第4章 二叉树

In [1]:
```
#include <iostream>
#include <algorithm>
using namespace std;
```

一、节点定义

In [2]:
```
enum NodeColor{
    white,
    black
};
```

In [3]:
```
#define BinNodePointer(T) BinNode<T>*
```

In [4]:
```
template <class T>
struct BinNode{
    BinNodePointer(T) parent;
    BinNodePointer(T) lc;
    BinNodePointer(T) rc;
    T data;
    int height;
    NodeColor color;
    bool visited;

    BinNode(T e, BinNodePointer(T) p = NULL, BinNodePointer(T) l = NULL, BinNodePointer(T) r = NULL);

    // 基本属性获取和更新
    int size();
    int computeHeight();
    void updateHeightAbove();

    // 插入删除
    BinNodePointer(T) insertAsLC(const T& e);
    BinNodePointer(T) insertAsRC(const T& e);
    void removeAt(BinNodePointer(T) p);
    void remove();
};
```

构造函数
1.用参数e初始化data；
2.parent，lc，rc默认为空指针，可以用于构造root节点，也允许传入节点进行连接；
3.默认颜色为白色，表示此节点未处理，我这里的颜色和书上为红黑树定义的颜色不是一个东西；
4.默认高度为0，需要在树的动态变化时进行更新。

In [5]:
```
template <class T>
BinNode<T>::BinNode(T e, BinNodePointer(T) p, BinNodePointer(T) l, BinNodePointer(T) r){
    data = e;
    parent = p;
    lc = l;
    rc = r;
    color = white;
    height = 0;
    visited = false;
}
```

宏定义函数
有些判断类型的函数写进去有些啰嗦，用带参数的宏定义更为简单，
这些宏定义的函数在遍历游走算法中特别方便，需要用到什么就在这里补充

In [6]:
```
#define haslchild(x) ((x).lc)
#define hasrchild(x) ((x).rc)
#define haschild(x) (haslchild(x) || hasrchild(x))
#define hasparent(x) ((x).parent)
```

关于size和height
关于每个节点，从这个节点开始的子树具有size属性，从这个节点到树根之间的路径长度就是这个节点深度，从叶子到这个节点的路径的长度就是这个节点的高度，因此，对于一个节点来说具有各种各样的属性，这些属性可以以变量的形式定义在节点中，也可以以函数返回值的形式临时获取，如果以属性的方式存在于节点中，每当涉及到节点的动态操作(插入删除)，就会更新这个节点以及其他相关节点的属性，如果以函数返回值的形式存在于节点中，需要用到的时候再去计算。

基于上述考虑，我们下面分析一下size和height各以什么样的方式实现：

size：
如果以属性的方式存在，插入删除一个节点或一个子树，会导致其所有祖先的size改变，由于祖先路径只有一条，并且仅与树高有关，因此复杂度也不高，所以，size按照属性实现没问题；
如果以函数形式实现，临时求解一个节点的size，则需要遍历这个子树的全部节点，算法复杂度与这颗子树的规模有关；

height：
如果以属性的方式存在，插入操作时，它自己已经是叶子，或者是已经知道height的子树，同样，这个节点的加入或者离开，只影响到一条路径上的祖先的高度，操作复杂度仅与树高有关，height按属性方式实现也没问题；
如果以函数形式存在，需要从这个节点出发，游走至这个节点的最深处，由此可见，也必须遍历这个子树所有的节点；

假如以操作复杂度考虑，如果以n个节点的完全二叉树为例，创造并更新属性，每个节点会有一个O(logn)的更新属性操作，一共n个节点，因此构建并更新完所有节点的属性的总复杂度为O(nlogn)，但是如果以函数形式实现，假设对每个节点都调用一下这个函数，第0层也就是根节点需要遍历n个点，第1层两个节点，每个节点各需要遍历 (n-1)/2 个节点，所以第二层需要遍历n-1个节点，依次类推，第2层需要遍历除去上面的3个节点的剩下的所有n-3个节点，然后是n-5个节点，...，最后一层是2^h个节点，然后加起来就是需要处理的所有节点数，但是不好加，所以做个简单的估计，即便每一层都按需要遍历n个节点计算，也是有logn层，所以复杂度不会超过O(nlogn)，这样看来在访问所有节点属性方面，似乎以函数的形式存在更有优势，但是不要忘了实际场景，在实际场景中，一颗树如果完全构造完毕，如果以属性进行访问，所有属性已经存在于节点中，后面无论访问多少次，每次都是O(1)的复杂度，如果以函数的形式存在，每次都要遍历某个节点的所有子节点，假设要访问千千万万次，每次都要遍历，复杂度要大的多。
所以到底是以属性的方式存在，还是以函数的形式存在，是根据应用场景的

