STL

顺序式容器

vector

定义

vector是动态数组，定义方法一般有

功能	例子	说明
定义int型数组	vector a; vetcor a(100,6);	默认初始化，a为空 a有100个值为6的元素
定义string型数组	vector a(10,"null");	10个值为null的元素
定义结构型数组	struct point {int x,y;}; vector a;	a用来存坐标

还有二维数组的储存方法 vector a[MAXN];

常用操作

功能	例子	说明
赋值	a.push_back(100);	在尾部添加元素
尾部插入	a.insert(a.end() ,10, 5);	在尾部插入10个值为5的元素
删除尾部	a.pop_back();	删除末尾元素
删除具体元素	a.erase(a.begin()+2);	删除第三个元素
排序	sort(a.begin(), a.end());	从小到大排序

栈 stack

定义

stack s;

功能	例子	说明
赋值	s.push(item);	把item放到栈顶
返回	s.top();	返回栈顶元素
删除	s.pop();	删除栈顶元素且不返回

队列 queue

定义

queue q;

功能	例子	说明
返回	q.front();	返回队首元素
返回	q.back();	返回队尾元素
数量	q.size();	返回元素个数

列表 list

list是数据结构的双向列表，通过指针进行数据访问，它可以高效地在任意地方删除插入。

图论

树

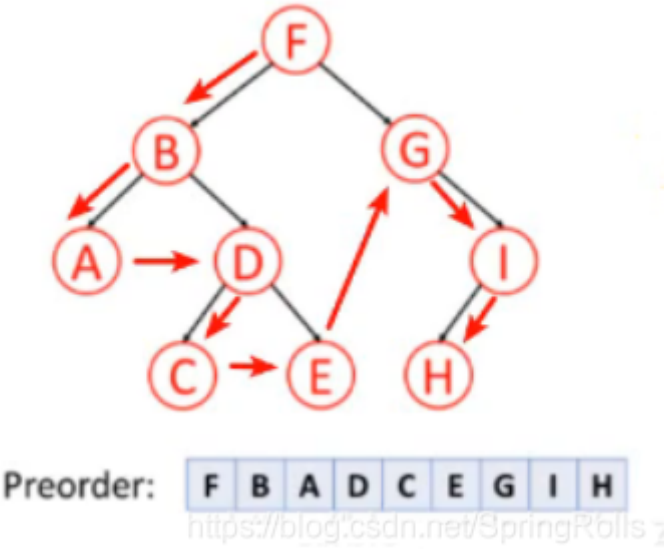
二叉树

```
#include<bits/stdc++.h>
using namespace std;
const int maxe = 1e5 + 5;
const int maxn = 1e4 + 5;

typedef struct node
{
    int date;
    struct node* left;
    struct node* right;
}BiTreeNode,*BiTree;
/*
BiTNode = struct BiTnode;
BiTree = struct BiTnode* 给指向结构体的指针换名字
*/
void CreatTree(BiTree* t)//先序构建 二重指针
{
    int i;
    cin >> i;
    if (i == -1)
    {
        *t = NULL;
    }
```

```
    else
    {
        *t = (BiTree)malloc(sizeof(BiTreeNode));
        if (!*t)//没有空间时报错
        {
            exit(-1);
        }
        (*t)->date = i;
        CreatTree(&(*t)->left);
        CreatTree(&(*t)->right);
    }
}

void PreOrderTraver(BiTree t)//先序遍历
{
    if (t == NULL)
        return;
    cout << t->date << " ";
    PreOrderTraver(t->left);
    PreOrderTraver(t->right);
}

void InOrderTraver(BiTree t)//中序遍历
{
    if (t == NULL)
        return;
    InOrderTraver(t->left);
    cout << t->date << " ";
    InOrderTraver(t->right);
}

void PostOrderTraver(BiTree t)//后序遍历
{
    if (t == NULL)
        return;
    PostOrderTraver(t->left);
    PostOrderTraver(t->right);
    cout << t->date << " ";
}

int main()
{
    BiTree t;
    CreatTree(&t);
    PreOrderTraver(t);
    cout << endl;
    InOrderTraver(t);
    cout << endl;
    PostOrderTraver(t);
}
```

