Array

- The array as an abstract data type
- Sparse matrix
- The string as an data type

String

A string $S = s_0, s_1, \ldots, s_{n-1}$,

where $s_i \in$ char, $0 \leq i < n$, n is the length.

- 子串：串中任意个连续字符组成的子序列。
- 主串：包含子串的串。
- 通常将子串在主串中首次出现时，该子串首字符对应的主串中的序号，定义为子串在主串中的位置。
- 例如，设A和B分别为： A = “This is a string” B = “is”

  则 B 是 A 的子串，A 为主串。

  B 在 A 中出现了两次，首次出现所对应的主串位置是2（从0开始）。因此，称 B 在 A 中的位置为2。
- 特别地，空串是任意串的子串，任意串是其自身的子串

String

```
#ifndef ASTRING_H //定义在文件“Astring.h”中
#define ASTRING_H
#define defaultSize = 128;     //字符串的最大长度
class AString {
//对象: 零个或多个字符的一个有限序列
private:
    char *ch;                      //串存放数组
    int curLength;                 //串的实际长度
     int maxSize;                  //存放数组的最大长度
public:
```

```
 AString(int sz = defaultSize);     //构造函数
AString(const char *init );         //构造函数
AString(const AString& ob);     //复制构造函数
～AString() {delete [ ]ch; }              //析构函数
int Length() const { return curLength; }     //求长度
Astring& operator() (int pos, int len);   //求子串
bool operator == (AString& ob) const
    { return strcmp (ch, ob.ch) == 0; }
     //判串相等. 若串相等, 则函数返回true
bool operator != (AString& ob) const
    { return strcmp (ch, ob.ch) != 0; }
    //判串不等. 若串不相等, 则函数返回true
```

```
bool operator ! () const { return curLength == 0; }
  //判串空否。若串空,则函数返回true
AString& operator = (AString& ob);       //串赋值
AString& operator += (AString& ob);      //串连接
char& operator [ ] (int i);              //取第 i 个字符
int Find (AString& pat, int k) const;    //串匹配
};
```

字符串的构造函数

```
AString::AString(int sz) {
//构造函数：创建一个空串
    maxSize = sz;
    ch = new char[maxSize+1];        //创建串数组
    if (ch == NULL)
        { cerr << “存储分配错!\n”;  exit(1); }
    curLength = 0;
    ch[0] = ‘\0’;
};
```

字符串的构造函数

```
AString::AString(const char *init) {
//复制构造函数: 从已有字符数组*init复制
    int len = strlen(init);
    maxSize = (len > defaultSize) ? len : defaultSize;
    ch = new char[maxSize+1];     //创建串数组
    if (ch == NULL)
      { cerr << "存储分配错！\n";  exit(1); }
    curLength = len;                    //复制串长度
    strcpy(ch, init);                   //复制串值
};
```

字符串的复制构造函数

```
AString :: AString(const AString& ob) {
 //复制构造函数：从已有串ob复制
    maxSize = ob.maxSize;              //复制串最大长度
    ch = new char[curLength+1];      //创建串数组
    if (ch == NULL)
        { cerr << "存储分配错！\n";  exit(1); }
    curLength = ob.curLength;          //复制串长度
    strcpy(ch, ob.ch);                 //复制串值
 };
```

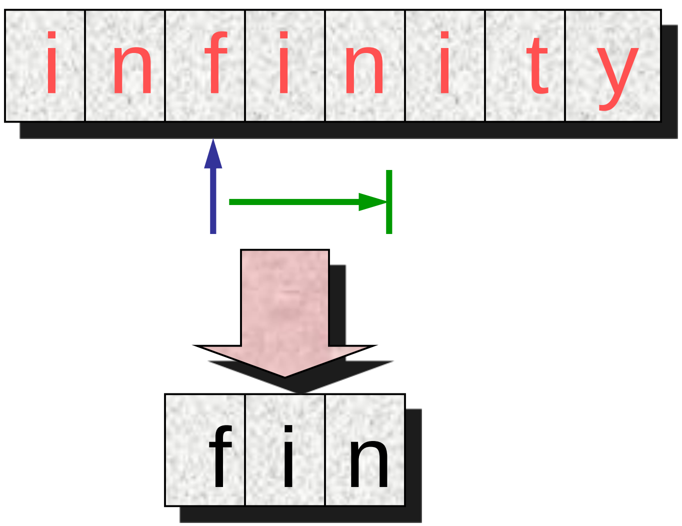
字符串重载操作的使用示例

序号	重载操作	操作	使用示例 (设使用操作的当前串为S: ‘southeast’)
1	( ) (int pos, int len)	取子串	S1 = S(3, 2) //S1结果为 ‘th’
2	== (const AString& ob)	判两串相等	S == S1, //若S与S1相等, 为true,否则为false
3	!= (const AString& ob)	判两串不等	S != S1 , //若S与S1不等,为true,否则为false
4	! ()	判串空否	!S //若串S为空,为 true,否则为false

序号	重载操作	操作	使用示例 (设使用操作的当前串为S: ‘southeast’)
5	= (const AString& ob)	串赋值	S1 = S S1结果为‘southeast’
6	+= (const AString& ob)	串连接	若设S1为‘ university’, 执行S += S1, //S结果为‘southeast university’
7	[ ] (int i)	取第 i 个字符	S[5] 取出字符为‘e’

提取子串的算法示例

pos = 2, len = 3

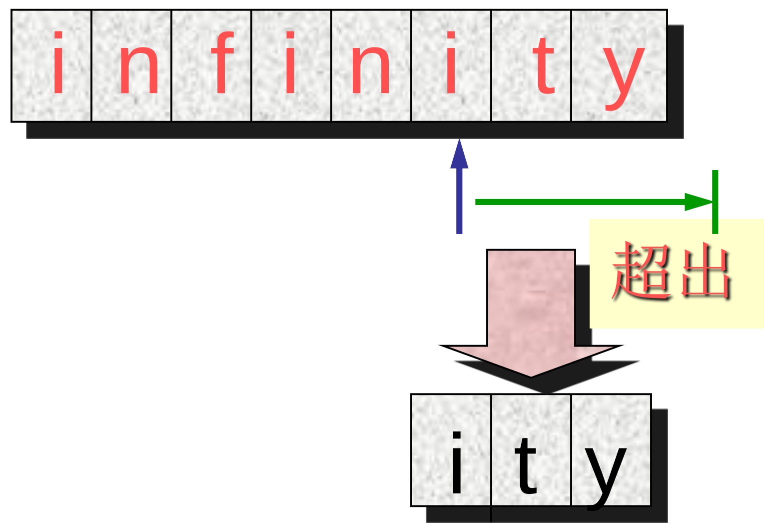


pos+len−1

≤ curLen−1

可以全部提取

pos = 5, len = 4



pos+len −1

≥ curLen

只能从pos取到串尾

串重载操作：提取子串

```
AString  AString::operator () (int pos, int len) {
//从串中第 pos 个位置起连续提取 len 个字符形成
//子串返回
    AString  temp;                       //建立空串对象
    if (pos >= 0 && pos+len−1 < maxLen && len > 0)
    {                                    //提取子串
        if (pos+len−1 >= curLength)
            len = curLength − pos;       //调整提取字符数
        temp.curLength = len;            //子串长度
```

```
        for (int i = 0, j = pos; i < len; i++, j++)
            temp.ch[i] = ch[j];        //传送串数组
        temp.ch[len] = '\0';           //子串结束
    }
    return  temp;
};
```

- 例： 串 st = “university”, pos = 3, len = 4
  使用示例　subSt = st(3, 4)
  提取子串　subSt = “vers”

串重载操作：串赋值

```
AString& AString::operator = (const AString& ob) {
    if (&ob != this) {   //若两个串相等为自我赋值
        delete []ch;
        ch = new char[maxSize+1];   //重新分配
        if (ch == NULL)
          { cerr << "存储分配失败!\n "; exit(1); }
        curLength = ob.curLength; strcpy(ch,ob.ch);
    }
    else cout << "字符串自身赋值出错!\n";
    return this;
};
```

串重载操作：取串的第i个字符

```
char AString::operator [ ] (int i) {
//串重载操作：取当前串*this的第i个字符
    if (i < 0 && i >= curLength)
        { cout << "字符串下标超界!\n "; exit(1); }
    return ch[i];
};
```

- 例：串 st = “university”,
  使用示例 newSt = st；newChar = st[1];
  数组赋值结果 newSt = “university”
  提取字符结果 newChar = ‘n’

串重载操作：串连接

```
AString& AString::operator += (const AString& ob)
{
    char *temp = ch;                    //暂存原串数组
    int n = curLength + ob.curLength;             //串长度累加
    int m = (maxSize >= n) ? maxSize : n; //新空间大小
    ch = new char[m];
    if (ch == = NULL)
        { cerr << "存储分配错!\n "; exit(1); }
    maxSize = m; curLength = n;
    strcpy(ch, temp);                    //拷贝原串数组
    strcat(ch, ob.ch);           //连接ob串数组
```

```
    delete []temp;
    return this;
};
```

- 例：串 st1 = “southeast ”,
           st2 = “university”,

  使用示例　st1 += st2;

  连接结果　st1 = “southeast university”

  　　　　　st2 = “university”

String Pattern Matching

- 在主串中寻找子串（第一个字符）在串中的位置
- 在模式匹配中，子串称为模式(Pattern)，主串称为目标(Target)
- 例　　目标 T : “Beijing”

  模式 P : “jin”

  匹配结果 = 3

BF Algorithm

i=2

第1趟 *T* a b b a b a

P a b a

j=2

i=1

第2趟 *T* a b b a b a

P a b a

j=0

i=2

第3趟 *T* a b b a b a

P a b a

j=0

i=6

第4趟 *T* a b b a b a

P a b a

j=3

BF Algorithm

```
int AString::Find(AString& pat, int k) const {
//在当前串中从第 k 个字符开始寻找模式 pat 在当
//前串中匹配的位置。若匹配成功，则函数返回首
//次匹配的位置，否则返回 -1。
    int n = curLength, m = pat.curLength;
    for (int i = k; i <= n-m; i++) {
            for (int j = 0; j < m; j++)
                if (ch[i+j] != pat.ch[j]) break;   //本次失配
          if (j == m) return i;
                         //pat扫描完，匹配成功
     }
   return -1;      //pat为空或在*this中找不到它
};
```

- Worst case
  - 若设 n 为目标串长度，m 为模式串长度，则匹配算法最多比较 n−m+1趟，
  - 每趟比较都在比较到模式串尾部才出现不等，要做 m 次比较，总比较次数将达到(n−m+1)*m。
  - 在多数场合下 m 远小于 n，因此，算法的运行时间为O(n*m)。
- Problem
  - rescanning

- 只要消除每趟失配后为实施下一趟比较时目标指针的回退，可以提高模式匹配效率。

The Knuth-Morris-Pratt Algorithm

目标 T t_0 t_1 t_2 …… t_{m-1} … t_{n-1}

$\updownarrow$ $\updownarrow$ $\updownarrow$ $\updownarrow$

模式 pat p_0 p_1 p_2 …… p_{m-1}

目标 T t_0 t_1 t_2 …… t_{m-1} t_m … t_{n-1}

$\updownarrow$ $\updownarrow$ $\updownarrow$ $\updownarrow$

模式 pat p_0 p_1 …… p_{m-2} p_{m-1}

目标 T t_0 t_1 …… t_i t_{i+1}…… t_{i+m-2} t_{i+m-1}… t_{n-1}

$\|$ $\|$ $\|$ $\|$

模式 pat p_0 p_1 …… p_{m-2} p_{m-1}

Finding K

- 对于不同的 j（失配位置），k 的取值不同，
- K
  - 仅依赖于模式 P 本身前 j 个字符的构成，与目标T无关。
  - 用一个 next 特征向量来确定：
    - 当模式 P 中第 j 个字符与目标 S 中相应字符失配时，
    - 模式 P 中应当由哪个字符（设为第k+1个）与目标S中刚失配的字符重新继续进行比较。

- 设模式 $P = p_0p_1 \ldots p_{m-2}p_{m-1}$, next特征向量定义如下：

$$\mathbf{next(j)} = \begin{cases} -1, & j = 0 \\ k+1, & 0 \le k < j-1 \text{且使得} p_0p_1\ldots p_k = p_{j-k-1}p_{j-k}\ldots p_{j-1} \text{的最大整数} \\ 0, & \text{其他情况} \end{cases}$$

