数据结构与算法

DATA STRUCTURE

第九讲 特殊矩阵和字符串

信息管理与工程学院

2017 - 2018 第一学期

课堂内容

- 特殊矩阵
- 字符串
- 匹配算法

特殊矩阵

特殊矩阵

- 特殊矩阵是指非零元素或零元素的分布有一定规律的矩阵。
- 特殊矩阵的压缩存储主要是针对阶数很高的特殊矩阵。为节省存储空间，对可以不存储的元素，如零元素或对称元素，不再存储。
  - 对称矩阵
  - 带状矩阵

对称矩阵的压缩存储

- 设有一个n×n的对称矩阵A：

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

在对称矩阵中，$a_{ij} = a_{ji}$

为节约存储，只存对角线及对角线以上的元素，或者只存对角线或对角线以下的元素。前者称为上三角矩阵，后者称为下三角矩阵。

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

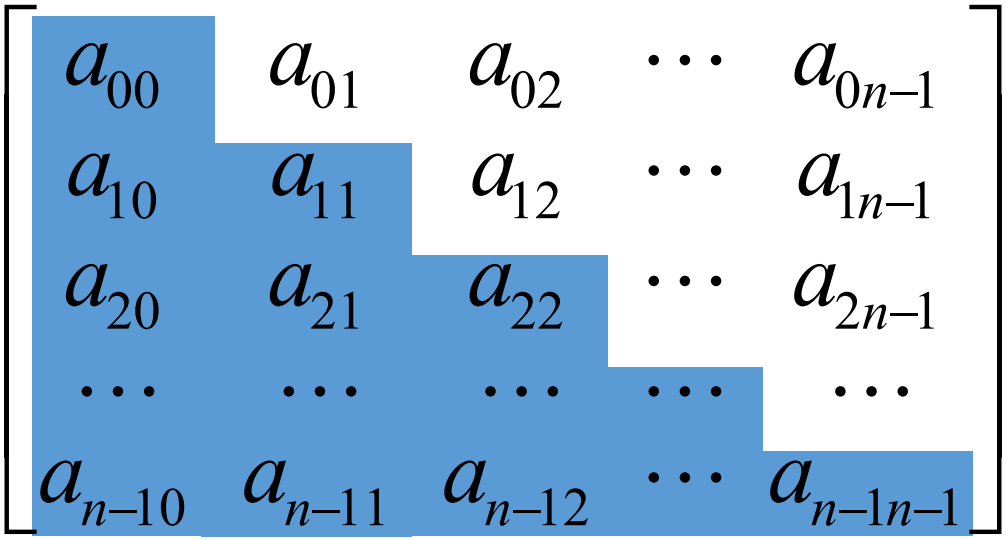
上三角矩阵

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

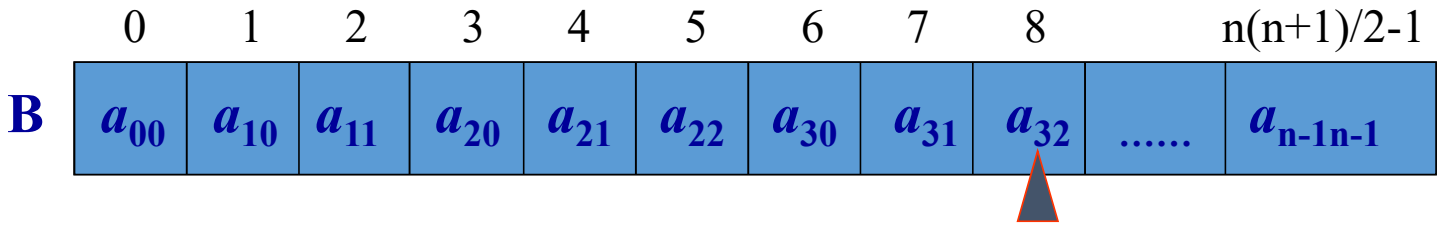
下三角矩阵

- 把它们按行存放于一个一维数组B中，称之为对称矩阵A的压缩存储方式。
- 数组B共有元素

$$n + (n-1) + \cdots + 1 = n * (n+1)/2$$



下三角矩阵



若$i \geq j$，数组元素A[i][j]在数组B中的存放位置为

$$1+2+\cdots+i+j = (i+1)*i/2 + j$$

前i行元素总数　　第i行第j个元素前元素个数

下三角矩阵位置和数组下标转换

