树(Tree)

2019 年 3 月 6 日

1 树（tree）

基本概念

度(degree) 某个结点(node)的子树的个数为该结点的度.其中最大值的为树的度.

分支结点(branch)与叶节点(leaf) 度不为零的结点,为分支结点.度为零的结点为叶节点.

子结点(child)，父结点(parent)和兄弟结点(sibling)

层(level)和高度(depth) 根节点为第一层(或第0层),树中结点的最大层次为树的高度.

有序树和无序树 各结点子树按照一定的次序从左向右,且次序不可以随意改变,称为有序树,否则为无序树.

森林(forest) $m(m \geq 0)$棵树的集合

树的性质

1. 树的结点总数=所有结点的度之和+1
2. 度为m的树的第i层,至多有m^{i-1}个结点.[1]

[1]因为度为m,第2层结点至多为m,第3层至多为$m \times m$.

3. 高度为h的m叉树至多有$\frac{m^b-1}{m-1}$个结点.[2]
4. 具有n个结点的m叉树最小高度为$ceil[\log_m n(m-1)+1]$.[3]

树概念的表示方法

1. 树形表示法
2. 文氏图表示法：使用大圆圈表示根，其中小圆圈表示子树。
3. 凹入表示法：根对应一个长矩形，其子树用较短的矩形表示并在父结点之下；所有兄弟结点长度一致。
4. 嵌套括号法

存储方式

1. **广义表存储**：根结点为表头，表头指向长子结点，长子结点链接次子结点。如果某结点为分支结点，则该结点的值为指向另一个子链表的指针。即用一个嵌套链表表示树。
2. **双亲存储结构**：在每个结点中，附一个指针域，指向父节点。(但找子节点需遍历整个序列)
3. **孩子存储结构**：根据树的度，设计每个结点中的指针域的个数，指向子结点。
4. **孩子兄弟存储结构**：每个结点附指针域指向长子和自己同层的兄弟结点。

[2]根据上一条性质,高度为h的m叉树的结点个数最多为:$m^0+m^1+..m^{h-1}$.
[3]ceil[n] 表示取大于n的最小整数.

2 二叉树（Binary Tree）

二叉树是一种特殊的树，其结点为一个有限集合（该集合可以为空），每个结点最多有两个子结点，并且左右子结点有次序之分。

m叉树可以通过*孩子兄弟存储法*转换成二叉树 [3]，且二叉树更为规范。

满二叉树是所有叶结点全部在最下层的二叉树，即树结构各个位置均被结点填满；**完全二叉树**是只有最下层不满的二叉树，即除最下层外，各层各位置均被结点填满。

二叉树可以可以分为*顺序存储*和*链式存储*方式。*顺序存储*用连续的存储单元来存放二叉树的数据元，最好用于存储完全二叉树，对于单分支结点较多的二叉树空间利用率低。另外，二叉树的插入，删除等十分不便。*链式存储*使用一个值域存放数据，两个指针域(左，右)指向左右子结点，有着链式存储方式的优点。

2.1 二叉树性质

1. 非空二叉树上叶结点数n_0等于双分支结点数n_2+1，$n_0=n_2+1$。
2. 如果树从第0层开始，非空二叉树上**第i层**上至多有$2^i(i\geq 0)$个结点。（如果从第1层开始，$2^{i-1}(i\geq 1)$个结点）
3. 高度为h的二叉树**一共**至多有$2^{h+1}-1,(h\geq 0)$个结点。

满二叉树 高度为h的**满二叉树**有$2^{h+1}-1,(h\geq 0)$个结点。

完全二叉树

1. 具有n个结点的完全二叉树的高度为$log_2(n+1)-1$
2. 对于完全二叉树，若其结点编号从0开始（各结点编号为0...$n-1$），则编号为$i(0\leq i\leq n-1)$的结点：
   (a) 若$i=0$,则i结点为根节点，没有父结点；若$i\geq 0$,则i结点的父结点编号为$ceil(\frac{i-1}{2})$
   (b) 编号为i的结点，其左子结点为$2\times i+1(2\times i+1<n)$,右子结点为$2\times i+2(2\times i+2<n)$.

