

# 法律声明

□ 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

□ 课程详情请咨询

■ 微信公众号：小象

■ 新浪微博：ChinaHadoop



# 海量数据处理与系统设计

---



小象学院  
ChinaHadoop.cn

邹博

# 主要内容

---

- 倒排索引
  - POI
- Trie树
  - Darts
  - 统计回文对
- Bloom Filter
  - 如何降低错误率
- 跳跃表
  - 代码实现/思考：红黑树
- MD5
  - 应用？

# 例：寻找程序员

- A、B、C、D四人应聘一个程序员职位，此职务的要求条件是：Java熟练；懂数据库开发；会web开发；有C++经验。谁满足的条件最多，谁就被雇用。把这四个要求条件两两组合，每个组合都恰有一人满足。已知：
- A和B Java熟练
  - B和C会web
  - C和D懂数据库
  - D有C++经验
- 那么，被雇用的应该谁？

# 整理信息

	Java	数据库	Web	C++
A	✓			
B	✓		✓	
C		✓	✓	
D		✓		✓

□ 根据条件得到表1:

□ 同时得到两两组合:

- 条件(甲): Java & Web: B
- 条件(乙): 数据库 & Web: C
- 条件(丙): 数据库 & C++: D
- 条件(丁): Web & C++: 待定
- 条件(戊): Java & C++: 待定
- 条件(己): Java & 数据库: 待定

# 整理信息

	Java	数据库	Web	C++
A	✓		×(甲)	
B	✓	×(乙)	✓	
C	×(甲)	✓	✓	×(丙)
D		✓	×(甲)	✓

□ 分析各个条件组合，

□ 得到表2：

- 条件(甲)：Java & Web：B
- 条件(乙)：数据库 & Web：C
- 条件(丙)：数据库 & C++：D
- 条件(丁)：Web & C++：待定
- 条件(戊)：Java & C++：待定
- 条件(己)：Java & 数据库：待定

## 继续分析表2

	Java	数据库	Web	C++
A	✓		×(甲)	
B	✓	×(乙)	✓	✓
C	×(甲)	✓	✓	×(丙)
D		✓	×(甲)	✓

- 通过表2可知，
- Web 只有B和C两人掌握，
- 因此，“条件(丁)：Web & C++”只能在这二人中产生，而C不会C++，所以，条件(丁)由B掌握。
- B已经掌握了Java，从而，“条件(戊)：Java & C++”也是B掌握。
- D不能会Java，否则，D将与条件(戊)矛盾；
- A不能会C++，否则，A将与条件(戊)矛盾。
- 这时，B不会数据库，C、D不会Java，
- 所以，条件(己)只能有A掌握。

# 最终结果

□ 可以看到，B掌握了3项技术，而其他只掌握了两项技术。因此，被雇佣的应该是B。

	Java	数据库	Web	C++
A	✓	✓	✗	✗
B	✓	✗	✓	✓
C	✗	✓	✓	✗
D	✗	✓	✗	✓



# 附：谁是情种？

□ 李逵、宝玉、西门、纳兰四才子穿越到女儿国，结识了西施、貂蝉、昭君、玉环。国王武氏将选择一个情种做贴身侍卫。武氏发现，任选两美人，恰只有一位才子喜欢他们俩。狄仁杰已探得：

■ 李逵、宝玉喜欢西施

■ 西门、纳兰喜欢貂蝉

■ 宝玉、西门喜欢昭君

■ 纳兰喜欢玉环

□ 未知情报需要你来帮助狄仁杰推断，请问，谁能够成为武氏的贴身侍卫？

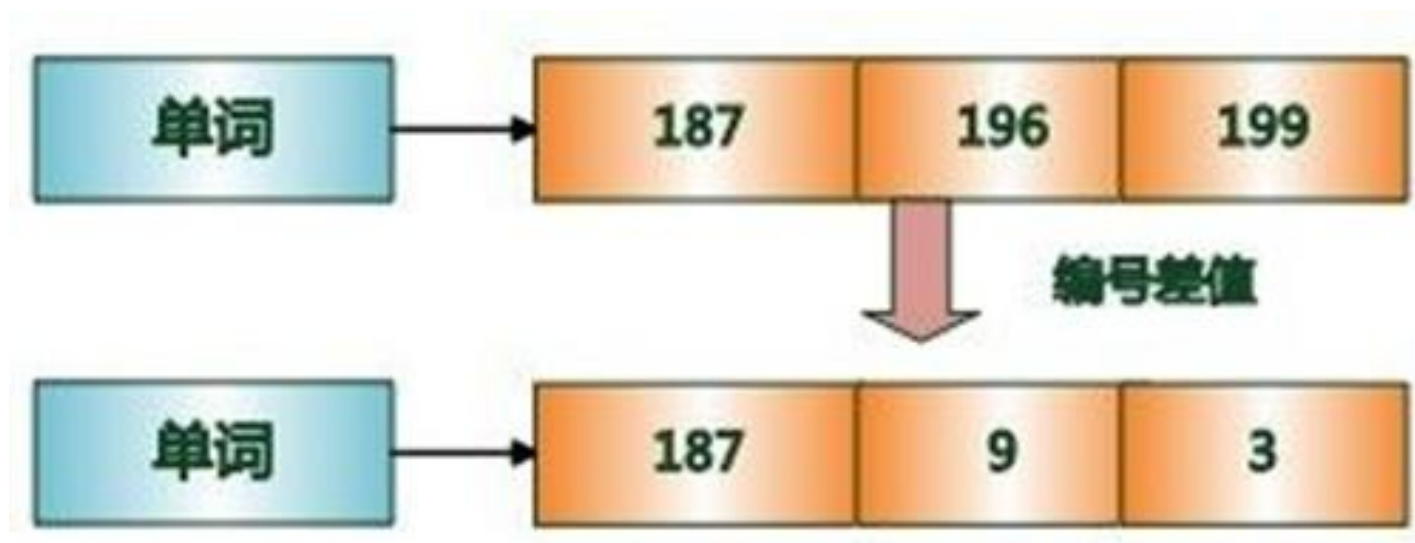
■ 不知情种是何物者，先面壁七十二时辰！

# 倒排索引

---

- 倒排索引的索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。
- 因为不是由记录来确定属性值，而是由属性来确定记录，因而称为倒排索引(inverted index)。
- 带有倒排索引的文件称为倒排索引文件，简称倒排文件(inverted file)。

# 倒排列表



# POI

- 跳跃链表、跳跃表、跳表；
- POI(Point of Interest)查询
  - 部分匹配：小象学院，简称小象
  - 跳跃匹配：中国科学院、中科院

网易公开课 搜索课程、视频、策划

欢迎来到网易公开课!

跳表

找到如下“跳表”相关内容



没有满足条件的内容

网易公开课 搜索课程、视频、策划

欢迎来到网易公开课! 登录注册

跳跃表

找到如下“跳跃表”相关内容

视频(1)



**跳跃表**

《算法导论》第12课里我们将学到一种简单而又十分有趣的动态搜索数据结构——**跳跃表**。这种数据结构的优势在于它易于实现，而且很好地保证了它总是能高效运作。教授通过纽约地铁

# POI信息点搜索总框架

---

```
void CFileObject::Search(LPCTSTR lpszContent, int nIndex)
{
    if(!lpszContent || !lpszContent[0])
        return;
    CreateSearchTree(nIndex);

    SearchFuzzy(lpszContent, nIndex);
}
```

# 建立查找树

```
bool CFileObject::CreateSearchTree(int nIndex)
{
    int nSymbolType = VerdictType();
    vector<CDataType*>* pAF = GetAF(nSymbolType);
    if(!pAF)
        return false;
    int size = (int)pAF->size();
    if((nIndex < 0) || (nIndex >= size))
        return false;

    if(IsSearchIndexValid(nIndex))
        return true;

    CSearchIndex* pSI = GetSearchIndex(nIndex);
    DWORD dwStart = GetTickCount();
    switch(m_iSymbolType)
    {
        case FO_SYMBOL:
        {
            CreateST(pSI, nIndex, m_vecAcnode);
            break;
        }
        case FO_ROUTE:
        case FO_THREAD:
        {
            CreateST(pSI, nIndex, m_vecRoute);
            break;
        }
        case FO_REGION:
        {
            CreateST(pSI, nIndex, m_vecTopology);
            break;
        }
    }
    return true;
}
```

# 建立查找树

```
bool CFileObject::CreateST(CSearchIndex* pSI, int nIndex, vector<CStamp*>& vecSymbol)
{
    vector<CStamp*>::iterator itEnd = vecSymbol.end();
    CStamp* pSymbol = NULL;
    CSimpleData* pData = NULL;
    LPCTSTR lpszString;
    for(vector<CStamp*>::iterator it = vecSymbol.begin(); it != itEnd; it++)
    {
        pSymbol = *it;
        if(pSymbol && !pSymbol->IsDelete())
        {
            pData = pSymbol->GetAttribute(nIndex);
            if(pData)
            {
                lpszString = pData->GetString();
                pSymbol->SetData(0);
                pSI->AddSymbol(lpszString, pSymbol);
            }
        }
    }

    pSI->SetValid(true);
    return true;
}
```

# 处理Hash冲突

```
bool CSearchIndex::AddSymbol(int nIndex, CStamp* pSymbol)
{
    ASSERT(nIndex >= 0);
    ASSERT(nIndex < SI_COUNT);
    if((nIndex < 0) || (nIndex >= SI_COUNT))
        return false;
    if(!m_dtSI[nIndex])
    {
        m_dtSI[nIndex] = new CBalanceTree<CStamp*>;
    }
    bool bInsert = m_dtSI[nIndex]->Insert(pSymbol);
    return bInsert;
}
```



# Hash查找

```
bool CSearchIndex::Search(int nIndex, CBalanceTree<CStamp*>& avlTree)
{
    ASSERT(nIndex >= 0);
    ASSERT(nIndex < SI_COUNT);
    if((nIndex < 0) || (nIndex >= SI_COUNT))
        return false;
    CBalanceTree<CStamp*>* pTreeSymbol = m_dtSI[nIndex];
    if(!pTreeSymbol)
        return false;
    pTreeSymbol->InOrder(SearchSetData, &avlTree);
    return true;
}
```

# 该复合结构可用性分析

---

- 假定POI总数为100万，每个POI平均字数为10个，那么，问题总规模为1000万；
- 假定常用汉字为1万个，那么，Hash之后，1万个汉字对应的槽slot平均含有1000个POI信息；
- $\log 1000 = 9.9658$ ：即，将1000万次搜索，降到10次搜索。
  - 注：以上只是定性考虑，非准确分析

# Trie树

---

- Trie树是一种哈希多叉树，又称字典树、单词查找树或前缀树，用于在大量字符串中快速检索。
  - 英文字母的字典树是一个26叉树
  - 数字的字典树是一个10叉树。

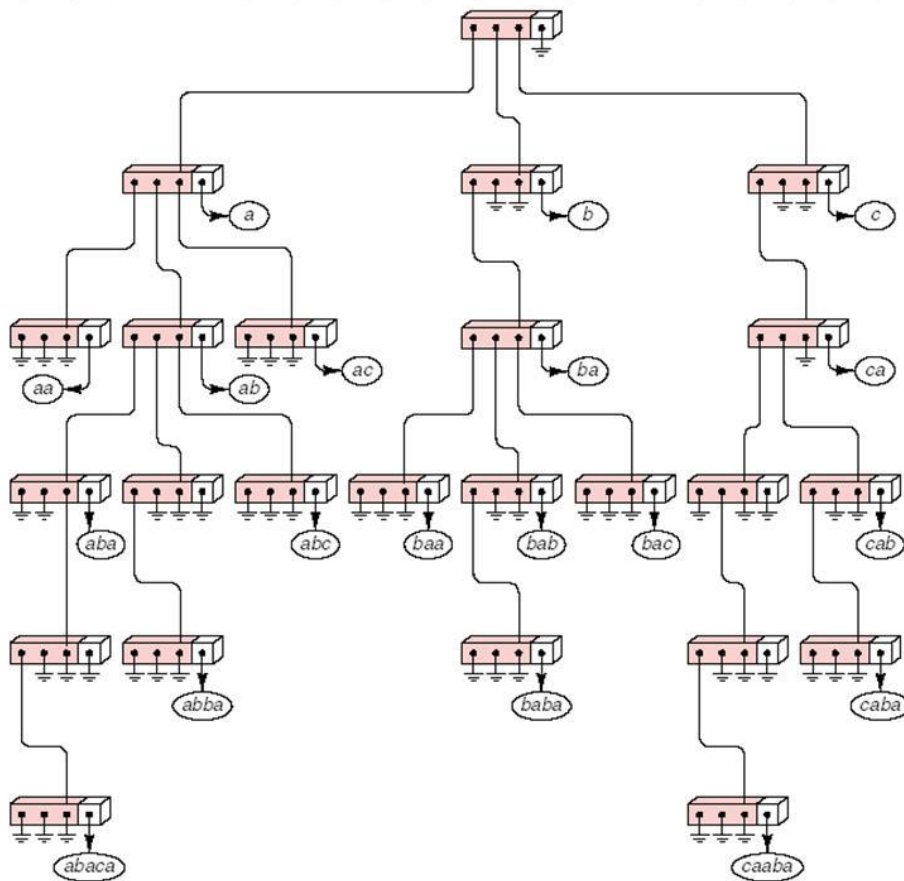
# Trie数的特点及性质

- 典型应用：统计和排序大量的字符串(但不仅限于字符串)，所以经常被文字处理系统用于文本词频统计。
  - 优点：利用字符串的公共前缀来节约存储空间,最大限度地减少无谓的字符串比较，查询效率比哈希表高。
  - 缺点：如果存在大量字符串且这些字符串基本没有公共前缀，则相应的Trie树将非常消耗内存。
- 三个基本性质：
  - 根结点不包含字符，除根结点外每一个结点都只包含一个字符。
  - 从根结点到某一结点，路径上经过的字符连接起来，为该结点对应的字符串。
  - 每个结点的所有子结点包含的字符都不相同。

