### 平摊分析

#### 平摊分析的目的

算法的[时间复杂度](https://so.csdn.net/so/search?q=%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_51059141/article/details/_blank)不好分析时，将总的代价平摊到每个操作上来分析总的时间复杂度的分析方法  
注：都要按最坏的情况算

#### 2.平摊分析的常用方法

**聚集法**：数学计算+结合具体场景更细致地分析，确定n个操作的总代价上界为T（n），因此每个操作的平均代价为T（n）/n

**会计法**：为每种操作分配不同的平摊代价，操作被执行时，若实际代价小于分配的平摊代价，则剩余的部分作为存款存到银行里；若实际代价大于分配的平摊代价，则从银行里取出“存款”来支付不足的代价。只要保证银行里的存款永远是非负的，则所有平摊代价求和就是算法的时间复杂度的上界

**势能法**：定义整个数据结构的势能函数。为每种操作分配不同的平摊代价，操作被执行时，若实际代价小于分配的平摊代价，则整个数据结构的势能增加；若实际代价大于操作的平摊代价，要用数据结构的一部分势能来支付不足的代价，整个数据结构的势能减少；只要满足n次操作后整体的势能大于初始的势能，则所有平摊代价求和就是算法时间复杂度的上界

平摊分析是一种用来分析执行一系列类似操作的算法的工具，在这种方法中,不是通过分别计算每一次操作的真实代价界确定一系列操作代价的界,而是对整个操作序列的真实代价限界

这种方法之所以奏效,原因之一是在一个操作序列中,不可能每一个都以其已知的最坏情况时

间界运行。可能某些操作的代价高些,而其他的则可能低些

平摊分析不仅仅是一种分析工具,它也是算法设计的一种思维方式,因为算法的设计和对其运

行时间的分析经常是紧密相连的

#### 3.例子

问题描述:有如下操作的栈,初始为空,分析其执行n次操作的时间复杂度是多少。

● PUSH(S,x):将对象x压入栈S。

● POP(S):弹出S的栈顶,返回弹出的对象。

● MULTIPOP(S,k):弹出S的k个栈顶对象,或者在s中不足k个对象时全部弹出。

如果孤立地分析,那么PUSH、POP操作为0(1),而由于1≤k≤n,所以MULTIPOP操作为O(n)。因此,n次操作的时间复杂度就是0(n2)

但若分析整体,则调用POP的次数至多等于PUSH的次数（即n/2次push和n/2次pop）,因此n次操作的时间复杂度只有0(n)。

**（1）聚集法**

n次操作中push的操作小于n次，pop的操作（multipop可以看成是多次pop叠加），则pop的操作要小于等于push的操作，设T ( n ) 为 总 的 操 作 的 代 价，则T ( n )等于总的push的代价+总的pop的 代 价 （ 包 括multipop），所以 T ( n ) ≤ 2 n T(n)，n次操作每次的平摊代价为O ( 1 )

