Trie 字典树

1. 字典树的数据结构
2. 字典树的核心思想
3. 字典树的基本性质

二叉搜索树：升序的序列

（前、中、后序遍历）

词频的感应 / 由前缀来推后面可能的词语

数据结构？

改变二叉树的结点里面的存储方式

二叉树 / BST：结点里存的全部的值

字典树：不再是存这里的单词本身，而是把这个字符串拆成单个单的字母，

每个字母存在这个结点里面去了

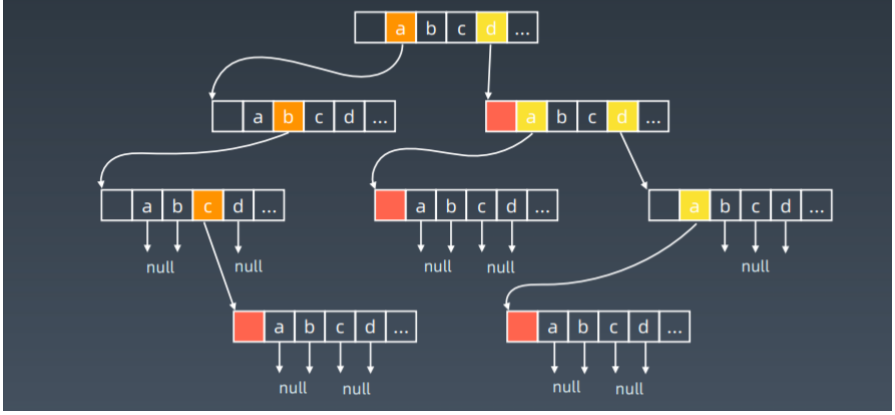
基本性质

1. 结点本身不存完整单词；

2. 从根结点到某一结点，路径上经过的字符连接起来，为该结点对应的字符串；

3. 每个结点的所有子结点路径代表的字符都不相同。

结点的内部实现



存储
用相应的字符来指向下一个结点

ASCII：255

查询很快
查的次数 → 这个单词到底有多少字符，就查多少次

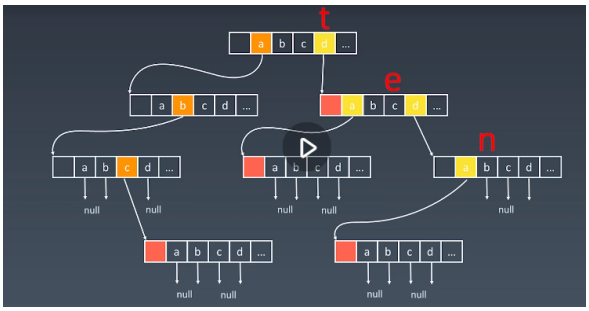
核心思想

Trie 树的核心思想是空间换时间。

利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

for

example：



Java 代码模板

```java
//Java
class Trie {
    private boolean isEnd;
    private Trie[] next;
    /** Initialize your data structure here. */
    public Trie() {
        isEnd = false;
        next = new Trie[26];
    }

    /** Inserts a word into the trie. */
    public void insert(String word) {
        if (word == null || word.length() == 0) return;
        Trie curr = this;
        char[] words = word.toCharArray();
        for (int i = 0;i < words.length;i++) {
            int n = words[i] - 'a';
            if (curr.next[n] == null) curr.next[n] = new Trie();
            curr = curr.next[n];
        }
        curr.isEnd = true;
    }


    /** Returns if the word is in the trie. */
    public boolean search(String word) {
        Trie node = searchPrefix(word);
        return node != null && node.isEnd;
    }

    /** Returns if there is any word in the trie that starts with the
given prefix. */
    public boolean startsWith(String prefix) {
        Trie node = searchPrefix(prefix);
        return node != null;
    }

    private Trie searchPrefix(String word) {
        Trie node = this;
        char[] words = word.toCharArray();
        for (int i = 0;i < words.length;i++) {
            node = node.next[words[i] - 'a'];
            if (node == null) return null;
        }
        return node;
    }
}
```