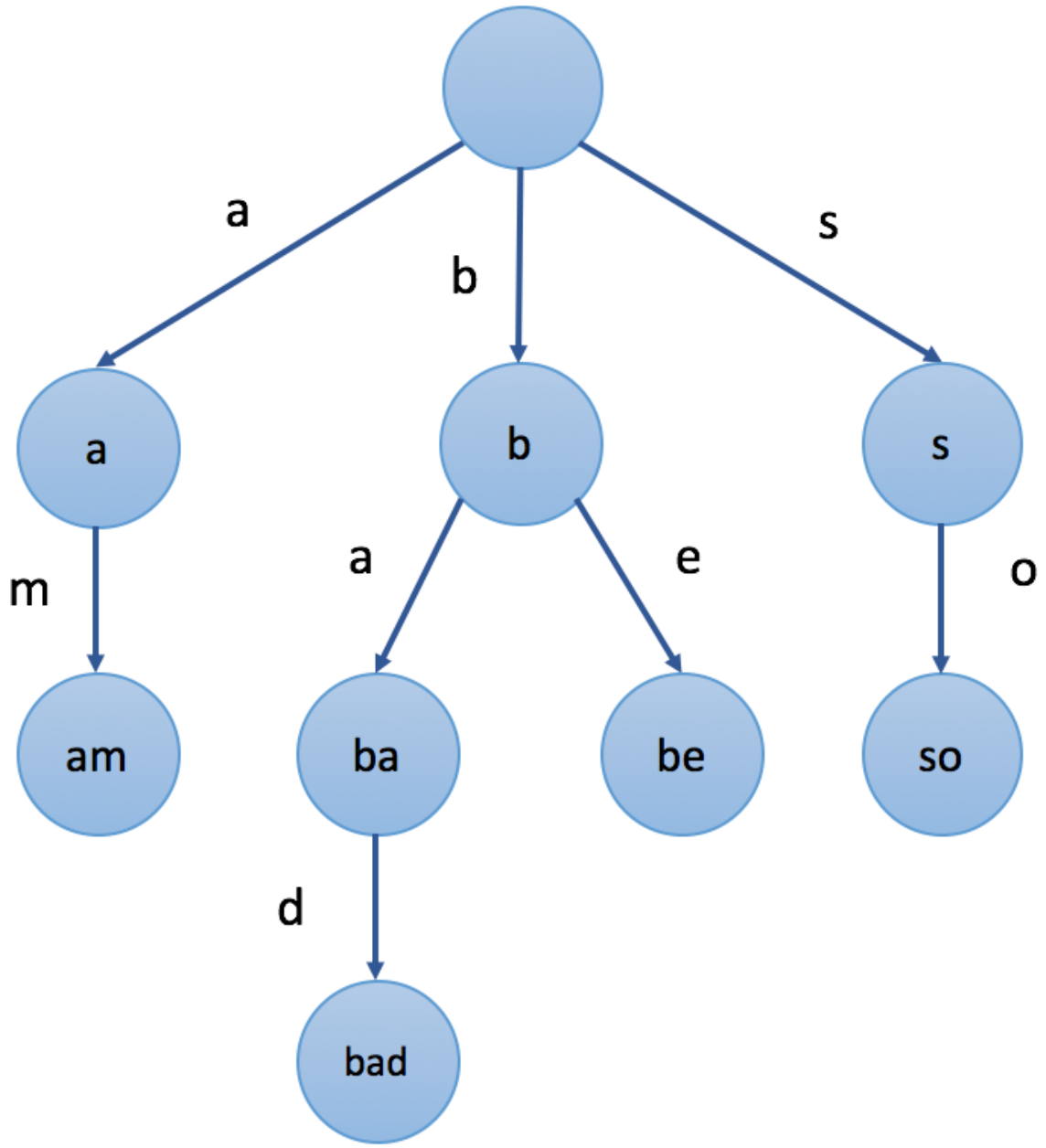
什么是前缀树?

前缀树 是 N叉树 的一种特殊形式。通常来说，一个前缀树是用来 存储字符串 的。前缀树的每一个节点代表一个 字符串 （ 前缀 ）。每一个节点会有多个子节点，通往不同子节点的路径上有着不同的字符。子节点代表的字符串是由节点本身的 原始字符串，以及 通往该子节点路径上所有的字符 组成的。

下面是前缀树的一个例子：



在上图示例中，我们在节点中标记的值是该节点对应表示的字符串。例如，我们从根节点开始，选择第二条路径 'b'，然后选择它的第一个子节点 'a'，接下来继续选择子节点 'd'，我们最终会到达叶节点 "bad"。节点的值是由从根节点开始，与其经过的路径中的字符按顺序形成的。

值得注意的是，根节点表示`空字符串`。

前缀树的一个重要的特性是，节点所有的后代都与该节点相关的字符串有着共同的前缀。这就是`前缀树`名称的由来。

我们再来看这个例子。例如，以节点 "b" 为根的子树中的节点表示的字符串，都具有共同的前缀 "b"。反之亦然，具有公共前缀 "b" 的字符串，全部位于以 "b" 为根的子树中，并且具有不同前缀的字符串来自不同的分支。

前缀树有着广泛的应用，例如自动补全，拼写检查等等。我们将在后面的章节中介绍实际应用场景。

如何表示一个前缀树?

