12讲树的深度优先搜索（下）：如何才能高效率地查字典

你好，我是黄申。今天咱们继续聊前缀树。

上节结尾我给你留了道思考题：如何实现前缀树的构建和查询？如果你动手尝试之后，你会发现，这个案例的实现没有我们前面讲的那些排列组合这么直观。

这是因为，从数学的思想，到最终的编程实现，其实需要一个比较长的流程。我们首先需要把问题转化成数学中的模型，然后使用数据结构和算法来刻画数学模型，最终才能落实到编码。

而在前缀树中，我们需要同时涉及树的结构、树的动态构建和深度优先搜索，这个实现过程相对比较复杂。所以，这节我就给你仔细讲解一下，这个实现过程中需要注意的点。只要掌握这些点，你就能轻而易举实现深度优先搜索。

如何使用数据结构表达树？

首先，我想问你一个问题，什么样的数据结构可以表示树？

我们知道，计算机中最基本的数据结构是数组和链表。数组适合快速地随机访问。不过，数组并不适合稀疏的数列或者矩阵，而且数组中元素的插入和删除操作也比较低效。相对于数组，链表的随机访问的效率更低，但是它的优势是，不必事先规定数据的数量，表示稀疏的数列或矩阵时，可以更有效地利用存储空间，同时也利于数据的动态插入和删除。

我们再来看树的特点。树的结点及其之间的边，和链表中的结点和链接在本质上是一样的，因此，我们可以模仿链表的结构，用编程语言中的指针或对象引用来构建树。

除此之外，我们其实还可以用二维数组。用数组的行或列元素表示树中的结点，而行和列共同确定了两个树结点之间是不是存在边。可是在树中，这种二维关系通常是非常稀疏的、非常动态的，所以用数组效率就比较低下。

基于上面这些考虑，我们可以设计一个TreeNode类，表示有向树的结点和边。这个类需要体现前缀树结点最重要的两个属性。

- 这个结点所代表的字符，要用label变量表示。
- 这个结点有哪些子结点，要用sons哈希映射表示。之所以用哈希，是为了便于查找某个子结点（或者说对应的字符）是否存在。

另外，我们还可以用变量prefix表示当前结点之前的前缀，用变量explanation表示某个单词的解释。和之前一样，为了代码的简洁，所有属性都用了public，避免读取和设置类属性的代码。

这里我写了一段TreeNode类的代码，来表示前缀树的结点和边，你可以看看。

```
/**
* @Description: 前缀树的结点
*
*/

 public class TreeNode {

  public char label;  // 结点的名称，在前缀树里是单个字母
  public HashMap<Character, TreeNode> sons = null; // 使用哈希映射存放子结点。哈希便于确认是否已经添加过某个字母对应的结点
  public String prefix = null;   // 从树的根到当前结点这条通路上，全部字母所组成的前缀。例如通路b->o->y，对于字母o结点而言，
  public String explanation = null;  // 词条的解释

  // 初始化结点
  public TreeNode(char l, String pre, String exp) {
   label = l;
   prefix = pre;
   explanation = exp;
   sons = new HashMap<>();

  }

 }
```

说到这里，你可能会好奇，为什么只有结点的定义，而没有边的定义呢？实际上，这里的有向边表达的是父子结点之间的关系，我把这种关系用sons变量来存储父结点。

需要注意的是，我们需要动态地构建这棵树。每当接收一个新单词时，代码都需要扫描这个单词的每个字母，并使用当前的前缀树进行匹配。如果匹配到某个结点，发现相应的字母结点并不存在，那么就建立一个新的树结点。这个过程不好理解，我也写了几行代码，你可以结合来看。其中，str表示还未处理的字符串，parent表示父结点。

```
// 处理当前字符串的第一个字母
char c = str.toCharArray()[0];
TreeNode found = null;

// 如果字母结点已经存在于当前父结点之下，找出它。否则就新生成一个
if (parent.sons.containsKey(c)) {
 found = parent.sons.get(c);
} else {
 TreeNode son = new TreeNode(c, pre, "");
 parent.sons.put(c, son);
 found = son;
}
```

如何使用递归和栈实现深度优先搜索?