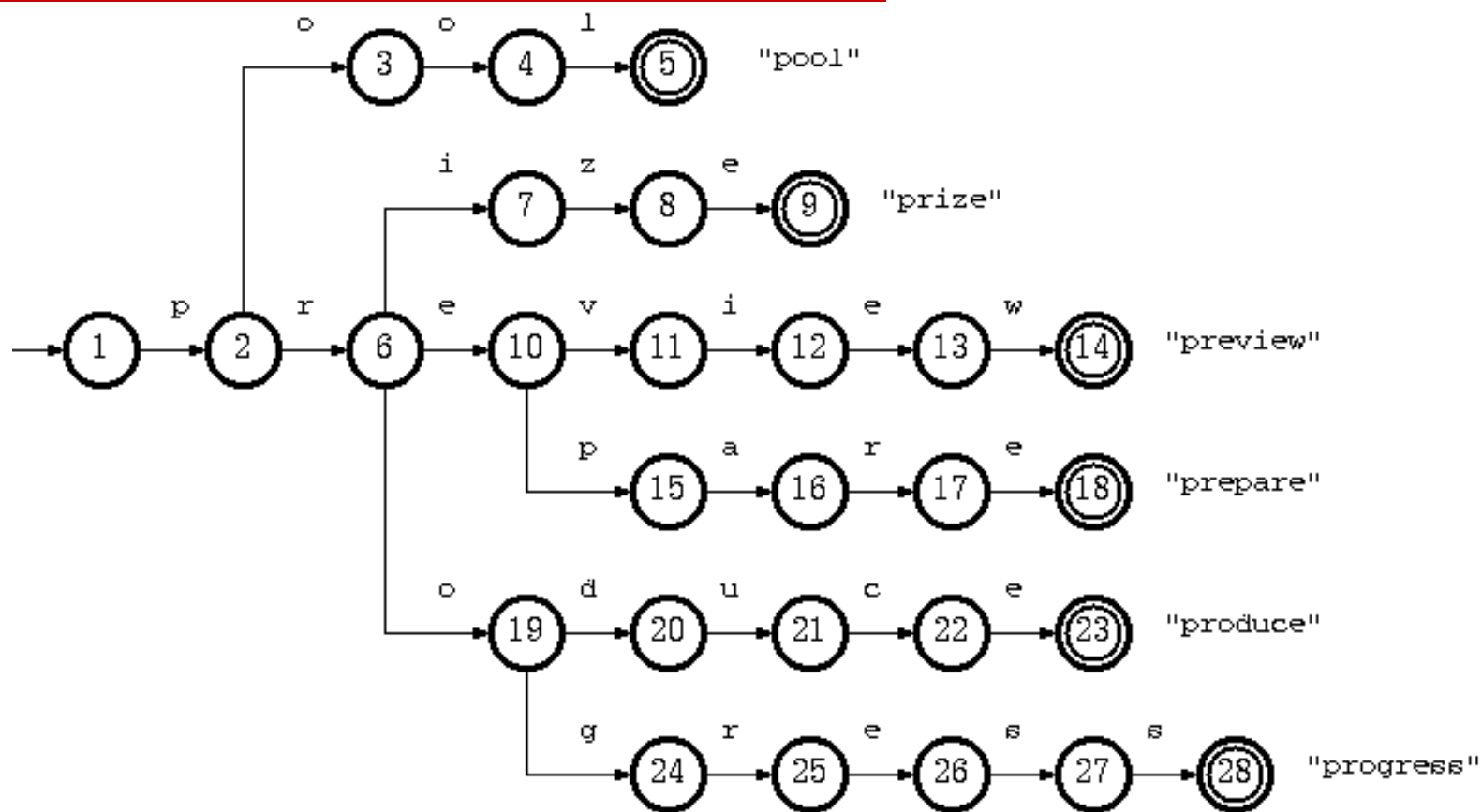
# Trie树举例

□ 词典:

□ a、b、c、aa、ab、ac、ba、ca、aba、abc、baa、bab、bac、cab、abba、baba、caba、abaca、caaba



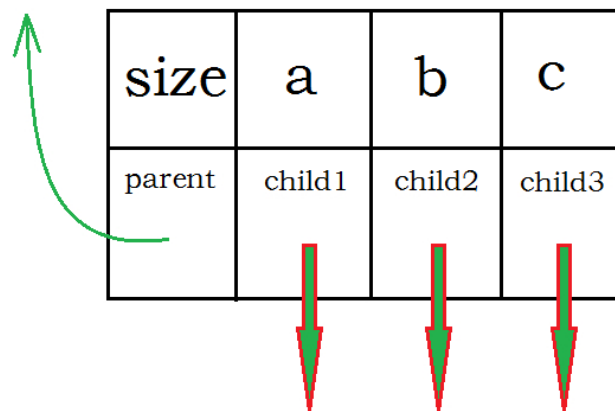
# Trie树举例



\_\_\_\_\_

# 对左孩子-右兄弟表示法的思考

□ 只记录有效索引指针，形成如下结构

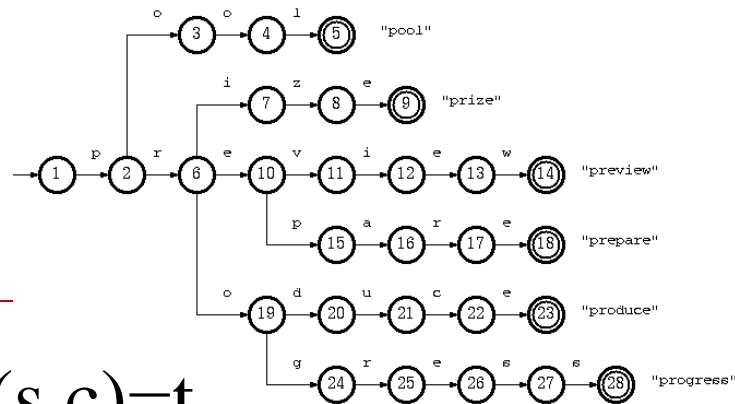


□ 利用“左孩子-右兄弟”表示方法，如何将一颗树转换成二叉树？

■ 如何将N颗树(森林)转换成二叉树？

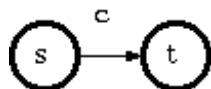


# 空间复杂度的简化

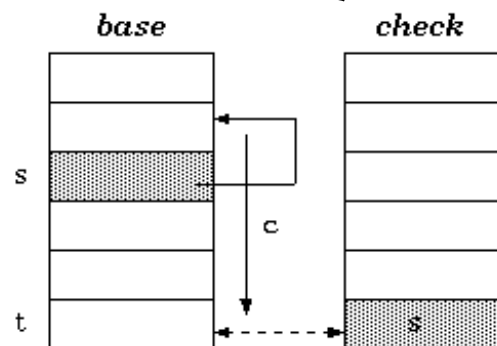


- Trie树的状态转移函数:  $g(s,c)=t$
- $s$ 表示当前状态,  $c$ 表示转移条件,  $t$ 表示下一个可接受状态: (DFA的状态转移)
- base数组中的每一个元素相当于Trie树的一个节点, 其值做状态转移的基值, check数组相当于校验值, 用于检查该状态是否存在。

■  $base[s]+c=t$



■  $check[t]=s$

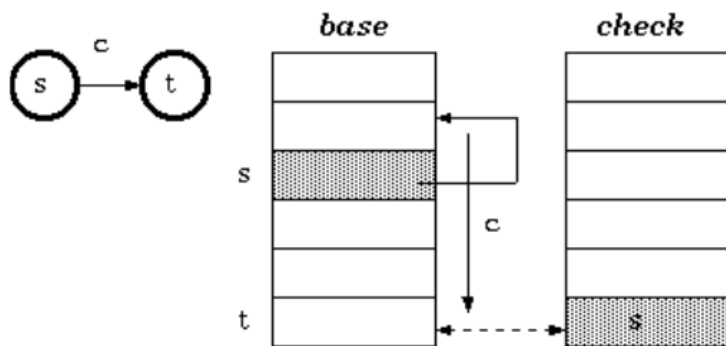


# Trie树的双数组表示法

下标	1	2	3	4	5	6	7	8	9	10	11
base	-1	1	1	0	1	-6	1	-8	-9	-1	-11
check	0	0	0	0	2	2	2	3	5	7	10
状态	啊	阿	埃		阿根	阿胶	阿拉	埃及	阿根廷	阿拉伯	阿拉伯人

编码(C):            啊-1, 阿-2, 埃-3, 根-4, 胶-5, 拉-6, 及-7, 廷-8, 伯-9, 人-10

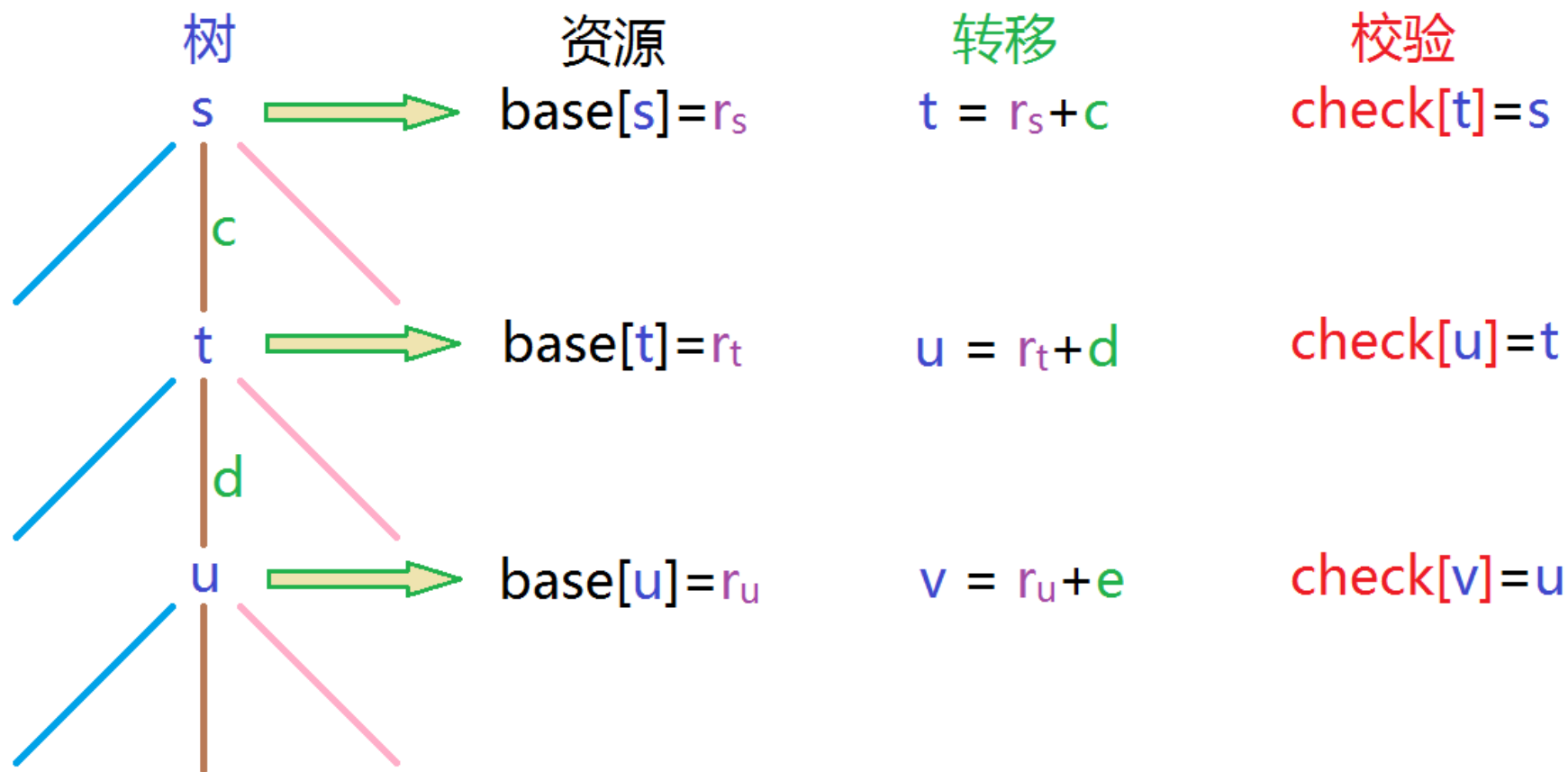
词表:                啊, 阿根廷, 阿胶, 阿拉伯, 阿拉伯人, 埃及