(c) 若i为偶数，则它是右结点（根结点除外，i != 0）；若i为奇数，则它是左结点。

(d) 编号为i的结点，其层次为$\log_2(i+1)$。

2.2　抽象数据类型接口（ADTI）

对于二叉树，除构造和析构函数外，还需提供插入结点，删除结点，遍历，搜索，和显示函数。

每种函数根据具体细节可能提供多个函数。

```
// 1.插入结点部分
int InsertAsRoot(T x);
// 在pointer的左子结点处，插入新结点
int InsertAsLeftChild(BinaryTreeNode<T>* pointer, T x);
// 在pointer的右子结点处，插入新结点
int InsertAsRightChild(BinaryTreeNode<T>* pointer, T x);

// 2.删除结点部分
// 删除sub_tree的所有子树(不包括sub_tree结点）
void RemoveSubTree(BinaryTreeNode<T>* sub_tree);
void Clear(); // 清空树

// 3.遍历部分
// 4.搜索部分
// 5.显示部分
void Display();  // 用凹入法显示二叉树
```

2.3　数据类型的实现：顺序存储

2.4　数据类型的实现：链式存储

结点的插入　插入操作必须是：在当前结点的子结点位置上，插入新结点。

链表在插入新结点时，只需new一个新结点，并将其按需要链接到原链表中即可。与链表不同的是，树的插入是需要指定具体位置，即指定位置的结点指针。

如果插入操作使用的是在当前位置插入新元素的话，则无法及时更新其父结点的指针域指针。将会有以下问题：

比如当需要在某结点插入新元素时（如果插入函数的参数为结点指针pointer和元素值x）使用如下代码的话：

```
auto* node = new BinaryTreeNode<T>;
node->_data = x;
node->_left = NULL;
node->_right = NULL;
pointer = node; // 结点指向node
```

如上述代码这样，用来指定插入位置的结点指针被更新后，但没有在父结点的指针域中更新相应的值[4]，将会导致断链，父结点不指向新的子结点。

如果指定插入位置的结点指针的话，就会有上述问题：指定的结点位置指针指向新创建的结点。

如此操作的话，必须有类似链表的操作：保持一个指针指向某结点的父结点。将新创建的结点，重新链接到父节点的指针域中。因此在某结点的子结点位置插入新结点较为简洁。

子树的删除 如果想要保存某结点Node==NULL的标志，则需要在删除左右子树后，将当前结点对应的指针域置为NULL。由于这里采用的是*二叉链表*的方案，没有指向父节点的指针。

所以删除操作应该是：删除输入结点的左右子树（并不删除输入结点），并且将对应的指针域置NULL。*如若想要删除某结点，输入结点应该是其父结点*。

使用递归调用时，应该尤其在意调用函数的执行顺序。什么时候调用下一级函数，**逻辑判断会不会影响递归调用**，该函数返回后需要执行什么操作。

考虑到该子树删除函数并不删除输入结点，所以在调用时，具体删除操作应该在递归调用之后。即删除子树后，再删除当前结点。

```
RemoveSubTree(sub_tree->_left); // 递归调用
delete sub_tree->_left; // 删除操作
```

[4]原来父结点中值为NULL

```
    sub_tree->_left = NULL;
```

如若不然，则在递归调用后，直接执行后续代码，会漏掉当前结点。

```
auto* child = sub_tree->_left; //child为输入结点的子结点
if (child->_left == NULL
          && child->_right == NULL) { // 若child是叶节点
    delete child;            // 删除操作
    sub_tree->_left = NULL;
} else {
    RemoveSubTree(child);  // 递归调用
    // 如果child不是叶结点，在递归调用删掉其子树后，
    // 会直接执行后续代码，漏删child结点(该函数不删除输入结点)
}
```

