高级数据结构-二叉树

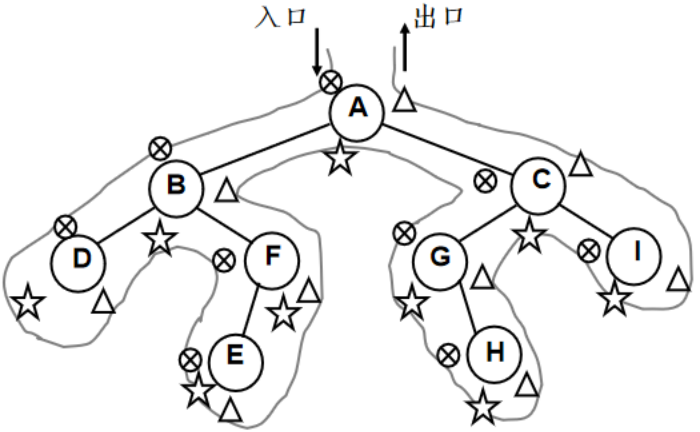
信智学院　陈琰宏

主要内容

6.1 二叉树的非递归遍历

- 从二叉树先序、中序和后序的遍历过程的遍历路径来看，都是从根结点A开始的，且在遍历过程中经过结点的路线是一样的，只是访问各结点的时机不同而已。

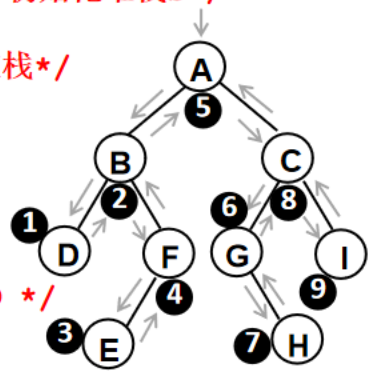
- 如图所示，并在从入口到出口的曲线上用⊗、☆ 和△三种符号分别标记出了先序、中序和后序遍历各结点的时刻。



6.1.1 中序遍历的非递归实现

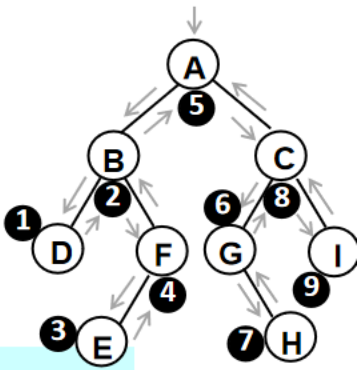
- 遇到一个结点，就把它压栈，并去遍历它的左子树；
- 当左子树遍历结束后，从栈顶弹出这个结点并访问它；
- 然后按其右指针再去中序遍历该结点的右子树。

```
void InOrderTraversal( BinTree BT )
{    BinTree T=BT;//表示当前节点
     Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈s*/
     while( T || !IsEmpty(S) ){
          while(T){    /*一直向左并将沿途结点压入堆栈*/
               Push(S,T);
               T = T->Left;
          }
          if(!IsEmpty(S)){
               T = Pop(S); /*结点弹出堆栈*/
               printf("%5d", T->Data); /*（访问）*/
               T = T->Right; /*转向右子树*/
          }
     }
}
```



6.1.1 中序遍历的非递归实现

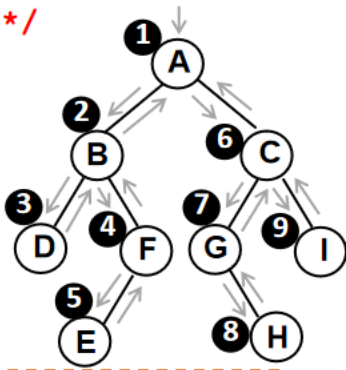
```
void InOrder21(BTNode *t)    //中序遍历的非递归算法
{
    BTNode *st[MaxSize];          //定义一个顺序栈
    int top=-1;                   //栈顶指针初始化
    BTNode *p=t;
    while (top!=-1 || p!=NULL)    //栈不空或者p不空时循环
    {
        while (p!=NULL)           //扫描*p的所有左结点并进栈
        {
            top++; st[top]=p;
            p=p->lchild;
        }
        if (top>-1)               //若栈不空
        {
            p=st[top]; top--;     //出栈*p结点
            cout << p->data;      //访问*p结点
            p=p->rchild;          //转向处理右子树
        }
    }
}
```