综上所述，我们后面size会按照函数形式实现，height会以属性方式实现，并不是基于算法复杂度考虑，而是两种方式都实现一遍看看。

size的函数实现
如何求解当前节点子树的size：
当前节点的size = 左子树size + 右子树size + 1，
如果左子树为空，返回0，如果右子树为空，返回0

In [7]:
```
template <class T>
int BinNode<T>::size(){
    int leftSize,rightSize;
    leftSize = (haslchild(*this)) ? lc -> size() : 0;
    rightSize = (hasrchild(*this)) ? rc -> size() : 0;
    return 1 + leftSize + rightSize;
}
```

补充：
由于计算以这个节点为root的子树的size，需要遍历这个子树所有的节点，因此，使用任何一个遍历算法都能计算size，在后面介绍完遍历算法后，我们重新设计算法求size

height更新
1.计算这个节点的高度 = 1 + max（左孩子高度，右孩子高度），虽然这个算法看上去好像要遍历所有后代，但实际在树的构造过程中，这个节点一般插入后，就已经是叶子了，所以没后代，所以虽然我们后面写的算法是按照遍历的形式存在，但是实际过程却只有一两步（其实完全可以就按这一两步写个简单的算法）
2.更新所有祖先的高度

In [8]:
```
template <class T>
int BinNode<T>::computeHeight(){
```

```
    int leftHeight,rightHeight;
    leftHeight = haslchild(*this) ? lc -> computeHeight() : -1;
    rightHeight = hasrchild(*this) ? rc -> computeHeight() : -1;
    height = 1 + max(leftHeight,rightHeight);
    return height;
}
```

对于叶子节点来说，leftHeight=-1，rightHeight=-1，因此计算height=1+max(-1,-1)=0
根据前面所说，这里的lc -> computeHeight()完全可以改成lc -> height，不需要递归计算，
因为lc -> height必定已知，或者为-1

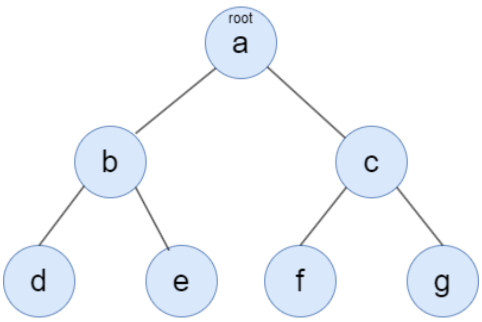
In [9]:
```
template <class T>
void BinNode<T>::updateHeightAbove(){
    computeHeight();  // 计算当前节点的高度
    BinNodePointer(T) p = parent;      // 控制权转移给parent
    while(p) {
        p -> computeHeight();
        p = p -> parent;
    }
}
```

插入算法
给定一个数据e，插入当前节点的左边或右边，并且约定左边或右边为NULL

In [10]:
```
template <class T>
BinNodePointer(T) BinNode<T>::insertAsLC(const T& e){
    lc = new BinNode<T>(e,this);      // 创建新节点，指定父亲为this，然后lc指向这个新节点
    lc -> updateHeightAbove();        // 更新新节点及其祖先的高度;
    return lc;
}
```

In [11]:
```
template <class T>
BinNodePointer(T) BinNode<T>::insertAsRC(const T& e){
    rc = new BinNode<T>(e,this);
    rc -> updateHeightAbove();
    return rc;
}
```

到这里，总算可以创建一颗简单的二叉树了



In [12]:
```
auto root = new BinNode<char>('a');
```

In [13]:
```
auto nodeb = root -> insertAsLC('b');
auto nodec = root -> insertAsRC('c');

auto noded = nodeb -> insertAsLC('d');
auto nodee = nodeb -> insertAsRC('e');

auto nodef = nodec -> insertAsLC('f');
auto nodeg = nodec -> insertAsRC('g');
```