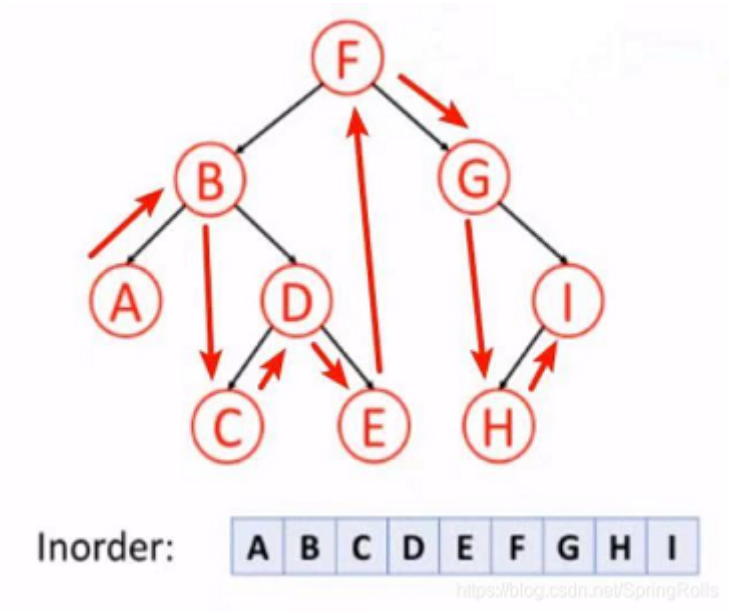
前序、中序、后序

前序



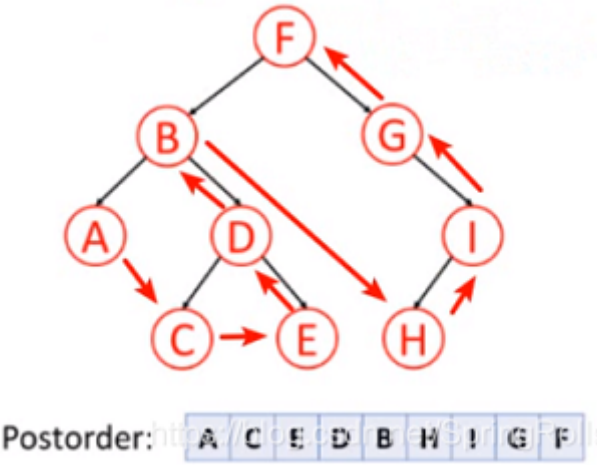

```
void preTrav(BiTree* root) {
  if (root) {
    cout << root->key << " ";
    preTrav(root->left);
    preTrav(root->right);
  }
}
```

中序




```
void midTrav(BiTree* root) {
  if (root) {
    midTrav(root->left);
    cout << root->key << " ";
    midTrav(root->right);
  }
}
```

后序




```
void lastTrav(BiTree* root) {
  if (root) {
    lastTrav(root->left);
    lastTrav(root->right);
    cout << root->key << " ";
  }
}
```

一、已知前序、中序遍历，求后序遍历

前序的每一位作为根节点，在中序中可以找到相对应的节点。在中序的节点对应的左右就是该节点的左右孩子节点。

直接上代码

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int date;
    struct Node *left;
    struct Node *right;
};
//这个就嗯背吧
Node *search(int *in, int *pre, int length) {
    if (length == 0) {
        return NULL;
    }
    Node *node = new Node;
    node->date = *pre;
    int index = 0;
    for (; index < length; index++) {
        if (in[index] == *pre) {
            break;
        }
    }
    //参数不要理解了属于是
    node->left = search(in, pre + 1, index);
    node->right = search(in + index + 1, pre + index + 1, length - (index + 1));
    return node;
```

```
}

void PreTra(Node *node) {
    if (node == NULL) {
        return;
    }
    printf("%d", node->date);
    PreTra(node->left);
    PreTra(node->right);
}


void Fanzhuan(Node *root) {
    if (root != NULL) {
        Fanzhuan(root->left);
        Fanzhuan(root->right);
        Node *temp;
        temp = root->right;
        root->right = root->left;
        root->left = temp;
    }

}

void bfs(Node *node) {
    queue<Node *> q;
    q.push(node);
    while (!q.empty()) {
        Node *node = q.front();
        cout << node->date << " ";
        if (node->left != NULL) {
            q.push(node->left);
        }
        if (node->right != NULL) {
            q.push(node->right);
        }

        q.pop();
    }

}


int main() {
    int in[1000];
    int pre[1000];
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        scanf("%d", &in[i]);
    }
    getchar();
    for (int i = 0; i < n; i++) {
        scanf("%d", &pre[i]);
    }

