);
        std::copy(data+numOfleft, data+size, right);
        MergeSort(left, numOfleft);
        MergeSort(right, numOfright);
        //合
        std::merge(left, left+numOfleft, right, right+numOfright, data);
        //清理
        delete[] left;
        delete[] right;
    }
}

/*
函数名:	   MergeSort
功能:	   归并排序
模板参数说明:T元素类型, Func函数对象或指针
参数说明:	   data待排序数组, size待排序数组大小,f函数对象或指针
```

```
前置条件:      data!=NULL, size>0
后置条件:      data按f排列
用法:
#include <algorithm>
bool cmp(int a, int b)
{ return a<b; }
int arr[]={10,9,8,4,5,7,6,3,1,4};
MergeSort(arr, 10, cmp);
*/
template <typename T, typename Func>
void MergeSort (T data[], int size, Func f)
{
    if ( size>1 )
    {
        int mid=size/2;
        int numOfleft=mid;
        int numOfright=size-mid;
        T* left=new T[numOfleft];
        T* right=new T[numOfright];
        std::copy(data, data+numOfleft, left);
        std::copy(data+numOfleft, data+size, right);
        MergeSort(left, numOfleft, f);
        MergeSort(right, numOfright, f);
        std::merge(left, left+numOfleft, right, right+numOfright, data,
f);
        delete[] left;
        delete[] right;
    }
}
```

堆排序

```
/*
函数名:        heap_down
功能:          堆排序辅助过程
*/
template <typename T>
void heap_down (T data[], int i, const int& size)
{
    int p=i*2+1;
    while ( p<size )
    {
        if ( p+1<size )
        {
            if ( data[p]<data[p+1] )
                ++p;
        }
        if ( data[i]<data[p] )
        {
            std::swap(data[p], data[i]);
            i=p;
            p=i*2+1;
        }
        else
            break;
    }
}
/*
函数名:        HeapSort
功能:          堆排序
模板参数说明:T必须支持小于操作
参数说明:      data待排序数组, size待排序数组大小
前置条件:      data!=NULL, size>0
后置条件:      data按非降序排列
用法:
#include <algorithm>
int arr[]={10,9,8,4,5,7,6,3,1,4};
MergeSort(arr, 10);
*/
template <typename T>
void HeapSort (T data[], int size)
{
    int i;
    for (i=(size-1)/2; i>=0; --i)
        heap_down(data, i, size);
    for (i=size-1; i>0; --i)
    {
        std::swap(data[0], data[i]);
        heap_down(data, 0, i);
    }
}



/*
函数名:        heap_down
功能:          堆排序辅助过程
```

```
*/
template <typename T, typename Func>
void heap_down (T data[], int i, const int& size, Func& f)
{
    int p=i*2+1;
    while ( p<size )
    {
        if ( p+1<size )
        {
            if ( f(data[p],data[p+1]) )
                ++p;
        }
        if ( f(data[i],data[p]) )
        {
            std::swap(data[p], data[i]);
            i=p;
            p=i*2+1;
        }
        else
            break;
    }
}
/*
函数名:       HeapSort
功能:         堆排序
模板参数说明: T元素类型, Func函数对象或指针
参数说明:     data待排序数组, size待排序数组大小,f函数对象或指针
前置条件:     data!=NULL, size>0
后置条件:     data按f排列
用法:
#include <algorithm>
bool cmp(int a, int b)
{ return a<b; }
int arr[]={10,9,8,4,5,7,6,3,1,4};
HeapSort(arr, 10, cmp);
*/
template <typename T, typename Func>
void HeapSort (T data[], int size, Func f)
{
    int i;
    for (i=(size-1)/2; i>=0; --i)
        heap_down(data, i, size, f);
    for (i=size-1; i>0; --i)
    {
        std::swap(data[0], data[i]);
        heap_down(data, 0, i, f);
    }
}
```

大整数处理

包含头文件

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <algorithm>
```

定义

```
//大整数类
class BigInteger
{
public:
    //默认构造
    explicit BigInteger(const std::string& str="0") : m_str(str)
{Normal();}
    //接受双精度浮点构造
    explicit BigInteger(double d)
    {
        m_str=ToString(d);
        Normal();
    }
    //接受单精度浮点构造
    explicit BigInteger(float f)
    {
        m_str=ToString(f);
        Normal();
    }
    //接受整数构造
    explicit BigInteger(int i)
    {
        m_str=ToString(i);
        Normal();
    }
    //输入流运算
    friend std::ostream& operator<< (std::ostream& out, const 
BigInteger& big);
    //输出流运算
    friend std::istream& operator>> (std::istream& in, BigInteger& big);
    //赋值
    BigInteger& operator= (const BigInteger& rbs);
    BigInteger& operator= (double d);
    BigInteger& operator= (float f);
    BigInteger& operator= (int i);
    //高精度乘法
    BigInteger operator* (const BigInteger& other)
    { return signMultiply(*this, other);}
    //高精度加法
    BigInteger operator +(const BigInteger& other)
    { return signAdd(*this, other);}
    //高精度减法
    BigInteger operator- (const BigInteger& other)
    { return signMinuse(*this, other);}
    //高精度乘方
    BigInteger operator ^(int n)
    { return BigInteger(pow(m_str, n));}
    //高精度正数取模
    BigInteger operator% (const BigInteger& other);
    //比较
    bool operator< (const BigInteger& rbs)
    { return signCompare(*this, rbs)==-1; }
    bool operator== (const BigInteger& rbs)
    { return signCompare(*this, rbs)==0; }
    //转换成字符串
    std::string ToString();
private:
    //规范化
    void Normal();
    void Unnormal();
    //转换
    std::string ToString(int n);
    std::string ToString(double n);
    std::string ToString(float n);
    //有符号高精度运算及其比较
    BigInteger signMultiply(const BigInteger& l, const BigInteger& r);
    BigInteger signAdd(const BigInteger& l, const BigInteger& r);
    BigInteger signMinuse(const BigInteger& l, const BigInteger& r);
    BigInteger signPow(const BigInteger& x, int n);
    int signCompare(const BigInteger& a, const BigInteger& b);
    //无符号比较
    int Compare(const std::string& a, const std::string& b);
    //无符号高精度运算
    std::string MultiplyEx (std::string s, std::string t);
    std::string Multiply (std::string lbs, std::string rbs);
    std::string AddEx(std::string a, std::string b);
    std::string Add(std::string lbs, std::string rbs);
    std::string MinusEx(std::string a, std::string b, bool& sign);
    std::string Minuss (std::string lbs, std::string rbs, bool& sign);
    std::string pow(const std::string& b, int n);
    std::string mod(std::string s, const std::string& t);
    //判断s是否全为零
    bool isZero(const std::string& s);
private:
    bool m_sign;        //符号
    std::string m_str; //内部字符串
};
```

实现

流输出

```
std::ostream& operator<< (std::ostream& out, const BigInteger& big)
{
    if (!big.m_sign)
        out.put('-');
    out<<big.m_str;
    return out;
}
```

流输入

```
std::istream& operator>> (std::istream& in, BigInteger& big)
{
    in>>big.m_str;
    big.Normal();
    return in;
}
```

赋值

```
BigInteger& BigInteger::operator= (const BigInteger& rbs)
{
    if ( this!=&rbs )
    {
        m_str=rbs.m_str;
        m_sign=rbs.m_sign;
    }
    return *this;
}

BigInteger& BigInteger::operator= (double d)
{
    m_str=ToString(d);
    Normal();
    return *this;
}

BigInteger& BigInteger::operator= (float f)
{
    m_str=ToString(f);
    Normal();
    return *this;
}

BigInteger& BigInteger::operator= (int i)
{
    m_str=ToString(i);
    Normal();
    return *this;
}

BigInteger BigInteger::operator% (const BigInteger& other)
{
    BigInteger ret;
    ret.m_str=mod(m_str, other.m_str);
    ret.m_sign=true;
    return ret;
}
```

转换函数

```
bool BigInteger::isZero(const std::string& s)
{
    for (size_t i=0; i<s.size(); ++i)
        if (s[i]!='0')
            return false;
    return true;
}

std::string BigInteger::ToString()
{
    std::string s;
    if (!m_sign)
        s.push_back('-');
    return s+m_str;
}

std::string BigInteger::ToString(int n)
{
    static char buf[100];
    sprintf(buf,"%d", n);
    return std::string(buf);
}

std::string BigInteger::ToString(double n)
{
    static char buf[100];
    sprintf(buf,"%f", n);
    return std::string(buf);
}
std::string BigInteger::ToString(float n)
{
    static char buf[100];
    sprintf(buf,"%f", n);
    return std::string(buf);
}
```

规范化符号化

```
void BigInteger::Normal()
{
    if (m_str[0]=='-')
    {
        m_sign=false;
        m_str.erase(0,1);
    }
    else
        m_sign=true;
}

void BigInteger::Unnormal()
{
    if (!m_sign)
        m_str="0"+m_str;
}
```

带符号乘法

```
BigInteger BigInteger::signMultiply(const BigInteger& l, const
BigInteger& r)
{
    BigInteger ret;
    ret.m_sign=!(l.m_sign^r.m_sign);
    ret.m_str=MultiplyEx(l.m_str, r.m_str);
    if (ret.m_str=="0")
        ret.m_sign=true;
    return ret;
}
```

无符号取模

```
std::string BigInteger::mod(std::string s, const std::string& t)
{
    std::string p;
    bool f;
    {
        int size=t.size();
        p=s.substr(0, size);
        s.erase(0, size);
        while (!s.empty())
        {
            while (Compare(p,t)>=0)
                p=Minuss(p,t, f);
            if (p=="0")
            {
                p=s.substr(0, size);
                s.erase(0, size);
                if (isZero(p))
                {
                    p="0";
                    break;
                }
            }
            else
            {
                p+=s.substr(0, 1);
                s.erase(0,1);
            }
        }
        if (p=="0")
            p=s;
        else
            p+=s;
        if (isZero(p))
            p="0";
        while (Compare(p,t)>=0)
            p=Minuss(p,t, f);
    }
    return p;
}
```

整数乘法

```
std::string BigInteger::Multiply(std::string lbs, std::string rbs)
{
    int* g_lbs;
    int* g_rbs;
    int* g_result;
    char buffer[10];
    int lenLbs=lbs.length();
    int lenRbs=rbs.length();
    int sizeLbs=lenLbs%4==0?lenLbs/4:lenLbs/4+1;
    int sizeRbs=lenRbs%4==0?lenRbs/4:lenRbs/4+1;
    g_lbs=new int[sizeLbs+1];
    g_rbs=new int[sizeRbs+1];
    int i,j;
    memset(g_lbs, 0, sizeof(int)*(sizeLbs+1));
    memset(g_rbs, 0, sizeof(int)*(sizeRbs+1));
    std::string str;
```

```
int count=1;
while (lbs.length()>=4)
{
    str=lbs.substr(lbs.length()-4);
    lbs.erase(lbs.length()-4);
    g_lbs[count]=atoi(str.c_str());
    ++count;
}
if (!lbs.empty())
{
    str=lbs;
    lbs.clear();
    g_lbs[count]=atoi(str.c_str());
}
count=1;
while (rbs.length()>=4)
{
    str=rbs.substr(rbs.length()-4);
    rbs.erase(rbs.length()-4);
    g_rbs[count]=atoi(str.c_str());
    ++count;
}
if (!rbs.empty())
{
    str=rbs;
    rbs.clear();
    g_rbs[count]=atoi(str.c_str());
}
g_result=new int[sizeLbs*sizeRbs+2];
memset(g_result, 0, sizeof(int)*(sizeLbs*sizeRbs+2));
for (i=1; i<=sizeLbs; ++i)
{
    for (j=1; j<=sizeRbs; ++j)
    {
        g_result[i+j-1]+=g_lbs[i]*g_rbs[j];
        g_result[i+j]+=g_result[i+j-1]/10000;
        g_result[i+j-1]=g_result[i+j-1]%10000;
    }
}
std::string ret;
i=sizeLbs*sizeRbs+1;
while (!g_result[i])
{
    --i;
```

```
        if (i==0)
        {
            ret="0";
            goto leave;
        }
    }
    sprintf(buffer,"%d", g_result[i--]);
    ret.append(buffer);
    for (j=i; j>=1; --j)
    {
        if (g_result[j]<1000)
            ret.append("0");
        if (g_result[j]<100)
            ret.append("0");
        if (g_result[j]<10)
            ret.append("0");
        sprintf(buffer,"%d", g_result[j]);
        ret.append(buffer);
    }
leave:
    delete[] g_lbs;
    delete[] g_rbs;
    delete[] g_result;
    return ret;
}
```

整数加法

```
std::string BigInteger::Add(std::string lbs, std::string rbs)
{
    int* g_lbs;
    int* g_rbs;
    int* g_result;
    char buffer[10];
    int lenLbs=lbs.length();
    int lenRbs=rbs.length();
    if (lenLbs<lenRbs)
    {
        std::swap(lbs, rbs);
        std::swap(lenLbs, lenRbs);
    }
    int sizeLbs=lenLbs%4==0?lenLbs/4:lenLbs/4+1;
    int sizeRbs=lenRbs%4==0?lenRbs/4:lenRbs/4+1;
    g_lbs=new int[sizeLbs];
```

```
g_rbs=new int[sizeRbs];
int i,j;
memset(g_lbs, 0, sizeof(int)*sizeLbs);
memset(g_rbs, 0, sizeof(int)*sizeRbs);
std::string str;
int count=0;
while (lbs.length()>=4)
{
    str=lbs.substr(lbs.length()-4);
    lbs.erase(lbs.length()-4);
    g_lbs[count]=atoi(str.c_str());
    ++count;
}
if (!lbs.empty())
{
    str=lbs;
    lbs.clear();
    g_lbs[count]=atoi(str.c_str());
}
count=0;
while (rbs.length()>=4)
{
    str=rbs.substr(rbs.length()-4);
    rbs.erase(rbs.length()-4);
    g_rbs[count]=atoi(str.c_str());
    ++count;
}
if (!rbs.empty())
{
    str=rbs;
    rbs.clear();
    g_rbs[count]=atoi(str.c_str());
}
g_result=new int[sizeLbs+1];
memset(g_result, 0, sizeof(int)*(sizeLbs+1));
for (j=0; j<sizeLbs; ++j)
{
    if (j<sizeRbs)
    {
        g_result[j]+=g_lbs[j]+g_rbs[j];
    }
    else
    {
        g_result[j]+=g_lbs[j];
        }
        g_result[j+1]+=g_result[j]/10000;
        g_result[j]%=10000;
    }
    std::string ret;
    i=sizeLbs;
    while (!g_result[i])
    {
        --i;
        if (i==-1)
        {
            ret.append("0");
            goto leave;
        }
    }
    sprintf(buffer, "%d", g_result[i--]);
    ret.append(buffer);
    for (j=i; j>=0; --j)
    {
        if (g_result[j]<1000)
            ret.append("0");
        if (g_result[j]<100)
            ret.append("0");
        if (g_result[j]<10)
            ret.append("0");
        sprintf(buffer, "%d", g_result[j]);
        ret.append(buffer);
    }
leave:
    delete[] g_lbs;
    delete[] g_rbs;
    delete[] g_result;
    return ret;
}
```

带符号加法

```
BigInteger BigInteger::signAdd(const BigInteger& l, const BigInteger& r)
{
    BigInteger ret;
    if (l.m_sign==r.m_sign)
    {
        ret.m_sign=l.m_sign;
        ret.m_str=AddEx(l.m_str, r.m_str);
    }
    else
    {
        bool f;
        if (Compare(l.m_str, r.m_str)<0)
        {
            ret.m_str=MinusEx(r.m_str, l.m_str, f);
            ret.m_sign=r.m_sign;
        }
        else
        {
            ret.m_str=MinusEx(l.m_str, r.m_str, f);
            ret.m_sign=l.m_sign;
        }
    }
    if (ret.m_str=="0")
        ret.m_sign=true;
    return ret;
}
```

浮点乘法

```
std::string BigInteger::MultiplyEx(std::string s, std::string t)
{
    int dots;
    int dott;
    int i;
    std::string ans;
    {
        dots=0;
        for (i=s.size()-1; i>=0; --i)
            if (s[i]=='.')
                break;
            else
                ++dots;
        if (dots<s.size())
            s.erase(i, 1);
        else
            dots=0;
        dott=0;
        for (i=t.size()-1; i>=0; --i)
            if (t[i]=='.')
                break;
            else
                ++dott;
        if (dott<t.size())
            t.erase(i, 1);
        else
            dott=0;
        int dot=dots+dott;
        std::string ret=Multiply(s, t);
        s.clear(); t.clear();
        s=ret;
        if (s.size()!=1)
        {
            if (dot!=0)
            {
                std::reverse(s.begin(), s.end());
                while (s[0]=='0')
                {
                    s.erase(0,1);
                    --dot;
                }
                std::reverse(s.begin(), s.end());
            }
        }
        else
            if (s[0]=='0')
                dot=0;
        int idx=s.size()-dot;
        if (idx<0)
        {
            ans="0.";
            while (idx<0)
            {
                ans.push_back('0');
                ++idx;
            }
            idx=-1;
        }
        for (i=0; i<s.size(); ++i)
        {
            if (i==idx)
            {
                if (i==0)
                    ans.push_back('0');
                ans.push_back('.');
            }
```

```
            ans.push_back(s[i]);
        }
    }
    return ans;
}
```

浮点加法

```
std::string BigInteger::AddEx(std::string a, std::string b)
{
    std::string ah, ab;
    std::string bh, bb;
    int i;
    int size;
    std::string ans;
    {
        for (i=0; i<a.size(); ++i)
            if (a[i]=='.')
                break;
        ah=a.substr(0, i);
        if (i<a.size())
            ab=a.substr(i+1);
        a.clear();
        for (i=0; i<b.size(); ++i)
            if (b[i]=='.')
                break;
        bh=b.substr(0, i);
        if (i<b.size())
            bb=b.substr(i+1);
        b.clear();
        a=Add(ah, bh);
        ah.clear(); bh.clear();
        size=(ab.size()<bb.size() ? bb.size() : ab.size());
        if (ab.size()<size)
        {
            int n=size-ab.size();
            while (n--)
                ab.push_back('0');
        }
        if (bb.size()<size)
        {
            int n=size-bb.size();
            while (n--)
                bb.push_back('0');
        }
        b=Add(ab, bb);
        ab.clear(); bb.clear();
        if (b.size()>size)
        {
            std::string c;
            c.push_back(b[0]);
            b.erase(0,1);
            a=Add(a, c);
        }
        else if (b.size()<size)
        {
            int n=size-b.size();
            while (n--)
                b="0"+b;
        }
        std::reverse(b.begin(), b.end());
        while (!b.empty()&&b[0]=='0')
            b.erase(0, 1);
        std::reverse(b.begin(), b.end());
        ans=a;
        if (!b.empty())
            ans.push_back('.');
        ans.append(b);
    }
    return ans;
}
```