In [14]:
```
root -> height
```

Out[14]: 2

In [15]:
```
root -> size()
```

Out[15]: 7

In [16]:
```
nodeb -> height
```

Out[16]: 1

In [17]:
```
nodeb -> size()
```

Out[17]: 3

删除算法
递归进入这个节点的左右子树，然后释放内存;
注意，这里的data不是动态分配的，所以直接delete这个结构体即可

In [18]:
```
template <class T>
void BinNode<T>::removeAt(BinNodePointer(T) p){
    if (haslchild(*p)) removeAt(p -> lc);
    if (hasrchild(*p)) removeAt(p -> rc);
    if (!haslchild(*p) && !hasrchild(*p)) {
        delete p;
        p = NULL;
    }
}
```

In [19]:
```
template <class T>
void BinNode<T>::remove(){
    if (parent -> lc == this) parent -> lc = NULL;  // 先将节点从树分离出来
    if (parent -> rc == this) parent -> rc = NULL;
    parent -> updateHeightAbove();                      // 更新上面节点的高度属性
    removeAt(this);                                    // 递归删除当前节点
}
```

删除节点(子树)c

In [20]:
```
nodec -> remove()
```

In [21]:
```
root -> size()
```

Out[21]: 4

In [22]:
```
root -> height
```

Out[22]: 2

删除节点(子树)b

In [23]:
```
nodeb -> remove()
```

In [24]:
```
root -> size()
```

Out[24]: 1

In [25]:
```
root -> height
```

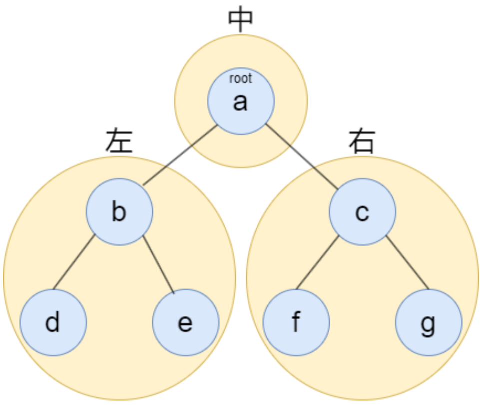
Out[25]: 0

二、遍历算法的递归形式

先介绍3种遍历算法，根据访问中间节点，左子树，右子树的顺序，可以分为先序遍历，中序遍历，后序遍历。
并且这三种遍历算法可以写成函数递归，也可以写成迭代形式，因为迭代形式效率更高，我们重点掌握。
迭代形式实际上多种多样，我们这里每种遍历方式只介绍一种迭代形式的算法。
这里这3种算法实现为外部函数的形式，暂时就不和上面混在一起了，如果想写入节点中，去掉节点指针参数用this代替即可。

其次我们会介绍层次遍历，因为前面三种遍历方式（中左右，左中右，左右中），无一例外都是先处理完左子树，再处理右子树，都是按"左式链"进行游走，而层次遍历，是同一个深度游走，一行一行遍历，不会像前面3种那样上天下地在不同深度上来回穿梭。

In [26]:
```
auto root = new BinNode<char>('a');

auto nodeb = root -> insertAsLC('b');
auto nodec = root -> insertAsRC('c');

auto noded = nodeb -> insertAsLC('d');
auto nodee = nodeb -> insertAsRC('e');

auto nodef = nodec -> insertAsLC('f');
auto nodeg = nodec -> insertAsRC('g');
```



先序遍历的递归形式

根据中左右的访问顺序，应该是
a左右
a(bde)右
a(bde)(cfg)

In [27]:
```
template <class T>
void printNode(const T& data) {
    cout << data << " ";
}
```

In [28]:
```
template <class T, class VST>
void travelPreorder(BinNodePointer(T) p, VST& visit){
    if (!p) return;
    visit(p -> data);
    travelPreorder(p -> lc, visit);
    travelPreorder(p -> rc, visit);
}
```

In [29]:
```
travelPreorder(root, printNode<char>);
```

```
a b d e c f g
```

注意，上面VST&表示任意类型的引用，所以填入一个函数，或者一个模板函数自然没问题

中序遍历的递归形式

根据左中右的访问顺序，应该是
左a右
(dbe)a右
(dbe)a(fcg)