构建好了数据结构，我们现在需要考虑，**什么样的编程方式可以实现对树结点和边的操作?**

仔细观察前缀树构建和查询，你会发现这两个不断重复迭代的过程，都可以使用递归编程来实现。换句话说，**深度优先搜索的过程和递归调用在逻辑上是一致的**。

我们可以把函数的嵌套调用，看作访问下一个连通的结点；把函数的返回，看作没有更多新的结点需要访问，回溯到上一个结点。在之前的案例中，我已经讲过很多次递归编程的例子，这里我就不列举代码细节了。如果忘记的话，你可以回去前面章节复习一下。

在查询的过程中，至少有三种情况是无法在字典里找到被查的单词的。于是，我们需要在递归的代码中做相应的处理。

第一种情况：被查单词所有字母都被处理完毕，但是我们仍然无法在字典里找到相应的词条。

每次递归调用的函数开始，我们都需要判断待查询的单词，看看是否还有字母需要处理。如果没有更多的字母需要匹配了，那么再确认一下当前匹配到的结点本身是不是一个单词。如果是，就返回相应的单词解释，否则就返回查找失败。对于结点是不是一个单词，你可以使用Node类中的explanation变量来进行标识和判断，如果不是一个存在的单词，这个变量应该是空串或者Null值。

第二种情况：搜索到前缀树的叶子结点，但是被查单词仍有未处理的字母，就返回查找失败。

我们可以通过结点对象的sons变量来判断这个结点是不是叶子结点。如果是叶子结点，这个变量应该是空的HashMap，或者Null值。

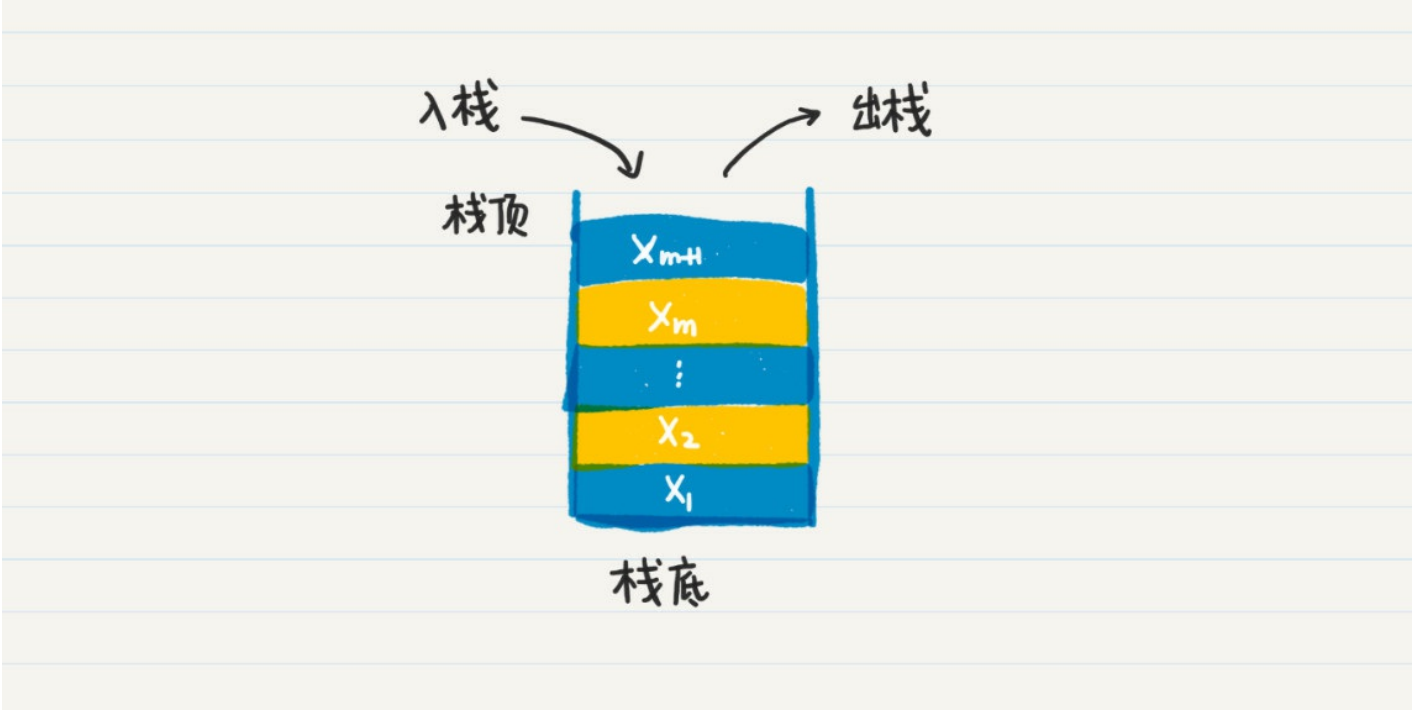
第三种情况：搜索到中途，还没到达叶子结点，被查单词也有尚未处理的字母，但是当前被处理的字母已经无法和结点上的label匹配，返回查找失败。是不是叶子仍然通过结点对象的sons变量来判断。