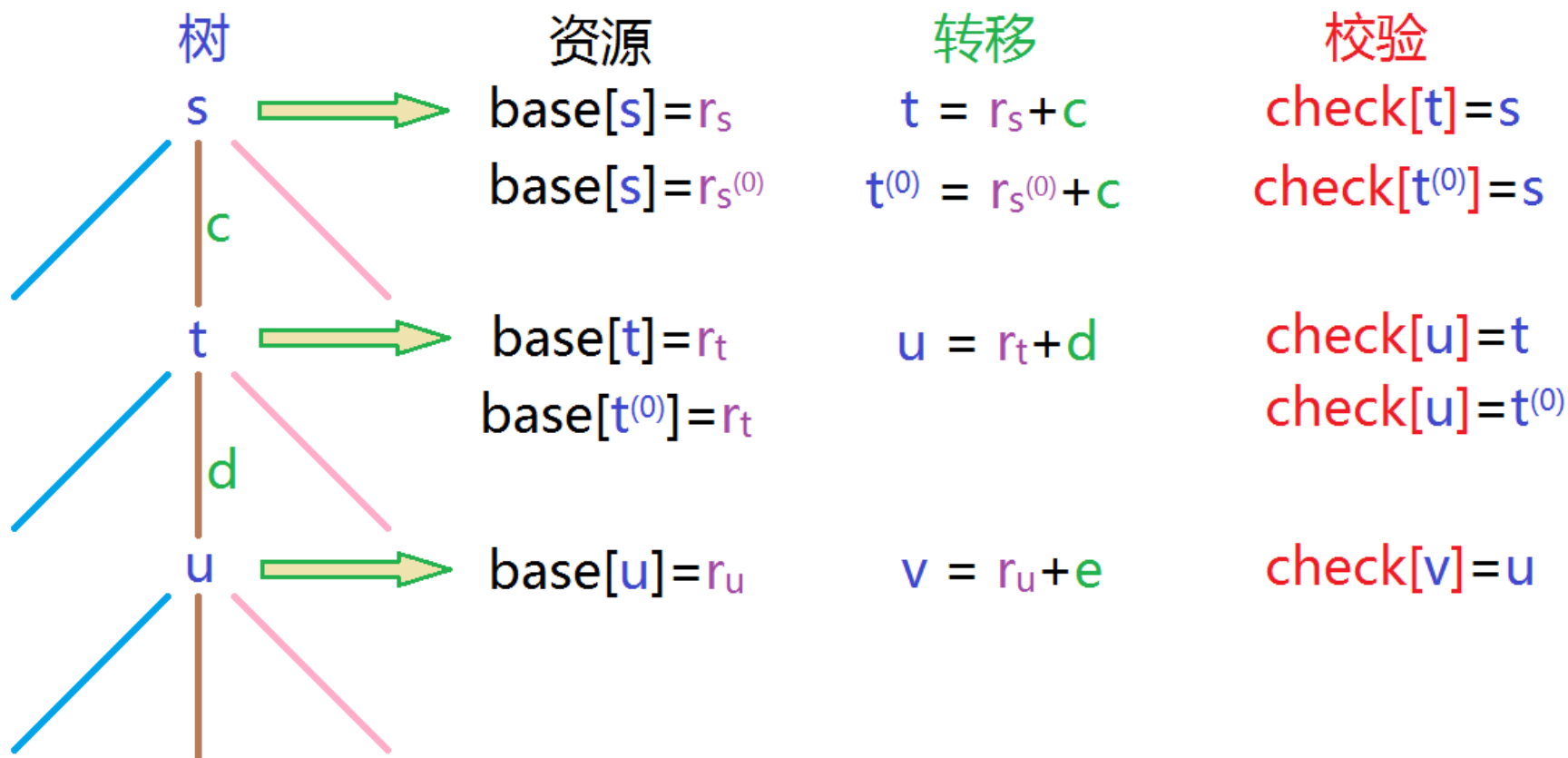
$$\text{base}[s] + c = t$$

$$\text{check}[t] = s$$

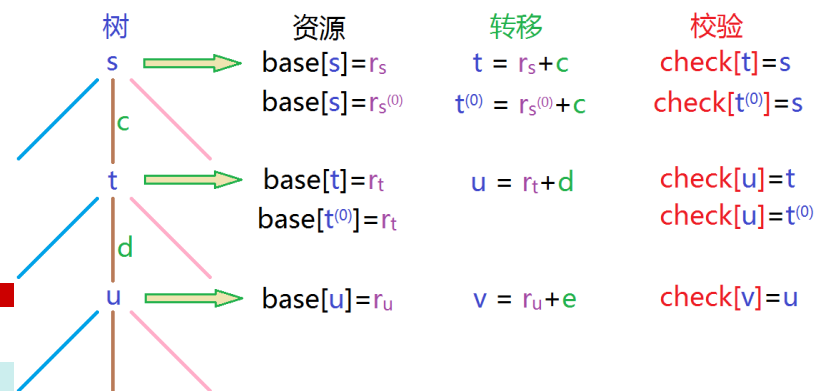
# Trie树逻辑结构



# Trie树的资源号修改

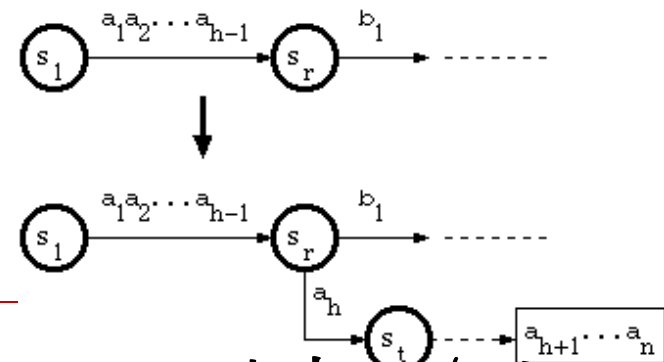


# 资源号修改伪代码



```

Procedure Relocate(s : state; b : base_index)
{ Move base for state s to a new place beginning at b }
begin
  foreach input character c for the state s
  { i.e. foreach c such that  $check[base[s] + c] = s$  }
  begin
     $check[b + c] := s;$       { mark owner }
     $base[b + c] := base[base[s] + c];$     { copy data }
    { the node  $base[s] + c$  is to be moved to  $b + c$ ;
      Hence, for any i for which  $check[i] = base[s] + c$ , update  $check[i]$  to  $b + c$  }
    foreach input character d for the node  $base[s] + c$ 
    begin
       $check[base[base[s] + c] + d] := b + c$ 
    end;
     $check[base[s] + c] := none$       { free the cell }
  end;
   $base[s] := b$ 
end
  
```



# 字符串的插入

□ 记待插入字符串为  $a_1a_2\ldots a_{h-1}a_ha_{h+1}\ldots a_n$ ，其中，在当前Trie树中找到了前缀  $a_1a_2\ldots a_{h-1}$ ，而  $a_h$  是第一个未找到的字符，记Trie树中字符  $a_{h-1}$  转移得到的结点为  $s_r$ 。

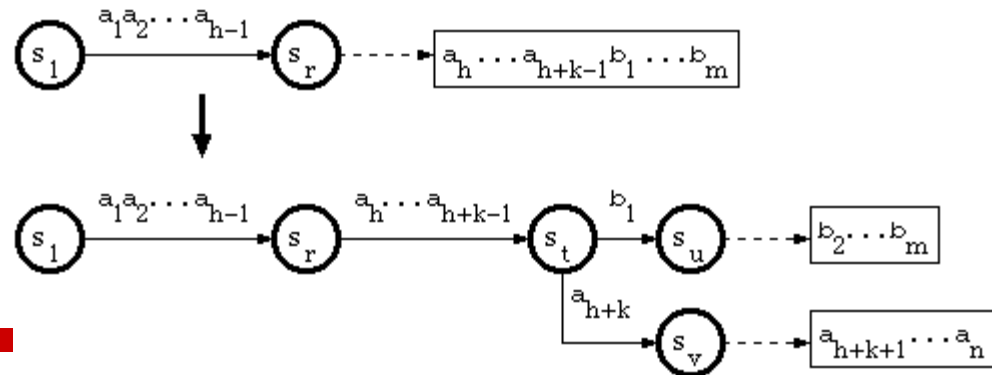
□ 算法：

■ 以  $s_r$  为根，插入新结点  $s_t$ ：  $s_t$  由  $a_h$  转移得到

■ 将  $s_t$  指向 **后缀**  $a_{h+1}\ldots a_n$

□ 这里，将后缀串  $a_{h+1}\ldots a_n$  直接加入后缀池，减少不必要的分支，进一步降低空间复杂度。

■ 由于 **后缀池** 的引入，如果插入的字符串最后查找成功的字符没有分支，而是在后缀池中，如何操作？



# 字符串的插入

□ 记待插入字符串为  $a_1 a_2 \dots a_{h-1} a_h \dots a_{h+k-1} a_{h+k} \dots a_n$ ，其中，在当前Trie树中找到了前缀  $a_1 a_2 \dots a_{h-1}$ ，而  $a_h \dots a_{h+k-1} b_1 \dots b_m$  是在后缀池中，记Trie树中最后一个分支结点为  $s_r$ ，即字符  $a_{h-1}$  转移得到  $s_r$ 。

□ 算法：

- 以  $s_r$  为根，插入新结点  $s_t$ ：  $s_t$  由  $a_h \dots a_{h+k-1}$  转移得到
- 以  $s_t$  为根，插入新结点  $s_u$ ：  $s_u$  由  $b_1$  转移得到
- 将  $s_u$  指向后缀池中的  $b_2 \dots b_m$
- 以  $s_t$  为根，插入新结点  $s_v$ ：  $s_v$  由  $a_{h+k}$  转移得到
- 将  $s_v$  指向后缀池中的  $a_{h+k+1} \dots a_n$

# 双数组Trie树结构总结

- Trie树逻辑结构清晰简练，优雅自然，在海量数据中查找某数据，和海量数据规模无关，只和待查找数据长度本身有关，时间复杂度为 $O(\text{len})$ ，常常可以认为是 $O(1)$ 。
  - 可以看做是以数据元素为关键字的多Hash结构；
  - 海量数据的复杂度分析未考虑内存调度等问题。
- 双数组的存储结构晦涩难懂，增删困难。实践中，往往离线将海量数据建立Trie树双数组结构，少量删除时可以继续使用。若大量删除，则离线建立新的Trie树双数组结构，适时替换。



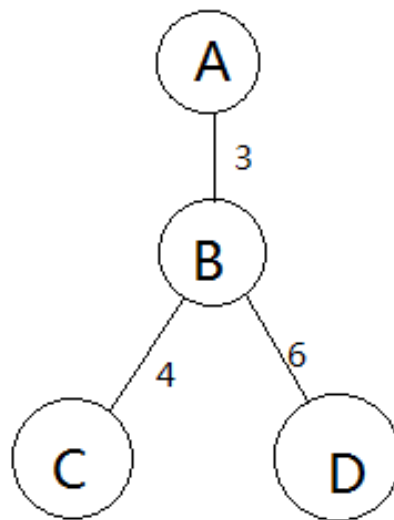
# 思考：Trie树应用

□ 给定一颗边的权值都是正整数的树，求某两个结点间的路径S，使得该路径所包含的所有边权的异或值最大。

■ A-B-C  $3^4=7$ (最优解)

■ A-B-D  $3^6=5$

■ C-B-D  $4^6=2$



# 回文对 Palindrome Pairs

- 给定若干单词组成的词典words，找到词典中的所有单词对(i,j)，使得words[i]+words[j]是回文串。
- 如：给定词典words=["abcd", "dcba", "lls", "s", "ssll", "sss"], 则应返回(0,1),(1,0),(3,2),(2,4)，即回文串是："dcbaabcd", "abcd dcba", "slls", "lls sll", "ssss", "ssll sss", "ssss"。

# 回文问题分析

- 若str1与str2可以构成回文串，则
  - 要么str1的全部与str2的后缀构成回文串，且str2的前缀本身是回文串。如：str1="lls", str2="sssl";
  - 要么str1的前缀与str2的全部构成回文串，且str1的后缀本身是回文串。如：str1="sssl", str2="sss";
  - 若str1的全部和str2的全部构成回文串，因空串是回文串，因此可看成以上两者的特殊情况。