- 若$i < j$，数组元素A[i][j]在矩阵的上三角部分，在数组B中没有存放，可以找它的对称元素A[j][i]的位置$j*(j+1)/2+i$。

- 若已知某矩阵元素位于数组B的第k个位置，可寻找满足

  $$i(i+1)/2 \le k < (i+1)*(i+2)/2 = i(i+1)/2 + (i+1)$$

  的i，此即为该元素的行号。

  $$j=k-i*(i+1)/2$$

  此即为该元素的列号。

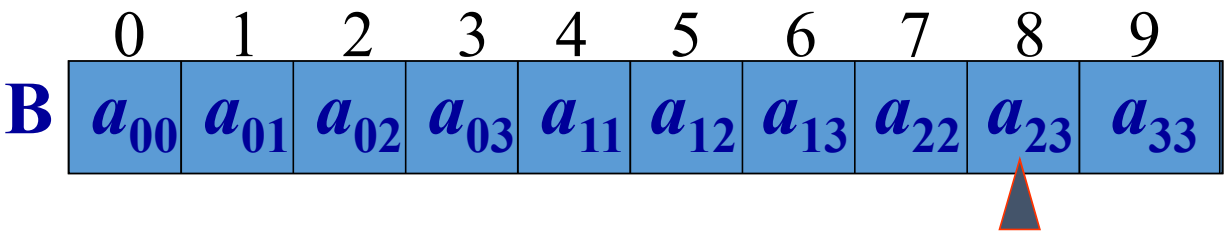
例如，当$k=8$，$3*4/2 = 6 \le k < 4*5/2 = 10$

取$i=3$，则 $j = 8-3*4/2 = 2$

$n=4$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

上三角矩阵



若$i \le j$，数组元素A[i][j]在数组B中的存放位置为

$$n+(n-1)+(n-2)+\cdots+(n-i+1)+j-i$$

前i行元素总数

第i行第j个元素前元素个数

上三角矩阵位置和数组下标转换

若$i \le j$，数组元素A[i][j]在数组B中的存放位置为

$n+(n-1)+(n-2)+\cdots+(n-i+1)+j-i$

$= (2*n-i+1)*i/2+j-i$

$= (2*n-i-1)*i/2+j$

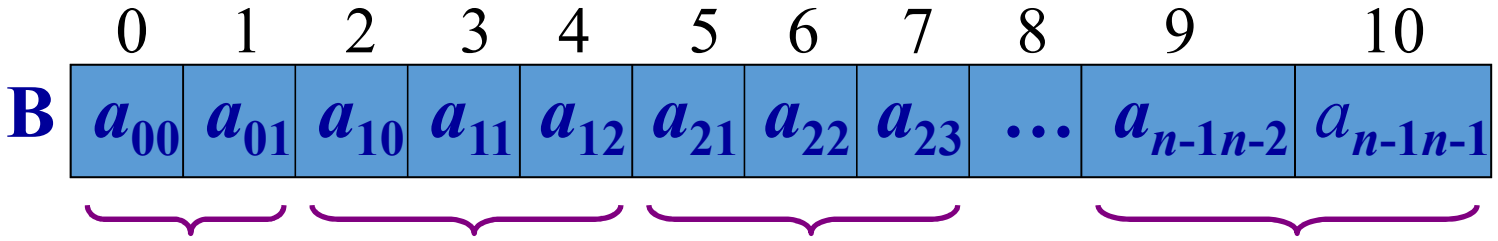
若$i>j$，数组元素A[i][j]在矩阵的下三角部分，在数组B中没有存放。因此，找它的对称元素A[j][i]。

A[j][i]在数组B的第$(2*n-j-1)*j/2+i$的位置中找到。

类似的可以计算数组B里的k位置对应矩阵A里的位置.

