树基础 | Tree Basic

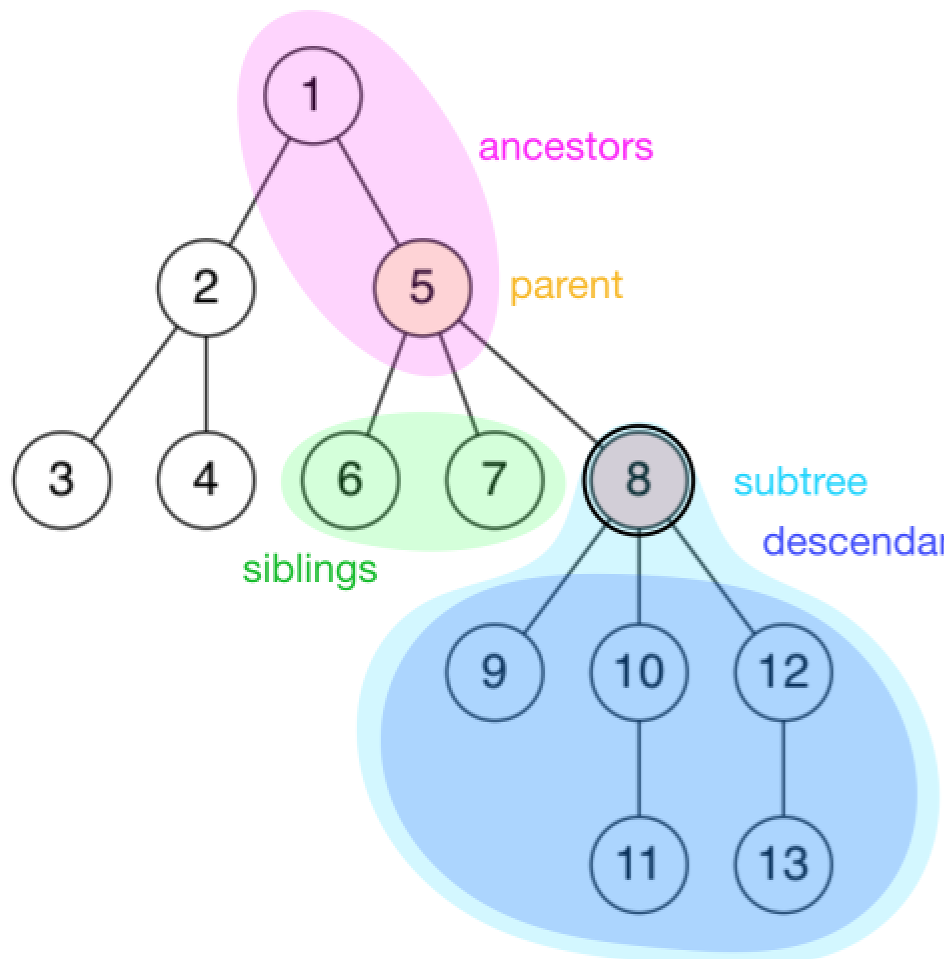
一些基本定义

- **无根树（unrooted tree）**：无根树有几种等价的形式化定义：
  - 有 n 个结点，$n-1$ 条边的连通无向图；
  - 无向无环的连通图；
  - 任意两个结点之间有且仅有一条简单路径的无向图；
  - 任何边均为桥的连通图；
    - 即，删去任何一条边则不再连通。
  - 没有圈，且在任意不同两点间添加一条边之后所得图含唯一的一个圈的图。

- **有根树（rooted tree）**：在无根树的基础上，指定一个结点称为 **根（root）**，则形成一棵有根树。有根树在很多时候仍以无向图表示，只是规定了结点之间的上下级关系。有根树上有如下概念：
  - **父结点（parent node）**：结点到根的路径上的第二个结点。根结点没有父结点。
    - 显然，树上任一结点到根结点的路径是存在且唯一的。
  - **祖先（ancestor）**：结点到根的路径上，除了它本身以外的全部结点。根结点没有祖先。
  - **子结点（child node）**：如果 u 是 v 的父结点，则 v 是 u 的子节点。一个结点的子结点可能有 0 到多个。在一般的树上，子结点的顺序不作区分。
  - **兄弟（sibling）**：父结点相同的子结点互为兄弟。
  - **后代（descendant）**：子结点和子结点的后代。或者说，所有以该结点为祖先的结点是该结点的后代。
  - **子树（subtree）**：删掉与父亲相连的边后，该结点所在的子图。