# 算法思路

---

- 将词典中的所有词逆序存放到Trie树中。
- 枚举词典中的所有词，若某词str1与Trie树中的单词str2发生部分匹配：
  - 若str1部分剩余，则计算str1的剩余部分是否是回文串；
  - 若str1完全匹配，则计算以str1匹配结束的Trie树结点为根的子树中是否存在回文串，若存在，则str1与其可以构成回文串。
- 总时间复杂度为 $O(N \cdot \text{len})$ ，其中N为单词数目，len为单词长度。

# Code

```
class CTrieTree
{
private:
    CTrieNode* m_pRoot;           //树根
    const vector<string>& m_words; //词典
    list<pair<int, int> > m_pair;  //回文对

    void Destroy(CTrieNode* m_pRoot);
    void InsertR(int nIndex);      //将m_words[nIndex]逆序插入Tree
    void CalcPair(int nIndex);     //计算m_words[nIndex]是否与Tree中的字符串构成回文串
    void Palindrome(CTrieNode* node, string& str, int nIndex); //从node开始的字符串是否是回文串
    bool IsPalindrome(const string& str) const; //判断str是否是回文串
public:
    CTrieTree(const vector<string>& words);
    ~CTrieTree() { Destroy(m_pRoot); }
    void PalindromePair();
    void PrintPair();
};

class CTrieNode
{
public:
    int nIndex; //原字符串在字典中的位置
    map<char, CTrieNode*> child;
public:
    CTrieNode() : nIndex(-1) {}
};
```

# Code

```
CTrieTree::CTrieTree(const vector<string>& words) : m_words(words)
{
    m_pRoot = new CTrieNode();
    for(int i = 0; i < (int)words.size(); i++)
        InsertR(i);
}

void CTrieTree::InsertR(int nIndex)
{
    CTrieNode* node = m_pRoot;
    const string& str = m_words[nIndex];
    for(auto it = str.rbegin(); it != str.rend(); it++)
    {
        if(!node->child[*it])
            node->child[*it] = new CTrieNode();
        node = node->child[*it];
    }
    node->nIndex = nIndex;
}
```

# Code

```
void CTrieTree::Palindrome(CTrieNode* node, string& str, int nIndex)
{
    if(!node)
        return;
    if((node->nIndex != -1) && (node->nIndex != nIndex)) //node是单词且非当前词
    {
        if(IsPalindrome(str))
            m_pair.push_back(make_pair(nIndex, node->nIndex));
    }
    for(auto c = node->child.begin(); c != node->child.end(); c++)
    {
        str.push_back(c->first);
        Palindrome(c->second, str, nIndex);
        str.pop_back();
    }
}

void CTrieTree::CalcPair(int nIndex)
{
    CTrieNode* node = m_pRoot;
    const string& str = m_words[nIndex];
    for(auto p = str.begin(); p != str.end(); p++)
    {
        if(!node)
            return;
        if((node->nIndex != -1) && IsPalindrome(string(p, str.end())))
            m_pair.push_back(make_pair(nIndex, node->nIndex));
        node = node->child[*p];
    }
    string s;
    Palindrome(node, s, nIndex);    //以node为根的所有的回文串都是解
}
```

# Code

(0, 1)	abcd	dcba
(1, 0)	dcba	abcd
(2, 4)	lls	sssl
(3, 2)	s	lls
(3, 5)	s	sss
(4, 5)	sssl	sss
(5, 3)	sss	s

```
void CTrieTree::PalindromePair()
{
    for(int i = 0; i < (int)m_words.size(); i++)
        CalcPair(i);
}

int _tmain(int argc, _TCHAR* argv[])
{
    vector<string> words(6);
    words[0] = "abcd";
    words[1] = "dcba";
    words[2] = "lls";
    words[3] = "s";
    words[4] = "sssl";
    words[5] = "sss";
    CTrieTree t(words);
    t.PalindromePair();
    t.PrintPair();
    return 0;
}
```



# 算法思路2

## □ 枚举词典中的所有词：

- str分成 $\langle \text{left} - \text{right} \rangle$ 两部分，若 $\text{left}$ 是回文串，且 $\text{right}$ 的翻转串 $T$ 在词典中，则 $\langle T - \text{left} - \text{right} \rangle$ 构成回文串。
- str分成 $\langle \text{left} - \text{right} \rangle$ 两部分，若 $\text{right}$ 是回文串，且 $\text{left}$ 的翻转串 $T$ 在词典中，则 $\langle \text{left} - \text{right} - T \rangle$ 构成回文串。
- 若 $\text{str}$ 是回文串，且 $\text{str}$ 的翻转串 $T$ 在词典中，则 $\langle T - \text{str} \rangle$ 构成回文串。若认为 $\text{str}$ 的前缀串可为空，则该分支可隶属于第1种情况。

## □ 计算字符串的前后缀是否回文，可使用Manacher算法，时间复杂度仅为 $O(\text{len})$ 。

## □ 该问题总时间复杂度为 $O(N * \text{len})$ ，其中 $N$ 为单词数目， $\text{len}$ 为单词长度；与Trie树的解决方案相同，但代码容易实现。

# Manacher

---

```
void Manacher(const char* str, list<int>& prefix, list<int>& suffix)
{
    int size = strlen(str);
    int N = 2*size+1;
    int* p = new int[N];
    p[0] = 1;
    int mx = 1;
    int id = 0;
    int i;
    for(i = 1; i < N; i++)
    {
        if(i < mx)
            p[i] = min(p[2*id-i], mx-i);
        else
            p[i] = 1;
        while(((i != p[i]) && (((i+p[i]) % 2 == 1)
            || (str[(i+p[i])/2-1] == str[(i-p[i])/2-1])))
            p[i]++;
        if(mx < i + p[i])
        {
            mx = i + p[i];
            id = i;
        }
    }
    for(i = 2; i < N; i++)
    {
        if(p[i] == i)
            prefix.push_back(i-1);
        if(i+p[i] == N+1)
            suffix.push_back(p[i]-1);
    }
    delete[] p;
}
```

# Code

(0, 1)	abcd	dcba
(1, 0)	dcba	abcd
(3, 2)	s	lls
(2, 4)	lls	sssl
(4, 5)	sssl	sss
(3, 5)	s	sss
(5, 3)	sss	s

```
void PalindromePair(const vector<string>& words, list<pair<int, int> >& result)
{
    hash_map<string, int> dict;
    int size = (int)words.size();
    int i, s;
    for(i = 0; i < size; i++)
        dict[words[i]] = i;
    list<int> prefix, suffix;
    string word, str;
    for(i = 0; i < size; i++)
    {
        prefix.clear();
        suffix.clear();
        word = words[i].c_str();
        s = (int)words[i].size();
        Manacher(word.c_str(), prefix, suffix); //计算word的前缀/后缀的回文串
        for(auto p = prefix.begin(); p != prefix.end(); p++) //前缀串
        {
            str = words[i].substr(*p, s - *p);
            reverse(str.begin(), str.end());
            if(dict.find(str) != dict.end())
            {
                result.push_back(make_pair(dict[str], i));
            }
        }
        for(auto p = suffix.begin(); p != suffix.end(); p++) //后缀串
        {
            str = words[i].substr(0, s - *p);
            reverse(str.begin(), str.end());
            if(dict.find(str) != dict.end())
            {
                result.push_back(make_pair(i, dict[str]));
            }
        }
        reverse(word.begin(), word.end()); //全串
        if(dict.find(word) != dict.end())
        {
            if(dict[word] != i)
                result.push_back(make_pair(i, dict[word]));
        }
    }
}
```

# Bloom Filter

---

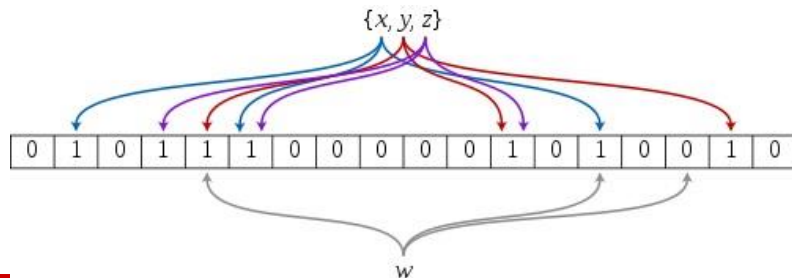
- ❑ 布隆过滤器(Bloom Filter)是由Burton Howard Bloom于1970年提出的,它是一种空间高效(space efficient)的概率型数据结构,用于判断一个元素是否在集合中。在垃圾邮件过滤的黑白名单、爬虫(Crawler)的网址判重等问题中经常被用到。
- ❑ 哈希表也能用于判断元素是否在集合中,但是Bloom Filter只需要哈希表的1/8或1/4的空间复杂度就能完成同样的问题。Bloom Filter可以插入元素,但不可以删除已有元素。集合中的元素越多,误报率(false positive rate)越大,但是不会漏报(false negative)。

# Bloom Filter

---

- 如果想判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过对比来判定是否在集合内：链表、树等数据结构都是这种思路。但是随着集合中元素数目的增加，我们需要的存储空间越来越大，检索速度也越来越慢( $O(n)$ ,  $O(\log n)$ )。
- 可以利用Bitmap：只要检查相应点是不是1就知道集合中有没有某个数。这就是Bloom Filter的基本思想。

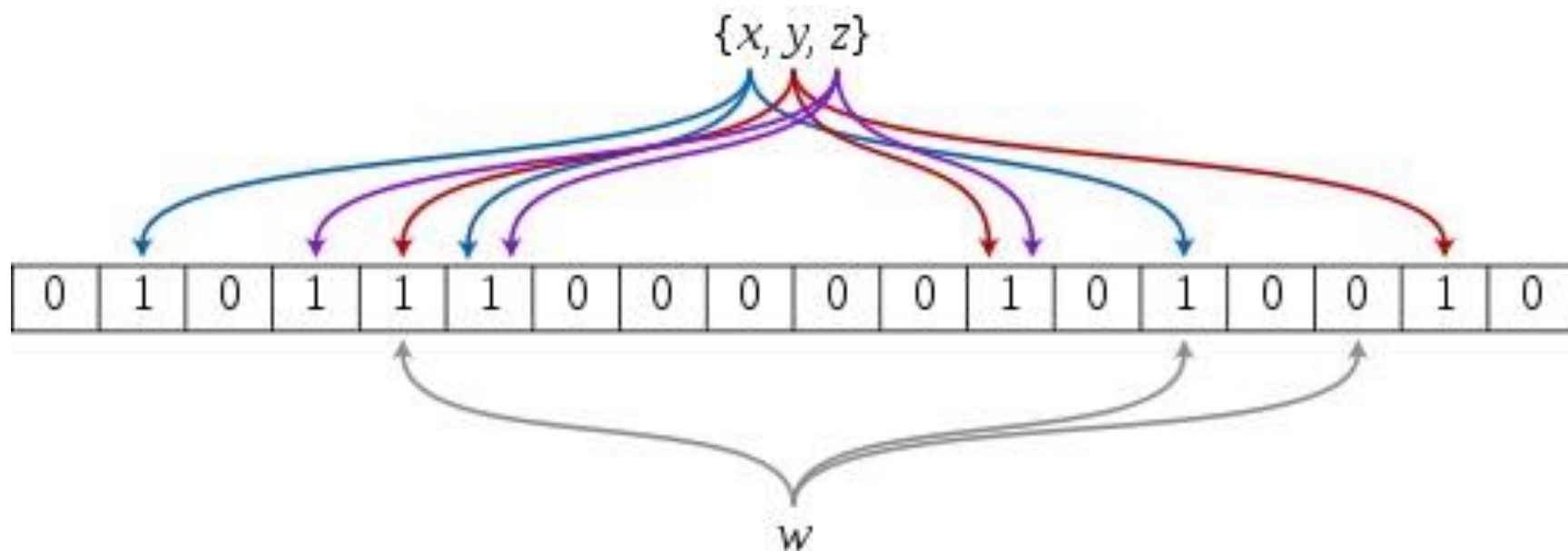
# Bloom Filter算法描述



- 一个空的Bloom Filter是一个有 $m$ 位的位向量 $B$ ，每一个bit位都初始化为0。同时，定义 $k$ 个不同的Hash函数，每个Hash函数都将元素映射到 $m$ 个不同位置中的一个。
  - 记： $n$ 为元素数， $m$ 为位向量 $B$ 的长度(位：槽 slot)， $k$ 为Hash函数的个数。
- 增加元素 $x$ 
  - 计算 $k$ 个Hash( $x$ )的值( $h_1, h_2 \dots h_k$ )，将位向量 $B$ 的相应槽 $B[h_1, h_2 \dots h_k]$ 都设置为1；
- 查询元素 $x$ 
  - 即判断 $x$ 是否在集合中，计算 $k$ 个Hash( $x$ )的值( $h_1, h_2 \dots h_k$ )。若 $B[h_1, h_2 \dots h_k]$ 全为1，则 $x$ 在集合中；若其中任一位不为1，则 $x$ 不在集合中；
- 删除元素 $x$ 
  - 不允许删除！因为删除会把相应的 $k$ 个槽置为0，而其中很有可能其他元素对应的位。

