

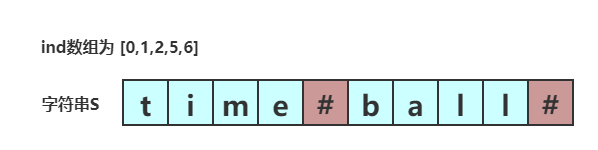
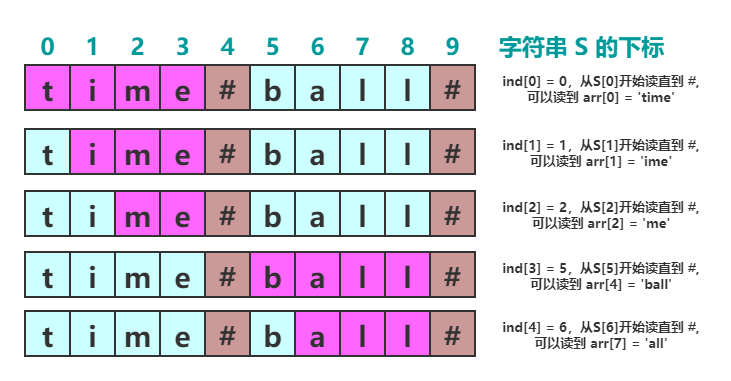
手撕字典树(Trie 树) C++ 另附暴力哈希解法

[前额叶没长好](https://leetcode-cn.com/u/time-limit/)发布于 1 天前2.3kC++

**先来瞧一瞧题意**

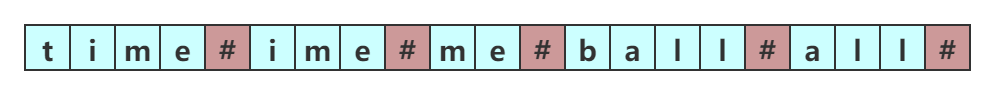
给出一个字符串数组记为 arr*arr*。通过一个算法将 arr*arr* 中的**全部**或**部分**字符串拼接起来，并用 # 分割，并给出一个与 arr*arr* 等长的 ind*ind* 数组。这个算法还必须保证得出 S*S* 是最短的。

这个 ind*ind* 是干啥的咧？ 设拼接后的字符串为 S*S*，那么 ind*ind* 里保存了 S*S* 的一些下标。这些下标满足：从S*S*的 ind[i]*ind*[*i*]开始读取，直到最近的 # 结束，可以读出 arr[i]*arr*[*i*]。

说不明白，直接上图，以 ['time', 'ime', 'me', 'ball', 'all'] 为例。  
构造的 S*S* 及 ind*ind* 如下：  
  
根据 S*S* 及 ind*ind* 还原 arr*arr* 步骤如下：  


**再来撕一撕题意**

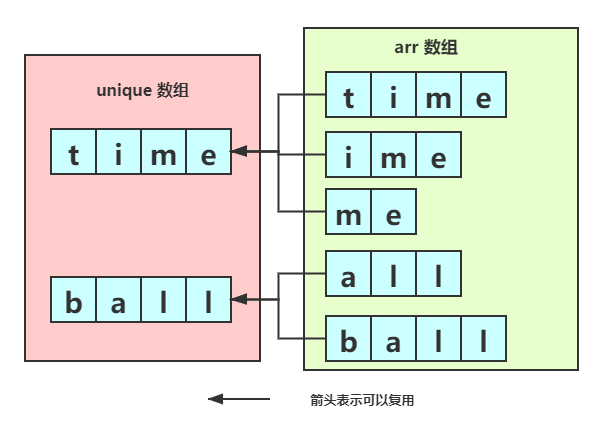
首先拼接很简单，直接将每个字符串拼接，中间塞上 # 即可。但是这样 **不是最短** 呀。  
我们先来考虑一下 # 的作用，它起到了提示读取停止的作用，即 **标记了字符串的结尾**。这也就使得arr[i]*arr*[*i*]在 S*S* 中必然紧靠着一个 #。  
又因为还原的时候总是从 ind[i]*ind*[*i*] **连续读取** 直到遇见 #。这也就使得 arr[i] 在 S*S* 中必然是 **连续** 的。