前缀树的特别之处在于字符和子节点之间的对应关系。有许多不同的表示前缀树节点的方法，这里我们只介绍其中的两种方法。

方法一 - 数组

第一种方法是用`数组`存储子节点。例如，如果我们只存储含有字母`a`到`z`的字符串，我们可以在每个节点中声明一个大小为26的数组来存储其子节点。对于特定字符`c`，我们可以使用`c - 'a'`作为索引来查找数组中相应的子节点。

```
// change this value to adapt to different cases
#define N 26

struct TrieNode {
    TrieNode* children[N];

    // you might need some extra values according to different cases
};

/** Usage:
 *  Initialization: TrieNode root = new TrieNode();
 *  Return a specific child node with char c: (root->children)[c - 'a']
 */
```

访问子节点十分`快捷`。访问一个特定的子节点比较`容易`，因为在大多数情况下，我们很容易将一个字符转换为索引。但并非所有的子节点都需要这样的操作，所以这可能会导致`空间的浪费`。

方法二 - Map

第二种方法是使用`Hashmap`来存储子节点。

我们可以在每个节点中声明一个Hashmap。Hashmap的键是字符，值是相对应的子节点。

```
struct TrieNode {
    unordered_map<char, TrieNode*> children;

    // you might need some extra values according to different cases
};

/** Usage:
 *  Initialization: TrieNode root = new TrieNode();
 *  Return a specific child node with char c: (root->children)[c]
 */
```

通过相应的字符来访问特定的子节点 `更为容易` 。但它可能比使用数组 `稍慢一些` 。但是，由于我们只存储我们需要的子节点，因此 `节省了空间` 。这个方法也更加 `灵活` ，因为我们不受到固定长度和固定范围的限制。

补充

我们已经提到过如何表示前缀树中的子节点。除此之外，我们也需要用到一些其他的值。

例如，我们知道，前缀树的每个节点表示一个字符串，但并不是所有由前缀树表示的字符串都是有意义的。如果我们只想在前缀树中存储单词，那么我们可能需要在每个节点中声明一个布尔值（Boolean）作为标志，来表明该节点所表示的字符串是否为一个单词。

基本操作

---

insertion in Trie

当我们在二叉搜索树中插入目标值时，在每个节点中，我们都需要根据 `节点值` 和 `目标值` 之间的关系，来确定目标值需要去往哪个子节点。同样地，当我们向前缀树中插入一个目标值时，我们也需要根据插入的 `目标值` 来决定我们的路径。

更具体地说，如果我们在前缀树中插入一个字符串 `S` ，我们要从根节点开始。 我们将根据 `S[0]` （S中的第一个字符），选择一个子节点或添加一个新的子节点。然后到达第二个节点，并根据 `S[1]` 做出选择。 再到第三个节点，以此类推。 最后，我们依次遍历 S 中的所有字符并到达末尾。 末端节点将是表示字符串 S 的节点。

我们来用伪代码总结一下以上策略：

```
1. Initialize: cur = root
2. for each char c in target string S:
3.      if cur does not have a child c:
4.          cur.children[c] = new Trie node
5.      cur = cur.children[c]
6. cur is the node which represents the string S
```

通常情况情况下，你需要自己构建前缀树。构建前缀树实际上就是多次调用插入函数。但请记住在插入字符串之前要 `初始化根节点` 。

search in Trie

1. 搜索前缀

   正如我们在前缀树的简介中提到的，所有节点的后代都与该节点相对应字符串的有着共同前缀。因此，很容易搜索以特定前缀开头的任何单词。

   同样地，我们可以根据给定的前缀沿着树形结构搜索下去。一旦我们找不到我们想要的子节点，搜索就以失败终止。否则，搜索成功。

   我们来用伪代码总结一下以上策略：

```
1. Initialize: cur = root
2. for each char c in target string S:
3.      if cur does not have a child c:
4.          search fails
5.      cur = cur.children[c]
6. search successes
```

2. 搜索单词

   你可能还想知道如何搜索特定的单词，而不是前缀。我们可以将这个词作为前缀，并同样按照上述同样的方法进行搜索。

   1. 如果搜索失败，那么意味着没有单词以目标单词开头，那么目标单词绝对不会存在于前缀树中。
   2. 如果搜索成功，我们需要检查目标单词是否是前缀树中单词的前缀，或者它本身就是一个单词。为了进一步解决这个问题，你可能需要稍对节点的结构做出修改。

