

常用算法模板库 (C++)

目录

目录.....	1
1 排序算法.....	3
1.1 冒泡排序.....	3
1.2 选择排序.....	4
1.3 插入排序.....	5
1.4 快速排序.....	6
1.5 哈希排序.....	8
2 数学问题.....	8
2.1 求最大公约数最小公倍数.....	8
2.2 求素数.....	9
2.2.1 穷举法.....	9
2.2.2 筛法.....	10
2.3 排列组合.....	10
2.3.1 排列数.....	10
2.3.2 组合数.....	11
2.3.3 全排列算法.....	12
2.3.4 从 n 个数中选 m 个的组合.....	13
2.4 进制转换.....	14
2.4.1 十进制转 N 进制.....	14
2.4.2 N 进制转十进制.....	14
3 查找.....	15
3.1 二分查找.....	15
4 栈.....	16
4.1 栈的定义.....	16
4.2 表达式求值.....	19
5 队列.....	21
5.1 队列的定义.....	21
6 串.....	24
6.1 基本操作.....	24
6.1.1 赋值.....	24
6.1.2 比较大小 (字典顺序).....	24
6.2 模式匹配.....	25

6.2.1 一般匹配算法.....	25
6.2.2 KMP 算法.....	26
7 树.....	27
7.1 二叉树的定义.....	27
7.2 根据先序序列和中序序列建立二叉树.....	31
7.3 二叉排序树	32
7.4 建立哈夫曼树.....	32
7.5 树的定义	34
8 图.....	38
8.1 图的定义	38
8.2 图的最短路径.....	43
8.2.1 Dijkstra 算法.....	43
8.2.2 Floyd 算法.....	44
8.3 最小生成树	44
8.3.1 prim 算法.....	44
8.3.2 Kruskal 算法	45
8.4 拓扑排序	46
8.5 关键路径	47
9 高精度计算.....	48
9.1 加法.....	48
9.2 减法.....	49
9.3 乘法.....	50

1 排序算法

1.1 冒泡排序

/*

函数功能：对数组中的某一部分进行冒泡排序。

函数原形：void BubSort(DataType a[], int l, int r, bool Up=true);

参数：

DataType a[]：欲排序的数组；

int l：有序序列在数组中的起始位置；

int r：有序序列在数组中的结束位置；

bool Up：true 按升序排列，false 按降序排列。

返回值：无。

备注：无。

*/

```
template <typename DataType>
```

```
void BubSort(DataType a[], int l, int r, bool Up=true)
```

```
{
```

```
    int i, j;
```

```
    DataType k;
```

```
    if (Up)
```

```
    {
```

```
        for (i=l; i<=r-1; i++)
```

```
            for (j=r; j>=i+1; j--)
```

```
                if (a[j-1]>a[j])
```

```
                {
```

```
                    k=a[j-1];
```

```
                    a[j-1]=a[j];
```

```
                    a[j]=k;
```

```
                }
```

```
    }
```

```
    else
```

```
        for (i=l; i<=r-1; i++)
```

```
            for (j=r; j>=i+1; j--)
```

```
                if (a[j-1]<a[j])
```

```
                {
```

```
                    k=a[j-1];
```

```
                    a[j-1]=a[j];
```

```
                    a[j]=k;
```

```

    }
}

```

1.2 选择排序

/*

函数功能：对数组中的某一部分进行选择排序。

函数原形：void SelectSort(DataType a[], int l, int r, bool Up=true);

参数：

DataType a[]：欲排序的数组；

int l：有序序列在数组中的起始位置；

int r：有序序列在数组中的结束位置；

bool Up：true 按升序排列，false 按降序排列。

返回值：无。

备注：无。

*/

/*普通版*/

template <typename DataType>

void SelectSort(DataType a[], int l, int r, bool Up=true)

```

{
    int i, j;
    DataType k;

    if (Up)
    {
        for (i=l; i<=r-1; i++)
            for (j=i+1; j<=r; j++)
                if (a[i]>a[j])
                {
                    k=a[i];
                    a[i]=a[j];
                    a[j]=k;
                }
    }
    else
        for (i=l; i<=r-1; i++)
            for (j=i+1; j<=r; j++)
                if (a[i]<a[j])
                {
                    k=a[i];
                    a[i]=a[j];
                    a[j]=k;
                }
}

```

```

/*优化版*/
template <typename DataType>
void SelectSort(DataType a[], int l, int r, bool Up=true)
{
    int i, j, k;
    DataType x;

    if (Up)
    {
        for (i=l; i<=r-1; i++)
        {
            k=i;
            for (j=i+1; j<=r; j++)
                if (a[j]<a[k]) k=j;
            x=a[i];
            a[i]=a[k];
            a[k]=x;
        }
    }
    else
        for (i=l; i<=r-1; i++)
        {
            k=i;
            for (j=i+1; j<=r; j++)
                if (a[j]>a[k]) k=j;
            x=a[i];
            a[i]=a[k];
            a[k]=x;
        }
}

```

1.3 插入排序

/*

函数功能：向有序数组中插入元素，并使插入新元素后仍然有序。

函数原形：void InsertSort(DataType s[], int &Count, DataType x, bool Up=true);

参数：

DataType s[]：欲插入元素的有序序列；

int &Count：有序序列中现有元素个数；

DataType x：欲插入的元素；

bool Up：true 按升序排列，false 按降序排列。

返回值：无。

备注：无。

*/

```

template <typename DataType>
void InsertSort(DataType s[], int &Count, DataType x, bool Up=true)
{
    int i,k=-1;

    if (Up)
    {
        for (i=0;i<=Count-1;i++)
            if (x<s[i])
            {
                k=i;
                break;
            }
    }
    else
        for (i=0;i<=Count-1;i++)
            if (x>s[i])
            {
                k=i;
                break;
            }
    if (k==--1) s[++Count-1]=x;
    else
    {
        for (i=Count-1;i>=k;i--) s[i+1]=s[i];
        s[k]=x;
        Count++;
    }
}

```

1.4 快速排序

/*

函数功能：对数组中的某一部分进行快速排序。

函数原形：void QuickSort(DataType a[], int l, int r, bool Up=true);

参数：

DataType a[]：欲排序的数组；

int l：有序序列在数组中的起始位置；

int r：有序序列在数组中的结束位置；

bool Up：true 按升序排列，false 按降序排列。

返回值：无。

备注：无。

*/

```

template <typename DataType>

```

```

void QuickSort(DataType a[], int l, int r, bool Up=true)
{
    int i, j;
    DataType Mid, k;

    i=l;
    j=r;
    Mid=a[(l+r)/2];
    if (Up)
    {
        do
        {
            while (a[i]<Mid) ++i;
            while (Mid<a[j]) --j;
            if (i<=j)
            {
                k=a[i];
                a[i]=a[j];
                a[j]=k;
                ++i;
                --j;
            }
        } while (i<=j);
    }
    else
    {
        do
        {
            while (a[i]>Mid) ++i;
            while (Mid>a[j]) --j;
            if (i<=j)
            {
                k=a[i];
                a[i]=a[j];
                a[j]=k;
                ++i;
                --j;
            }
        } while (i<=j);
    }
    if (i<r) QuickSort(a, i, r, Up);
    if (l<j) QuickSort(a, l, j, Up);
}

```

1.5 哈希排序

/*

函数功能：对数组中的某一部分进行哈希排序。

函数原形：void QuickSort(DataType a[], int l, int r, bool Up=true);

参数：

DataType a[]：欲排序的数组；

int l：有序序列在数组中的起始位置；

int r：有序序列在数组中的结束位置；

bool Up：true 按升序排列，false 按降序排列。

返回值：无。

备注：只能对 0~32767 范围内的整数排序。

*/

```
void HashSort(int a[], int l, int r, bool Up=true)
{
    int i, j, k, Hash[32767]={0};

    for (i=l; i<=r; i++) Hash[a[i]]++;
    k=l;
    if (Up)
    {
        for (i=0; i<=32767-1; i++)
        {
            for (j=1; j<=Hash[i]; j++) a[k++]=i;
            if (k>r) break;
        }
    }
    else
        for (i=32767-1; i>=0; i--)
        {
            for (j=1; j<=Hash[i]; j++) a[k++]=i;
            if (k>r) break;
        }
}
```