由此说明：分析递归调用除了要**分析递归调用过程**之外，还要**分析递归返回过程**。即应首先从上而下的分析调用过程；随后应从底层自下而上的分析逐级返回的过程。并且应**清晰的定义递归函数要实现的功能**，在自下而上和自上而下的分析中，作为基准。

如果递归函数定义不清晰，很容易造成代码冗余，甚至造成混乱。因为会很自然的像调用非递归函数那样，企图在调用之前为被调用函数做一些处理。但因为是递归调用，自己调用自己，这些处理完全可以在递归调用时交由下一层调用来完成。

父结点的查找　因为没有使用指向父节点的指针，所以查找父节点会不可避免的涉及到递归（recurve）操作。在递归调用过程中，如果遇到所求结点，则可以将此结点返回（逐级返回）；或使用全局变量存储所求结点，函数逐级返回找到所求结点的标志。但考虑到代码的简洁性，应该优先选择直接返回所求结点的方法。

并且，因为是递归调用，所以递归函数应当输入当前结点指针，以保证递归调用可以逐级检查到树的各层结点。

查找当前结点N的父节点的思路，就是检查每一个结点的子结点，是否就是当前结点N。即如果某结点C 的子结点就是当前结点N，那该结点C 就是当前结点N的父结点。

每个结点的状态可以分为三种：1）为NULL—当该结点为叶结点时；2）为Node—已找到父结点；3）都不是—需要继续递归调用。

代码中必须**顺序**检查左结点，随后右结点。为了保证可以依次调用检查左结点和右结点的代码，检查为NULL部分不应该在此处进行，否则代码应该是*先判断左结点是否为NULL，如果不是NULL，进而递归调用；再判断右结点是否为NULL，如果不是进而递归调用*。但是这样，不能实现左右结点的依次检查，同时返回结果。

遍历（Traversal） 所有遍历均使用递归的思想实现。

前（先）序遍历（pre-order） 是（递归地）先对当前结点进行操作，再访问左结点，再访问右结点。即先对当前结点进行操作，之后再访问别的结点，称之为“先序”。*先序遍历大体顺序是：第一个结点是根结点，第二个结点是其左子结点，再左下角点，随后是右子结点。*

中序遍历（in-order） 是先访问左结点，之后再对当前结点进行操作，最后访问右结点。即对当前结点额操作操作，在访问两个结点之间，称之为“中序”。*中序遍历大体顺序是：第一个结点是最左下角点，第二个结点是其父结点，随后是次左下角点，最后是右子结点。*

后序遍历（post-order） 是先访问左结点，再访问右结点，最后对当前结点进行操作。即访问左右结点之后，再对当前结点进行操作，称之为“后序”。*后序遍历大体顺序是：第一个结点是最左下角点，第二个结点是次左下角点，整体看上去像是从左下角点开始的逆时针外包线。*

上述三种顺序只是对当前结点的操作的位置有区别，实际程序在访问一颗树时的顺序是一致的。其中先序变量是程序第一次访问到当前结点时就进行操作，而中序变量是第二次访问该结点时进行操作，而后序遍历是第三次访问该结点时进行操作。

同时给定一棵树的前序序列和中序序列，可以唯一确定一颗树。前序序列中的第一个元素肯定是树的根，用该元素可以将中序序列划分为左子树，当前结点，右子树，三部分。

如此，顺序取出前序序列中的元素，作为当前结点，可以不停地将中序序列进行划分，直至形成一棵树。

不用递归实现遍历 上述过程使用递归实现，也可以使用栈和队列实现。用循环和栈代替递归，用栈记录访问回退路径。

用栈实现先序遍历（pre-order） 其思路是，先对当前结点进行操作，再访问其左结点，同时将其右结点压栈；一直循环到叶节点，再依次对出栈的结点进行上述操作。最后，直至左子结点为空，栈也为空，整个循环结束。

用队列实现层序遍历 其思路是，当访问某结点时，将其子结点依次入队。如果队列非空，则执行出队的元素，对出队的某元素进行操作时，依然要将其子结点入队，直至队空。这里对队列的使用与计算杨辉三角形思路类似。