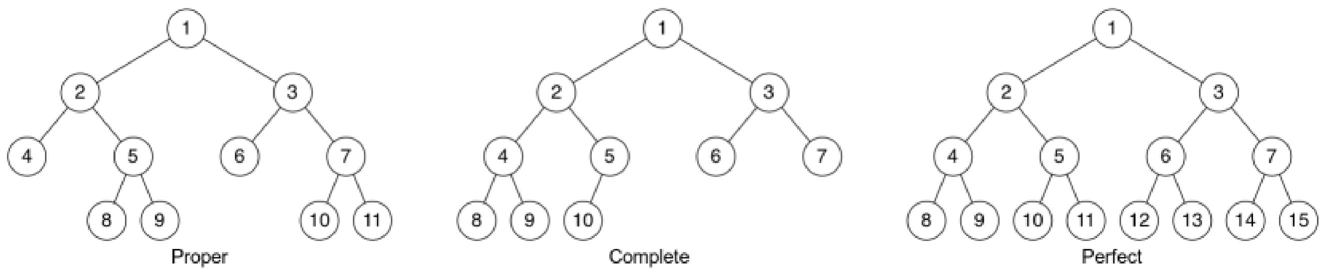
- **叶结点（leaf node）**：
  - 对无根树：度数不超过 1 的结点（当且仅当 n = 1 时存在度数为 0 的情况）；
  - 对有根树：没有子结点的结点。

- **结点的度（degree）**：结点子树的个数。
- **树的深度（depth）**：从根结点到叶结点的最长路径长度。



树上一些概念的示意图。图源 OI Wiki

- **二叉树（binary tree）**：通常指有根二叉树。每个结点至多有两个子结点的树。通常将子结点确定一个顺序，称左子结点和右子结点。
  - **完整二叉树（full / proper binary tree）**：每个结点的子结点均为 0 个或 2 个；
  - **完美二叉树（即满二叉树，perfect binary tree）**：所有叶结点深度均相同的二叉树；
  - **完全二叉树（complete binary tree）**：仅最深两层结点的度可以小于 2，且最深一层的结点都集中在该层最左边的连续位置上。或，所有结点的编号都与满二叉树中的编号相同的二叉树。



有特殊性质的二叉树的示意图。图源 OI Wiki

一个判断题：

There exists a binary tree with 2016 nodes in total, and with 16 nodes having only one child.

答案：错误。

每个完整二叉树的结点数均为奇数个。一个有 16 个结点只有一个子结点的二叉树相较一个完整二叉树来讲缺少了 16 个子树，且这些子树均为完整二叉树，因此这些子树的结点数之和必为偶数（奇 * 偶）。因此剩余的节点个数必为奇数。