2 数学问题

2.1 求最大公约数最小公倍数

/*

函数功能：求两数的最大公约数。

函数原形: `int GCD(int m, int n);`

参数:

`int m, int n`: 求这两个数的最大公约数。

返回值: `int` 型, `m`、`n` 的最大公约数。

备注: 最小公倍数= $m*n/GCD(m,n)$ 。

```
*/
int GCD(int m, int n)
{
    int Temp, x, y;

    x=m;
    y=n;
    Temp=x%y;
    while (Temp!=0)
    {
        x=y;
        y=Temp;
        Temp=x%y;
    }
    return y;
}
```

2.2 求素数

2.2.1 穷举法

/*

函数功能: 判断一个数是否是素数。

函数原形: `bool Prime(int x);`

参数:

`int x`: 欲判断的数。

返回值: `bool` 型, 返回 `true` 表示 `x` 是素数, 返回 `false` 表示 `x` 不是素数。

备注: 需包含头文件 `<math.h>`。

*/

```
bool Prime(int x)
{
    int i;

    x=abs(x);
    if (x<=1) return false;
    for (i=2;i<=int(sqrt(float(x)));i++)
        if (x%i==0) return false;
    return true;
}
```

```
}
```

2.2.2 筛法

```
/*
```

函数功能：判断小于等于 n 的数是否是素数。

函数原形：void Prime(bool Hash[], int n);

参数：

bool Hash[]：最终结果，Hash[i]=true 表示 i 是素数，否则表示 i 不是素数；

int n：判断范围。

返回值：无。

备注：无。

```
*/
```

```
void Prime(bool Hash[], int n)
{
    int i, k;

    Hash[0]=false;
    Hash[1]=false;
    for (i=2; i<=n; i++) Hash[i]=true;
    i=2;
    while (i<=n/2)
    {
        k=i;
        while (k+i<=n)
        {
            k=k+i;
            Hash[k]=false;
        }
        i++;
        while (Hash[i]==false && i<n) i++;
    }
}
```

2.3 排列组合

2.3.1 排列数

```
/*
```

函数功能：排列数公式 $A(m, n)$ 。

函数原形：int A(int m, int n);

参数：

int m, int n: 排列数公式的两个参数。

返回值: int 型, 排列数。

备注: 无。

*/

```
int A(int m, int n)
{
    int i, Ans;

    Ans=1;
    for (i=1; i<=m; i++)
    {
        Ans=Ans*n;
        n--;
    }
    return Ans;
}
```

2.3.2 组合数

/*

函数功能: 组合数公式 $C(m, n)$ 。

函数原形: int C(int m, int n);

参数:

int m, int n: 组合数公式的两个参数。

返回值: int 型, 组合数。

备注: 无。

*/

```
int C(int m, int n)
{
    int i, Ans;

    Ans=1;
    for (i=1; i<=m; i++)
    {
        Ans=Ans*n;
        n--;
    }
    for (i=2; i<=m; i++) Ans=Ans/i;
    return Ans;
}
```

2.3.3 全排列算法

/*

函数功能：输出 n 的全排列。

函数原形：void Arrange(int n);

参数：

int n：全排列元素个数。

返回值：无。

备注：无。

*/

```
void Swap(int &a, int &b)
```

```
{
```

```
    int k;
```

```
    k=a;
```

```
    a=b;
```

```
    b=k;
```

```
}
```

```
void Inverse(int Num[], int l, int r)
```

```
{
```

```
    int i,Count;
```

```
    Count=(r-l+1)/2;
```

```
    for (i=1;i<=Count;i++) Swap(Num[l++],Num[r--]);
```

```
}
```

```
void Arrange(int n)
```

```
{
```

```
    int *Num=new int[n+1];
```

```
    char i, x, y;
```

```
    Num[0]=0;
```

```
    for (i=1;i<=n;i++) //输出原始序列
```

```
    {
```

```
        Num[i]=i;
```

```
        cout<<Num[i]<<' ';
```

```
    }
```

```
    cout<<'\\n';
```

```
    do
```

```
    {
```

```
        for (i=1;i<=n;i++)
```

```
            if (Num[i]>Num[i-1]) x=i;
```

```

        for (i=x;i<=n;i++)
            if (Num[i]>Num[x-1]) y=i;
        if (x>1)
        {
            Swap(Num[x-1],Num[y]); //交换 Num[x-1]和 Num[y]
            Inverse(Num, x, n); //对 num[x]~num[n]作逆序处理
            for (i=1;i<=n;i++) cout<<Num[i]<<' '; //输出当前序列
            cout<<' \n';
        }
    } while(x!=1);
    delete[] Num;
}

```

2.3.4 从 n 个数中选 m 个的组合

```

/*
函数功能：输出 n 选 m 的组合方案。
函数原形：void Combination(int m, int n);
参数：
int m：欲选出的元素个数；
int n：元素总个数。
返回值：无。
备注：无。
*/
void Combination(int m, int n)
{
    int *a=new int[n+1],i,j;

    a[0]=1;
    for (i=1;i<=m;i++) a[i]=i;
    while (a[0]==1)
    {
        for (i=1;i<=m;i++) cout<<a[i]<<' '; //输出当前序列
        cout<<' \n';
        j=m;
        while (a[j]==n-m+j && j>0) j--;
        a[j]++;
        for (i=j+1;i<=n;i++) a[i]=a[i-1]+1;
    }
    delete[] a;
}

```

2.4 进制转换

2.4.1 十进制转 N 进制

```
/*
函数功能：将一个十进制数转换成一个 N 进制数。
函数原形：string DecToN(DataType x, int n);
参数：
    DataType x: 欲转换的数;
    int n: 返回结果的进制。
返回值：string 类对象，转换结果。
备注：需包含<string>。
*/
template <typename DataType>
string DecToN(DataType x, int n)
{
    string Ans;
    int Count, i, j=0, a[64];

    Count=0;
    if (x==0)
    {
        Ans[0]='0';
        Ans[1]='\0';
    }
    else
    {
        while (x>0)
        {
            a[Count++]=x%n;
            x=x/n;
        }
        for (i=Count-1; i>=0; i--)
            if (a[i]<10) Ans.append(1, a[i]+'0');
            else Ans.append(1, a[i]+'A'-10);
    }
    return Ans;
}
```

2.4.2 N 进制转十进制

```
/*
```

函数功能：将一个 N 进制数转换成一个十进制数。

函数原形：long NToDec(string x, int n);

参数：

string x: 欲转换的数;

int n: 被转换数的进制。

返回值：long 型，转换结果。

备注：需包含<string>、<ctype.h>。

*/

```
long NToDec(string x, int n)
{
    int a[32], y, Count, i, j;
    long Ans;

    Ans=0;
    for (i=0; i<=(int)x.length()-1; i++)
        if ('0' <=x[i] && x[i]<='9') a[i]=x[i]-'0';
        else a[i]=toupper(x[i])-'A'+10;
    Count=0;
    for (i=(int)x.length()-1; i>=0; i--)
    {
        y=1;
        for (j=1; j<=Count; j++) y=y*n;
        Count++;
        Ans=Ans+y*a[i];
    }
    return Ans;
}
```

3 查找

3.1 二分查找

/*

函数功能：查找指定元素在数组中的位置。

函数原形：int BinarySearch(DataType a[], DataType x, int l, int r);

参数：

DataType a[]: 目标数组;

DataType x: 待查找的元素;

int l: 数组下标上限;

int r: 数组下标下限。

返回值：int 型，元素 x 在数组 a 中的下标。

备注：无。

```

*/
template <typename DataType>
int BinarySearch(DataType a[], DataType x, int l, int r)
{
    int Mid;

    while (l<=r)
    {
        Mid=(l+r)/2;
        if (x==a[Mid]) return Mid;
        else
        {
            if (x<a[Mid]) r=Mid-1;
            if (x>a[Mid]) l=Mid+1;
        }
    }
    return -1;
}