使用队列可以将待操作的数据入队，在执行某步操作时，将待执行操作入队。类似代办清单（list），根据队列顺序依次执行操作即可。

用栈实现中序遍历（in-order） 中序遍历在访问某结点时，先将其压栈，再访问其左子结点，直至到叶结点。随后不断出栈，访问该结点和其右子结点。只将当前结点压栈，符合中序遍历，第二次访问才进行操作的特点。

2.5 后序遍历应用举例：树结点计数和高度计数

后序遍历的特点是：对当前结点进行操作之前，已经遍历过左右子树，即已经对左右子树进行过操作。

在树结点计数的例子中，使用后序遍历，在将本结点累加到总数变量之前，已经拿到了左右子树的结点总数，继续累加即可。对于高度计数也是类似思路。

3 线索二叉树（Thread Binary Tree）

由于树的遍历比较浪费资源，因此可以使用称之为线索（*thread*）的指针域，指向后继结点，简化遍历过程。

比如某树采用先序遍历排序，根据先序序列，树各结点增加一个指针域指向先序序列顺序中的后继结点，由此遍历一棵树可以像遍历链表一样。

使用线索的树，称之为线索树。

4 通用树和森林

4.1 森林和二叉树的转换

4.2 森林和树的遍历

长子-兄弟存储-寻找p节点的父节点 从某一节点开始寻找(通常是根节点).顺序是先检查子结点,后检查兄弟结点,用递归调用检查子结点,返回时继续检查兄弟结点.直至遇上终止条件:找到p或者为NULL.

5 堆

堆典型的有最小堆（*min-heap*）和最大堆（*max-heap*）。

关键码（*key*）：在数据元素中增加一个域，存放一个数字（或其他数据类型），用于组织这个数据结构，该数据称为关键码（key）。比如可以在数据元素中增加一个key，表示优先级。

key与下标不同，下标从0− >n-1，表示数据元素的逻辑位置。而key与元素位置无关，key是根据需要增加的，可以用于排序或组织数据结构的数据。

key与数据元素的值（value）无关，一般可以根据key对数据结构进行排序后，再读取各个数据元素的值。

堆是结点附加key的二叉树。最小堆中所有父节点的key小于等于两个子结点的key，所以根结点（堆顶）中的key是整棵树中最小值；最大堆中所有父节点的key大于等于两个子结点的key，所以堆顶的key是整棵树中最大值。

上述的最小堆和最大堆可以表述为具有堆序（heap-ordered），最小堆序就是父结点小于等于子结点；最大堆序就是父结点大于等于子结点。

最小堆原理和最大堆一致，这里以最小堆为例。

5.1 最小堆（min-heap）

最小堆可以使用数组实现。

5.1.1 抽象数据类型接口（ADTI）

堆的构造可以分为两种方式，一种是先创建空堆，再根据最小堆序依次插入新元素；另一种方式是输入一个数组，将其调整为为最小堆。

除了构造和析构函数外，还需提供插入函数，和删除元素函数用于删除堆顶元素。

```
MinHeap(int max_size); // 构建空堆
MinHeap(T* arr, int n); // 利用数组arr构建堆
~MinHeap();

bool Insert(T item);  // 将元素插入尾部，再调整至正确位置
```

```
T Remove(); // 删除第一个元素
void SiftDown(int start, int end);// 将结点start下滤到合适位置
void SiftUp(int start); // 将结点start上滤到合适位置
```

5.2 数据类型的实现

利用已有数组建堆 将已有数组复制到堆中数组后，对最后一个分支结点进行调整，使其有序；随后对其兄弟结点进行调整，保持下层有序后，再对上层进行调整；直至整个树有序。

上滤下滤操作 即将某一个元素，根据其值不断上移（下移）至合适位置的操作。

根据父结点确定子结点时，可以使用*child = 2*father + 1*，但一定要事先确定*2*father + 1*是否在数组范围（$2 * father + 1 <= n - 1$）内，否则（无子结点）超出范围引发错误。