In [30]:
```
template <class T, class VST>
void travelInorder(BinNodePointer(T) p, VST& visit){
    if (!p) return;
    travelInorder(p -> lc, visit);
    visit(p -> data);
    travelInorder(p -> rc, visit);
}
```

In [31]:
```
travelInorder(root, printNode<char>);
```

```
d b e a f c g
```

后序遍历的递归形式

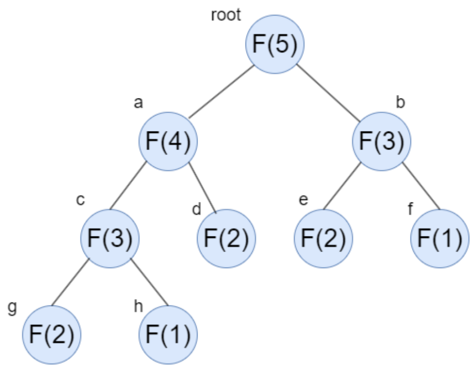
根据左右中的访问顺序，应该是
左右a
(deb)右a
(deb)(fgc)a

In [32]:
```
template <class T, class VST>
void travelPostorder(BinNodePointer(T) p, VST& visit){
    if (!p) return;
    travelPostorder(p -> lc, visit);
    travelPostorder(p -> rc, visit);
    visit(p -> data);
}
```

In [33]:
```
travelPostorder(root, printNode<char>);
```

d e b f g c a

注意，后序遍历的访问顺序经常是我们需要的，比如求Fib数列，F(5) = F(4) + F(3)
如果把F(5)看成父节点，F(4)和F(3)看成左右孩子，那么必须左右计算完成后，才能计算父节点，
这就要求必须按照后序的遍历顺序（当然也可以按照层次遍历的自底向上的顺序，下面一层算完后，再算上层）
此外，这里的遍历是遍历所有节点，实际问题中，存在大量节点冗余，不必访问全部，就比如上面的Fib数列，只有左侧链上面的节点需要遍历求解，其他部分都是重复的。
下面就按普通的后序遍历方式求解F(5),看一下到底怎么用的



In [34]:
```
auto root = new BinNode<int>(0);

auto a = root -> insertAsLC(0);
auto b = root -> insertAsRC(0);
auto c = a -> insertAsLC(0);
auto d = a -> insertAsRC(0);
auto e = b -> insertAsLC(0);
auto f = b -> insertAsRC(0);
auto g = c -> insertAsLC(0);
auto h = c -> insertAsRC(0);
```

In [35]:
```
template <class T>
int travelFib(BinNodePointer(T) p){
    if (!haschild(*p)) {
        p -> data = 1;
        return p -> data;
    }
    auto leftValue = travelFib(p -> lc);     // 先访问左边
    auto rightValue = travelFib(p -> rc);    // 再访问右边
    p -> data = leftValue + rightValue;  // 最后访问中间
    return p -> data;
}
```