好了，现在你已经可以很方便地在字典里查找某个单词，看看它是否存在，或者看看它的解释是什么。我这里又有一个新的问题了：**如果我想遍历整个字典中所有的单词，那该怎么办呢?**

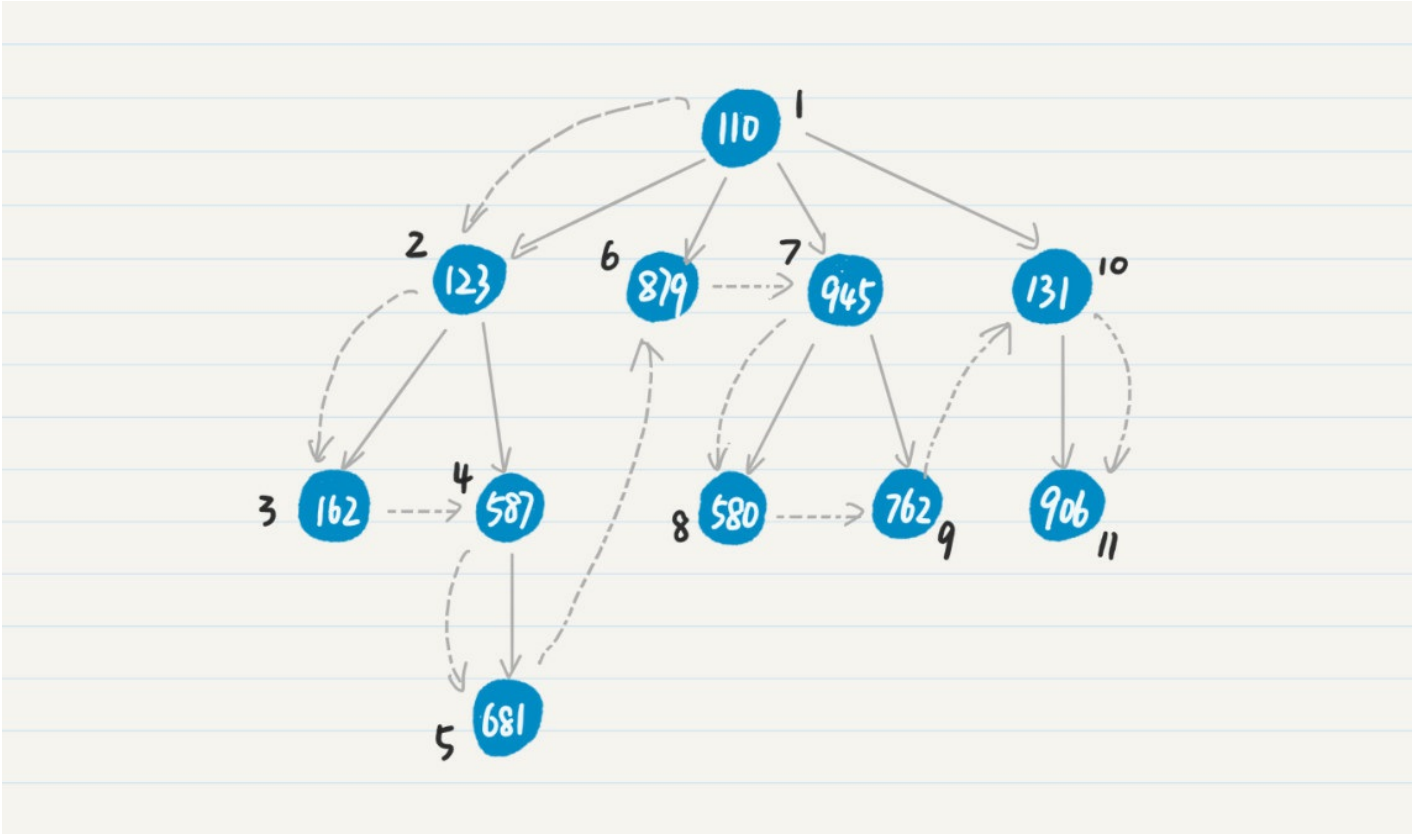
仔细观察一下，你应该能发现，查找一个单词的过程，其实就是在有向树中，找一条从树的根到代表这个单词的结点之通路。那么如果要遍历所有的单词，就意味着我们要找出从根到所有代表单词的结点之通路。所以，在每个结点上，我们不再是和某个待查询单词中的字符进行比较，而是要遍历该结点所有的子结点，这样才能找到所有可能的通路。我们还可以用递归来实现这一过程。