   > 提示：往每个节点中加入布尔值可能会有效地帮助你解决这个问题。

208.实现 Trie (前缀树)（中等）

1. 题目描述

   实现一个 Trie (前缀树)，包含 `insert`, `search`, 和 `startsWith` 这三个操作。

   示例:

```
Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // 返回 true
trie.search("app");     // 返回 false
trie.startsWith("app"); // 返回 true
trie.insert("app");
trie.search("app");     // 返回 true
```

   说明:

   - 你可以假设所有的输入都是由小写字母 `a-z` 构成的。
   - 保证所有输入均为非空字符串。

2. 简单实现

```
class Trie {
public:
    unordered_map<char, Trie*> children;
```

```
    bool isWord;
    /** Initialize your data structure here. */
    Trie() {
        isWord = false;
    }
    /** Inserts a word into the trie. */
    void insert(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0)
                cur->children[word[i]] = new Trie();
            cur = cur->children[word[i]];
        }
        cur->isWord = true;
    }
    /** Returns if the word is in the trie. */
    bool search(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0)
                return false;
            cur = cur->children[word[i]];
        }
        return cur->isWord;
    }
    /** Returns if there is any word in the trie that starts with the given prefix.
*/
    bool startsWith(string prefix) {
        Trie* cur = this;
        for(int i = 0; i < prefix.size(); i++){
            if(cur->children.count(prefix[i]) <= 0)
                return false;
            cur = cur->children[prefix[i]];
        }
        return true;
    }
};
/**
 * Your Trie object will be instantiated and called as such:
 * Trie* obj = new Trie();
 * obj->insert(word);
 * bool param_2 = obj->search(word);
 * bool param_3 = obj->startsWith(prefix);
 */
```

实际应用1

前缀树广泛应用于 `存储` 字符串和 `检索` 关键字，特别是和 `前缀` 相关的关键字。前缀树的实际应用场景非常简单。通常，你只需要提供插入的方法和一些与单词前缀相关的搜索操作。我们会在本章节后，为你提供一些配套习题。

当然，前缀树还会涉及到一些稍微复杂的实际问题：

1. 自动补全
2. 一个简单的实现自动补全的方法是在前缀树中存储ngram，并根据词频进行搜索推荐。请仔细思考，什么是解决这个问题的理想的节点结构。
3. 拼写检查（添加与搜索单词 - 数据结构设计）
4. 在前缀树中找到具有相同前缀的单词很容易。但是怎么找到相似的单词呢？你可能需要运用一些搜索算法来解决这个问题。

677.键值映射（中等）

1. 题目描述

   实现一个 MapSum 类里的两个方法，`insert` 和 `sum`。

   对于方法 `insert`，你将得到一对（字符串，整数）的键值对。字符串表示键，整数表示值。如果键已经存在，那么原来的键值对将被替代成新的键值对。

   对于方法 `sum`，你将得到一个表示前缀的字符串，你需要返回所有以该前缀开头的键的值的总和。

   示例 1:

   ```
   输入: insert("apple", 3), 输出: Null
   输入: sum("ap"), 输出: 3
   输入: insert("app", 2), 输出: Null
   输入: sum("ap"), 输出: 5
   ```

2. 简单实现

   对Trie做简单修改，以适用于本题

   ```
   class Trie {
   public:
       unordered_map<char, Trie*> children;
       bool isWord;
       int val;//值
       Trie() {
           isWord = false;
       }
       void insert(string word, int val) {
           Trie* cur = this;
           for(int i = 0; i < word.size(); i++){
               if(cur->children.count(word[i]) <= 0)
                   cur->children[word[i]] = new Trie();
               cur = cur->children[word[i]];
           }
           cur->isWord = true;
           cur->val = val;
       }
       Trie* startsWith(string prefix) {//返回prefix对应的Trie节点
           Trie* cur = this;
           for(int i = 0; i < prefix.size(); i++){
               if(cur->children.count(prefix[i]) <= 0)
                   return NULL;
               cur = cur->children[prefix[i]];
   ```

```
        }
        return cur;
    }
};

class MapSum {
public:
    Trie* root;
    MapSum() {
        root = new Trie();
    }
    void insert(string key, int val) {
        root->insert(key, val);
    }
    int sum(string prefix) {
        Trie* cur = root->startsWith(prefix);
        if(!cur) return 0;
        else return count(cur);
    }
    int count(Trie* root){
        int ans = 0;
        if(root->isWord) ans += root->val;
        for(auto it = root->children.begin(); it != root->children.end(); it++)
            ans += count(it->second);
        return ans;
    }
};
```

3. 其他解法

利用map，遍历查找前缀

```
class MapSum {
private:
    map<string, int>myMap;

public:
    MapSum() {
    }
    void insert(string key, int val) {
        myMap[key] = val;
    }
    int sum(string prefix) {
        int length = prefix.size();
        int sum=0;
        for (auto &iter : myMap)
            if (iter.first.substr(0, length) == prefix)//前缀匹配
                sum += iter.second;
        return sum;
    }
};
```