6.1.2 先序遍历的非递归实现

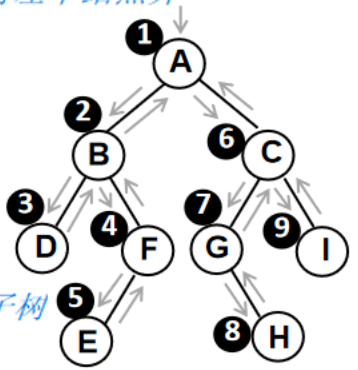
- 遇到一个结点，就把它压栈并访问它，然后去遍历它的左子树；
- 当左子树遍历结束后，从栈顶弹出这个结点；
- 然后按其右指针再去中序遍历该结点的右子树。

```
void InOrderTraversal( BinTree BT )
{   BinTree T=BT;
    Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈s*/
    while( T || !IsEmpty(S) ){
        while(T){    /*一直向左并将沿途结点压入堆栈*/
            printf("%5d", T->Data); /*（访问）*/
            Push(S,T);
            T = T->Left;
        }
        if(!IsEmpty(S)){
            T = Pop(S); /*结点弹出堆栈*/
            T = T->Right; /*转向右子树*/
        }
    }
}
```



6.1.2 先序遍历的非递归实现

```
void PreOrder31(BTNode *t)          //先序遍历的非递归算法2
{
    BTNode *st[MaxSize];            //定义一个顺序栈
    int top=-1;                     //栈顶指针初始化
    BTNode *p=t;
    while (top!=-1 || p!=NULL)
    {
        while (p!=NULL)             //访问*p及其所有左下结点并
        {
            cout << p->data;        //访问*p结点
            top++;  st[top]=p;
            p=p->lchild;
        }
        if (top!=-1)                //若栈不空
        {
            p=st[top];top--;        //出栈*p结点
            p=p->rchild;            //转向处理其右子树
        }
    }
}
```



6.1.3 后序遍历的非递归实现

❖后序遍历非递归遍历算法

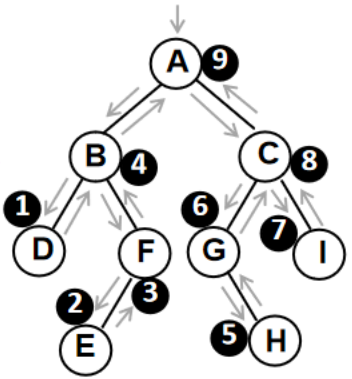
- 遇到一个结点，就把它（附带标志0以后）压栈，并去遍历它的左子树；
- 当左子树遍历结束后，检查栈顶元素的附带标志是否为0；
- 若标志为0，则把标志改成1，并按其右指针再去遍历该结点的右子树；
- 若标志为1，则从栈顶弹出这个结点并访问它。

1. 初始flag=0
2. 如果左子树访问完，记flag=1; 访问右子树
3. 如果右子树为空或者右子树已经访问则访问该节点

p->lchild=q；（此处q=NULL）

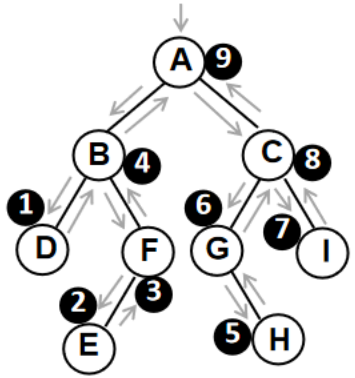
p->lchild=q;（此处q为上一个访问点；

如果上一个访问点是右孩子，表明右孩子访问完）



6.1.3 后序遍历的非递归实现

```
//---------------后序遍历算法----------------------
void PostOrder21(BTNode *t)      //被PostOder2函数调用//后序遍历的非递归算法
{
    BTNode *st[MaxSize];              //定义一个顺序栈
    int top=-1;                       //栈指针置初值
    BTNode *p=t,*q;
    bool flag;                        //若当前结点的左子树已处理则为true,否则为false
    do
    {
        while (p!=NULL)               //将*p结点及其所有左下结点进栈
        {
            top++; st[top]=p;
            p=p->lchild;
        }
        q=NULL;                       //q指向栈顶结点的前一个已访问的结点或为NULL
        flag=true;                    //表示*p结点的左子树已遍历或为空
        while (top!=-1 && flag==true)
        {
            p=st[top];                //取出当前的栈顶结点
            if (p->rchild==q)         //若*p结点右子树已访问或为空
            {
                cout << p->data;      //访问*p结点
                top--;                //结点访问后退栈
                q=p;                  //让q指向刚被访问的结点
            }
            else                      //若*p结点右子树没有遍历
            {
                p=p->rchild;          //转向处理其右子树
                flag=false;           //此时*p结点的左子树未遍历
            }
        }
    } while (top!=-1);
}
```