尽管函数递归调用非常直观，可是也有它自身的弱点。函数的每次嵌套，都可能产生新的变量来保存中间结果，这可能会消耗大量的内存。所以这里我们可以用一个更节省内存的数据结构，栈（Stack）。

栈的特点是先进后出（First In Last Out），也就是，最先进入栈的元素最后才会得到处理。我画了一张元素入栈和出栈的过程图，你可以看看。



为什么栈可以进行深度优先搜索呢？你可以先回顾一下上一节，我解释深度优先搜索时候的例子。为了方便你回想，我把图放在这里了。



然后，我们用栈来实现一下这个过程。

第1步，将初始结点110压入栈中。

第2步，弹出结点110，搜出下一级结点123、879、945和131。

第3步，将结点123、879、945和131压入栈中。

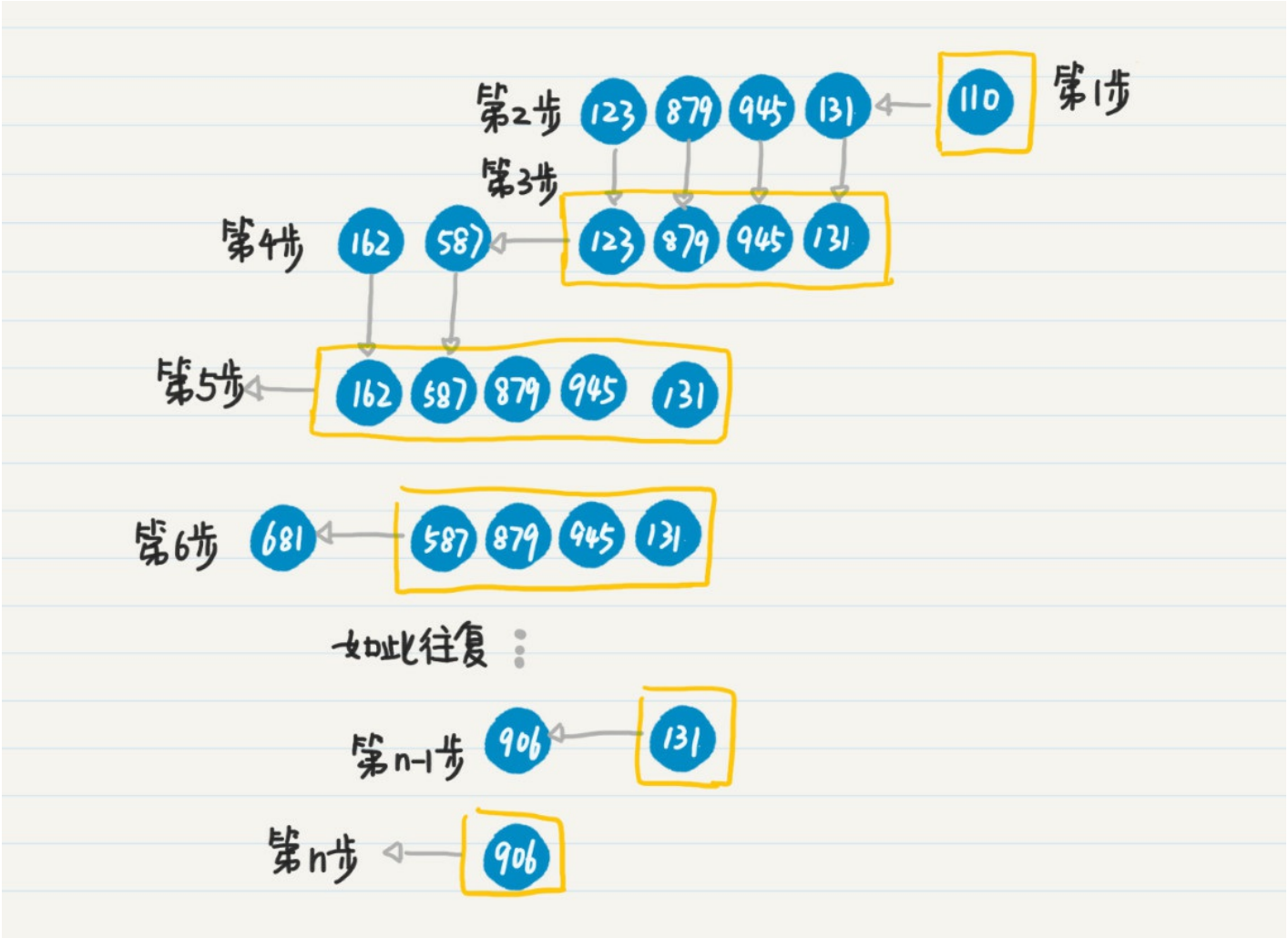
第4步，重复第2步和第3步弹出和压入的步骤，处理结点123，将新发现结点162和587压入栈中。