In [36]:
```
travelFib(root)
```

Out[36]: 5

In [37]:
```
a -> data
```

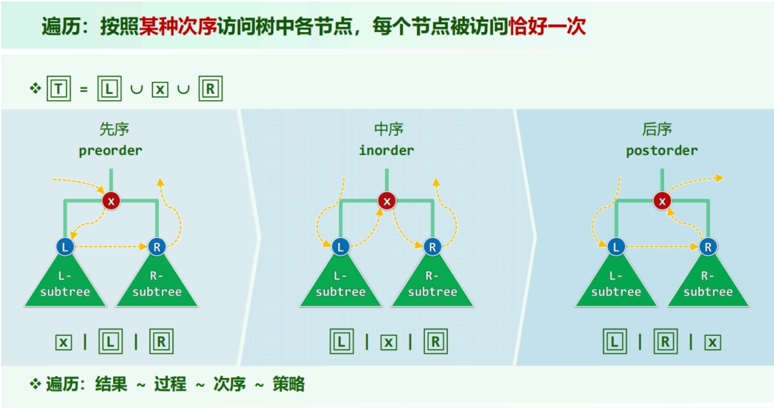
Out[37]: 3

In [38]:
```
c -> data
```

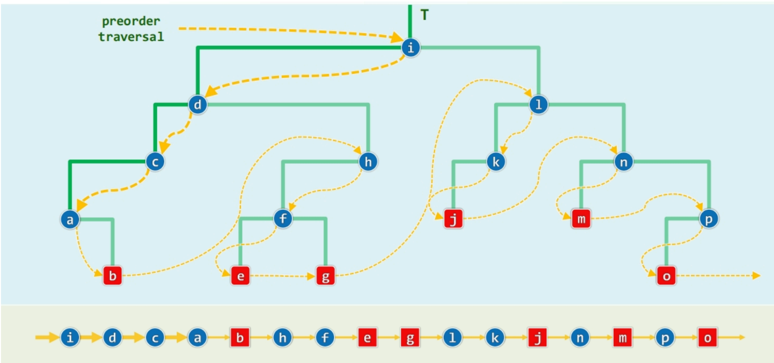
Out[38]: 2

上面的算法思路完全就是后序遍历，因此解决类似问题主要存在下面几个挑战：
1.如何将递归求解问题转化为上面这种树；
2.如何简便表示出来，把上面的树构造出来，显然上面我这种手动构造非常繁琐，
如果能像深度学习中的计算图那样构造就再好不过了；
3.是不是该考虑剔除某些不必要的重复冗余的节点，进而得到一条简单的路径。

三、遍历算法的迭代形式



前序遍历的迭代形式



观察上图不难看出，前序遍历的游走应该是这样的：
1.以root为出发点，沿着左侧链前进，沿途节点直接访问，直到走到左侧尽头；
2.从尽头回头走，并检查右侧是否存在节点子树迷宫：
如果存在，以右侧节点为出发点，探索右子树迷宫(重复执行1，2)；
如果不存在，继续往回走，并检查右侧是否存在尚未探索的区域；
3.直到回到root出发点，并离开，游戏结束。
比如上图，我们是这样探索整棵树的，从i出发，依次经过d-c抵达尽头a，此时前方无路可走，
本想退回，但是发现右侧还有道路，于是按照同样的方式探索右侧，探索完成后，退回到c点，
此时c右侧无路可走，于是退回到d点，然后探索d的右侧h子树，探索完成后回到i，本想直接离开
迷宫，却发现右侧还有l区域没有探索，于是把l也探索完成了，此时所有节点均已探索完毕，离开。

算法实现：
像上面这种走迷宫一样的，存在深度遍历和广度遍历，这里是深度遍历，一直往最深处走，然后往回走，
像这种往回走的都是利用栈来实现的，我们把走过的节点都压入栈中，然后弹出的时候出发点转移至
尚未探索的右侧，进行同样的探索活动(就是从新出发点开始走，并把走过的节点压入栈中,然后弹出...);
既然有个关键的步骤就是沿着左侧走并访问，我们把这个步骤先单独实现，然后再实现这个算法的全部。

In [39]:
```
#include "stack.h"
```

In [40]:
```
template <class T, class VST>
void visitAlongLeft(BinNodePointer(T) x, Stack<BinNodePointer(T)>& S, VST& visit){  // 注意栈一定要用引用，否则每次重新创建
    while(x){
        S.push(x);          // 入栈
        visit(x -> data);   // 访问
        x = x -> lc;        // 往左下走
```

```
    }
}
```

In [41]:
```
template <class T, class VST>
void travelPreorderIter(BinNodePointer(T) x, VST& visit){
    auto S = Stack<BinNodePointer(T)>();
    while(true){
        visitAlongLeft(x, S, visit);   // 沿着左侧探索
        if (S.empty()) break;          // 无路可回的时候，探索结束
        x = S.pop();              // 准备回头，并看看右边有没有探索
        x = x -> rc;
    }
}
```