带符号减法

```
BigInteger BigInteger::signMinuse(const BigInteger& l, const
BigInteger& r)
{
    BigInteger ret;
    if (l.m_sign && r.m_sign)
        ret.m_str=MinusEx(l.m_str, r.m_str, ret.m_sign);
    else if (!l.m_sign && !r.m_sign)
    {
        ret.m_str=MinusEx(l.m_str, r.m_str, ret.m_sign);
        ret.m_sign=!ret.m_sign;
    }
    else if (l.m_sign && !r.m_sign)
    {
        ret.m_str=AddEx(l.m_str, r.m_str);
        ret.m_sign=true;
    }
    else
    {
        ret.m_str=AddEx(l.m_str, r.m_str);
        ret.m_sign=false;
    }
    if (ret.m_str=="0")
        ret.m_sign=true;
    return ret;
}
```

整数减法

```
std::string BigInteger::Minuss (std::string lbs, std::string rbs, bool&
sign)
{
    int* g_lbs;
    int* g_rbs;
    int* g_result;
    char buffer[10];
    int lenLbs=lbs.length();
    int lenRbs=rbs.length();
    sign=true;
    if (lenLbs<lenRbs)
        sign=false;
    else if (lenLbs==lenRbs)
    {
        sign=(lbs>=rbs);
        if (!sign)
        {
            std::swap(lbs, rbs);
            std::swap(lenLbs, lenRbs);
        }
    }
    if (lenLbs<lenRbs)
    {
        std::swap(lbs, rbs);
        std::swap(lenLbs, lenRbs);
    }
    int sizeLbs=lenLbs%4==0?lenLbs/4:lenLbs/4+1;
    int sizeRbs=lenRbs%4==0?lenRbs/4:lenRbs/4+1;
    g_lbs=new int[sizeLbs];
    g_rbs=new int[sizeRbs];
```

```
int i,j;
memset(g_lbs, 0, sizeof(int)*sizeLbs);
memset(g_rbs, 0, sizeof(int)*sizeRbs);
std::string str;
int count=0;
while (lbs.length()>=4)
{
    str=lbs.substr(lbs.length()-4);
    lbs.erase(lbs.length()-4);
    g_lbs[count]=atoi(str.c_str());
    ++count;
}
if (!lbs.empty())
{
    str=lbs;
    lbs.clear();
    g_lbs[count]=atoi(str.c_str());
}
count=0;
while (rbs.length()>=4)
{
    str=rbs.substr(rbs.length()-4);
    rbs.erase(rbs.length()-4);
    g_rbs[count]=atoi(str.c_str());
    ++count;
}
if (!rbs.empty())
{
    str=rbs;
    rbs.clear();
    g_rbs[count]=atoi(str.c_str());
}
g_result=new int[sizeLbs+1];
memset(g_result, 0, sizeof(int)*(sizeLbs+1));
for (j=0; j<sizeLbs; ++j)
{
    if (j<sizeRbs)
    {
        if (g_lbs[j]-g_rbs[j]<0)
        {
            --g_lbs[j+1];
            g_lbs[j]+=10000;
        }
        g_result[j]=g_lbs[j]-g_rbs[j];
        }
        else
        {
            if (g_lbs[j]<0)
            {
                --g_lbs[j+1];
                g_lbs[j]+=10000;
            }
            g_result[j]=g_lbs[j];
        }
    }
    std::string ret;
    i=sizeLbs;
    while (!g_result[i])
    {
        --i;
        if (i==-1)
        {
            ret.append("0");
            goto leave;
        }
    }
    sprintf(buffer, "%d", g_result[i--]);
    ret.append(buffer);
    for (j=i; j>=0; --j)
    {
        if (g_result[j]<1000)
            ret.append("0");
        if (g_result[j]<100)
            ret.append("0");
        if (g_result[j]<10)
            ret.append("0");
        sprintf(buffer, "%d", g_result[j]);
        ret.append(buffer);
    }
leave:
    delete[] g_lbs;
    delete[] g_rbs;
    delete[] g_result;
    return ret;
}
```

浮点减法

```
std::string BigInteger::MinusEx(std::string a, std::string b, bool&
sign)
{
    std::string ah, ab;
    std::string bh, bb;
    int i;
    int size;
    std::string ans;
    {
        sign=true;
        if (Compare(a, b)==-1)
        {
            a.swap(b);
            sign=false;
        }
        for (i=0; i<a.size(); ++i)
            if (a[i]=='.')
                break;
        ah=a.substr(0, i);
        if (i<a.size())
            ab=a.substr(i+1);
        a.clear();
        for (i=0; i<b.size(); ++i)
            if (b[i]=='.')
                break;
        bh=b.substr(0, i);
        if (i<b.size())
            bb=b.substr(i+1);
        b.clear();
        bool f;
        a=Minuss(ah, bh, f);
        ah.clear(); bh.clear();
        size=(ab.size()<bb.size() ? bb.size() : ab.size());
        if (ab.size()<size)
        {
            int n=size-ab.size();
            while (n--)
                ab.push_back('0');
        }
        if (bb.size()<size)
        {
            int n=size-bb.size();
            while (n--)
                bb.push_back('0');
        }
        if (ab<bb)
        {
            if (a!="0")
            {
                a=Minuss(a, "1", f);
                ab="1"+ab;
                b=Minuss(ab, bb, f);
            }
            else
                b=Minuss(ab, bb, f);
        }
        else
            b=Minuss(ab, bb, f);
        ab.clear(); bb.clear();
        if (b.size()<size)
        {
            int n=size-b.size();
            while (n--)
                b="0"+b;
        }
        std::reverse(b.begin(), b.end());
        while (!b.empty()&&b[0]=='0')
            b.erase(0, 1);
        std::reverse(b.begin(), b.end());
        ans=a;
        if (!b.empty())
            ans.push_back('.');
        ans.append(b);
    }
    return ans;
}
```

带符号比较

```
int BigInteger::signCompare(const BigInteger& a, const BigInteger& b)
{
    if (a.m_sign && b.m_sign)
        return Compare(a.m_str, b.m_str);
    else if (!a.m_sign && !b.m_sign)
    {
        int ret=Compare(a.m_str, b.m_str);
        if (ret!=0)
            ret=0-ret;
        return ret;
    }
    else
        return a.m_sign ? a.m_sign : b.m_sign;
}
```

无符号比较

```
int BigInteger::Compare(const std::string& a, const std::string& b)
{
    std::string ah, ab;
    std::string bh, bb;
    int i;
    for (i=0; i<a.size(); ++i)
        if (a[i]=='.')
            break;
    ah=a.substr(0, i);
    ab.clear();
    if (i<a.size())
        ab=a.substr(i+1);
    for (i=0; i<b.size(); ++i)
        if (b[i]=='.')
            break;
    bh=b.substr(0, i);
    bb.clear();
    if (i<b.size())
        bb=b.substr(i+1);

    if (ah.size()<bh.size())
        return -1;
    else if (ah.size()>bh.size())
        return 1;
    else
    {
        if (ah<bh)
            return -1;
        else if (ah>bh)
            return 1;
        else
        {
            if (ab<bb)
```

```
                return -1;
            else if (ab>bb)
                return 1;
            else
                return 0;
        }
    }
}
```

无符号乘方

```
std::string BigInteger::pow(const std::string& b, int n)
{
    if (n==0)
        return std::string("1");
    if ( n==1 )
    {
        return b;
    }
    else
    {
        if ( n%2==0 )
        {
            std::string t=pow(b,n/2);
            return MultiplyEx(t, t);
        }
        else
        {
            std::string t=pow(b,n/2);
            return MultiplyEx(MultiplyEx(t,t),b);
        }
    }
}
```

带符号乘方

```
BigInteger BigInteger::signPow(const BigInteger& x, int n)
{
    BigInteger ret;
    ret.m_sign=true;
    if (!x.m_sign && n%2!=0)
        ret.m_sign=false;
    ret.m_str=pow(x.m_str, n);
    return ret;
}
```

使用方法

```
    std::cout<<"Input any number:\n";
    BigInteger a, b;
    std::cin>>a>>b;
    std::cout<<"a*b="<<a*b<<std::endl;
    std::cout<<"a+b="<<a+b<<std::endl;
    std::cout<<"a-b="<<a-b<<std::endl;
    std::cout<<"Input one of normal positive integer number:\n";
    int n;
    std::cin>>a>>n;
    std::cout<<"a^n="<<(a^n)<<std::endl;
    std::cout<<"All integer number: \n";
    std::cin>>a>>b;
    std::cout<<"a%b="<<(a%b)<<std::endl;
    if (a<b)
        std::cout<<"a less b\n";
    else
        std::cout<<"a not less b\n";
    if (a==b)
        std::cout<<"a equal b\n";
    else
        std::cout<<"a not equal b\n";
```

高级数据结构

普通二叉搜素树

包含头文件

```
#include <stddef.h>
```

定义

```
//---------- begin class BinarySearchTree definition ----------//
template <typename Key, typename Value>
```

```
class BinarySearchTree
{
private:
    //二叉搜索树节点类型
    struct BSTnode
    {
        const Key m_key;        //比较关键字
        Value m_value;           //映射类型
        BSTnode* m_left;        //左子树
        BSTnode* m_right;       //右子树
        BSTnode* m_parent;      //双亲节点
        BSTnode (const Key& k, const Value& v);
        //撤销结构
        void Destroy ();
    private:
        ~BSTnode (); //保证只能在堆上创建对象
    }; //end of struct BSTnode
public:
    //二叉搜索树访问引用类型
    class Node
    {
    public:
        explicit Node (typename BinarySearchTree<Key, Value>::BSTnode*
target=NULL);
        bool IsNull () const;   //判断节点是否为空
        bool IsRoot () const;   //判断节点是否为根
        Node GetLeft () const;  //返回左子树
        Node GetRight () const; //返回右子树
        Node GetParent () const;//返回父节点
        void MoveLeft ();       //移动到左子树
        void MoveRight ();      //移动到右子树
        void MoveParent ();     //移动到双亲
        const Key& GetKey () const;      //返回当前节点的键值
        void SetValue (const Value& v);  //设置当前节点数据
        Value& GetValue () const;         //返回当前节点数据
        BSTnode* getSelf () const        //算法实现内部调用不要操作此函
数
        {
            return m_subTree;
        }
    private:
        //引用的子树指针
        typename BinarySearchTree<Key, Value>::BSTnode* m_subTree;
    }; //end of class Node
```

```
public:
    //构建操作
    BinarySearchTree ();
    BinarySearchTree (const BinarySearchTree<Key, Value>& other);
   BinarySearchTree<Key, Value>& operator= (const BinarySearchTree<Key,
Value>& other);
    //拆除操作
    ~BinarySearchTree ();
    //变异性操作
    void InsertItem (const Key& k, const Value& v); //插入新的元素
    void DeleteItem (const Key& k);                 //删除特定的元素
    bool InsertSubTree (typename BinarySearchTree<Key, Value>::Node&
subTree); //插入子树
    void DeleteSubTree (typename BinarySearchTree<Key, Value>::Node&
subTree); //删除子树
    //复制一颗新子树
    BinarySearchTree<Key, Value> CopySubTree (typename
BinarySearchTree<Key, Value>::Node& subTree);
    //访问性操作
    Node GetRoot () const;    //获得根访问对象
    Node Find (const Key& k); //查找特定元素
    //返回树的高度
    unsigned int GetDepth (typename BinarySearchTree<Key, Value>::Node&
subTree);
    //返回叶子节点个数
   void CountLeaf (typename BinarySearchTree<Key, Value>::Node& subTree,
int& outSum);
    //遍历操作前、中、后序遍历
    template <typename Visit>
    void PreOrder (typename BinarySearchTree<Key, Value>::Node& subTree,
Visit visitor);
    template <typename Visit>
    void InOrder (typename BinarySearchTree<Key, Value>::Node& subTree,
Visit visitor);
    template <typename Visit>
   void PostOrder (typename BinarySearchTree<Key, Value>::Node& subTree,
Visit visitor);
private:
    //内部辅助实现
    static typename BinarySearchTree<Key, Value>::BSTnode* CreateNode
(const Key& k, const Value& v);
    static void DeleteNode (typename BinarySearchTree<Key,
Value>::BSTnode* n);
    typename BinarySearchTree<Key, Value>::BSTnode* copySubTree
```

```
(typename BinarySearchTree<Key, Value>::BSTnode* subtree);
    void deleteSubTree (typename BinarySearchTree<Key, Value>::BSTnode*
subtree);
    //travese
    template <typename Visit>
    void InOrder (BSTnode* subtree, Visit& visitor);
    template <typename Visit>
    void PreOrder (BSTnode* subtree, Visit& visitor);
    template <typename Visit>
    void PostOrder (BSTnode* subtree, Visit& visitor);
    //depth
    unsigned int GetDepth (typename BinarySearchTree<Key,
Value>::BSTnode* subtree);
    //leaf
    void CountLeaf (typename BinarySearchTree<Key, Value>::BSTnode*
subtree, int& outSum);
private:
    typename BinarySearchTree<Key, Value>::BSTnode* m_root;  //树根指
针
}; //end of class BinarySearchTree
//----------end class BinarySearchTree definition ----------//
```

实现

```
//----------begin struct BSTnode----------//
template <typename Key, typename Value>
BinarySearchTree<Key, Value>::BSTnode::BSTnode (const Key& k, const
Value& v)
: m_key(k), m_value(v), m_left(NULL), m_right(NULL), m_parent(NULL)
{
    //do nothing else
}

template <typename Key, typename Value>
inline void BinarySearchTree<Key, Value>::BSTnode::Destroy ()
{
    delete this;
}

template <typename Key, typename Value>
inline BinarySearchTree<Key, Value>::BSTnode::~BSTnode ()
{
    //do nothing else
}
//----------end struct BSTnode----------//

//----------begin class Node----------//
template <typename Key, typename Value>
BinarySearchTree<Key, Value>::Node::Node (typename
BinarySearchTree<Key, Value>::BSTnode* target)
: m_subTree(target)
{
    //do nothing else
}

template <typename Key, typename Value>
inline bool BinarySearchTree<Key, Value>::Node::IsNull () const
{
    return m_subTree==NULL;
}

template <typename Key, typename Value>
inline bool BinarySearchTree<Key, Value>::Node::IsRoot () const
{
    return m_subTree==m_root->m_left;
}

template <typename Key, typename Value>
inline typename BinarySearchTree<Key, Value>::Node BinarySearchTree<Key,
Value>::Node::GetLeft () const
{
    return Node(m_subTree->m_left);
}

template <typename Key, typename Value>
inline typename BinarySearchTree<Key, Value>::Node BinarySearchTree<Key,
Value>::Node::GetRight () const
{
    return Node(m_subTree->m_right);
}

template <typename Key, typename Value>
inline typename BinarySearchTree<Key, Value>::Node BinarySearchTree<Key,
Value>::Node::GetParent () const
{
    return Node(m_subTree->m_parent);
```

```
}

template <typename Key, typename Value>
inline void BinarySearchTree<Key, Value>::Node::MoveLeft ()
{
   m_subTree=m_subTree->m_left;
}

template <typename Key, typename Value>
inline void BinarySearchTree<Key, Value>::Node::MoveRight ()
{
   m_subTree=m_subTree->m_right;
}

template <typename Key, typename Value>
inline void BinarySearchTree<Key, Value>::Node::MoveParent ()
{
   m_subTree=m_subTree->m_parent;
}

template <typename Key, typename Value>
inline const Key& BinarySearchTree<Key, Value>::Node::GetKey () const
{
   return m_subTree->m_key;
}

template <typename Key, typename Value>
inline void BinarySearchTree<Key, Value>::Node::SetValue (const Value&
v)
{
   m_subTree->m_value=v;
}

template <typename Key, typename Value>
inline Value& BinarySearchTree<Key, Value>::Node::GetValue () const
{
   return m_subTree->m_value;
}
//----------end class Node----------//

//----------begin class BinarySearchTree ----------//
template <typename Key, typename Value>
inline typename BinarySearchTree<Key, Value>::BSTnode*
BinarySearchTree<Key, Value>::CreateNode (const Key& k, const Value& v)
```

```
{
    return new BSTnode(k, v);
}

template <typename Key, typename Value>
inline void BinarySearchTree<Key, Value>::DeleteNode (typename
BinarySearchTree<Key, Value>::BSTnode* n)
{
    n->Destroy();
}
```

删树

```
template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::deleteSubTree (typename
BinarySearchTree<Key, Value>::BSTnode* subtree)
{
    if (subtree!=NULL)
    {
        deleteSubTree(subtree->m_left);
        deleteSubTree(subtree->m_right);
        DeleteNode(subtree);
    }
}

template <typename Key, typename Value>
BinarySearchTree<Key, Value>::BinarySearchTree ()
{
    m_root=CreateNode(Key(), Value());
}

template <typename Key, typename Value>
BinarySearchTree<Key, Value>::~BinarySearchTree ()
{
    deleteSubTree(m_root);
}
```

插入元素到树

```
template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::InsertItem (const Key& k, const
Value& v)
{
   BSTnode* tmp=CreateNode(k, v);
   if (m_root->m_left==NULL)
   {
       tmp->m_parent=m_root;
       m_root->m_left=tmp;
       return;
   }
   BSTnode* root=m_root->m_left;
   BSTnode* parent;
   while (root!=NULL)
   {
       parent=root;
       if (root->m_key==k)
       {
           root->m_value=v;
           DeleteNode(tmp);
           return;
       }
       else if (root->m_key<k)
           root=root->m_right;
       else
           root=root->m_left;
   }
   if (parent->m_key<k)
       parent->m_right=tmp;
   else
       parent->m_left=tmp;
   tmp->m_parent=parent;
}

template <typename Key, typename Value>
inline typename BinarySearchTree<Key,Value>::Node BinarySearchTree<Key,
Value>::GetRoot () const
{
   return Node(m_root->m_left);
}

template <typename Key, typename Value>
template <typename Visit>
inline void BinarySearchTree<Key, Value>::InOrder (typename
BinarySearchTree<Key, Value>::Node& subTree, Visit visitor)
{
   InOrder(subTree.getSelf(), visitor);
}
```

```
template <typename Key, typename Value>
template <typename Visit>
void BinarySearchTree<Key, Value>::InOrder (BSTnode* subtree, Visit&
visitor)
{
    if (subtree!=NULL)
    {
        InOrder(subtree->m_left, visitor);
        visitor(subtree->m_value);
        InOrder(subtree->m_right, visitor);
    }
}

template <typename Key, typename Value>
template <typename Visit>
inline void BinarySearchTree<Key, Value>::PreOrder (typename
BinarySearchTree<Key, Value>::Node& subTree, Visit visitor)
{
    PreOrder(subTree.getSelf(), visitor);
}

template <typename Key, typename Value>
template <typename Visit>
void BinarySearchTree<Key, Value>::PreOrder (BSTnode* subtree, Visit&
visitor)
{
    if (subtree!=NULL)
    {
        visitor(subtree->m_value);
        PreOrder(subtree->m_left, visitor);
        PreOrder(subtree->m_right, visitor);
    }
}

template <typename Key, typename Value>
template <typename Visit>
inline void BinarySearchTree<Key, Value>::PostOrder (typename
BinarySearchTree<Key, Value>::Node& subTree, Visit visitor)
{
    PostOrder(subTree.getSelf(), visitor);
}

template <typename Key, typename Value>
template <typename Visit>
void BinarySearchTree<Key, Value>::PostOrder (BSTnode* subtree, Visit&
visitor)
{
    if (subtree!=NULL)
    {
        PostOrder(subtree->m_left, visitor);
        PostOrder(subtree->m_right, visitor);
        visitor(subtree->m_value);
    }
}
```