第5步，处理结点162，由于162是叶子结点，所以没有发现新的点。第6步，重复第2和第3步，处理结点587，将新发现结点681压入栈中。

……

第n-1步，重复第2和第3步，处理结点131，将新发现结点906压入栈中。

第n步，重复第2和第3步，处理结点906，没有发现新的结点，也没有更多待处理的结点，整个过程结束。



从上面的步骤来看，栈先进后出的特性，可以模拟函数的递归调用。实际上，计算机系统里的函数递归，在内部也是通过栈来实现的。如果我们不使用函数调用时自动生成的栈，而是手动使用栈的数据结构，就能始终保持数据的副本只有一个，大大节省内存的使用量。

用TreeNode类和栈实现深度优先搜索的代码我写出来了，你可以看看。

```
// 使用栈来实现深度优先搜索
public void dfsByStack(TreeNode root) {

  Stack<TreeNode> stack = new Stack<TreeNode>();
   // 创建堆栈对象，其中每个元素都是TreeNode类型
  stack.push(root);  // 初始化的时候，压入根结点

  while (!stack.isEmpty()) { // 只要栈里还有结点，就继续下去

 TreeNode node = stack.pop(); // 弹出栈顶的结点

 if (node.sons.size() == 0) {
  // 已经到达叶子结点了，输出
  System.out.println(node.prefix + node.label);
 } else {
  // 非叶子结点，遍历它的每个子结点
  Iterator<Entry<Character, TreeNode>> iter
   = node.sons.entrySet().iterator();

  // 注意，这里使用了一个临时的栈stackTemp
  // 这样做是为了保持遍历的顺序，和递归遍历的顺序是一致的
  // 如果不要求一致，可以直接压入stack
  Stack<TreeNode> stackTemp = new Stack<TreeNode>();
  while (iter.hasNext()) {
   stackTemp.push(iter.next().getValue());
  }
  while (!stackTemp.isEmpty()) {
   stack.push(stackTemp.pop());
  }
 }
  }

}
```

