17. 字典树 Trie 树

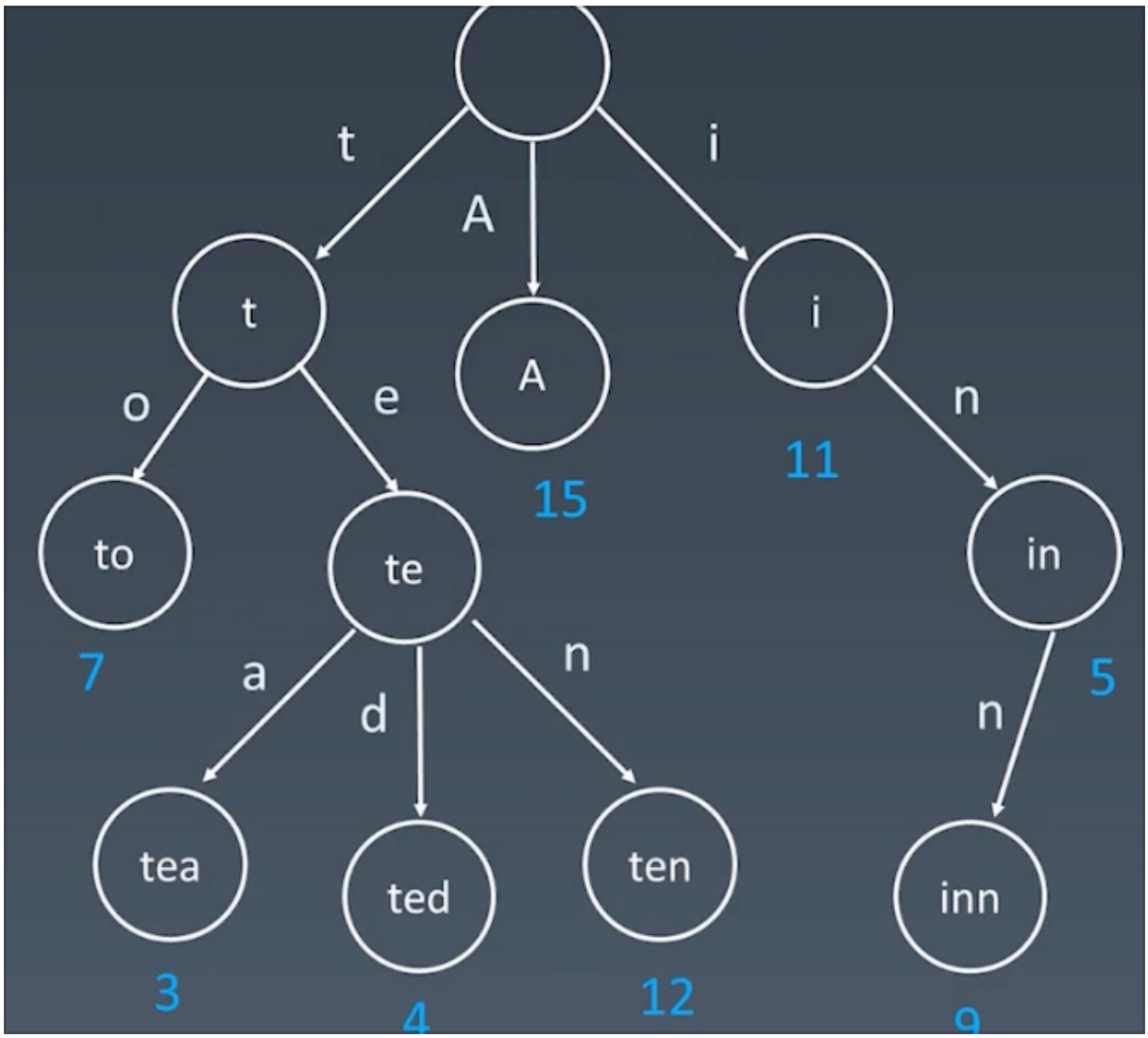
> 复习 ： 树 Tree、BFS Search、DFS Search、二叉搜索树、前中后序遍历

字典树的数据结构

字典树，即 Trie 树，又称单词查找或键树，是一种树形结构.
典型应用于统计和排序大量的字符串（但不仅限于字符串）, 所以经常被搜索引擎系统用于文本词频统计.

它的优点：最大限度地减少无谓的字符串比较，查询效率比哈希表高.