# Bloom Filter 插入查找数据

- ❑ 插入  $x, y, z$
- ❑ 判断  $w$  是否在该数据集中



# BloomFilter的特点

---

- ❑ 不存在漏报：某个元素在某个集合中，肯定能报出来；
- ❑ 可能存在误报：某个元素不在某个集合中，可能也被认为存在：false positive；
- ❑ 确定某个元素是否在某个集合中的代价和总的元素数目无关
  - 查询时间复杂度： $O(1)$



# Bloom Filter参数的确定

- 单个元素某次没有被置位为1的概率为： $1 - \frac{1}{m}$
- k个Hash函数中没有一个对其置位的概率为： $\left(1 - \frac{1}{m}\right)^k$
- 如果插入n个元素，仍未将其置位的概率为： $\left(1 - \frac{1}{m}\right)^{kn}$
- 因此，此位被置位的概率为： $1 - \left(1 - \frac{1}{m}\right)^{kn}$

# Bloom Filter参数的确定

- 查询中，若某个待查元素对应的k位都被置位，则算法会判定该元素在集合中。因此，该元素被误判的概率(上限)为：

$$q(k) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

- 考虑到：

$$\left(1 - \frac{1}{m}\right)^{kn} = \left(1 + \frac{1}{-m}\right)^{-m \cdot \frac{kn}{m}} = \left(\left(1 + \frac{1}{-m}\right)^{-m}\right)^{-\frac{kn}{m}} \approx e^{-\frac{kn}{m}}$$

- 从而：

$$P(k) \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

# Bloom Filter参数的确定

□  $P(k)$  为幂指数函数，取对数后求导：

$$P(k) = \left(1 - e^{-\frac{kn}{m}}\right)^k \xrightarrow{\text{令 } b = e^{-\frac{n}{m}}} (1 - b^{-k})^k$$
$$\Rightarrow \ln P(k) = k \ln(1 - b^{-k})$$
$$\xrightarrow{\text{取关于 } k \text{ 的导数}} \frac{1}{P(k)} P'(k) = \ln(1 - b^{-k}) + k \frac{b^{-k} \ln b}{1 - b^{-k}}$$

$$\ln(1 - b^{-k}) + k \frac{b^{-k} \ln b}{(1 - b^{-k})} = 0$$
$$\Rightarrow (1 - b^{-k}) \ln(1 - b^{-k}) = b^{-k} \ln b^{-k}$$
$$\Rightarrow 1 - b^{-k} = b^{-k} \Rightarrow b^{-k} = \frac{1}{2} \Rightarrow e^{-\frac{kn}{m}} = \frac{1}{2}$$
$$\Rightarrow k = \ln 2 \cdot \frac{m}{n} \approx 0.693 \cdot \frac{m}{n}$$

$$P(k) = \left(1 - \frac{1}{2}\right)^k = 2^{-k} = 2^{-\ln 2 \frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

# 参数m、k的确定

□ m的计算公式:

■ 由 
$$P(k) = \left(1 - \frac{1}{2}\right)^k = 2^{-k} = 2^{-\ln 2 \frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

■ 得 
$$P = 2^{-\ln 2 \frac{m}{n}} \Rightarrow \ln P = \left(\ln 2 \cdot \frac{m}{n}\right) \ln 2 \Rightarrow m = \frac{\ln P^{-1}}{(\ln 2)^2} \cdot n$$

□ 此外, k的计算公式:

$$k = \ln 2 \cdot \frac{m}{n} = \frac{\ln P^{-1}}{\ln 2}$$

□ 至此, 任意先验给定可接受的错误率, 即可确定参数空间m和Hash函数个数k。

# Bloom Filter参数的讨论

□ 1.442695041

□ 若接收误差率为 $10^{-6}$ 时，  
需要位的数目为 $29 \cdot n$ 。

$$\begin{cases} m = \frac{\ln P^{-1}}{(\ln 2)^2} \cdot n \\ k = \ln 2 \cdot \frac{m}{n} = \frac{\ln P^{-1}}{\ln 2} \end{cases}$$

p	m/n	k
0.5	1.442695041	1
$2^{-2}$	2.885390082	2
$2^{-3}$	4.328085123	3
$2^{-4}$	5.770780164	4
$2^{-5}$	7.213475204	5
$2^{-6}$	8.656170245	6
$2^{-7}$	10.09886529	7
$2^{-8}$	11.54156033	8
$2^{-9}$	12.98425537	9
$2^{-10}$	14.42695041	10
$2^{-20}$	28.85390082	11
$2^{-30}$	43.28085123	12
$2^{-40}$	57.70780164	13
$2^{-50}$	72.13475204	14

# Bloom Filter的特点

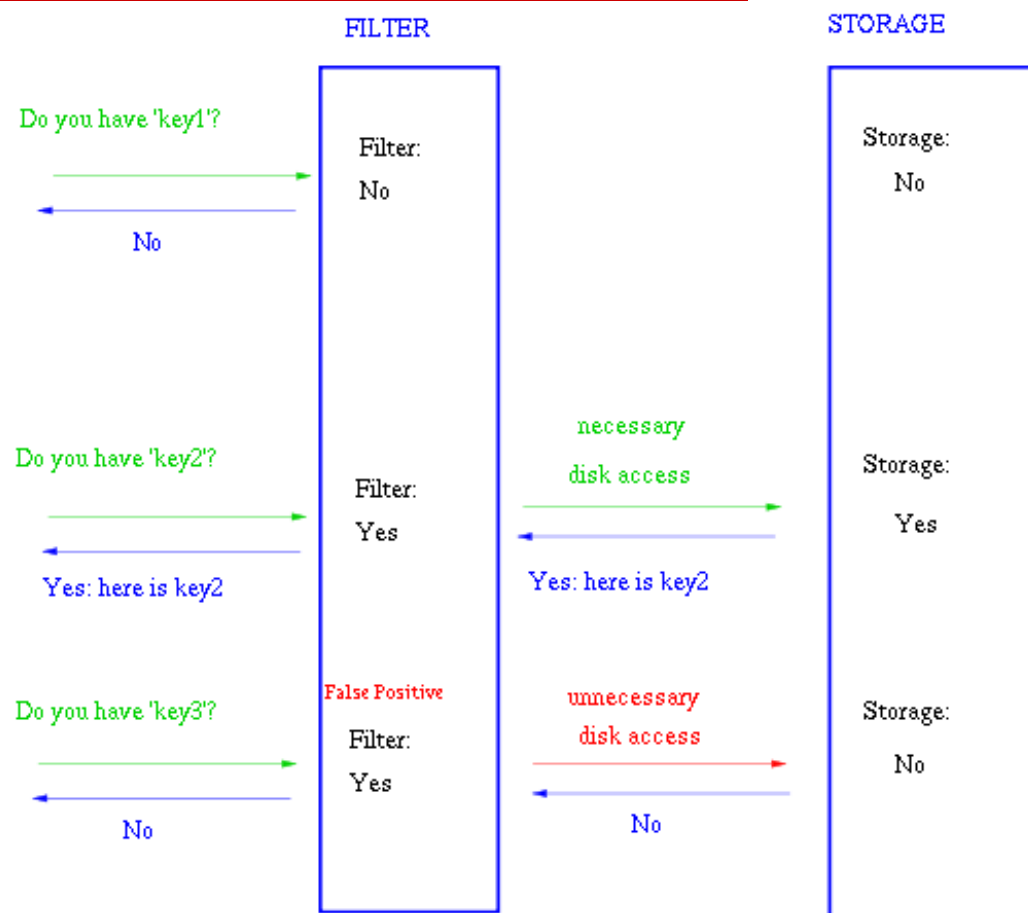
- 优点：相比于其它的数据结构，Bloom Filter在空间和时间方面都有巨大的优势。Bloom Filter存储空间是线性的，插入/查询时间都是常数。另外，Hash函数相互之间没有关系，方便由硬件并行实现。Bloom Filter不存储元素本身，在某些对保密要求非常严格的场合有优势。
- 很容易想到把位向量变成整数数组，每插入一个元素相应的计数器加1，这样删除元素时将计数器减掉就可以了。然而要保证安全的删除元素并非如此简单。首先我们必须保证删除的元素的确在BloomFilter里面。这一点单凭这个过滤器是无法保证的。另外计数器下溢出也会造成问题(槽的值已经是0了，仍然执行删除操作)。

# BloomFilter用例

---

- ❑ Google著名的分布式数据库Bigtable使用了布隆过滤器来查找不存在的行或列，以减少磁盘查找的IO次数；
- ❑ Squid网页代理缓存服务器在cachedigests中使用了BloomFilter；
- ❑ Venti文档存储系统采用BloomFilter来检测先前存储的数据；
- ❑ SPIN模型检测器使用BloomFilter在大规模验证问题时跟踪可达状态空间；
- ❑ Google Chrome浏览器使用BloomFilter加速安全浏览服务；
- ❑ 在很多Key-Value系统中也使用BloomFilter来加快查询过程，如Hbase，Accumulo，Leveldb。
  - 一般而言，Value保存在磁盘中，访问磁盘需要花费大量时间，然而使用BloomFilter可以快速判断某个Key是否存在，因此可以避免很多不必要的磁盘IO操作；另外，引入布隆过滤器会带来一定的内存消耗。

# Bloom Filter + Storage结构





# 排序的目的

---

- 排序本身：得到有序的序列
- 方便查找
  - 长度为N的有序数组，查找某元素的时间复杂度是多少？
  - 长度为N的有序链表，查找某元素的时间复杂度是多少？
    - 单链表、双向链表
    - 如何解决该问题？

# 跳跃链表(Skip List)

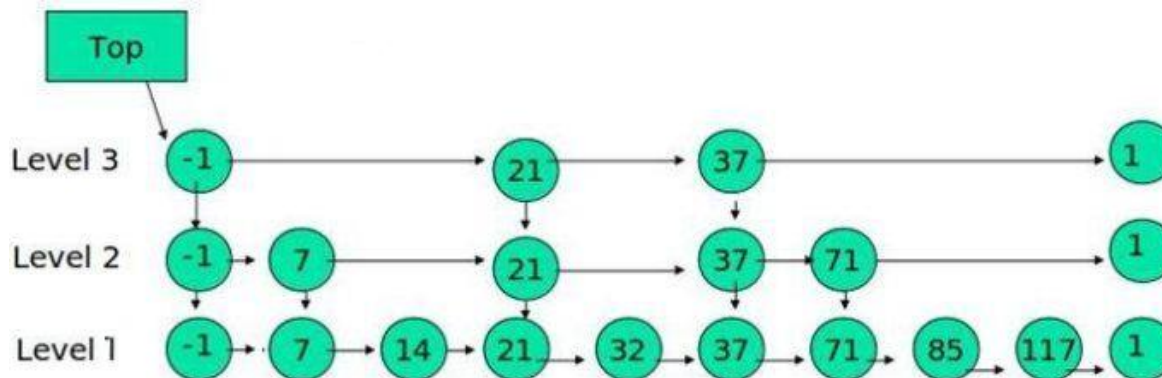
---

- Treaps/RB-Tree/BTree
- 跳跃链表是一种随机化数据结构，基于并联的链表，其效率与RBTree相当。具有简单、高效、动态(Simple、Effective、Dynamic)的特点。
- 跳跃链表对有序的链表附加辅助结构，在链表中的查找可以快速的跳过部分结点(因此得名)。
  - 查找、增加、删除的期望时间都是 $O(\log N)$ 
    - with high probability(W.H.P.  $\approx 1 - 1/(n^\alpha)$ )

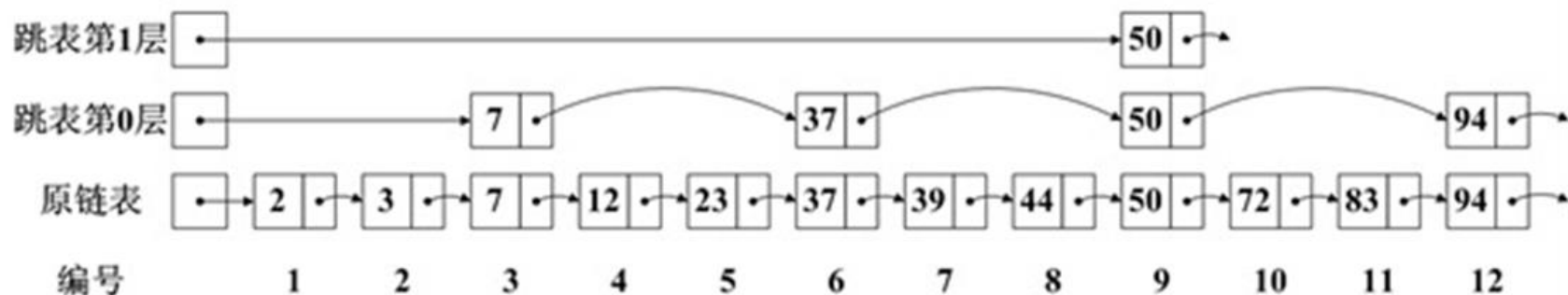
# 跳跃链表(Skip List)

- 跳跃列表在并行计算中非常有用，数据插入可以在跳表的不同部分并行进行，而不用全局的数据结构重新平衡。
- 跳跃列表是按层建造的。底层是一个普通的有序链表。每个更高层都充当下面列表的“快速跑道”，这里在层 $i$ 中的元素按某个固定的概率 $p$ 出现在层 $i+1$ 中。平均起来，每个元素都在 $1/(1-p)$ 个列表中出现。
- 思考：为什么是 $1/(1-p)$ ?