复制树

```
template <typename Key, typename Value>
typename BinarySearchTree<Key, Value>::BSTnode* BinarySearchTree<Key,
Value>::copySubTree (typename BinarySearchTree<Key, Value>::BSTnode*
subtree)
{
    BSTnode* left=NULL, *right=NULL, *parent=NULL;
    if (subtree==NULL)
        return NULL;
    if (subtree->m_left==NULL)
        left=NULL;
    else
        left=copySubTree(subtree->m_left);
    if (subtree->m_right==NULL)
        right=NULL;
    else
        right=copySubTree(subtree->m_right);
    parent=CreateNode(subtree->m_key, subtree->m_value);
    parent->m_left=left;
    parent->m_right=right;
    if (left!=NULL)
        left->m_parent=parent;
    if (right!=NULL)
        right->m_parent=parent;
    return parent;
}

template <typename Key, typename Value>
BinarySearchTree<Key, Value>::BinarySearchTree (const
BinarySearchTree<Key, Value>& other)
{
    m_root=CreateNode(Key(), Value());
    if (other.m_root->m_left!=NULL)
        m_root->m_left=copySubTree(other.m_root->m_left);
}

template <typename Key, typename Value>
BinarySearchTree<Key, Value>& BinarySearchTree<Key, Value>::operator=
(const BinarySearchTree<Key, Value>& other)
{
    if (this!=&other)
    {
        deleteSubTree(m_root->m_left);
        m_root-m_left=copySubTree(other.m_root->m_left);
    }
    return *this;
}

template <typename Key, typename Value>
bool BinarySearchTree<Key, Value>::InsertSubTree (typename
BinarySearchTree<Key, Value>::Node& subTree)
{
    BSTnode* subtree=subTree.getSelf();
    bool bleft=true;
    if (subtree->m_parent!=NULL)
    {
        if (subtree->m_parent->m_left==subtree)
            subtree->m_parent->m_left=NULL;
        else
        {
            bleft=false;
            subtree->m_parent->m_right=NULL;
        }
    }
    if (m_root->m_left==NULL) //对于树为空时
    {
        subtree->m_parent=m_root;
        m_root->m_left=subtree;
        return true;
    }
    BSTnode* root=m_root->m_left;
    BSTnode* parent;
    while (root!=NULL)
    {
        parent=root;
        if (root->m_key==k)
        {
            if (subtree->m_parent!=NULL)
            {
                if (bleft)
                    subtree->m_parent->m_left=subtree;
                else
                    subtree->m_parent->m_right=subtree;
            }
            return false;
        }
        else if (root->m_key<k)
            root=root->m_right;
        else
            root=root->m_left;
    }
    if (parent->m_key<k)
        parent->m_right=subtree;
    else
        parent->m_left=subtree;
    subtree->m_parent=parent;
    return true;
}

template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::DeleteSubTree (typename
BinarySearchTree<Key, Value>::Node& subTree)
{
    BSTnode* subtree=subTree.getSelf();
    if (subtree->m_parent!=NULL)
    {
        if (subtree->m_parent->m_left==subtree)
            subtree->m_parent->m_left=NULL;
        else
            subtree->m_parent->m_right=NULL;
    }
    deleteSubTree(subtree);
    subTree=Node(NULL);
}

template <typename Key, typename Value>
inline BinarySearchTree<Key, Value> BinarySearchTree<Key,
Value>::CopySubTree (typename BinarySearchTree<Key, Value>::Node&
subTree)
```

```
{
    BinarySearchTree<Key, Value> tmp;
    tmp.m_root->m_left=copySubTree(subTree->getSelf());
    return tmp;
}

template <typename Key, typename Value>
typename BinarySearchTree<Key, Value>::Node BinarySearchTree<Key,
Value>::Find (const Key& k)
{
    BSTnode* root=m_root->m_left;
    while (root!=NULL)
    {
        if (root->m_key==k)
            return Node(root);
        else if (root->m_key<k)
            root=root->m_right;
        else
            root=root->m_left;
    }
    return Node(NULL);
}

template <typename Key, typename Value>
inline unsigned int BinarySearchTree<Key, Value>::GetDepth (typename
BinarySearchTree<Key, Value>::Node& subTree)
{
    return GetDepth(subTree.getSelf());
}
```

求树的高度

```
template <typename Key, typename Value>
unsigned int BinarySearchTree<Key, Value>::GetDepth (typename
BinarySearchTree<Key, Value>::BSTnode* subtree)
{
    unsigned int left=0, right=0, root=0;
    if (subtree==NULL)
        return 0;
    left=GetDepth(subtree->m_left);
    right=GetDepth(subtree->m_right);
    root=( left>right ? left : right )+1;
    return root;
}
```

```
template <typename Key, typename Value>
inline void BinarySearchTree<Key, Value>::CountLeaf (typename
BinarySearchTree<Key, Value>::Node& subTree, int& outSum)
{
    outSum=0;
    CountLeaf(subTree.getSelf(), outSum);
}
```

求叶子的个数

```
template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::CountLeaf (typename
BinarySearchTree<Key, Value>::BSTnode* subtree, int& outSum)
{
    if (subtree->m_left==NULL && subtree->m_right==NULL)
    {
        ++outSum;
        return;
    }
    if (subtree->m_left!=NULL)
        CountLeaf(subtree->m_left, outSum);
    if (subtree->m_right!=NULL)
        CountLeaf(subtree->m_right, outSum);
}
```

删除元素

```
template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::DeleteItem (const Key& k)
{
    if (m_root->m_left==NULL)
        return;
    BSTnode* root=m_root->m_left; //待删除的节点
    BSTnode* parent=m_root; //待删除节点的父节点
    while ( root!=NULL )
    {
        if ( root->m_key==k )
            break;
        else if ( root->m_key<k )
            root=root->m_right;
        else
            root=root->m_left;
```

```
}
if ( root==NULL )
    return;
else
    parent=root->m_parent;
if ( root->m_left==NULL && root->m_right==NULL )
{
    if ( parent->m_left==root )
        parent->m_left=NULL;
    else
        parent->m_right=NULL;
}
else if ( root->m_left!=NULL && root->m_right!=NULL )
{
    BSTnode* left=root->m_left; //待删除节点左子树根
    BSTnode* pleft=left;  //左子树最大值节点 新根
    BSTnode* ppleft=NULL; //左子树最大值节点的父节点
    while ( pleft->m_right!=NULL )
        pleft=pleft->m_right;
    ppleft=pleft->m_parent;
    pleft->m_right=root->m_right;
    if ( root->m_right!=NULL )
        root->m_right->m_parent=pleft;
    if ( pleft==left )
    {
        if ( parent!=root )
        {
            if ( parent->m_left==root )
                parent->m_left=left;
            else
                parent->m_right=left;
            left->m_parent=parent;
        }
        else
        {
            m_root->m_left=left;
            left->m_parent=m_root;
        }
    }
    else
    {
        ppleft->m_right=pleft->m_left;
        if ( pleft->m_left!=NULL )
            pleft->m_left->m_parent=ppleft;
            pleft->m_left=left;
            left->m_parent=pleft;
            if ( parent!=root )
            {
                if ( parent->m_left==root )
                    parent->m_left=pleft;
                else
                    parent->m_right=pleft;
                pleft->m_parent=parent;
            }
            else
                m_root->m_left=pleft;
                pleft->m_parent=m_root;
        }
    }
    else if ( root->m_left!=NULL )
    {
        if ( parent->m_left==root )
            parent->m_left=root->m_left;
        else
            parent->m_right=root->m_left;
        root->m_left->m_parent=parent;
    }
    else
    {
        if ( parent->m_left==root )
            parent->m_left=root->m_right;
        else
            parent->m_right=root->m_right;
        root->m_right->m_parent=parent;
    }
    DeleteNode(root);
}
```

使用方法

```
#include <iostream>
#include <stddef.h>
#include <stdlib.h>
#include <time.h>
void print(const int& t)
{
    std::cout<<t<<" ";
}
```

```
    srand(time(0));
    BinarySearchTree<int, int> trees;
    int i, t;
    for (i=0; i<100; ++i)
    {
        t=rand()%1000;
        trees.InsertItem(t, t);
    }
    BinarySearchTree<int,int>::Node root=trees.GetRoot();
    std::cout<<"InOrder: ";
    trees.InOrder(root, print);
    std::cout<<std::endl;
    int del;
    std::cout<<"Delete: ";
    std::cin>>del;
    trees.DeleteItem(del);
    std::cout<<"InOrder: ";
    trees.InOrder(root, print);
    std::cout<<std::endl;
    trees.CountLeaf(root, t);
    std::cout<<"Leaf: "<<t<<std::endl;
    std::cout<<"Depth: "<<trees.GetDepth(root)<<std::endl;
```

基本线段树模式

```
#include <iostream>

//线段树结构
struct  LS
{
    int nl;       //下界
    int nr;       //上界
    int cover;    //当前线段覆盖次数
    int m;        //侧度
    LS* left;     //左孩子
    LS* right;    //右孩子
    //构建线段树
    void Create(int l, int r);
    //插入线段
    void Insert(int l, int r);
    //删除线段
    void Delete(int l, int r);
};
```

```
#define SIZE 10000

LS tree[SIZE]={0}; //存储空间
int total; //用到的空间

void LS::Create(int l, int r)
{
    nl=l; nr=r;
    //元线段
    if ( l+1==r )
    {
        left=NULL;
        right=NULL;
        return;
    }
    int mid=(l+r)/2; //中点
    left=&tree[total++];
    right=&tree[total++];
    //左子树
    left->Create(l, mid);
    //右子树
    right->Create(mid, r);
}

void LS::Insert(int l, int r)
{
    if ( l==nl && nr==r )  //包含
        ++cover;
    else
    {
        if ( left==NULL && right==NULL )
            return;
        int mid=(nl+nr)/2;

        if ( r<=mid ) //左边
            left->Insert(l, r);
        else if ( l>=mid ) //右边
            right->Insert(l, r);
        else
        {
            left->Insert(l, mid);
            right->Insert(mid, r);
        }
    }
}

void LS::Delete(int l, int r)
{
    if ( l==nl && nr==r )  //包含
        --cover;
    else
    {
        if ( left==NULL && right==NULL )
            return;
        int mid=(nl+nr)/2;

        if ( r<=mid ) //左边
            left->Delete(l, r);
        else if ( l>=mid ) //右边
            right->Delete(l, r);
        else
        {
            left->Delete(l, mid);
            right->Delete(mid, r);
        }
    }
}

int int(int argc, char* argv[])
{
    total=1;
    tree[0].Create(0, 10);
    tree[0].Insert(4, 6);
    tree[0].Delete(4, 6);
    for ( int i=0; i<total; ++i )
        std::cout<<tree[i].nl<<' '<<tree[i].nr<<'
'<<tree[i].cover<<'\n';
    return 0;
}
```

基本并查集模式

```
#define  SIZE 1000
int v[SIZE];
int rank[SIZE];

void init(int n)
{
   for (int i=1; i<=n; ++i)
   {
       v[i]=i;
       rank[i]=0;
   }
}

int find(int p)
{
   if (v[p]==p)
       return p;
   else
   {
       v[p]=find(v[p]);
       return v[p];
   }
}

void judge(int a, int b)
{
   int fa, fb;
   fa=find(a);
   fb=find(b);
   if (fa==fb)
       return;
   if (rank[fa]>rank[fb])
       v[fb]=fa;
   else
   {
       v[fa]=v[fb];
       if (rank[fa]=rank[fb])
           ++rank[fb];
   }
}
```

散列实现的一种方式参考

定义与实现

```
//开放定址策略
template <typename Key,  //键值类型
typename Func> //必须是仿函数类
class OpenAddressPolicy
```

```
{
public:
    OpenAddressPolicy();
    ~OpenAddressPolicy();
    void Init(int size);  //初始化散列表大小
    void Clear();         //清空散列表
    int Count() const;    //计算当前散列表中元素个数
    bool Insert (const Key& k);  //插入键值
    bool Lookup (const Key& k);  //查找是否存在键值
private:
    //用于开放定址策略的数据结构
    Key* m_pData;   //一个存放键值的数组
    bool* m_bUsage; //一个标志对应位置是否使用的数组
    int m_nSize;    //散列表大小
};

template <typename Key, typename Func>
OpenAddressPolicy<Key, Func>::OpenAddressPolicy()
{
    m_pData=NULL;
    m_bUsage=NULL;
    m_nSize=0;
}

template <typename Key, typename Func>
OpenAddressPolicy<Key, Func>::~OpenAddressPolicy()
{
    if ( m_pData!=NULL )
    {
        delete[] m_pData;
        m_pData=NULL;
    }
    if ( m_bUsage!=NULL )
    {
        delete[] m_bUsage;
        m_bUsage=NULL;
    }
    m_nSize=0;
}

template <typename Key, typename Func>
void OpenAddressPolicy<Key, Func>::Init(int size)
{
    m_pData=new Key[size];
    m_bUsage=new bool[size];
    m_nSize=size;
    Clear();
}

template <typename Key, typename Func>
void OpenAddressPolicy<Key, Func>::Clear()
{
    //将使用标志写为假
    for ( int i=0; i<m_nSize; ++i )
        m_bUsage[i]=false;
}

template <typename Key, typename Func>
int OpenAddressPolicy<Key, Func>::Count() const
{
    int count=0;
    //统计使用位置个数
    for ( int i=0; i<m_nSize; ++i )
    {
        if ( m_bUsage[i] )
            ++count;
    }
    return count;
}

template <typename Key, typename Func>
bool OpenAddressPolicy<Key, Func>::Insert(const Key &k)
{
    int pos=Func()(k); //得到散列位置
    if ( m_bUsage[pos] ) //发生冲突
    {
        int i=pos+1; //冲突后一个位置
        //向后找第一个没有使用的位置
        while ( i<m_nSize )
        {
            if ( !m_bUsage[i] )
            {
                m_bUsage[i]=true;
                m_pData[i]=k;
                return true;
            }
            ++i;
        };
        //在冲突位置前找第一个没有使用的位置
        for ( i=0; i<pos; ++i )
        {
            if ( !m_bUsage[i] )
            {
                m_bUsage[i]=true;
                m_pData[i]=k;
                return true;
            }
        }
        //无法找到空位置
        return false;
    }
    else //没有冲突
    {
        m_bUsage[pos]=true;
        m_pData[pos]=k;
    }
    return true;
}

template <typename Key, typename Func>
bool OpenAddressPolicy<Key, Func>::Lookup(const Key &k)
{
    int pos=Func()(k); //得到散列位置
    if ( m_bUsage[pos] )  //可能存在或冲突
    {
        if ( m_pData[pos]==k ) //直接映射成功
            return true;
        else //可能冲突
        {
            int i;
            //遍历冲突位置以后
            for ( i=pos+1; i<m_nSize; ++i )
            {
                if ( m_bUsage[i] )
                {
                    if ( m_pData[i]==k )
                        return true;
                }
                else
                    return false;
            }
            //遍历冲突位置之前
            for ( i=0; i<pos; ++i )
            {
                if ( m_bUsage[i] )
                {
                    if ( m_pData[i]==k )
                        return true;
                }
                else
                    return false;
            }
        }
    }
    return false;
}

//溢出链策略
template <typename Key,     //键值类型
typename Func>   //必须是仿函数类
class OverflowChainPolicy
{
protected:
    //内部数据结构
    struct Node  //桶
    {
        Key m_k;       //键值
        Node* m_next; //下一个桶
    };
    struct Chain //链
    {
        Chain (bool use=false, Node* p=NULL ):m_bUsage(use), m_list(p) {}
        bool m_bUsage;  //该位是否使用标志
        Node* m_list;   //桶链
    };
    void DestroyAllNode(); //删除所有桶节点
public:
    OverflowChainPolicy();
    ~OverflowChainPolicy();
    void Init(int size);       //初始化散列表大小
    void Clear();              //清空散列表
    int Count() const;         //计算当前散列表中元素个数
    bool Insert (const Key& k);  //插入键值
    bool Lookup (const Key& k);  //查找是否存在键值
private:
    //用于溢出链策略的数据结构
    Chain* m_pData; //链
    int m_nSize;     //散列表大小
};

template <typename Key, typename Func>
OverflowChainPolicy<Key, Func>::OverflowChainPolicy()
{
    m_pData=NULL;
    m_nSize=0;
}

template <typename Key, typename Func>
OverflowChainPolicy<Key, Func>::~OverflowChainPolicy()
{
    if ( m_pData!=NULL )
    {
        DestroyAllNode();  //先删除桶节点
        delete[] m_pData;
        m_pData=NULL;
        m_nSize=0;
    }
}

template <typename Key, typename Func>
void OverflowChainPolicy<Key, Func>::DestroyAllNode()
{
    Node* p=NULL;
    Node* q=NULL;
    for ( int i=0; i<m_nSize; ++i )
    {
        if ( m_pData[i].m_bUsage ) //存在映射
        {
            p=m_pData[i].m_list;
            //删除所有关联桶
            while ( p!=NULL && p->m_next!=NULL )
            {
                q=p->m_next;
                p->m_next=q->m_next;
                delete q;
            }
            if ( p!=NULL )
            {
                delete p;
                m_pData[i].m_list=NULL;
                m_pData[i].m_bUsage=false;
            }
        }
    }
}

template <typename Key, typename Func>
void OverflowChainPolicy<Key, Func>::Init(int size)
{
    m_pData=new Chain[size];
    m_nSize=size;
}

template <typename Key, typename Func>
void OverflowChainPolicy<Key, Func>::Clear()
{
    DestroyAllNode();
}

template <typename Key, typename Func>
int OverflowChainPolicy<Key, Func>::Count() const
{
    int count=0;
    Node* p=NULL;
    for ( int i=0; i<m_nSize; ++i )
    {
        if ( m_pData[i].m_bUsage ) //存在映射
        {
            p=m_pData[i].m_list;
            //统计桶数，即映射后元素个数
            while ( p!=NULL )
            {
                ++count;
                p=p->m_next;
            }
        }
    }
    return count;
}

template <typename Key, typename Func>
bool OverflowChainPolicy<Key, Func>::Insert(const Key& k)
{
    int pos=Func()(k);  //获得映射地址
    if ( m_pData[pos].m_bUsage ) //冲突
    {
        Node* p=new Node;
        p->m_k=k;
        p->m_next=m_pData[pos].m_list;
        m_pData[pos].m_list=p;
    }
    else //没有冲突
    {
        m_pData[pos].m_bUsage=true;
        Node* p=new Node;
        p->m_k=k;
        p->m_next=NULL;
        m_pData[pos].m_list=p;
    }
    return true; //总会成功
}

template <typename Key, typename Func>
bool OverflowChainPolicy<Key, Func>::Lookup(const Key &k)
{
    int pos=Func()(k); //获得映射位置
    if ( m_pData[pos].m_bUsage )  //存在映射
    {
        Node* p=m_pData[pos].m_list;
        //在关联桶链表中找特定键值
        while ( p!=NULL )
        {
            if ( p->m_k==k )
            {
                return true;
            }
            p=p->m_next;
        }
    }
    return false;
}

//集合哈希表
template <typename Key,  //键值类型
typename Func, //键值类型到位置的映射仿函数类型
template< typename, typename> class ManagerPolicy > //冲突避免策略
class Hash_Set : protected ManagerPolicy<Key, Func>
{
```

```
public:
    explicit Hash_Set (int size);  //初始化散列表大小
    int Count() const;  //统计已经散列的元素个数
    void Clear();       //清空散列表
    bool Insert(const Key& k);  //插入关键值
    bool Lookup(const Key& k);  //找特定关键值是否已经映射
protected:
    //不可访问函数
    Hash_Set (const Hash_Set& other);
    Hash_Set& operator= (const Hash_Set& rbs);
};

template <typename Key, typename Func, template< typename, typename> class
ManagerPolicy >
Hash_Set<Key, Func, ManagerPolicy>::Hash_Set(int size)
{
    ManagerPolicy<Key, Func>::Init(size);
}

template <typename Key, typename Func, template< typename, typename> class
ManagerPolicy >
int Hash_Set<Key, Func, ManagerPolicy>::Count() const
{
    return ManagerPolicy<Key, Func>::Count();
}

template <typename Key, typename Func, template< typename, typename> class
ManagerPolicy >
void Hash_Set<Key, Func, ManagerPolicy>::Clear()
{
    ManagerPolicy<Key, Func>::Clear();
}

template <typename Key, typename Func, template< typename, typename> class
ManagerPolicy >
bool Hash_Set<Key, Func, ManagerPolicy>::Insert(const Key &k)
{
    return ManagerPolicy<Key, Func>::Insert(k);
}

template <typename Key, typename Func, template< typename, typename> class
ManagerPolicy >
bool Hash_Set<Key, Func, ManagerPolicy>::Lookup(const Key& k)
{
    return ManagerPolicy<Key, Func>::Lookup(k);
}

class Cmp1
{
public:
    int operator() (int k)
    {
        return k%101;
    }
};
```

