LZ77压缩与解压缩

- 什么是LZ77
- LZ77原理介绍
- LZ77压缩实现
- LZ77解压缩实现
- LZ77压缩比率

1. 什么是LZ77

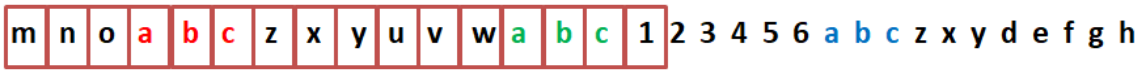
> **1977年**，两位以**色列人Jacob Ziv**和**Abraham Lempel**，发表了一篇论文《A Universal Algorithm for Sequential Data Compression》，**一种通用的数据压缩算法，所谓通用压缩算法指的是这种压缩算法没有对数据的类型有什么限定**，该算法奠基了今天大多数无损数据压缩的核心，为了纪念两位科学家，该算法被称为LZ77，过了一年他们又提了一个类似的算法，称为LZ78。

2. LZ77压缩原理介绍

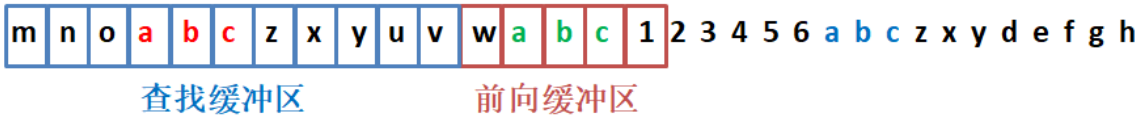
2.1 LZ77原理介绍

> LZ77是基于字节的通用压缩算法，它的原理就是将**源文件中的重复字节**(即在前文中出现的重复字节)使用**(offset，length，nextchar)**的三元组进行替换。比如：
>
> 源文件内容：**mnoabczxyuvwabc123456abczxydefgh**
>
> 上文中**"abc"**字符串有多次重复，那如果用(offset，length，nextchar)方式替换，肯定可以起到压缩的目的。offset：表示待匹配的当前字符距离匹配字符串首字母的距离，length表示匹配字符串的长度，即有多少个字符与前文匹配，nextchar表示当前匹配串的下一个字符，上述原文采用(offset, length, nextchar)三元组替换完成后的结果为：**mnoabczxyuvm(9,3,1)23456(18,6,d)efgh。**
>
> 但是GZIP并没有采用上述的三元组进行替换，而是进行了一个小小的改变，因为nextchar是否出现在三元组中，对压缩率的提升并不能起到什么作用，因此GZIP采用(距离，长度)对的方式进行替换，具体如下：
>
> **mnoabczxyuvm(9,3)123456(18,6)defgh。**

LZ77压缩时，是在一个滑动窗口中进行的，初始状态下，先加载一个窗口的数据：



随着压缩的进行，滑动窗口可以分为两部分：已扫描过的数据与待压缩数据，为理解简单，可以称为：**查找缓冲区与前向缓冲区**



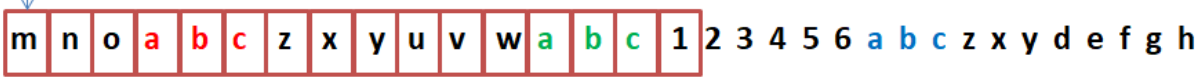
问题：在压缩时，匹配串多长进行距离对替换呢
1个字符？不替换，因为单个字符如果使用长度距离对替换，**替换后文件变大**
2个字符？不替换，因为长度距离对占两个字节，**替换后文件大小不变**
3个字符？替换，替换后长度变短可以起到压缩的效果
结论：**只有当匹配串至少有三个字符时，将进行替换**

```
// 最小匹配长度
static const size_t MIN_MATCH = 3;

// 最大匹配长度
// GZIP认为：长度超过255之后，长度必须要用两个字节表示，会影响压缩率，而大部分情况下，
//           能够匹配的长度都不会超过255，因此长度使用一个字节表示
//           而一个字节能够表示的范围是[0，255]，如果让0表示匹配长度为3个字符，1表示匹配长度为
//           4个字符，...，则一个字节最多可以表示的匹配长度为255+3=258，即最长的匹配长度，
//           如果某个匹配长度超过258，则拆成两个匹配来进行表示
static const size_t MAX_MATCH = 258;
```