# 跳跃表示例

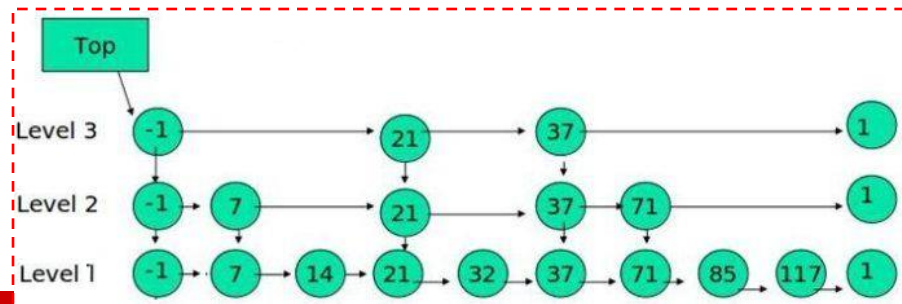


跳跃表：跳跃间隔(Skip Interval) 为 3，层次(Level)共2层



注：各个文献中对于“层”、“间隔”的定义略有差

# 双层跳表时间计算

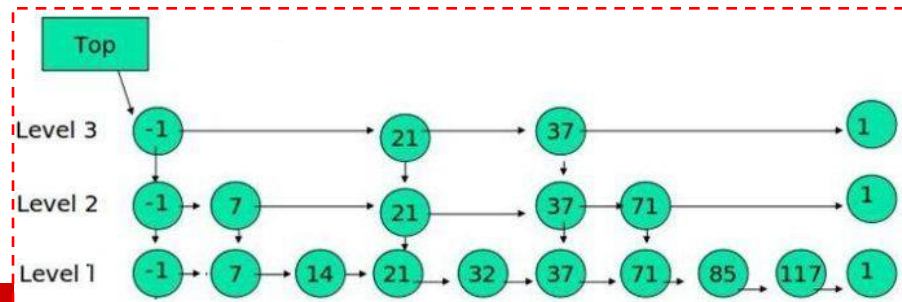


- 粗略估算查找时间： $T = L1 + L2/L1$  ( $L1$ 是稀疏层， $L2$ 是稠密层)
  - 在 $L2$ 上均匀取值，构成 $L1$ ；则 $L2/L1$ 是 $L1$ 上相邻两个元素在 $L2$ 上的平均长度
  - $L1$ ：在稀疏层的最差查找次数
  - $L1/L2$ ：在稀疏层没有找到元素，跳转到稠密层需要找的次数
- 若基本链表的长度为 $n$ ，即 $|L2|=n$ ， $|L1|$ 为多少， $T$ 最小呢？
  - $T(x) = x + n/x$ ，对 $x$ 求导，得到 $x = \sqrt{n}$
  - $\min(T(x)) = 2\sqrt{n}$

# 时间复杂度分析

- 粗略估算查找时间： $T=L1 + L2/L1 + L3/L2$ ( $L1$ 是稀疏层， $L2$ 是稠密层， $L3$ 是基本层)
  - 在 $L3$ 上均匀取值，构成 $L2$ ；在 $L2$ 上均匀取值，构成 $L1$ ；则 $L2/L1$ 是 $L1$ 上相邻两个元素在 $L2$ 上的平均长度， $L3/L2$ 是 $L2$ 上相邻两个元素在 $L3$ 上的平均长度
- 若基本链表的长度为 $n$ ，即 $|L3|=n$ ， $|L1|$ 、 $|L2|$ 为多少， $T$ 最小呢？
  - $T(x,y)=x+y/x+n/y$ ，对 $x,y$ 求偏导，得到 $x=\sqrt[3]{n}$
  - $\min(T(x,y))= 3*\sqrt[3]{n}$

# 跳表最优时间分析



- 建立k层的辅助链表，可以得到最小时间  $T(n) = k * \sqrt[k]{n}$
- 问题：在n已知的前提下，k取多大最好呢？
- 显然，当  $k = \log N$  时， $T(n) = \log N * n^{(-\log N)}$ 
  - 问：  $n^{(-\log N)}$  等于几？
- 一个很容易在实践中使用的结论是：
  - 当基本链表的数目为N时，建立  $k = \log N$  个辅助链表，每个上层链表的长度取下层链表的一半，则能够达到  $O(\log N)$  的时间复杂度。
- 理想跳表：ideal skip list

# 插入元素

- 随着底层链表的插入，某一段上的数据将不满足理想跳表的要求，需要做些调整。
  - 将底层链表这一段上元素的中位数在拷贝到上层链表中；
  - 重新计算上层链表，使得上层链表仍然是底层链表的1/2；
  - 如果上述操作过程中，上层链表不满足要求，继续上上层链表的操作。
- 新的数据应该在上层甚至上上层链表中吗？因为要找一半的数据放在上层链表(为什么是一半？)，因此：**抛硬币！**



# 插入元素后的跳表维护

---

- 考察待需要提升的某段结点。
- 若抛硬币得到的随机数 $p > 0.5$ ，则提升到上层，继续抛硬币，直到 $p > 0.5$ ；
  - 或者到了顶层仍然 $p > 0.5$ ，建立一个新的顶层

# 删除元素

---

- 在某层链表上找到了该元素，则删除；如果该层链表不是底层链表，跳转到下一层，继续本操作。

# 进一步说明的问题

- 若多次查找，并且相邻查找的元素有相关性(相差不大)，可使用**记忆化查找**进一步加快查找速度。
- 对关于k的函数 $T(n) = k * \sqrt[k]{n}$ 求导，可计算得到 $k = \ln N$ 处函数取最小值，最小值是 $e \ln N$ 。
  - 对于 $N=10000$ ，k取 $\ln N$ 和 $\log N$ ，两个最小值分布是：25.0363和26.5754。
- 强调：编程方便，尤其方便将增删改查操作扩展成并行算法。
- 跳表用单链表可以实现吗？用双向链表呢？

# 进行 $10^6$ 次随机操作后的统计结果

P	平均操作时间	平均列高	总结点数	每次查找跳跃次数 (平均值)	每次插入跳跃次数 (平均值)	每次删除跳跃次数 (平均值)
2/3	0.0024690 ms	3.004	91233	39.878	41.604	41.566
1/2	0.0020180 ms	1.995	60683	27.807	29.947	29.072
1/e	0.0019870 ms	1.584	47570	27.332	28.238	28.452
1/4	0.0021720 ms	1.330	40478	28.726	29.472	29.664
1/8	0.0026880 ms	1.144	34420	35.147	35.821	36.007

进行 $10^6$ 次随机操作后的统计结果

# Code: Structure

```
class CSkipList
{
private:
    SSkipNode* m_pHead;
    int m_nSize;

public:
    CSkipList()
    {
        m_pHead = new SSkipNode(0);
        m_nSize = 0;
    }

    SSkipNode* Find(int value) const;
    bool Insert(int value);
    bool Delete(int value);
    SSkipNode* FindIndex(int n);
    int GetSize() const {return m_nSize;}
    bool IsEmpty() const {return (m_nSize <= 0);}
    void Print() const;
private:
    void PrintLayer(const SSkipNode* pNode) const;
    bool IsSuccess() const
    {
        return rand() > RAND_MAX * 0.36787944117;
    }
};

typedef struct tagSSkipNode
{
    int value;
    tagSSkipNode* pNext;
    tagSSkipNode* pNextLayer;

    tagSSkipNode(int v) : value(v), pNext(NULL), pNextLayer(NULL) {}
} SSkipNode;
```

# Code: Find

```
SSkipNode* CSkipList::Find(int value) const
{
    if(!m_pHead)
        return NULL;
    SSkipNode* pre = m_pHead;
    SSkipNode* cur = m_pHead->pNext;
    while(true)
    {
        while(cur && (cur->value < value)) //当前值小，则遍历下一个
        {
            pre = cur;
            cur = cur->pNext;
        }
        if(cur && cur->value == value)
            return cur;
        if(!pre->pNextLayer)
            break;
        pre = pre->pNextLayer;
        cur = pre->pNext;
    }
    return NULL;
}
```

# Code: Insert

```
bool CSkipList::Insert(int value)
{
    if(!m_pHead->pNext)
    {
        m_pHead->pNext = new SSkipNode(value);
        m_nSize = 1;
        return true;
    }
    SSkipNode* pre = m_pHead;
    SSkipNode* cur = m_pHead->pNext;
    stack<SSkipNode*> path;
    while(true)
    {
        while(cur && (cur->value < value)) //当前值小, 则遍历下一个
        {
            pre = cur;
            cur = cur->pNext;
        }
        if(cur && cur->value == value) //已经存在
            return false;
        path.push(pre); //记录插入点
        if(!pre->pNextLayer)
            break;
        pre = pre->pNextLayer;
        cur = pre->pNext;
    }
    //插入到pre的后面, cur的前面
    SSkipNode* now = new SSkipNode(value);
    now->pNext = cur;
    pre->pNext = now;
    //随机上升
    SSkipNode* nowInLayer;
    SSkipNode* pLayerHead;
    while(!path.empty())
    {
        if(!IsSuccess())
            break;
        path.pop();
        //得到层的插入位置
        if(path.empty())
            pre = m_pHead;
        else
            pre = path.top();

        //生成结点
        nowInLayer = new SSkipNode(value);
        nowInLayer->pNextLayer = now;
        if(path.empty()) //说明顶层后仍然成功, 则新建层
        {
            pLayerHead = new SSkipNode(0); //生成层的新头指针
            pLayerHead->pNext = m_pHead->pNext;
            pLayerHead->pNextLayer = m_pHead->pNextLayer;
            m_pHead->pNextLayer = pLayerHead; //退化到下一层
            m_pHead->pNext = nowInLayer;
        }
        else
        {
            nowInLayer->pNext = pre->pNext;
            pre->pNext = nowInLayer;
        }

        //为下次上升做准备
        now = nowInLayer;
    }
    m_nSize++;
    return true;
}
```

# Insert part1

```
bool CSkipList::Insert(int value)
{
    if(!m_pHead->pNext)
    {
        m_pHead->pNext = new SSkipNode(value);
        m_nSize = 1;
        return true;
    }
    SSkipNode* pre = m_pHead;
    SSkipNode* cur = m_pHead->pNext;
    stack<SSkipNode*> path;
    while(true)
    {
        while(cur && (cur->value < value)) //当前值小，则遍历下一个
        {
            pre = cur;
            cur = cur->pNext;
        }
        if(cur && cur->value == value) //已经存在
            return false;
        path.push(pre); //记录插入点
        if(!pre->pNextLayer)
            break;
        pre = pre->pNextLayer;
        cur = pre->pNext;
    }
    //插入到pre的后面，cur的前面
    SSkipNode* now = new SSkipNode(value);
    now->pNext = cur;
    pre->pNext = now;
}
```



# Insert part2

```
//随机上升
SSkipNode* nowInLayer;
SSkipNode* pLayerHead;
while(!path.empty())
{
    if(!IsSuccess())
        break;
    path.pop();
    //得到层的插入位置
    if(path.empty())
        pre = m_pHead;
    else
        pre = path.top();

    //生成结点
    nowInLayer = new SSkipNode(value);
    nowInLayer->pNextLayer = now;
    if(path.empty()) //说明顶层后仍然成功, 则新建层
    {
        pLayerHead = new SSkipNode(0); //生成层的新头指针
        pLayerHead->pNext = m_pHead->pNext;
        pLayerHead->pNextLayer = m_pHead->pNextLayer;
        m_pHead->pNextLayer = pLayerHead; //退化到下一层
        m_pHead->pNext = nowInLayer;
    }
    else
    {
        nowInLayer->pNext = pre->pNext;
        pre->pNext = nowInLayer;
    }

    //为下次上升做准备
    now = nowInLayer;
}
m_nSize++;
return true;
}
```