648.单词替换（中等）

1. 题目描述

在英语中，我们有一个叫做 `词根`(root)的概念，它可以跟着其他一些词组成另一个较长的单词——我们称这个词为 `继承词`(successor)。例如，词根 `an`，跟随着单词 `other`(其他)，可以形成新的单词 `another`(另一个)。

现在，给定一个由许多词根组成的词典和一个句子。你需要将句子中的所有 `继承词` 用 `词根` 替换掉。如果 `继承词` 有许多可以形成它的 `词根`，则用最短的词根替换它。

你需要输出替换之后的句子。

示例 1:

```
输入: dict(词典) = ["cat", "bat", "rat"]
sentence(句子) = "the cattle was rattled by the battery"
输出: "the cat was rat by the bat"
```

注:

1. 输入只包含小写字母。
2. 1 <= 字典单词数 <=1000
3. 1 <= 句中词语数 <= 1000
4. 1 <= 词根长度 <= 100
5. 1 <= 句中词语长度 <= 1000

2. 简单实现

把词根用Trie树当作词根来存储；searchPrefix函数返回输入字符串的最短词根或原字符串（没有词根的时候）

```
class Trie {
public:
    unordered_map<char, Trie*> children;
    bool isWord;
    Trie() {
        isWord = false;
    }
    void insert(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0)
                cur->children[word[i]] = new Trie();
            cur = cur->children[word[i]];
        }
        cur->isWord = true;
    }
    string searchPrefix(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->isWord) return word.substr(0, i);//一旦找到一个词根就返回
            if(cur->children.count(word[i]) <= 0)//无词根
                return word;
            cur = cur->children[word[i]];
        }
```

```
            return word;
        }
    };
    class Solution {
    public:
        string replaceWords(vector<string>& dict, string sentence) {
            Trie* root = new Trie();
            if(dict.size() == 0 || sentence.size() == 0) return sentence;
            for(int i = 0; i < dict.size(); i++)
                root->insert(dict[i]);
            string ans = "";
            int l = 0;
            int r = 0;
            while(r < sentence.size()){
                if(sentence[r] == ' '){
                    ans += root->searchPrefix(sentence.substr(l, r-l)) + ' ';
                    l = r+1;
                    r = r+1;
                }
                else
                    r++;
            }
            ans += root->searchPrefix(sentence.substr(l, r-l));
            return ans;
        }
    };
```

211.添加与搜索单词 - 数据结构设计（中等）

1. 题目描述

设计一个支持以下两种操作的数据结构：

```
void addWord(word)
bool search(word)
```

search(word) 可以搜索文字或正则表达式字符串，字符串只包含字母 `.` 或 `a-z` 。 `.` 可以表示任何一个字母。

示例:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

说明:

你可以假设所有单词都是由小写字母 `a-z` 组成的。

2. 简单实现

插入和Trie树一样，搜索时其实就是遇到'.'就依次遍历所有的孩子替换'.'

```
class WordDictionary {
public:
    unordered_map<char, WordDictionary*> children;
    bool isWord;
    /** Initialize your data structure here. */
    WordDictionary() {
         isWord = false;
    }
    /** Adds a word into the data structure. */
    void addWord(string word) {
        WordDictionary* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0)
                cur->children[word[i]] = new WordDictionary();
            cur = cur->children[word[i]];
        }
        cur->isWord = true;
    }
    /** Returns if the word is in the data structure. A word could contain the dot
character '.' to represent any one letter. */
    bool search(string word) {
        return searchPart(this, word);
    }
    bool searchPart(WordDictionary* root, string word){//以root为根搜多word
        WordDictionary* cur = root;
        for(int i = 0; i < word.size(); i++){
            if(word[i] == '.'){//遇到'.', 发生替换
                if(i == word.size() - 1) {//'.'出现在最后一个位置
                    for(auto it = cur->children.begin(); it != cur->children.end();
it++)
                        if(it->second->isWord) return true;//任一孩子是单词就可替换，返
回true
                    return false;//没有孩子是单词，false
                }
                else{
                    //依次将'.'替换为所有的孩子，并以孩子为根继续搜索剩下的字符
                    for(auto it = cur->children.begin(); it != cur->children.end();
it++){
                        string temp = word.substr(i+1, word.size()-i-1);
                        if(searchPart(it->second, temp)) return true;//任一替换搜索到
就行
                    }
                    return false;
                }
            }
            else{//和Trie树的一样
                if(cur->children.count(word[i]) <= 0)
                    return false;
                cur = cur->children[word[i]];
            }
```

```
        }
        return cur->isWord;
    }
};
```