- Trie 树 结构

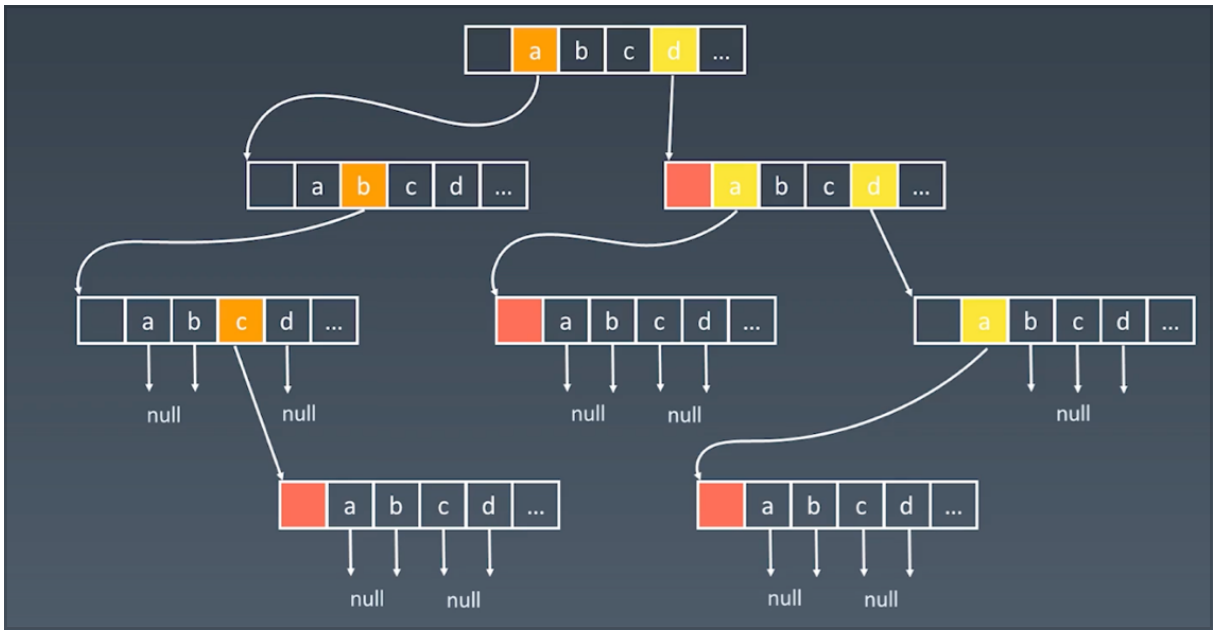


字典树的基本性质

1. 节点本身不存完整单词；
2. 从根节点到某一节点，路径上经过的字符连接起里，为该节点对应的字符串；
3. 每个节点的所有子节点路径代表的字符都不相同;
4. 统计频次：节点存储额外信息，即节点下的蓝色数字就是它被统计的计数.

节点的内部实现

用相应的字符来指向下一个节点



字典树的核心思想

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.

代码模板

```
class Trie(object):
  def __init__(self):
        self.root = {}
      self.end_of_word = "#"

    def insert(self, word):
        node = self.root
        for char in word:
            node = node.setdefault(char, {})
```

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

    def search(self, word):
        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):
        node = self.root
        for char in prefix:
            if char not in node:
                return False
            node = node[char]

        return True
```

```
class Trie {
  struct TrieNode {
      map<char, TrieNode *> child_table;
          int end;
          TrieNode() : end(0) {}
  };

public :
  /** Initialize your data structure here. */

  Trie() {
          root = new TrieNode();
  }

  /** Inserts a word into the trie. */
  void insert(string word) {
```

```
            TrieNode *curr = root;
        for (int i = 0; i < word.size(); i++) {
                if (curr->child_table.count(word[i]) == 0)
                curr->child_table[word[i]] = new TrieNode();
            curr = curr->child_table[word[i]];
        }
        curr->end = 1;
    }

    /** Returns if the word is in the trie. */    bool
    search(string word) {
        return find(word, 1);
    }
    /** Returns if there is any word in the */
    /** trie that starts with the given prefix. */
    bool startsWith(string prefix) {
        return find(prefix, 0);
    }

private:
    TrieNode *root;
    bool find(string s, int exact_match) {
            TrieNode *curr = root;

        for (int i = 0; i < s.size(); i++) {
            if (curr->child_table.count(s[i]) == 0)
                return false;
            else
                curr = curr->child_table[s[i]];
        }

        if (exact_match)
            return (curr->end) ? true : false;
        else
            return true;
    }
};
```

```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;
    }
}
```

#Algorithm/Part II : Theory/Data Structure#