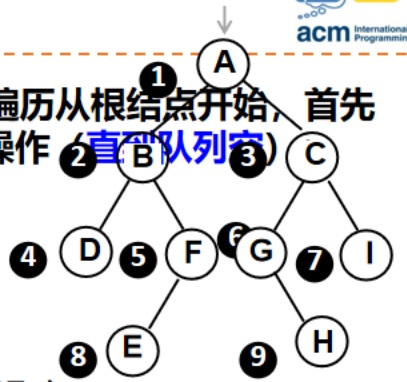
6.1.4 层次遍历的非递归实现

工作队列：见可以设置一个队列结构，遍历从根结点开始，首先将根结点指针入队，然后开始执行下面三个操作，直至队列空：

① 从队列中取出一个元素；

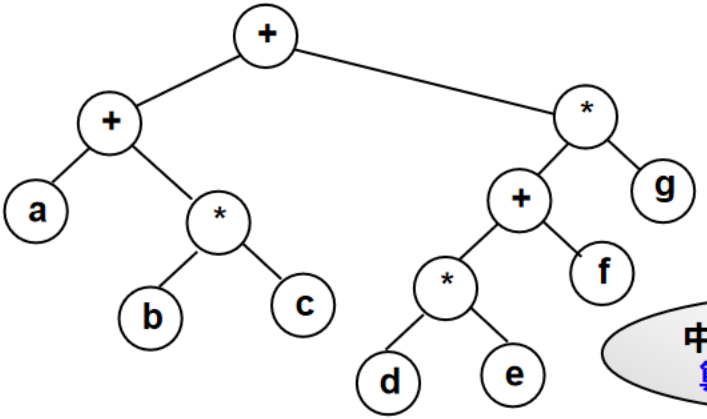
② 访问该元素所指结点；

层序遍历 => 若该结点的左、右孩子结点非空，则将其左、右孩子的指针顺序入队。




```
void LevelOrderTraversal ( BinTree BT )
{   Queue Q;  BinTree T;
    if ( !BT ) return; /* 若是空树则直接返回 */
    Q = CreatQueue( MaxSize ); /*创建并初始化队列Q*/
    AddQ( Q, BT );
    while ( !IsEmptyQ( Q ) ) {
        T = DeleteQ( Q );
        printf("%d\n", T->Data); /*访问取出队列的结点*/
        if ( T->Left )  AddQ( Q, T->Left );
        if ( T->Right )  AddQ( Q, T->Right );
    }
}
```

【例4.5 】 二元运算表达式树及其遍历。



中缀表达式会受到运算符优先级的影响

- 三种遍历可以得到三种不同的访问结果：
  - 中序遍历得到中缀表达式：$a + b * c + d * e + f * g$
  - 先序遍历得到前缀表达式：$+ + a * b c * + * d e f g$
  - 后序遍历得到后缀表达式：$a b c * + d e * f + g * +$

6.2 二叉树的确定

由两种遍历序列确定二叉树

已知三种遍历中的任意两种遍历序列，能否唯一确定一棵二叉树呢?

答案是：必须要有中序遍历才行!