压缩过程：

压缩开始后，每扫描到一个新字符后，用当前字符与其后的两个字符构成当前串，在查找缓冲区中查找，若找到匹配，使用长度距离对进行替换，否则将该字符写到压缩文件中。



压缩文件：

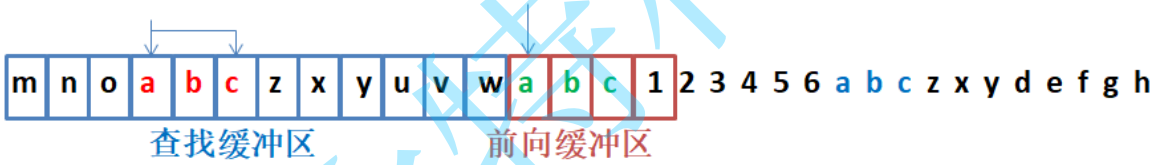
m n o a b c z x y u v

比如：当前字符为w，用"wab"在查找缓冲区中查找匹配串，此时为找到，则将w写入压缩文件中，查找缓冲区向前滑动，前向缓冲区向后移动

压缩文件：

m n o a b c z x y u v w

再比如：当前字符为a，用"abc"在查找缓冲区中查找匹配串，在查找缓冲区中距离当前字符为9的位置找到匹配串，并且匹配长度为3个字符，用距离长度对进行替换，将替换结果写入压缩文件中。



压缩文件：

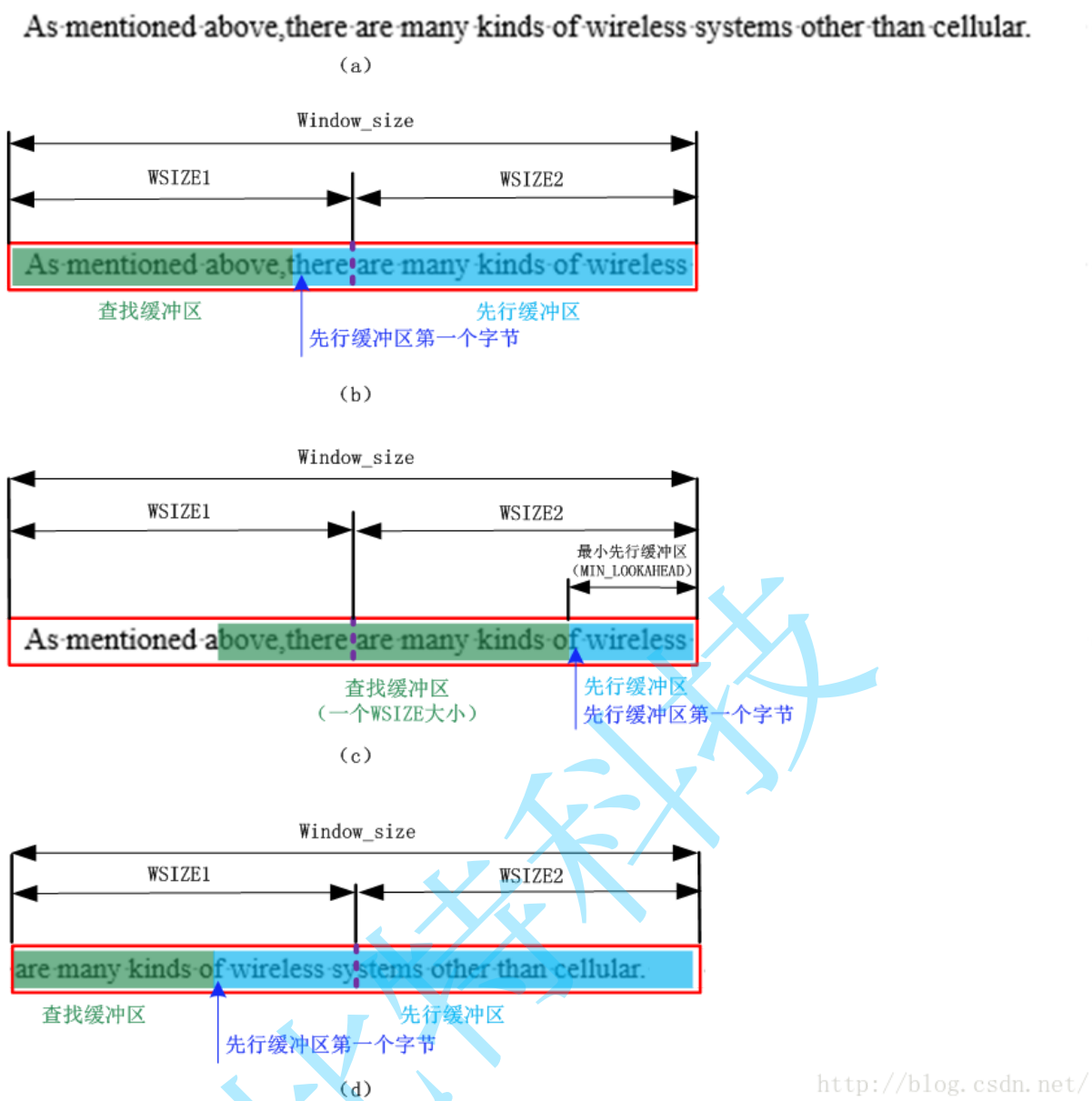
m n o a b c z x y u v w (9,3)