使用方法

```
#include <iostream>   //cout, cin
#include <iterator>   //ostream_iterator
#include <stdlib.h>   //srand rand
#include <time.h>     //time
    Hash_Set<int, Cmp1, OpenAddressPolicy> hs(101);
    int x;
    bool ret;
    for ( int i=0; i<101; ++i )
    {
        x=rand()%RANGE;
        ret=hs.Insert(x);
    }
    std::cout<<hs.Count()<<"\n";
```

堆

包含头文件

```
#include <algorithm>
#include <functional>
```

定义与实现

```
template <typename T, typename Compare=std::less<T> >
class Heap
{
public:
    explicit Heap (int size);
    ~Heap ();
    const T& operator[] (int pos)
    { return m_pdata[pos]; }
    int getSize () const
    { return curr_size; }
    bool isEmpty() const
    { return curr_size==0; }
    bool isFull () const
    { return curr_size==max_size; }
    void insertItem (const T& item);
    T deleteItem ();
    void clear ()
    { curr_size=0; }
private:
    T* m_pdata;
    Compare m_f;
    int max_size;
    int curr_size;
    void FilterDown (int i);
    void FilterUp (int i);
};

template <typename T, typename Compare >
Heap<T, Compare>::Heap(int size)
{
    max_size=size;
    curr_size=0;
    m_pdata=new T[size];
}

template <typename T, typename Compare >
Heap<T, Compare>::~Heap()
{
    delete[] m_pdata;
}

template <typename T, typename Compare >
void Heap<T, Compare>::FilterUp (int i)
{
    int currentpos, parentpos;
    T target;
    currentpos=i;
    parentpos=(i-1)/2;
    target=m_pdata[i];
    while (currentpos)
    {
        if (!(m_f(m_pdata[currentpos],m_pdata[parentpos])))
            break;
        else
        {
            std::swap(m_pdata[parentpos], m_pdata[currentpos]);
            currentpos=parentpos;
            parentpos=(currentpos-1)/2;
        }
    }
    m_pdata[currentpos]=target;
}

template <typename T, typename Compare >
void Heap<T, Compare>::insertItem (const T& item)
{
    if (isFull())
        ;
    else
    {
        m_pdata[curr_size]=item;
        FilterUp(curr_size);
        ++curr_size;
    }
}

template <typename T, typename Compare >
void Heap<T, Compare>::FilterDown (int i)
{
    int currentpos, childpos;
    T target;
    currentpos=i;
    childpos=2*i+1;
    target=m_pdata[i];
    while (childpos<curr_size)
    {
        if ( (childpos+1<curr_size) &&
m_f(m_pdata[childpos+1],m_pdata[childpos]))
            ++childpos;
        if (m_f(target,m_pdata[childpos]))
            break;
        else
        {
            std::swap(m_pdata[currentpos], m_pdata[childpos]);
            currentpos=childpos;
            childpos=2*currentpos+1;
        }
    }
    m_pdata[currentpos]=target;
}

template <typename T, typename Compare >
T Heap<T, Compare>::deleteItem ()
{
    T temp;
    if (isEmpty())
        ;
    else
    {
        temp=m_pdata[0];
        m_pdata[0]=m_pdata[curr_size-1];
        --curr_size;
        FilterDown(0);
    }
    return temp;
}
```

使用方法

```
class cmp
{
public:
    bool operator()(const int& a, const int& b)
    { return a<b;}
};
    Heap<int, cmp> heap(100);
    int i;
    for (i=100; i>0; --i)
        heap.insertItem(i);
    while (!heap.isEmpty())
        std::cout<<heap.deleteItem()<<" ";
```

图相关算法

图的深度优先和广度优先算法举例

```
#include <iostream>
#include <cassert>
#include <stack>
#include <queue>

const short NUM_VERTICE=9;
static short graph[NUM_VERTICE][NUM_VERTICE]=
{
    {0, 1, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 1, 0, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 0, 0, 1},
    {0, 0, 0, 0, 1, 0, 1, 0, 0},
    {0, 0, 0, 0, 0, 1, 0, 1, 1},
    {0, 0, 0, 0, 0, 1, 1, 0, 1},
    {0, 0, 0, 0, 1, 0, 1, 1, 0}
};

static bool old[9]={false};

//深度优先
void DFS (short G[][NUM_VERTICE], int numOfVertice, bool visited[], short
start, void Func(short& vertice))
{
    std::stack<short> S;
    S.push(start);
    int i;
    while (!S.empty())
    {
        start=S.top();
        S.pop();
        Func(start);
        for (i=0; i<numOfVertice; ++i)
        {
            if (G[start][i]!=0 && !visited[i])
            {
                visited[i]=true;
                S.push(i);
            }
        }
    }
}

//广度优先
void BFS (short G[][NUM_VERTICE], int numOfVertice, bool visited[], short
start, void Func(short& vertice))
{
    std::queue<short> Q;
    Q.push(start);
    short i;
    while (!Q.empty())
    {
        start=Q.front();
        Q.pop();
        for (i=0; i<numOfVertice; ++i)
        {
            if (G[start][i]!=0 && !visited[i])
            {
                visited[i]=true;
                Func(i);
                Q.push(i);
            }
        }
    }
}

void show (short& target)
{
    std::cout<<target<<" ";
}

int main (int argc, char* argv[])
{
    short i;
    for (i=0; i<NUM_VERTICE; ++i)
    {
        if (!old[i])
        {
            old[i]=true;
            DFS(graph, NUM_VERTICE, old, i, show);
        }
    }
    std::cout<<std::endl;
    for (i=0; i<NUM_VERTICE; ++i)
    {
        old[i]=false;
    }
    for (i=0; i<NUM_VERTICE; ++i)
    {
        if (!old[i])
        {
            old[i]=true;
            show(i);
            BFS(graph, NUM_VERTICE, old, i, show);
        }
    }
    std::cout<<std::endl;
    return 0;
}
```

无向图最小生成树的 Kruskal 算法举例

```
#include <iostream>
#include <cassert>
#include <queue>

const short NUM_VERTICE=9;
static short graph[NUM_VERTICE][NUM_VERTICE]=
{
    {0, 3, 4, 8, 0, 0, 0, 0, 0},
    {3, 0, 7, 0, 0, 0, 0, 0, 0},
    {4, 7, 0, 0, 0, 0, 0, 0, 0},
    {8, 0, 0, 0, 10, 0, 0, 0, 0},
    {0, 0, 0, 10, 0, 7, 0, 0, 7},
    {0, 0, 0, 0, 7, 0, 1, 2, 0},
    {0, 0, 0, 0, 0, 1, 0, 9, 4},
    {0, 0, 0, 0, 0, 2, 9, 0, 3},
    {0, 0, 0, 0, 7, 0, 4, 3, 0}
};

struct Edge //边结构用于排序
{
    short from;
    short to;
    int weight;
```

```
};

class Cmp
{
public:
    bool operator() (const Edge& lbs, const Edge& rbs)
    {
        return lbs.weight>rbs.weight;
    }
};

void Kruskal (short G[][NUM_VERTICE], int numOfVertice)
{
    assert(G!=NULL);
    assert(numOfVertice>0);
    std::priority_queue<Edge, std::vector<Edge>, Cmp> PQ; //最小堆的优先队列
    int i,j;
    Edge temp;
    //将图中的边转化成边结构放进优先队列
    for (i=0; i<numOfVertice; ++i)
    {
        for (j=0; j<numOfVertice; ++j)
        {
            if (G[i][j]>0)
            {
                temp.from=i;
                temp.to=j;
                temp.weight=G[i][j];
                PQ.push(temp);
            }
        }
    }
    //将每一个顶点看成森林自己为根
    short* root=new short[numOfVertice];
    for (i=0; i<numOfVertice; ++i)
    {
        root[i]=i;
    }
    //Kruskal算法核心
    while (!PQ.empty())
    {
        temp=PQ.top();
        PQ.pop();
        //两个树的根不同
        if (root[temp.from]!=root[temp.to])
        {
            std::cout<<"Edge: "<<temp.from<<" "<<temp.to<<std::endl;
            //将第二棵树的根该为第一棵树的根
            for (j=0; j<numOfVertice; ++j)
            {
                if (root[j]==root[temp.to] && j!=temp.to)
                {
                    root[j]=root[temp.from];
                }
            }
            root[temp.to]=root[temp.from];
        }
    }
    delete[] root;
}

int main (int argc, char* argv[])
{
    Kruskal(graph, NUM_VERTICE);
    return 0;
}
```

无向图最小生成树的 Prim 算法举例

```
#include <iostream>
#include <cassert>
#include <limits>

const short NUM_VERTICE=9;
static short graph[NUM_VERTICE][NUM_VERTICE]=
{
    {0, 3, 4, 8, 0, 0, 0, 0, 0},
    {3, 0, 7, 0, 0, 0, 0, 0, 0},
    {4, 7, 0, 0, 0, 0, 0, 0, 0},
    {8, 0, 0, 0, 10, 0, 0, 0, 0},
    {0, 0, 0, 10, 0, 7, 0, 0, 7},
    {0, 0, 0, 0, 7, 0, 1, 2, 0},
    {0, 0, 0, 0, 0, 1, 0, 9, 4},
    {0, 0, 0, 0, 0, 2, 9, 0, 3},
    {0, 0, 0, 0, 7, 0, 4, 3, 0}
};
```

```
//在已标记的点集合中寻找与未标记点关联最小权的边
void min (short G[][NUM_VERTICE], int numOfVertice, bool* visited, int&
a, int& b)
{
    int i,j;
    short MIN=std::numeric_limits<short>::max();
    a=b=-1;
    for (i=0; i<numOfVertice; ++i)
    {
        if (visited[i])  //确定在已标记点集合中
        {
            for (j=0; j<numOfVertice; ++j)
            {
                //源点与目标点不同并且目标点未被标记过与源点关联且小于
最小值
                if (j!=i && !visited[j] && G[i][j]>0 && G[i][j]<MIN)
                {
                    a=i;
                    b=j;
                    MIN=G[i][j];
                }//end of if
            }//end of for
        }//end of if
    }// end of for
}

void Prim (short G[][NUM_VERTICE], int numOfVertice, int start)
{
    assert(G!=NULL);
    assert(numOfVertice>0);
    assert(start>=0 && start<numOfVertice);
    bool* visited=new bool[numOfVertice];
    assert(visited!=NULL);
    int i;
    for (i=0; i<numOfVertice; ++i)
    {
        visited[i]=false;
    }
    int a,b;
    visited[start]=true; //放入起点到标记集合
    int count=numOfVertice-1; //找numOfVertice-1条边
    while (count)
    {
        min(G, numOfVertice, visited, a, b);
        std::cout<<"Edge: "<<a<<"  "<<b<<std::endl;
        visited[b]=true; //标记选过的点添加到标记点集合
        --count;
    }
    delete[] visited;
}

int main (int argc, char* argv[])
{
    Prim(graph, NUM_VERTICE, 0);
    return 0;
}
```

有向图的单源最短路径 Dijkstra 算法举例

```
#include <iostream>
#include <cassert>
#include <limits>

const short NUM_VERTICE=9;
const short MAX=std::numeric_limits<short>::max();

static short graph[NUM_VERTICE][NUM_VERTICE]=
{
    {MAX, 3, 4, 8, MAX, MAX, MAX, MAX, MAX},
    {3, MAX, 2, MAX, MAX, MAX, MAX, MAX, MAX},
    {4, 2, MAX, MAX, MAX, MAX, MAX, MAX, MAX},
    {8, MAX, MAX, MAX, 10, MAX, MAX, MAX, MAX},
    {MAX, MAX, MAX, 10, MAX, 7, MAX, MAX, 7},
    {MAX, MAX, MAX, MAX, 7, MAX, 1, 2, MAX},
    {MAX, MAX, MAX, MAX, MAX, 1, MAX, 9, 4},
    {MAX, MAX, MAX, MAX, MAX, 2, 9, MAX, 3},
    {MAX, MAX, MAX, 7, MAX, 4, 3, MAX, MAX}
};

static int cost[NUM_VERTICE]={0};
static bool visited[NUM_VERTICE]={false};

void Dijkstra (short G[][NUM_VERTICE], int numOfVertice, int start)
{
    assert(G!=NULL);
    assert(numOfVertice>0);
    assert(start>=0 && start<numOfVertice);
    cost[start]=0;  //起点花费为
    visited[start]=true; //起点被访问
    int i,j;
    int best_j; //当前最小花费端点
    int best;  //当前最小花费
    do //Dijkstra算法核心
    {
        best=0;
        for (i=0; i<NUM_VERTICE; ++i)
        {
            //在已访问的点中搜索
            if (visited[i])
            {
                for (j=0; j<NUM_VERTICE; ++j)
                {
                    //端点未访问且与i关联
                    if (!visited[j] && graph[i][j]<MAX)
                    {
                        //当前花费最小
                        if (best==0 || graph[i][j]+cost[i]<best)
                        {
                            best=graph[i][j]+cost[i];
                            best_j=j;
                        }// end of if
                    }// end of if
                }//end of for
            }// end of if
        }// end of for
        if (best>0) //找到最小花费更新花费
        {
            cost[best_j]=best;
            visited[best_j]=true;
        }
    }while (best!=0);
}

int main (int argc, char* argv[])
{
    Dijkstra(graph, NUM_VERTICE, 0);
    for (int i=0; i<NUM_VERTICE; ++i)
    {
        std::cout<<cost[i]<<" ";
    }
    std::cout<<std::endl;
    return 0;
```

```
}
```

有向图的多源最短路径 Floyd 算法举例

```
#include <cassert>
#include <limits>

const short NUM_VERTICE=9;
const short MAX=std::numeric_limits<short>::max();

static short graph[NUM_VERTICE][NUM_VERTICE]=
{
    {MAX, 3, 4, 8, MAX, MAX, MAX, MAX, MAX},
    {3, MAX, 2, MAX, MAX, MAX, MAX, MAX, MAX},
    {4, 2, MAX, MAX, MAX, MAX, MAX, MAX, MAX},
    {8, MAX, MAX, MAX, 10, MAX, MAX, MAX, MAX},
    {MAX, MAX, MAX, 10, MAX, 7, MAX, MAX, 7},
    {MAX, MAX, MAX, MAX, 7, MAX, 1, 2, MAX},
    {MAX, MAX, MAX, MAX, MAX, 1, MAX, 9, 4},
    {MAX, MAX, MAX, MAX, MAX, 2, 9, MAX, 3},
    {MAX, MAX, MAX, 7, MAX, 4, 3, MAX, MAX}
};

static int cost[NUM_VERTICE]={0};
static bool visited[NUM_VERTICE]={false};

static short result[NUM_VERTICE][NUM_VERTICE];

void Floyd (short dest[][NUM_VERTICE], short src[][NUM_VERTICE], int
numOfVertice)
{
    assert(dest!=NULL);
    assert(src!=NULL);
    assert(numOfVertice>0);
    int i,j,k;
    for (i=0; i<numOfVertice; ++i)
    {
        for (j=0; j<numOfVertice; ++j)
        {
            dest[i][j]=src[i][j];
        }
    }
    for (k=0; k<numOfVertice; ++k)
    {
        for (i=0; i<numOfVertice; ++i)
        {
            for (j=0; j<numOfVertice; ++j)
            {
                if (dest[i][k]+dest[k][j]<dest[i][j])
                {
                    dest[i][j]=dest[i][k]+dest[k][j];
                }
            }//end of for j
        }// end of for i
    }//end of for k
}

int main (int argc, char* argv[])
{
    Floyd(result, graph, NUM_VERTICE);

    return 0;
}
```

