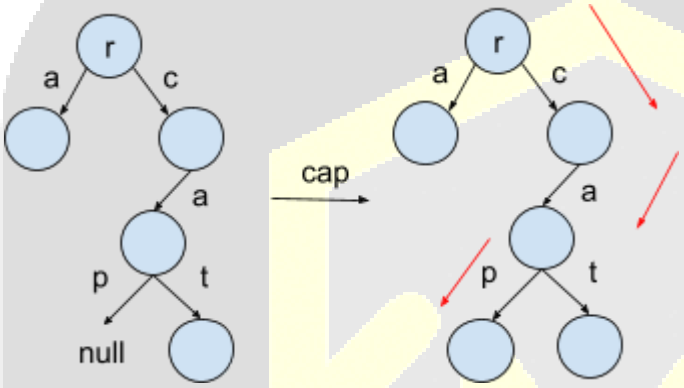
建議對演算法與數據結構已經有足夠熟悉的學員跳過此章節

主要原因是我們課程的設計核心為實際去模擬當你真實遇到該問題時會有什麼樣的想法，但講義內容其實已經有部分暗示解法，但實際遇到問題時並不會知道接下來的題目會用到解法的方向。

Trie 字典樹

當題目有大量字串然後我們需要快速搜尋多個字串時，且題目跟字串前綴或字典排序有關，使用 Trie 會比較方便處理（比起使用 hash table，Trie 在存字串上更省記憶體空間且搜尋字串時間為 $O(m)$，m 為字串長度）。Trie 的好處就是只要一開始先花時間建立好，後續多個 request 就能夠短時間處理完畢。

建立 Trie 一開始會先設一個 root，然後 root 會指向 26 個字母，然後再將那些字母指向 null。接下來再一一把題目給的字串慢慢放進 Trie 裡。從 root 往下遍歷，如果字母有出現繼續走，沒出現的話就將原本指向 null 的點指向下一個字母。建立完包含 n 個字串的 Trie，時間複雜度是 O(總字串長度)。



Add/Search

基本上 Trie 的實作會分成 Add 和 Search 兩部分（根據題目不同可能會有其他操作），Add 是在現有的 Trie 中加入新的字串，也就是建立 Trie 時會用到，而 Search 就是在建立完 Trie 後搜尋裡面的資料。
先定義好 Trie 的 structure

```
struct Trie {
    Trie* children[26];
    for (int i = 0; i < 26; ++i) {
        children[i] = nullptr;
     }
}
```

Add

```
void insert(const string word) {
    TrieNode* curr = root;
    for (char ch : word) {
        int index = ch - 'a';
        if (curr->children[index] == nullptr) {
            curr->children[index] = new TrieNode();
        }
```

```
        curr = curr->children[index];
    }
    curr->isEndOfWord = true;
}
```

Search

```
bool search(const string word) {
    const TrieNode* curr = root;
    for (char ch : str) {
        int index = ch - 'a';
        if (curr->children[index] == nullptr) {
            return nullptr;
        }
        curr = curr->children[index];
    }
    return curr;
}
```

範例

題目會給定 n 個字串，且每個字串都會有對應的熱門排名（有第 0 名到第 n-1 名）。基於不同的前綴輸入我們要回傳他前三熱門的字串。例如字串總共有 can, cat, cut, dog, cap四個字串且熱門程度分別為 0, 1, 2, 3, 4，若輸入 c 則要回傳 can, cat, cut，但若輸入 ca 則要回傳 can, cat, cap。
我們可以先照上面建立好一個 Trie，不過現在每個 node 要有個陣列存熱門排名。我們在把每個字串放入 Trie 時，要順便把熱門排名放入他經過的所有 node。例如 cat 會經過三個 nodes 就要把 “0” 這個熱門排名都放入那些 node
之後當題目輸入某個前綴字串時，我們就可以用上面 Search 的操作從 root 走到那個前綴字串到達的 node，node 上面前三小的數字就是前三熱門的字串。例如輸入 ca 我們從 root 走到 c 再走到 a，我們可以看到最後那個點有 0, 1, 3 三個數字，因此我們就知道前熱門的字串為 can, cat, cup。

Rank: can = 0, cat = 1, cut = 2, cap = 3, a = 4