- 没有中序的困扰：
  - 先序遍历序列：A B
  - 后序遍历序列：B A

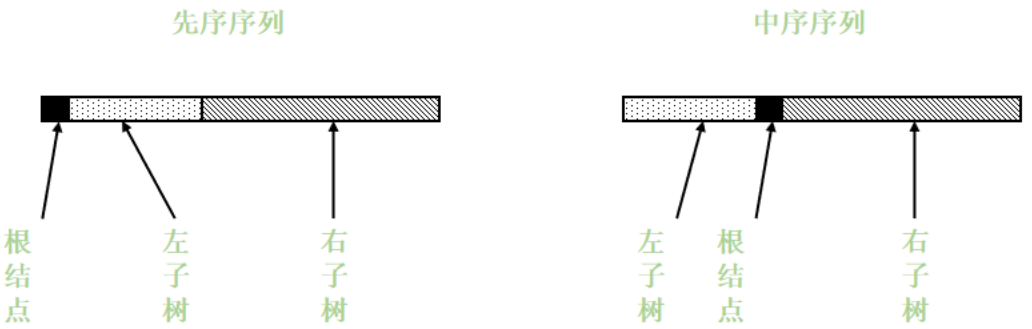
6.2.1 先序和中序遍历序列来确定一棵二叉树

〖分析〗 先序遍历序列的第一个结点就是根结点；

这个根结点能够在中序遍历序列中将其余结点分割成两个子序列，根结点前面部分是左子树上的结点，而根结点后面的部分是右子树上的结点。

根据这两个子序列，在先序序列中找到对应的左子序列和右子序列，它们分别对应左子树和右子树。

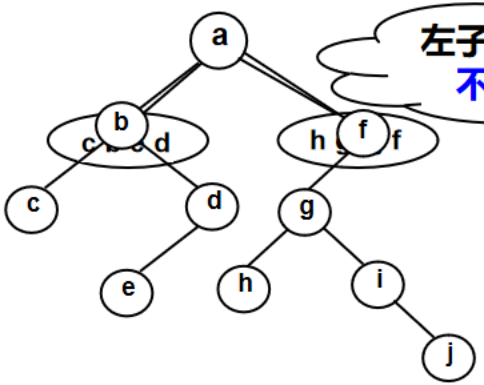
然后对左子树和右子树分别递归使用相同的方法继续分解。



6.2.1 先序和中序遍历序列来确定一棵二叉树

〖例〗 先序序列： a b c d e f g h i j

中序序列： c b e d a h g i j f



左子树（或右子树）序列不配套意味着什么?

❖ 类似地，配套的后序和中序遍历序列也可以确定一棵二叉树。

分析

递归求解

递归建立整棵二叉树：先递归创建左右子树，然后创建根节点，并让指针指向两棵子树。具体步骤如下：

1、先利用先序遍历找根节点：先序遍历的第一个数，就是根节点的值；在中序遍历中找到根节点的位置 k，则 k 左边是左子树的中序遍历，右边是右子树的中序遍历；假设左子树的中序遍历的长度是 l，则在前序遍历中，根节点后面的 l 个数，是左子树的先序遍历，剩下的数是右子树的先序遍历；有了左右子树的先序遍历和中序遍历，我们可以先递归创建出左右子树，然后再创建根节点。

例 [3045] 求先序排列

给出一棵二叉树的中序与后序排列。求出它的先序排列。（约定树结点用不同的大写字母表示，长度<=8）。

输入

每个测试文件只包含一组测试数据，每组输入包含两行，第一行输入一个字符串表示二叉树的中序排列，第二行输入一个字符串表示二叉树的后序排列。

输出

对于每组输入数据，输出二叉树的先序排列。

样例输入

BADC

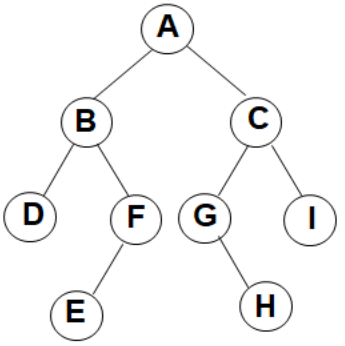
BDCA

样例输出

ABCD

[3045] 求先序排列-写法1

```
#include<bits/stdc++.h>
using namespace std;
char s1[100],s2[100];
/*
DBEFAGHCI
DEFBHGICA
*///ABDFECGHI
//          中序的左右端点 后续的左右端点
void solve(int l1,int r1,int l2,int r2){
    int i,j,flag=0;
    for(i=r2;i>=l2;i--){ //后续第一个字符为树根
        for(j=l1;j<=r1;j++){
            if(s2[i]==s1[j]){//在中序中找到根的位置
                cout<<s1[j];
                flag=1;
                break;
            }
        }
        if(flag==1)break;
    }
    if(j>l1)solve(l1,j-1,0,j-1);//继续递归求解
    if(j<r1)solve(j+1,r1,j-1,r2-1);
}
int main(){
    cin>>s1>>s2;
    int len1=(strlen(s1)),len2=strlen(s2);
    solve(0,len1-1,0,len2-1);
    return 0;
}
```



[3045] 求先序排列-写法2

```
#include<iostream>
#include<cstring>
using namespace std;
void change(string begin,string end){
    if(begin.empty())return;
    char ch=end[end.size()-1];
    int k=begin.find(ch);
    cout<<ch;
    change(begin.substr(0,k),end.substr(0,k));
    change(begin.substr(k+1,begin.size()-k-1),end.substr(k,begin.size()-k-1));
}

int main(){
    string middle,last;
    cin>>middle>>last;
    change(middle,last) ;
    return 0;
}
```