```

4 栈

4.1 栈的定义

```

template <typename DataType>
class StackClass
{
private:
    struct BodyNodeStruct
    {
        DataType Data;
        BodyNodeStruct *Pre,*Next;
    } Bottom,*Top;

public:
    StackClass(); //构造函数, 创建一个空栈
    bool Empty(); //判断栈是否为空, 若为空则返回 true, 否则返回 false
    int Push(DataType Data); //将 Data 压栈, 如果压栈成功则返回 0, 否则返回 1
    int Pop(DataType &Data); //弹出栈顶元素, 保存在 Data 中, 若栈不为空则返回 0,
    否则返回 1
    int GetTop(DataType &Data); //读取栈顶元素, 保存在 Data 中, 若栈不为空则返回
    0, 否则返回 1
    int Size(); //返回栈中元素个数

```

```

        void Clear(); //清空栈
        ~StackClass(); //析构函数
    } ;

template <typename DataType>
StackClass<DataType>::StackClass()
{
    Top=&Bottom;
    Top->Pre=NULL;
    Top->Next=NULL;
}

template <typename DataType>
bool StackClass<DataType>::Empty()
{
    if (Top==&Bottom) return true;
    else return false;
}

template <typename DataType>
int StackClass<DataType>::Push(DataType Data)
{
    BodyNodeStruct *Temp=new BodyNodeStruct;

    if (Temp!=NULL)
    {
        Temp->Pre=Top;
        Temp->Next=NULL;
        Top->Next=Temp;
        Top->Data=Data;
        Top=Temp;
        return 0;
    }
    else return 1;
}

template <typename DataType>
int StackClass<DataType>::Pop(DataType &Data)
{
    if (!Empty())
    {
        Top=Top->Pre;
        Data=Top->Data;
        delete Top->Next;
    }
}

```

```

        Top->Next=NULL;
        return 0;
    }
    else return 1;
}

template <typename DataType>
int StackClass<DataType>::GetTop(DataType &Data)
{
    if (!Empty())
    {
        Data=Top->Pre->Data;
        return 0;
    }
    else return 1;
}

template <typename DataType>
int StackClass<DataType>::Size()
{
    BodyNodeStruct *Temp=&Bottom;
    int Count=0;

    while (Temp!=Top)
    {
        Count++;
        Temp=Temp->Next;
    }
    return Count;
}

template <typename DataType>
void StackClass<DataType>::Clear()
{
    DataType Temp;

    while (!Empty()) Pop(Temp);
}

template <typename DataType>
StackClass<DataType>::~StackClass()
{
    Clear();
}

```

4.2 表达式求值

/*

函数功能：计算四则运算表达式的值。

函数原形：long Evaluate(string Exp);

参数：

string Exp: 欲求值的四则运算表达式。

返回值：long 型，表达式的值。

备注：需包含<string>。

*/

```
long Evaluate(string Exp)
{
    StackClass<int> Optr;
    StackClass<long> Opnd;
    long a, b, Ans;
    int
TempOptr, x, i, Length, Operator[7][7]={ {2, 2, 0, 0, 0, 2, 2}, {2, 2, 0, 0, 0, 2, 2}, {2, 2, 2, 2, 0,
2, 2}, {2, 2, 2, 2, 0, 2, 2}, {0, 0, 0, 0, 0, 1, -1}, {2, 2, 2, 2, -1, 2, 2}, {0, 0, 0, 0, 0, -1, 1}};
    char TempOpnd[6];

    Optr.Push(6);
    Opnd.Push(0);
    Length=0;
    i=0;
    Optr.GetTop(x);
    while (Exp[i]!='=' || x!=6)
    {
        if (48<=Exp[i] && Exp[i]<=57)
        {
            TempOpnd[Length++]=Exp[i];
            i++;
        }
        else
            if (Exp[i]=='+' || Exp[i]=='-' || Exp[i]=='*' || Exp[i]=='/' ||
Exp[i]=='(' || Exp[i]==')' || Exp[i]=='=')
            {
                if (Length>0)
                {
                    TempOpnd[Length]='\0';
                    Length=0;
                    Opnd.Push(atoi(TempOpnd));
                }
                switch (Exp[i])
```

```

{
case '+' :
    TempOptr=0;
    break;
case '-' :
    TempOptr=1;
    break;
case '*' :
    TempOptr=2;
    break;
case '/' :
    TempOptr=3;
    break;
case '(' :
    TempOptr=4;
    break;
case ')' :
    TempOptr=5;
    break;
case '=' :
    TempOptr=6;
    break;
}
Optr.GetTop(x);
switch (Operator[x][TempOptr])
{
case 0:
    Optr.Push(TempOptr);
    i++;
    break;
case 1:
    Optr.Pop(x);
    i++;
    break;
case 2:
    Optr.Pop(TempOptr);
    Opnd.Pop(b);
    Opnd.Pop(a);
    switch (TempOptr)
    {
case 0:
        Opnd.Push(a+b);
        break;
case 1:

```

```

        Opnd.Push(a-b);
        break;
    case 2:
        Opnd.Push(a*b);
        break;
    case 3:
        Opnd.Push(a/b);
        break;
    }
    break;
}

Optr.GetTop(x);
}
Opnd.Pop(Ans);
return Ans;
}

```

5 队列

5.1 队列的定义

```

template <typename DataType>
class QueueClass
{
private:
    struct BodyNodeStruct
    {
        DataType Data;
        BodyNodeStruct *Pre,*Next;
    } *Front,*Rear;

public:
    QueueClass(); //构造函数，构造一个空队列
    bool Empty(); //判断队列是否为空，若为空则返回 true，否则返回 false
    int Push(DataType Data); //将 Data 压入队尾，如果压入成功则返回 0，否则返回 1
    int Pop(DataType &Data); //弹出队头元素，保存在 Data 中，若队列不为空则返回
0，否则返回 1
    int GetFront(DataType &Data); //读取队头元素，保存在 Data 中，若队列不为空则
返回 0，否则返回 1
    int Size(); //返回队列中元素个数
    void Clear(); //清空队列

```

```

    ~QueueClass(); //析构函数
} ;

template <typename DataType>
QueueClass<DataType>::QueueClass()
{
    Front=new BodyNodeStruct;
    Front->Pre=NULL;
    Front->Next=NULL;
    Rear=Front;
}

template <typename DataType>
bool QueueClass<DataType>::Empty()
{
    if (Front==Rear) return true;
    else return false;
}

template <typename DataType>
int QueueClass<DataType>::Push(DataType Data)
{
    BodyNodeStruct *Temp=new BodyNodeStruct;

    if (Temp!=NULL)
    {
        Temp->Pre=Rear;
        Temp->Next=NULL;
        Rear->Next=Temp;
        Rear->Data=Data;
        Rear=Temp;
        return 0;
    }
    else return 1;
}

template <typename DataType>
int QueueClass<DataType>::Pop(DataType &Data)
{
    if (!Empty())
    {
        Data=Front->Data;
        Front=Front->Next;
        delete Front->Pre;
    }
}

```

```

        Front->Pre=NULL;
        return 0;
    }
    else return 1;
}

template <typename DataType>
int QueueClass<DataType>::GetFront(DataType &Data)
{
    if (!Empty())
    {
        Data=Front->Data;
        return 0;
    }
    else return 1;
}

template <typename DataType>
int QueueClass<DataType>::Size()
{
    BodyNodeStruct *Temp=Front;
    int Count=0;

    while (Temp!=Rear)
    {
        Count++;
        Temp=Temp->Next;
    }
    return Count;
}

template <typename DataType>
void QueueClass<DataType>::Clear()
{
    DataType Temp;

    while (!Empty()) Pop(Temp);
}

template <typename DataType>
QueueClass<DataType>::~QueueClass()
{
    Clear();
    delete Front;
}