3. 其他解法

   将同样长度的字符串存在一起，查找时只遍历该长度的字符串们

```
class WordDictionary {
public:
    unordered_map<int, vector<string>> dict;
    /** Initialize your data structure here. */
    WordDictionary(){
    }
    /** Adds a word into the data structure. */
    void addWord(string word) {
        int n = word.length();
        if(dict.count(n) > 0)
            dict[n].push_back(word);
        else
            dict[n] = {word};
        return;
    }
    /** Returns if the word is in the data structure. A word could contain the dot
character '.' to represent any one letter. */
    bool search(string word) {
        for (string s:dict[word.length()]){
            int i;
            for (i=0; i<s.length(); i++)
                if (s[i] != word[i] && word[i] != '.')
                    break;
            if (i == word.length())
                return true;
        }
        return false;
    }
};
```

实际应用2

在前一章节中，我们练习了几道经典的的前缀树问题。但是，前缀树的实际应用并不只是那么简单。本章中，我们将为你提供一些更有趣的实际应用问题，帮助你探索前缀树的更多可能：

1. `加速深度优先搜索`
2. 有时，我们会用前缀树来加速深度优先搜索。特别是当我们使用深度优先搜索解决文字游戏类问题的时候。我们在文章后面为你提供习题：单词搜索 II 。请先尝试深度优先算法，再使用前缀树进行优化。
3. `存储其他数据类型`
4. 我们通常会使用前缀树来存储字符串，但并非总是如此。 数组中两个数的最大异或值 是一道很有意思的问题。
5. 还有一些其他应用，例如 IP 路由（最长前缀匹配）。

421.数组中两个数的最大异或值（中等）

1. 题目描述

   给定一个非空数组，数组中元素为 a0, a1, a2, … , an-1，其中 $0 \le a_i < 2^{31}$ 。

   找到 ai 和aj 最大的异或 (XOR) 运算结果，其中0 ≤ *i*, *j* < *n* 。

   你能在O(*n*)的时间解决这个问题吗?

   示例:

   ```
   输入: [3, 10, 5, 25, 2, 8]

   输出: 28

   解释: 最大的结果是 5 ^ 25 = 28.
   ```

2. 最优解法

   前缀树的方法太麻烦了，看到别人的超棒的代码，主要思想如下：

   - 首先要把所有的数字看作是31位的二进制数
   - 对于结果ans，从高位到低位，如果每一位都尽可能为1，这样的结果一定是最大的
   - 因此，对于`i = [30...0]`，我们从高位开始，仅考虑前 31-i 位能取得的最大异或值
   - 定理：若`a ^ b = c`，则`a ^ c = b`，`c ^ b = a`，因此，我们假设res的第 31-i 位为1，如果nums里存在两个数num1, num2满足`num1 ^ res = num2`，则这个res是满足的，否则不满足，res的第 31-i 位为0

   ```
   class Solution {
   public:
       int findMaximumXOR(vector<int>& nums) {
           int mask = 0;
           int ans = 0;
           for(int i = 30; i >= 0; i--){
               mask |= (1<<i);//mask前31-i位为1
               unordered_set<int> s;//保存nums内数字前31-i位的情况
               for(int j = 0; j < nums.size(); j++)
                   s.insert(mask & nums[j]);
               int temp = ans | (1 << i);//用temp暂存res第31-i位为1的值，探索是否有满足条件
   的两个数
               for(auto it = s.begin(); it != s.end(); it++){
                   if(s.count(temp^(*it)) > 0){//找到满足的，ans赋值，跳出循环继续看下一位
                       ans = temp;
                       break;
                   }
               }
           }
           return ans;
       }
   };
   ```

212.单词搜索 II（困难）

1. 题目描述

给定一个二维网格 **board** 和一个字典中的单词列表 **words**，找出所有同时在二维网格和字典中出现的单词。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

示例:

```
输入:
words = ["oath","pea","eat","rain"] and board =
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]

输出: ["eat","oath"]
```

说明: 你可以假设所有输入都由小写字母 `a-z` 组成。

提示:

- 你需要优化回溯算法以通过更大数据量的测试。你能否早点停止回溯？
- 如果当前单词不存在于所有单词的前缀中，则可以立即停止回溯。什么样的数据结构可以有效地执行这样的操作？散列表是否可行？为什么？ 前缀树如何？如果你想学习如何实现一个基本的前缀树，请先查看这个问题： 实现Trie（前缀树）。