[3045] 求先序排列-写法3

```cpp
#include <iostream>
#include <string.h>
using namespace std;
void out (char a[], char b[], int s1, int s2, int len) {
    if (len) {
        cout << b[s2 + len - 1];
    }
    int x = b[s2 + len - 1];
    for (int i = s1; i < s1 + len; i++) {
        if (a[i] == x) {
            out(a, b, s1, s2, i - s1);
            out(a, b, i + 1, s2 + i - s1, len - i + s1 - 1);
        }
    }
    return;
}
int main () {
    char a[10], b[10];
    cin >> a >> b;
    out(a, b, 0, 0, strlen(a));
    return 0;
}
```

[3045] 求先序排列-写法4

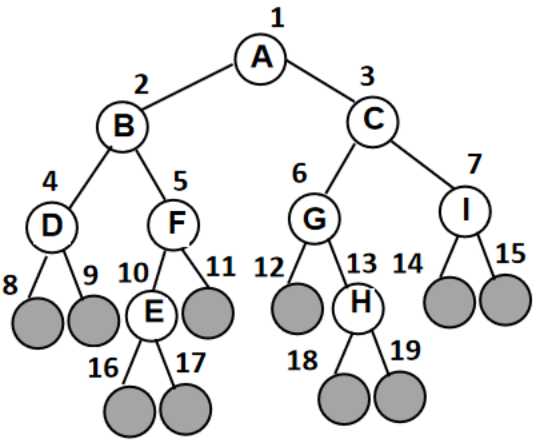
```
#include <iostream>
#include<string>
using namespace std;
string post,mid;
struct tree//二叉树
{
    char letter;
    tree* left=NULL;
    tree* right=NULL;

};
void output(tree *p)//先序输出
{
    if(p==NULL)return;
    cout<<p->letter;
    output(p->left);output(p->right);
}
char buildTree(string &post,string mid,tree *p)//建立二叉树
{
int main()
 {
    cin>>mid>>post;
    tree *p=new tree;
    buildTree(post,mid,p);
    output(p);
    return 0;
}
```

[3045] 求先序排列-写法4

```
18  char buildTree(string &post,string mid,tree *p)//建立二叉树
19  {
20      char postLast=post[post.size()-1];//取后序排列中最后一个字符
21      for(int i=0;i<mid.size();i++)
22      {
23          if(mid[i]==postLast)//寻找根
24          {
25              string left(mid,0,i);//分出左树
26              string right(mid,i+1);//分出右树
27              //建立子树
28              if(left.size()!=0)
29                  p->left=new tree;
30              if(right.size()!=0)
31                  p->right=new tree;
32
33              p->letter=postLast;//赋值
34              post.erase(post.size()-1);
35              //抹去后序排列的最后一个字符，这样倒数第二个字符成为最后一个字符
36              buildTree(post,right,p->right); //继续递归
37              buildTree(post,left,p->left);
38          }
39      }
40
41      return 0;
42  }
```

6.2.2 二叉树的创建

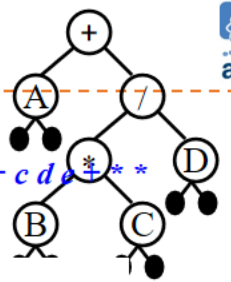
- 常用的方法是先序创建和层序创建两种。



先序创建的输入序列： A, B, D, 0, 0, F, E, 0, 0, 0, C, G, 0, H, 0, 0, I, 0, 0

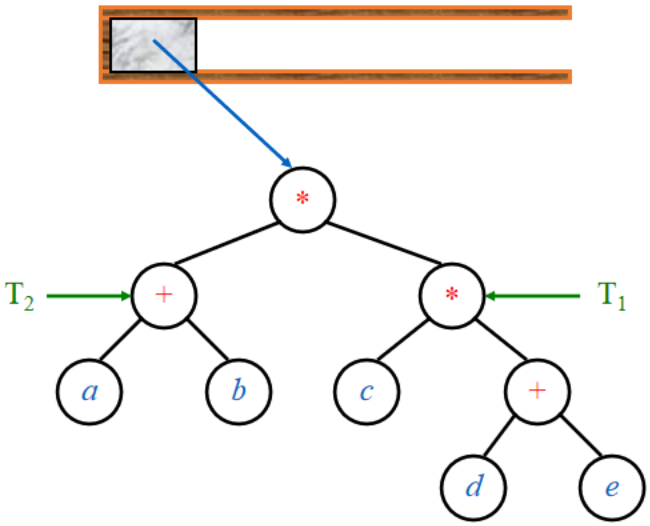
层序创建的输入序列： A, B, C, D, F, G, I, 0, 0, E, 0, 0, H, 0, 0, 0, 0, 0, 0

❖表达式树的构造(语义树)