    Node *node = search(in, pre, n);
    Fanzhuan(node);
```

```
    bfs(node);
}
```

二、已知中序和后序遍历，求前序遍历

依然是上面的题，这次我们只给出中序和后序遍历：

中序遍历: ADEFGHMZ

后序遍历: AEFDHZMG

画树求法：
第一步，根据后序遍历的特点，我们知道后序遍历最后一个结点即为根结点，即根结点为G。

第二步，观察中序遍历ADEFGHMZ。其中root节点G左侧的ADEF必然是root的左子树，G右侧的HMZ必然是root的右子树。

第三步，观察左子树ADEF，左子树的中的根节点必然是大树的root的leftchild。在前序遍历中，大树的root的leftchild位于root之后，所以左子树的根节点为D。

第四步，同样的道理，root的右子树节点HMZ中的根节点也可以通过前序遍历求得。在前后序遍历中，一定是先把root和root的所有左子树节点遍历完之后才会遍历右子树，并且遍历的左子树的第一个节点就是左子树的根节点。同理，遍历的右子树的第一个节点就是右子树的根节点。

第五步，观察发现，上面的过程是递归的。先找到当前树的根节点，然后划分为左子树，右子树，然后进入左子树重复上面的过程，然后进入右子树重复上面的过程。最后就可以还原一棵树了。该步递归的过程可以简洁表达如下：

1 确定根,确定左子树，确定右子树。

2 在左子树中递归。

3 在右子树中递归。

4 打印当前根。

这样，我们就可以画出二叉树的形状，如上图所示，这里就不再赘述。

那么，前序遍历: GDAFEMHZ

编程求法：（并且验证我们的结果是否正确）

```
#include <iostream>
#include <fstream>
#include <string>

struct TreeNode
{
    struct TreeNode* left;
    struct TreeNode* right;
    char  elem;
};


TreeNode* BinaryTreeFromOrderings(char* inorder, char* aftorder, int length)
{
    if(length == 0)
    {
```

```
        return NULL;
    }
    TreeNode* node = new TreeNode;//Noice that [new] should be written out.
    node->elem = *(aftorder+length-1);
    std::cout<<node->elem<<std::endl;
    int rootIndex = 0;
    for(;rootIndex < length; rootIndex++)//a variation of the loop
    {
        if(inorder[rootIndex] ==  *(aftorder+length-1))
            break;
    }
    node->left = BinaryTreeFromOrderings(inorder, aftorder , rootIndex);
    node->right = BinaryTreeFromOrderings(inorder + rootIndex + 1, aftorder +
rootIndex , length - (rootIndex + 1));

    return node;
}

int main(int argc, char** argv)
{
    char* af="AEFDHZMG";
    char* in="ADEFGHMZ";
    BinaryTreeFromOrderings(in, af, 8);
    printf("\n");
    return 0;
}
```

输出结果：GDAFEMHZ

二叉树树的遍历

BFS

```
void bfs(Node *node) {
    queue<Node *> q;
    q.push(node);//将根节点压入队列中
    while (!q.empty()) {
        Node *node = q.front();//创建一个临时节点
        cout << node->date << " ";

        if (node->left != NULL) {
            q.push(node->left);
        }
        if (node->right != NULL) {
            q.push(node->right);
        }

        q.pop();
    }

}
```

DFS

```
void dfs(BiTree root)
{
    if(root==NULL)
        return;
    cout<<root->date<<" ";
    dfs(root->left);
    dfs(root->right);
}
```

树的直径

定义

树上最远两点（叶子结点）的距离

求法

从树上任意点u开始DFS(BFS)遍历图，得到距离u最远的结点v,然后从v点开始DFS遍历图，得到距离v最远的结点w， 则v、w之间的距离就是树的直径。（确定的结论）

```
//源点:起始点
//maxv：源点能到的最远点，maxdis:最远点对应的距离
int maxv, maxdis;

