week07

1.字典树

数据结构

- Trie 数， 单词查找树或键树，一种树形结构
- 用于（但不仅限于）统计和排序大量的字符串

—>经常被搜索引擎系统用于文本词频统计
优点：
最大限度地减少所谓的字符串比较， 查询效率比哈希表高。

基本性质

- 结点本身不存在完整单词；
- 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；
- 每个结点的所有子节点路径代表的字符都不相同

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

2.并查集

适用场景：组团，配对问题

基本操作

- makeSet(s): 建立一个新的并查集， 其中包含s个单元素集合
- 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， 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ok

```
//C/C++
class UnionFind {
public:
    UnionFind(vector<vector<char>>& grid) {
        count = 0;
        int m = grid.size();
        int n = grid[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
```

```
                if (grid[i][j] == '1') {
                    parent.push_back(i * n + j);
                    ++count;
                }
                else {
                    parent.push_back(-1);
                }
                rank.push_back(0);
            }
        }
    }
//递归
    int find(int i) {
        if (parent[i] != i) {
            parent[i] = find(parent[i]);
        }
        return parent[i];
    }
    void unite(int x, int y) {
        int rootx = find(x);
        int rooty = find(y);
        if (rootx != rooty) {
            if (rank[rootx] < rank[rooty]) {
                swap(rootx, rooty);
            }
            parent[rooty] = rootx;
            if (rank[rootx] == rank[rooty]) rank[rootx] += 1;
            --count;
        }
    }
    int getCount() const {
        return count;
    }
private:
    vector<int> parent;
    vector<int> rank;
    int count;
};
```

3.剪枝

REVIEW 初级搜索

- 朴素搜索
  优化方向：1. 尽量重复，尽早剪枝 2. 在搜索方向上面进行加强
- 优化方式： 不重复（fibonacci），剪枝（生成括号问题）
- 搜索方向：
  DFS:depth first search 深度优先搜索
  BFS: breadth first search 广度优先搜索

双向搜索，启发式搜索

优化从而减少计算量（减少明显）

4.双向BFS

5.启发式搜索（A*）

Heuristic Search

6.AVL树

- 发明者：G.M.Adelson-Velsky 和 Evgenii Landis
- Balance Factor (平衡因子):
  左子树高度 – 右子树高度（有时候相反）balance factor ={-1, 0, 1}
- 通过旋转操作来进行平衡（四种）
  左旋，右旋，左右旋，右左旋

不足： 结点需要存储额外信息， 且调整次数频繁

7.红黑树

是一种近似平衡的二叉搜索树（Binary Search Tree），确保任一节点的左右子树的高度差小于两倍。

- 每个节点要么是红色，要么是黑色
- 根节点是黑色
- 每个叶节点（NIL节点， 空节点）是黑色的
- 不能有相邻接的两个红色节点
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点

AVL： lookups更快， 更多的内存（factors or heights）
Red Black Trees: insertation and removal operation 更快, 只需存红或者黑