1 树

1.1 树的形成

树结合了向量和列表，在**插入**和**查找**上都有不错的效率，是一种**半线性**的数据结构。

1.2 树的应用

- 层次关系的数据表示
- 表达式
- 文件系统结构

1.3 有根树

树是一种特殊的图，把图中一顶点作为根（root）后，则图可以称作有根树（rooted tree）。

对任意几个有根树进行合并的结果一样是有根树。

相对于树 T，它的子树记做 $T_i = subtree(r_i)$。

1.4 有序树

了解以下概念：

- 孩子（child）
- 兄弟（sibling）
- 父亲（parent）
- 祖先（ancestor）
- 度（degree）：一个顶点孩子的数目。

可以用顶点数 n 作为复杂度的参照，因为有以下公式：

$$edge = n - 1 = \sum_{r \in V} degree(r)$$

1.5 路径和环路

如果 k+1 个点通过 k 条边相邻，则构成一条路径（path），也称通路，路径长度 = k。

如果一个路径的首尾顶点相同，则成为环路（cycle/loop）。

1.6 连通和无环

如果任意两顶点之间均有路径，称作连通图（connected）；不含环路，称为无环图（acyclic）。

树是一种**无环连通图**，也是原图的**极小连通图**和**极大无环图**。

1.7 深度和层次

不引起歧义的情况下，路径，顶点，子树可以相互指代，即：

$$path(v) \sim v \sim subtree(v)$$

有定义 v 的深度：

$$depth(v) = |x|$$

在 $path(v)$ 上的顶点，均称作 v 的祖先（ancestor），v 是它们的后代（descendent），同时除 v 自身外，是真（proper）祖先/后代。

半线性：在任一深度，祖先（前驱）唯一，后代（后继）不唯一。根顶点是所有顶点的公共祖先，深度为 0。

没有后代的顶点被称为叶子 (leaf)。

叶子深度中的最大者，称作树的高度 (height)，且有

$$height(v) = height(subtree(v))$$

特别地，定义空树的高度是-1。

$$depth(v) + height(v) \leq height(T)$$

2 树的表示

2.1 树的接口

1. root()：获取根节点

2. parent()：获取父节点
3. firstChild()：获取第一个子节点
4. nextSibling()：兄弟
5. insert(i, e)：将 e 作为第 i 个孩子插入
6. remove(i)：删除第 i 个节点
7. traverse()：遍历

2.2 父亲

- rank：序号
- data：数据
- parent：父节点序号，根节点为-1。

时间复杂度在寻找父亲与根节点时是 $O(1)$，查找孩子兄弟是 $O(n)$。

2.3 孩子

- rank：序号
- data：数据
- children：孩子列表

时间复杂度在寻找孩子节点时是 $O(1)$，查找父亲是 $O(n)$。

2.4 父亲 + 孩子

- rank：序号
- data：数据
- parent：父亲
- children：孩子列表

时间复杂度在寻找孩子和父亲节点时都是 $O(1)$, 但各个节点的 children 可能要保留 n 个引用。

2.5 长子 + 兄弟

- rank：序号
- data：数据
- parent：父亲
- firstChild：长子
- nextSibling：下一个兄弟

时间复杂度在寻找孩子和父亲节点时都是 $O(1)$, 且各个节点的 children 最多 2 个引用。

3 二叉树

3.1 概述

每个节点度数不超过 2 的树，成为二叉树（binary tree）。
可以划分为左子树和右子树，默认左子树在前。
一些规律：

1. 深度为 k 的节点，至多 2^k 个。
2. 含 n 各节点，高度为 h 的二叉树中，$h < n < 2^{h+1}\ sim 2 \times 2^h$。
3. $n = h + 1$ 时，为单链；$n = 2^{h+1}$ 时，为满二叉树（full binary tree）。
4. 高度增长的缓慢，宽度增长的迅速。

3.2 真二叉树

每个节点的度数均为偶数的二叉树为真二叉树。可以通过为单孩子的节点添加虚拟孩子来实现真二叉树。

3.3 描述多叉树

多叉树可以通过长子-兄弟表示法并进行旋转后，得到一颗二叉树。

4 二叉树实现

4.1 BinNode 模板类

- parent
- lChild
- rChile
- height
- npl：左式堆
- color：红黑树

4.2 BinNode 接口实现

插入孩子 return leftChild = new BinNode(e, this)

4.3 高度更新

高度发生变化是因为左子树或右子树高度发生变化，树的高度是左子树或右子树高度中的最大者 +1。

更新高度时，循环更新 x 的父级节点的高度直到根节点。

复杂度为节点的深度。

4.4 节点插入

1. 增加树的长度
2. 挂上节点
3. 更新全树的高度

4.5 遍历

- 先序（preorder）
- 中序（inorder）

- 后序（postorder）
- 层次（广度）：自上而下，自作而右

4.6 递归遍历

1. 如果节点为空，则返回
2. 访问当前节点
3. 遍历节点的左孩子节点
4. 遍历节点的右孩子节点

4.7 迭代先序遍历 1

如果递归形式的调用都出现在函数的末尾，则成为尾递归，尾递归通常可以转换为迭代的形式进行优化。

1. 声明一个空栈并让根节点入栈
2. 弹出栈元素并访问
3. 将右孩子入栈
4. 将左孩子入栈
5. 循环步骤 234 直到栈为空

4.8 迭代先序遍历 2

先序遍历的逻辑可以简化为先无限从左子树往下进行访问，然后再从右子树进行倒序回溯。假定某棵子树的所有左子节点为 $L_1 \cdots L_n$，与节点平级的右子树为 $T_1 \cdots T_n$，则二叉树的遍历顺序可以简单表示为

$$L_1 \to \cdots L_n \to T_n \to \cdots T_1$$

1. 声明一个空栈
2. (a) 访问当前节点
   (b) 将该节点右孩子（右子树）入栈

(c) 迭代到下一个左子树

(d) 循环直到左侧链被遍历完毕

3. 如果栈为空，则退出循环

4. 不为空则弹出下一棵右子树的根

5. 循环 2 至 4 步

4.9 中序遍历

中序遍历的逻辑可以简化为先访问最深的左子树，然后再访问对应的右子树，然后将这科子树剔除，递归的。假定某棵子树的所有左子节点为 $L_1 \cdots L_n$，与节点平级的右子树为 $T_1 \cdots T_n$，则二叉树的遍历顺序可以简单表示为

$$L_n \to T_n \to L_{n-1} \to T_{n-1} \to \cdots L_1 \to T_1$$

1. 声明一个空栈

2. (a) 将该节点左孩子（左子树）入栈

   (b) 迭代到下一个左子树

   (c) 循环直到左侧链被遍历完毕

3. 如果栈为空，则退出循环

4. 不为空则弹出栈内节点

5. 访问新节点

6. 转向右子树（注意为空）

7. 循环 2 至 6 步

复杂度不为 $O(n^2)$，因为循环中对 n 个节点进行了分摊。

4.10 层次遍历

因为先序，中序，后序遍历都有下一层节点先于上一层节点被访问的情况，所以无法实现需求。层次遍历是严格意义上的先进先出，自热而然想到队列的数据结构。

1. 声明一个空队列
2. 根节点入队
3. (a) 从队列里取一个节点
   (b) 访问该节点
   (c) 左孩子入队
   (d) 右孩子入队
4. 循环执行 3 直至队列为空

4.11 重构

根据树的序列还原树的结构
先序 | 后序 + 中序先序 + 后序无法确定树。
[先序 + 后序]* 真