1. **会计法**

为不同的操作分配不同的平摊代价：

PUSH:2

POP:0

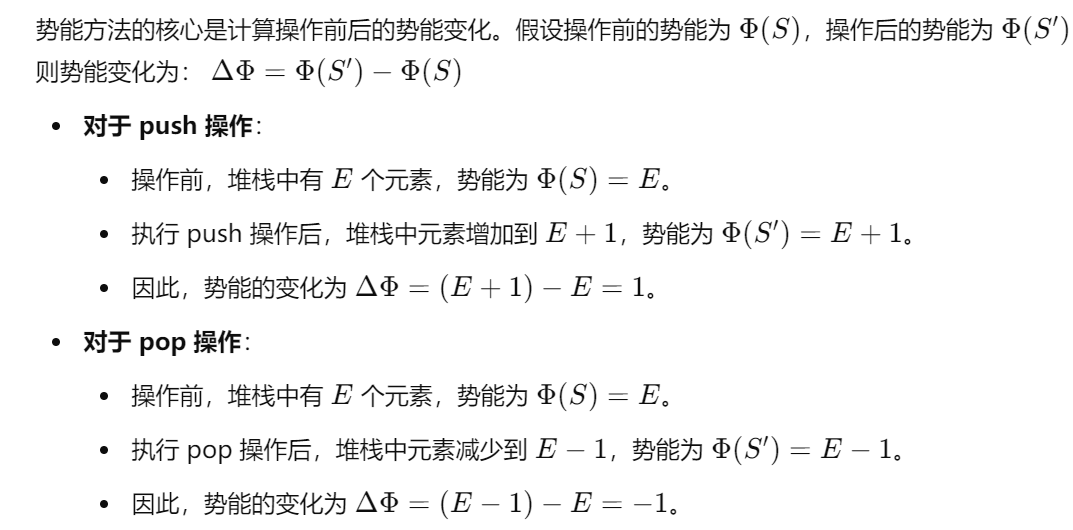
MULTIPOP:0

每push一次支付平摊代价2, 1作为实际代价，1作为存款存放在push的数据对象上；每pop一次支付平摊代价0（multipop）看成是多次pop，pop的实际代价由数据对象上的存款来支付

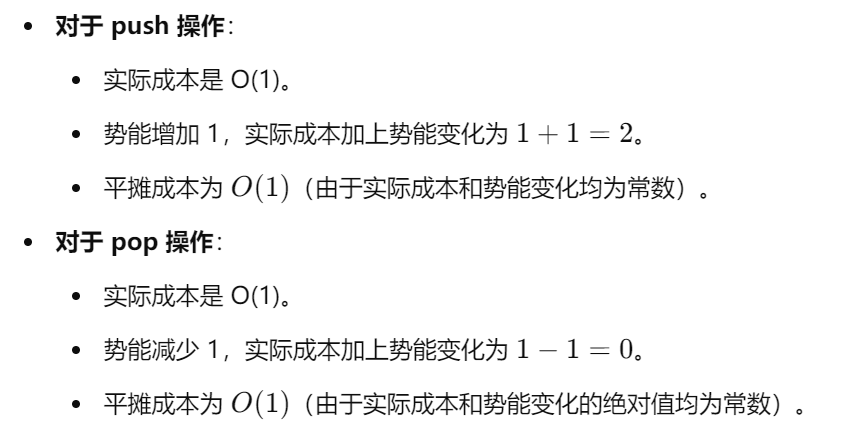
n次操作中push的次数小于等于n，故平摊代价的总和小于等于2n

1. **势能法**

在堆栈中，我们定义势能函数来反映堆栈中元素的数量。具体地，势能函数可以定义为堆栈中元素的数量 E。即：Φ(S)=E 其中，S 表示堆栈的状态，E是堆栈中当前的元素数。



计算操作的平摊成本：



### 二叉树的遍历

#### 理解问题

已知二叉树的根节点，遍历二叉树的时间性能为O（n），空间性能为O(1)，n为节点数。

二叉树的遍历通常分为前序遍历、中序遍历、后序遍历、层序遍历四种情况，只是打印顺序的不同，复杂度均相同。

递归遍历：时间复杂度为O(n)，空间复杂度为O(n)；

非递归遍历（辅助栈）：时间复杂度为O(n)，空间复杂度为O(n)；

#### 算法设计

利用Morris遍历可以达到时间复杂度为O(n)，空间复杂度为O(1)。因为这种方式利用了二叉树节点中大量指向NULL的指针，来达到节省空间的目的。

**自然语言描述：**

开始cur指向根结点

如果cur没有左孩子则cur=cur->right向右移动

如果cur有左孩子，找到左子树上最右的节点mostright;

如果mostright右指针为空且不指向cur时，说明是第一次来到cur，则mostright->right=cur，cur=cur->left

如果mostright右指针指向cur，说明是第二次来到cur，则mostright->right=null, cur=cur->right;