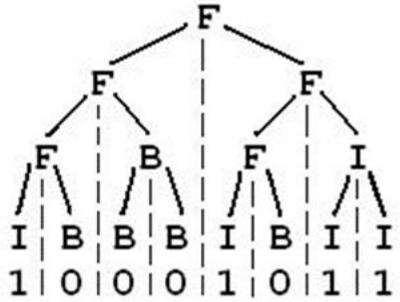
〖例〗现在给定中缀表达式，构建语义树 *D*

〖Example〗 (*a* + *b*) * (*c* * (*d* + *e*)) = *a b* + *c d e* + * *



6.3.2[3081] FBI树

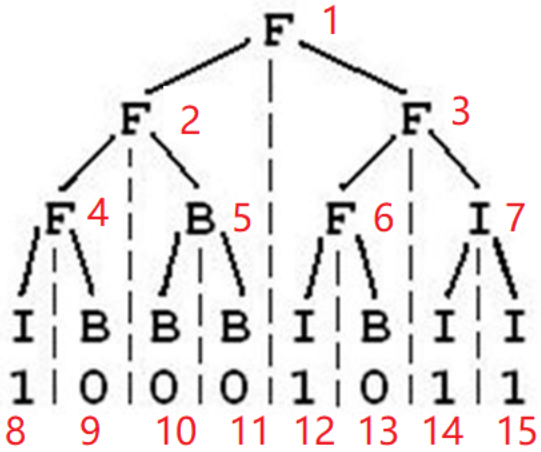
我们可以把由“0”和“1”组成的字符串分为三类：全“0”串称为B串，全“1”串称为I串，既含“0”又含“1”的串则称为F串。
FBI树是一种二叉树（如下图），它的结点类型也包括F结点，B结点和I结点三种。
由一个长度为2N的“01”串S可以构造出一棵FBI树T，递归的构造方法如下：
1) T的根结点为R，其类型与串S的类型相同；
2) 若串S的长度大于1，将串S从中间分开，分为等长的左右子串S1和S2；由左子串S1构造R的左子树T1，由右子串S2构造R的右子树T2。
现在给定一个长度为2N的“01”串，请用上述构造方法构造出一棵FBI树，并输出它的后序遍历序列。

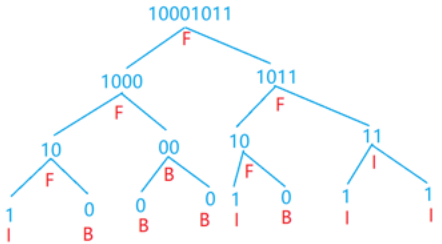


输入每组输入数据的第一行是一个整数N（0<=N<=10），第二行是一个长度为2N的“01”串。

输出
每组输出包括一行，这一行只包含一个字符串，即FBI树的后序遍历序列。
样例输入
3

6.3.2[3081] FBI树-方法1（常规）



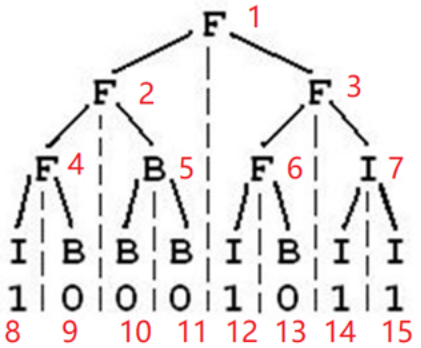


这是一颗完全二叉树，根据题目的条件，可以
给每个点进行编号，并根据最下面一层的初始条件，推导出父节点的值。

```
26      string s;
27      cin>>n>>s;
28  //  len=n<<1;
29      for(int i=1;i<=n;i++)len=2*len;
30      for(int i=len;i<2*len;i++)
31          node[i]=s[i-len]-'0';
```

6.3.2[3081] FBI树-方法1-建树

```
6  void bulidTree(int root){
7      if(root>=len)return;//建树的退出条件，当前点的编号大于
8      bulidTree(2*root);// 遍历左子树
9      bulidTree(2*root+1);//遍历右子树
10     // 类似于后续遍历,根据左右孩子求解父节点
11     if(node[2*root]==0&&node[2*root+1]==0)node[root]=0;
12     else if(node[2*root]==1&&node[2*root+1]==1)node[root]=1;
13     else node[root]=2;
14 }
```



