vector

就很像动态数组

头文件：`#include<vector>`

①构造

```
vector<int> v1; //1.默认构造，无参构造

for (int i = 0; i < 10; ++i)
{
    v1.push_back(i);
}
PrintVector(v1);

//2.利用区间方式构造
vector<int> v2(v1.begin(), v1.end());
PrintVector(v2);

//3.n个element方式构造
vector<int> v3(10, 100);    //10个100
PrintVector(v3);

//4.拷贝构造
vector<int> v4(v3);
PrintVector(v4);
```

②赋值操作

赋值的话可以使用assign()函数，也可以使用其他方式

```
//直接赋值
vector<int> v2;
v2 = v1;

//assign赋值
vector<int> v3;
v3.assign(v1.begin(), v1.end());

//n个element赋值
vector<int> v4;
v4.assign(10, 100);
```

③插入和删除

插入主要是使用push_back()，也可使用insert()；删除操作主要是pop_back()，也可使用erase()

```
vector<int> v;
//尾插
```

```
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    PrintVector(v);

    //尾删
    v.pop_back();
    PrintVector(v);

    //插入 - 提供迭代器
    v.insert(v.begin(), 100);
    PrintVector(v);

    //重载
    v.insert(v.begin(), 2, 100);
    PrintVector(v);

    //删除 - 提供迭代器
    v.erase(v.begin());
    PrintVector(v);

    //重载
    v.erase(v.begin(), v.end());        //相当于清空操作
    PrintVector(v);

    v.clear();      //清空容器中所有元素
    PrintVector(v);
```

④容量和大小

对于容量用的是capacity()，对于大小是size()，当然你也可以用resize()来改变其大小，不够在此之前都需用empty()这个函数来判断一下容器是否为空；

```
    vector<int> v;

    for (int i = 0; i < 10; ++i)
    {
        v.push_back(i);
    }
    PrintVector(v);

    if (v.empty())
    {
        cout << "vector容器为空" << endl;
    }
    else
    {
        cout << "vector容器不为空" << endl;
        cout << "vector容器的容量为: " << v.capacity() << endl;
        cout << "vector容器的大小为: " << v.size()<< endl;
    }
```

```
    //重新指定大小 - 变大
//  v.resize(15);
    v.resize(15,10);        //重载
    PrintVector(v);

    //重新指定大小 - 变小
    v.resize(5);
    PrintVector(v);     //超过部分将会删除
```

除此之外还有很多函数，例如at()返回元素，front()返回首元素，back()返回尾元素，clear()清空容器等等

链表

单向静态链表

用`Struct node`定义链表节点上的元素：id表示编号，一般用不上，可以用存储这个节点的nodes[i]的i来表示节点编号；data表示节点存储的数据，nextid表示next指针，指向下一个节点的编号。

```
struct Node{
    int id;
    int data;
    int nextid;
}nodes[1000];
    nodes[0].nextid = 1;
    for(int i = 1; i <= n; i++){
        nodes[i].id = i;
        nodes[i].nextid = i + 1;
    }
    nodes[n].nextid = 1;//循环链表
    //删除节点
    nodes[pre].nextid = nodes[now].nextid;
    now = ndoes[pre].nextid;
```

还有一种直接用指针来实现的双向链表

```
struct Node{
    int data;
    Node* pre;
    Node* next;
    Node(int d) : data(t), pre(nullptr), next(nullptr);
}
```

STL List

```
#include<list>
using namespace std;
list<int> node;
//为链表赋值
for(int i = 1; i <= n; i++){
    node.push_back(i);
}
```

```
//遍历链表，使用it迭代器遍历链表
list<int>::iterator it = node.begin();
while(it!=node.end()){//node.end()是尾迭代器，在最后一个节点之后一个
    cout << *it << " ";//it就相当于一个指针
    it++;
}
//如果要获取下一个元素的迭代器，可以(it1 = it)++;
//也可以
for(auto x : node) cout << x << " ";
```

函数	说明
front()	返回第一个元素的引用（左值）
back()	返回最后一个元素的引用（左值）
push_front(g)	把g加到链表首
push_back(g)	把g加到链表尾
pop_front()	删除链表首
pop_back()	删除链表尾
empty()	返回1(empty)或0(not empty)
size()	返回当前容器实际包含的元素个数。
iterator insert(pos,elem)	在迭代器 pos 指定的位置之前插入一个新元素 elem，并返回表示新插入元素位置的迭代器。
iterator insert(pos,n,elem)	在迭代器 pos 指定的位置之前插入 n 个元素 elem，并返回表示第一个新插入元素位置的迭代器。
iterator insert(pos,first,last)	在迭代器 pos 指定的位置之前，插入其他容器（例如 array、vector、deque 等）中位于 [first,last) 区域的所有元素，并返回表示第一个新插入元素位置的迭代器。
iterator insert(pos,initlist)	在迭代器 pos 指定的位置之前，插入初始化列表（用大括号 { } 括起来的多个元素，中间有逗号隔开）中所有的元素，并返回表示第一个新插入元素位置的迭代器。
erase()	该成员函数既可以删除 list 容器中指定位置处的元素，也可以删除容器中某个区域内的多个元素。
iterator erase (iterator position);	
iterator erase (iterator first, iterator last);	
clear()	删除 list 容器存储的所有元素。
remove(val)	删除容器中所有等于 val 的元素。
unique()	删除容器中相邻的重复元素，只保留一份。

函数	说明
swap()	交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。
sort()	通过更改容器中元素的位置，将它们进行排序。
reverse()	反转容器中元素的顺序。

队列（FIFO）

队列就是先进先出（FIFO），元素只能从队首离开队列，从队尾进入队列。
队列有两种实现方式：链队列和循环队列。

手写循环队列

循环队列，对于一个大小为n的数组，只能存(n-1)个队列元素，队列相当于`[head,rear)`左开右闭，`rear`指向的是最后一个元素的后一个位置。
那么，`size = (rear + n - head) % n`，如果`size`等于0，则队列为空。

```
q.push(x);      //入队，将元素 x 从队尾插入（尾插法）
q.pop();        //出队，删除对首元素，并返回其值
q.size();       //返回队中元素个数
q.front();      //返回对首元素
q.back();       //返回队尾元素
q.empty();      //判断是否为空（空返回 1，非空返回 0）
```

实现代码：

```
struct queue{
    int data[MAXQSIZE];
    int head = 0;
    int rear = 0;
    bool push(int x){
        if((rear+1)%MAXQSIZE==head)return false;
        data[rear] = x;
        rear = (rear+1)%MAXQSIZE;
        return true;
    }
    int pop(){
        int t = data[head];
        head = (head+1)%MAXQSIZE;
        return t;
    }
    int size(){
        return (rear+MAXQSIZE-head)%MAXQSIZE;
    }
    bool empty(){
        if(size()==0)return true;
        else return false;
    }
    int front(){
        return data[head];
```

```
    }
    int back(){
        return data[(rear+MAXQSIZE-1)%MAXQSIZE];
    }
};
```

stl queue

头文件：`#include<queue>`

声明：`queue<int> q;`

基本操作：

```
q.push(x);      //入队，将元素 x 从队尾插入（尾插法）
q.pop();        //出队，删除对首元素，并返回其值
q.size();       //返回队中元素个数
q.front();      //返回对首元素
q.back();       //返回队尾元素
q.empty();      //判断是否为空（空返回 1，非空返回 0）
```

stl优先队列（priority_queue）

优先队列是最优元素永远位于队首，用堆来实现的，每次插入操作的复杂度是O(logn)。
头文件`#include<queue>`

创建priority_queue对象：

模板：`priority_queue< 数据类型，容器类型，优先规则> pq;`

数据类型：可以是`int`、`double`等基本类型，也可以是自定义的结构体。

容器类型：一般为`deque`（双端列表）、`vector`（向量容器），可省略，省略时以`vector`为默认容器。

pq：优先队列名。

声明代码如下：

默认声明：

```
priority_queue<int> pq;
```

手动声明：

```
priority_queue<int,vector<int>,less<int> > pq;   //以less为排列规则（默认，大顶堆，表示顶堆元素比其他都大）
priority_queue<int,vector<int>,greater<int> > pq; //以greater为排列规则（小顶堆，表示顶堆元素比其他都小）
```

自定义排序：

```
struct lei
{
    int n;
    double m;
}a,b;
struct mop
{
    bool operator ()(lei a,lei b)
    {
        return a.n>b.n;//相当于greater(小顶堆)
    }
};
```

引用方式：

```
priority_queue<lei,vector<lei>,mop> pq;
```

栈（LIFO）

栈是后进先出，它的基本操作为：

- empty()：返回栈是否为空
- size()：查询栈的长度
- top()：查看栈顶元素
- push()：向栈顶添加元素
- pop()：删除栈顶元素

手写栈较为简单，这里就不给出代码了

stl stack

头文件：`#include<stack>`

二叉树

定义

二叉树的第1层是一个节点，称为根，它最多由两个子节点，分别是左子节点和右子节点，以它们为根的子树称为左子树和右子树。

每个节点不必都有左右子节点，可以只有一个子节点或没有子节点，没有子节点的节点被称为叶子节点。

二叉树节点的编号是从上到下，从左到右的。

显然，第1层有2^0个节点，第1层有2^1个节点，则第n层有2^{n-1}个节点。

根据等比数列求和公式，前n层有2^n-1个节点。

（1）满二叉树：每一层的节点数都是满的。

（2）完全二叉树：如果满二叉树只在最后一层缺失节点，且缺失的节点编号都在最后，则称它为完全二叉树

满二叉树和完全二叉树都是平衡二叉树，平衡二叉树是每个节点的左右子树的数量都差不多。

完全二叉树的性质

一颗节点总数量为k的完全二叉树，设1号节点为根节点。

1. $i > 1$的节点，其父节点是$i/2$

证明如下：设节点i在第m层，则$i = 2^{m-1} - 1 + 2j + (1,2)$，
$i/2 = 2^{m-2} - 1 + j + [(1,2) + 1]/2 = 2^{m-2} - 1 + j + 1$，证毕

2. 如果节点i有子节点，那么它的左子节点是$2 \times i$，右子节点是$2 \times i + 1$。

证明如下：设节点i在第m层，则$i = 2^{m-1} - 1 + j$，则左子节点为$2^m - 1 + 2j - 1 = 2 \times i$，右子节点在左子节点右边一个，证毕。

3. 如果$2 \times i > k$，那么节点i就没有子节点；如果$2 \times i + 1 > k$，那么节点i没有右子节点。

证明可以由2得出。

一般来说，如果用数组`tree[N]`来储存由m个元素的二叉树，则$N = 4m$

二叉树的遍历

宽度优先搜索（BFS）

按层遍历，可以用队列，先把根节点入队，然后根节点出队，把根节点的子节点入队，再出队一个元素m，把元素m的子节点入队，以此类推。

深度优先搜索（DFS）

分为先序遍历，中序遍历和后序遍历三种顺序，分别就是（父节点、左子节点、右子节点），（左子节点、父节点、右子节点），（左子节点、右子节点、父节点）。

用递归最方便，下面给出先序遍历的伪代码。

```
void preorder(node* root){
    cout << root->value;
    preorder(root->lson);
    preorder(root->rson);
}
```

只要知道中序遍历和任意另一种遍历的结果，就能把这颗二叉树构造出来。

例题

```
/*
把由01组成的字符串分为3类，全0串称为B串，全1串称为I串，既含0又含1的串称为F串。FBI树是一种二叉树，它的节点类型包括F节点，B节点和I节点3种。用长度为2^N的01串可以构造出一颗FBI树T，其递归构造方法如下：
1.T的根节点为R，其类型与串S的类型相同。
2.若串S的长度大于1，则二分为等长的左右子串S1和S2，用左子串S1构造R的左子树T1，用右子串S2构造R的右子树T2。
现给定一长度为2^N的01串，请用上述构造方法构造出一颗FBI树，并输出它的后序遍历序列。
*/
#include<iostream>
#include<string>
using namespace std;
```

```
int N;
string s;
enum NodeType{F,B,I};
struct Node{
    NodeType type;
    Node* left;
    Node* right;
    Node(NodeType t) : type(t), left(nullptr), right(nullptr){}
};
//构造FBI树，返回父节点
Node* buildTree(int left, int right){
    if(left == right) return new Node(s[left]=='0'?B:I);
    int mid = (left+right) / 2;
    Node* root = new Node(F);
    root->left = buildTree(left,mid);
    root->right = buildTree(mid+1,right);
    if(root->left->type == root->right->type) root->type = root->left->type;
    return root;
}
//后序遍历
void postOrder(Node *node){
    if(node->left!=nullptr)postOrder(node->left);
    if(node->right!=nullptr)postOrder(node->right);
    switch (node->type)
    {
    case F:
        cout<<'F';
        break;
    case B:
        cout<<'B';
        break;
    case I:
        cout<<'I';
        break;
    default:
        break;
    }

}
int main(){
    cin >> N;
    cin >> s;
    Node* root = buildTree(0,s.length()-1);
    postOrder(root);
}
```