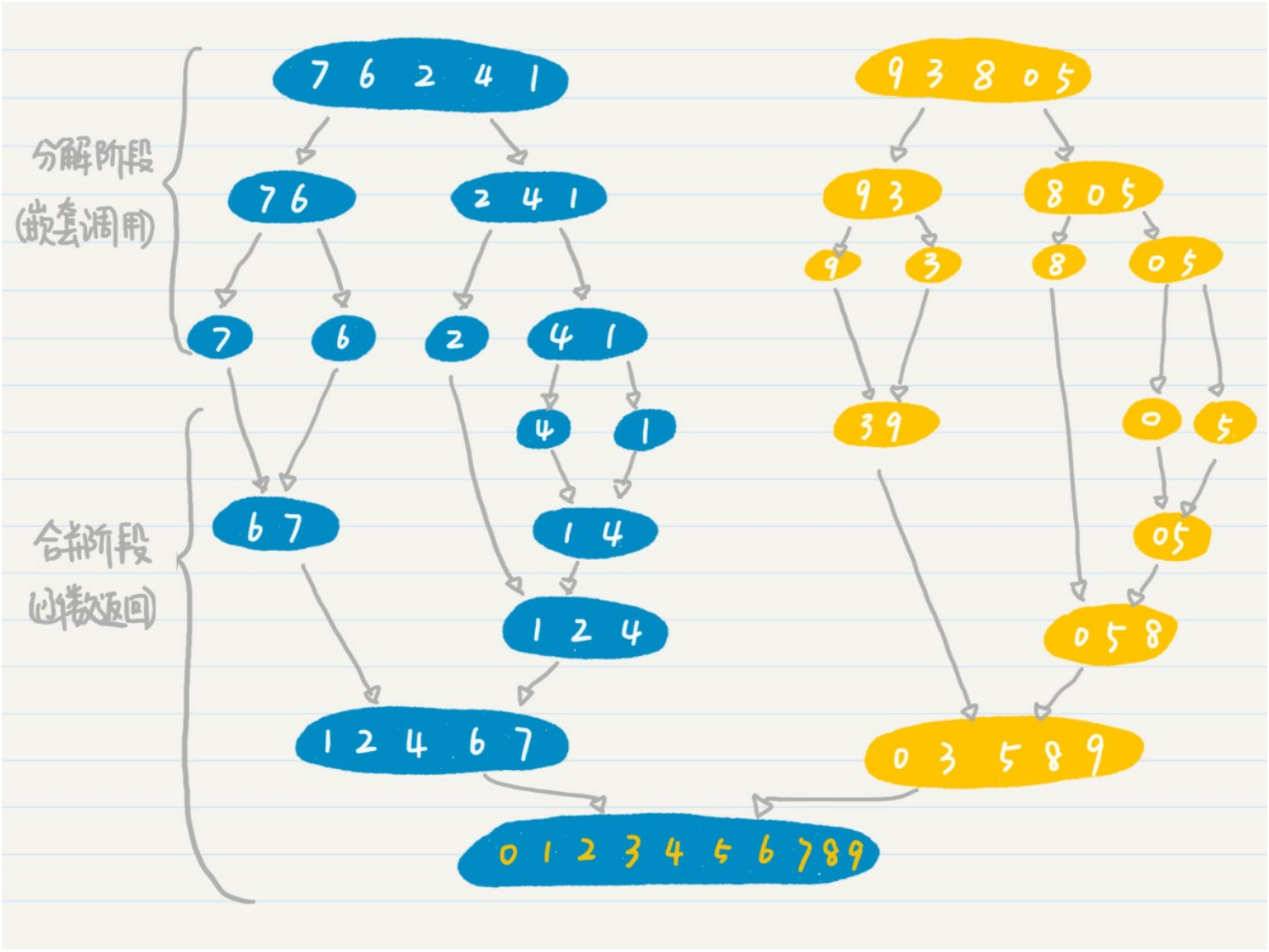
这里面有个细节需要注意一下。当我们把某个结点的子结点压入栈的时候，由于栈“先进后出”的特性，会导致子结点的访问顺序，和递归遍历时子结点的访问顺序相反。如果你希望两者保持一致，可以用一个临时的栈stackTemp把子结点入栈的顺序颠倒过来。

小结

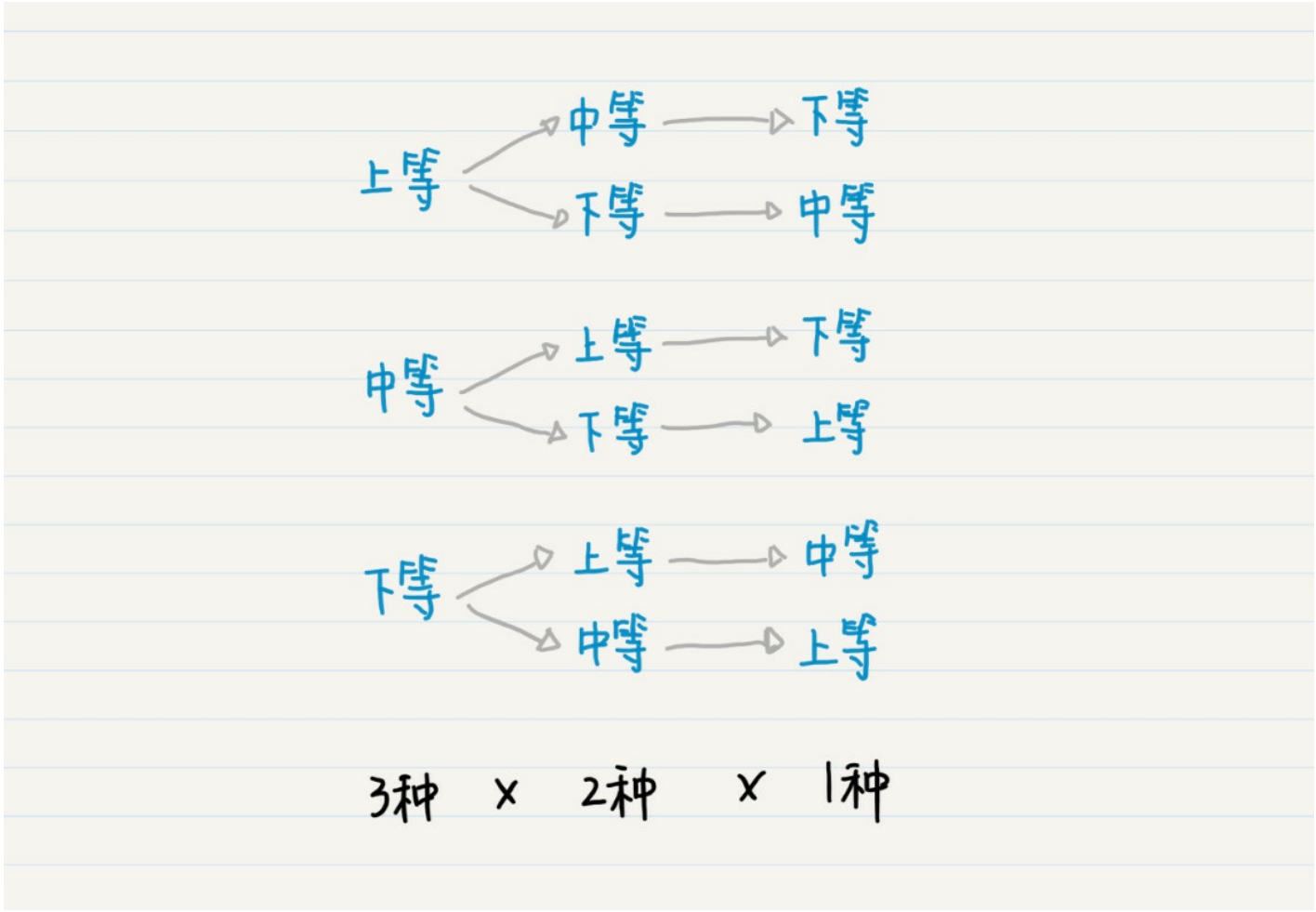
这一节我们用递归来实现了深度优先搜索。说到这，你可能会想到，之前讨论的归并排序、排列组合等课题，也采用了递归来实现，那它们是不是也算深度优先搜索呢?