6.3.2[3081] FBI树-方法1

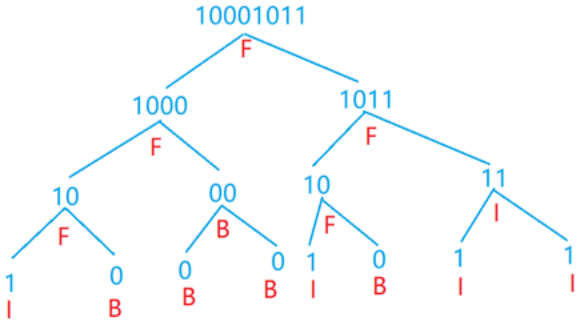
```
#include<bits/stdc++.h>
using namespace std;
int node[3000];
int len=1;

void bulidTree(int root){

void lastTree(int root){

int main(){
    int n;
    string s;
    cin>>n>>s;
//  len=n<<1;
    for(int i=1;i<=n;i++)len=2*len;
    for(int i=len;i<2*len;i++)
        node[i]=s[i-len]-'0';
    bulidTree(1);
    lastTree(1);
    return 0;
}
```

```
void bulidTree(int root){
    if(root>=len)return;
    bulidTree(2*root);// 遍历左子树
    bulidTree(2*root+1);//遍历右子树
    // 类似于后续遍历,根据左右孩子求解父节点
    if(node[2*root]==0&&node[2*root+1]==0)
        node[root]=0;
    else if(node[2*root]==1&&node[2*root+1]==1)
        node[root]=1;
    else node[root]=2;
}
void lastTree(int root){
    if(root>=2*len)return;
    lastTree(2*root);
    lastTree(2*root+1);
    if(node[root]==1)cout<<"I";
    else if(node[root]==0)cout<<"B";
    else cout<<"F";
}
```

6.3.2[3081] FBI树-方法2（专业）

有题意可知，树的递归构造方法如下：
1) T的根结点为R，其类型与串S的类型相同；
2) 若串S的长度大于1，将串S从中间分开，分为等长的左右子串S1和S2；由左子串S1构造R的左子树T1，由右子串S2构造R的右子树T2。

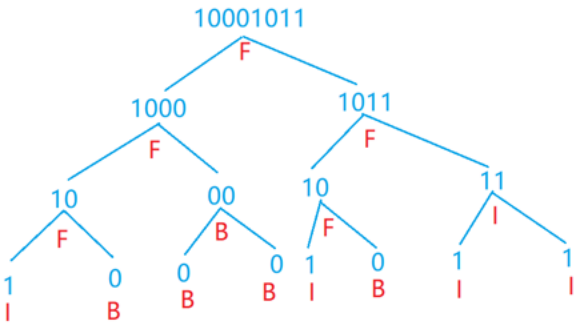
对于样例来说，我们可以得到如图所示的满二叉树。



我们发现当输入字符串固定时，整棵树也就固定了，因此可以将字符串当做递归函数传入的参数。由于要输出后序遍历，因此需要先输出左子树和右子树，再输出当前节点的信息。从图中可以发现，左子树所对应的字符串即是当前字符串的前半段，右子树所对应的字符串是后半段。因此依次递归处理字符串的前半段和后半段即可。最后需要判断当前节点的类型，这里可以统计一下当前字符串中0和1的包含情况。

时间复杂度分析

在每个递归函数内判断当前节点的类型时，均需要遍历一遍整个字符串，因此时间复杂度是线性的。从上图中可以发现，整棵树一共有 N+1 层，每层总共会遍历 2^N 的长度，因此总时间复杂度是 $O((N+1)*2^N)$。



6.3.2[3081] FBI树-方法2（专业）

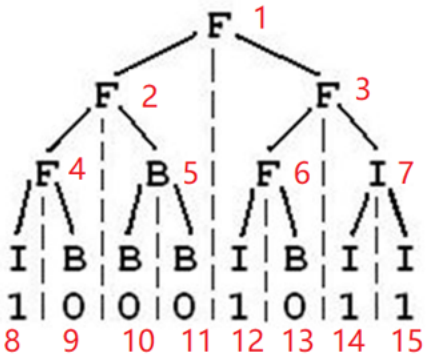
```cpp
#include <bits/stdc++.h>
using namespace std;
void dfs(string str)
{//如果左右子树的长度为1,就是叶子节点
    if (str.size() > 1)//如果不是叶子节点
    {
        dfs(str.substr(0, str.size() / 2));//遍历左子树
        dfs(str.substr(str.size() / 2));//遍历右子树
    }
    //以下代码是判断当前节点的值
    int one = 0, zero = 0;
    for (int i = 0; i < str.size(); i ++ )
        if (str[i] == '0') zero ++ ;
        else one ++ ;//求当前字符串0,1的个数

    if (one && zero) cout << 'F';//如果0,1都有
    else if (one) cout << 'I';
    else cout << 'B';
}
int main()
{
    int n;
    string str;
    cin >> n >> str;
    dfs(str);
    return 0;
}
```

```cpp
/*后续遍历
dfs(str){
    dfs(str的前一半);//遍历左子树
    dfs(str的后一半);//遍历右子树
    访问 根节点 ;
} */
```