带状（三对角）矩阵的压缩存储

$$A=\begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & a_{n-2n-3} & a_{n-2n-2} & a_{n-2n-1} \\ 0 & 0 & 0 & 0 & a_{n-1n-2} & a_{n-1n-1} \end{bmatrix}$$



带状矩阵位置和数组下标转换

- 三对角矩阵中除主对角线及在主对角线上下最临近的两条对角线上的元素外，所有其它元素均为0，总共有$3n-2$个非零元素。
- 将三对角矩阵A中三条对角线上的元素按行存放在一维数组B中，且a_{00}存放于B[0]。
- 在三条对角线上的元素a_{ij}满足

$$0 \le i \le n-1, \quad i-1 \le j \le i+1$$

- 在一维数组B中A[i][j]在第i行，它前面有$3*i-1$个非零元素，在本行中第j列前面有$j-i+1$个，所以元素A[i][j]在B中位置为 $k=2*i+j$。

带状矩阵位置和数组下标转换

- 若已知三对角矩阵中某元素A[i][j]在数组B[ ]存放于第k个位置，则有

$$i = \lfloor (k + 1)/3 \rfloor$$
$$j = k - 2*i$$

- 例如，当$k = 8$时，

$$i = \lfloor (8+1)/3 \rfloor = 3, \quad j = 8-2*3 = 2$$

当k=10时，

$$i = \lfloor (10+1)/3 \rfloor = 3, \quad j = 10-2*3 = 4$$

稀疏矩阵 (Sparse Matrix)

$$\mathbf{A}_{6\times7}=\begin{pmatrix}0 & 0 & 0 & 22 & 0 & 0 & 15\\ 0 & 11 & 0 & 0 & 0 & 17 & 0\\ 0 & 0 & 0 & -6 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 39 & 0\\ 91 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 28 & 0 & 0 & 0 & 0\end{pmatrix}$$

非零元素个数远远少于矩阵元素个数

稀疏矩阵的定义

- 设矩阵$A_{m\times n}$中有*t*个非零元素，若*t*远远小于矩阵元素的总数$m\times n$，则称矩阵A为稀疏矩阵。
- 为节省存储空间，应只存储非零元素。
- 非零元素的分布一般没有规律，应在存储非零元素时，同时存储该非零元素的行下标*row*、列下标*col*、值*value*。
- 每一个非零元素由一个三元组唯一确定：

  (行号 *row*, 列号 *col*, 值 *value*)

稀疏矩阵的类定义

```
struct Triple
{
    int Row;
    int Col;
    int Val;
};

class SparseMatrix
{
public:
    //构造函数，建立一个rowNum行与colNum列的稀疏矩阵
    SparseMatrix(int rowNum, int colNum, int nonZeroSize);
    ~SparseMatrix();

    //对*this指示的三元组数组中各个三元组交换其行、列的值，
    //得到其转置矩阵
    SparseMatrix Transpose();

    //快速转置
    SparseMatrix FastTranspose();

    //实现两矩阵相乘
    SparseMatrix Multiply(const SparseMatrix & other);

 private:
    SparseMatrix(const SparseMatrix & other);
    SparseMatrix & operator= (const SparseMatrix & mat);

    int _rowSize;
    int _colSize;
    int _tripleSize;
    Triple* _smArray;
};
```

稀疏矩阵的转置

- 一个$m \times n$的矩阵A，其转置矩阵B是一个$n \times m$的矩阵，且A[i][j]=B[j][i]。即矩阵A 的行成为矩阵B的列，矩阵A的列成为矩阵B的行。