接下来再看一眼用最朴素的方法拼接成的S*S*：  
  
我们能做的就是尽量减少 # 的数量。减少 # 就意味着必然有若干个 arr[i]*arr*[*i*] **复用** 同一个 #。

那么如何确定 arr*arr* 中哪些元素可以复用一个 # 呢？  
设 a, b 为 arr 中的两个元素，并有len(a) >= len(b)。当 b 是 a 的后缀时，显然两者可以复用一个 #。当 b 不是 a 的后缀时，显然不可以复用。

进一步思考一下后缀的两个性质：

1. 对于任意字符串 a, a 必然是自身的后缀。
2. 对于任意字符串 a, b, c， 如果 b 是 a 的后缀，c 是 b 的后缀，那么c 必然也是 a 的后缀。

如果我们可以找出 arr*arr* 中那些仅为自身后缀的字符串，将这些字符串记为数组 unique*unique*，长度为 n*n*。  
那么最终答案为unique*unique*中字符串长度的累加和，在加上 n 个 # 的长度，即为 \sum\_{i=0}^{n}unique[i].size() + n∑*i*=0*n*​*unique*[*i*].*size*()+*n*。  
**因为 unique*unique* 中的所有元素两两之间都不可能复用 #。而 arr*arr* 中的其他字符串都至少可以与一个unique*unique* 中的元素复用 #。**  
如下图所示：  


**字典树**

字典树是一中多叉树，可以根据组成字符串的不同的元素数量来变化子节点的数量。以只包含小写字符为例，那么对应的字典树为 **二十六叉树**，每个子节点对应一个字符。如果字符串既包含小写又包含大写，那么对应的字典树为 **五十二叉树**。用 C++ 描述字典树的结点，如下：

struct Node {

Node \*next[26]; //指针域。当前结点的边，指向对应的子节点。

bool flag; //标记字段。如果为 true，说明该树记录了一个由根节点到当前结点的路径组成的字符串。反之，则不存在。

Node() {

for(int i = 0 ; i < 26; i++) {

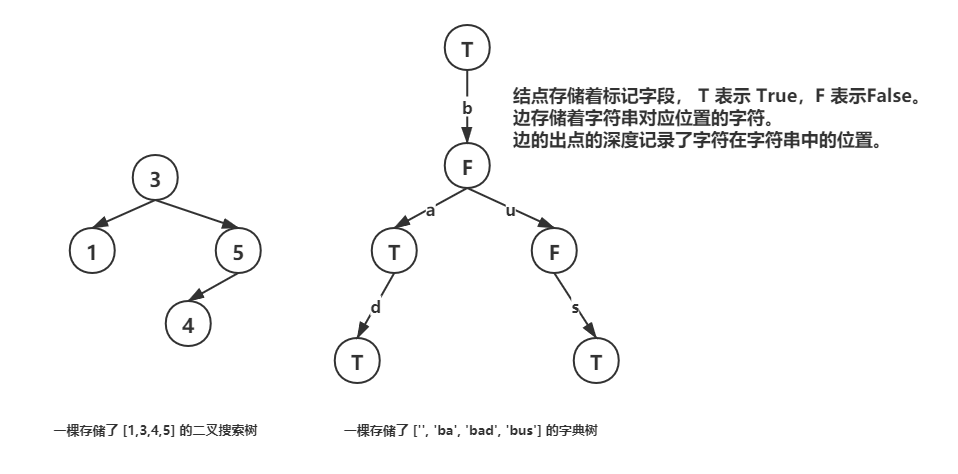
next[i] = nullptr;

}

flag = false;

}

};