6.3.2[3081] FBI树-方法2

首先因为后序遍历为左右根，与dfs的思路很像，所以我们只要把左子树先于右子树处理即可（根节点一定在子节点后）
那么我们只要先找到子树的叶子节点即可一步一步往上推导得出上一层的根节点，最终可以得到一颗完整的树
接下来是两种解法的不同点
dfs解法是通过递归回溯将每个点的状态求出然后得出解
而非dfs解法是通过这道题的性质，对于一个节点来说，如果他的叶子节点中不存在1那他一定是I节点，如果不存在0那他一定是B节点，如果两者都存在，那他一定是F节点

*/



6.3.2[3081] FBI树-写法3

```cpp
#include<bits/stdc++.h>
using namespace std;
string s;
int ll,n=0;
void fbi(int l,int r){
    int B=1,I=1;//初始化标记 //
    if(r>l){//如果不是叶子节点
    // r==l时是第一个叶子节点，即样例数据的第8个点
        fbi(l,(l+r)/2);//查找左子树
        fbi((l+r)/2+1,r);//查找右子树
    }
    for(int i=0;i<=r-l;i++){//遍历区间中0,1的情况，根据情况标记
        if(s[i+l]=='1') B=0;
        else if(s[i+l]=='0')I=0;
    }
    if(B!=0)    cout<<"B";//如果不存在1则为B串
    else if(I!=0)    cout<<"I";//如果不存在0则为I串
    else cout<<"F";//否则为F串
}
int main(){
    cin>>n>>s;
    fbi(0,(1<<n)-1);//遍历字符串
    return 0;
}
```

6.3.2[3081] FBI树-写法4

```cpp
#include<bits/stdc++.h>
using namespace std;
int m;
char str[3000];
char put[3] = {'B', 'I' ,'F'};//先存下FBI各自对应的情况，
//全“ 0” 串称为B串，全“ 1” 串称为I串，既含“ 0” 又含“ 1” 的串则称为F串。
int dfs(int sx,int ex)
{
    int a,b;
    if(sx==ex)//搜索到叶子节点
    {
        printf("%c", put[str[sx] - '0']);//输出该节点类型
        return str[sx] - '0';//返回上一节点，将本节点状态返回给上一个节点
    }
    a=dfs(sx,(sx+ex)/2);//搜索本节点的左子树
    b=dfs((ex+sx)/2+1,ex);//搜索本节点的右子树
    if(a == b)//同叶子节点一样
    {
        printf("%c", put[a]);
        return a;
    }
    else
    {
        cout<<'F';
       return 2;
    }
}
int main()
{
    scanf("%d",&m);
    scanf("%s",str);
    dfs(0,strlen(str)-1);//dfs遍历整个字符串
    cout<<endl;
    return 0;
}
```

6.3.3[2909] 二叉树遍历

树和二叉树基本上都有先序、中序、后序、按层遍历等遍历顺序，给定中序和其它一种遍历的序列就可以确定一棵二叉树的结构。
假定一棵二叉树一个结点用一个字符描述，现在给出中序和按层遍历的字符串，求该树的先序遍历字符串。
输入
两行，每行是由字母组成的字符串（一行的每个字符都是唯一的），分别表示二叉树的中序遍历和按层遍历的序列。

输出
一行，表示二叉树的先序序列。
样例输入
DBEAC
ABCDE
样例输出
ABDEC

```cpp
#include<bits/stdc++.h>
using namespace std;
string s1,s2;
void f(int l1,int r1,int l2,int r2)
{
    int i,j;
    for(i=l2;i<=r2;i++){
        int t=0;
        for(j=l1;j<=r1;j++){
            if(s2[i]==s1[j]){
                cout<<s1[j];
                t=1;//在中序查找
                break;
            }
        }
        if(t) break;

    }
    if(j>l1) f(l1,j-1,0,r2);
    if(j<r1) f(j+1,r1,0,r2);
}
int main()
{
    cin>>s1>>s2;//s1中序  s2层遍历
    f(0,s1.size()-1,0,s2.size()-1);
    return 0;
}
```

今天的课程结束啦……

下课了…
同学们再见！