2. 回溯法

需要添加大量的剪枝代码以防止超时，但是最终用时排名还是惨不忍睹，我辛辛苦苦写的代码啊！！！

```
class Solution {
public:
    //以board[x][y]为起点，寻找word[idx...]是否存在
    bool find(vector<vector<char>>& board, string& word, int x, int y, int idx,
vector<vector<bool>>& visited){
        if(idx >= word.size()) return true;
        if(x>0 && board[x-1][y]==word[idx] && !visited[x-1][y]){
            visited[x-1][y] = true;
            if(find(board, word, x-1, y, idx+1, visited)) return true;
            visited[x-1][y] = false;
        }
        if(y>0 && board[x][y-1]==word[idx] && !visited[x][y-1]){
            visited[x][y-1] = true;
            if(find(board, word, x, y-1, idx+1, visited)) return true;
            visited[x][y-1] = false;
        }
        if(x<board.size()-1 && board[x+1][y]==word[idx] && !visited[x+1][y]){
            visited[x+1][y] = true;
            if(find(board, word, x+1, y, idx+1, visited)) return true;
            visited[x+1][y] = false;
        }
        if(y<board[0].size()-1 && board[x][y+1]==word[idx] && !visited[x][y+1]){
```

```
            visited[x][y+1] = true;
            if(find(board, word, x, y+1, idx+1, visited)) return true;
            visited[x][y+1] = false;
        }
        return false;
    }

    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
        vector<string> ans;
        if(board.size() == 0 || board[0].size() == 0 || words.size() == 0) return
ans;
        int m = board.size();
        int n = board[0].size();
        //记录每个字母所有出现的位置
        vector<vector<pair<int,int>>> map = vector<vector<pair<int,int>>>(26);
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                map[board[i][j] - 'a'].push_back(make_pair(i, j));
            }
        }
        int max_len = m*n;
        for(int i = 0; i < words.size(); i++){
            if(words[i].size() > max_len) continue;//字符超长一定不行
            bool flag = true;//words[i]内的字符出现次数是否都小于等于出现在board上的次数
            for(int j = 0; j < words[i].size(); j++){
                vector<pair<int,int>> locs = map[words[i][j] - 'a'];
                if(locs.size() < count(words[i].begin(), words[i].end(), words[i]
[j])) {
                    flag == false;
                    break;
                }
            }
            if(flag){//words[i]内的字符都出现在board上
                vector<pair<int,int>> locs = map[words[i][0] - 'a'];
                for(int j = 0; j < locs.size(); j++){//对单词首字母出现的每一个位置
                    int x = locs[j].first;
                    int y = locs[j].second;
                    vector<vector<bool>> visited = vector<vector<bool>>(m,
vector<bool>(n, false));
                    visited[x][y] = true;
                    if(find(board, words[i], x, y, 1, visited)){//以该位置为起点查找
words[i]
                        ans.push_back(words[i]);
                        break;//找到一个即可
                    }
                }
            }
        }
        return ans;
    }
};
```

3. 前缀树方法

用前缀树存储words内所有字符串，这样只要某个前缀prefix在board中找不到，整个分支都不用再找了

```
//以board[x][y]为起点，寻找word[idx...]是否存在
bool find(vector<vector<char>>& board, string& word, int x, int y, int idx,
vector<vector<bool>>& visited){
        if(idx >= word.size()) return true;
        if(x>0 && board[x-1][y]==word[idx] && !visited[x-1][y]){
            visited[x-1][y] = true;
            if(find(board, word, x-1, y, idx+1, visited)) return true;
            visited[x-1][y] = false;
        }
        if(y>0 && board[x][y-1]==word[idx] && !visited[x][y-1]){
            visited[x][y-1] = true;
            if(find(board, word, x, y-1, idx+1, visited)) return true;
            visited[x][y-1] = false;
        }
        if(x<board.size()-1 && board[x+1][y]==word[idx] && !visited[x+1][y]){
            visited[x+1][y] = true;
            if(find(board, word, x+1, y, idx+1, visited)) return true;
            visited[x+1][y] = false;
        }
        if(y<board[0].size()-1 && board[x][y+1]==word[idx] && !visited[x][y+1]){
            visited[x][y+1] = true;
            if(find(board, word, x, y+1, idx+1, visited)) return true;
            visited[x][y+1] = false;
        }
        return false;
}

class Trie {//存储words的前缀树
public:
    unordered_map<char, Trie*> children;
    bool isWord;
    string val;//当前节点代表的string值
    Trie() {
        isWord = false;
        val = "";
    }
    void insert(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0){
                cur->children[word[i]] = new Trie();
                cur->children[word[i]]->val = cur->val + word[i];
            }
            cur = cur->children[word[i]];
        }
        cur->isWord = true;
    }
   //为前缀树中的每一个单词在board中寻找是否存在对应单词并返回答案
   vector<string> search(vector<vector<char>>& board, vector<vector<pair<int,int>>>
map){
        vector<string> ans;
```

```
        int m = board.size();
        int n = board[0].size();
        //层序遍历
        queue<Trie*> q;
        q.push(this);
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){//当前在队列内的所有节点对应的string值在board
中均有对应
                Trie* cur = q.front();
                q.pop();
                for(auto it = cur->children.begin(); it != cur->children.end();
it++){//遍历所有的孩子
                    //寻找孩子对应的string值是否出现在board中
                    string temp = it->second->val;
                    vector<pair<int,int>> locs = map[temp[0] - 'a'];
                    for(int j = 0; j < locs.size(); j++){//以所有出现temp[0]的位置为起
点寻找
                        int x = locs[j].first;
                        int y = locs[j].second;
                        vector<vector<bool>> visited = vector<vector<bool>>(m,
vector<bool>(n, false));
                        visited[x][y] = true;
                        if(find(board, temp, x, y, 1, visited)){//找到了
                            if(it->second->isWord)//是单词就加入ans中
                                ans.push_back(temp);
                            q.push(it->second);//只有找到对应的值的节点才能入队列，其余分
支被剪枝
                            break;
                        }
                    }
                }
            }
        }
        return ans;
    }
};
class Solution {
public:
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
        vector<string> ans;
        if(board.size() == 0 || board[0].size() == 0 || words.size() == 0) return
ans;
        int m = board.size();
        int n = board[0].size();
        vector<vector<pair<int,int>>> map = vector<vector<pair<int,int>>>(26);
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                map[board[i][j] - 'a'].push_back(make_pair(i, j));
            }
        }
        Trie* root = new Trie();
        int max_len = m*n;
```

```
        for(int i = 0; i < words.size(); i++){
            if(words[i].size() > max_len) continue;//字符超长一定不行，不用加入前缀树
            root->insert(words[i]);
        }
        return root->search(board, map);
    }
};
```