- 示例

j	0	1	2	3	4	5	6	7
P	a	b	a	a	b	c	a	c
next(j)	−1	0	0	1	1	2	0	1

The KMP Algorithm

- 若设在进行某一趟匹配比较时在模式 P 的第 j 位失配：
  - 如果 $j > 0$，那么在下一趟比较时，模式串 P的起始比较位置是 $p_{next(j)}$，目标串 T 的指针不回溯，仍指向上一趟失配的字符；
  - 如果 $j = 0$，则目标串指针 T 进一，模式串指针 P 回到 p_0，继续进行下一趟匹配比较。

The KMP Algorithm

```
int AString::fastFind(AString& pat, int k,
        int next[]) const {
//从 k 开始寻找 pat 在 this 串中匹配的位置。若找
//到，函数返回 pat 在 this 串中开始位置，否则函
//数返回−1。数组next[ ] 存放 pat 的next[j] 值。
    int posP = 0,  posT = k;//两个串的扫描指针
    int lengthP = pat.curLength;       //模式串长度
    int lengthT = curLength;           //目标串长度
     while (posP < lengthP && posT < lengthT)
          if (posP == −1 || pat.ch[posP] == ch[posT])
           { posP++;  posT++; }          //对应字符匹配
          else posP = next[posP];   //求pat下趟比较位置
     if (posP < lengthP) return −1;    //匹配失败
     else return posT−lengthP;                //匹配成功
};
```

An Example

第1趟 Target *a c a b a a b a a b c a c a a b c*

Pattern *a b a a b c a c*

× j=1 ⇨ next(1) = 0，下次p_0

第2趟 Target *a c a b a a b a a b c a c a a b c*

Pattern *a b a a b c a c*

× j=0 ⇨ 下次p_0, 目标指针进 1

第3趟 Target *a c a b a a b a a b c a c a a b c*

Pattern *a b a a b c a c*

× j=5 ⇨ next(5) = 2, 下次p_2

第4趟 Target *a c a b a a b a a b c a c a a b c*

Pattern (*a b*) *a a b c a c* √

- 此算法的时间复杂度取决于 while 循环
- 由于是无回溯的算法，执行循环时，目标串字符比较有进无退
  - 要么执行 posT 和 posP 进 1，
  - 要么查找next[ ] 数组进行模式位置的右移，
- 然后继续向后比较
- 字符的比较次数最多为 O(lengthT)
  - 不超过目标的长度

Computing Next

- 设模式 $P = p_0 p_1 p_2 \ldots p_{m-1}$ 由 m 个字符组成，
  - next特征向量为next = $n_0 n_1 n_2 \ldots n_{m-1}$，表示了模式的字符分布特征。
- next特征向量从0, 1, 2, …, $m-1$逐项递推计算:

❶ 当 $j = 0$时，$n_0 = -1$；设 $j > 0$ 时 $n_{j-1} = k$。

❷ 当 $k = -1$或 $j > 0$且 $p_{j-1} = p_k$，则 $n_j = k+1$。

❸ 当 $p_{j-1} \neq p_k$ 且 $k \neq -1$，令 $k = n_k$，并让❸循环直到条件不满足。

❹ 当 $p_{j-1} \neq p_k$ 且 $k = -1$，则 $n_j = 0$。

Computing Next

```
void AString::getNext(int next[]) {
    int j = 0, k = –1,  lengthP = curLength;
    next[0] = –1;
    while (j < lengthP)        //计算next[j]

        if ( k == –1 || ch[j] == ch[k]) {

            j++;  k++;

            next[j] = k;
         }
        else
            k = next[k];

};
```

j	0	1	2	3	4	5	6	7
P	a	b	a	a	b	c	a	c
next [*j*]	**−1**	0	0	1	1	2	0	1

$j=0$
$n_0=-1$

$j=1$
$k=-1$
$n_1=$
$=k+1$
$=0$

$j=2$
$k=0$
$p_1\neq p_0$
$k=n_k=$
$=-1$
$n_2=$
$=k+1$
$=0$

$j=3$
$k=0$
$n_3=$
$=k+1$
$=1$

$j=4$
$k=1$
$p_3\neq p_1$
$k=n_k=0$
$p_3=p_0$
$n_4=$
$=k+1$
$=1$

$j=5$
$k=1$
$p_4=p_1$
$n_5=$
$=k+1$
$=2$

$j=6$
$k=2$
$p_5\neq p_2$
$k=n_k=0$
$p_5\neq p_0$
$k=n_k=-1$
$n_5=k+1=0$

$j=7$
$k=0$
$p_6=p_0$
$n_7=k+1$
$=1$

The END