我把归并排序和排列的分解过程放在这里，它们是不是也可以用有向树来表示呢?

在归并排序的数据分解阶段，初始的数据集就是树的根结点，二分之前的数据集代表父节点，而二分之后的左半边的数据集和右半边的数据集都是父结点的子结点。分解过程一直持续到单个的数值，也就是最末端的叶子结点，很明显这个阶段可以用树来表示。如果使用递归编程来进行数据的切分，那么这种实现就是深度优先搜索的体现。



在排列中，我们可以把空集认为是树的根结点，如果把每次选择的元素作为父结点，那么剩下可选择的元素，就构成了这个父结点的子结点。而每多选择一个元素，就会把树的高度加1。因此，我们也可以使用递归和深度优先搜索，列举所有可能的排列。



从这两个例子，我们可以看出有些数学思想都是相通的，例如递归、排列和深度优先搜索等等。

我来总结一下，其实深度优先搜索的核心思想，就是按照当前的通路，不断地向前进，当遇到走不通的时候就回退到上一个结点，通过另一个新的边进行尝试。如果这一个点所有的方向都走不通的时候，就继续回退。这样一次一次循环下去，直到到达目标结点。树中的每个结点，既可以表示某个子问题和它所对应的抽象状态，也可以表示某个数据结构中一部分具体的值。

所以，我们需要做的是，观察问题是否可以使用递归的方式来逐步简化，或者是否需要像前缀树这样遍历，如果是，就可以尝试使用深度优先搜索来帮助我们思考并解决问题。

今日学习笔记

第12节　树的深度优先搜索（下）

1. 如何用数据结构表示树？

我们可以设计一个TreeNode类，表示有向树的结点和边。我们还可以用变量prefix表示当前结点之前的前缀，用变量explanation表示某个单词的解释。和之前一样，为了代码的简洁，所有属性都用了public，避免读取和设置类属性的代码。

2. 如何实现深度优先搜索？

前缀树构建和查询，这两个不断重复迭代的过程，都可以使用递归编程来实现。换句话说，深度优先搜索的过程和递归调用在逻辑上是一致的。虽然函数递归调用非常直观，但是在遍历整个字典中所有单词的时候，函数的每次嵌套都可能产生新的变量来保存中间结果，这可能会消耗大量的内存。所以我们还可以用一个更节省内存的数据结构，栈。