拓扑排序举例

```
#include <iostream>
#include <cassert>
#include <queue>

const short NUM_VERTICE=10;

struct Node  //邻接链表元素节点
{
    short m_name;
    Node* m_next;
    Node (short name, Node* next=NULL): m_name(name), m_next(next) {}
};

Node* graph[NUM_VERTICE]; //图
short indegree[NUM_VERTICE]={0}; //与点相关的入度

void preinit ()
{
    for (short i=0; i<NUM_VERTICE; ++i)
    {
        graph[i]=new Node(i);
    }
```

```
}

void init ()
{
    graph[0]->m_next=new Node(9);
    graph[3]->m_next=new Node(1);
    graph[3]->m_next->m_next=new Node(2);
    graph[3]->m_next->m_next->m_next=new Node(4);
    graph[4]->m_next=new Node(1);
    graph[4]->m_next->m_next=new Node(2);
    graph[5]->m_next=new Node(4);
    graph[5]->m_next->m_next=new Node(6);
    graph[7]->m_next=new Node(0);
    graph[8]->m_next=new Node(5);
    graph[8]->m_next->m_next=new Node(7);
}

void caculateIndegree ()
{
    short i;
    Node* p=NULL;
    for (i=0; i<NUM_VERTICE; ++i)
    {
        p=graph[i]->m_next;
        while (p!=NULL)
        {
            ++indegree[p->m_name];
            p=p->m_next;
        }
    }
}

void sort ()
{
    std::queue<short> Q;
    short i;
    //入度为的点入队
    for (i=0; i<NUM_VERTICE; ++i)
    {
        if (indegree[i]==0)
        {
            Q.push(i);
        }
    }
    int count=0;
    int v;
    Node* p=NULL;
    //访问入度为的点使与它关联的点入度减
    while (!Q.empty())
    {
        v=Q.front();
        Q.pop();
        std::cout<<v<<" ";
        p=graph[v]->m_next;
        ++count;
        while (p!=NULL)
        {
            --indegree[p->m_name];
            if (indegree[p->m_name]==0)
            {
                Q.push(p->m_name);
            }
            p=p->m_next;
        }
    }
    std::cout<<std::endl;
    if (count<NUM_VERTICE)
    {
        std::cout<<"There is circle in the digraph."<<std::endl;
    }
}

void free ()
{
    Node* p=NULL;
    for (int i=0; i<NUM_VERTICE; ++i)
    {
        while (graph[i])
        {
            p=graph[i]->m_next;
            delete graph[i];
            graph[i]=p;
        }
    }
}

int main (int argc, char* argv[])
{
    preinit();
    init();
    caculateIndegree();
    sort();
    free();
    return 0;
}
```

AOE 网的算法举例

```
#include <iostream>
#include <algorithm>
#include <limits>
#include <queue>
#include <list>

const short MAX=std::numeric_limits<short>::max();
const short NUM_VERTICE=6;

struct Active  //活动信息结构
{
    short beforeEvent; //发生该活动前一个事件
    short afterEvent;  //发生该活动后一个事件
    short eTime;       //活动发生最早时间
    short lTime;       //活动发生最晚时间
    short space;       //活动时间余量
};

static short AOE[NUM_VERTICE][NUM_VERTICE]=
{
    {MAX, 3, 2, MAX, MAX, MAX},
    {MAX, MAX, MAX, 2, 3, MAX},
    {MAX, MAX, MAX, 4, MAX, 3},
    {MAX, MAX, MAX, MAX, MAX, 2},
    {MAX, MAX, MAX, MAX, MAX, 1},
    {MAX, MAX, MAX, MAX, MAX, MAX},
};

static short indegree[NUM_VERTICE]; //顶点的入度用于拓扑排序
static short tpArray[NUM_VERTICE];  //拓扑排序序列内存放顶点值

static short VE[NUM_VERTICE]; //事件的最早发生时间
static short VL[NUM_VERTICE]; //事件的最晚发生时间
static std::list<Active> activelist;

//计算入度用于拓扑排序
void CaculateIndegree ()
{
    int count;
    int i,j;
    for (i=0; i<NUM_VERTICE; ++i)
    {
        count=0;
        for (j=0; j<NUM_VERTICE; ++j)
        {
            if (AOE[j][i]<MAX)
            {
                ++count;
            }
        }
        indegree[i]=count;
    }
}

//拓扑排序
void TpSort ()
{
    std::queue<short> Q;
    int i;
    for (i=0; i<NUM_VERTICE; ++i)
    {
        if (indegree[i]==0)
        {
            Q.push(i);
        }
    }
    int level=-1;
    int v;
    while (!Q.empty())
    {
        v=Q.front();
        Q.pop();
        tpArray[++level]=v;
        for (i=0; i<NUM_VERTICE; ++i)
        {
            if (AOE[v][i]<MAX)
            {
                --indegree[i];
                if (indegree[i]==0)
                {
                    Q.push(i);
                }
            }
        }// end of for
    }// end of while
}

//正向扫描拓扑序列计算事件的最早发生时间
void GoForward ()
{
    int v=tpArray[0];
    VE[v]=0;
    int i,j;
    int e;
    for (i=1; i<NUM_VERTICE; ++i)
    {
        v=tpArray[i];
        e=0;
        for (j=0; j<NUM_VERTICE; ++j)
        {
            if (AOE[j][v]<MAX)
            {
                if (e==0 || AOE[j][v]+VE[j]>e)
                {
                    e=AOE[j][v]+VE[j];
                }
            }
        }
        VE[v]=e;
    }
}

//逆向扫描拓扑序列计算事件的最晚发生时间
void GoBack ()
{
    int v=tpArray[NUM_VERTICE-1];
    VL[v]=VE[v];
    int i, j;
    int e;
    for (i=NUM_VERTICE-2; i>=0; --i)
    {
        v=tpArray[i];
        e=0;
        for (j=0; j<NUM_VERTICE; ++j)
        {
            if (AOE[v][j]<MAX)
            {
                if (e==0 || VL[j]-AOE[v][j]<e)
                {
                    e=VL[j]-AOE[v][j];
                }
            }
        }
        VL[v]=e;
    }
}

//计算各活动信息
void fillInfo ()
{
    int i,j;
    Active temp;
    for (i=0; i<NUM_VERTICE; ++i)
    {
        for (j=0; j<NUM_VERTICE; ++j)
        {
            if (AOE[i][j]<MAX)
            {
                temp.beforeEvent=i;
                temp.afterEvent=j;
                temp.eTime=VE[i];
                temp.lTime=VL[j]-AOE[i][j];
                temp.space=temp.lTime-temp.eTime;
                activelist.push_back(temp);
            }
        }// enf of for j
    }// end of for i
}

//显示活动记录供for_each调用
void showActive (const Active& t)
{
    std::cout<<"Before event "<<t.beforeEvent
        <<" After event "<<t.afterEvent
        <<" early time "<<t.eTime
        <<" last time "<<t.lTime
        <<" space time "<<t.space<<std::endl;
}

//显示处理信息
void showInfo ()
{
    int i,v;
    for (i=0; i<NUM_VERTICE; ++i)
    {
        v=tpArray[i];
        std::cout<<"Event "<<v<<" et "<<VE[v]<<" lt "<<VL[v]<<std::endl;
    }
    std::for_each(activelist.begin(), activelist.end(), showActive);
}

int main (int argc, char* argv[])
{
    CaculateIndegree();
    TpSort();
    GoForward();
    GoBack();
    fillInfo();
    showInfo();
    return 0;
}
```

求图的一个中心算法举例

```
#include <iostream>

const int MAXN=100;    //邻接矩阵最大容量
float chart[MAXN][MAXN]; //有无图的邻接矩阵
bool mark[MAXN];       //标志表true Vi到Vj的最短路径已求出
float b[MAXN];           //Vi到Vj的最短路径长度
int n;                   //顶点数

void init ()
{
    int i,j,a,k;
    float c;
    do
    {
        std::cout<<"Please input the number of vertice: ";
        std::cin>>n;
    } while (n<=0 || n>=MAXN);
    for (i=0; i<n; ++i)
    {
        for (j=0; j<n; ++j)
        {
            chart[i][j]=0;
        }
    }
    std::cout<<"Please input the number of edge: ";
    std::cin>>a;
    std::cout<<"Please input the weight of the edge: "<<std::endl;
    for (k=0; k<a; ++k)
    {
        std::cin>>i>>j>>c;
        chart[i][j]=c; //这两个赋数体现为无向图
        chart[j][i]=c;
    }
}

//返回顶点x的最大服务距离即在顶点x与其它顶点距离中最大值
float findFarest (int x)
{
    int best_j; //顶点x与顶点j最短路径终点
    int i,j;
    float best, farest;//最短路径  最大服务距离
    //初始化标志数组与最短路径数组
    for (i=0; i<n; ++i)
    {
        b[i]=0;
        mark[i]=false;
    }
    mark[x]=true;
    farest=0;
    do
    {
        best=0;
        //从所有起点已编号，终点为编号的弧集合中，选一条弧
        //x best_j使顶点x到best_j最短
        for (i=0; i<n; ++i)
        {
            if (mark[i])
            {
                for (j=0; j<n; ++j)
                {
                    if (!mark[j] && chart[i][j]>0)
                    {
                        if (best==0 || b[i]+chart[i][j]<best)
                        {
                            best=b[i]+chart[i][j];
                            best_j=j;
                        }
                    }
                }
            }
        }
        if (best)
        {
            b[best_j]=best;
            mark[best_j]=true;
            //筛选最大的最短路径
            if (best>farest)
            {
                farest=best;
            }
        }
    }while (best);
    return farest;
}

void find ()
{
    int i,best_i;
    float e,best;
    best=0;
    //求出各顶点的最大服务距离中的最小值
    for (i=0; i<n; ++i)
    {
        e=findFarest(i);
        if (e<best || best==0)
        {
            best=e;
            best_i=i;
        }
    }
    std::cout<<"Best vertice is "<<best_i<<" and length is 
"<<best<<std::endl;
}
```

```
int main(int argc, char* argv[])
{
    init();
    find();
    return 0;
}
```

求图的 P 个中心算法举例

```
#include <iostream>
#include <iomanip>

const int MAXN=50;    //邻接矩阵最大容量
float chart[MAXN][MAXN]; //有无图的邻接矩阵
float sheet[MAXN][MAXN];   //距离表
bool mark[MAXN];        //标志表true Vi到Vj的最短路径已求出
int s[MAXN], b_s[MAXN];     //s组合方案b_s目前最好的组合
int way[MAXN], b_way[MAXN]; //way辅助变量b_way最佳方案
//即服务点i选择p中心中最近的服务点b_s[b_way[i]]
int n,p; //n顶点数 p中心点数
float bst;  //目前最大服务距离的最小值

void init ()
{
    int i,j,a,k;
    float c;
    //初始化点数
    do
    {
        std::cout<<"Input the number of vertice: ";
        std::cin>>n;
    }while (n<=0 || n>=MAXN);
    for (i=0; i<n; ++i)
    {
        for (j=0; j<n; ++j)
        {
            chart[i][j]=0;
        }
    }
    //初始化边及权
    std::cout<<"Input the number of edge: ";
    std::cin>>a;
    for (k=0; k<a; ++k)
    {
        std::cin>>i>>j>>c;
        chart[i][j]=c;
        chart[j][i]=c;
    }
    //初始化服务点数
    do
    {
        std::cout<<"Input the number of center point: ";
        std::cin>>p;
    }while (p<=0 || p>=n);
}

//构造距离表sheet中的x行各元素
void makeOneLine (int x)
{
    int i,j, best_j;
    float best;
    //初始化标志标
    for (i=0; i<n; ++i)
    {
        mark[i]=false;
        sheet[x][i]=0;
    }
    mark[x]=true;
    sheet[x][x]=0;
    do
    {
        best=0;
        for (i=0; i<n; ++i)
        {
            if (mark[i])
            {
                for (j=0; j<n; ++j)
                {
                    if (!mark[j] && chart[i][j]>0)
                    {
                        if (best==0 || sheet[x][i]+chart[i][j]<best)
                        {
                            best=sheet[x][i]+chart[i][j];
                            best_j=j;
                        }
                    }
                }
            }
        }
        if (best)
        {
            sheet[x][best_j]=best;
            mark[best_j]=true;
        }
    }
    while (best);
}

//构造距离表
void makeSheet ()
{
    int i;
    for (i=0; i<n; ++i)
    {
        makeOneLine(i);
    }
}

void check ()
{
    int i,j,k;
    float t,v;
    //v为当前方案的最大服务距离
    v=0;
    for (i=0; i<n; ++i) //vi是目的点,
    {
        t=sheet[s[0]][i];
        k=1;
        //在当前p节点中，选择离vi较近的一个结点vk，vi至vk距离最短
        //为t（vk为p组合的第k个元素）
        for (j=2; j<p; ++j)   //扫描选中原点到目的点距离取最小者
        {
            if (sheet[s[j]][i]<t)
            {
                t=sheet[s[j]][i];
                k=j;
            }
        }
        if (t>v)
        {
            v=t; //该方案中这一个原点到目标点最大距离中最大一个
        }
        way[i]=k;
    }
    if (v<bst || bst==-1) //所有方案服务距离最大中最小
    {
        bst=v;
        //存放各顶点最近的服务点
        for (i=0; i<p; ++i)
        {
            b_s[i]=s[i];
        }
        for (i=0; i<n; ++i)
        {
            b_way[i]=way[i];
        }
    }
}

void find ()
{
    int lev;
    bst=-1;
    lev=0;
    s[0]=-1; //s中是p个服务点的方案
    //产生排列方案并计算最佳值
    while (lev>-1)
    {
        while (s[lev]<n-1)
        {
            ++s[lev];
            if (lev==p-1)
            {
                //若产生一个p组合，则求当前p组合的最大服务距离，并且得出
目前
                //扩展出来的所有p组合最小的最大服务距离方案
                check();
            }
            else
            {
                ++lev;
                s[lev]=s[lev-1];
            }
        }
        --lev;
    }
}

void show ()
{
    int i;
    //输出服务点
    std::cout<<"You should select: ";
    for (i=0; i<p; ++i)
    {
        std::cout<<b_s[i]<<" ";
    }
    std::cout<<std::endl;
    //输出个点与服务点的关系
    for (i=0; i<n; ++i)
    {
        std::cout<<i<<" go to "<<b_s[b_way[i]]<<std::endl;
    }
}

int main(int argc, char* argv[])
{
    init();       //初始化
    makeSheet(); //构造距离表
    find();       //计算p中心
    show();       //输出
    return 0;
}
```

SPFA 算法举例

```
#include <limits>
#include<iostream>
#include <queue>

using namespace std;

#define SIZE 26
#define MAX std::numeric_limits<int>::max()

int graph[SIZE][SIZE]={0}; //邻接矩阵
int l; //顶点数目
bool flag[SIZE]; //入队标志false已经入过队, 否则没有还可以扩展
int times[SIZE]; //扩展次数如果大于顶点数就不再扩展
int distances[SIZE]; //各点到起始点距离

int spfa(int start, int end)
{
    memset(times, 0, sizeof(times));
    memset(flag, true, sizeof(flag));

    int i, curr;
    bool f=true;
    i=l+1;
    while (i--)
        distances[i]=MAX;

    distances[start]=0;
    flag[start]=false;

    std::queue<int> q;
    q.push(start);
    while (!q.empty())
    {
        curr=q.front();
        q.pop();
        for (int i=1; i<=l; ++i)
        {
            if (graph[curr][i]!=MAX) //边
            {
                //可以扩展
                if (distances[i]>distances[curr]+graph[curr][i])
                {
                    distances[i]=distances[curr]+graph[curr][i];
                    //增加扩展次数
                    times[i]=times[curr]+1;
                    if (times[i]>l)
                    {
                        f=true;
                        break;
                    }
                    //继续扩展
                    if (flag[i])
                    {
                        q.push(i);
                        flag[i]=false;
                    }
                }
            }
        }
        if (!f)
        {
            f=false;
            flag[curr]=true;
        }
        else
            f=false;
    }
    return distances[end];
}

int main()
{
    int t;
    std::cin>>t;
    int n, b;
    int s, e, k, m;
    while (t--)
    {
        std::cin>>n>>b>>l;
        for (int i=1; i<=l; ++i)
            for (int j=1; j<=l; ++j)
                graph[i][j]=MAX;

        for (int i=0; i<n; ++i)
        {
            std::cin>>s>>e>>k>>m;
            if (k==0)
                graph[s][e]=-m;
            else
                graph[s][e]=m;
        }
        std::cout<<spfa(1, b)<<'\n';
    }
    return 0;
}
```

割顶和块的算法举例

```
#include<iostream>
```

```cpp
#include <algorithm>
#include <fstream>

#define maxn 30

struct node
{
    int k; //顶点类型，dfn编码
    int l; //low编码
};

typedef int ghtype[maxn][maxn]; //邻接矩阵类型
typedef node ltype[maxn]; //点类型

ghtype g; //邻接矩阵
int n; //顶点数
int p; //栈指针
int x; //根的子树个数
int i; //辅助变量

std::ifstream f("input.txt"); //文件对象
ltype l; //顶点序列

int st[maxn]; //栈
bool k[maxn]; //割顶集合

void push (int a)
{
    ++p;
    st[p]=a;
}

int pop()
{
    int t=st[p];
    --p;
    return t;
}

void read_graph()
{
    f>>n;
    for (int i=0; i<n; ++i)
    {
        for (int j=0; j<n; ++j)
        {
            f>>g[i][j];
        }
    }
    f.close();
    memset(l, 0, sizeof(l));
    p=0;
    push(0); //顶点入栈
    //顶点首次被访问 dfn(0)=low(0)=1
    i=1;
    l[0].k=i;
    l[0].l=i;
    //割顶集合为空
    memset(k, false, sizeof(k));
    x=0;
}

void search(int v)
{
    int u;
    for (u=0; u<n; ++u)
    {
        if (g[v][u]>0) //<v,u>边存在
        {
            if (l[u].k==0) //u首次被访问
            {
                //设置u的dfn和low值
                ++i;
                l[u].k=i;
                l[u].l=i;
                push(u);
                //从u出发递归求解
                search(u);
                //v的儿子u完全扫描完毕,设置v的low值
                l[v].l=std::min(l[v].l, l[u].l);
                //从u及u的后代不会追溯到比v更早的祖先点
                if (l[u].l>=l[v].k)
                {
                    //若v不是根且不属于割顶集合,则打印割顶v，v进割顶集合
                    if (v!=0 && !k[v])
                    {
                        std::cout<<"["<<v<<"]"<<"\n";
                        k[v]=true;
                    }
                    //若v是根，计算其子树个数
                    if (v==0)
                        ++x;
                    //从栈中依次取出至v的个顶点,产生一个连通分支
                    while (st[p]!=v)
                        std::cout<<pop()<<" ";
                    std::cout<<std::endl;
                }//
            }//
            else
                l[v].l=std::min(l[v].l, l[u].k); //<v,u>是后向边u重复访问, 则设置v的low值
        }
    }
}

int main(int argc, char* argv[])
{
    read_graph();
    search(0);
    if (x>1)
        std::cout<<"[0]\n";
    return 0;
}
```

