Metadata

Note_Type::Literature Note
Topics::Trie Tree
Source_Url::Wiki-Trie, LeetCode-Problem 208, 知乎-Trie應用

說明

題目簡述

實作 `Trie` 結構體(發音同 try)，並建立對應的方法：

- `insert()`：將**字串存入**結構體中
- `search()`：**搜尋特定字元**，並回傳 `boolean` 揭示是否存在於結構體中
- `startsWith()`：**搜尋是否具備特定開頭的字串**，並回傳 `boolean` 揭示是否存在於結構體中

題目範例：

Example 1:

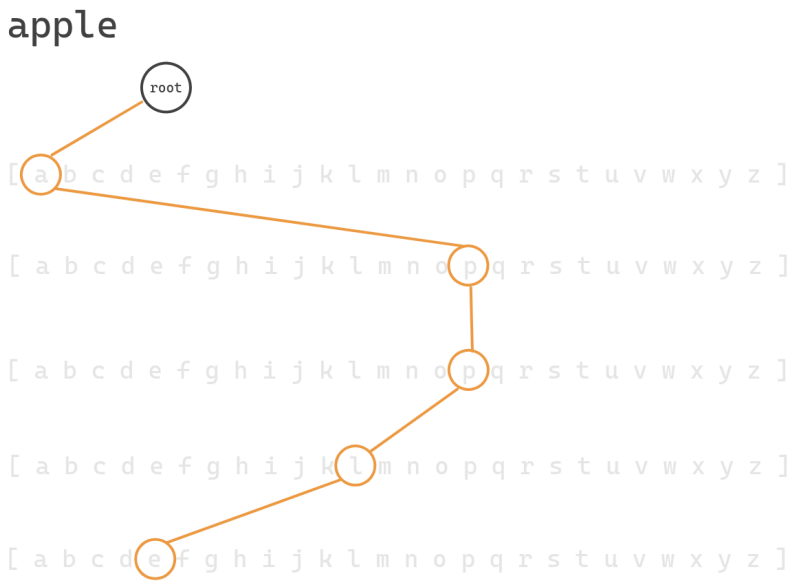
```
Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]

Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // return True
trie.search("app");     // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");     // return True
```

`Trie` 結構體組成如下：

- 具備一個 `isEnd` 成員，辨別當前節點是否為字串的最後一個字元
- 具備一個大小為26的 `Array` (英文字母的26個單字)來紀錄當前字元，並將當前的字元依據索引存入**空節點**

Trie 特性

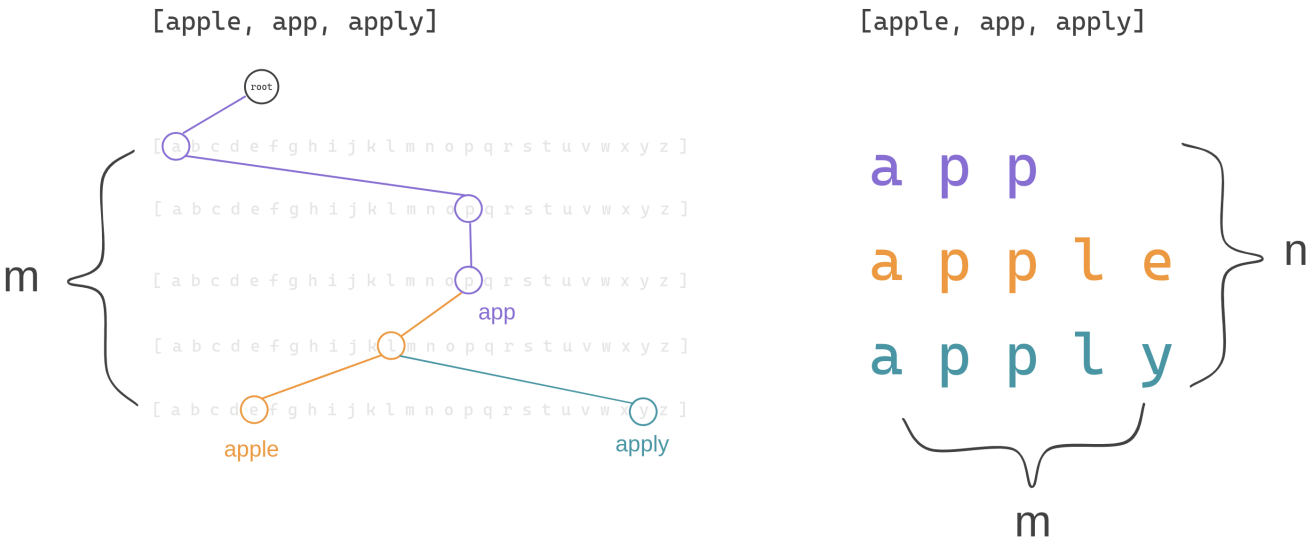
- 與一般的 `Tree` 不同，節點並不會儲存值，而是利用節點身處於陣列中的位置來辨識當前字元