- 在稀疏矩阵的三元组表中，非零矩阵元素按行存放。当行号相同时，按列号递增的顺序存放。

- 稀疏矩阵的转置运算要转化为对应三元组表的转置。

稀疏矩阵

$$\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	行 (row)	列 (col)	值 (value)
[0]	0	3	22
[1]	0	6	15
[2]	1	1	11
[3]	1	5	17
[4]	2	3	−6
[5]	3	5	39
[6]	4	0	91
[7]	5	2	28

转置矩阵

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 28 \\ 22 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 17 & 0 & 39 & 0 & 0 \\ 15 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	行 (row)	列 (col)	值 (value)
[0]	0	4	91
[1]	1	1	11
[2]	2	5	28
[3]	3	0	22
[4]	3	2	−6
[5]	5	1	17
[6]	5	3	39
[7]	6	0	16

用三元组表表示的稀疏矩阵及其转置

	行 (row)	列 (col)	值 (value)
[0]	0	3	22
[1]	0	6	15
[2]	1	1	11
[3]	1	5	17
[4]	2	3	−6
[5]	3	5	39
[6]	4	0	91
[7]	5	2	28

	行 (row)	列 (col)	值 (value)
[0]	0	4	91
[1]	1	1	11
[2]	2	5	28
[3]	3	0	22
[4]	3	2	−6
[5]	5	1	17
[6]	5	3	39
[7]	6	0	16

稀疏矩阵转置算法思想

- 设矩阵列数为*Cols*，对矩阵三元组表扫描*Cols*次，第*k*次检测列号为*k*的项。

- 第*k*次扫描找寻所有列号为*k*的项，将其行号变列号、列号变行号，顺次存于转置矩阵三元组表。

稀疏矩阵的转置

```
SparseMatrix SparseMatrix::Transpose()
{
    SparseMatrix mat(_rowSize, _colSize, _tripleSize);

    int index = 0;

    //按列号做扫描，做Cols趟
    for (int col = 0; col < _colSize; col++)
    {
        for (int i = 0; i < _tripleSize; i++)
        {
            if (_smArray[i].Col != col)
            {
                continue;
            }

            //新三元组的行号，列号，和值
            mat._smArray[index].Row = _smArray[i].Col;
            mat._smArray[index].Col = _smArray[i].Row;
            mat._smArray[index].Val = _smArray[i].Val;

            index++;
        }
    }

    return mat;
}
```

时间复杂度

- 设矩阵三元组表总共有t项，或者说非零元个数是t，上述算法的时间代价为$O(n*t)$。

- 若矩阵有300行、300列、10,000个非零元素，总共有3,000,000次处理。

快速转置算法

- 能不能只扫描一遍三元数组？

- 建立辅助数组trowSize和trowStart，记录矩阵转置后各行非零元素个数和各行元素在转置三元组表中开始存放位置。

- 扫描矩阵三元组表，根据某项列号，确定它转置后的行号，查trowStart表，按查到的位置直接将该项存入转置三元组表中。

三元组	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)
行row	0	0	1	1	2	3	4	5
列col	3	6	1	5	3	5	0	2
值value	22	15	11	17	-6	39	91	28

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	语 义
trowSize	1	1	1	2	0	2	1	矩阵 A 各列非零元素个数
trowStart	0	1	2	3	5	5	7	矩阵 B 各行开始存放位置

快速转置

```
SparseMatrix SparseMatrix::FastTranspose()
{
    SparseMatrix mat(_rowSize, _colSize, _tripleSize);

    int trowSize[_colSize] = {0};
    int trowStart[_colSize] = {0};

    //统计矩阵b中第i列非零元素个数
    for (int i=0; i < _tripleSize; i++)
    {
        trowSize[_smArray[i].Col]++;
    }

    //trowStart[i]等于矩阵b的第i列的开始存放位置
    for (int i=1; i < _colSize; i++)
    {
        trowStart[i] = trowStart[i-1] + trowSize[i-1];
    }

    for (int i=0; i < _tripleSize; i++)
    {
       //j为第i个非零元素在b中应存放的位置
        int j = trowStart[_smArray[i].Col];

        mat._smArray[j].Row = _smArray[i].Col;
        mat._smArray[j].Col = _smArray[i].Row;
        mat._smArray[j].Val = _smArray[i].Val;
        trowStart[_smArray[i].Col]++;
    }

    return mat;
}
```