计算几何算法

```
#include <math.h>

using namespace std;

#define PRECISION 1e-8

struct Point
{
    double x, y;
};//平面上的点

int dbcmp(double d)
{
    if (fabs(d) < PRECISION)
        return 0;
    return d > 0 ? 1 : -1;
}//三叉口函数,避免精度误差
```

向量模

```
double length(double x, double y)
{
    return sqrt(x*x + y*y);
}//向量的模
```

向量点积

```
double dotdet(double x1, double y1, double x2, double y2)
{
    return x1*x2 + y1*y2;
}//向量点积
```

向量叉积

```
double det(double x1, double y1, double x2, double y2)
{
    return x1*y2 - x2*y1;
}//向量叉积

int cross(Point a, Point c, Point d)
{
    return dbcmp( det(a.x-c.x, a.y-c.y, d.x-c.x, d.y-c.y) );
}//右手螺旋定则,a在cd右侧返回,左侧返回-1,共线返回
```

左右判断

```
bool between(Point a, Point c, Point d)
{
    return dbcmp( dotdet(c.x-a.x, c.y-a.y, d.x-a.x, d.y-a.y) ) != 1;
}//在cross(a, c, d)==0的前提下,点a在线段cd内部返回true
```

相交判断

```
int segIntersect(Point a, Point b, Point c, Point d)
{
    int a_cd = cross(a, c, d);
    if (a_cd == 0 && between(a, c, d))
        return 2;

    int b_cd = cross(b, c, d);
    if (b_cd == 0 && between(b, c, d))
        return 2;

    int c_ab = cross(c, a, b);
    if (c_ab == 0 && between(c, a, b))
        return 2;

    int d_ab = cross(d, a, b);
    if (d_ab == 0 && between(d, a, b))
        return 2;

    if ((a_cd ^ b_cd) == -2 && (c_ab ^ d_ab) == -2)
        return 1;

    return 0;
}//两线段相交情况,0--不相交,1--规范相交,2--不规范相交(交于端点或重合)
```

正规相交交点

```
void intersectPoint(Point a, Point b, Point c, Point d, Point &e)
{
    double sc, sd;

    sc = fabs( det(b.x-a.x, b.y-a.y, c.x-a.x, c.y-a.y) );
    sd = fabs( det(b.x-a.x, b.y-a.y, d.x-a.x, d.y-a.y) );
    e.x = (sc * d.x + sd * c.x) / (sc + sd);
    e.y = (sc * d.y + sd * c.y) / (sc + sd);
}//两线段规范相交时,用点e返回交点
```

判断多边形凸

```
bool convex(const std::vector<Point>& v)
{
    Point c, d;
    c=v[0];
    for (int i=1; i<v.size(); ++i)
    {
        d=v[i];
        for (int j=0; j<v.size(); ++j)
```

```
            if (!(v[j]==c || v[j]==d))
            {
                if (cross(v[j], c, d)!=-1)
                    return false;
            }
            c=d;
    }
    return true;
}
```

任意多变形面积

```
double linesqr(double x1,double y1,double x2,double y2)
{ return (x2-x1)*(y1+y2)/2.0; }

//clockwise +
double anyS(std::vector<Point>& v)
{
    double fx,fy,x1,y1,x2,y2;
    double s=0.0;
    int n,i;
    n=v.size();
    x1=v[0].m_x;
    y1=v[0].m_y;
    fx=x1;fy=y1;
    x2=v[1].m_x;
    y2=v[1].m_y;
    s=linesqr(x1,y1,x2,y2);
    for(i=2;i<n;++i)
    {
        x1=x2;y1=y2;
        x2=v[i].m_x;
        y2=v[i].m_y;
        s+=linesqr(x1,y1,x2,y2);
    }
    s+=linesqr(x2,y2,fx,fy);//首尾相连

    return s;
}
```

凸包问题的快包实现举例

```
#include <iostream>    //cout, cin
```

```
#include <algorithm>  //copy sort
#include <iterator>   //ostream_iterator
#include <vector>     //vector
#include <queue>      //queue
#include <stdlib.h>   //srand rand
#include <time.h>     //time
#include <limits>     //numeric_limits
#include <math.h>     //sqrt
#include <assert.h>   //assert

class Point
{
public:
    Point( int x=0, int y=0 ) : m_x(x), m_y(y) {}
public:
    int m_x;
    int m_y;
};

std::istream& operator>> (std::istream& in, Point& p)
{
    return (in>>p.m_x>>p.m_y);
}

std::ostream& operator<< (std::ostream& out, const Point& p)
{
    return (out<<"x="<<p.m_x<<" y="<<p.m_y);
}

class CmpXY
{
public:
    bool operator() (const Point& l, const Point& r)
    {
        bool ret;
        if ( l.m_x<r.m_x )
            ret=true;
        else if ( l.m_x==r.m_x )
        {
            if ( l.m_y<r.m_y )
                ret=true;
            else
                ret=false;
        }
        else
            ret=false;
        return ret;
    }
};

void findup(std::vector<Point>& in, const Point& p1, const Point& p2,
std::vector<Point>& out)
{
    std::vector<Point> up;  //新上包点
    Point upp; //极点
    int uph=0; //极高
    int h;
    std::vector<Point>::iterator iter=in.begin();
    while ( iter!=in.end() )
    {

    h=p1.m_x*p2.m_y+iter->m_x*p1.m_y+p2.m_x*iter->m_y-iter->m_x*p2.m_
y-p2.m_x*p1.m_y-p1.m_x*iter->m_y;
        if ( h>0 )  //p3在直线p1p2左边在上包中
        {
            up.push_back(*iter);
            if ( h>uph )
            {
                uph=h;
                upp=*iter;
            }
        }
        ++iter;
    }
    if ( uph>0 )
        out.push_back(upp);
    //递归构造新上包
    if ( !up.empty() )
    {
        findup(up, p1, upp, out);
        findup(up, upp, p2, out);
    }
}

void finddown(std::vector<Point>& in, const Point& p1, const Point& p2,
std::vector<Point>& out)
{
    std::vector<Point> down; //新下包点
    Point downp;  //极点
    int downh=0;  //极高
    int h;
    std::vector<Point>::iterator iter=in.begin();
    while ( iter!=in.end() )
    {

    h=p1.m_x*p2.m_y+iter->m_x*p1.m_y+p2.m_x*iter->m_y-iter->m_x*p2.m_
y-p2.m_x*p1.m_y-p1.m_x*iter->m_y;
        if ( h<0 ) //p3在直线p1p2右边在下包中
        {
            down.push_back(*iter);
            if ( h<downh )
            {
                downh=h;
                downp=*iter;
            }
        }
        ++iter;
    }
    if ( downh<0 )
        out.push_back(downp);
    //递归构造新下包
    if ( !down.empty() )
    {
        finddown(down, p1, downp, out);
        finddown(down, downp, p2, out);
    }
}

void FindPoint(std::vector<Point>& in, std::vector<Point>& out)
{
    //排序
    std::sort(in.begin(), in.end(), CmpXY());
    Point p1=in.front(); //凸包一点
    Point p2=in.back();  //凸包一点
    out.push_back(p1);
    out.push_back(p2);
    //迭代变量
    std::vector<Point>::iterator iter=in.begin()+1;
    std::vector<Point> up; //上凸包点
    std::vector<Point> down; //下凸包点
    Point upp, downp;  //凸包极点
    int uph=0, downh=0;    //极点离基线高度
    int h;
    while ( iter!=in.end()-1 )
    {

    h=p1.m_x*p2.m_y+iter->m_x*p1.m_y+p2.m_x*iter->m_y-iter->m_x*p2.m_
y-p2.m_x*p1.m_y-p1.m_x*iter->m_y;
        if ( h>0 ) //p3在直线p1p2左边在上包中
        {
            up.push_back(*iter);
            if ( h>uph )
            {
                uph=h;
                upp=*iter;
            }
        }
        if ( h<0 ) //p3在直线p1p2右边在下包中
        {
            down.push_back(*iter);
            if ( h<downh )
            {
                downh=h;
                downp=*iter;
            }
        }
        ++iter;
    }
    if ( uph>0 )
        out.push_back(upp);
    if ( downh<0 )
        out.push_back(downp);
    //构造上包
    if ( !up.empty() )
    {
        findup(up, p1, upp, out);
        findup(up, upp, p2, out);
    }
    //构造下包
    if ( !down.empty() )
    {
        finddown(down, p1, downp, out);
        finddown(down, downp, p2, out);
    }
}
```

```
#define SIZE 100000

int main(int argc, char* argv[])
{
    srand(time(0));
    std::vector<Point> in;
    std::vector<Point> out;
    for ( int i=0; i<SIZE; ++i )
    {
        Point p;
        p.m_x=rand()%SIZE;
        p.m_y=rand()%SIZE;
        in.push_back(p);
    }
    //std::copy(in.begin(), in.end(),
std::ostream_iterator<Point>(std::cout, " "));
    //system("pause");
    FindPoint(in, out);
    std::copy(out.begin(), out.end(),
std::ostream_iterator<Point>(std::cout, " "));
    return 0;
}
```

STL 算法参考

accumulate()

```
template < class InputIterator, class Type >
Type accumulate(
                    InputIterator first, InputIterator last,
                    Type init );

template < class InputIterator, class Type,
class BinaryOperation >
Type accumulate(
                   InputIterator first, InputIterator last,
                   Type init, BinaryOperation op );
```

accumulate()的第一个版本把由iterator 对[first,last) 标记的序列中的元素之和加到一个由init 指定的初始值上例如已知序列{1,1,2,3,5,8}和初始值0 则结果是20 在第二个版本中不再是做加法而是传递进来的二元操作被应用在元素上例如如果向accumulate()传递函数对象times<int> 则结果是240 当然假设初始值是1 而不是0 accumulate()是一个算术算法要使用它我们必须包含<numeric>头文件

adjacent_difference()

```
template < class InputIterator, class OutputIterator >
OutputIterator adjacent_difference(
                        InputIterator first, InputIterator last,
                        OutputIterator result );

template < class InputIterator, class OutputIterator,
class BinaryOperation  >
OutputIterator adjacent_difference(
                        InputIterator first, InputIterator last,
                        OutputIterator result, BinaryOperation  op );
```

adjacent_differece()的第一个版本创建了一个新的序列该序列中的每个新值第一个元素除外都代表了当前元素与上一个元素的差例如已知序列{0,1,1,2,3,5,8} 则新序列的第一个元素只是原来序列第一个元素的拷贝0 第二个元素是前两个元素的差1 第三个元素是第二个和第三个元素的差即1-1 为0 等等新序列是{0,1,0,1,1,2,3}第二个版本用指定的二元操作计算相邻元素的差例如使用同一个序列让我们传递times<int>函数对象同样新序列的第一个元素只是原来序列第一个元素的拷贝0 第二个元素是原来第一个和第二个元素的积也是0 第三个元素是第二和第三个元素的积即1*1 为1 等等新序列是{0,0,1,2,6,15,40}
在两个版本中OutputIterator 总是指向新序列末元素的下一个位置
adjacent_difference()是一种算术算法使用这两个版本都必须包含头文件<numeric>

adjacent_find()

```
template  < class ForwardIterator >
ForwardIterator
adjacent_find( ForwardIterator first, ForwardIterator last );

template  < class ForwardIterator, class BinaryPredicate >
ForwardIterator
adjacent_find( ForwardIterator first,
               ForwardIterator last, Predicate pred );
```

adjacent_find()在由[first,last)标记的元素范围内查找第一对相邻的重复元素如果找到则返回一个ForwardIterator 并指向这对元素的第一个元素否则返回last 例如已知序列{0,1,1,2,2,4} 元素对{1,1}被找到函数返回指向第一个1 的iterator

binary_search()

```
template< class ForwardIterator, class Type >
bool
```

```
binary_search( ForwardIterator first,
               ForwardIterator last, const Type &value );

bool
binary_search( ForwardIterator first,
             ForwardIterator last, const Type &value,
             Compare comp );
```

binary_search()在由[first,last)标记的有序序列中查找value　如果找到则返回true　否则返回false　第一个版本假设该容器是用底层类型的小于操作符排序的在第二个版本中我们指出了该容器是用指定的函数对象进行排序的

copy()

```
template < class InputIterator, class OutputIterator >
OutputIterator
copy( InputIterator first1, InputIterator last,
      OutputIterator first2 );
```

copy()把由[first,last)标记的序列中的元素拷贝到由first2　标记为开始的地方它返回first2　但此时first2　已经被移动到最后一个插入元素的下一位置例如已知序列{0,1,2,3,4,5}我们可以用下列调用将序列左移1　位

```
int ia[] = { 0, 1, 2, 3, 4, 5 };
// 左移 1 位, 结果为 {1,2,3,4,5,5}
copy( ia+1, ia+6, ia );
```

copy()从ia　的第二个元素开始把1　拷贝到第一个位置上直到所有元素都被拷贝到它左边的位置上

copy_backward()

```
template < class BidirectionalIterator1,
           class BidirectionalIterator2  >
BidirectionalIterator2
copy_backward( BidirectionalIterator1 first,
               BidirectionalIterator1 last1,
               BidirectionalIterator2 last2 );
```

copy_backward()除了元素以相反的顺序被拷贝外其他行为与copy()相同也就是说拷贝操作从last-1　开始直到first　这些元素也被从后向前拷贝到目标容器中从last2-1　开始一直拷贝last1-first　个元素例如已知序列{0,1,2,3,4,5}　我们可以把最后三个元素(3,4,5)拷贝到前三个(0,1,2)中做法是把first　设为值0　的地址last1　设为值3　的地址而last2　设为值5　的后一个位置值为5　的元素被赋给前面值为2　的元素而元素4　被赋给前面值为1　的元素最后元素3　被赋给前面值为0　的元素结果序列是{3,4,5,3,4,5}

count()

```
template< class InputIterator, class Type >
iterator_traits<InputIterator>::distance_type
count( InputIterator first,
       InputIterator last, const Type& value );
```

count()利用等于操作符把[first,last)标记范围内的元素与value 进行比较并返回容器中与value 相等的元素的个数[注意标准库的实现支持早期的count()版本]

count_if()

```
template< class InputIterator, class Predicate >
iterator_traits<InputIterator>::distance_type
count_if( InputIterator first,
          InputIterator last, Predicate pred );
```

count_if()对于[first,last]标记范围内的每个元素都应用pred 并返回pred 计算结果为true的次数

equal()

```
template< class InputIterator1, class InputIterator2 >
bool
equal( InputIterator1 first1,
       InputIterator1 last, InputIterator2 first2 );

template< class InputIterator1, class InputIterator2,
          class BinaryPredicate >
bool
equal( InputIterator1 first1, InputIterator1 last,
       InputIterator2 first2, BinaryPredicate pred );
```

如果两个序列在范围[first,last)内包含的元素都相等则equal()返回true 如果第二序列包含更多的元素则不会考虑这些元素如果我们希望保证两个序列完全相等则需要写

```
if ( vec1.size() == vec2.size() &&
equal( vec1.begin(), vec1.end(), vec2.begin() );
```

或使用该容器的等于操作符比如vecl==vec2 如果第二个容器比第一个容器的元素少算法的迭代过程应该超过其末尾则运行时刻的行为是未定义的缺省情况下底层元素类型的等于操作符用来作比较第二个版本应用pred

equal_range()

```
template< class ForwardIterator, class Type >
```

```
pair< ForwardIterator, ForwardIterator >
equal_range(  ForwardIterator first,
              ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
pair< ForwardIterator, ForwardIterator >
equal_range(  ForwardIterator first,
              ForwardIterator last, const Type &value,
              Compare comp );
```

equal_range()返回一对iterator 第一个iterator 表示由lower_bound()返回的iterator 值第二个表示由upper_bound()返回的iterator 值它们的语义描述见相应的算法例如已知下面的序列

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

用值21 调用equal_range() 返回一对iterator 这两个iterator 都指向值22 用值22 调用equal_range() 返回一对iterator 其中first 指向值22 second 指向值23 第一个版本使用底层类型的小于操作符第二个版本则用comp 对元素进行排序

fill()

```
template< class ForwardIterator, class Type >
void
fill(  ForwardIterator first,
    ForwardIterator last, const Type& value );
```

fill()将value 的拷贝赋给[first,last)范围内的所有元素

fill_n()

```
template< class ForwardIterator, class Size, class Type >
void
fill_n(  ForwardIterator first,
      Size  n, const Type& value );
```

fill_n()把value 的拷贝赋给[first,first+count)范围内的count 个元素

find()

```
template< class InputIterator, class T >
InputIterator
find( InputIterator first,
      InputIterator last, const T &value );
```

find()利用底层元素类型的等于操作符对[first,last)范围内的元素与value 进行比较当发现匹配时结束搜索过程且find()返回指向该元素的一个InputIterator 如果没有发现匹配则返回last

find_if()

```
template< class InputIterator, class Predicate >
InputIterator
find_if( InputIterator first,
         InputIterator last, Predicate pred );
```

依次检查[first,last)范围内的元素并把pred 应用在这些元素上面如果pred 计算结果为true 则搜索过程结束find_if()返回指向该元素的InputIterator 如果没有找到匹配则返回last

find_end()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1
find_end( ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2 );

template< class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate >
ForwardIterator1
find_end( ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          BinaryPredicate pred );
```

在由[first,last)标记的序列中查找由iterator 对[first2,last2)标记的第二个序列的最后一次出现例如已知字符序列mississippi 和第二个序列ss 则find_end()返回一个ForwardIterator 指向第二个ss 序列的第一个s 如果在第一个序列中没有找到第二个序列则返回last1 在第一个版本中使用底层的等于操作符在第二个版本中使用用户传递进来的二元操作pred

find_first_of()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1
find_first_of( ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2

template< class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate >
ForwardIterator1
find_first_of( ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              BinaryPredicate pred );
```

