212. Word Search II

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

```
class TrieNode {
  HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();
  String word = null;
  public TrieNode() {}
}

class Solution {
  char[][] _board = null;
  ArrayList<String> _result = new ArrayList<String>();

  public List<String> findWords(char[][] board, String[] words) {

    // Step 1). Construct the Trie
    TrieNode root = new TrieNode();

    for (String word : words) {
      TrieNode node = root;

      for (Character letter : word.toCharArray()) {
        if (node.children.containsKey(letter)) {
          node = node.children.get(letter);
        } else {
          TrieNode newNode = new TrieNode();
          node.children.put(letter, newNode);
          node = newNode;
        }
      }

      node.word = word;  // store words in Trie
    }

    this._board = board;
    // Step 2). Backtracking starting for each cell in the board
    for (int row = 0; row < board.length; ++row) {
      for (int col = 0; col < board[row].length; ++col) {
        if (root.children.containsKey(board[row][col])) {
          backtracking(row, col, root);
        }
      }
    }

    return this._result;
  }

  private void backtracking(int row, int col, TrieNode parent) {
    Character letter = this._board[row][col];
    TrieNode currNode = parent.children.get(letter);
```

```
    // check if there is any match
    if (currNode.word != null) {
      this._result.add(currNode.word);
      currNode.word = null;
    }

    // mark the current letter before the EXPLORATION
    this._board[row][col] = '#';

    // explore neighbor cells in around-clock directions: up, right, down,
left
    int[] rowOffset = {-1, 0, 1, 0};
    int[] colOffset = {0, 1, 0, -1};

    for (int i = 0; i < 4; ++i) {
      int newRow = row + rowOffset[i];
      int newCol = col + colOffset[i];

      if (newRow < 0 || newRow >= this._board.length || newCol < 0
          || newCol >= this._board[0].length) {
        continue;
      }

      if (currNode.children.containsKey(this._board[newRow][newCol])) {
        backtracking(newRow, newCol, currNode);
      }
    }

    // End of EXPLORATION, restore the original letter in the board.
    this._board[row][col] = letter;

    // Optimization: incrementally remove the leaf nodes
    if (currNode.children.isEmpty()) {
      parent.children.remove(letter);
    }

  }
}
```