# Code: Delete

```
bool CSkipList::Delete(int value)
{
    if(!m_pHead)
        return false;
    SSkipNode* pre = m_pHead;
    SSkipNode* cur = m_pHead->pNext;
    SSkipNode* pHeadPre = NULL;
    SSkipNode* pHead = m_pHead;
    bool bDelete = false;
    while(true)
    {
        while(cur && (cur->value < value)) //当前值小，则遍历下一个
        {
            pre = cur;
            cur = cur->pNext;
        }
        if(cur && cur->value == value)
        {
            bDelete = true;
            pre->pNext = cur->pNext;
            delete cur;
            if(!pHead->pNext) //该层没有元素，则删除该层
            {
                if(pHead == m_pHead) //顶层
                {
                    SSkipNode* pNL = m_pHead->pNextLayer;
                    m_pHead->pNextLayer = pNL ? pNL->pNextLayer : NULL;
                    m_pHead->pNext = pNL ? pNL->pNext : NULL;
                    delete pNL;
                }
                else
                {
                    pHeadPre->pNextLayer = pHead->pNextLayer;
                    delete pHead;
                    pHead = pHeadPre;
                }
                pre = pHead;
                cur = pre->pNext;
                continue; //删除该层后，pre/cur已经向下移动了一层
            }
        }
        if(!pre->pNextLayer)
            break;
        pre = pre->pNextLayer;
        cur = pre->pNext;

        pHeadPre = pHead;
        pHead = pHead->pNextLayer;
    }
    m_nSize--;
    return bDelete;
}
```

# Delete part

```
if(!m_pHead)
    return false;
SSkipNode* pre = m_pHead;
SSkipNode* cur = m_pHead->pNext;
SSkipNode* pHeadPre = NULL;
SSkipNode* pHead = m_pHead;
bool bDelete = false;
```

```
m_nSize--;
return bDelete;
```

```
while(true)
{
    while(cur && (cur->value < value)) //当前值小，则遍历下一个
    {
        pre = cur;
        cur = cur->pNext;
    }
    if(cur && cur->value == value)
    {
        bDelete = true;
        pre->pNext = cur->pNext;
        delete cur;
        if(!pHead->pNext) //该层没有元素，则删除该层
        {
            if(pHead == m_pHead) //顶层
            {
                SSkipNode* pNL = m_pHead->pNextLayer;
                m_pHead->pNextLayer = pNL ? pNL->pNextLayer : NULL;
                m_pHead->pNext = pNL ? pNL->pNext : NULL;
                delete pNL;
            }
            else
            {
                pHeadPre->pNextLayer = pHead->pNextLayer;
                delete pHead;
                pHead = pHeadPre;
            }
            pre = pHead;
            cur = pre->pNext;
            continue; //删除该层后，pre/cur已经向下移动了一层
        }
    }
    if(!pre->pNextLayer)
        break;
    pre = pre->pNextLayer;
    cur = pre->pNext;

    pHeadPre = pHead;
    pHead = pHead->pNextLayer;
}
```

# Code: Test

```
int _tmain(int argc, _TCHAR* argv[])
{
    CSkipList sl;
    int i;
    for(i = 0; i < 100; i++)    //随机插入数据
        sl.Insert(rand() % 100);
    sl.Print();
    while(!sl.IsEmpty())    //随机删除数据
    {
        SSkipNode* p = sl.FindIndex(rand() % sl.GetSize());
        if(p)
        {
            int num = p->value;
            if(sl.Delete(num))
                cout << "Delete " << num << endl;
            else
                cout << "No Delete " << num << endl;
            sl.Print();
            cout << "=====\n";
        }
    }
    return 0;
}
```

# Test

→ 3 ↓ 44 ↓ 69 ↓ 82 ↓

→ 3 ↓ 5 ↓ 44 ↓ 69 ↓ 82 ↓

→ 3 ↓ 5 ↓ 44 ↓ 47 ↓ 58 ↓ 69 ↓ 82 ↓

→ 0 ↓ 3 ↓ 5 ↓ 11 ↓ 44 ↓ 47 ↓ 58 ↓ 69 ↓ 82 ↓ 91 ↓

→ 0 3 5 11 24 27 41 44 47 53 58 61 67 69 82 91 95

→ 3 ↓ 44 ↓ 69 ↓ 82 ↓

→ 3 ↓ 5 ↓ 44 ↓ 69 ↓ 82 ↓

→ 3 ↓ 5 ↓ 44 ↓ 47 ↓ 69 ↓ 82 ↓

→ 0 ↓ 3 ↓ 5 ↓ 11 ↓ 44 ↓ 47 ↓ 69 ↓ 82 ↓ 91 ↓

→ 0 3 5 11 24 27 41 44 47 53 61 67 69 82 91 95

# MD5

---

- MD5(Message Digest Algorithm), 消息摘要算法, 为计算机安全领域广泛使用的一种散列函数, 用以提供消息的完整性保护。
  - 文件号RFC 1321(R.Rivest,MIT Laboratory for Computer Science and RSA Data Security Inc. April 1992)
- 对于任意长度的信息, 经过MD5算法, 得到长度为128bit的摘要。

# MD5的框架理解

□ 对于长度为512bit的信息，可以通过处理，得到长度为128bit的摘要。

□ 初始化摘要：

0x0123456789ABCDEF FEDCBA9876543210

■ A=0x01234567      B=0x89ABCDEF

■ C=0xFEDCBA98      D=0x76543210

□ 现在的工作，是要用长度为512位的信息，变换初始摘要。

# MD5的总体理解

- 定义变量a,b,c,d,分别记录A,B,C,D;
- 将512bit的信息按照32bit一组, 分成16组;  
分别记为 $M_j$  ( $0 \leq j \leq 15$ );
- 取某正数s、 $t_k$ , 定义函数:  
$$FF(a,b,c,d,M_j,s,t_k) = (a + F(b,c,d) + M_j + t_k) \ll s$$
- 利用 $M_j$ 分别进行信息提取, 将结果保存到a
  - 其中,  $F(X,Y,Z) = (X \& Y) \mid (\sim X \& Z)$

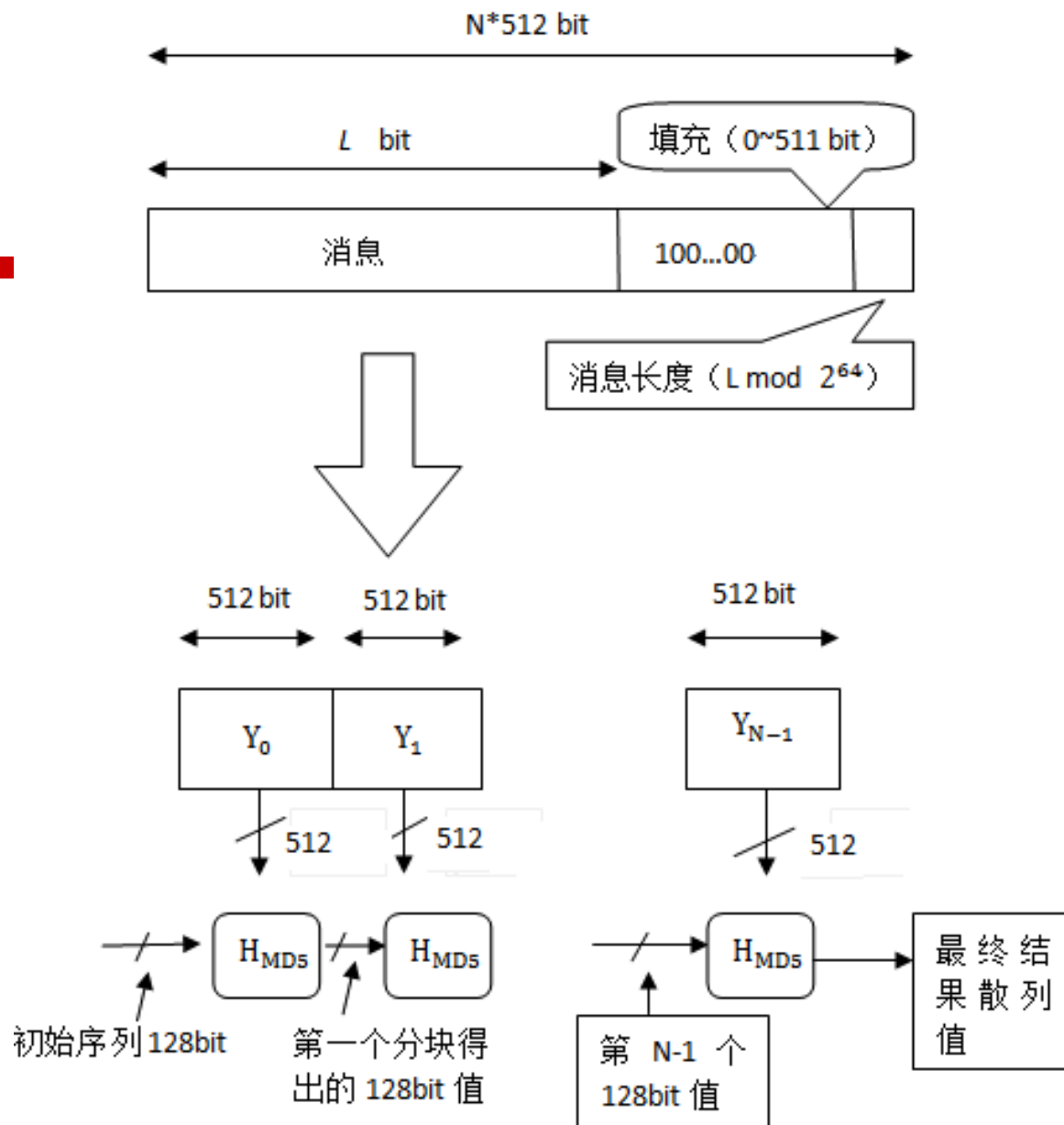


# MD5的总体理解

---

- 经过以上16次变换， $a, b, c, d$ 带有了 $M_j$ 的信息
- 事实上经过四轮这样的变换(4轮\*16次=64次)
- 经过64次变换后，将 $a, b, c, d$ 累加给 $A, B, C, D$
- 此时，完成了512bit信息的提取；进行下一个512bit信息的相同操作

# MD5框架



# MD5细致算法

---

- 在算法中，首先需要对信息进行填充，使其长度对512求余的结果等于448。
- 填充的方法如下，在信息的后面填充一个1和若干个0，直到满足上面的条件。然后，在这个结果后面附加一个以64位二进制表示的原始信息长度Length。
- 经过这两步的处理，数据总长度为 $N*512+448+64=(N+1)*512$ ，即长度恰好是512的整数倍。

# MD5的细致算法

□ 定义四个非线性函数：

■  $F(X, Y, Z) = (X \& Y) | ((\sim X) \& Z)$        $H(X, Y, Z) = X \wedge Y \wedge Z$

■  $G(X, Y, Z) = (X \& Z) | (Y \& (\sim Z))$        $I(X, Y, Z) = Y \wedge (X | (\sim Z))$

□  $M_j$  表示数据的第  $j$  个子分组 ( $0 \leq j \leq 15$ , 循环表示), 常数  $t_k$  是  $2^{32} * \text{abs}(\sin(k))$  的整数部分,  $1 \leq k \leq 64$ , 单位是弧度。

□  $FF(a, b, c, d, M_j, s, t_k) = (a + F(b, c, d) + M_j + t_k) \ll s$

□  $GG(a, b, c, d, M_j, s, t_k) = (b + G(b, c, d) + M_j + t_k) \ll s$

□  $HH(a, b, c, d, M_j, s, t_k) = (b + H(b, c, d) + M_j + t_k) \ll s$

□  $II(a, b, c, d, M_j, s, t_k) = (b + I(b, c, d) + M_j + t_k) \ll s$

# 64次操作

## 第一轮

FF(a, b, c, d, M0, 7, 0xd76aa478)  
FF(d, a, b, c, M1, 12, 0xe8c7b756)  
FF(c, d, a, b, M2, 17, 0x242070db)  
FF(b, c, d, a, M3, 22, 0xc1bdceee)  
FF(a, b, c, d, M4, 7, 0xf57c0faf)  
FF(d, a, b, c, M5, 12, 0x4787c62a)  
FF(c, d, a, b, M6, 17, 0xa8304613)  
FF(b, c, d, a, M7, 22, 0xfd469501)  
FF(a, b, c, d, M8, 7, 0x698098d8)  
FF(d, a, b, c, M9, 12, 0x8b44f7af)  
FF(c, d, a, b, M10, 17, 0xfffff5bb1)  
FF(b, c, d, a, M11, 22, 0x895cd7be)  
FF(a, b, c, d, M12, 7, 0x6b901122)  
FF(d, a, b, c, M13, 12, 0xfd987193)  
FF(c, d, a, b, M14, 17, 0xa679438e)  
FF(b, c, d, a, M15, 22, 0x49b40821)

