Trie 树优于哈希表的另一个理由是，随着哈希表大小增加，会出现大量的冲突，时间复杂度可能增加到 O(n)，其中 n是插入的键的数量。与哈希表相比，Trie 树在存储多个具有相同前缀的键时可以使用较少的空间。此时 Trie 树只需要 O(m)的时间复杂度，其中 m 为键长。而在平衡树中查找键值需要 O(mlogn) 时间复杂度。

应用：

1. 自动补全，eg: 谷歌的搜索建议
2. 拼写检查，eg: 文字处理软件中的拼写检查
3. IP 路由 (最长前缀匹配)，eg: 使用Trie树的最长前缀匹配算法，Internet 协议（IP）路由中利用转发表选择路径。
4. T9 (九宫格) 打字预测，eg: T9（九宫格输入），在 20 世纪 90 年代常用于手机输入
5. 单词游戏，eg: Trie 树可通过剪枝搜索空间

208. 实现 Trie (前缀树)

```
208. 实现 Trie (前缀树)
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。
```

```python
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.root = {}
        self.end_of_word = "#"

    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
        """
        node = self.root
        for char in word:
            node = node.setdefault(char, {})
```

```
        node[self.end_of_word] = self.end_of_word

    def search(self, word: str) -> bool:
        """
        Returns if the word is in the trie.
        """
        node = self.root
        for char in word:
            if char not in node:
                return False
            node = node[char]
        return self.end_of_word in node

    def startsWith(self, prefix: str) -> bool:
        """
        Returns if there is any word in the trie that starts with the given
prefix.
        """
        node = self.root
        for char in prefix:
            if char not in node:
                return False
            node = node[char]
        return True



# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```

547. 朋友圈[中等]

[1 [0726]]

547. 朋友圈
班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。

给定一个 N * N 的矩阵 M，表示班级中学生之间的朋友关系。如果M[i][j] = 1，表示已知第 i 个和 j 个学生互为朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。

示例 1:

输入:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
输出: 2
说明：已知学生0和学生1互为朋友，他们在一个朋友圈。
第2个学生自己在一个朋友圈。所以返回2。
示例 2:

输入:
[[1,1,0],
 [1,1,1],
 [0,1,1]]
输出: 1
说明：已知学生0和学生1互为朋友，学生1和学生2互为朋友，所以学生0和学生2也是朋友，所以他们三个在一个朋友圈，返回1。
注意：

N 在[1,200]的范围内。
对于所有学生，有M[i][i] = 1。
如果有M[i][j] = 1，则有M[j][i] = 1。

```python
class Solution:
    def findCircleNum(self, M: List[List[int]]) -> int:
        if not M: return 0

        n = len(M)
        p = [i for i in range(n)]

        for i in range(n):
            for j in range(n):
                if M[i][j] == 1:
                    # 遍历矩阵，合并i，j
                    self._union(p, i, j)

        return len(set([self._parent(p, i) for i in range(n)]))

    def _union(self, p, i, j):
        p1 = self._parent(p, i)
        p2 = self._parent(p, j)
        p[p2] = p1

    def _parent(self, p, i):
        root = i
        while p[root] != root:
            root = p[root]
        while p[i] != i:
            x = i; i = p[i]; p[x] = root
        return root
```