当cur为空的时候遍历停止。（用while循环控制，不用递归）

**实质：**

对于没有左子树的节点只到达一次，有左子树的节点会到达两次，该遍历的时间复杂度仍然是O(n)。

**通过该方法加工出先序、中序、后序遍历：**

**先序：**

对于可以访问到两次的节点，第一次访问时打印，第二次忽略；对于只访问一次的节点，直接处理打印。

**中序：**

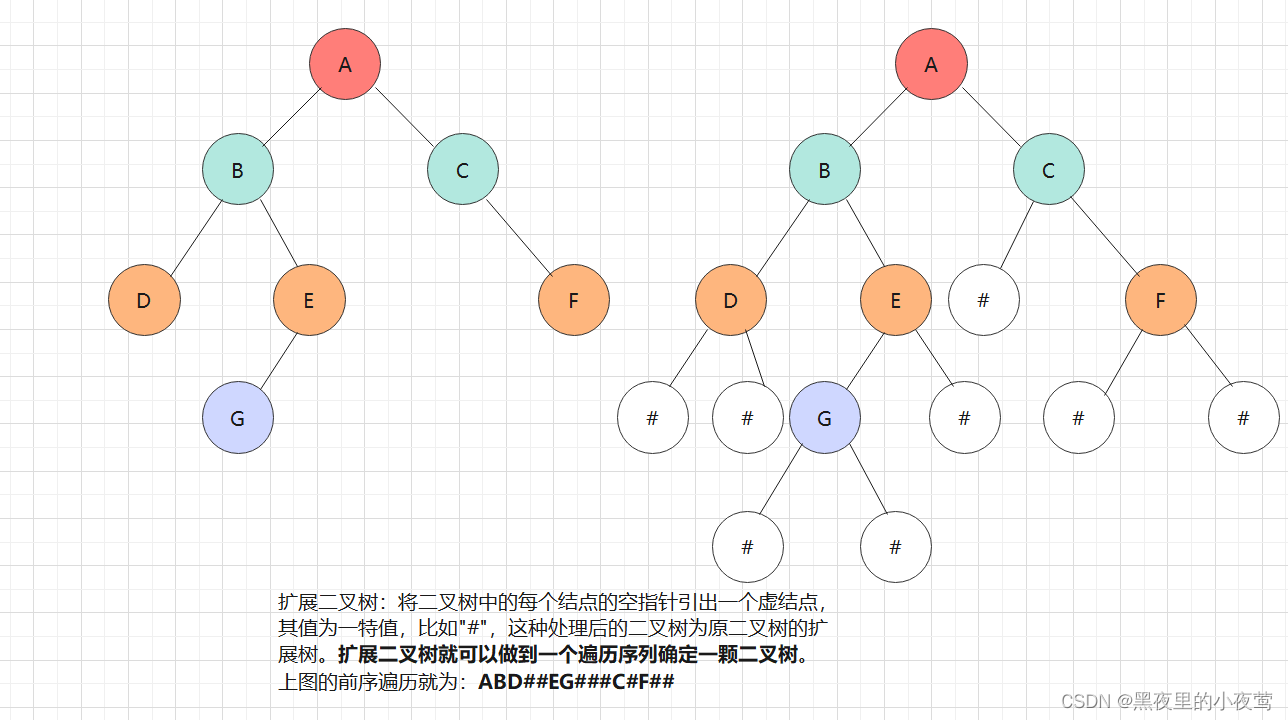
对于可以访问到两次的节点，第一次访问时忽略，第二次打印；对于只访问一次的节点，直接处理打印。

**后序：**

处理时机放在能访问两次的节点的第二次访问上，当一个节点是第二次被访问时，逆序打印左树到右边界

其中，二叉树的创建采用扩展二叉树的方法。

**扩展二叉树**：将二叉树中的每个结点的空指针引出一个虚结点，其值为一特值，比如"#"，这种处理后的二叉树为原二叉树的扩展树。扩展二叉树就可以做到一个遍历序列确定一颗二叉树。如下图



此方法只需要一个遍历序列即可确定一个二叉树，而使用普通遍历序列需要先序+中序或者后序+中序来确定。

#### 算法代码

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

typedef int Status;

typedef char TElemType;

//二叉树的链式存储：二叉链表

typedef struct BiNode{

    TElemType data;

    struct BiNode \*lchild, \*rchild;

}BiNode,\*BiTree;

//如果当前节点左树为空，则直接打印；如果左树不为空，在第一次访问的时候打印

void MorrisPreOrder(BiNode \*root)

{

    if (root==NULL)

    {

        printf("The tree is empty");

        return;

    }

    //开始cur指向根结点

    BiNode\* cur=root;

    //这里没初始化为空

    BiNode\* mostright=NULL;

    //用while循环控制 不用递归 当cur为空就停止

    while (cur!=NULL)

    {

        mostright=cur->lchild;

        //如果cur没有左孩子则cur=cur->right向右移动

        if (mostright!=NULL)