```

```
}
```

6 串

6.1 基本操作

6.1.1 赋值

```
/*
```

函数功能：复制一个串到另一个串中。

函数原形：void Copy(char Str[], char Source[]);

参数：

char Str[]：目标串；

char Source[]：源串。

返回值：无。

备注：需包含<string>。

```
*/
```

```
void Copy(char Str[], char Source[])
```

```
{
```

```
    unsigned i;
```

```
    for (i=0;i<=strlen(Source);i++) Str[i]=Source[i];
```

```
}
```

6.1.2 比较大小（字典顺序）

```
/*
```

函数功能：比较两个串的大小（按字典顺序）。

函数原形：int Compare(char Str1[], char Str2[]);

参数：

char Str1[], Str2[]：欲比较的两个串。

返回值：int 型，如果 Str1>Str2 返回 1，如果 Str1=Str2 返回 0，如果 Str1<Str2 返回-1。

备注：需包含头文件<string>、<ctype.h>。

```
*/
```

```
int Compare(char Str1[], char Str2[])
```

```
{
```

```
    unsigned Len, Len1, Len2, i=0;
```

```
    Len1=(unsigned)strlen(Str1);
```

```
    Len2=(unsigned)strlen(Str2);
```

```

    if (Len1<Len2) Len=Len1;
    else Len=Len2;
    while (toupper(Str1[i])==toupper(Str2[i]) && i<=Len-1) i++;
    if (i<Len)
    {
        if (toupper(Str1[i])>toupper(Str2[i])) return 1;
        else return -1;
    }
    else
        if (Len1==Len2) return 0;
        else
            if (Len1>Len2) return 1;
            else return -1;
}

```

6.2 模式匹配

6.2.1 一般匹配算法

/*

函数功能：返回模式串在目标串中的位置。

函数原形：int Index(char s[], char t[], int Pos);

参数：

char s[]：目标串；

char t[]：模式串；

int Pos：起始检测位置。

返回值：int 型，如果模式串能够匹配，则返回其在目标串中的位置，否则返回-1。

备注：需包含头文件<string>。

*/

```

int Index(char s[], char t[], int Pos)
{
    int i, j;

    i=Pos;
    j=0;
    while (i<=(int)strlen(s)-1 && j<=(int)strlen(t)-1)
        if (s[i]==t[j])
        {
            i++;
            j++;
        }
        else
        {

```

```

        i=i-j+1;
        j=0;
    }
    if (j>(int)strlen(t)-1) return i-strlen(t);
    else return -1;
}

```

6.2.2 KMP 算法

/*

函数功能：返回模式串在目标串中的位置。

函数原形：int IndexKMP(char s[], char t[], int Pos);

参数：

char s[]：目标串；

char t[]：模式串；

int Pos：起始检测位置。

返回值：int 型，如果模式串能够匹配，则返回其在目标串中的位置，否则返回-1。

备注：需包含头文件<string>。

*/

```

void GetNext(char t[], int Next[])
{
    int i, j;

    i=0;
    Next[0]=-1;
    j=-1;
    while (i<(int)strlen(t)-1)
        if (j==-1 || t[i]==t[j])
        {
            i++;
            j++;
            if (t[i]!=t[j]) Next[i]=j;
            else Next[i]=Next[j];
        }
        else j=Next[j];
}

```

```

int IndexKMP(char s[], char t[], int Pos)
{
    int i, j, *Next=new int[strlen(t)];

    GetNext(t, Next);
    i=Pos;
    j=0;
}

```

```

while (i<=(int)strlen(s)-1 && j<=(int)strlen(t)-1)
    if (j==-1 || s[i]==t[j])
    {
        i++;
        j++;
    }
    else j=Next[j];
delete[] Next;
if (j>(int)strlen(t)-1) return i-strlen(t);
else return -1;
}

```

7 树

7.1 二叉树的定义

```

template <typename DataType>
class BinaryTreeNode
{
private:
    DataType Data;
    BinaryTreeNode<DataType> *Prt, *Lch, *Rch;

public:
    BinaryTreeNode(); //构造函数，创建一个二叉树根结点
    BinaryTreeNode(DataType x); //构造函数，创建一个值为 x 的二叉树根结点
    BinaryTreeNode(BinaryTreeNode<DataType> *Parent); //构造函数，创建一个父
        亲结点为 Parent 的二叉树叶子结点
    BinaryTreeNode(DataType x, BinaryTreeNode<DataType> *Parent); //构造函数，
        创建一个父亲为 Parent 值为 x 的二叉树叶子结点
    void Assign(DataType x); //将结点赋值为 x
    DataType Value(); //返回结点的值
    BinaryTreeNode<DataType> *Root(); //返回本结点所在二叉树的根结点
    BinaryTreeNode<DataType> *Parent(); //返回本结点的父亲结点
    BinaryTreeNode<DataType> *LeftChild(); //返回本结点的左儿子结点
    BinaryTreeNode<DataType> *RightChild(); //返回本结点的右儿子结点
    void PreOrder(int (*Visit)(BinaryTreeNode<DataType> *)); //对以本结点为根
        的二叉树，函数 int Visit(BinaryTreeNode<DataType> *) 为访问方式进行先序遍历
    void InOrder(int (*Visit)(BinaryTreeNode<DataType> *)); //对以本结点为根
        的二叉树，函数 int Visit(BinaryTreeNode<DataType> *) 为访问方式进行中序遍历
    void PostOrder(int (*Visit)(BinaryTreeNode<DataType> *)); //对以本结点为根
        的二叉树，函数 int Visit(BinaryTreeNode<DataType> *) 为访问方式进行后序遍历

```

```

        int InsertChild(int Child, BinaryTreeClass<DataType> *SubTree); //将二叉树
SubTree 作为本结点的某一子树插入本结点所在二叉树
        int DeleteChild(int Child); //删除本结点的某一子树
        ~BinaryTreeClass(); //析构函数
    } ;

```

```

template <typename DataType>
BinaryTreeClass<DataType>::BinaryTreeClass():Prt(NULL), Lch(NULL), Rch(NULL) {}

```

```

template <typename DataType>
BinaryTreeClass<DataType>::BinaryTreeClass(DataType
x):Data(x), Prt(NULL), Lch(NULL), Rch(NULL) {}

```

```

template <typename DataType>
BinaryTreeClass<DataType>::BinaryTreeClass(BinaryTreeClass<DataType>
*Parent):Prt(Parent), Lch(NULL), Rch(NULL) {}

```

```

template <typename DataType>
BinaryTreeClass<DataType>::BinaryTreeClass(DataType x, BinaryTreeClass<DataType>
*Parent):Data(x), Prt(Parent), Lch(NULL), Rch(NULL) {}

```

```

template <typename DataType>
void BinaryTreeClass<DataType>::Assign(DataType x)
{
    Data=x;
}

```

```

template <typename DataType>
DataType BinaryTreeClass<DataType>::Value()
{
    return Data;
}

```

```

template <typename DataType>
BinaryTreeClass<DataType> *BinaryTreeClass<DataType>::Root()
{
    BinaryTreeClass<DataType> *Temp=this;

    while (Temp->Prt!=NULL) Temp=Temp->Prt;
    return Temp;
}

```

```

template <typename DataType>
BinaryTreeClass<DataType> *BinaryTreeClass<DataType>::Parent()

```

```

{
    return Prt;
}

template <typename DataType>
BinaryTreeClass<DataType> *BinaryTreeClass<DataType>::LeftChild()
{
    return Lch;
}

template <typename DataType>
BinaryTreeClass<DataType> *BinaryTreeClass<DataType>::RightChild()
{
    return Rch;
}

template <typename DataType>
void BinaryTreeClass<DataType>::PreOrder(int (*Visit) (BinaryTreeClass<DataType>
*))
{
    Visit(this);
    if (Lch!=NULL) Lch->PreOrder(Visit);
    if (Rch!=NULL) Rch->PreOrder(Visit);
}

template <typename DataType>
void BinaryTreeClass<DataType>::InOrder(int (*Visit) (BinaryTreeClass<DataType>
*))
{
    if (Lch!=NULL) Lch->InOrder(Visit);
    Visit(this);
    if (Rch!=NULL) Rch->InOrder(Visit);
}

template <typename DataType>
void BinaryTreeClass<DataType>::PostOrder(int (*Visit) (BinaryTreeClass<DataType>
*))
{
    if (Lch!=NULL) Lch->PostOrder(Visit);
    if (Rch!=NULL) Rch->PostOrder(Visit);
    Visit(this);
}

template <typename DataType>

```

```

int BinaryTreeClass<DataType>::InsertChild(int Child, BinaryTreeClass<DataType>
*SubTree)
{
    if (SubTree==NULL) return 1;
    else
        if (Child==0)
        {
            if (Lch==NULL)
            {
                Lch=SubTree;
                SubTree->Prt=this;
                return 0;
            }
            else return 2;
        }
        else
            if (Rch==NULL)
            {
                Rch=SubTree;
                SubTree->Prt=this;
                return 0;
            }
            else return 2;
}

```

```

template <typename DataType>
int BinaryTreeClass<DataType>::DeleteChild(int Child)
{
    if (Child==0)
    {
        if (Lch!=NULL)
        {
            delete Lch;
            Lch=NULL;
            return 0;
        }
        else return 1;
    }
    else
        if (Rch!=NULL)
        {
            delete Rch;
            Rch=NULL;
            return 0;
        }
}

```

```

    }
    else return 1;
}

template <typename DataType>
BinaryTreeClass<DataType>::~BinaryTreeClass()
{
    DeleteChild(0);
    DeleteChild(1);
}

```

7.2 根据先序序列和中序序列建立二叉树

/*

函数功能：根据先序序列和中序序列建立二叉树。

函数原形：BinaryTreeClass<char> *Create(string PreOrder, string InOrder);

参数：

string PreOrder：先序序列；

string InOrder：中序序列。

返回值：BinaryTreeClass<char> *型，二叉树根结点地址。

备注：需包含头文件<string>。

*/

```

BinaryTreeClass<char> *Create(string PreOrder, string InOrder)
{
    BinaryTreeClass<char> *Root=new BinaryTreeClass<char>(PreOrder[0]);
    string LTI,RTI,LTP,RTP;
    int x;

    if (Root!=NULL)
    {
        x=(int)InOrder.find(Root->Value());
        LTI.assign(InOrder,0,x);
        RTI.assign(InOrder,x+1,InOrder.length()-x-1);
        LTP.assign(PreOrder,1,x);
        RTP.assign(PreOrder,x+1,RTI.length());
        if (LTI.length()>0) Root->InsertChild(0,Create(LTP,LTI));
        if (RTI.length()>0) Root->InsertChild(1,Create(RTP,RTI));
    }
    return Root;
}

```

7.3 二叉排序树

/*

函数功能：向二叉排序树中插入元素。

函数原形：BinaryTreeClass<DataType> *Insert(BinaryTreeClass<DataType> *Root, DataType x);

参数：

BinaryTreeClass<DataType> *Root：目标二叉排序树；

DataType x：欲插入二叉排序树*Root 的元素。

返回值：BinaryTreeClass<DataType> *型，插入新元素后的二叉排序树根结点地址。

备注：无。

*/

```
template <typename DataType>
BinaryTreeClass<DataType> *Insert(BinaryTreeClass<DataType> *Root, DataType x)
{
    BinaryTreeClass<DataType> *Point=Root,*Pre=NULL;

    while (Point!=NULL)
    {
        Pre=Point;
        if (x<=Point->Value()) Point=Point->LeftChild();
        else Point=Point->RightChild();
    }
    Point=new BinaryTreeClass<DataType>(x, Pre);
    if (Point->Parent() !=NULL)
        if (x<=Point->Parent()->Value()) Point->Parent()->InsertChild(0, Point);
        else Point->Parent()->InsertChild(1, Point);
    return Point->Root();
}
```

7.4 建立哈夫曼树

/*

函数功能：根据指定权值建立一棵哈夫曼树。

函数原形：BinaryTreeClass<DataType> *Huffman(DataType Weight[], int n);

参数：

DataType Weight[]：权值；

int n：权值个数。

返回值：BinaryTreeClass<DataType> *型，生成的哈夫曼树根结点地址。

备注：无。

*/

```
template <typename DataType>
void SortData(BinaryTreeClass<DataType> *a[], int l, int r)
```

```

{
    int i, j;
    DataType Mid;
    BinaryTreeClass<DataType> *k;

    i=1;
    j=r;
    Mid=a[(1+r)/2]->Value();
    do
    {
        while (a[i]->Value()<Mid) ++i;
        while (Mid<a[j]->Value()) --j;
        if (i<=j)
        {
            k=a[i];
            a[i]=a[j];
            a[j]=k;
            ++i;
            --j;
        }
    } while (i<=j);
    if (i<r) SortData(a, i, r);
    if (l<j) SortData(a, l, j);
}

template <typename DataType>
BinaryTreeClass<DataType> *Huffman(DataType Weight[], int n)
{
    BinaryTreeClass<DataType> **Tree=new BinaryTreeClass<DataType> *[n], *Root;
    int i;

    for (i=0; i<=n-1; i++) Tree[i]=new BinaryTreeClass<DataType>(Weight[i]);
    while (n>1)
    {
        SortData(Tree, 0, n-1); //按 Data 域升序排序
        Root=new BinaryTreeClass<DataType>(Tree[0]->Value()+Tree[1]->Value());
        Root->InsertChild(0, Tree[0]);
        Root->InsertChild(1, Tree[1]);
        Tree[0]=Root;
        Tree[1]=Tree[n-1];
        n--;
    }
    return Root;
}

```

7.5 树的定义

/*注：在调用成员函数 void BFS(int (*Visit)(TreeClass<DataType> *))前需要定义 QueueClass<DataType>类*/

```
template <typename DataType>
class TreeClass
{
private:
    DataType Data;
    int Dgr;
    TreeClass *Prt, **Chd;

public:
    TreeClass(int Degree); //构造函数，创建树的一个度为 Degree 的根结点
    TreeClass(int Degree, DataType x); //构造函数，创建树的一个度为 Degree 值为 x 的根结点
    TreeClass(int Degree, TreeClass<DataType> *Parent); //构造函数，创建一个度为 Degree 父亲结点为 Parent 的叶子结点
    TreeClass(int Degree, DataType x, TreeClass<DataType> *Parent); //构造函数，创建一个度为 Degree 值为 x 父亲为 Parent 值为 x 的叶子结点
    void Assign(DataType x); //将结点赋值为 x
    DataType Value(); //返回结点的值
    int Degree(); //返回结点的度
    int ChildAmount(); //返回结点的孩子个数
    TreeClass<DataType> *Root(); //返回本结点所在树的根结点
    TreeClass<DataType> *Parent(); //返回本结点的父亲结点
    TreeClass<DataType> *Child(int Child); //返回本结点的某个儿子结点
    void DFS(int (*Visit)(TreeClass<DataType> *)); //对以本结点为根的树，函数 int Visit(BinaryTreeClass<DataType> *)为访问方式进行深度优先遍历
    void BFS(int (*Visit)(TreeClass<DataType> *)); //对以本结点为根的树，函数 int Visit(BinaryTreeClass<DataType> *)为访问方式进行宽度优先遍历
    int InsertChild(TreeClass<DataType> *SubTree); //将树 SubTree 作为本结点的某一子树插入本结点所在树
    int DeleteChild(int Child); //删除本结点的某一子树
    ~TreeClass(); //析构函数
};
```

```
template <typename DataType>
TreeClass<DataType>::TreeClass(int Degree)
{
    int i;

    Dgr=Degree;
```

```

    Prt=NULL;
    Chd=new TreeClass *[Dgr];
    for (i=0;i<=Dgr-1;i++) Chd[i]=NULL;
}

template <typename DataType>
TreeClass<DataType>::TreeClass(int Degree, DataType x)
{
    int i;

    Dgr=Degree;
    Data=x;
    Prt=NULL;
    Chd=new TreeClass *[Dgr];
    for (i=0;i<=Dgr-1;i++) Chd[i]=NULL;
}

template <typename DataType>
TreeClass<DataType>::TreeClass(int Degree, TreeClass<DataType> *Parent)
{
    int i;

    Dgr=Degree;
    Prt=Parent;
    Chd=new TreeClass *[Dgr];
    for (i=0;i<=Dgr-1;i++) Chd[i]=NULL;
}

template <typename DataType>
TreeClass<DataType>::TreeClass(int Degree, DataType x, TreeClass<DataType>
*Parent)
{
    int i;

    Dgr=Degree;
    Data=x;
    Prt=Parent;
    Chd=new TreeClass *[Dgr];
    for (i=0;i<=Dgr-1;i++) Chd[i]=NULL;
}

template <typename DataType>
void TreeClass<DataType>::Assign(DataType x)
{

```

```

        Data=x;
    }

    template <typename DataType>
    DataType TreeClass<DataType>::Value()
    {
        return Data;
    }

    template <typename DataType>
    int TreeClass<DataType>::Degree()
    {
        return Dgr;
    }

    template <typename DataType>
    int TreeClass<DataType>::ChildAmount()
    {
        int i=0;

        for (i=0;i<=Dgr-1;i++)
            if (Chd[i]==NULL) return i;
        return i;
    }

    template <typename DataType>
    TreeClass<DataType> *TreeClass<DataType>::Root()
    {
        TreeClass<DataType> *Temp=this;

        while (Temp->Prt!=NULL) Temp=Temp->Prt;
        return Temp;
    }

    template <typename DataType>
    TreeClass<DataType> *TreeClass<DataType>::Parent()
    {
        return Prt;
    }

    template <typename DataType>
    TreeClass<DataType> *TreeClass<DataType>::Child(int Child)
    {
        if (Child>=Dgr) return NULL;

```

```

        else return Chd[Child];
    }

template <typename DataType>
void TreeClass<DataType>::DFS(int (*Visit)(TreeClass<DataType> *))
{
    int i;

    Visit(this);
    for (i=0;i<=Dgr-1;i++)
        if (Chd[i]!=NULL) Chd[i]->DFS(Visit);
        else break;
}

template <typename DataType>
void TreeClass<DataType>::BFS(int (*Visit)(TreeClass<DataType> *))
{
    TreeClass<DataType> *Temp=this;
    QueueClass<TreeClass<DataType> *> Queue;
    int i;

    Queue.Push(Temp);
    while (!Queue.Empty())
    {
        Queue.Pop(Temp);
        Visit(Temp);
        for (i=0;i<=Dgr-1;i++)
            if (Temp->Chd[i]!=NULL) Queue.Push(Temp->Chd[i]);
            else break;
    }
}

template <typename DataType>
int TreeClass<DataType>::InsertChild(TreeClass<DataType> *SubTree)
{
    int i;

    if (SubTree!=NULL)
    {
        for (i=0;i<=Dgr-1;i++)
            if (Chd[i]==NULL)
            {
                Chd[i]=SubTree;
                SubTree->Prt=this;
            }
    }
}

```

```

        return 0;
    }
    return 2;
}
else return 1;
}

template <typename DataType>
int TreeClass<DataType>::DeleteChild(int Child)
{
    if (Chd[Child]!=NULL)
    {
        delete Chd[Child];
        Chd[Child]=NULL;
        return 0;
    }
    else return 1;
}

template <typename DataType>
TreeClass<DataType>::~~TreeClass()
{
    int i;

    for (i=0;i<=Dgr-1;i++)
        if (Chd[i]!=NULL) DeleteChild(i);
        else break;
}

```

8 图

8.1 图的定义

/*注：在调用成员函数 int BFS(int (*Visit) (GraphClass<VexType, ArcType> &, int), int Start)前需要定义 QueueClass<DataType>类*/

```

template <typename VexType, typename ArcType>
class GraphClass
{
private:
    int Size;
    VexType *Vex;
    ArcType **Arc;

```

```
void DfsTravel(int (*Visit) (GraphClass<VexType, ArcType> &, int), int Now, bool Visited[]);
```

```
public:
```

```
void Init(); //初始化图顶点和边
GraphClass(int x); //构造函数，创建一个定点数为 x 的图
int UpdateVex(int Num, VexType Data); //更改顶点 Num 的数据为 Data
int UpdateArc(int x, int y, ArcType Data, bool Untl=false); //更改边<x,y>
的数据为 Data, Untl=true 时表示<x,y>这条边是单向边
int GetSize(); //返回图的顶点数
VexType GetVex(int Num); //获取顶点 Num 的数据
ArcType GetArc(int x, int y); //获取边<x,y>的数据
int InDegree(int Num); //求顶点 Num 的入度
int OutDegree(int Num); //求顶点 Num 的出度
int DFS(int (*Visit) (GraphClass<VexType, ArcType> &, int), int Start); //以
Start 为起点, Visit 为访问方式, 深度优先遍历图
int BFS(int (*Visit) (GraphClass<VexType, ArcType> &, int), int Start); //以
Start 为起点, Visit 为访问方式, 宽度优先遍历图
~GraphClass(); //析构函数
} ;
```

```
template <typename VexType, typename ArcType>
void GraphClass<VexType, ArcType>::Init()
{
    int i, j;

    for (i=0; i<=Size-1; i++) Vex[i]=0;
    for (i=0; i<=Size-1; i++)
        for (j=0; j<=Size-1; j++) Arc[i][j]=0;
}
```

```
template <typename VexType, typename ArcType>
GraphClass<VexType, ArcType>::GraphClass(int x)
{
    int i;

    Size=x;
    Vex=new VexType[Size];
    Arc=new ArcType *[Size];
    for (i=0; i<=Size-1; i++) Arc[i]=new ArcType[Size];
    Init();
}
```

```

template <typename VexType, typename ArcType>
int GraphClass<VexType, ArcType>::UpdateVex(int Num, VexType Data)
{
    if (Num<Size)
    {
        Vex[Num]=Data;
        return 0;
    }
    else return 1;
}

template <typename VexType, typename ArcType>
int GraphClass<VexType, ArcType>::UpdateArc(int x, int y, ArcType Data, bool
Untl=false)
{
    if (x<Size && y<Size)
    {
        Arc[x][y]=Data;
        if (Untl) Arc[y][x]=0;
        else Arc[y][x]=Data;
        return 0;
    }
    return 1;
}

template <typename VexType, typename ArcType>
int GraphClass<VexType, ArcType>::GetSize()
{
    return Size;
}

template <typename VexType, typename ArcType>
VexType GraphClass<VexType, ArcType>::GetVex(int Num)
{
    if (Num<Size) return Vex[Num];
    else return 0;
}

template <typename VexType, typename ArcType>
ArcType GraphClass<VexType, ArcType>::GetArc(int x, int y)
{
    if (x<Size && y<Size) return Arc[x][y];
    else return 0;
}

```

```

template <typename VexType, typename ArcType>
int GraphClass<VexType, ArcType>::InDegree(int Num)
{
    int Ans=0, i;

    for (i=0; i<=Size-1; i++)
        if (Arc[i][Num]>0) Ans++;
    return Ans;
}

template <typename VexType, typename ArcType>
int GraphClass<VexType, ArcType>::OutDegree(int Num)
{
    int Ans=0, i;

    for (i=0; i<=Size-1; i++)
        if (Arc[Num][i]>0) Ans++;
    return Ans;
}

template <typename VexType, typename ArcType>
void GraphClass<VexType, ArcType>::DfsTravel(int
(*Visit) (GraphClass<VexType, ArcType> &, int), int Now, bool Visited[])
{
    int i;

    Visit(*this, Now);
    Visited[Now]=true;
    for (i=0; i<=Size-1; i++)
        if (Arc[Now][i]>0 && !Visited[i]) DfsTravel(Visit, i, Visited);
}

template <typename VexType, typename ArcType>
int GraphClass<VexType, ArcType>::DFS(int (*Visit) (GraphClass<VexType, ArcType>
&, int), int Start)
{
    int i;
    bool *Visited=new bool[Size];

    if (Start<Size)
    {
        for (i=0; i<=Size-1; i++) Visited[i]=false;
        DfsTravel(Visit, Start, Visited);
    }
}