查找缓冲区向前移动吞噬3个字符，前向缓冲区向后退3个字符长度，继续压缩。

但是真正的压缩，是在一个比较大的窗口中进行的，窗口越大，找到匹配的可能性就越大，但不是无限大，因为无限大实，存在两个问题：

1. 空间成本：窗口越大，需要的内存空间就越大
2. 时间成本：窗口越大，查找匹配串时需要耗费的时间也就越多

因此，GZIP决定，窗口的大小取为64K，分为两部分，一个WSIZE大小为32K，如下图所示：



从上图可以看出，随着压缩的进行，窗口被分割成了两个部分，**查找缓冲区和先行缓冲区**。

先行缓冲区：即待压缩的数据，每次都用先行缓冲区中的第一个字符与其后紧跟的两个字符在**查找缓冲区中找匹配**，随着压缩的不断进行，查找缓冲区不断增大，先行缓冲区不断缩小，当查找缓冲区增大到32K之后，就不增大了，随着先行缓冲区向右移动。如果先行缓冲区中的数据少于一个MIN_LOOKAHEAD时，将右窗口中的数据搬移到左窗口，给右窗口中重新补充32K的数据，继续压缩，直到压缩结束。

MIN_LOOKAHEAD=MAX_NATCH + 1；即：保证待压缩区域至少有一个字节以及该字节的一个匹配长度。

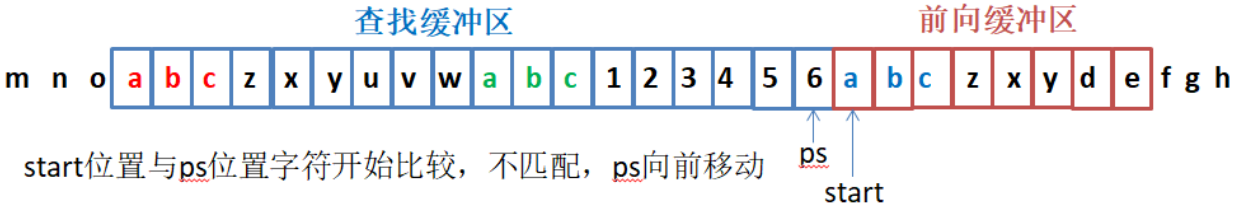
通过以上过程介绍，发现两个问题：

1. 如何高效找最长匹配串
2. 找不到怎么办?
3. 当前向缓冲区中没有字符时或者不够三个字符时如何处理?

