

week7

1)字典树

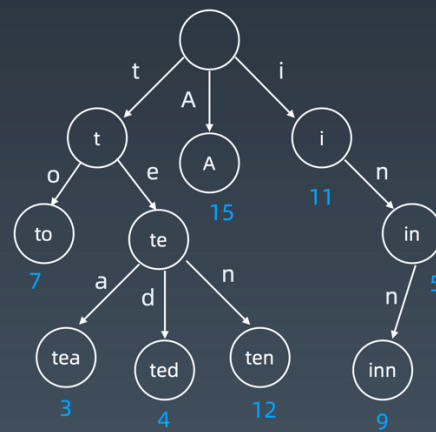
-性质：

- 1.节点本身不存单词，只存要去到的路径上的这个路径所代表的字符
- 2.从根节点到某一节点，路径上经过的字符连接起来，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

基本结构

字典树，即 Trie 树，又称单词查找树或键树，是一种树形结构。典型应用是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。

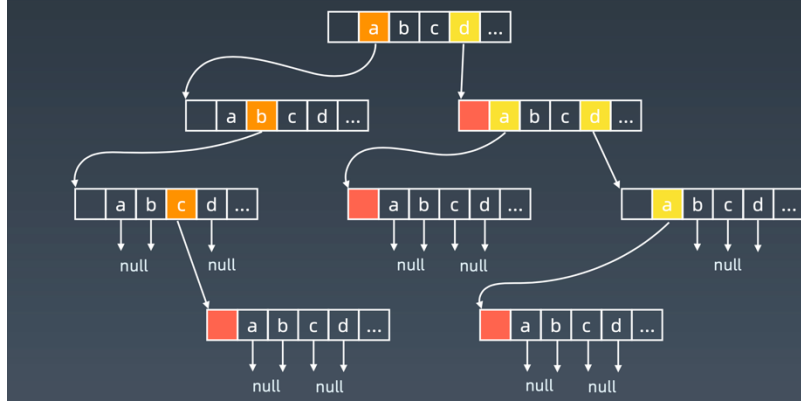
它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。



-节点可以存储额外信息（例如该节点被访问过的频次）

-核心思想：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

结点的内部实现



-todo : 实现 trie

2) 并查集, union find

- 组团+配对，用于快速判断某两个个体是否在同一个集合中
- group or not? a 和 b 是不是朋友

基本操作

- makeSet(s): 建立一个新的并查集，其中包含 s 个单元元素集合。
- unionSet(x, y): 把元素 x 和元素 y 所在的集合合并，要求 x 和 y 所在的集合不相交，如果相交则不合并。
- find(x): 找到元素 x 所在的集合的代表，该操作也可以用于判断两个元素是否位于同一个集合，只要将它们各自的代表比较一下就可以了。

-todo: 实现 union find

```
class UnionFind {
private int count = 0;
private int[] parent;
public UnionFind(int n) {
    count = n;
    parent = new int[n];
    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }
}
public int find(int p) {
    while (p != parent[p]) {
        parent[p] = parent[parent[p]];
        p = parent[p];
    }
    return p;
}
public void union(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (rootP == rootQ) return;
    parent[rootP] = rootQ;
    count--;
}
}
```

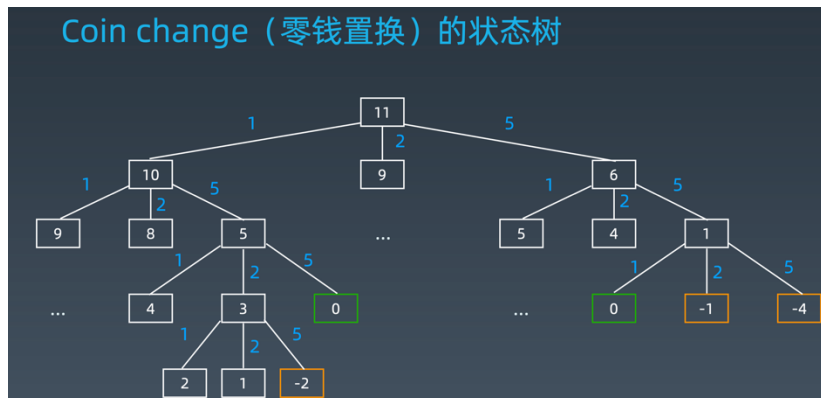
```
def init(p):
    # for i = 0 .. n: p[i] = i;
    p = [i for i in range(n)]

def union(self, p, i, j):
    p1 = self.parent(p, i)
    p2 = self.parent(p, j)
    p[p1] = p2

def parent(self, p, i):
    root = i
    while p[root] != root:
        root = p[root]
    while p[i] != i: # 路径压缩 ?
        x = i; i = p[i]; p[x] = root
    return root
```

3) 高级搜索

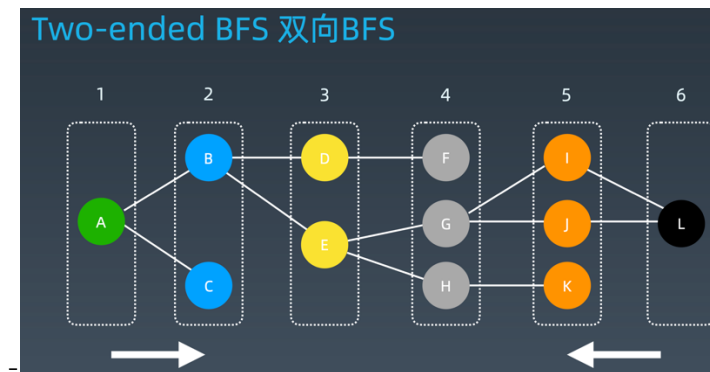
- 数形结合, 状态树
- 剪枝



-回溯

- 递归+分治+试错
- 回溯法采用试错的思想, 它尝试分步的去解决一个问题, 在分步解决问题的过程中, 当它通过尝试发现现有的分步答案不能得到有效的正确的解答时, 它将取消上一步甚至是上几步的计算, 再通过其它的可能的分步解答再次尝试寻找问题的答案

-双向 bfs



-启发式搜索 (a*) heuristic search

A* search

```
def AstarSearch(graph, start, end):  
    pq = collections.priority_queue() # 优先级 -> 估价函数  
    pq.append([start])  
    visited.add(start)  
  
    while pq:  
        node = pq.pop() # can we add more intelligence here ?  
        visited.add(node)  
  
        process(node)  
        nodes = generate_related_nodes(node)  
        unvisited = [node for node in nodes if node not in visited]  
        pq.push(unvisited)
```

4) 红黑树和 avl 树

-avl

AVL 树

1. 发明者 G. M. Adelson-Velsky 和 Evgenii Landis
2. Balance Factor (平衡因子) :
是它的左子树的高度减去它的右子树的高度 (有时相反) 。
balance factor = {-1, 0, 1}
3. 通过旋转操作来进行平衡 (四种)
4. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

-red black tree

Red-black Tree

红黑树是一种近似平衡的二叉搜索树 (Binary Search Tree)，它能够确保任何一个结点的左右子树的高度差小于两倍。具体来说，红黑树是满足如下条件的二叉搜索树：

- 每个结点要么是红色，要么是黑色
- 根结点是黑色
- 每个叶结点 (NIL结点，空结点) 是黑色的。
- 不能有相邻接的两个红色结点
- 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点。

Red-black Tree