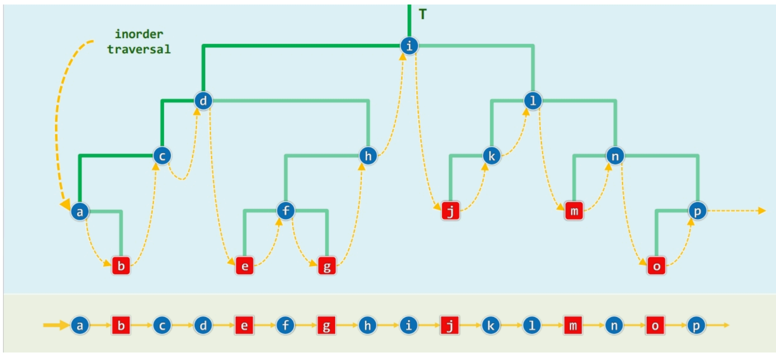
In [42]:
```
auto root = new BinNode<char>('a');

auto nodeb = root -> insertAsLC('b');
auto nodec = root -> insertAsRC('c');

auto noded = nodeb -> insertAsLC('d');
auto nodee = nodeb -> insertAsRC('e');

auto nodef = nodec -> insertAsLC('f');
auto nodeg = nodec -> insertAsRC('g');
```

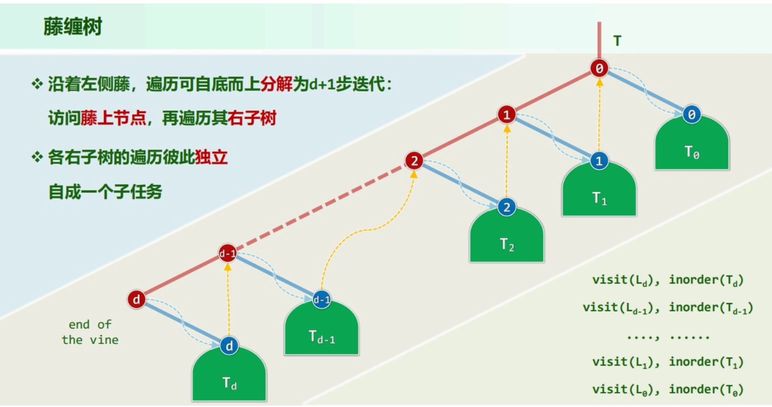
In [43]:
```
travelPreorderIter(root, printNode<char>);
```

```
a b d e c f g
```

当然，上面的visitAlongLeft中，也可以将右侧未访问的子树根节点入栈，
然后弹出时，就把弹出的节点当成探索的起点。
这种算法的套路，就是找到前两个起点即可

中序遍历的迭代形式





观察上图，发现还是沿着左侧链下行，不过这次下行的时候并没有访问节点，而是走回头路的时候进行访问，
参考前面的算法，这里不难写出，具体代码直接给出如下：

In [44]:
```
template <class T>
void goAlongLeft(BinNodePointer(T) x, Stack<BinNodePointer(T)>& S){  // 注意栈一定要用引用，否则每次重新创建，直接卡死
    while(x){
        S.push(x);          // 入栈
        x = x -> lc;        // 往左下走
    }
}
```

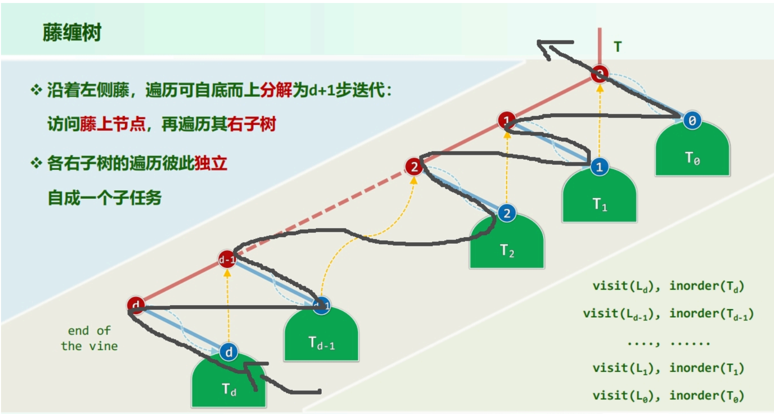
In [45]:
```
template <class T, class VST>
void travelInorderIter(BinNodePointer(T) x, VST& visit){
    auto S = Stack<BinNodePointer(T)>();
    while(true){
```

```
        goAlongLeft(x, S);   // 沿着左侧探索
        if (S.empty()) break;         // 无路可回的时候，探索结束
        x = S.pop(); visit(x -> data);           // 回头，并访问
        x = x -> rc;                 // 设定右边为出发点
    }
}
```