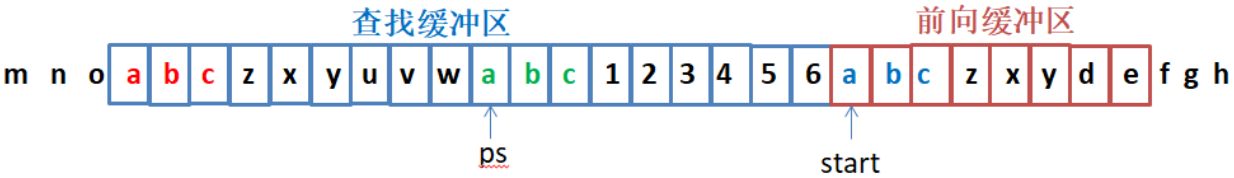
2.2 高效查找最长匹配串

2.2.1 暴力求解

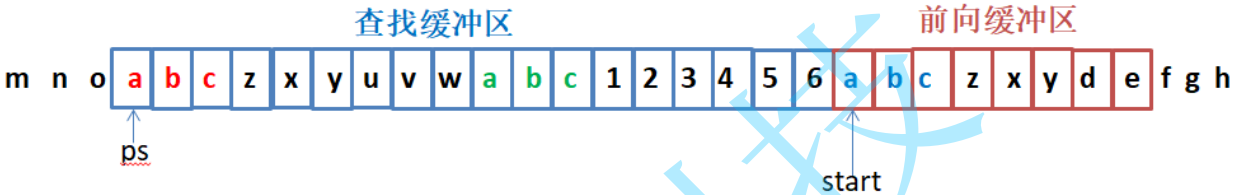
比如: 当前匹配到字符a的位置



start位置与ps位置字符开始比较，不匹配，ps向前移动



start与ps位置字符比较，匹配了3个字符，查找还不能停止，因为此时不一定是最长匹配



start与ps位置字符比较，匹配了6个字符，比上一次匹配长，找到最长匹配，用距离对替换

该算法的性能比较差，是一个O(N^2)的算法，如果待压缩文件比较大，会严重影响压缩的速度。

2.2.2 采用哈希

使用哈希表来提高查询的效率：**使用哈希“桶”保存每三个相邻字符构成的字符串中首字符的窗口索引。** 压缩过程中每遇到新字符时，进行如下操作：

1. **利用哈希函数计算该字符与紧跟其后的两个字符构成字符串的哈希地址**
2. **将该字符串中首字符在窗口中的索引插入上述计算出哈希位置的哈希桶中，返回插入之前该桶的状态**
3. **根据2返回的状态监测是否找到匹配串**
   - 如果当前桶为空，说明未找到匹配，
   - 否则：可能找到匹配，再定位到匹配串位置详细进行匹配即可。

利用哈希的思想，可大大提高查找匹配串的效率。

关于"哈希桶"，引发出一下问题：

1. **哈希桶的大小分析**

   三个字符总共可以组成2^{24}种取值，桶的个数需要2^{24}个，而索引大小占2个字节，总共桶占32M字节，是一个非常大的开销。随着窗口的移动，表中的数据会不断过时，维护这么大的表，会降低程序运行的效率。因此本文哈希桶的个数设置为：2^{15}(即32K)。

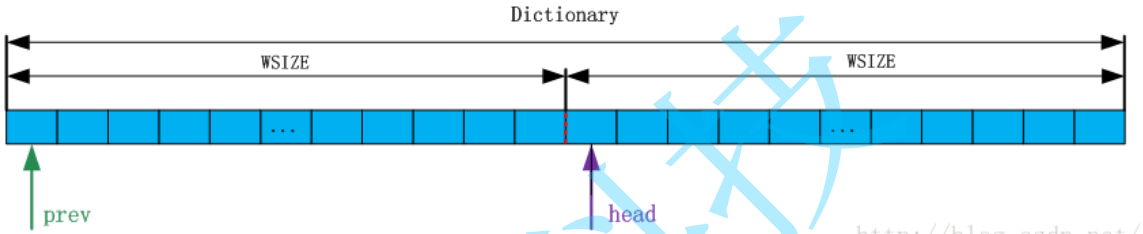
```
// 哈希桶的个数为2^15
const USH HASH_BITS = 15;

// 哈希表的大小
const USH HASH_SIZE = (1 << HASH_BITS);

// 哈希掩码：主要作用是将右窗数据往左窗搬移时，用来更新哈希表中数据，具体参见后文
const USH HASH_MASK = HASH_SIZE - 1;
```