//u：dfs的源点，f: u点的父节点，d2s：u点到源点的距离
void dfs(int u, int f, int d2s) {
    if (maxdis < d2s) {
        maxdis = d2s;
        maxv = u;
    }
    for (int i = head[u]; ~i; i = e[i].next) {
        int v = e[i].to;//链接的下一个节点
        if (v == f) continue; //防止回到父节点
        dfs(v,u,d2s+1);
    }
}
/*
若加入边权
则在距离加边权w
dfs(v,u,d2s+w);
*/
int main() {
    int n;
    cin >> n;
    memset(head, -1, sizeof(head));
    for (int i = 0; i < n-1; i++) {
        int u, v;
        cin >> u >> v;
        add(u, v);
        add(v, u);
    }
    dfs(1,-1,0);//两次查找
    maxdis=0;
    dfs(maxv,-1,0);//此次一定是最远的
```

```
    cout<<maxdis<<endl;
    return 0;

}
```

树的重心

链式前向星

创建

```
#include <bits/stdc++.h>
using namespace std;
const int maxe = 1e5 + 5;
const int maxn = 1e4 + 5;

struct EDGE {
    int to;//指向的节点
    int next;//下一条边
    int w;//边权
} edge[maxe]; //边集数组
int head[maxn];//头结点数组
int tot;//计数器变量 指向最近没有利用的节点

void add(int u, int v, int w) { //表示u到v添加一条路径
    edge[tot].to = v;
    edge[tot].w = w;
    edge[tot].next = head[u];//指向前一条边
    head[u] = tot++;
}

int main() {
    memset(head, -1, sizeof(head));//head数组全部赋值为-1
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        add(u, v, w), add(v, u, w); //无向图 加两条边
    }
}
```

遍历

DFS

```cpp
bool vis[maxe];
void dfs(int u) { //深度优先
    cout << u << " ";
    vis[u] = true;
    for (int i = head[u]; i != -1; i = edge[i].next) {
        int to = edge[i].to;
        if (!vis[to]) { //阻止回头
            dfs(to);
        }
    }
}
```

BFS

```cpp
bool vis[maxe];//标记数组
void bfs(int u) { //队列实现广度优先
    queue<int> q;
    vis[u] = true;
    q.push(u);
    while (!q.empty()) {
        int k = q.front();
        q.pop();
        cout << k << " ";
        for (int i = head[k]; ~i; i = edge[i].next) {
            int to = edge[i].to;
            if (!vis[to]) {
                q.push(to);
                vis[to] = true;//因为不是递归实现，所以每次放入队列后都需要立即标记。
            }
        }
    }
}
```

搜索

DFS

记忆化搜索

存储已经出现的值，这里因为x很大且随机所以用map

```cpp
using namespace std;
typedef long long ll;

map<ll,ll> mp;

ll f(ll x){
    if(x==1) return 1;
    if(mp[x]) return mp[x];
    return mp[x]=f(x/2)+f(x/2+x%2);
}
```

```
int main() {
    ll x;
    cin>>x;
    cout<<f(x);
}
```

BFS

BFS探索全图

```
char room[23][23];
int dir[4][2] = {
    {-1, 0}, // 向左
    {0, -1}, // 向上
    {1, 0}, // 向右
    {0, 1} // 向下
};
int Mx, My;
int num; // 记录所有可去点
struct node {
    int x, y;
    int step = 0; // 记录步数
};
#define CHECK(x,y) (x<=Mx && x>=1 && y>=1 && y<=My)
void BFSCount(int dx,int dy){
    num = 1;
    queue<node> q;
    node start,next1;
    start.x=dx;
    start.y=dy;
    q.push(start); // 将起始位置入队
    while(!q.empty()){
        start=q.front(); // 不断重复入队过程
        q.pop();
        for(int i=0;i<4;i++){
            next1.x=start.x+dir[i][0];
            next1.y=start.y+dir[i][1];
            if((CHECK(next1.x, next1.y)&&room[next1.x][next1.y]=='.'){
                num++;
                room[next1.x][next1.y]='#';
                q.push(next1); // 下一个节点入队
            }
        }
    }
}
int main() {
    int x, y, dx, dy, endx, endy;
    while (cin >> Mx >> My) {
        for (y = 1; y <= My; y++) {
            for (x = 1; x <= Mx; x++) { //一次读入一行
                cin >> room[x][y];
                if (room[x][y] == '@') {
                    dx = x;
                    dy = y;
```

```
                }
            }
        }
        num = 0;
        BFSCount(dx, dy);
        cout << num << endl;
    }