由[first2,last2)标记的序列包含了一组元素的集合find_first_of()将在由[first1,last1)标记的序列中搜索这些元素例如假设我们希望在字符序列synesthesia 中找到第一个元音为了做到这一点我们把第二个序列定义为aeiou find_first_of()返回一个ForwardIterator 指向元音序列中的元素的第一个出现本例中指向第一个e 如果第一个序列不含有第二个序列中的任何元素则返回last1 在第一个版本中使用底层元素类型的等于操作符在第二个版本中使用二元操作pred

for_each()

```
template< class InputIterator, class Function >
Function
for_each( InputIterator first,
          InputIterator last, Function func );
```

for_each()依次对[first,last)范围内的所有元素应用函数func func 不能对元素执行写操作因为前两个参数都是InputIterator 所以不能保证支持赋值操作如果我们希望修改元素则应该使用transform()算法func 可以返回值但是该值会被忽略

generate()

```
template< class ForwardIterator, class Generator >
void
generate( ForwardIterator first,
          ForwardIterator last, Generator gen );
```

generate()通过对gen 的连续调用来填充一个序列的[first,last)范围gen 可以是函数对象或函数指针

generate_n()

```
template< class ForwardIterator,
class Size, class Generator >
void
generate_n( OutputIterator first, Size n, Generator gen );
```

generate_n()通过对gen 的n 次连续调用来填充一个序列中从first 开始的n 个元素gen可以是函数对象或函数指针

includes()

```
template< class InputIterator1, class InputIterator2 >
bool
includes( InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2 );
template< class InputIterator1, class InputIterator2,
class Compare >
bool
includes( InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          Compare comp );
```

includes()判断[first1,last1)的每一个元素是否被包含在序列[first2,last2)中 第一个版本假设这两个序列是用底层元素类型的小于操作符排序的第二个版本用comp 来判定元素顺序

inner_product()

```
template < class InputIterator1, class InputIterator2,
           class Type >
Type
inner_product(
              InputIterator1 first1, InputIterator1 last,
              InputIterator2 first2, Type init );

template < class InputIterator1, class InputIterator2,
           class Type,
class BinaryOperation1,  class BinaryOperation2  >
Type
inner_product(
             InputIterator1 first1, InputIterator1 last,
             InputIterator2 first2, Type init,
              BinaryOperation1  op1, BinaryOperation2  op2 );
```

inner_product()的第一个版本对两个序列做内积对应的元素相乘再求和并将内积加到一个由init 指定的初始值上第一个序列由[first1,last)标记第二个序列由first2 开始随着第一个序列而逐渐递增例如已知序列{2,3,5,8}和{1,2,3,4} 则下列乘积对的和就是结果

```
2*1 + 3*2 + 5*3 + 8*4
```

如果提供初始值0 则结果是55第二个版本用二元操作op1 代替缺省的加法操作用二元操作op2 代替缺省的乘法操作例如如果同样用上两个序列指定op1 为减法op2 为加法则结果是下列加法对的差

```
(2+1) - (3+2) - (5+3) - (8+4)
```

inner_product()是一个算术算法要使用它必须包含头文件<numeric>

inplace_merge()

```
template< class BidirectionalIterator  >
void
inplace_merge( BidirectionalIterator  first,
               BidirectionalIterator  middle,
                BidirectionalIterator  last );
template< class BidirectionalIterator,  class Compare >
void
inplace_merge(  BidirectionalIterator  first,
                BidirectionalIterator  middle,
                BidirectionalIterator  last, Compare comp );
```

inplace_merge()合并两个排过序的连续序列分别由[first,middle)和[middle,last)标记结果序列覆盖了由first 开始的这两段范围第一个版本使用底层类型的小于操作符对元素进行排序第二个版本根据程序员传递的二元比较操作对元素进行排序

iter_swap()

```
template <class ForwardIterator1, class ForwardIterator2>
void
iter_swap ( ForwardIterator1 a, ForwardIterator2 b );
```

iter_swap()交换由两个ForwardIterator a 和b 所指向的元素中的值

lexicographical_compare()

```
template <class InputIterator1, class InputIterator2 >
bool
lexicographical_compare(
                InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2 );

template < class InputIterator1, class InputIterator2,
class Compare >
bool
lexicographical_compare(
                    InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    Compare comp );
```

lexicographical_compare()比较由[first1,last1)和[first2,last2)标识的两个序列的对应元素对比较操作将一直进行下去直到某个元素对不匹配或者到达[last1,last2]对或者到达last1 或last2
如果两个序列长度不等对于第一个不匹配的元素对发生以下事情
如果第一个序列的元素小则返回true 否则返回false
如果到达last1 而last2 未到则返回true
如果到达last2 而未到达last1 则返回false
如果last1 和last2 都已到达所有元素都匹配则返回false 即第一个序列在字典序上不小于第二个序列
例如已知下列两个序列

```
string arr1[] = { "Piglet",  "Pooh", "Tigger" };
string arr2[] = { "Piglet",  "Pooch", "Eeyore"  };
```

本算法在第一个元素对上匹配但是在第二个上不匹配Pooh 大于Pooch 因为c 在字典序上小于h 想像一下字典中的单词是如何排序的算法在这一点上停止不再比较第三个元素比较的结果是false算法的第二个版本使用了比较对象而不再使用底层元素类型的小于操作符

lower_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
lower_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
ForwardIterator
lower_bound( ForwardIterator first,
               ForwardIterator last, const Type &value,
               Compare comp );
```

lower_bound()返回一个iterator 它指向在[first,last)标记的有序序列中可以插入value而不会破坏容器顺序的第一个位置而这个位置标记了一个大于等于value的值例如已知下列序列

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

用值21 调用lower_bound() 返回一个指向值22 的iterator 用值22 调用lower_bound()也返回一个指向值22 的iterator 第一个版本使用底层类型的小于操作符第二个版本根据comp 对元素进行排序和比较

max()

```
template< class Type >
const Type&
max( const Type &aval, const Type &bval );
template< class Type, class Compare >
const Type&
max( const Type &aval, const Type &bval, Compare comp );
```

max()返回aval 和bval 两个元素中较大的一个第一个版本使用与Type 相关联的大于操作符第二个版本使用比较操作comp

max_element()

```
template< class ForwardIterator >
ForwardIterator
```

```
max_element(  ForwardIterator first,
              ForwardIterator last );

template< class ForwardIterator, class Compare >
ForwardIterator
max_element(  ForwardIterator first,
             ForwardIterator last, Compare comp );
```

max_element()返回一个iterator 指向[first,last)序列中值为最大的元素第一个版本使用底层元素类型的大于操作符第二个版本使用比较操作comp

min()

```
template< class Type >
const Type&
min( const Type &aval, const Type &bval );
template< class Type, class Compare >
const Type&
min( const Type &aval, const Type &bval, Compare comp );
```

min()返回aval 和bval 两个元素中较小的一个第一个版本使用与Type 相关联的小于操作符第二个版本使用比较操作comp

min_element()

```
template< class ForwardIterator >
ForwardIterator
min_element(  ForwardIterator first,
             ForwardIterator last );
template< class ForwardIterator, class Compare >
ForwardIterator
min_element(  ForwardIterator first,
              ForwardIterator last, Compare comp );
```

min_element()返回一个iterator 指向[first,last)序列中值为最小的元素第一个版本使用底层元素类型的小于操作符第二个版本使用比较操作comp

merge()

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator
merge( InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result );
```

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare >
OutputIterator
merge( InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result, Compare comp );
```

merge()把两个分别由[first1,last1)和[first2,last2)标记的有序序列合并到一个从 result 开始的单个序列中并返回一个OutputIterator 指向新序列中最后一个元素的下一位置第一个版本使用底层类型的小于操作符对元素进行排序第二个版本根据comp 对元素进行排序

mismatch()

```
template< class InputIterator1, class InputIterator2 >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first1,
          InputIterator1 last, InputIterator2 first2 );

template< class InputIterator1, class InputIterator2,
          class BinaryPredicate >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first1, InputIterator1 last,
          InputIterator2 first2, BinaryPredicate pred );
```

mismatch()并行地比较两个序列指出第一个元素不匹配的位置它返回一对iterator 标识出第一个元素不匹配的位置如果所有的元素都匹配则返回指向每个容器last 元素的iterator 例如已知序列meet 和meat 则两个被返回的iterator 分别指向第三个元素缺省情况下用等于操作符对元素进行比较第二个版本允许用户指定一个比较操作如果第二个序列比第一个序列的元素多这些元素将被忽略如果第二个序列比第一个序列的元素少则运行时刻的行为是未定义的

next_permutation()

```
template< class BidirectionalIterator >
bool
next_permutation( BidirectionalIterator first,
                  BidirectionalIterator last );

template< class BidirectionalIterator, class Compare >
bool
next_permutation( BidirectionalIterator first,
                  BidirectionalIterator last, Compare comp );
```

next_permutation()取出由[first,last)标记的排列并将其重新扫序为下一个排列关

于怎样确定上一个排列的讨论见12.5.4 节如果不存在下一个排列则返回false 否则返回true 第一个版本使用底层类型的小于操作符来确定下一个排列第二个版本根据comp 对元素进行排序如果原始字符串是排过序的则连续调用next_permutation()生成整个排列集合例如在下列程序中如果我们不能把musil 排序成ilmsu 则不能生成排列的全集

nnth_element()

```
template< class RandomAccessIterator >
void
nth_element( RandomAccessIterator first,
             RandomAccessIterator nth,
             RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
nth_element( RandomAccessIterator first,
             RandomAccessIterator nth,
             RandomAccessIterator last, Compare comp );
```

nnth_element()将[first,last)标记的序列重新排序使所有小于第n 个元素的元素都出现在它前面而大于它的元素出现在它后面例如已知数组

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

下面的nth_element()调用使第七个元素为第n 个它的值是26

```
nth_element( &ia[0], &ia[6], &ia[12] );
```

产生一个序列其中小于26 的七个元素在它的左边余下大于26 的四个元素在它的右边{23,20,22,17,15,19,12,26,51,35,40,29} 但是第n 个元素两边的元素并不保证存在某种特定的顺序第一个版本使用底层类型的小于操作符第二个版本根据程序员传递的二元比较操作对元素调整顺序

partial_sort()

```
template< class RandomAccessIterator >
void
partial_sort( RandomAccessIterator first,
             RandomAccessIterator middle
             RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
partial_sort( RandomAccessIterator first,
             RandomAccessIterator middle,
             RandomAccessIterator last, Compare comp );
```

partial_sort()对整个序列作部分排序被排序元素的个数正好可以被放到[first,middle)范围内在[middle,last)中的元素是未经排序的它们都落在实际被排序

的序列之外例如已知数组

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

调用partial_sort() 使第六个元素为middle

```
stable_sort( &ia[0], &ia[5], &ia[12] );
```

则产生了一个序列其中五个最小的元素被排序即middle-first 个元素 {12,15,17,19,20,29,23,22,26,51,35,40} 从middle 到last-1 的元素并没有按任何特定的顺序但是它们的值都落在实际被排序的序列之外第一个版本用底层类型的小于操作符第二个版本根据comp 对元素进行排序

partial_sort_copy()

```
template< class InputIterator, class RandomAccessIterator >
RandomAccessIterator
partial_sort_copy( InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last );

template< class InputIterator, class RandomAccessIterator,
          class Compare >
RandomAccessIterator
partial_sort_copy( InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last,
                  Compare comp );
```

partial_sort_copy()的行为与partial_sort()相同只不过它把经过部分排序的序列拷贝到由[result_first,result_last)标记的容器中因此如果我们指定了一个独立的容器去接受拷贝则结果是一个完全排序的序列例

partial_sum()

```
template < class InputIterator, Class OutputIterator >
OutputIterator
partial_sum(
          InputIterator first, InputIterator last,
          OutputIterator result );

template < class InputIterator, Class OutputIterator,
class BinaryOperation >
OutputIterator
partial_sum(
          InputIterator first, InputIterator last,
          OutputIterator result, BinaryOperation op );
```

partial_sum()的第一个版本创建一个新的元素序列其中每个新元素的值代表了

[first,last)序列中这位置之前包括该位置所有元素的和例如已知序列{0,1,1,2,3, 5,8} 则新序列是{0,1,2,4,7,12,20} 例如第四个元素是前三个值{0,1,1}的部分和加上它自己(2) 产生值4第二个版本使用程序员传递的二元操作例如已知序列{1,2,3,4} 我们传递函数对象times<int> 结果序列是{1,2,6,24} 在两个版本中OutputIterator 指向新序列末元素的下一个位置partial_sum()是一个算术算法我们必须包含标准头文件<numeric>

prev_permutation()

```
template < class BidirectionalIterator  >
bool
prev_permutation( BidirectionalIterator  first,
                  BidirectionalIterator  last );
template < class BidirectionalIterator,  class Compare >
bool
prev_permutation( BidirectionalIterator  first,
                  BidirectionalIterator  last, Compare comp );
```

prev_permutation 取出由[first,last)标记的排列并将它重新排序为上一个排列关于怎样判断上一个排列的讨论见12.5.4 节如果不存在上一个排列则返回false 否则返回true第一个版本使用底层类型的小于操作符来确定上一个排列第二个版本根据程序员传递的二元比较操作对元素进行排序

random_shuffle()

```
template< class RandomAccessIterator >
void
random_shuffle( RandomAccessIterator first,
                RandomAccessIterator last );

template< class RandomAccessIterator,
          class RandomNumberGenerator >
void
random_shuffle( RandomAccessIterator first,
                RandomAccessIterator last,
                RandomNumberGenerator rand );
```

random_shuffle()对[first,last)范围内的元素随机调整顺序第二个版本使用一个专门产生随机数的函数对象或函数指针rand 返回一个double 类型的位于区间[0,1]内的值

remove()

```
template< class ForwardIterator, class Type >
```

```
ForwardIterator
remove( ForwardIterator first,
        ForwardIterator last, const Type &value );
```

remove()删除在[first,last)范围内的所有value 实例remove()以及remove_if()并不真正地把匹配到的元素从容器中清除即容器的大小保留不变而是每个不匹配的元素依次被赋值给从first 开始的下一个空闲位置上返问的ForwardIterator 标记了新的元素范围的下一个位置例如考虑序列{0,1,0,2,0,3,0,4} 假设我们希望删除所有的0 则结果序列是{1,2,3,4,0,3,0,4} 1 被拷贝到第一个位置上2 被拷贝到第二个位置上3 被拷贝到第三个位置上4 被拷贝到第四个位置上返回的ForwardIterator 指向第五个位置上的0 典型的做法是该iterator 接着被传递给erase() 以便删除无效的元素内置数组不适合于使用remove()和remove_if()算法因为它们不能很容易地被改变大小由于这个原因对于数组而言remove_copy()和remove_copy_if()是更受欢迎的算法

remove_copy()

```
template< class InputIterator, class OutputIterator,
          class Type >
OutputIterator
remove_copy( InputIterator first, InputIterator last,
             OutputIterator result, const Type &value );
```

remove_copy()把所有不匹配的元素都拷贝到由result 指定的容器中返回的OutputIterator 指向被拷贝的末元素的下一个位置但原始容器没有被改变

remove_if()

```
template< class ForwardIterator, class Predicate >
ForwardIterator
remove_if( ForwardIterator first,
           ForwardIterator last, Predicate pred );
```

remove_if()删除所有在[first,last)范围内并且pred 计算结果为true 的元素remove_if()以及remove()并不真正地把匹配到的元素从容器中清除而是将每个不匹配的元素依次赋值给从first 开始的下一个空闲位置上返回的ForwardIterator 标记了新的元素范围的下一个位置一般是将这个iterator 传递给erase() 以便真正地删除掉无效的元素remove_copy_if()更加适用于内置数组

remove_copy_if()

```
template< class InputIterator, class OutputIterator,
          class Predicate >
OutputIterator
remove_copy_if( InputIterator first, InputIterator last,
                    OutputIterator result, Predicate pred );
```
remove_copy_if()把所有不匹配的元素拷贝到由result 指定的容器中返回的OutputIterator标记了被拷贝的末元素的下一个位置原始容器没有被改变

replace()

```
template< class ForwardIterator, class Type >
void
replace( ForwardIterator first, ForwardIterator last,
const Type& old_value, const Type& new_value );
```
replace()将[first,last)范围内的所有old_value 实例都用new_value 替代

replace_copy()

```
template< class InputIterator, class OutputIterator,
          class Type >
OutputIterator
replace_copy( InputIterator first, InputIterator last,
              OutputIterator result,
              const Type& old_value, const Type& new_value );
```
replace_copy()的行为与replace()类似只不过是把新序列拷贝到由result 开始的容器内返回的OutputIterator 指向被拷贝的末元素的下一个位置但原始序列没有被改变

replace_if()

```
template< class ForwardIterator, class Predicate, class Type >
void
replace_if( ForwardIterator first, ForwardIterator last,
        Predicate pred, const Type& new_value );
```
replace_if()将[first,last)范围内的pred 计算结果为true 的所有元素都用new_value 替代

replace_copy_if()

```
template< class ForwardIterator, class OutputIterator,
           class Predicate, class Type >
OutputIterator
replace_copy_if( ForwardIterator first, ForwardIterator last,
                 OutputIterator result,
                 Predicate pred, const Type& new_value );
```
replace_copy_if()的行为与replace_if()类似只不过是把新序列拷贝到由result 开始

的容器中返回的OutputIterator 指向被拷贝的末元素的下一个位置原始序列没有被改变

reverse()

```
template< class BidirectionalIterator >
void
reverse( BidirectionalIterator first,
         BidirectionalIterator last );
```

reverse()对于容器中[first,last)范围内的元素重新按反序排列例如已知序列{0,1,1,2,3}则反序序列是{3,2,1,1,0}

reverse_copy()

```
template< class BidirectionalIterator, class OutputIterator >
OutputIterator
reverse_copy( BidirectionalIterator first,
              BidirectionalIterator last, OutputIterator result );
```

reverse_copy()的行为与reverse()类似只不过把新序列拷贝到由result 开始的容器中返回的OutputIterator 指向被拷贝的元素的下一个位置原始序列没有被改变

rotate()

```
template< class ForwardIterator >
void
rotate( ForwardIterator first,
        ForwardIterator middle, ForwardIterator last );
```

rotate()把[first,middle)范围内的元素移到容器末尾由middle 指向的元素成为容器的第一个元素例如已知单词hissboo 则以元素b 为轴的旋转将单词变成boohiss

rotate_copy()

