躲不过的图型题目以及字典树。

拓扑排序和字典树的实现

1:拓扑排序

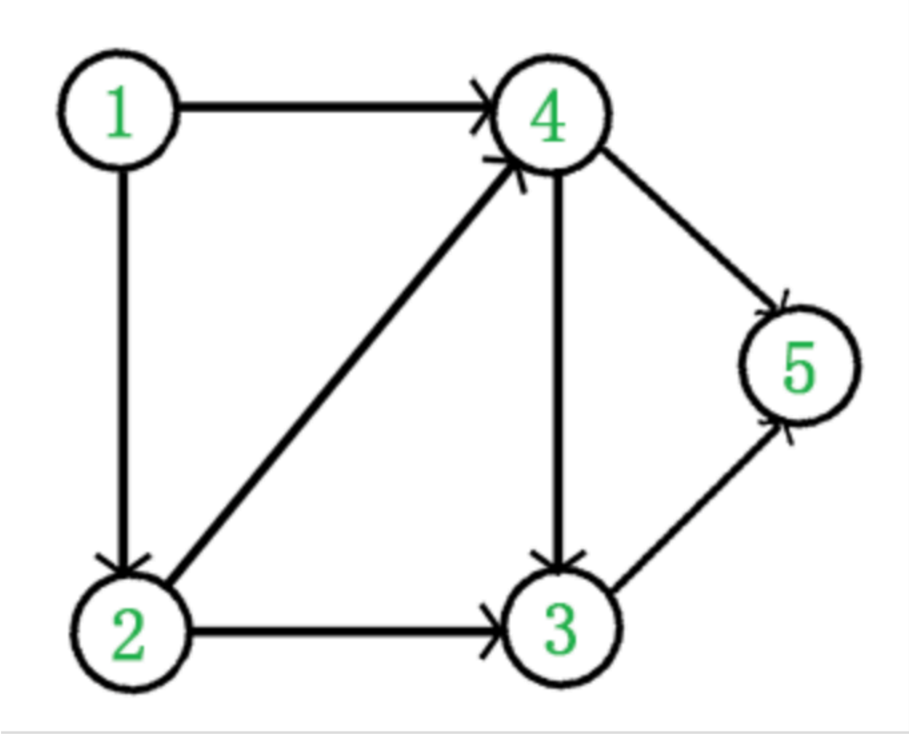
在刷题过程中，图的题目并不多，最经典的莫过于这个课程安排问题了。趁此机会学习和应用一下拓扑排序。共勉！！

1: 什么是拓扑排序

在图论中，拓扑排序是一个有向无环图(DAG)所有顶点的线性序列，且所有顶点应该满足一下条件：

- 每个顶点出现且只出现一次；
- 在图中若存在一条边从顶点A到顶点B，那么在拓扑排序的线性序列中A应该在B前面；

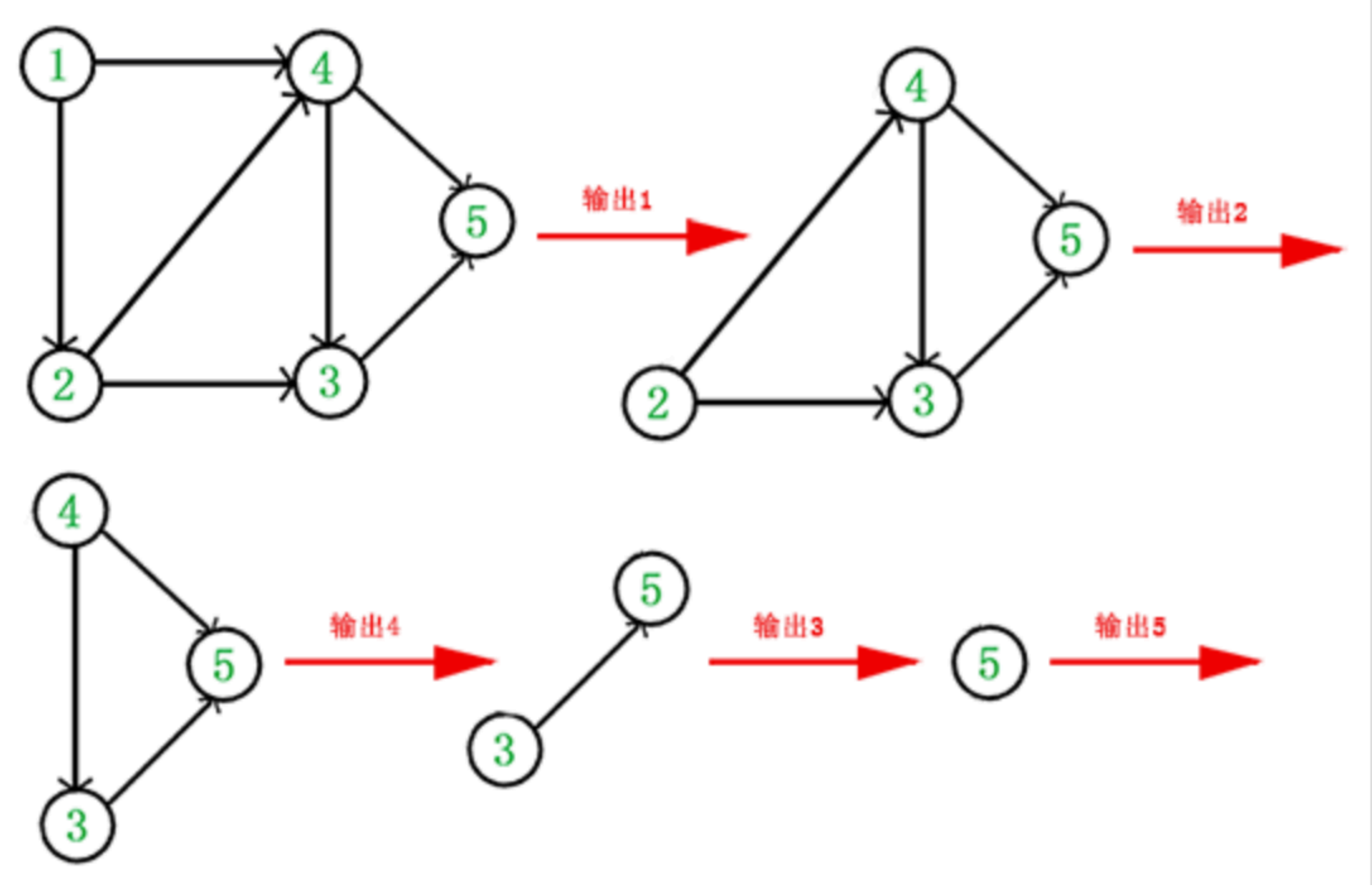
看下面这个有向无环图



拓扑排序的步骤(称之为剥洋葱)：

- 找出DAG图中入度为0(没有前驱)的顶点，并且输出这个顶点值；
- 从DAG中找出以这个顶点为前驱的所有有向边，然后删除这条有向边；
- 重复以上两个步骤，直至图为空或者不存在无前驱的顶点为止，对于后面这一种情况，说明图中存在环；

看上面那个有向无环图的拓扑排序执行顺序



最后输出的结果是｛1，2，3，4，5｝
请注意在DAG中，可能存在多个拓扑排序。

2: 拓扑排序的运用：

拓扑排序的运用通常都在具有“依赖关系”的例子当中。比如修课程中的前导课程，或者项目之间的先后顺序，这都是拓扑排序的重要运用

3: 拓扑排序的实现

拓扑排序就像剥洋葱，一层一层剥掉入度为0的顶点。
在图的表示方式中，有两种方式，一是邻接表，一个是邻接矩阵。在这里我们可以采用邻接表来实现拓扑排序。看代码：

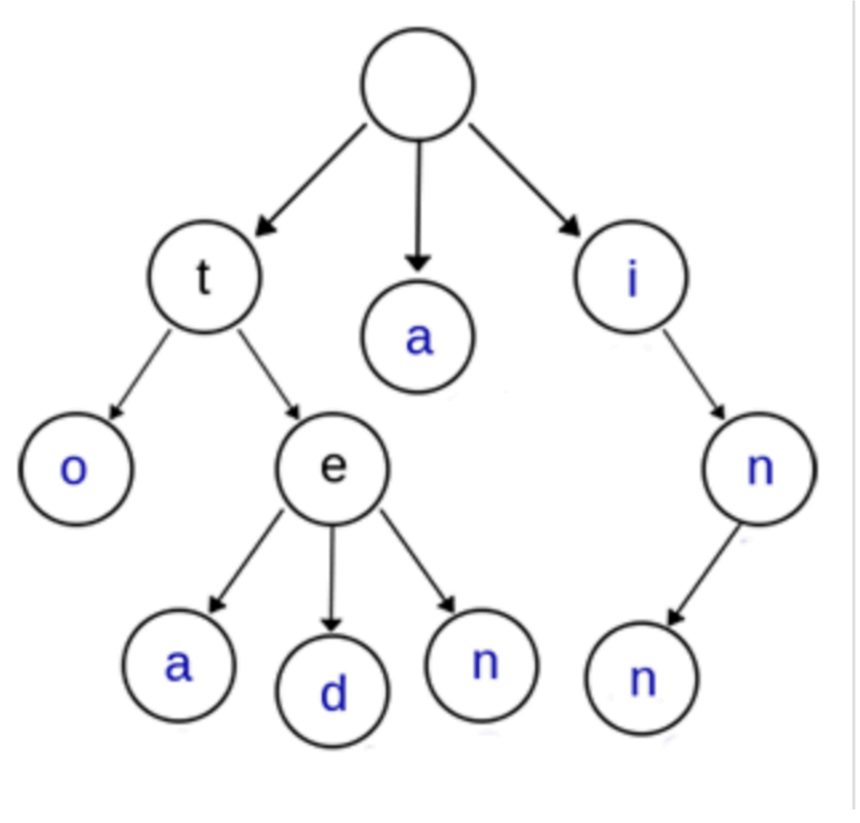
```
public class Graph {
    int V; //顶点个数
    List[] list; //邻接表
    int[] degrees;//每个顶点的度数
    Deque<Integer> queue;//维护一个顶点为0的队列
    public Graph(int V) {
        this.V = V;
        list = new List[V];
        for (int i = 0; i < V; i++) {
            list[i] = new ArrayList<Integer>();
        }
        degrees = new int[V];
        queue = new LinkedList<Integer>();
    }
    //添加有向边
    public addEdge(int v, int w) {
        list[v].add(w);
        degrees[w]++;
    }
    //拓扑排序
    public boolean topological_sort() {
        for (int i = 0; i < V; i++) {
            if (degrees[i] == 0) {
                queue.offer(i);
            }
        }
        int count = 0;
        while (!queue.isEmpty()) {
            int cur = queue.poll();
            System.out.println(cur);
            count++;
            List<Integer> adj = list[cur];
            for (Integer n : adj) {
                if (--degrees[n] == 0) {
                    queue.offer(n);
                }
            }
        }
        if (count != V) {
            return false;
        }else {
            return true;
        }
    }
}
上面代码定义了一个图，并且给出了添加有向边和拓扑排序的代码。看拓扑排序的过程，其实就是维护了一个用于储存入度为0的顶点的队列，原理很简单。
```

2: 字典树的原理以及实现

现在来继续看字典树——Trie。

1: 什么是字典树

字典树又被称为前缀树(Prefix Tree), 单词查找树，或者键树。是一种多叉树结构。看下图：



从这幅图可以看出字典树的性质：

- 字典树的根节点不包含字符
- 从根节点开始到字典树中到**某个节点**为止，连成一个字符串
- 每个节点的字节点所包含的字符都不相同

从上图我们还可以看出，有些字符有颜色，而有的字符没有颜色。那些有颜色的字符就是我们说的某个节点，证明从根节点开始到这个节点所构成的字符串是属于这个字典树里面的合法多字符串。因此上幅图所代表的字符串是：
{“to”，”tea”, “ted”, “ten”, “a”, “i”, “in”, “inn”}，并不存在“t”和”te”这样的字符串。

2: 字典树的优缺点

字典树的核心思想就是**用空间换取时间**，利用公共前缀来提高字符串的查找和插入。

- 优点
    - 插入和查询的效率都是O(n)，n为查询或者插入字符串的长度。请注意哈希表的查询效率是和哈希函数有很大关联的，当冲突严重时，查询效率不一定比字典树好；
    - 字典树中不同的关键字不会引起碰撞，因为他们代表的是不同的字符串；
    - 字典树可以对字符串按照字典序排序
- 缺点
    - 耗费空间较大；
    - 当哈希函数足够好时，查询效率不如哈希表

3: 字典树当运用

- 检索／查询
  利用字典树进行查询是其最基本的操作。

```
public class TrieNode {
    boolean isKey; //是否为关键字
    TrieNode[] subNodes;
    public TrieNode(boolean isKey) {
        this.isKey = isKey;
        subNodes = new TrieNode[26];
    }
}
```

  字典树查询的基本步骤就是：

    - 在字典树中查找每个字符，如果但凡出现不匹配，查询失败；
    - 如果沿路查询匹配成功，还要判定最后一个字符是否为关键字。如果是，查询成功；否则查询不成功；

- 词频统计

```
public class TrieNode {
    int count;
    TrieNode[] subNodes;
    public TrieNode() {
        count = 0;
        subNodes = new TrieNode[26];
    }
}
```

- 字符串排序
  字符串按照字典序排序很简单，因为所有节点的字节点都是已经排序好的，所以我们只要前序遍历字典树然后输出路径上的所有关键字就好了。
- 前缀匹配
  前缀匹配也是同样，找到公共前缀路径，然后输出所有在这条路径之后的关键字即可。

4: 字典树的实现

看代码：

```
public class Node {
    int count;
    Node[] children;
    public Node() {
        count = 0;
        children = new Node[26];
    }
}

public void insert(Node root, char[] key) {
    Node node = root;
    for (int i = 0; i < key.length; i++) {
        if (node.children[key[i] - 'a'] == null) {
            node.children[key[i] - 'a'] = new Node();
        }
        node = node.children[key[i] - 'a'];
    }
    node.count++;
}

public int search(Node root, char[] key) {
    Node node = root;
    for (int i = 0; i < key.length; i++) {
        if (node.children[key[i] - 'a'] == null) {
            return 0;
        }
        node = node.children[key[i] - 'a'];
    }
    return node.count;
}
以上代码经过测试
```