In [46]:
```
travelInorderIter(root, printNode<char>);
```

d b e a f c g

后序遍历的迭代形式



由于书上没有，这里我用黑笔描出游走轨迹，我们只要按照这个轨迹的逆序把节点入栈即可，
注意标记颜色，右侧需要循环进行s型探索，所以回溯的时候检查颜色，看看是否需要进行s型探索
算法实现如下（最新版PPT上有后序的迭代版本，有兴趣可以去看看，这里是我的思路）：

In [47]:
```
template <class T>
void goAlongS(BinNodePointer(T) x, Stack<BinNodePointer(T)>& S){  // 注意栈一定要用引用，否则每次重新创建，直接卡死
    while(x){
        S.push(x);          // 当前节点入栈
        if(x -> rc) {
            S.push(x -> rc); // 当前节点右节点入栈
            if (haschild(*(x -> rc)))         // 右侧不是单节点，而是一颗子树
                x -> rc -> color = black;  // 染成黑色，出栈的时候这颗子树重新探索
        }
        x = x -> lc;        // 往左下走
    }
}
```

In [48]:
```
template <class T, class VST>
void travelPostorderIter(BinNodePointer(T) x, VST& visit){
    auto S = Stack<BinNodePointer(T)>();
    goAlongS(x, S);   // 沿着s入栈
    while(!S.empty()){
        x = S.pop();      // 取出栈顶，也就是上面路径最后一个右节点
        if (x -> color == black) {
            x -> color = white;  // 如果为黑色，以这个节点为起点，重新按s游走,起点作为左侧点入栈，需要变成白色
            goAlongS(x, S);
            continue;
        }
        visit(x -> data);  // 如果这个节点已经是最底下的叶子了，进行访问
    }
}
```

In [49]:
```
travelPostorderIter(root, printNode<char>);
```

d e b f g c a

上面s型走位看起来还是不够直观，现在考虑写一个比较直观容易理解的，我们从三种遍历算法的访问顺序出发
三种迭代算法比较：
前序和中序，都是先弹出左侧节点，再处理右边，但是后序不能轻易把左侧节点弹出，因为后序是右边先处理，
因此我们这样设计算法：
1.左侧节点全部入栈；
2.检查栈顶，也就是最后一个左侧节点，看看右方是否有未处理的节点或者子树：
如果是一个未处理的节点，访问它，然后右边算是搞定了，这时候栈顶弹出，并访问；
如果是一个未处理的子树，控制权交给这个子树的根，重复前面的步骤；
所以这里是先转移控制权处理右边，再弹出自己，与前序中序正好相反，前序中序go完直接弹出，这里go完可能还要继续go

In [50]:
```
template <class T>
void goAlongLeft2(BinNodePointer(T) x, Stack<BinNodePointer(T)>& S){  // 注意栈一定要用引用，否则每次重新创建，直接卡死
    while(x){
        S.push(x);          // 当前节点入栈
```

```
        if(x -> rc) {
            if (haschild(*(x -> rc)))         // 右侧不是单节点，而是一颗子树
                x -> rc -> color = black;  // 染成黑色，出栈的时候这颗子树重新探索
        }
        x = x -> lc;        // 往左下走
    }
}
```

In [51]:
```
template <class T, class VST>
void travelPostorderIter2(BinNodePointer(T) x, VST& visit){
    auto S = Stack<BinNodePointer(T)>();
    goAlongLeft2(x, S);   // 沿着s入栈
    while(!S.empty()){
        x = S.top();      // 栈顶不急着弹出，先处理右边
        auto rchild = x -> rc;
        if (rchild && !(rchild -> visited)){              // 根据右孩子是子树还是叶子，分类讨论
            if (rchild -> color == black) {
                rchild -> color = white;  // 右孩子是颗子树，继续go
                goAlongLeft2(rchild, S);
                continue;
            }
            visit(rchild -> data);  // 右孩子已经是最底下的叶子了，访问右节点
            rchild -> visited = true;
        }
        visit(x -> data);
        x -> visited = true;
        S.pop();
    }
}
```

In [52]:
```
travelPostorderIter2(root, printNode<char>);
```

d e b f g c a

层次遍历

In [53]:
```
#include "queue.h"
template <class T, class VST>
void _travelLevel(BinNodePointer(T) x, VST& visit){
    Queue<BinNodePointer(T)> q;
    q.enqueue(x);
    while(!q.empty()){
        auto t = q.dequeue();
        visit(t -> data);
        if(t -> lc) q.enqueue(t -> lc);
        if(t -> rc) q.enqueue(t -> rc);
    }
}
```