- 搜尋速度相較 `Array` 查詢快

具備最長 `m` 大小字串、`n` 個字元的 `Array`，搜尋單一字串時間複雜度為 $O(m \times n)$，而使用 `Trie` 則是 $O(m)$

相同情境下，空間複雜度兩者則都是$O(m \times n)$



依據圖中案例，實際上是 `Array` 演算次數較少，不過是發生在字串筆數少的情況。當需要搜尋數量是上萬筆、上千萬筆時，性能差距則顯而易見。

解題思路

~~上網查 `Trie` 的原始碼...~~

須考慮的情境：

1. 字串結束了嗎?
2. 如何有順序地紀錄字元?

演算法設計

- 先建立一個名為 `TrieNode` 的結構體，`Trie` 的最小單位

其中包含成員：

- `isEnd`：紀錄存入的字串是否抵達結尾了，或是有沒有包含遍歷至此的字串
- `children`：紀錄看到的字元，依據英文26個字母順序填入**空節點**，也是說自己本身會提示下一個字元。

```
class TrieNode
{
public:
  // 字串結束了嗎?
  bool isEnd;
  // 如何有順序地紀錄字元?
  vector<TrieNode*> children;

  TrieNode()
  {
    isEnd = false;
    children.resize(26);
  }
};
```

- 節點組合起來便是樹，建立一個包含多筆 `TrieNode` 的結構體 `Trie`

```
class Trie
{
  TrieNode* root;
  Trie()
  {
    root = new TrieNode();
  }

  void insert(string s)
  {
    // ...
  }

  bool search(string s)
  {
    // ...
  }

  bool startsWith(string s)
  {
    // ...
  }
};
```

- `insert()`

```
void insert(string s)
{
  TrieNode* curr = root;
  for (char c : s)
  {
    int idx = c - 'a'; // 依據字母的順序填入
    if (curr->children[idx] == nullptr)
    {
      curr->children[idx] = new TrieNode();
    }
    curr = curr->children[idx];
  }
  curr->isEnd = true; // 成功儲存字串，並紀錄最後結尾
}
```

- `search()`

```
bool search(string s)
{
  TrieNode* curr = root;
  for (char c : s)
  {
    int idx = c - 'a';
    if (curr->children[idx] == nullptr)
    {
      return false;
    }
    curr = curr->children[idx];
  }
  return curr->isEnd; // `isEnd`是`boolen`同時表示有沒有搜尋至此的字串存在
}
```

- `startsWith()`

```
bool startsWith(string s)
{
  TrieNode* curr = root;
  for (char c : s)
  {
    int idx = c - 'a';
    if (curr->children[idx] == nullptr)
    {
      return false;
    }
    curr = curr->children[idx];
  }
  return true; // 與`search()`不同，只需要確認有沒有包含，不必知道節點是否為結尾
}
```