2. **哈希表的结构**

原本需要2^{24}个哈希桶，现在减少为2^{15}个，必然会产生哈希冲突。如果采用开散列解决，链表中的节点要不断申请与释放，而且浪费空间，影响呈现效率。因此本文**哈希表由一整块连续的内存构成，分为两个部分，每部分大小为一个WSIZE**(32K)，如下图所示：



prev指向该字典整个内存的起始位置，head = prev + WSIZE，内存是连续的，所以prev和head可以看作两个数组，即prev[]和head[]。

head数组用来保存三个字符串首字符的索引位置，head的索引为三个字符通过哈希函数计算的哈希值。

而**prev就是来解决冲突的**，具体参见后文介绍。

3. **哈希函数**

哈希函数原则：简单、离散。因此本文哈希函数设计如下：

A(4,5) + A(6,7,8) ^ B(1,2,3) + B(4,5) + B(6,7,8) ^ C(1,2,3) + C(4,5,6,7,8)

说明：A 指 3 个字节中的第 1 个字节，B 指第 2 个字节，C 指第 3 个字节，

A(4,5) 指第一个字节的第 4,5 位二进制码，“^”是二进制位的异或操作，

“+”是“连接”而不是“加”，“^”优先于“+”)

这样使 3 个字节都尽量“参与”到最后的结果中来，而且每个结果值 h 都等于 ((前1个h << 5) ^ c)取右 15 位

```
// hashAddr：上一个字符串计算出的哈希地址
// ch：当前字符
// 本次的哈希地址是在前一次哈希地址基础上，再结合当前字符ch计算出来的
// HASH_MASK为WSIZE-1，&上掩码主要是为了防止哈希地址越界
void HashTable::HashFunc(USH& hashAddr, UCH ch)
{
    hashAddr = (((hashAddr) << H_SHIFT()) ^ (ch)) & HASH_MASK;
}

USH HashTable::H_SHIFT()
{
    return (HASH_BITS + MIN_MATCH - 1) / MIN_MATCH;
}
```

4. **哈希表构建(插入字符串)**

   哈希表的构建即将字符串插入到哈希表中，该过程伴随着压缩过程一块进行：

   - 获取当前字符ch(假设其在窗口中的位置为pos)
   - 用ch之后紧邻的两个字符构成当前串curStr
   - 插入curStr

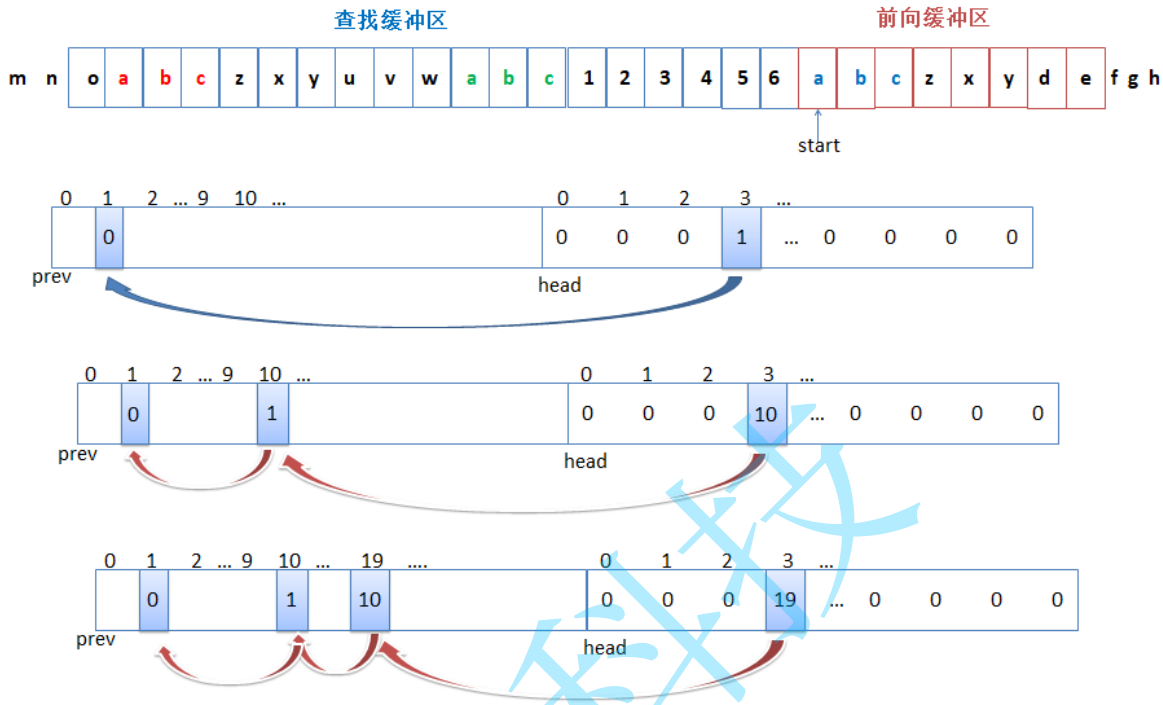
     ```
     // hashAddr：上一次哈希地址                ch：先行缓冲区第一个字符
     // pos：ch在滑动窗口中的位置              matchHead：如果匹配，保存匹配串的起始位置
     void HashTable::InsertString(USH& hashAddr, UCH ch, USH pos, USH& macthHead)
     {
         // 计算哈希地址
         HashFunc(hashAddr, ch);