根据子结点确定父结点，只需*child =*（*father-1*）*/2*，没有超出范围的问题。

结点的插入删除 插入操作是将新元素放置在数组尾部，调用“上滤”操作使其移动到合适位置。

删除操作是删除一个元素，随后将最后一个元素覆盖至第一位置，再对第一个位置元素进行“下滤”操作，使其回到正确位置。

6 Huffman树

Huffman树，即最优二叉树，是加权路径长度最短的二叉树。

路径长度就是两个结点间路径上的分支条数。树的路径长度，就是根结点到每个结点的路径长度之和。

对于完全二叉树，其第k层（根为第0层）结点个数最多为2^k，且该层结点的到根结点的路径长度为k，所以这个树的路径长度为：

$$PathLength = \sum_{i=0}^{n-1} ceil[\log_2(i+1)]$$

其中i表示某结点的序号，ceil[]表示向下取整。即第0个元素（根结点）的路径长度为$ceil[\log_2(0+1)] = 0$，第1，2个元素的$PL = ceil[\log_2(1+1)] = 1, ceil[log_2(2+1)] = 1$

上述公式是树的路径长度的下限，即最小值，称之为最小路径长度。满足上述公式的树均具有最小路径长度，典型的例子除完全二叉树外还有理想平衡二叉树。

加权路径长度 给二叉树的每个叶节点一个权值，则称该二叉树为“扩充二叉树”，其中带有权值的叶节点称为外结点，不带权值的分支结点称为内结点。对于该“扩充二叉树”的外结点的带权路径长度为：

$$WeightedPathLength = \sum_{i=1}^{n} w_i l_i$$

其中l_i是外结点的路径长度，w_i是附加的权值。

附加权值后，具有最小加权路径长度的扩充二叉树，不一定是完全二叉树。扩充二叉树权值越大的外结点，离根节点越近，其带权路径长度越小。

Huffman树是加权路径长度最短的二叉树。

6.1 Huffman算法

当给定权值$\{w_1, w_2...w_n\}$后，构建一颗Huffman树。

1）首先构建有n棵扩充二叉树的森林F，每棵树仅有一个根结点。

2）在F中挑选两个根结点权值最小的二叉树，构建一颗新树。这棵新树根结点权值设置为其左右子结点的权值之和。

3）在F中删除上述两棵树，并将新树加入到F中。

4）不断重复2，3步骤，直至F中仅剩一棵树。

6.1.1 Huffman算法实现

huffman树的构建，因为涉及到挑选权值最小的二叉树，所以可以利用最小堆来实现。

将所有树插入到最小堆后，依次弹出两棵树，并将其合并为一棵新树，再插入到最小堆中。由于共有n棵树，每次操作合并两棵树，所有这样的操作重复n-1次就可以获得一棵huffman树。

6.2 应用举例：Huffman编码

假定某编码方案，有a,b,c,d,e五个字符组成。如果各个字符出现概率相等，则可以使用定长编码方案，每个字符的编码长度一样。但实际上，各个字符出现概率并不一致，因此对于出现概率低的字符使用较长的编码；而出现概率高的字符，使用较短的编码方案，可以提高信息传输速率。

表 1: 各个字符出现概率

字符	概率	定长编码	变长编码
a	0.12	000	1111
b	0.40	001	0
c	0.15	010	110
d	0.08	011	1110
e	0.25	100	10

因此上述问题可以表述为使$\sum w_i l_i$最小，可以使用huffman编码算法解决，但结果可能不唯一。

前缀编码，任何一个字符均不是其他字符的前缀，这样可以避免在解码时引发歧义。通过树构建的二进制编码，自然的是前缀编码。

参考文献

[1] 严蔚敏. 数据结构（C语言版）. 北京: 清华大学出版社, 2007.

[2] 邓俊辉. 数据结构（C++语言版）(第三版). 北京: 清华大学出版社, 2013.

[3] 李春葆. 数据结构考研指导. 北京: 清华大学出版社, 2002.

[4] 殷人昆. 数据结构：用面向对象方法与C++描述. 北京: 清华大学出版社,1999.