        {

            //如果cur有左孩子，找到左子树上最右的节点mostright;

            while(mostright->rchild!=NULL&&mostright->rchild!=cur)

            {

                mostright=mostright->rchild;

            }

            //第一次来到cur,直接打印输出进行访问

            if (mostright->rchild==NULL)

            {

                //先打印再移动

                printf("%c ",cur->data);

                mostright->rchild=cur;

                cur=cur->lchild;

                continue;

            }else

            {   //mostright右指针指向cur，说明是第二次来到cur，忽略不访问

                //所以将mostright右指针置空，cur向右移动

                mostright->rchild=NULL;

            }

        }else{

            //没有左树，直接打印

            printf("%c ",cur->data);

        }

        cur=cur->rchild;

    }

}

//如果当前节点左树为空，则直接打印；如果左树不为空，在第二次访问的时候打印

void MorrisInOrder(BiNode \*root)

{

    if (root==NULL)

    {

        printf("The tree is empty");

        return;

    }

    //开始cur指向根结点

    BiNode\* cur=root;

    //这里没初始化为空

    BiNode\* mostright=NULL;

    //用while循环控制 不用递归 当cur为空就停止

    while (cur!=NULL)

    {

        mostright=cur->lchild;

        //如果cur没有左孩子则cur=cur->right向右移动

        if (mostright!=NULL)

        {

            //如果cur有左孩子，找到左子树上最右的节点mostright;

            while(mostright->rchild!=NULL&&mostright->rchild!=cur)

            {

                mostright=mostright->rchild;

            }

            //第一次来到cur,不打印

            if (mostright->rchild==NULL)

            {

                mostright->rchild=cur;

                cur=cur->lchild;

                continue;

            }else

            {   //mostright右指针指向cur，说明是第二次来到cur，访问

                //所以将mostright右指针置空，cur向右移动

                printf("%c ",cur->data);

                mostright->rchild=NULL;

            }

        }else{

            //没有左树，直接打印

            printf("%c ",cur->data);

        }

        cur=cur->rchild;

    }

}

//建立二叉树 先序  递归调用

//读入ABC##DE#G##F###

Status CreateBiTree(BiTree\* T) //T是指针的地址 \*T是根节点指针

{

    TElemType ch;

    //递归里不用while循环

    scanf("%c",&ch);

    if(ch=='#')T=NULL;

    else{

        \*T=(BiNode\*)malloc(sizeof(BiNode));

        if (!\*T)

        {

            printf("分配空间错误");

            exit(FALSE);

        }

        (\*T)->data=ch;                 //生成根结点

        (\*T)->lchild=NULL;

        (\*T)->rchild=NULL;

        CreateBiTree(&(\*T)->lchild);    //构造左子树

        CreateBiTree(&(\*T)->rchild);    //构造右子树

    }

    return OK;

}

//二叉树的递归先序遍历

Status PreOrderTraverse(BiTree T)

{

    if(T==NULL)return OK;//空二叉树

    else{

        printf("%c ",T->data);;//访问根节点 例如 printf("%d ",T->data);

        PreOrderTraverse(T->lchild);//递归遍历左子树

        PreOrderTraverse(T->rchild);//递归遍历右子树

    }

}

//二叉树的递归中序遍历

Status InOrderTraverse(BiTree T)

{

    if(T==NULL)return OK;//空二叉树

    else{

        InOrderTraverse(T->lchild);//递归遍历左子树

        printf("%c ",T->data);;//访问根节点 例如 printf("%d ",T->data);

        InOrderTraverse(T->rchild);//递归遍历右子树

    }

}

测试程序：

int main(){

    BiTree T=NULL;

    printf("Please input a sequence of binary tree nodes(eg:ABC##DE#G##F###):\n");

    CreateBiTree(&T);//取的指针的地址

    printf("PreOrder traversal: ");  // 先序  ABCDEGF

    PreOrderTraverse(T);

    printf("\n");

    printf("Morris PreOrder traversal: ");

    MorrisPreOrder(T);

    printf("\n");

    printf("InOrder traversal: ");  // 先序  ABCDEGF

    InOrderTraverse(T);

    printf("\n");

    printf("Morris InOrder traversal: ");

    MorrisInOrder(T);

    return 0;

}

测试结果：