## 第二轮

GG(a, b, c, d, M1, 5, 0xf61e2562)  
GG(d, a, b, c, M6, 9, 0xc040b340)  
GG(c, d, a, b, M11, 14, 0x265e5a51)  
GG(b, c, d, a, M0, 20, 0xe9b6c7aa)  
GG(a, b, c, d, M5, 5, 0xd62f105d)  
GG(d, a, b, c, M10, 9, 0x02441453)  
GG(c, d, a, b, M15, 14, 0xd8a1e681)  
GG(b, c, d, a, M4, 20, 0xe7d3fbc8)  
GG(a, b, c, d, M9, 5, 0x21e1cde6)  
GG(d, a, b, c, M14, 9, 0xc33707d6)  
GG(c, d, a, b, M3, 14, 0xf4d50d87)  
GG(b, c, d, a, M8, 20, 0x455a14ed)  
GG(a, b, c, d, M13, 5, 0xa9e3e905)  
GG(d, a, b, c, M2, 9, 0xfcefa3f8)  
GG(c, d, a, b, M7, 14, 0x676f02d9)  
GG(b, c, d, a, M12, 20, 0x8d2a4c8a)

## 第三轮

HH(a, b, c, d, M5, 4, 0xffffa3942)  
HH(d, a, b, c, M8, 11, 0x8771f681)  
HH(c, d, a, b, M11, 16, 0x6d9d6122)  
HH(b, c, d, a, M14, 23, 0xfde5380c)  
HH(a, b, c, d, M1, 4, 0xa4beea44)  
HH(d, a, b, c, M4, 11, 0x4bdecfa9)  
HH(c, d, a, b, M7, 16, 0xf6bb4b60)  
HH(b, c, d, a, M10, 23, 0xbebfbcb70)  
HH(a, b, c, d, M13, 4, 0x289b7ec6)  
HH(d, a, b, c, M0, 11, 0xeaal27fa)  
HH(c, d, a, b, M3, 16, 0xd4ef3085)  
HH(b, c, d, a, M6, 23, 0x04881d05)  
HH(a, b, c, d, M9, 4, 0xd9d4d039)  
HH(d, a, b, c, M12, 11, 0xe6db99e5)  
HH(c, d, a, b, M15, 16, 0x1fa27cf8)  
HH(b, c, d, a, M2, 23, 0xc4ac5665)

## 第四轮

II(a, b, c, d, M0, 6, 0xf4292244)  
II(d, a, b, c, M7, 10, 0x432aff97)  
II(c, d, a, b, M14, 15, 0xab9423a7)  
II(b, c, d, a, M5, 21, 0xfc93a039)  
II(a, b, c, d, M12, 6, 0x655b59c3)  
II(d, a, b, c, M3, 10, 0x8f0ccc92)  
II(c, d, a, b, M10, 15, 0xffefff47d)  
II(b, c, d, a, M1, 21, 0x85845dd1)  
II(a, b, c, d, M8, 6, 0x6fa87e4f)  
II(d, a, b, c, M15, 10, 0xfe2ce6e0)  
II(c, d, a, b, M6, 15, 0xa3014314)  
II(b, c, d, a, M13, 21, 0x4e0811a1)  
II(a, b, c, d, M4, 6, 0xf7537e82)  
II(d, a, b, c, M11, 10, 0xbd3af235)  
II(c, d, a, b, M2, 15, 0x2ad7d2bb)  
II(b, c, d, a, M9, 21, 0xeb86d391)

# 思考

## 新浪微博长URL的压缩（存储、查找相关）

取消关注 | 14

...

### 【题目】

新浪微博发布内容要求字符不超过140，但是用户如果在发布内容中有很长的url时，会认为是很多字符。所以新浪上发布内容包含一个URL时，时把他压缩成一个TinyURL(缩小)。比如：

(因为无法发站外链接，所以去掉了链接中的http://头)

输入：zhidao.baidu.com/search?ct=17&pn=0&tn=ikaslist&rn=10&word=helloworld&ie=utf-8&fr=www

实际显示：//asdfa.cn/ak78ss

前面asdfa.cn是对应域名zhidao.baidu.com，后面长长的字符串被压缩成ak78ss。

### 【问题】

现在让你来设计TinyURL的实现，以下问题要怎么设计：

- (1)：域名后面的编码如何实现？
- (2)：对于已经映射过的一个URL，怎么查找已存在的TinyUrl？
- (3)：有10亿个url，一个服务上存不下，需要多台服务器，怎么设计实现
- (4)：让你来设计这样一个服务，最大的问题是什么？

### 【探讨】

暂时的思路(根据网友的回答汇总)，欢迎探讨：

(1)：两种思路，一种是把真实网址存数据库，然后取自增id，做个哈希或者进制转换之类的，生成短网址，用的时候查一下就行了；另一种是直接真实网址哈希然后截取特定位数，6位的话  $(26+26+10)^6$  种组合，应该够用了，实在不行再做一次碰撞检测

- (2)：直接key-value存储查询
- (3)：二次哈希，根据ak78ss这样的值映射到不同机器上，hash或者字母序层次下去
- (4)：查询速度？响应时间？还有过期的url在浪费存储空间？

# 海量数据系统设计小结

---

## □ 相信理论，重视实践

- $O(N\log N)$  优于  $O(N^2)$

- $O(N^{2.81})$  VS  $O(N^3)$

- 内存命中、并行开销

## □ 融会贯通，适度改造

- POI结构/R树变体

# 思考：这段代码什么作用？

---

```
int HammingWeight(unsigned int n)
{
    n = (n & 0x55555555) + ((n & 0xaaaaaaaa)>>1);
    n = (n & 0x33333333) + ((n & 0xcccccccc)>>2);
    n = (n & 0x0f0f0f0f) + ((n & 0xf0f0f0f0)>>4);
    n = (n & 0x00ff00ff) + ((n & 0xff00ff00)>>8);
    n = (n & 0x0000ffff) + ((n & 0xffff0000)>>16);
    return n;
}
```



# 思考：这段代码在做什么？

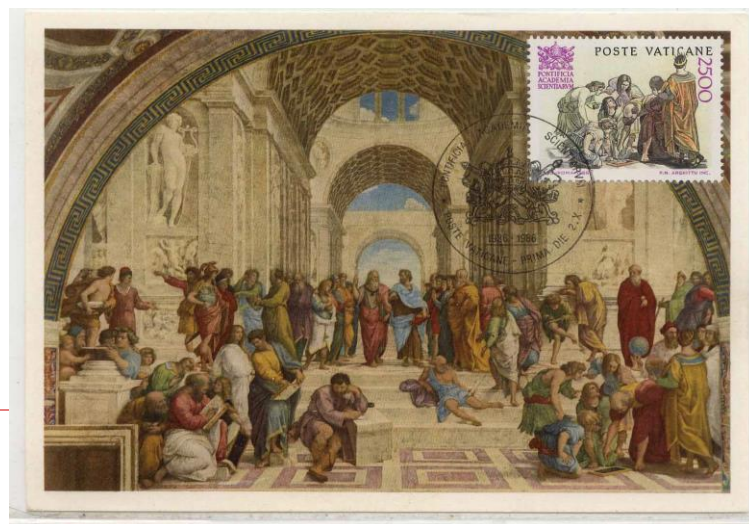
```
int _tmain(int argc, _TCHAR* argv[])
{
    const int N = 100;
    int M = 32;
    unsigned a[N];
    a[0] = 0;
    for(int i = 0; i < N; i++)
    {
        a[i] = ((a[i]>>1]) >> 1) | ((i&1) << (M-1));
        Print(i, a[i]);
    }
    return 0;
}
```

0	000000000
1	100000000
2	010000000
3	110000000
4	001000000
5	101000000
6	011000000
7	111000000
8	000100000
9	100100000
10	010100000
11	110100000
12	001100000
13	101100000
14	011100000
15	111100000
16	000010000
17	100010000
18	010010000
19	110010000
20	001010000
21	101010000
22	011010000
23	111010000
24	000110000
25	100110000
26	010110000

# 算法



- 所学并非无用，而是知识体系尚未达到能够用它的程度。
- 思想恒久远，算法永流传
  - 据传，三国时期诸葛亮实施空城计。
  - 傅作义偷袭西柏坡，毛主席重演空城计。
- 三千年前欧几里德发明辗转相除法，迄今仍被广大程序员所使用。



# 结束语

□ 知识的掌握是1,2,4,8,16.....的速度;

- 每天递增0.01:  $1.01^{365} = ?$
- 设置适合自己的“学习率”。

□ 掌握算法的根本途径是多练习代码。

- 书读百遍，其义自见。

□ 算法远远没有到此为止.....

- 下列属于算法范畴吗?
- 网站开发/OA工作流
- 操作系统资源调度/编译原理词法、语法、语义分析/数据库设计/计算机网络协议包解析
- 机器学习/数据挖掘/计算机视觉
- .....

```
double Pow(double x, int n)
{
    if(n == 0)
        return 1;
    if(n == 1)
        return x;
    if(n == 2)
        return x*x;
    double p = Pow(x, n/2);
    p *= p;
    return (n % 2 == 0) ? p : p*x;
}

double Power(double x, int n)
{
    if(n < 0)
        return 1/Pow(x, -n);
    return Pow(x, n);
}

int _tmain(int argc, _TCHAR* argv[])
{
    cout << Power(1.01, 365) << endl;
    return 0;
}
```

# 我们在这里

□ <http://wenda.ChinaHadoop.cn>

■ 视频/课程/社区

□ 微博

■ @ChinaHadoop

■ @邹博\_机器学习

□ 微信公众号

■ 小象学院

■ 大数据分析挖掘

互联网新技术在线教育领航者

小象问答 搜索标题、用户 全站内容搜索 提问 首页 动态 发现 话题 通知

全部 招聘求职 机器学习 大数据平台技术 DCon 大数据行业应用 NoSQL数据库 数据科学 江湖救急

发现 最新 推荐 热门 等待回复

graphviz has no attribute 'write' 贡献  
邹博 回复了问题 • 2 人关注 • 1 个回复 • 3 次浏览 • 2017-04-09 15:48

sklearn中如何理解Pipeline机制 贡献  
数据分析与数据挖掘 邹博 回复了问题 • 2 人关注 • 1 个回复 • 28 次浏览 • 2017-04-09 15:39

关于9.Logistic回归的ppt中第9页的对数线性函数 贡献  
机器学习 邹博 回复了问题 • 3 人关注 • 3 个回复 • 39 次浏览 • 2017-04-09 15:35

关于“贝叶斯估计中，最大后验概率估计就是结构化风险最小化的例子：当模型是条件概率分布，损失函数为对数损失函数，模型的复杂度由模型的先验概率表示，结构化风险最小化就等价于最大后验概率估计” 贡献  
机器学习 邹博 回复了问题 • 2 人关注 • 1 个回复 • 26 次浏览 • 2017-04-09 15:27

关于连续值的预测 贡献  
咨询 邹博 回复了问题 • 2 人关注 • 1 个回复 • 31 次浏览 • 2017-04-09 15:24

拉格朗日对偶函数为什么一定是凸函数 贡献  
数据科学 邹博 回复了问题 • 2 人关注 • 2 个回复 • 26 次浏览 • 2017-04-09 15:20

梯度下降公式中的斯堪J 是 贡献  
机器学习 邹博 回复了问题 • 2 人关注 • 1 个回复 • 29 次浏览 • 2017-04-09 15:17

深度学习适合做预测吗？ 贡献  
深度学习 邹博 回复了问题 • 2 人关注 • 1 个回复 • 27 次浏览 • 2017-04-09 15:15

关于6.4PCA\_FeatureSelection.py中plt.legend的参数疑问 贡献  
机器学习 邹博 回复了问题 • 2 人关注 • 1 个回复 • 28 次浏览 • 2017-04-09 15:04

@邹博 有哪些可以下载数据源的网站？ 贡献  
数据分析与数据挖掘 邹博 回复了问题 • 4 人关注 • 1 个回复 • 31 次浏览 • 2017-04-09 14:53

LDA主题模型 贡献  
机器学习 邹博 回复了问题 • 2 人关注 • 1 个回复 • 29 次浏览 • 2017-04-09 14:45

代码10.6bagging\_ridged老师提到了采样率设为0.2能够使峰值部分的数据被体现出来。这是为什么呢？ 贡献  
机器学习 邹博 回复了问题 • 2 人关注 • 1 个回复 • 22 次浏览 • 2017-04-09 14:26

GraphViz's executables not found 贡献  
机器学习 邹博 回复了问题 • 3 人关注 • 2 个回复 • 23 次浏览 • 2017-04-09 13:47

决策树中关于feature\_importances代码的问题 贡献  
机器学习 邹博 回复了问题 • 2 人关注 • 1 个回复 • 6 次浏览 • 2017-04-09 13:11

专题  
招聘求职  
大数据行业应用  
数据科学  
系统与编程  
云计算技术

热门话题 更多 >  
机器学习 907 个问题, 230 人关注  
spark 387 个问题, 172 人关注  
hadoop 1059 个问题, 155 人关注  
python数据分析 171 个问题, 28 人关注  
数据分析与数据挖掘 54 个问题, 111 人关注

热门用户 更多 >  
小心巴 14 个问题, 0 次赞同  
叉叉V 45 个问题, 22 次赞同  
铁甲无声 10 个问题, 0 次赞同  
带刀锦衣卫 13 个问题, 0 次赞同

---

感谢大家！

恳请大家批评指正！