In [54]:
```
_travelLevel(root, printNode<char>);
```

a b c d e f g

上面的遍历次序是自顶而下，如果再加个栈，便可实现从底下到顶部的一层一层访问，
实际就是利用栈逆序输出

In [55]:
```
template <class T, class VST>
void _travelLevel2(BinNodePointer(T) x, VST& visit){
    Queue<BinNodePointer(T)> q;
    Stack<BinNodePointer(T)> s;
    q.enqueue(x);
    while(!q.empty()){
        auto t = q.dequeue();
        s.push(t);
        if(t -> lc) q.enqueue(t -> lc);
        if(t -> rc) q.enqueue(t -> rc);
    }
```

```
    while(!s.empty()){
        auto t = s.pop();
        visit(t-> data);
    }
}
```

In [56]:
```
_travelLevel2(root, printNode<char>);
```

g f e d c b a

四、树的定义

实际上前面节点已经把该实现的都实现了，这里定义的树，仅仅是定义一个root节点，以及树的size属性；
构造函数： 就是创建一个单root节点，size=1；高度根据节点的默认初始化为0；
插入删除： 照搬节点的插入算法，需要提供插入到哪个节点，插入的值，对于每个节点的高度，
已经内置的更新高度算法，对于树的size属性，插入一次+1
遍历算法： 直接调用前面的遍历算法，传入root，遍历这棵树。

In [57]:
```
template <class T>
class BinTree{
    public:
        int size;
        BinNodePointer(T) root;

        BinTree(T e){
            root = new BinNode<T>(e);
        }

        BinNodePointer(T) insertAsLC(BinNodePointer(T) x, T e);
        BinNodePointer(T) insertAsRC(BinNodePointer(T) x, T e);
        BinNodePointer(T) remove(BinNodePointer(T) x);

        template <class VST> void travePre(VST& visit);
        template <class VST> void travIn(VST& visit);
        template <class VST> void travPost(VST& visit);
        template <class VST> void travLevel(VST& visit);
};
```

插入删除

In [58]:
```
template <class T>
BinNodePointer(T) BinTree<T>::insertAsLC(BinNodePointer(T) x, T e){
    size++;
    return x -> insertAsLC(e);
}
```

In [59]:
```
template <class T>
BinNodePointer(T) BinTree<T>::insertAsRC(BinNodePointer(T) x, T e){
    size++;
    return x -> insertAsRC(e);
}
```

In [60]:
```
template <class T>
BinNodePointer(T) BinTree<T>::remove(BinNodePointer(T) x){
    size--;
    return x -> remove();
}
```

构建一棵简单的树

In [61]:
```
auto tree1 = BinTree<char>('a')
```

In [62]:
```
auto nodeb = tree1.insertAsLC(tree1.root,'b');
auto nodec = tree1.insertAsRC(tree1.root,'c');

auto noded = tree1.insertAsLC(nodeb,'d');
auto nodee = tree1.insertAsRC(nodeb,'e');

auto nodef = tree1.insertAsLC(nodec,'f');
auto nodeg = tree1.insertAsRC(nodec,'g');
```

遍历
这里只写一种，其他完全类似

In [63]:
```
template <class T>
template <class VST>
void BinTree<T>::travLevel(VST& visit){
    if (root) _travelLevel(root, visit);
}
```

In [64]:
```
tree1.travLevel(printNode<char>)
```

```
a b c d e f g
```

五、遍历算法的意义

这里简单提两句，主要是要有这个意识，正如前面讲解Fib数列的问题，很多问题的解集可以
抽象为一棵二叉树进行解决，这就是第一层抽象，这个意识相当重要；
其二，上面看似遍历来访问所有的节点，实际你查看最后的输出，就是一个序列，也就是遍历
实际上还是一种能将树这种半线性的结构转换为一个线性序列，然后针对这个线性序列做查找排序等算法。
这里所说的第二个意识，就是每种数据结构（向量，列表，二叉树，图），存在着一定的转换关系，
如果在自己的这种数据结构下进行某种算法行不通，可以转换成其他结构考虑。

In [ ]:

In [ ]:

In [ ]: