湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课 程 名 称	算法设计与分析
章节、专题	基本数据结构
教学目标及基 本 要 求	掌握线性表、栈、队列和矩阵等数据结构的基本知识。
教 学 重 点	矩阵
教 学 难 点	队列
教学内容与时 间 分 配	（1）线性表（1课时） （2）栈（0.5课时） （3）队列（0.5课时） （4）矩阵（0.5课时） 共计2.5课时。
习　　题	第2.5节（练习题）。

第2章　基本数据结构

要设计一个有效的算法，必须选择或设计适合该问题的数据结构，使得算法采用这种数据结构时能对数据施行有效的运算，因此构造数据结构是改进算法的基本方法之一。

本章简要介绍线性表、栈、队列和矩阵等数据结构的基本知识，树、图、堆和集合等数据结构的基本知识在第4章和第5章介绍。

- 线性表
- 栈
- 队列
- 矩阵

2.1 线性表※

2.1.1 线性表的含义

线性表是在计算机程序中经常用到的一种最简单的数据组织形式，由某个集合中的一些元素组成，通常写成 $(a_0, a_1, \ldots, a_{n-1})$。这里 $n \geq 0$，当 $n = 0$ 时是一个空表。插入和删除元素是线性表的两个实质性运算，本节主要讨论这2种运算。

线性表的存储结构一般使用顺序表示（使用一维数组表示）和链式表示（使用链接表表示）这两种方法。这里只介绍线性表的顺序表示。

2.1.2 线性表的顺序表示

1. 表示方法

用一个一维数组buf[*n*]来表示线性表，其中*n*是允许存入表中元素的最大数目。表中的第0号元素存放在buf[0]中，第1号元素存放在buf[1]中，第*i*号元素存放在buf[*i*]中。这种存储结构的线性表通常称为顺序表。

```
// Vector.h
#pragma once
#include <algorithm> // copy_n, etc.
template <class T>
class Vector
{   T *buf = 0; // 一维数组
    int cap = 0, sz = 0; // 数组容量，实际元素个数
public:
```

2. 初始化与赋值

（1）初始化。

```
Vector() {} // 由成员变量声明中的默认值完成空顺序表的初始化
```

```
Vector(int n): // 大小为n的顺序表
    buf(new T[n]), cap(n), sz(n) {}
```

```
~Vector()
{   delete []buf; // 释放数组
}
```

（2）拷贝构造与赋值。

```
Vector(const Vector &X): Vector(X.sz) // 使用另一个顺序表初始化
{   std::copy_n(X.buf, X.sz, buf); // 复制元素
} // 耗时O(sz)
```

```
Vector &operator=(const Vector &X) // 复制顺序表
{   if(&X == this) return *this;
    if(cap < X.cap) // 需要新建数组
        delete []buf, buf = new T[X.cap];
    cap = X.cap, sz = X.sz; // 新容量和新大小
    std::copy_n(X.buf, X.sz, buf); // 复制元素
    return *this;
} // 耗时O(sz)
```

```
Vector(int n): // 大小为n的顺序表
    buf(new T[n]), cap(n), sz(n) {}
```

3. 实质性操作

元素的访问、插入和删除是顺序表的3个实质性操作。

（1）访问元素。提供使用指针访问和使用下标访问2种方式，其中begin()和end()分别得到首元素地址和结束元素的地址。

```
T &operator[](int i) const
{   return buf[i];
}
```

```
T &back() const
{   return buf[sz - 1];
}
```

```
T *begin() const
{   return buf;
}
```

```
T *end() const
{   return buf + sz;
}
```

（2）插入元素。将元素v插入到位置pos。如图2-1所示。

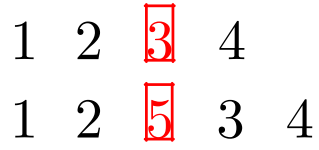


图2-1　插入元素

```
void insert(int pos, const T &v) // 将元素v插入到位置pos
{   if(pos < 0 or pos > sz) return;
    reserve(sz + 1); // 必要时扩大容量(描述见后)
    for(int i = sz; i > pos; --i) // 从pos开始后移元素
        buf[i] = buf[i - 1];
    buf[pos] = v, ++sz; // 插入v
} // 耗时O(sz)
```

```
void push_back(const T &v)
{   insert(sz, v);
}
```

（3）删除元素。删除pos位置的元素。如图2-2所示。

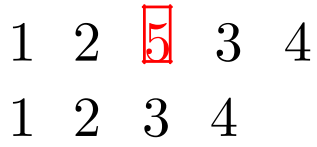


图2-2　删除元素

```
void erase(int pos) // 删除pos位置的元素
{   if(sz <= 0 or pos >= sz) return;
    for(int i = pos + 1; i < sz; ++i) // 从pos+1开始前移元素
        buf[i - 1] = buf[i];
    --sz;
} // 耗时O(sz)
```

```
void pop_back()
{   erase(sz - 1);
}
```

4. 其余常用操作

（1）空表操作。包括空表判断与顺序表清空。

```
bool empty() const
{   return sz <= 0;
}
```

```
void clear()
{   sz = 0;
}
```

（2）大小。

```
int size() const
{   return sz;
}
```

```
void resize(int n) // 更改大小
{   reserve(n); // 必要时扩大容量(描述见后)
    sz = n;
} // 耗时O(sz)
```

（3）容量。

```
    int capacity() const
    {   return cap;
    }
    void reserve(int n) // 扩大容量，新容量不小于n
    {   if(n <= cap) return; // 无需扩容
        n = std::max(2 * cap, n); // 新容量至少加倍，以免扩容太频繁
        T *X = new T[n]; // 新建数组
        std::copy_n(buf, sz, X); // 原有元素复制到新数组
        delete []buf, buf = X, cap = n; // 使用新数组和新容量
    } // 一次耗时O(sz)，均摊耗时O(1)
};
```

（4）比较。两顺序表内容是否相同。方法是首先比较大小，大小相同时对元素逐个比较。

```
template<class T> // 两顺序表内容相同
bool operator==(const Vector<T> &X, const Vector<T> &Y)
{   if(X.size() != Y.size()) return false; // 大小不相等
    for(int i = 0; i < X.size(); ++i) // 逐个比较
        if(X[i] != Y[i]) return false; // 对应元素不相同
    return true;
} // O(|X|)
```

```
template<class T> // 两顺序表内容不相同
bool operator!=(const Vector<T> &X, const Vector<T> &Y)
{   return not(X == Y);
} // O(|X|)
```

2.2 栈※

2.2.1 栈的含义

栈（stack）是一种特殊的线性表。栈的插入和删除都只能在称为栈顶（top）的一端进行。栈的运算意味着如果依次将元素1, 2, 3, 4插入栈，那么从栈中移出的第一个元素必定是4，即最后进入栈的元素将首先移出，因此栈又称为后进先出（LIFO）表。如图2-3所示。

图2-3 栈的例子

2.2.2 栈的顺序表示

1. 表示方法

用一个顺序表buf表示栈，栈中的第0号元素存放在buf[0]，第1号元素存放在buf[1]，第i号元素存放在buf[i]。

```
// Stack.h
#pragma once
#include "Vector.h"
template<class T>
class Stack
{   Vector<T> buf; // 一维数组
public:
```

其中，构造函数和析构函数都使用默认的，无需显式给出。

2. 实质性操作

元素的访问、插入和删除是栈的3个实质性操作。

（1）访问元素。只能直接访问栈顶元素（数组尾部）。

```
const T &top() const // 只读，栈内元素不允许修改
{   return buf.back(); // 尾部元素
}
```

（2）插入元素。只能向栈顶一端（数组尾部）插入元素。如图2-4所示。

1 2 3 4

1 2 3 4 5

图2-4 入栈

```
void push(const T &v)
{   buf.push_back(v); // 向尾部添加一个元素
}
```

（3）删除元素。只能从栈顶一端（数组尾部）删除元素。如图2-5所示。

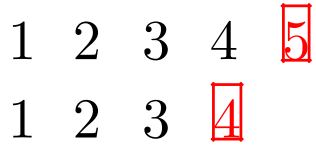


图2-5　出栈

```
void pop()
{   buf.pop_back(); // 从尾部删除一个元素
}
```

3. 其余常用操作

只介绍空栈判断。

```
bool empty() const
{   return buf.empty();
}
};
```

2.3 队列※

2.3.1 队列的含义

队列（queue）也是一种特殊的线性表。队列的插入只能在称为尾部的一端进行，删除只能在称为前部或首部的另一端进行。队列的运算要求第一个插入队中的元素也第一个被移出，因此队列是一个先进先出（FIFO）表。如图2-6所示。

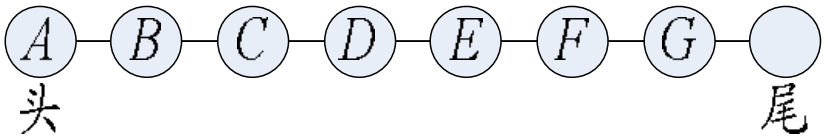


图2-6 队列

2.3.2 队列的顺序表示

1. 表示方法

用一个一维数组buf[n]来表示队列，其中n是队列容量，表示允许存入队列中的元素的最大数目。此时，可以把队列当成一个环形队列来看待。first表示队首元素的位置，last表示队尾元素的下一个位置（插入位置）。当队尾元素位于n-1时，则last=0。最开始时first=last=0。图2-7描述了一个n=8的数组中含有元素A, B, C, D的环形队列的两种可能存放方式。

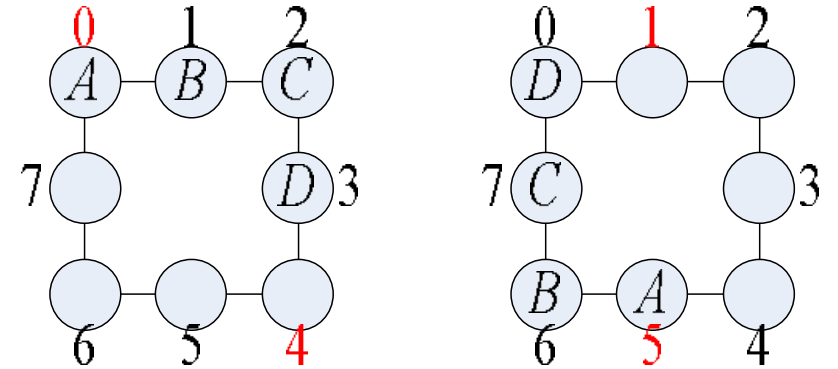


图2-7 两个4元素队列

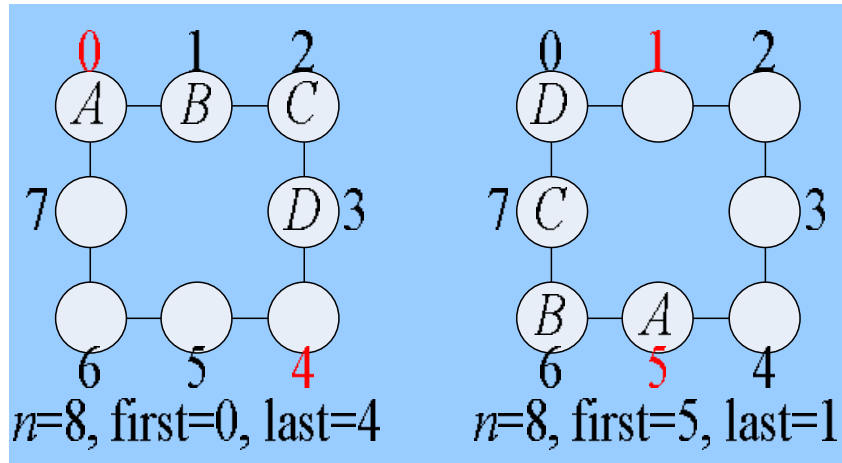
用上述方法表示队列不好区分队列是满的还是空的，因为在这两种情况下都有first = last。为了方便区分队列是满的还是空的，可用变量first和sz代替变量first和last，其中，sz表示队列的实际大小。此时last = (first + sz) mod *n*，且sz <= 0表示队列是空的，sz >= *n*表示队列是满的。

```
// Queue.h
#pragma once
#include "Vector.h"
```

```
template<class T>
class Queue
{   Vector<T> buf = Vector<T>(8); // 缓冲区，至少可容纳8个元素
    int first = 0, sz = 0; // 队首位置，实际元素数
```

```
public:
```

其中，构造函数和析构函数都使用默认的，无需显式给出。



2. 实质性操作

元素的访问、插入和删除是栈的3个实质性操作。

（1）访问元素。只能直接访问队首元素。如图2-8所示。

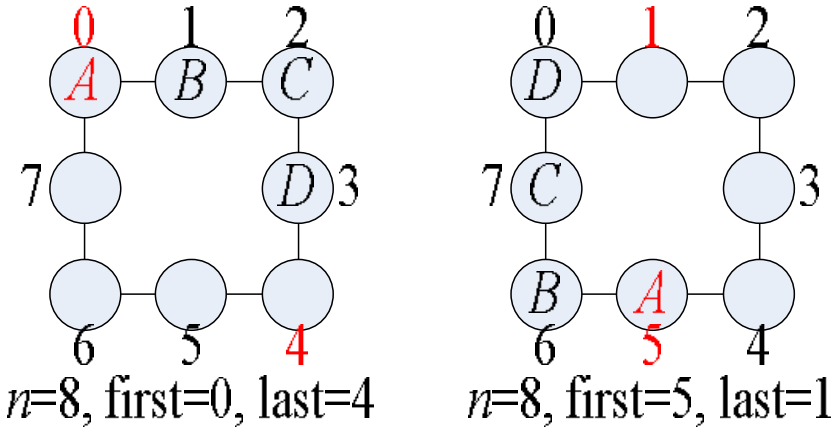


图2-8　访问队列元素

```
const T &front() const
{   return buf[first];
}
```

（2）插入元素。只能向队尾一端插入元素。如果队列已满，则需要先扩容，并将被循环到缓冲区首部的元素（在[0, first)中）移到新增空间。如图2-9所示。

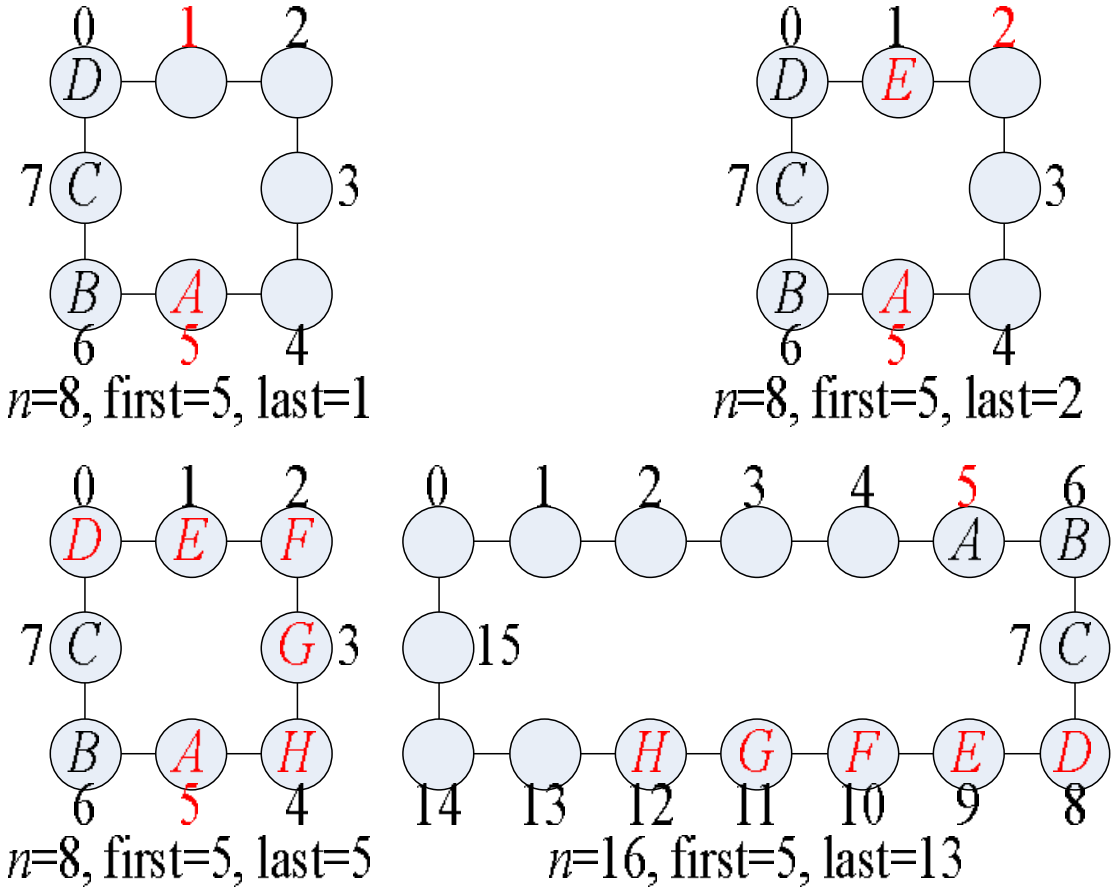
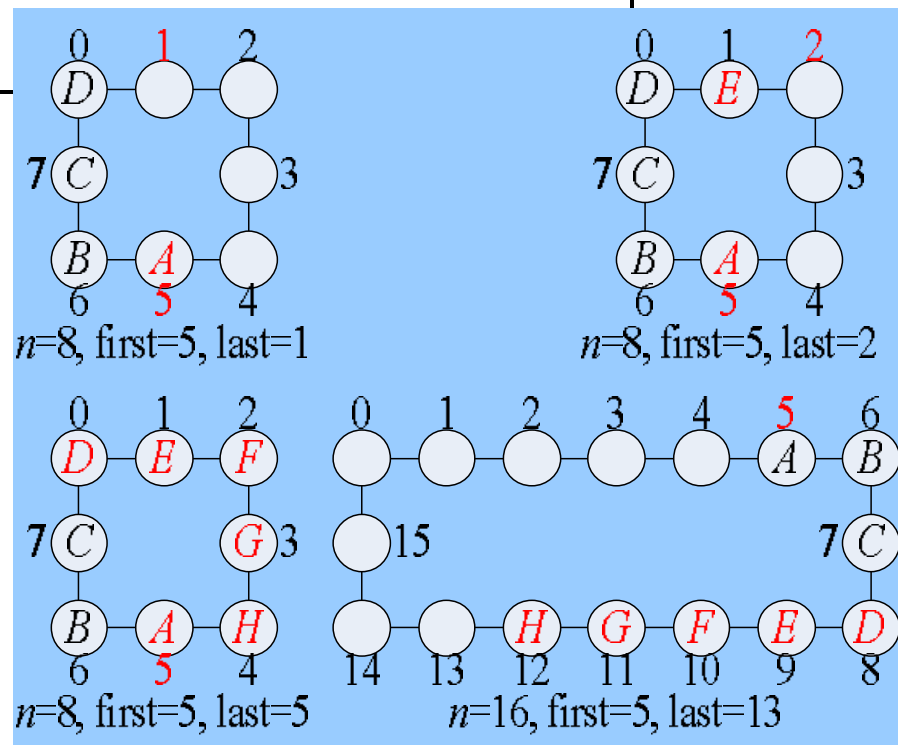


图2-9　入队

```
void push(const T &v)
{   if(sz >= buf.capacity()) // 队列已满，需扩容
    {   buf.reserve(2 * sz); // 扩容
        // 扩容后，需将循环到数组首部的元素移到新空间
        for(int i = 0; i < first; ++i) // 原[0, first)中的元素移到新空间
            buf[i + sz] = buf[i];
    }
    int last = (first + sz) % buf.capacity(); // 插入位置
    buf[last] = v; // 指定插入位置的元素
    ++sz; // 增加了一个元素
}
```



（3）删除元素。只能从队首一端删除元素。如图2-10所示。

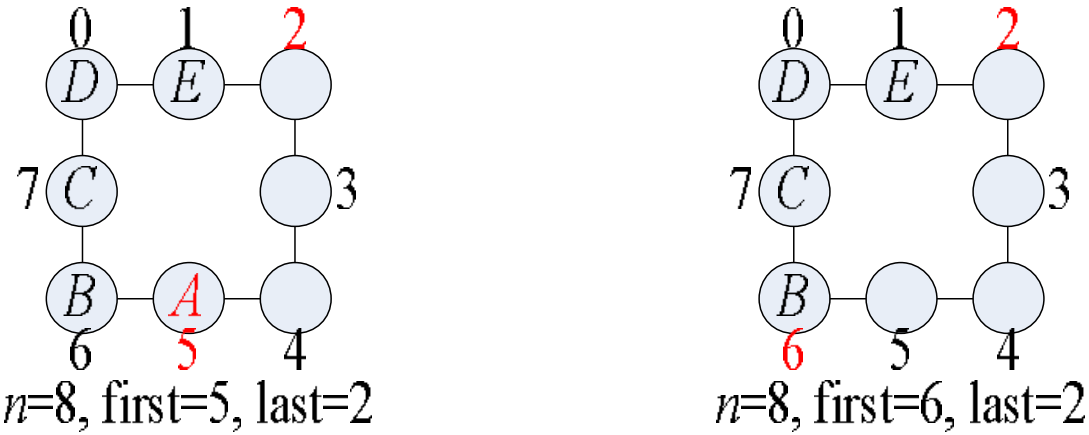


图2-10　出队

```
void pop()
{   if(buf.empty()) return;
    first = (first + 1) % buf.capacity();
    --sz; // 减少了一个元素
}
```

3. 其余常用操作

只介绍空队列判断。

```
    bool empty() const
    {   return sz <= 0;
    }
};
```

2.4 矩阵

C++ STL没有专门提供矩阵类，这里给出矩阵类的一种C++实现，保存在文件Matrix.hpp中。这里只给出了矩阵类最基本的组成部分，相关运算的实现请读者自行补充。

1. 预处理

需要引入下列预处理文件。

```
// Matrix.hpp, C++ 11
#pragma once
#include <vector>
#include <iostream>
using namespace std;
```

2. 数据成员

包括元素数组、行数和列数等3个成员。

```
template<class T>
class Matrix
{   vector<T> buf; // 用一维数组保存矩阵元素
    size_t r = 0, c = 0; // 行数和列数
public:
```

3. 初始化和赋值

（1）默认的初始化和赋值。默认初始化由成员变量声明中的默认值完成，拷贝构造函数、析构函数和赋值运算符函数均由编译器生成。

```
Matrix() {} // 默认初始化, 由成员变量声明中的默认值完成
```

（2）指定初始值。指定元素数组、行数和列数等成员的初始值。

```
// 根据行数和列数初始化
Matrix(size_t row, size_t col):
    buf(vector<T>(row * col)), r(row), c(col) {}
```

```
// 用行数和列数及指定值初始化
Matrix(size_t row, size_t col, const T &v):
    buf(vector<T>(row * col, v)), r(row), c(col) {}
```

（3）使用初始值列表初始化。即使用{}初始化。

```
// 使用形如{{1, 2}, {3, 4}}的初始值列表初始化
Matrix(const initializer_list<initializer_list<T>> &m):
    buf(begin(m)->begin(), prev(end(m))->end()), // 指定元素
    r(m.size()), c(begin(m)->size()) // 指定行数和列数
{}
```

（4）元素填充。每个元素都是指定的值。

```
void assign(const T &v) // 每个元素都是v
{   buf.assign(r * c, v);
} // 耗时O(size)
```

4. 元素访问

提供只读版本和可读写版本。

```
// 使用M(i, j)的形式访问矩阵元素
auto operator()(size_t i, size_t j) -> decltype(buf[0])
{   return buf[i * c + j];
}
```

```
auto operator()(size_t i, size_t j) const -> decltype(buf[0])
{   return buf[i * c + j];
}
```

5. 大小

包括行数和列数。

```
size_t rows() const // 行数
{   return r;
}
```

```
size_t cols() const // 列数
{   return c;
}
```

```
};
```

6. 输出

输出矩阵的所有元素。

```
template<class T> // 输出矩阵的所有元素
ostream &operator<<(ostream &out, const Matrix<T> &m)
{   size_t r = m.rows(), c = m.cols();
    for(size_t i = 0; i < r; ++i)
    {   for(size_t j = 0; j < c; ++j)
            out << m(i, j) << ' ';
        out << endl; // 输出一行元素后换行
    }
    return out;
} // 耗时O(r * c)
```

2.5# 练习题

1. 参考C++ STL的vector类模板构造一个自定义的顺序表类模板。
2. 参考C++ STL的forward_list类模板构造一个自定义的单向链表类模板。
3. 参考C++ STL的stack类模板构造一个自定义的栈类模板。
4. 参考C++ STL的queue类模板构造一个自定义的队列类模板。
5. 为2.4节介绍的Matrix类模板补充矩阵相加、矩阵相减、矩阵相乘、矩阵与数相加、矩阵与数相减、矩阵与数相乘、矩阵与数相除、矩阵转置和元素反号等运算。