树的存储

只记录父结点

用一个数组 `parent[N]` 记录每个结点的父亲结点。

这种方式可以获得的信息较少，不便于进行自顶向下的遍历。常用于自底向上的递推问题中。

邻接表

给每个结点开辟一个线性表（vector 或用链表），记录所有与之相连的结点，或记录其所有子结点。

FirstChildNextSibling 表示法

是课本中使用的表示法。首先对所有结点的子结点确定一个顺序。然后对每一个结点，储存它的第一个子结点和下一个兄弟结点。

课本中是用链表实现的。当然，这个表示法用数组实现相当方便。

二叉树的存储

由于二叉树结点个数有限，我们可以使用两个数组来表示每一个结点的子结点。

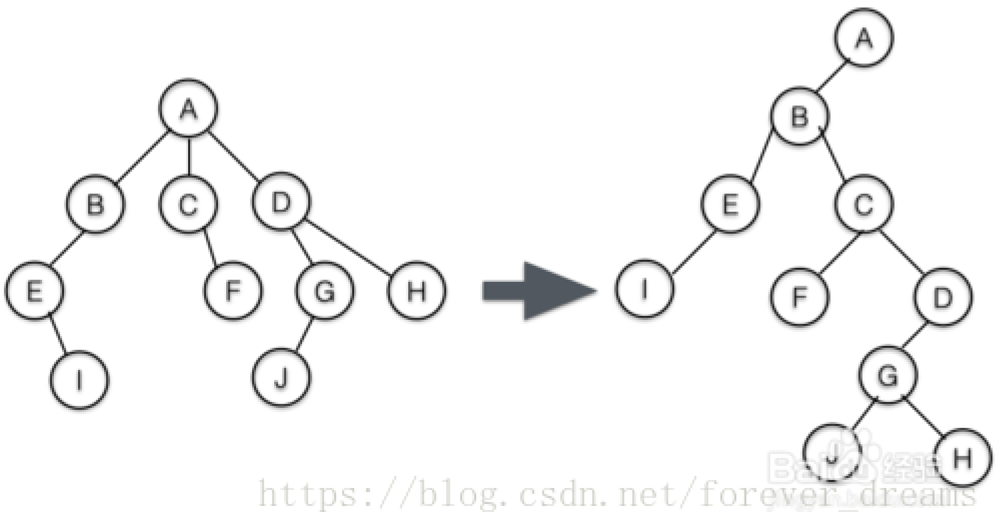
课本上也是使用链表实现的。每个结点，存储其两个子结点的指针。

一般树转为二叉树

（本部分内容来自参考资料 3）

- 将树的根节点直接作为二叉树的根节点

- 将树的根节点的第一个子节点作为根节点的左儿子，若该子节点存在兄弟节点，则将该子节点的第一个兄弟节点（方向从左往右）作为该子节点的右儿子
- 将树中的剩余节点按照上一步的方式，依序添加到二叉树中，直到树中所有的节点都在二叉树中



一般树转为二叉树示意图

```
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int N=105;
int son[N],left[N],right[N];
int main()
{
    int n,i,x,y;
    scanf("%d",&n);          //表示有n个点
    for(i=1;i<=n;i++)
    {
        scanf("%d",&x);      //x是i号节点的父亲
            if(!son[x])  left[x]=i;      //这两步就是根据左儿子右兄弟的方式转二叉树
            else  right[son[x]]=i;
            son[x]=i;
    }
    for(i=1;i<=n;++i)
      printf("%d %d\n",left[i],right[i]);
    return 0;
}
```

树的遍历

先序遍历（preorder traversal, DLR） 先访问根，再访问子结点；

二叉树的中序遍历（inorder traversal, LDR） 先访问左子树，再访问根，再访问右子树；

后序遍历（postorder traversal, LRD） 先访问子结点，再访问根。

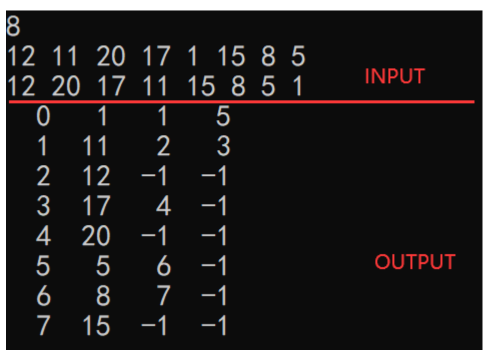
例程可以在下面的表达式树例程中找到。

已知中序遍历和另外一种遍历，可以确定出这棵树。下面是根据一棵二叉树的中序和后序排列建立树的例程：

```
/* Suppose that all the keys in the binary tree are distinct positive integers. */

#include<stdio.h>

const int MAXN = 105;
int n, inorder[MAXN], postorder[MAXN];                  // Input
int left[MAXN], right[MAXN], value[MAXN], count = -1;   // Tree Nodes

int find(int value){
    for(int i = 0; i < n; i++)  if(value == inorder[i]) return i;
    return -1;
}

/* return value of buildTree is the index of the root */
int buildTree(int inLeft, int inRight, int postLeft, int postRight){
    if(inLeft > inRight)    return -1;
    if(inLeft == inRight){
        value[++count] = postorder[postRight];
        left[count] = right[count] = -1;
        return count;
    }
    value[++count] = postorder[postRight];  // In this subtree, postorder[r] is the root.

    int root = find(value[count]), thisIndex = count;
```

```
25     left [thisIndex] = buildTree(inLeft, root-1, postLeft, postRight-(inRight-root)-1);
26     // (inRight - root) is the size of the right subtree
27     right[thisIndex] = buildTree(root+1, inRight, postLeft+(root-inLeft), postRight-1);
28     // (root - inLeft)  is the size of the left  subtree
29 
30     return thisIndex;
31 }
32 
33 int main(){
34     scanf("%d", &n);
35     for(int i = 0; i < n; i++)  scanf("%d", &inorder[i]);
36     for(int i = 0; i < n; i++)  scanf("%d", &postorder[i]);
37     buildTree(0, n-1, 0, n-1);
38 
39     for(int i = 0; i < n; i++)  printf("%3d %3d %3d %3d\n", i, value[i], left[i], right[i]);
40 }
```