```

```

        return 0;
    }
    else return 1;
}

template <typename VexType, typename ArcType>
int GraphClass<VexType, ArcType>::BFS(int (*Visit) (GraphClass<VexType, ArcType>
&, int), int Start)
{
    QueueClass<int> Queue;
    int i, Temp;
    bool *Visited=new bool[Size];

    if (Start<Size)
    {
        for (i=0;i<=Size-1;i++) Visited[i]=false;
        Queue.Push(Start);
        Visited[Start]=true;
        while (!Queue.Empty())
        {
            Queue.Pop(Temp);
            Visit(*this, Temp);
            for (i=0;i<=Size-1;i++)
                if (Arc[Temp][i]>0 && !Visited[i])
                {
                    Queue.Push(i);
                    Visited[i]=true;
                }
        }
        return 0;
    }
    else return 1;
}

template <typename VexType, typename ArcType>
GraphClass<VexType, ArcType>::~~GraphClass()
{
    int i;

    delete[] Vex;
    for (i=0;i<=Size-1;i++) delete[] Arc[i];
    delete[] Arc;
}

```

8.2 图的最短路径

8.2.1 Dijkstra 算法

```
/*
函数功能：计算图的单源最短路径。
函数原形：void Dijkstra(GraphClass<VexType, ArcType> &Graph, int Start);
参数：
GraphClass<VexType, ArcType> &Graph：目标图；
int Start：起点下标。
返回值：ArcType *型，结果数组首地址。
备注：无。
*/
template <typename VexType, typename ArcType>
ArcType *Dijkstra(GraphClass<VexType, ArcType> &Graph, int Start)
{
    ArcType *Distance=new ArcType[Graph.GetSize()], Min, Infinite=ArcType(unsigned(~0)/2);
    int i, j, m;
    bool *Used=new bool[Graph.GetSize()];

    for (i=0; i<=Graph.GetSize()-1; i++)
    {
        Used[i]=false;
        if (Graph.GetArc(Start, i)>0) Distance[i]=Graph.GetArc(Start, i);
        else Distance[i]=Infinite;
    }
    Distance[Start]=0;
    Used[Start]=1;
    for (i=1; i<=Graph.GetSize()-1; i++)
    {
        Min=Infinite;
        m=Start;
        for (j=0; j<=Graph.GetSize()-1; j++)
            if (!Used[j] && Distance[j]<Min)
            {
                m=j;
                Min=Distance[j];
            }
        Used[m]=true;
        for (j=0; j<=Graph.GetSize()-1; j++)
            if (Graph.GetArc(m, j)>0 && !Used[j] &&
                Min+Graph.GetArc(m, j)<Distance[j]) Distance[j]=Min+Graph.GetArc(m, j);
    }
}
```

```

    }
    return Distance;
}

```

8.2.2 Floyd 算法

```

/*
函数功能：计算图的单源最短路径。
函数原形：void Floyd(GraphClass<VexType, ArcType> &Graph);
参数：
GraphClass<VexType, ArcType> &Graph: 目标图。
返回值：ArcType **型，结果数组首地址。
备注：无。
*/
template <typename VexType, typename ArcType>
ArcType **Floyd(GraphClass<VexType, ArcType> &Graph)
{
    ArcType **Distance, Infinite=ArcType(unsigned(~0)/2);
    int i, j, k;

    Distance=new ArcType *[Graph.GetSize()];
    for (i=0; i<=Graph.GetSize()-1; i++) Distance[i]=new ArcType[Graph.GetSize()];
    for (i=0; i<=Graph.GetSize()-1; i++)
        for (j=0; j<=Graph.GetSize()-1; j++)
            if (Graph.GetArc(i, j)>0) Distance[i][j]=Graph.GetArc(i, j);
            else Distance[i][j]=Infinite;
    for (k=0; k<=Graph.GetSize()-1; k++)
        for (i=0; i<=Graph.GetSize()-1; i++)
            for (j=0; j<=Graph.GetSize()-1; j++)
                if (i!=j && i!=k && k!=j)
                    if (Distance[i][k]+Distance[k][j]<Distance[i][j])
                        Distance[i][j]=Distance[i][k]+Distance[k][j];
    return Distance;
}

```

8.3 最小生成树

8.3.1 prim 算法

```

/*
函数功能：求图的最小生成树。
函数原形：GraphClass<VexType, ArcType> *Prim(GraphClass<VexType, ArcType> &Graph);

```

参数:

GraphClass<VexType, ArcType> &Graph: 目标图。

返回值: GraphClass<VexType, ArcType> *型, 最小生成树地址。

备注: 无。

```
*/
template <typename VexType, typename ArcType>
GraphClass<VexType, ArcType> *Prim(GraphClass<VexType, ArcType> &Graph)
{
    ArcType Min, Infinite=ArcType(unsigned(~0)/2);
    GraphClass<VexType, ArcType> *Mcst=new
GraphClass<VexType, ArcType>(Graph.GetSize());
    int i, j, k, x, y;
    bool *Vertex=new bool[Graph.GetSize()];

    for (i=0; i<=Graph.GetSize()-1; i++)
    {
        Vertex[i]=false;
        Mcst->UpdateVex(i, Graph.GetVex(i));
    }
    Vertex[0]=true;
    for (k=1; k<=Graph.GetSize()-1; k++)
    {
        Min=Infinite;
        for (i=0; i<=Graph.GetSize()-1; i++)
            for (j=0; j<=Graph.GetSize()-1; j++)
                if (Graph.GetArc(i, j)>0 && Vertex[i] && !Vertex[j] &&
Graph.GetArc(i, j)<Min)
                {
                    Min=Graph.GetArc(i, j);
                    x=i;
                    y=j;
                }
        Vertex[y]=true;
        Mcst->UpdateArc(x, y, Graph.GetArc(x, y));
    }
    return Mcst;
}
```

8.3.2 Kruskal 算法

/*

函数功能: 求图的最小生成树。

函 数 原 形 : GraphClass<VexType, ArcType> *Kruskal (GraphClass<VexType, ArcType>
&Graph);

参数:

GraphClass<VexType, ArcType> &Graph: 目标图。

返回值: GraphClass<VexType, ArcType> *型, 最小生成树地址。

备注: 无。

```
*/
template <typename VexType, typename ArcType>
GraphClass<VexType, ArcType> *Kruskal (GraphClass<VexType, ArcType> &Graph)
{
    ArcType Min, Infinite=ArcType(unsigned(~0)/2);
    GraphClass<VexType, ArcType> *Mcst=new
GraphClass<VexType, ArcType>(Graph.GetSize());
    int i, j, k, x, y, a, b, *VexSet=new int[Graph.GetSize()];

    for (i=0; i<=Graph.GetSize()-1; i++)
    {
        VexSet[i]=i;
        Mcst->UpdateVex(i, Graph.GetVex(i));
    }
    for (k=1; k<=Graph.GetSize()-1; k++)
    {
        Min=Infinite;
        for (i=0; i<=Graph.GetSize()-1; i++)
            for (j=0; j<=Graph.GetSize()-1; j++)
                if (Graph.GetArc(i, j)>0 && VexSet[i]!=VexSet[j] &&
Mcst->GetArc(i, j)==0 && Graph.GetArc(i, j)<Min)
                {
                    Min=Graph.GetArc(i, j);
                    x=i;
                    y=j;
                    a=VexSet[i];
                    b=VexSet[j];
                }
        for(i=0; i<=Graph.GetSize(); i++)
            if(VexSet[i]==b) VexSet[i]=a;
        Mcst->UpdateArc(x, y, Graph.GetArc(x, y));
    }
    return Mcst;
}
```

8.4 拓扑排序

/*

函数功能: 求有向图的拓扑序列。

函数原形: int TopSort (GraphClass<VexType, ArcType> &Graph, int Sequence[]);