```
template< class ForwardIterator, class OutputIterator >
OutputIterator
rotate_copy( ForwardIterator first, ForwardIterator middle,
             ForwardIterator last, OutputIterator result );
```

rotate_copy()的行为与rotate()类似只不过把旋转后的序列拷贝到由result 标记的容器中返回的OutputIterator 指向被拷贝的末元素的下一个位置原始序列没有被改变

search()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator
search( ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2 );

template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator
search( ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred );
```

给出了两个范围search()返回一个iterator 指向在[first1,last1)范围内第一次出现子序列[first2,last2]的位置如果子序列未出现则返回last1 例如在mississippi 中子序列iss出现两次则search()返回一个iterator 指向第一个实例的起始处缺省情况下使用等于操作符进行元素的比较第二个版本允许用户提供一个比较操作

search_n()

```
template< class ForwardIterator, class Size, class Type >
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
          Size count, const Type &value );

template< class ForwardIterator, class Size,
          class Type, class BinaryPredicate >
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
          Size count, const Type &value, BinaryPredicate pred );
```

search_n()在[first,last)序列中查找value 出现count 次的子序列如果没有找到value的count 次出现则返回last 例如为了在序列mississippi 中找到了序列ss value 将被设置为s 而count 为2 为了找到子串ssi 的两个实例value 应该为ssi 而count仍是2 search_n()返回一个iterator 指向被找到的value 的第一个元素缺省情况下使用等于操作符来比较元素第二版本允许用户提供一个比较操作

set_difference()

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator >
OutputIterator
set_difference( InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result );

template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator
set_difference( InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result, Compare comp );
```

set_difference()构造一个排过序的序列其中的元素出现在第一个序列中由[first,last)标记但是不包含在第二个序列中由[first2,last2]标记例如已失两个序列{0,1,2,3}和{0,2,4,6} 则差集为{1,3} 返回的OutputIterator 指向被放入result 所标记的容器中的最后元素的下一个位置第一个版本假设该序列是用底层元素类型的小于操作符来排序的第二个版本假设该序列是用comp 来排序的

set_intersection()

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator >
OutputIterator
set_intersection( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result );

template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator
set_intersection( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result, Compare comp );
```

set_intersection()构造一个排过序的序列其中的元素在[first1,last1)和[first2,last2)序列中都存在例如已知序列{0,1,2,3}和{0,2,4,6} 则交集为{0,2} 这些元素被从第一个序列中拷贝出来返回的OutputIterator 指向被放入result 所标记的容器内的最后元素的下一个位置第一个版本假设该序列是用底层类型的小于操作符来排序的而第二个版本假设该序列是用comp 来排序的

set_symmetric_difference()

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator >
OutputIterator
set_symmetric_difference(
             InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result );

template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator
set_symmetric_difference(
              InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result, Compare comp );
```

set_symmetric_difference()构造一个排过序的序列其中的元素在第一个序列中出现但不出现在第二个序列中或者在第二个序列中出现但不出现在第一个序列中例如已知两个序列{0,1,2,3}和{0,2,4,6} 则对称差集是{1,3,4,6} 返回的OutputIterator 指向被放入result所标记的容器内的最后元素的下一个位置第一个版本假设该序列是用底层类型的小于操作符来排序的而第二个版本假设该序列是用comp 来排序的

set_union()

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator >
OutputIterator
set_union( InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result );
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator
set_union( InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result, Compare comp );
```

set_union()构造一个排过序的序列它包含了[first1,last1)和[first2,last2)这两个范围内的所有元素例如已知两个序列{0,1,2,3}和{0,2,4,6} 则并集为{0,1,2,3,4,6} 如果一个元素在两个容器中都存在比如0 和2 则拷贝第一个容器中的元素返回的OutputIterator 指向被放入result 所标记的容器内的最后元素的下一个位置第一个版本假设该序列是用底层类型的小于操作符来排序的而第二个版本假设该序列是用comp 来排序的

sort()

```
template< class RandomAccessIterator >
void
sort( RandomAccessIterator first,
        RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
sort( RandomAccessIterator first,
        RandomAccessIterator last, Compare comp );
```

sort()利用底层元素的小于操作符以升序重新排列[first,last)范围内的元素第二版本根据comp 对元素进行排序为了保留相等元素之间的顺序关系要使用stable_sort() 而不是sort() 我们不提供专门的程序来说明sort()的用法因为它在许多其他的例子中会被用到比如binary_search() equal_range()和inplace_merge()

stable_partition()

```
template< class BidirectionalIterator, class Predicate >
BidirectionalIterator
stable_partition( BidirectionalIterator first,
                  BidirectionalIterator last,
                  Predicate pred );
```

stable_partition()的行为与partition()类似只不过它保证会保留容器中元素的相对顺序下面是与partition()的例子相同的一个程序但是它被修改为调用stable_partition()

stable_sort()

```
template< class RandomAccessIterator >
void
stable_sort( RandomAccessIterator first,
             RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
stable_sort( RandomAccessIterator first,
             RandomAccessIterator last, Compare comp );
```

stable_sort()利用底层类型的小于操作符以升序重新排列[first,last)范围内的元素并且保留相等元素之间的顺序关系第二版本根据comp 对元素进行排序

swap()

```
template< class Type >
void
swap ( Type &ob1, Type &ob2 );
```

swap()交换存贮在对象ob1 和ob2 中的值

swap_range()

```
template <class ForwardIterator1, class ForwardIterator2 >
ForwardIterator2
swap_range( ForwardIterator1 first1, ForwardIterator1 last,
                ForwardIterator2 first2 );
```

swap_range()将[first,last)标记的元素值与从first2 开始相同个数的元素值进行交
976 附录泛型算法换这两个序列可以是同一容器中不相连的序列也可以位于两个独立的容器中如果从first2开始的序列小于由[first1,last)标记的序列或者两个序列在同一容器中有重叠则该算法的运行时刻行为是未定义的swap_range()返回第二个序列的iterator 指向最后一个被交换的元素的下一个位置

transform()

```
template< class InputIterator, class OutputIterator,
             class UnaryOperation >
OutputIterator
transform( InputIterator first, InputIterator last,
             OutputIterator result, UnaryOperation op );

template< class InputIterator1, class InputIterator2,
           class OutputIterator, class BinaryOperation >
OutputIterator
transform( InputIterator1 first1, InputIterator1 last,
             InputIterator2 first2, OutputIterator result,
             BinaryOperation bop );
```

transform()的第一个版本将op 作用在[first,last)范围内的每个元素上从而产生一个新的序列例如已知序列{0,1,1,2,3,5}和函数对象Double 它使每个元素加倍那么结果序列是{0,2,2,4,6,10}第二个版本将bop 作用在一对元素上其中一个元素来自序列[first1,last) 另一个来自由first2 开始的序列最终产生一个新的序列如果第二个序列包含的元素少于第一个序列则运行时刻行为是未定义的例如已知序列{1,3,5,9}和{2,4,6,8} 以及函数对象AddAndDouble它把两个元素相加并将和加倍则结果序列是{6,14,22,34}两个版本的transform()都把结果序列放在由result 标记的容器中result 可以指向两个输入容器之一则实际达到的效果是用transform()返回的元素取代当前的元素返回的OutputIterator 指向最后被赋给result 的元素的下一个位置

unique()

```
template< class ForwardIterator >
ForwardIterator
unique( ForwardIterator first,
        ForwardIterator last );

template< class ForwardIterator, class BinaryPredicate >
ForwardIterator
unique( ForwardIterator first,
        ForwardIterator last, BinaryPredicate pred );
```
对于连续的元素如果它们包含相同的值使用底层类型的等于操作符来判断或者把它们传给pred 的计算结果都为true 则这些元素被折叠成一个元素例如在单词mississippi中语义上的结果是misisipi 注意四个i 不是连续的所以不会被折叠类似地因为两对s 也是不连续的所以也没有被折叠成单个实例为了保证所有重复的实例都被折叠起来我们必须先对容器进行排序实际上unique()的行为有些不太直观类似于remove()算法在这两种情况下容器的实际大小并没有变化每个惟一的元素都被依次拷贝到从first 开始的下一个空闲位置上因此在我们的例子中实际的结果是misisipippi 这里的ppi 字符序列可以说是算法的残留物返回的ForwordIterator 指向残留物的起始处典型的做法是这个iterator 被传递给erase()以便删除无效的元素由于内置数组不支持erase()操作所以unique()不太适合于数组unique_copy()对数组更为合适一些

unique_copy()

```
template< class InputIterator, class OutputIterator >
OutputIterator
unique_copy( InputIterator first, InputIterator last,
             OutputIterator result );

template< class InputIterator, class OutputIterator,
          class BinaryPredicate >
OutputIterator
unique_copy( InputIterator first, InputIterator last,
             OutputIterator result, BinaryPredicate pred );
```
unique_copy()把每组含有相同的值使用底层类型的等于操作符来判断或被传递给pred 时计算结果为true 描述见unique() 的连续元素拷贝一个实例为了保证所有重复的元素都被清除掉必须先对容器进行排序返回的OutputIterator 指向目标容器的尾部

upper_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
upper_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );
template< class ForwardIterator, class Type, class Compare >
ForwardIterator
```

```
upper_bound( ForwardIterator first,
                         ForwardIterator last, const Type &value,
                         Compare comp );
```
upper_bound()返回一个iterator 它指向在[first,last)标记的有序序列中可以插入value而不会破坏容器顺序的最后一个位置这个位置标记了一个大于value 的值例如已知序列

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```
用值21 调用upper_bound() 返回一个指向值22 的iterator 用值22 调用upper_bound()则返回一个指向值23 的iterator 第一个版本使用底层类型的小于操作符而第二个版本根据comp 对元素进行排序和比较

make_heap()

```
template< class RandomAccessIterator >
void
make_heap( RandomAccessIterator first,
                   RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
make_heap( RandomAccessIterator first,
                     RandomAccessIterator last, Compare comp );
```
make_heap()把[first,last)范围内的元素做成一个堆双参数版本使用底层类型的小于操作符作为排序准则第二个版本根据comp 对元素进行排序

pop_heap()

```
template< class RandomAccessIterator >
void
pop_heap( RandomAccessIterator first,
                 RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
pop_heap( RandomAccessIterator first,
                 RandomAccessIterator last, Compare comp );
```
pop_heap()并不真正地把最大元素从堆中弹出而是重新排序堆它把first 和last-1 交换然后将[first,last-1)范围的序列重新做成一个堆之后我们就可以用容器的成员操作back()来访问被弹出的元素或者用pop_back()将它真正删除掉双参数版本使用底层类型的小于操作符作为排序难则第二个版本根据comp 对元素进行排序

push_heap()

```
template< class RandomAccessIterator >
void
push_heap( RandomAccessIterator first,
           RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
push_heap( RandomAccessIterator first,
            RandomAccessIterator last, Compare comp );
```

push_heap()假设由[first,last-1)标记的序列是一个有效的堆要被加到堆中的新元素在位置last-1 上它将[first,last)序列重新做成一个堆在调用push_heap()之前我们必须先把元素插入到容器的后面或许可以使用push_back()操作符这将在下一个程序例子中说明双参数版本使用底层类型的小于操作符作为排序准则第二个版本根据comp 对元素进行排序

sort_heap()

```
template< class RandomAccessIterator >
void
sort_heap( RandomAccessIterator first,
           RandomAccessIterator last );

template< class RandomAccessIterator, class Compare >
void
sort_heap( RandomAccessIterator first,
           RandomAccessIterator last, Compare comp );
```

sort_heap()对范围[first,last)中的序列进行排序它假设该序列是一个有效的堆否则它的行为是未定义的当然经过排序之后的堆就不再是一个有效的堆双参数版本使用底层类型的小于操作符作为排序准则第二个版本根据comp 对元素进行排序

字符串处理

KMP 算法举例

```
#include <iostream>
#include <cstdio>
#include <string>
```

```
//计算next数组
static int* __getNextofString(const std::string& p)
{
    int size=p.size();
    int* next=new int[size+1];
    int i, j;
    i=1; next[1]=0; j=0;
    while ( i<size )
    {
        if ( j==0 || p[i-1]==p[j-1] )
        {
            ++i; ++j;
            next[i]=j;
        }
        else
            j=next[j];
    }
    return next;
}
//查找
static int __findSubString(const std::string& t, const std::string& p,
int* next, int index=1)
{
    int i, j;
    int tlen, plen;
    tlen=t.size(); plen=p.size();
    i=index; j=1;
    while ( i<=tlen && j<=plen )
    {
        if ( j==0 || t[i-1]==p[j-1] )
        { ++i; ++j; }
        else
            j=next[j];
    }
    if ( j>plen )
        return i-plen;
    else
        return 0;
}
//主函数
int IndexSubString(const std::string& text, const std::string& pattern,
int* next=NULL, int pos=0)
{
    bool bAlloc=false;
    if (next==NULL)
    {
        bAlloc=true;
        next=__getNextofString(pattern);
    }
    int ret=__findSubString(text, pattern, next, pos+1)-1;
    if (bAlloc)
        delete[] next;
    return ret;
}
//使用方法
int main (int argc, char* argv[])
{
    std::string text, pattern;
    while ( std::cin>>text>>pattern )
    {
        std::cout<<IndexSubString(text, pattern)<<std::endl;
    }
    return 0;
}
```

C++语言可用头文件

<algorithm>

Defines Standard Template Library (STL) container template functions that perform algorithms.

<bitset>

Defines the template class bitset and two supporting template functions for representing and manipulating fixed-size sequences of bits.

<complex>

Defines the container template class complex and its supporting templates.

<deque>

Defines the container template class deque and several supporting templates.

<exception>

Defines several types and functions related to the handling of exceptions. Exception handling is used in situations in which the system can recover from an error. It provides a means for control to be returned from a function to the program. The objective of incorporating exception handling is to increase the program's robustness while providing a way to recover from an error in an orderly fashion.

<fstream>

Defines several classes that support iostreams operations on sequences stored in external files.

<functional>

Defines Standard Template Library (STL) functions that help construct function objects, also known as functors, and their binders.

<iomanip>

Define several manipulators that each take a single argument.

<ios>

Defines several types and functions basic to the operation of iostreams. This header is typically included for you by another iostream headers; you rarely include it directly.

<iosfwd>

Declares forward references to several template classes used throughout iostreams. All such template classes are defined in other standard headers. You include this header explicitly only when you need one of its declarations, but not its definition.

<iostream>

Declares objects that control reading from and writing to the standard streams. This is often the only header you need to include to perform input and output from a C++ program.

<iso646.h>

Provides readable alternatives to certain operators or punctuators. The standard header <iso646.h> is available even in a freestanding implementation.

<istream>

Defines the template class basic_istream, which mediates extractions for the iostreams, and the template class basic_iostream, which mediates both insertions and extractions. The header also defines a related manipulator. This header file is typically included for you by another iostreams header; you rarely have to include it directly.

<iterator>

Defines the iterator primitives, predefined iterators and stream iterators, as well as several supporting templates. The predefined iterators include insert and reverse adaptors. There are three classes of insert iterator adaptors: front, back, and general. They provide insert semantics rather than the overwrite semantics that the container member function iterators provide.

<limits>

Defines the template class numeric_limits and two enumerations concerning floating-point representations and rounding.

<list>

Defines the container template class list and several supporting templates.

<locale>

Defines template classes and functions that C++ programs can use to encapsulate and manipulate different cultural conventions regarding the representation and formatting of numeric, monetary, and calendric data, including internationalization support for character classification and string collation.

<map>

Defines the container template classes map and multimap and their supporting templates.

<memory>

Defines a class, an operator, and several templates that help allocate and free objects.

<new>

Defines several types and functions that control the allocation and freeing of storage under program control. It also defines components for reporting on storage management errors.

<numeric>

Defines container template functions that perform algorithms provided for numerical processing.

<ostream>

Defines the template class basic_ostream, which mediates insertions for the iostreams. The header also defines several related manipulators. (This header is typically included for you by another of the iostreams headers. You rarely need to include it directly.)

<queue>

Defines the template classes priority_queue and queue and several supporting templates.

<set>

Defines the container template classes set and multiset and their supporting templates.

<sstream>

Defines several template classes that support iostreams operations on sequences stored in an allocated array object. Such sequences are easily converted to and from objects of template class basic_string.

<stack>

Defines the template class stack and two supporting templates.

<stdexcept>

Defines several standard classes used for reporting exceptions. The classes form a derivation hierarchy all derived from class exception and include two general types of exceptions: logical errors and run-time errors. The logical errors are caused programmer mistakes. They derive from the base class logic_error and include:

<streambuf>

Include the iostreams standard header <streambuf> to define the template class basic_streambuf, which is basic to the operation of the iostreams classes. This header is typically included for you by another of the iostreams headers; you rarely need to include it directly.

<string>

Defines the container template class basic_string and various supporting templates.

<strstream>

Defines several classes that support iostreams operations on sequences stored in an allocated array of char object. Such sequences are easily converted to and from C strings.

<utility>

Defines Standard Template Library (STL) types, functions, and operators that help to construct and manage pairs of objects, which are useful whenever two objects need to be treated as if they were one.

<valarray>

Defines the template class valarray and numerous supporting template classes and functions.

<vector>

Defines the container template class vector and several supporting templates.

<cassert>

Effectively includes the Standard C library header <assert.h>.

<cctype>

Defines the macros traditionally defined in the Standard C library header <ctype.h>.

<cerrno>

Effectively includes the Standard C library header <errno.h>.

<cfloat>

Effectively includes the Standard C library header <float.h>.

<ciso646>

Effectively includes the standard header <iso646.h>.

<climits>

Effectively includes the Standard C library header <limits.h>.

<clocale>

Defines the macros traditionally defined in the Standard C library header <locale.h>.

<cmath>

Defines the macros traditionally defined in the Standard C library header <math.h>.

<csetjmp>

Defines the macros traditionally defined in the Standard C library header <setjmp.h>.

<csignal>

Defines the macros traditionally defined in the Standard C library header <signal.h>.

<cstdarg>

Defines the macros traditionally defined in the Standard C library header <stdarg.h>.

<cstddef>

Defines the macros traditionally defined in the Standard C library header <stddef.h>.

<cstdio>

Defines the macros traditionally defined in the Standard C library header <stdio.h>.

<cstdlib>

Defines the macros traditionally defined in the Standard C library header <stdlib.h>.

<cstring>

Defines the macros traditionally defined in the Standard C library header <string.h>.

<ctime>

Defines the macros traditionally defined in the Standard C Library header <time.h>.

<cwchar>

Defines the macros traditionally defined in the Standard C library header <wchar.h>.

<cwctype>

Defines the macros traditionally defined in the Standard C library header <wctype.h>.