         // 随着压缩的不断进行，pos肯定会大于WSIZE，与上WMASK保证不越界
         _prev[pos & WMASK] = _head[hashAddr];
         macthHead = _head[hashAddr];
         _head[hashAddr] = pos;
     }
     ```

   - matchHead带出匹配链的起始位置

比如：压缩当前进行到蓝色a的位置，此时：abc、bcz、czx、zxy...345、456已经插入到哈希表中，前向缓冲区中的第一个字符现在为a，用其后2个字符bc构成目标串，在查找缓冲区中找匹配串位置。

1. 计算“abc”的哈希地址hashAddr(假设为3)
2. 检测head[hashAddr]位置是否为空，空即没有找到匹配，说明”abc”字符串为未现过，否则将“abc”插入哈希表中；



通过matchHead判断是否发生匹配。

问题：当pos超过WSIZE时，在插入函数中如果直接使用pos肯定会越界，因此需要与上WMASK，即_prev[pos & WMASK] = _head[hashAddr]，但是该语句可能会破坏匹配链，让匹配链构成环而造成死循环，该情况如何处理?

设置一个最长匹配次数，比如：255，匹配了255次也没有匹配到，放弃本次匹配

5. **查找最长匹配**

字符串插入后，如果matchHead为空，表示为遇到匹配串，比如第一个"abc"的插入过程；否则，表示在查找缓冲区出现过该字符串。此时，顺着匹配链查找所有的匹配串，直到找到最长匹配。

```
// 功能：在当前匹配链中找最长匹配
// 参数：
//      hashHead：匹配链的起始位置
//      matchStart：最长匹配串在滑动窗口中的起始位置
// 返回值：最长匹配串的长度
USH BitZip::LongestMatch(USH hashHead, USH& matchStart)
{
    // 哈希链的最大遍历长度，防止造成死循环
    int chain_length = 256;

    // 始终保持滑动窗口为WSIZE，因为最小的超前查看窗口中有MIN_LOOKAHEAD的数据
    // 因此只搜索_start左边MAX_DIST范围内的串
    USH limit = _start > MAX_DIST ? _start - MAX_DIST : 0;


    // 待匹配字符串的最大位置
```

```
    // [pScan, strend]
    UCH* pScan = _pwin + _start;
    UCH* strend = pScan + MAX_MATCH - 1;

    // 本次链中的最佳匹配
    int bestLen = 0;
    UCH* pCurMatchStart;
    USH curMatchLen = 0;

    // 开始匹配
    do
    {
        // 从搜索区hashHead的位置开始匹配
        pCurMatchStart = _pwin + hashHead;
        while (pScan < strend && *pScan == *pCurMatchStart)
        {
            pScan++;
            pCurMatchStart++;
        }

        // 本次匹配的长度和匹配的起始位置
        curMatchLen = (MAX_MATCH - 1) - (int)(strend - pScan);
        pScan = strend - (MAX_MATCH - 1);

        /*更新最佳匹配的记录*/
        if (curMatchLen > bestLen)
        {
            matchStart = hashHead;
            bestLen = curMatchLen;
        }
    } while ((hashHead = _hash._prev[hashHead & WMASK]) > limit
        && --chain_length != 0);

    return curMatchLen;
}
```