与传统的二叉树不同的是，传统二叉树数据大多存在结点内，边只有链接的作用。字典树的大多数数据都存储在边上，结点可能只存储了一个标记字段。如下图所示：  


所有的数据结构无非都是为了实现高效的增删改查。下面来看下字典树的增。

**插入字符串**

这是一个**递归**的过程，设根节点为 root = new Node()，待插入的字符串 为 S。

1. 如果 S 为空串，则标记 root->flag = true。插入结束。
2. 设 S 的首字符为 c。如果root->next[c] 为空，则创建一个新结点：root->next[c] = new Node(); 如果root->next[c] 不为空，说明之前插入过具有相同前缀的字符串，**没有关系，字典树就是为了高效存储字符串而生的，结点复用是它的灵魂。**
3. 将 c 从 S 中删除。root下移：root = root->next[c]。跳转第一步。

示例代码如下：

// root 为当前子树根节点； str 待插入的字符串； str[pos] 为等待插入的字符串。

void Insert(Node \*root, const string &str, int pos) {

if(pos == str.size()) { //所有字符都已经插入了。

root->flag = true; //记录有 str 到此结束了。

return ;

}

int ind = str[pos]-'a'; //计算str[pos]对应的子节点的下标。

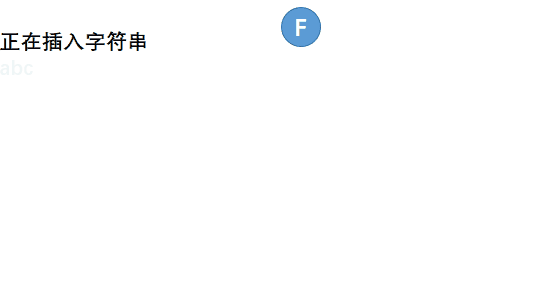
if(root->next[ind] == nullptr) {

root->next[ind] = getNodePtr(); //记录 str 在 pos 处的字符。

}

Insert(root->next[ind], str, pos+1); pos+1 等价于将已插入的字符删除了。

}

可以结合下图理解：  


**回到题目**

对于该题来说，我们会根据字符串建立字典树就已经够用了。字典树的其他操作以后再做分享~

我们发现从根节点开始遍历，到达任意节点时，所经过的路径总是对应着字符串的前缀，和这题不相干呀。但是如果我们将字符串倒着插入呢？那也就意味着字符串后缀变前缀了，而且并不影响本题的正确性。  
好了，现在我们将 arr*arr* 数组中的所有元素都倒着插入字典树。但是和我们的 unique*unique* 还是没有得到unique*unique* 数组呀。憋慌，让我们来观察下刚刚构造的字典树，我们发现从根节点到所有叶子结点的路径对应的字符串就是 unique*unique* 数组呀。  
那么答案就变成了，**从根节点到所有叶子结点的路径长度之和**加上**n个#的长度**。

打完收工，上代码。

**代码是用 C++ 写的，用了 placement new operator，可以复用内存，省去从堆分配内存的时间。**

**如果感觉有点意思，可以关注 👏**[**HelloNebula**](https://pic.leetcode-cn.com/f18e68ed2306a147b615407e0b0260900672940df4a2fd9a7f7dc9b91e23e55e.jpg)**👏 ~**

struct Node {

Node \*next[26];

bool flag; //标记字段，本地中并没有用到该字段

Node() {

for(int i = 0 ; i < 26; i++) {

next[i] = nullptr;

}

flag = false;

}

};

const int MAXN = 2000\*7+1;

Node pool[MAXN]; //定义一个结点池，结合 placement new 实现内存复用。

class Solution {

int poolPtr;

inline Node \*getNodePtr() {

return new (&pool[poolPtr++]) Node();

}

void Insert(Node \*root, const string &str, int pos) {

if(pos < 0) {

root->flag = true; //记录有 str 到此结束了。

return ;

}

int ind = str[pos]-'a';

if(root->next[ind] == nullptr) {

root->next[ind] = getNodePtr(); //记录 str 在 pos 处的字符。

}

Insert(root->next[ind], str, pos-1);

}

int Count(Node \*root, int edgeCnt) {

bool isLeaf = true;

int sum = 0;

for(int i = 0; i < 26; i++) {

if(root->next[i] != nullptr) {

isLeaf = false;

sum += Count(root->next[i], edgeCnt+1);