这里树的存储方式为 此处 所示的方法。函数 buildTree 的参数略为复杂，尤其是 25~26 行的递归参数设置。建议结合实例进行理解。



一组样例

例程：表达式树 | Expression Tree

表达式树是一棵二叉树。

我们试图实现将后缀表达式转变为表达式树。我们建立一个用来存放树根指针的栈，扫描该后缀表达式：

- 如果遇到操作数，创建一棵单结点树存储它，并将它的指针压入栈中；
- 如果遇到操作符，创建一棵单结点树存储它，并弹出栈顶的两个指针，将这两个指针指向的树作为操作符的两个子结点，然后将新生成的这棵树压入栈中；

- 扫描结束后，栈中只留下一个指针，这就是表达式树的指针。

对这棵表达式树进行前序/中序/后序遍历，得到的结果即为前缀表达式（波兰式）/中缀表达式/后缀表达式（逆波兰式）。

源码如下。为了减少不必要的代码，我们规定：所有操作数均由一个字母代替；所有输入由一个空格分隔，以换行结束。

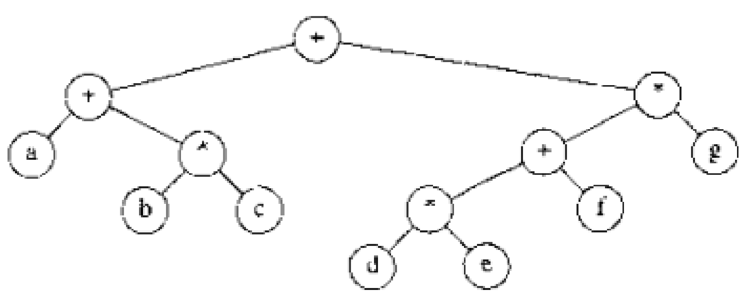
```
#include<stdio.h>
#include<stdlib.h>

typedef char ElementType;
typedef struct TreeNode *Tree;

struct TreeNode{
    ElementType value;
    Tree        leftChild, rightChild;
};

/* 后序遍历，输出为后缀表达式 */
void LRD(Tree T){
    if(T == NULL)   return;
    LRD(T->leftChild);
    LRD(T->rightChild);
    putchar(T->value);
    putchar(' ');
}

/* 中序遍历，输出为中缀表达式 */
void LDR(Tree T){
    if(T == NULL)   return;

    /* 如果当前运算符优先级较低，输出括号 */
    if(T->value == '+' || T->value == '-')
        putchar('(');

    LDR(T->leftChild);
    putchar(T->value);
    LDR(T->rightChild);
```

```

    if(T->value == '+' || T->value == '-')
        putchar(')');
}

/* 读入后缀表达式并建立树 */
Tree buildTree(){
    Tree stack[1005], temp;
    int stackHead = -1;          /* stackHead 记录栈顶元素位置 */
    for(char input = getchar(); input != '\n'; input = getchar()){
        switch(input){
            case ' ':
                break;
            case '+': case '-': case '*': case '/':
                //if(stackHead < 2) return NULL;
                temp = (Tree)malloc(sizeof(struct TreeNode));
                temp->value = input;
                temp->rightChild = stack[stackHead--];
                temp->leftChild = stack[stackHead--];
                stack[++stackHead] = temp;
                break;
            default:
                temp = (Tree)malloc(sizeof(struct TreeNode));
                temp->value = input;
                temp->rightChild = temp->leftChild = NULL;
                stack[++stackHead] = temp;
                break;
        }
    }
    if(stackHead == 0)  return stack[0];
    return NULL;
}

int main(){
    Tree tree = buildTree();
    if(tree == NULL)
        puts("ERROR");
    else{
        LRD(tree); puts("");
```

```
        LDR(tree); puts("");
    }
    return 0;
}
```

输入 1：
    A B C * + D E * F + G * +
输出 1：
    A B C * + D E * F + G * +
    ((A+B*C)+(D*E+F)*G)

输入 2：
    A B C * + D E * F + G *
输出 2：
    ERROR



输入 1 的表达式树图示。图源《数据结构与算法分析：C 语言描述》

关于树的更多知识，在其他文档中记录。

参考资料

1. 树基础 | OI Wiki
2. 《数据结构与算法分析》
3. https://blog.csdn.net/forever_dreams/article/details/81032861

EOF