参数:

GraphClass<VexType, ArcType> &Graph: 目标图;

int Sequence[]: 拓扑序列。

返回值: int 型, 返回 1 表示有向图中有环路。

备注: 需定义 StackClass 类。

*/

```
template <typename VexType, typename ArcType>
int TopSort(GraphClass<VexType, ArcType> &Graph, int Sequence[])
{
    StackClass<int> Stack;
    int i, x, Count, *InDegree=new int[Graph.GetSize()];

    for (i=0; i<=Graph.GetSize()-1; i++) InDegree[i]=Graph.InDegree(i);
    for (i=0; i<=Graph.GetSize()-1; i++)
        if (InDegree[i]==0) Stack.Push(i);
    Count=0;
    while (!Stack.Empty())
    {
        Stack.Pop(x);
        Sequence[Count++]=x;
        for (i=0; i<=Graph.GetSize()-1; i++)
            if (Graph.GetArc(x, i)>0)
                if (--InDegree[i]==0) Stack.Push(i);
    }
    if (Count==Graph.GetSize()) return 0;
    else return 1;
}
```

8.5 关键路径

/*

函数功能: 求有向图的关键路径。

函数原形: GraphClass<VexType, ArcType> *CriticalPath(GraphClass<VexType, ArcType> &Graph);

参数:

GraphClass<VexType, ArcType> &Graph: 目标图。

返回值: GraphClass<VexType, ArcType> *型, 关键路径地址, 如果图中有环路, 返回 NULL。

备注: 需定义 TopSort, 和 StackClass 类。

*/

```
template <typename VexType, typename ArcType>
GraphClass<VexType, ArcType> *CriticalPath(GraphClass<VexType, ArcType> &Graph)
{
    ArcType Min, Max, *VexEarlest=new ArcType[Graph.GetSize()], *VexLatest=new
    ArcType[Graph.GetSize()], Infinite=ArcType(unsigned(~0)/2);
```

```

    GraphClass<VexType, ArcType>                                *Path=new
GraphClass<VexType, ArcType>(Graph.GetSize());
    int i, j, *Sequence=new int[Graph.GetSize()];

    if (TopSort(Graph, Sequence)==0)
    {
        for (i=0; i<=Graph.GetSize()-1; i++) Path->UpdateVex(i, Graph.GetVex(i));
        VexEarlest[0]=0;
        for (i=1; i<=Graph.GetSize()-1; i++)
        {
            Max=0;
            for (j=0; j<=Graph.GetSize()-1; j++)
                if (Graph.GetArc(j, Sequence[i])>0 &&
VexEarlest[j]+Graph.GetArc(j, Sequence[i])>Max)
Max=VexEarlest[j]+Graph.GetArc(j, Sequence[i]);
            VexEarlest[Sequence[i]]=Max;
        }
        VexLatest[Graph.GetSize()-1]=VexEarlest[Graph.GetSize()-1];
        for (i=Graph.GetSize()-2; i>=0; i--)
        {
            Min=Infinite;
            for (j=0; j<=Graph.GetSize()-1; j++)
                if (Graph.GetArc(Sequence[i], j)>0 &&
VexLatest[j]-Graph.GetArc(Sequence[i], j)<Min)
Min=VexLatest[j]-Graph.GetArc(Sequence[i], j);
            VexLatest[Sequence[i]]=Min;
        }
        for (i=0; i<=Graph.GetSize()-1; i++)
            for (j=0; j<=Graph.GetSize()-1; j++)
                if (Graph.GetArc(i, j)>0 && VexEarlest[i]==VexLatest[i] &&
VexEarlest[j]==VexLatest[j]) Path->UpdateArc(i, j, Graph.GetArc(i, j), 1);
        return Path;
    }
    else return NULL;
}

```

9 高精度计算

9.1 加法

```

/*
函数功能：高精度加法。

```

函数原形: string Add(string Num1, string Num2);

参数:

string Num1, string Num2: 被加数与加数。

返回值: string 型, 两数之和。

备注: 需包含<string>。

```
*/  
string Add(string Num1, string Num2)  
{  
    string Ans;  
    int *a, *b, i, Max, Len, Len1, Len2, k;  
  
    Len1=(int)Num1.length();  
    Len2=(int)Num2.length();  
    if (Len1>Len2) Len=Len1;  
    else Len=Len2;  
    Max=Len;  
    a=new int[Len+1];  
    for (i=0;i<=Len;i++) a[i]=0;  
    b=new int[Len];  
    for (i=0;i<=Len-1;i++) b[i]=0;  
    k=0;  
    for (i=Len1-1;i>=0;i--) a[k++]=Num1[i]-'0';  
    k=0;  
    for (i=Len2-1;i>=0;i--) b[k++]=Num2[i]-'0';  
    for (i=0;i<=Len-1;i++)  
    {  
        a[i]=a[i]+b[i];  
        if (a[i]>=10)  
        {  
            a[i]=a[i]-10;  
            a[i+1]++;  
        }  
    }  
    while (Len+1>Max || a[Len+1]==0) Len--;  
    for (i=Len+1;i>=0;i--) Ans.append(1, a[i]+'0');  
    return Ans;  
}
```

9.2 减法

/*

函数功能: 高精度减法。

函数原形: string Sub(string Num1, string Num2);

参数:

string Num1, string Num2: 被减数与减数。

返回值: string 型, 两数之差。

备注: 需包含<string>。

*/

```
string Sub(string Num1, string Num2)
{
    string Ans;
    int *a, *b, i, Max, Len, Len1, Len2, k;

    Len1=(int)Num1.length();
    Len2=(int)Num2.length();
    Len=Len1;
    Max=Len;
    a=new int[Len];
    b=new int[Len];
    for (i=0;i<=Len-1;i++)
    {
        a[i]=0;
        b[i]=0;
    }
    k=0;
    for (i=Len1-1;i>=0;i--) a[k++]=Num1[i]-'0';
    k=0;
    for (i=Len2-1;i>=0;i--) b[k++]=Num2[i]-'0';
    for (i=0;i<=Len-1;i++)
        if (a[i]<b[i])
        {
            a[i+1]--;
            a[i]=a[i]+10;
            a[i]=a[i]-b[i];
        }
        else a[i]=a[i]-b[i];
    while (a[Len]==0 && Len>0 || Len>=Max) Len--;
    for (i=Len;i>=0;i--) Ans.append(1, a[i]+'0');
    return Ans;
}
```

9.3 乘法

/*

函数功能: 高精度乘法。

函数原形: string Sub(string Num1, string Num2);

参数:

string Num1, string Num2: 被乘数与乘数。

返回值: string 型, 两数之积。

备注: 需包含<string>。

*/

```
string Mul(string Num1, string Num2)
{
    string Ans;
    int *a, *b, *c, Max, Len1, Len2, Len, k, x, y, z, i, j, w;

    Len1=(int)Num1.length();
    Len2=(int)Num2.length();
    Len=Len1+Len2;
    Max=Len;
    a=new int[Len1];
    for (i=0;i<=Len1-1;i++) a[i]=0;
    b=new int[Len2];
    for (i=0;i<=Len2-1;i++) b[i]=0;
    c=new int[Len];
    for (i=0;i<=Len-1;i++) c[i]=0;
    k=0;
    for (i=Len1-1;i>=0;i--) a[k++]=Num1[i]-'0';
    k=0;
    for (i=Len2-1;i>=0;i--) b[k++]=Num2[i]-'0';
    for (i=0;i<=Len1-1;i++)
        for (j=0;j<=Len2-1;j++)
        {
            x=a[i]*b[j];
            y=x/10;
            z=x%10;
            w=i+j;
            c[w]=c[w]+z;
            c[w+1]=c[w+1]+c[w]/10+y;
            c[w]=c[w]%10;
        }
    while (c[Len]==0 && Len>0 || Len>=Max) Len--;
    for (i=Len;i>=0;i--) Ans.append(1, c[i]+'0');
    return Ans;
}
```