黄申 · 程序员的数学基础课

思考题

这两节我讲的是树的深度优先搜索。如果是在一般的图中进行深度优先搜索，会有什么不同呢？

欢迎在留言区交作业，并写下你今天的学习笔记。你可以点击“请朋友读”，把今天的内容分享给你的好友，和他一起精进。

精选留言

Jsoulan

感觉后面栈描述的过程像广度优先遍历呢

2019-01-09 09:20

作者回复

栈是先进后出，还是深度优先。队列先进先出，适合广度优先

2019-01-09 23:33

Joe

老师讲解的树有多个分支，这里用C++简单演示了下二叉树的DFS。

```
/**
* Objective:Given a b-tree, do the depth-first-search(DFS) or traversal.
* Appraoch: stack, no recursion.
* Example:
* 1
* / \
* 2 3
* / \ / \
* 4 5 6 7
* Output:
* preorder: 1 2 4 5 3 6 7
*/

#include <iostream>
#include <stack>
using namespace std;

// tree node
class TreeNode {
```

```
public:
int data;
TreeNode* left = NULL;
TreeNode* right = NULL;

public:
TreeNode(int data)
: data(data) {
}
};
// depth-first-search
class DFS {
public:
void printDFS(TreeNode* root) {
stack<TreeNode*> s;
s.push(root);
// begins!
while (!s.empty()) {
TreeNode* temp = s.top();
s.pop();
// push right first
if (temp->right != NULL) {
s.push(temp->right);
}
if (temp->left != NULL) {
s.push(temp->left);
}
// print data
cout << temp->data << " ";
}
}
};
// test!
int main(void) {
// build tree.
TreeNode* root = new TreeNode(1);
root->left = new TreeNode(2);
root->right = new TreeNode(3);
root->left->left = new TreeNode(4);
root->left->right = new TreeNode(5);
root->right->left = new TreeNode(6);
root->right->right = new TreeNode(7);

DFS test;
cout << "Depth-First-Search: " << endl;
test.printDFS(root);
}
```

输出：

```
Depth-First-Search:
1 2 4 5 3 6 7
```

2019-01-17 22:51

作者回复

很好的实现，代码简洁

2019-01-18 03:24

mickey

勘误：

栈是先进后出。

第3步将结点123、879、945和131压入栈中。

第4步，重复第2步和第3步弹出和压入的步骤，处理结点123，.....

123结点应该是最后处理的，应该先处理131.

2019-01-10 08:55

作者回复

图里画的是对的，文字表达有歧义，我稍后改一下

2019-01-10 23:11

Being

广度优先一般用队列来做，FIFO，这样做到层级遍历；深度优先则用栈来做，FILO，这样做到按深度一条条的遍历下去。在实现上是这么区别的，我看上面有同学混淆了。

2019-01-09 19:25

作者回复

总结的很好

2019-01-09 23:22

xlcoder

你好，黄老师 示例代码是否有 github 的地址，拿手机看示例代码 太痛苦了

2019-01-12 10:35

作者回复

我会在专栏第一个大部分结束后，整理并放到Github上，请关注后续的总结和加餐文章。

2019-01-14 03:08

梅坊帝卿

有没有reverse iterator 这样里面那个临时栈就不需要了

2019-01-12 10:09

作者回复

好想法，Java里可以使用collection.reverse，这样不用自己实现reverse功能了

2019-01-14 03:07

Jsoulan

感觉先序遍历根结点应该也是最后出栈的吧

2019-01-10 23:12

作者回复

如果只要求输出结点，输出后并且获得它的子结点之后，可以出栈

2019-01-11 23:24

Jsoulan

第一步110节点入栈，然后找其子节点，子节点入栈，直到叶子节点，然后叶子节点出栈，回溯父节点依次出栈，110节点不应该是最后出栈吗？？

2019-01-09 23:39

作者回复

好问题，其实深度优先有几种细分，前序、中序和后序，我这里讲的是前序，每个父结点第一次被访问的时候就输出

2019-01-10 23:09

溯雪

正好最近在用neo4j做一些无向图遍历的东西，遍历的时候还要考虑到节点是否已访问过。在多线程环境下，直接在节点对象上作访问标记不太好搞，我是用一个hashset来记录已访问过的节点id并作判断，当数据量大的时候这个hashset还是比较占空

间的，老师有木有比较好的方法。。

还有就是辅助遍历的栈java.util.Stack，jdk中是这样建议的：
“
Deque 接口及其实现提供了 LIFO 堆栈操作的更完整和更一致的 set，应该优先使用此 set，而非此类。例如：

Deque<Integer> stack = new ArrayDeque<Integer>();
”
请问老师这里的更完整和更一致指的是?
2019-01-09 14:29

作者回复

对于neo4j我不太熟悉，你可以看看这种语言下boolean是几个字节，如果是1位甚至1个bit，而且结点相对稳定，那就可以考虑boolean数组。或者自己实现一个二进制加餐那节提到的bit array，不过工作量比较大一点
2019-01-09 23:31