Tip：对于存在于board中的单词会反复多次调用find，因此可以尝试优化每次记录下各个前缀find后的结果，再找下一个字符时继续找————————或者不用层序遍历，用前序遍历写递归?

4. 最优解法

构造words的前缀树，然后以board的每一个位置为起点进行dfs，在前缀树里找对应

```
struct tireNode{
    tireNode* next[26]={};
    int whichWord=-1;
};
class Solution {
public:
    int rowSize;
    int colSize;
    char *corMap;
    tireNode head;
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
        colSize=board.size();
        if(colSize<=0) return vector<string>();
        rowSize=board.front().size();

        corMap=(char*)malloc(rowSize*colSize);
        memset(corMap,1,rowSize*colSize);
        vector<bool> hitMap(words.size(),false);

        int which=0;
        for(auto &word : words){
            auto tmp=&head;
            for(int i=0;i<word.length();++i){
                char x=word[i]-'a';
                if(!tmp->next[x])
                    tmp->next[x]=new tireNode();
                tmp=tmp->next[x];
            }
            tmp->whichWord=which++;
        }

        for(int i=0;i<colSize;++i)
            for(int j=0;j<rowSize;++j)
                DFS(board,i,j,&head,hitMap);

        vector<string> result;
        for(int i=0;i<hitMap.size();++i)
            if(hitMap[i]) result.push_back(words[i]);
        return result;
```

```
    }
    void DFS(vector<vector<char>>& board,int row,int col,tireNode*
current,vector<bool>& hitMap){
        corMap[row*rowSize+col]=0;
        int x=board[row][col]-'a';
        if(current->next[x]){
            current=current->next[x];
            if(current->whichWord>=0) hitMap[current->whichWord]=true;
            if(row>0&&corMap[(row-1)*rowSize+col])
                DFS(board,row-1,col,current,hitMap);
            if(col>0&&corMap[row*rowSize+col-1])
                DFS(board,row,col-1,current,hitMap);
            if(row+1<colSize&&corMap[(row+1)*rowSize+col])
                DFS(board,row+1,col,current,hitMap);
            if(col+1<rowSize&&corMap[row*rowSize+col+1])
                DFS(board,row,col+1,current,hitMap);
        }
        corMap[row*rowSize+col]=1;
    }
};
```

336.回文对（困难）

1. 题目描述

   给定一组**唯一**的单词， 找出所有**不同** 的索引对 `(i, j)`，使得列表中的两个单词， `words[i] + words[j]` ，可拼接成回文串。

   示例 1:

   ```
   输入: ["abcd","dcba","lls","s","sssll"]
   输出: [[0,1],[1,0],[3,2],[2,4]]
   解释: 可拼接成的回文串为 ["dcbaabcd","abcddcba","slls","llssssll"]
   ```

   示例 2:

   ```
   输入: ["bat","tab","cat"]
   输出: [[0,1],[1,0]]
   解释: 可拼接成的回文串为 ["battab","tabbat"]
   ```

2. 简单实现

   - 将所有的单词放入前缀树，注意对于空字符串，它与所有的回文字符串构成两对回文对，可直接加入答案
   - 依次遍历所有非空字符串word，将其取逆序，在前缀树中搜索匹配，一共有三种情况：
     - 前缀树中某个单词匹配为word的前缀，则如果word剩余的部分是回文的，那么该词和word构成回文对（word是逆序，故wrod在后）
     - 前缀树中某个词正好与word完全匹配，此时会产生两对回文对（见示例2），但为了防止多次遍历重复添加，这里只添加其中一对即可