通过上述方式获取到的最长匹配串，一定是最长的吗？如何优化？

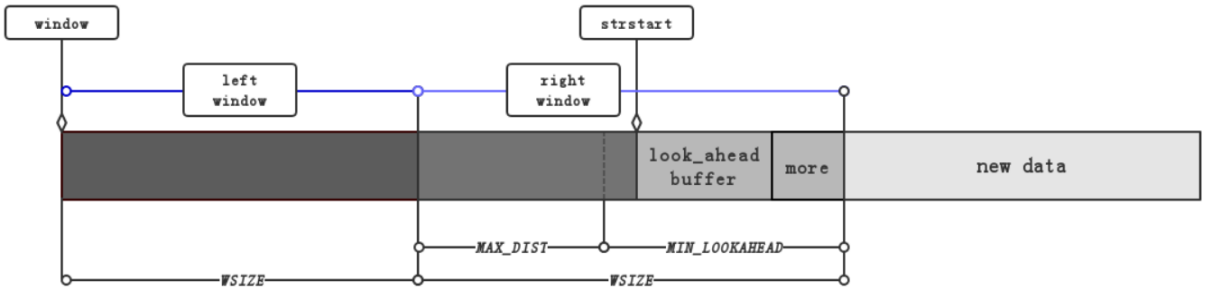
“1abc23bcdefghijklm456abcdefghijklmnopq”

2.3 找不到最长匹配怎么办

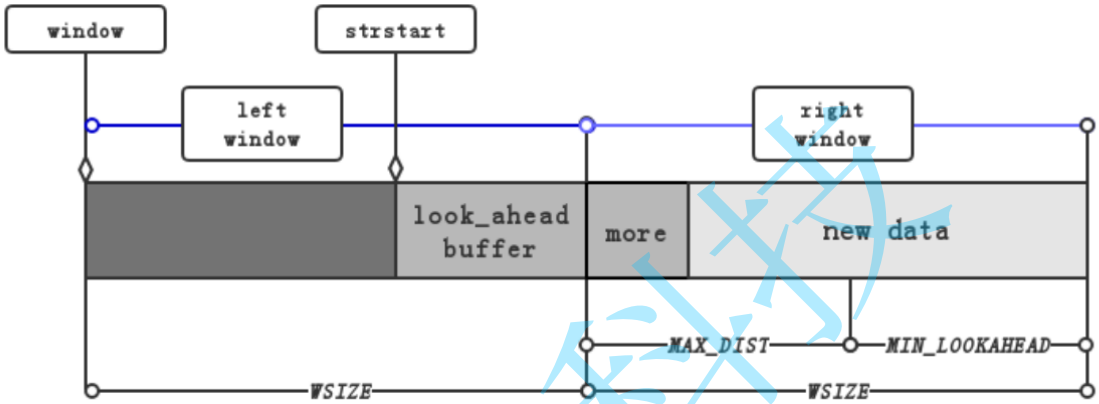
> **找不到最长匹配时，将该源字符直接写入压缩文件。**比如：比如：**mnoabczxyuvm(9,3)123456(18,6)defgh。**但是在真正压缩结果中，<距离，长度>对实际是没有括号的，因此上述的压缩结果实际为：**mnoabczxyuvm93123456186defgh**，如何区分<距离，长度>对与源文件中的数字？
>
> 为了区分源字符与<距离，长度>对，在向压缩文件中写数据时可用0和1来进行区分，比如用0代表源字符，1代表<距离，长度>对。
>
> 但真正的GZIP在保存压缩数据时，是将源字符和长度放在一块保存，将距离单独保存，为什么按照该种方式保存，后面再进行介绍。

2.4 滑动窗口中数据不够时怎么办?