}

}

if(isLeaf) {

return edgeCnt + 1; // 1 是 '#' 的长度

}

return sum;

}

public:

int minimumLengthEncoding(vector<string>& words) {

poolPtr = 0;

auto root = getNodePtr();

for(auto it = words.cbegin(); it != words.cend(); ++it) {

Insert(root, \*it, int(it->size())-1);

}

return Count(root, 0);

}

};

**暴力哈希解法**

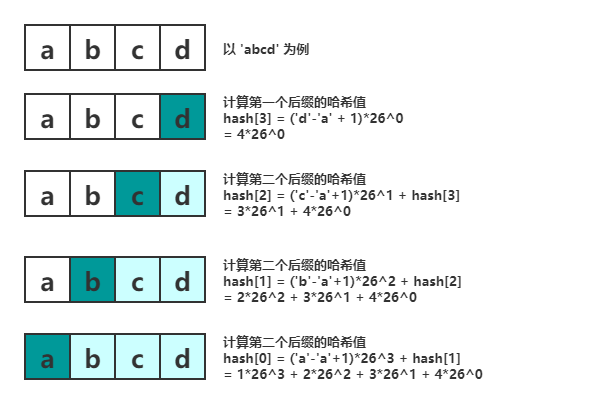
首先来思考两个事情：  
设有两个字符串 a, b，那么：

1. 如果a.size() > b.size()，那么 a 必然不是 b 的后缀。
2. 如果a.size() == b.size() 且 a 是 b 的后缀，那么 b 也是 a 后缀，因为两者相等。

将 arr 按字符串长度降序排序，设排序后的数组为 sarr。  
假设 sarr 中的所有元素的长度不相等，那么当 sarr[i] 是 sarr[j] 的后缀时， 必有 i < j。  
基于此假设，我们有如下处理过程，设初始时 i = 0, 哈希标记容器 mark 为空，答案 sum = 0：

1. 如果 sarr 的 所有元素均已处理，处理结束。
2. 计算 sarr[i] 的哈希值，记为 hv。
3. 如果 hv 已在mark中，则说明 sarr[i] 为 sarr[0..i-1] 中某个串的后缀，i = i+1， 跳转第一步
4. 反之，说明sarr[i]不是 sarr[0...i-1] 中字符串的后缀，将 hv 及 sarr[i]所有后缀的哈希值插入 mark，i=i+1。

**当sarr中有长度相等的字符串时，也不会影响上述处理过程的正确性，可以思考下为什么~**

哈希计算过程如下图所示：  
  
代码如下：

class Solution {

public:

int minimumLengthEncoding(vector<string>& words) {

sort(words.begin(), words.end(), [](const string &lhs, const string &rhs) ->bool {

if(lhs.size() == rhs.size()) {

return lhs < rhs; //二级排序，按字典序，这一个其实不影响最终结果。

}

return lhs.size() > rhs.size(); //一级排序，按长度降序。

});

unordered\_set<int64\_t> mark;

int sum = 0;

int64\_t pow[7] = {0}; //初始化pow数组

pow[0] = 1;

for(int i = 1; i < 7; ++i) {

pow[i] = pow[i-1]\*26; //pow[i] = 26^i

}

for(auto it = words.cbegin(); it != words.cend(); ++it) {

const string &str = \*it;

int64\_t hv = 0;

for(int i = str.size()-1, len = 0; i >= 0; --i, ++len) {

hv += pow[len]\*(str[i]-'a'+1);

if(i != 0) {

mark.insert(hv); // 将后缀的hash值插入mark

} else {

// 此时 hv 是整个 str 的hash值。

// 判断一下 str 是否已经出现过，即是否是某些已拼接字符串的后缀。

if(mark.find(hv) != mark.end()) {

continue; //str是其他字符串的后缀，continue。

}

mark.insert(hv); // str 是unique 中的一个，插入 mark。

sum += str.size() + 1; //拼接

}

}

}

return sum;

}

};

**扫码关注 快乐加倍**



下一篇：这道中等题超简单，最基础的字符串知识即可

© 著作权归作者所有