# C、C++基础&STL的使用

## C

#include<stdio.h>

**输入:**

int n;

**scanf(“%d”,&n);//输入整数n**

char c[10]

scanf(“%s”,c);//输入字符串c（不用取地址符）

**输出：**

**printf(“%d”,n);//输出整数n**

printf(“%s”,c);//输出字符串c

**printf(“%04d”,n);//输出整数n，如果不足4位则用0补齐（一些输出id的题上会用到）**

**printf(“%.2f”,f);//输出浮点数f，保留两位小数**

占位符：

%a,%A 读入一个浮点值(仅C99有效)

%c 读入一个字符

%d 读入十进制整数

%i 读入十进制，八进制，十六进制整数

%o 读入八进制整数

%x,%X 读入十六进制整数

%s 读入一个字符串，遇空格、制表符或换行符结束。

%f,%F,%e,%E,%g,%G 用来输入实数，可以用小数形式或指数形式输入。

%p 读入一个指针

%u 读入一个无符号十进制整数

%n 至此已读入值的等价字符数

%[] 扫描字符集合

%% 读%符号

**#include<cstring>**

**memset(n,0,sizeof(n));//将数组n中的sizeof(n)字节的数据赋值为0**

## C++

**#include<iostream>**

**using namespace std;**

**int main()**

**{**

**}**

**输入：**

**cin>>n;**

**cin.clear();//这句不知道具体啥用**

**cin.sync();//这两句清空输入缓冲区，否则getline不能正确输入，其实不用clear()光这句也可以的**

**string s;**

**getline(cin,s);//输入整行作为一个string**

**输出：**

**cout<<n<<M<<endl;**

字符串流

#include<sstream>

string s;

getline(cin,s);

stringstream ss(s);

int a;

ss>>a;//字符串流向外输出

ss.str(“”);//清除或重载

## 字符串string

#include<string>//可省

string可直接用运算符比较，默认字典序

string s = “”;

**s.length();//返回s的长度（字符数）**

**s[i];//返回第i个字符char**

**string a = s.substr(0,10);//获取s从0开始长度为10的字串（如果s长度不足10，则截至末尾）**

## 向量vector

**#include<vector>**

**vector<int> a(10);//初始化向量大小为10**

**vector<int> a(10,1);//初始化10个1在向量中**

**a[i];//访问a中第i个元素**

a.back();//返回最后一个元素

a.front();//返回第一个元素

**a.clear();//清空a**

a.empty();//判断a是否为空，空返回true

**a.push\_back(n);//在a的末尾插入n**

a.pop\_back();//删除a的最后一个元素

**a.size();//返回a中的元素个数**

a.capacity();//返回a在内存中总共可以容纳的个数

**a.erase(a.begin()+m);//删除a中第m个元素（从0算起）**

a.erase(a.begin()+m,a.begin()+n);//删除a中从m开始到n的元素（不包括n）

a.insert(a.begin()+1,5);//在a的位置1插入5

a.insert(a.begin()+1,3,5);//在a的位置1插入3个5

**a.resize(10);//将a的现有元素个数改为10个，多删少补，值随机**

**a.resize(10,2);//将a的现有元素改为10个2**

迭代器

for(vector<int>::iterator it=a.begin();it!=a.end();it++)

{

cout<\*it<<endl;

}

**#include<algorithm>**

**sort(a.begin(),a.end());//默认从小到大排**

sort(a.begin(),a.end(),greater<int>());//从大到小排序

**sort(a.begin(),a.end(),cmp);//cmp是自己写的比较函数**

struct node

{

int n;

};

bool cmp(node n1, node n2)

{

return n1.n < n2.n;

}

//当排序函数调用这个函数作为比较函数时，返回true则n1排在n2前面，返回false则n2排在n1前面

## 队列queue

**#include<queue>**

**queue<int> q;**

**q.push(x);//将x入队，放置队列末尾**

**q.pop();//从队首弹出一个元素，不会返回元素值**

**q.front();//访问队首元素值**

q.back();//访问队尾元素值

**q.size();//返回队列中元素个数**

**q.empty();//队列为空返回true**

优先队列：

priority\_queue<int> q;

priority\_queue<int, vector<int>, greater<int> > q;//参数：元素类型，容器类型（可省，默认vector），比较算子（可省，默认less，小的后出队，大的排在前）

q.top();//访问队首元素

q.pop();//队首元素出队

class T

{

public:

int x,y,z;

T(int a,int b,int c):x(a),y(b),z(c)

{

}

};

bool operator<(const T&t1,const T&t2)

{

return t1.z<t2.z;

}

priority\_queue<T> q;//该优先队列中z小的后出队

## 栈stack

**#include<stack>**

**stack<int> a;**

**a.size();//返回元素个数**

**a.top();//返回栈顶元素值**

**a.pop();//从栈顶删除元素**

**a.push(x);//x压入栈**

**a.empty();//栈为空返回true**

## 映射Map

**#include<map>**

map<string,string> mp;

**mp[“aa”] = “123”;//插入/赋值**

mp.insert(pair<string,string>(“aa”,”123”));//pair方式插入

//查找方式1：迭代器

**map<string,string>::iterator it = mp.begin();**

**it = mp.find(“aa”);**

if(it != mp.end())

cout<<(it->second)<<endl;

for(;it!=mp.end();it++)//遍历map

…

//查找方式2：直接通过键key值value查找

**map<int,int> mp;**

**if(mp[1] == 0)//当value类型为int时，默认初始为0，故可用0表示没有找到**

mp.empty();//map空返回true

mp.clear();//清空map

mp.erase(it);//删除迭代器所指的

mp.erase(mp.begin(),mp.end());//删除所有

## 集合set

**#include<set>//集合默认从小到大排序且元素不重复**

**set<int> s;**

**s.begin();//**

**s.end();//**

**s.clear();//清空**

**s.empty();//集合空返回true**

**s.size();//返回集合元素个数**

**s.erase(it);//删除迭代器指针it所指的元素**

**s.insert(a);//插入元素a**

**s.find(a);//返回查找到元素a的迭代器位置**

# 算法

## 二分查找

vector<int> v;//要先对v进行排序（下面这个假定为已经从小到大排序）

int binarySearch(int b,int e,int x)//开始位置b，结束位置e，查找x

{

int mid;

while(b<e)

{

mid = (b+e)/2;

if(x > v[mid])

{

b = mid + 1;

}

else if(x < v[mid])

{

e = mid - 1;

}

else

{

return mid;

}

}

return -1;//没找到

}

## DFS深搜

邻接表：

int d[100][100];//记录连通路，一般用邻接表比较好，数据大的话邻接矩阵会超时超内存

int vis[100];//记录是否访问

void dfs(int p)//当前位置p

{

vis[p] = 1;

for(int i=0;i<100;i++)

{

if(d[p][i] != 0 && vis[i] == 0)//连通并且未访问

{

dfs(i);

}

}

}

邻接矩阵：

int vis[100];//记录是否访问

struct Conn

{

vector<int> v;//储存邻接点

};

Conn conn[100];

void dfs(int p)//当前位置p

{

vis[p] = 1;

for(int i=0;i<conn[p].v.size();i++)

{

int nextp = conn[p].v[i];

if(vis[nextp] == 0)//连通并且未访问

{

dfs(nextp);

}

}

}

## BFS广搜

使用上面的邻接表：

void bfs(int p)//从p点开始广搜

{

queue<int> q;//广搜一般用队列实现

q.push(p);

while(!q.empty())

{

p = q.front();

q.pop();

vis[p] = 1;

for(int i=0;i<conn[p].v.size();i++)

{

int nextp = conn[p].v[i];

if(vis[nextp] == 0)

{

q.push(nextp);

}

}

}

}

## Dijkstra最短路径

int d[100][100];//记录路径长度，一般用邻接表比较好，数据大的话邻接矩阵会超时超内存

int closed[100];//close集，标记某个点是否已经找到最短路径

int pre[100];//记录路径

int dis[100];//记录最短路径

void dijkstra(int s)//开始位置s

{

bool flag = true;

dis[s] = 0;

while(flag)

{

flag = false;

close[s] = 1;

for(int i=0;i<100;i++)//更新

{

if(dis[i] > dis[s] + d[s][i])//dis需要初始化为很大的数

{

pre[i] = s;//记录更新路径，i的前一个点更新为s

dis[i] = dis[s] + d[s][i];

}

}

int minLen = 0x3f3f3f3f;

int minPos = -1;

for(int i=0;i<100;i++)

{

if(close[i] == 0 && minLen > dis[i])//找到未进入close集且距离最小的

{

flag = true;

minLen = dis[i];

minPos = i;

}

}

s = minPos;

}

}

## 二叉树

struct node

{

int value;

node\* left;

node\* right;

};

**node\* root = NULL;//定义根节点，初始一定要置空，C++初始分配指针时如果不指定，则会随机分配一个地址而非null**

### 前序遍历：

void preOrder(node\* p)

{

cout<<p->value<<endl;

if(p->left!=NULL)

{

preOrder(p->left);

}

if(p->right!=NULL)

{

preOrder(p->right);

}

}

### 中序遍历：

void inOrder(node\* p)

{

if(p->left!=NULL)

{

inOrder(p->left);

}

cout<<p->value<<endl;

if(p->right!=NULL)

{

inOrder(p->right);

}

}

### 后序遍历：

void postOrder(node\* p)

{

if(p->left!=NULL)

{

postOrder(p->left);

}

if(p->right!=NULL)

{

postOrder(p->right);

}

cout<<p->value<<endl;

}

### 层序遍历：

void levelOrder(node\* p)//层序即广搜

{

queue<node\*> q;

q.push(p);

while(!q.empty())

{

p = q.front();

q.pop();

cout<<p->value<<endl;

if(p->left!=NULL)

{

q.push(p->left);

}

if(p->right!=NULL)

{

q.push(p->right);

}

}

}

### 二叉树的构建：

已知中序和其他三种中的任意一种顺序可以唯一确定一棵二叉树

**已知中序和前序：**

int pre[100];//前序序列

int in[100];//中序序列

node\* createTree(int b, int e, int cur, node\* p)

{

//从中序序列中找到和当前前序序列相等的

int i = b;

for(;i<=e;i++)

{

if(in[i] == pre[cur])

{

break;

}

}

//以找到的点为界，将中序化为两段，分别表示该点的左右子树

p = new node();

p->value = pre[cur];

p->left = NULL;

p->right = NULL;

if(i < e)//右子树存在

{

p->right = createTree(i+1,e,cur+i-b+1,p->right);

}

if(i > b)//左子树存在

{

p->left = createTree(b,i-1,cur+1,p->left);

}

return p;

}

调用如下：

int main()

{

pre[0] = 1;pre[1] = 2;pre[2] = 3;pre[3] = 4;pre[4] = 5;

in[0] = 2;in[1] = 1;in[2] = 4;in[3] = 5;in[4] = 3;

root = createTree(0,4,0,root);

postOrder(root);

}

可正确输出后序。

**已知中序和后序：**

int pre[100];//前序序列

int in[100];//中序序列

int post[100];//后序序列

node\* createTree2(int b,int e,int cur,node\* p)

{

//从中序序列中找到和当前后序序列相同的点

int i = b;

for(;i<=e;i++)

{

if(in[i] == post[cur])

{

break;

}

}

//以找到的点为界，将中序化为两段，分别表示该点的左右子树

p = new node();

p->value = post[cur];

p->left = NULL;

p->right = NULL;

if(i > b)//左子树存在

{

p->left = createTree2(b,i-1,cur-e+i-1,p->left);

}

if(i < e)//右子树存在

{

p->right = createTree2(i+1,e,cur-1,p->right);

}

return p;

}

调用测试：

int main()

{

//pre[0] = 1;pre[1] = 2;pre[2] = 3;pre[3] = 4;pre[4] = 5;

in[0] = 2;in[1] = 1;in[2] = 4;in[3] = 5;in[4] = 3;

post[0] = 2;post[1] = 5;post[2] = 4;post[3] = 3;post[4] = 1;

root = createTree2(0,4,4,root);

postOrder(root);

}

已知中序和层序：

int lvl[100];//层序排列

int cur = 0;//层序的游标不需要回溯，用队列也可以实现

node\* createTree3(int b,int e,int c,node\* p)

{

//从中序序列中找到和当前层序序列相同的点

int i = b;

for(;i<=e;i++)

{

if(in[i] == lvl[c])

{

break;

}

}

//以找到的点为界，将中序化为两段，分别表示该点的左右子树

p = new node();

p->value = lvl[c];

p->left = NULL;

p->right = NULL;

if(i>b && i<e)//左右子树都存在

{

int c = cur;

cur += 2;

p->left = createTree3(b,i-1,c+1,p->left);

p->right = createTree3(i+1,e,c+2,p->right);

}

else if(i>b && i>=e)//左子树存在，右子树不存在

{

cur++;

int c = cur;

p->left = createTree3(b,i-1,c,p->left);

}

else if(i<=b && i<e)//仅右子树存在

{

cur++;

int c = cur;

p->right = createTree3(i+1,e,c,p->right);

}

return p;

}

已知前序和后序什么时候可以确定二叉树：

PAT 1119 Pre- and Post-order Traversals（30 分）

题目大意为已知前序和后序，是否能确定唯一一棵二叉树，若能，则输出Yes并给出中序排列；若不能，则输出No并给出任意一种中序排列。

从理论上讲：如果一个节点只有一个孩子，那么由前序和后序不能唯一确定这颗树，因为不知道这个孩子是左子还是右子

#include<iostream>

using namespace std;

int pre[35],pos[35],ino[35];

bool unique = 1;

int cnt = 0;

void build(int prel,int prer,int posl,int posr)

{

if(prel>prer)

{

return;

}

if(prel==prer)

{

ino[cnt++] = pre[prel];

return;

}

int e=pre[prel+1],prex=prel+1,posx=posl;//e为前序的下一个节点

while(prex<prer&&e!=pos[posx])//找到后序中值为e的节点，同时划分左右子树

{

prex++;

posx++;

}

if(posx == posr-1)//如果找到的节点刚好为当前节点的上一个节点，则表示当前节点只有一个孩子，树不唯一

{

unique = 0;

}

build(prel+1,prex,posl,posx);

ino[cnt++] = pre[prel];

build(prex+1,prer,posx+1,posr-1);

}

int main()

{

int n;

cin>>n;

for(int i=0;i<n;i++)

{

cin>>pre[i];

}

for(int i=0;i<n;i++)

{

cin>>pos[i];

}

build(0,n-1,0,n-1);

cout<<(unique? "Yes":"No")<<endl;

for(int i=0;i<n;i++)

{

cout<<(i==0? "":" ")<<ino[i];

}

cout<<endl;

}

### BST：

BST为二叉搜索树，遍历方式同上，通常左节点小于父节点，右节点大于等于父节点；但也有相反的，记得看清题目定义！

通常在已知前序和这棵树为BST的情况下可以确定这棵树，或者给定一个数列的情况下构建BST，构建函数如下：

node\* createBST(node\* p,int key)

{

if(p==NULL)

{

p = new node();

p->value = key;

p->left = NULL;

p->right = NULL;

return p;

}

if(key < p->value)

{

p->left = createBST(p->left,key);

}

else

{

p->right = createBST(p->right,key);

}

return p;

}

注：已知中序不能构建BST，因为BST中序一定是从小到大（从大到小）排列的，这样构建的树和链表一样。已知后序也不能构建BST。

### AVL：

AVL是自平衡树，对于任何节点其左右子树的高度差不超过1，通常和BST一起，构建自平衡二叉搜索树。

struct node

{

int data;

int h;

node\* left = NULL;

node\* right = NULL;

};

int max(int a,int b)

{

if(a > b)

return a;

else

return b;

}

int getHeight(node \*p)

{

if(p==NULL)

{

return -1;//空为-1，叶子节点才是0

}

else

{

return p->h;

}

}

bool isBalanced(node\* l,node\* r)

{

return abs(getHeight(l) - getHeight(r)) < 2;

}

node\* rightRotate(node \*p)

{

node\* k = NULL;

k = p->left;

p->left = k->right;

k->right = p;

p->h = max(getHeight(p->left), getHeight(p->right)) + 1;

k->h = max(getHeight(k->left), getHeight(k->right)) + 1;

return k;

};

node\* leftRotate(node \*p)

{

node\* k = NULL;

k = p->right;

p->right = k->left;

k->left = p;

p->h = max(getHeight(p->left), getHeight(p->right)) + 1;

k->h = max(getHeight(k->left), getHeight(k->right)) + 1;

return k;

};

node\* insertValue(node\* root,int d)

{

if(root == NULL)

{

root = new node();

root->data = d;

root->h = 0;

return root;

}

if(d < root->data)

{

root->left = insertValue(root->left,d);

if(!isBalanced(root->left, root->right))

{

if(d > root->left->data)//插在了右边，左右转

{

root->left = leftRotate(root->left);

root = rightRotate(root);

}

else//插在了左边，右单转

{

root = rightRotate(root);

}

}

root->h = max(getHeight(root->left),getHeight(root->right)) + 1;

}

else if(d > root->data)

{

root->right = insertValue(root->right,d);

if(!isBalanced(root->left, root->right))

{

if(d > root->right->data)//插在了右边，左单转

{

root = leftRotate(root);

}

else//插在了左边，右左转

{

root->right = rightRotate(root->right);

root = leftRotate(root);

}

}

root->h = max(getHeight(root->left),getHeight(root->right)) + 1;

}

return root;

}

## 并查集

并查集通常可以用数组来记录

int father[10000];

for(int i=0;i<10000;i++)

{

father[i] = i;

}

一开始的时候将每个节点的父亲都初始化为自己

用循环向上的方式可以查找到当前节点的根节点

int findroot(int x)

{

while(father[x] != x)

{

x = father[x];

}

return x;

}

用下面这个函数可以将两个节点统一至一个根节点

void unionroot(int a,int b)

{

int ra = findroot(a);

int rb = findroot(b);

if(ra < rb)//通常图的题目里不会规定根节点这种说法，这里可以采用一片连通区域的最小节点作为根节点

{

father[rb] = ra;

}

else

{

father[ra] = rb;

}

}

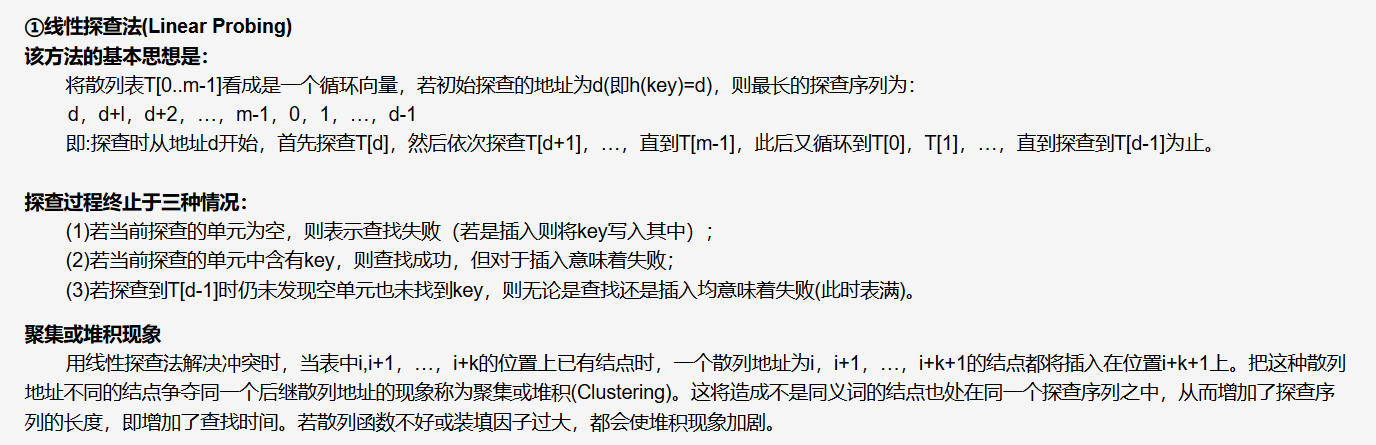
当然，记录下所有节点之间的连通关系后，也可以用对循环+DFS来判断有几个连通域。

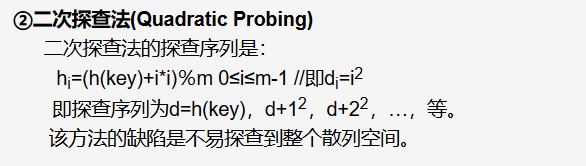
使用并查集的好处在于：不需要在所有数据输入后再进行连通域判断，每次输入一组连通节点的数据就可以进行连通计算，不需要存储下整幅图。

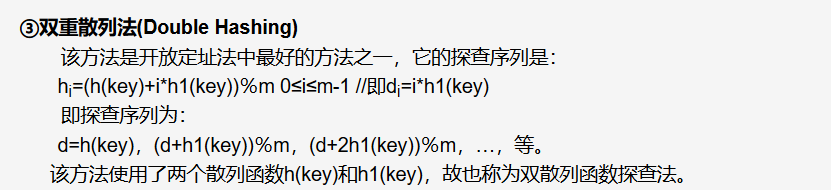
## 哈希散列

主要是hash表插入时的冲突解决方法：

h(key) = key % m;//key为要插入的值，m为hash表大小（通常要求为素数）



这个Pat 1078题上用到了，而且没有解释Quadratic Probing的意思，应该是默认当我们知道了233333333



其他还有好多方法23333

By the way，素数表可以这样建：

vector<int> notprime(20020,0);//素数表

void setNotPrime(int x)

{

int tmp = x;

tmp += x;

for(;tmp<20020;)

{

notprime[tmp] = 1;

tmp += x;

}

}

int main()

{

//构建素数表

notprime[1] = 1;

for(int i=2;i<20020;i++)

{

if(notprime[i] != 0)//如果i不是素数

{

continue;

}

//如果i是素数，则将i的倍数全都置为非素数

//cout<<i<<endl;

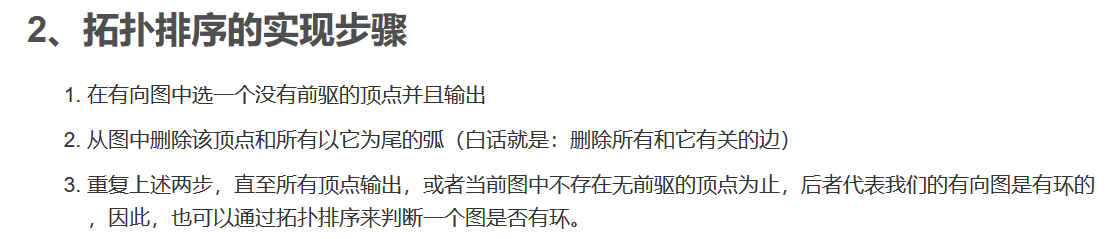
setNotPrime(i);

}

}

通常要求hash表的大小为素数，题目中会规定一个数值，要找到比这个数值大的最小素数作为hash表的m，用这种方法构建素数表比较快。

## 拓扑排序



## 树状数组

树状数组通常用来求一段数组的和

一个神奇的函数：得到一个数在二进制下从右往左出现的第一个1的值，利用了补码来实现这个函数

int lowbit(int t) {

return t&-t;

}

//求原数组从下标0到x的和（tree为树状数组）

int getsum(int x) {

int ans = 0;

while(x > 0) {

ans += tree[x];

x -= lowbit(x);

}

return ans;

}

//原数组中插入新的值，则需要将该位置后的所有树状数组都更新

void update(int x,int y) //x为位置，y为值

{

while(x <= MAX)

{

tree[x] += y;

x += lowbit(x);

}

}

## 贪心、动态规划

感觉这两个是随缘了，每道题贪心策略和动规dp数组都不一样，get不到某个点就真心想不到该怎么实现

自求多福了