时间复杂度

- 设矩阵三元组表总共有t项，或者说非零元个数是t，上述算法的时间代价为

$$t + n + t = O(n + t)。$$

- 若矩阵有300行、300列、10,000个非零元素，总共有20,300次处理。

字符串string

字符串例子

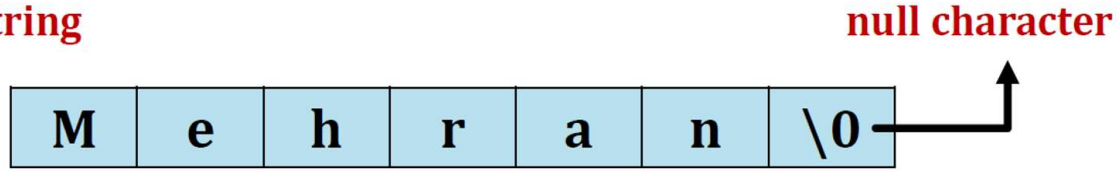
注意单引号是指单个字符，‘c’内只能包含一个字符。

```
"Mehran University"    //String with multiple words
"14CS"                 //String with single word
""                     //Empty string
"c"                    //String with single character
```

C-style字符串

定义 char name[] = “Mehran”;

- This is C-String



- 这里name其实就是char array，不过C++会在最后加上null terminator ‘\0’
- 所以sizeof(name)是7，不是6
- 不过cout << name; 还是Mehran，最后 ‘\0’ 看不到
- 注意char name[6] = “Mehran” C++会报错
- 注意char name[8] = “Mehran” 是可以的

C-style字符串初始化

```
char name [ ] = “Asghar” ;
char name [10] = “Asghar” ;
char name [ ] = { ’A’ , ’s’ , ’g’ , ’h’ , ’a’ , ’r’ , ’\0’ };
char name [7] = { ’A’ , ’s’ , ’g’ , ’h’ , ’a’ , ’r’ , ’\0’ };
```

注意点

- 只能在初始化的时候用字符串赋值
- 之后得按数组的方式单个字符更新
- 如果指定的数组长度大，那么不足的元素补\0

```
char mystring[] = "string";   // OK
mystring = "news";            // NOT OK
mystring[1] = 'p';            // OK

cout << mystring << endl;     // 输出spring

char name[256] = "Zhang";     // OK,不足的补0
```

用户输入字符串

注意使用cin.getline更安全

```
char name[255]{};
cout << "Enter your name: ";
cin >> name;                    // 可能会溢出
cin.getline(name, 255); // 更安全，多于254的输入会被忽略

cout << "Your name is " << name << endl;
```

更多函数#include <cstring>

- strcpy(dest, src)，把src字符串复制到dest
- strcpy_s(dest, size, src), 更安全，能指定复制长度size
- strlen()，返回字符串长度，不包括null terminator
- strcat()，把一个字符串复制到另一个后面
- strncat()，安全版本，能指定长度
- strcmp()，比较两个字符串是否相等
- strncmp()，比较两个字符串到指定长度为止是否相等

std::string in <string>

- C++在c-style字符串基础上实现的自定义数据类型，更容易使用，更安全
- 定义string变量

```
string name1 = "Asghar" ;
string name2 ( "Asghar" );
string name3 ( name2 );
```

- 输出cout << name << endl;
- 输入std::getline(cin, name);

std::string in <string>

- 直接用加号+来合并两个字符串

```
string firstName("Zhu");
string lastName("Zhu");

string fullName = lastName + " " + firstName;
cout << fullName << endl;
```