- word匹配为前缀树中某些字符串的前缀，则继续找到所有以word为前缀的字符串，如果其剩余部分为回文，则该词和word构成回文对（word是逆序，故wrod在后）

```
//判断是否回文
bool judge(string s){
    int l = 0, r = s.size()-1;
    while(l < r){
        if(s[l] != s[r]) return false;
        l++;
        r--;
    }
    return true;
}

class Trie {
public:
    unordered_map<char, Trie*> children;
    int idx;//节点对应字符串在words中的idx
    Trie() {
        idx = -1;
    }
    void insert(string word, int idx) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0)
                cur->children[word[i]] = new Trie();
            cur = cur->children[word[i]];
        }
        cur->idx = idx;
    }
    //找到所有以root为起点的字符串，保存其idx于ans中
    void find(Trie* root, vector<int>& ans){
        if(root->idx >= 0) ans.push_back(root->idx);
        for(auto it = root->children.begin(); it != root->children.end(); it++)
            find(it->second, ans);
    }
    //寻找与reverse(word)相对应的回文对，idx为word在words中的idx
    void search(string word, int idx, vector<string>& words, vector<vector<int>>&
ans) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->idx >= 0){//word[0..i-1]已经匹配某个单词，检查剩余部分是否构成回文
                string temp = word.substr(i, word.size()-i);
                if(cur->idx!=idx && judge(temp)) ans.push_back({cur->idx, idx});
            }
            if(cur->children.count(word[i]) <= 0)//至此无匹配了，返回
                return;
            cur = cur->children[word[i]];
        }
        if(cur->idx!=idx && cur->idx >= 0) //有单词正好和word互逆，就像示例中的0、1一样
            ans.push_back({cur->idx, idx}); //为防止重复添加，这里只加入其中一种情况
        if(cur->children.size() > 0){//word已经匹配，但前缀树之后还有字符
            vector<int> idxs;
```

```
            //找到所有以word为前缀的字符串，保存其idx于idxs中
            for(auto it = cur->children.begin(); it != cur->children.end(); it++)
                find(it->second, idxs);
            for(int i = 0; i < idxs.size(); i++){//判断是否与word构成回文
                string temp = words[idxs[i]];
                temp = temp.substr(word.size(), temp.size()-word.size());//去掉word
前缀的部分
                if(idxs[i]!=idx && judge(temp)) ans.push_back({idxs[i], idx});
            }
        }
    }
};

class Solution {
public:
    vector<vector<int>> palindromePairs(vector<string>& words) {
        vector<vector<int>> ans;
        if(words.size() <= 1) return ans;
        Trie* root = new Trie();
        int null = -1;
        for(int i = 0; i < words.size(); i++){//构造前缀树
            if(words[i] != "")
                root->insert(words[i], i);
            else{//为空字符串
                for(int j = 0; j < words.size(); j++){
                    if(j != i && judge(words[j])){//自身构成回文
                        ans.push_back({i, j});
                        ans.push_back({j, i});
                    }
                }
                null = i; //记录空字符串idx，不再遍历
            }
        }
        for(int i = 0; i < words.size(); i++){
            if(i == null) continue;//跳过空字符串
            string cur = words[i];
            reverse(cur.begin(), cur.end());
            root->search(cur, i, words, ans);
        }
        return ans;
    }
```

3. 最优解法

   与我的想法相同，用map代替前缀树

```
class Solution {
public:
    bool IsPalindrom(string& s, int start, int end) {//判断s[start...end]是否构成回文
串
        while (start < end) {
            if (s[start++] != s[end--]) {
                return false;
```

```
            }
        }
        return true;
    }

    vector<vector<int>> palindromePairs(vector<string>& words) {
        vector<vector<int>> res;
        unordered_map<string, int> lookup;//相当于前缀树的作用，存放所有字符串
        for (int i = 0; i < words.size(); ++i) {
            lookup[words[i]] = i;
        }

        for (int i = 0; i < words.size(); ++i) {
            for (int j = 0; j <= words[i].length(); ++j) {
                if (IsPalindrom(words[i], j, words[i].length() - 1)) {//s[j...]构成
回文
                    //只需看有没有字符串与s[0...j-1]构成回文
                    string suffix = words[i].substr(0, j);
                    reverse(suffix.begin(), suffix.end());
                    if (lookup.find(suffix) != lookup.end() && i !=
lookup[suffix])//找到了
                        res.push_back({i, lookup[suffix]});
                }
                if (j > 0 && IsPalindrom(words[i], 0, j - 1)) {//s[0...j-1]构成回文
                    //只需看有没有字符串与s[j...]构成回文
                    string prefix = words[i].substr(j);
                    reverse(prefix.begin(), prefix.end());
                    if (lookup.find(prefix) != lookup.end() && lookup[prefix] != i)
//找到了
                        res.push_back({lookup[prefix], i});
                }
            }
        }
        return res;
    }
};
```