随着滑动窗口的不断移动，右侧窗口中的数据不足MIN_LOOKAHEAD时怎么办？在压缩时，如果文件没有读到结尾，为了保证最大匹配，必须保持look_ahead中至少有MIN_LOOKAHEAD的源数据。



此时，需要将右窗中的数据搬移到左窗中。



注意：窗口中的数据移动，此时必须更新哈希表

```
void LZ77::FillWindow(FILE* fIn)
{
// 滑动窗口中的数据不足时
// 把右窗中数据(32K)移到左窗
if (_start >= WSIZE + MAX_DIST)
{
    memcpy(_pWin, _pWin + WSIZE, WSIZE);
    _start -= WSIZE;

    //更新哈希表，若是旧左窗的字串，则删除该词条，重置为nil，
    //注意，哈希表中越靠近头部的串，在窗口位置越靠右（就是更加新鲜），
    _ht.Update();
}

size_t readSize = 0;
if (!feof(fIn))
{
    readSize = fread(_pWin + _start + _lookAhead, 1, WSIZE, fIn);
    if (0 == readSize)
        memset(_pWin + _start + _lookAhead, 0, MIN_MATCH - 1);
    else
        _lookAhead += readSize;
}

}
```

```
void HashTable::UpdateDictionary()
{
    // 更新_head数组
for (int i = 0; i < HASH_SIZE; i++)
{
        if (_head[i] >= WSIZE)
            _head[i] -= WSIZE;
        else
        _head[i] = 0;
}

    // 更新prev数组
for (int i = 0; i < WSIZE; i++)
{
        if (_prev[i] >= WSIZE)
            _prev[i] -= WSIZE;
        else
            _prev[i] = 0;
}
}
```

3. LZ77压缩和解压缩原理介绍

3.1 压缩

上述在压缩中用到的理论知识介绍完成之后，就可以开始LZ77的压缩了，LZ77的压缩过程具体如下：

1. 打开带压缩的文件(注意：必须按照二进制格式打开，因为用户进行压缩的文件不确定)
2. 获取文件大小，如果文件大小小于3个字节，则不进行压缩
3. 读取一个窗口的数据，即64K
4. 用前两个字符计算第一个字符与其后两个字符构成字符串哈希地址的一部分，因为哈希地址是通过三个字节算出来的，先用前两个字节算出一部分，在压缩时，再结合第三个字节算出第一个字符串完整的哈希地址。
5. 循环开始压缩
    - 计算哈希地址，将该字符串首字符在窗口中的位置插入到哈希桶中，并返回该桶的状态matchHead
    - 根据matchHead检测是否找到匹配
        - 如果matchHead等于0，未找到匹配，表示该三个字符在前文中没有出现过，将该当前字符作为源字符写到压缩文件中
        - 如果matchHead不等于0，表示找到匹配，matchHead代表匹配链的首地址，从哈希桶matchHead位置开始找最长匹配，找到后用该(距离，长度对)替换该字符串写到压缩文件中，然后将该替换串三个字符一组添加到哈希表中。
6. 如果窗口中的数据小于MIN_LOOKAHEAD时，将右窗口中数据搬移到左窗口，从文件中新读取一个窗口的数据放置到右窗，更新哈希表，继续压缩，直到压缩结束。

3.2 压缩格式数据保存

压缩格式分三个文件保存：

> 1. 文件1保存比特标记位信息，用8个字节表示标记为长度，后面紧跟标记位
> 2. 文件2保存原字符和长度
> 3. 文件3保存所有的距离
>
> 为什么要按照该种方式保存，而不将所有的结果保存到一个文件中，后序再解释。

3.2 解压缩

> LZ77的解压缩非常简单：
>
> 1. 从文件1中读取标记，并对该标记进行分析
> 2. 如果当前标记是0，表示原字符，从文件2中读取一个字节，直接写到解压缩之后的文件中
> 3. 如果当前标记是1，表示遇到(距离，长度对)，从文件3中读取一个两个字节表示距离，从文件1中读取一个字节表示长度，构建(距离，长度)对，然后从解压缩过的结果中找出匹配长度
> 4. 获取下一个标记，直到所有的标记解析完。