- 自带输出长度功能，和strlen一样，不包括null terminator

```
cout << fullName.length() << endl;
```

- 尽量使用std::string，而不是c-style string，或者说字符数组。

自定义串

- 我们同样要对c-style string进行封装
- 使得支持insert，delete，search， replace，etc
- 尤其是string匹配算法

串的数组存储表示

（1）静态数组存储方法

- 为字符串变量分配一个固定长度的存储空间
- 一般用定长数组加以实现

```
const int nMaxLen = 1024;          //字符串的最大长度
class Sstring
{
private:
    int  nLen;                     //字符串的当前长度
    char ch[nMaxLen+1];            //字符串存储空间
};
```

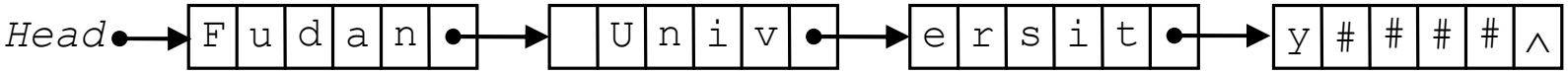
串的数组存储表示

（2）动态数组存储方法

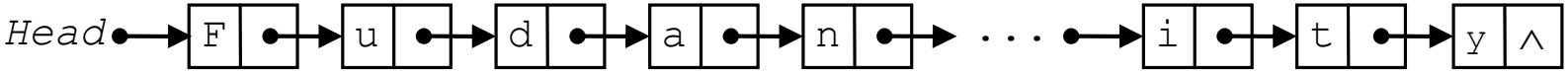
- 使用C++提供的new操作符在heap上分配一块连续的内存空间

```
class MyString
{
private:
    char * _szData;          //指向当前字符串的指针
    int _size;               //字符串长度
};
```

串的块链存储表示



(a) 结点大小为 5



(b) 结点大小为 1

图3-1 字符串“Fudan University”的块链存储方式

- 结点大小的选择和子串存储方式是影响效率的重要因素

$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

串的实现

作业：

- 实现MyString类，
- 参考书上例子和之前的IntArray，
- 尤其是加亮的三个函数
  - Find，现在用暴力匹配，之后我们再添加优化算法。
  - Resize，把字符串长度改变，原有的内容复制剪切过来
  - Reserve，改变数组内存分配大小
  - Append，和“+”区别，+是指两个字符串合并，然后返回一个新的字符串；append是指把别的字符串附加在自己后面，返回的也是对自己的引用。注意这里要考虑性能问题，也就是如果对几十万个小字符串进行append操作。

```
class MyString
{
public:
    MyString(const char * pszValue);
    ~MyString();

    // copy constructor
    MyString(const MyString & other);

    // Overloaded assignment
    MyString& operator= (const MyString& rhs);
    // see Append
    MyString& operator+=(const MyString& rhs);
    // return a new MyString
    MyString operator+(const MyString& rhs);
    // Access character by index
    char & operator[] (int index);
    const char & operator[](int index) const;
    // Compare two strings
    int operator== (const MyString& str) const;

    // TODO: String matching function. Brute force now.
    int Find(MyString & subStr) const;

    // TODO: implement an efficient algorithm. Test append lots of small string.
    MyString & Append(const MyString & str);

    // TODO: need to copy/cut previous string, if size is within capacity.
    void Resize(int n);
    // change capacity size.
    void Reserve(int capacity);

    // Insert a new string at index; previous content after index are placed afterwards.
    void Insert(int index, const char * pszStr);
    // Delete substring of size length starting at index.
    void Remove(int index, int length);

    const char * GetString() const { return _data; }
    inline int Length() { return _size; }
    inline bool IsEmpty() { return _size <= 0 || _data == nullptr; }

    friend std::ostream& operator<<(std::ostream& out, const MyString &str);

private:
    int _capacity;
    int _size;
    char* _data;
};
```

Q&A

Thanks!