}
```

全排列

next_permutation

使用函数next_permutation()

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int data[4]={5,3,1,4};
    sort(data,data+4);
    do{
        for(int i=0;i<4;i++){
            cout<<data[i]<<" ";
        }
    cout<<endl;
    }while(next_permutation(data,data+4));
    return 0;
}
```

next_permutation函数将按字母表顺序生成给定序列的**下一个较大**的排列，直到整个序列为降序为止。

prev_permutation函数与之相反，是生成给定序列的上一个较小的排列。

DFS

其方法是把第一个数和后面每一个数进行交换。

```
void dfsfun(int bin,int ed){
    // 如果是n个数字中取m个数的组合，则bin==m
    if(bin==ed){
        for(int i=0;i<10;i++){
            printf("%d ",dataarr[i]);
        }
        printf("\n");
    }
    else{
        for(int i=bin;i<=ed;i++){
            Swap(dataarr[bin],dataarr[i]);
            dfsfun(bin+1,ed);
            Swap(dataarr[bin],dataarr[i]);
        }
    }
}
```

子集

增量构造法

```
void print_subset(int n, int *A, int cur) {//cur是当前A数组的位置
    for(int i = 0; i < cur; ++i) {
        printf("%d ", A[i]);//如果有数据数组， 就写成data[A[i]]，此处不再赘述
    }
    printf("\n");


    int s = cur ? A[cur - 1] + 1 : 0;
    //当cur等于0的时候，这是第一次进入函数，所以选取的集合下标元素为0就可以了
    //cur不等于0的时候，即cur前面还有下标元素，为了得到全部的子集，所以这里不能漏掉，便从最小的那一个选择
    //那为什么A[cur-1]+1就是最小的下标呢，刚刚输出的最后一个下标就是A[cur-1]，所以这一个下标+1，即还未选择过的最小下标


    //然后就是从当前最小的还未选择过的下标作为下一个下标集合的首元素开始选取
    //直到n-1的下标位置，每一次选取首下标之后都开始递归
    for(int i = s; i < n; ++i) {
        A[cur] = i;
        print_subset(n, A, cur + 1);
    }
}
```

二分法

寻找一个数

```
int binarySearch(int target) {
    int left = 0;
    int right = v.size() - 1;// 从0开始计算的话是要减一的
    while (left <= right) { // 终止条件是left==right+1
        int mid = left + (right - left) / 2;
        if (v[mid] == target)
            return target;
        else if (v[mid] < target)
            left = mid + 1;
        else if (v[mid] > target)
            right = mid - 1;
    }
    return 0;
}
```

具体的详细写法详解二分查找算法 - murphy_gb - 博客园 (cnblogs.com)

寻找左侧边界的二分搜索

情景：比如说给你有序数组 nums = [1,2,2,2,3]，target = 2，此算法返回的索引是 2，没错。但是如果我想得到 target 的左侧边界，即索引 1，或者我想得到 target 的右侧边界，即索引 3。

一般是给定一个数K，寻找最小的下标i使得Ai大于等于K

```
int binarySearch(int target) {
    int left = 0;
    int right = v.size();// 左闭右开区间
    while (left < right) { // 左闭右开区间  while 的终止条件是 left == right
        int mid = left + (right - left) / 2;
        if (v[mid] == target)
            right= mid;
        else if (v[mid] < target)
            left = mid + 1;
        else if (v[mid] > target)
            right = mid; //注意
    }
    return left; //注意
}
```

寻找右侧边界的二分查找

```
int binarySearch(int target) {
    int left = 0;
    int right = v.size();// 左闭右开区间
    while (left < right) { // 左闭右开区间  while 的终止条件是 left == right
        int mid = left + (right - left) / 2;
        if (v[mid] == target)
            left= mid + 1;
        else if (v[mid] < target)
            left = mid + 1;
        else if (v[mid] > target)
            right = mid;
    }
    return left - 1;
}
```

当 nums[mid] == target 时，不要立即返回，**而是增大「搜索区间」的下界 left**，使得区间不断向右收缩，达到锁定右侧边界的目的。

Q：为什么返回的left-1？

由于

```
if (v[mid] == target)
    left= mid + 1;
```

因为我们对 left 的更新必须是 left = mid + 1，就是说 while 循环结束时，v[left] **一定不等于** target 了，而 v[left - 1]**可能是**target。

upper_bound&lower_bound

lower_bound(begin,end,num)：从数组的begin位置到end-1位置二分查找**第一个首个不小于(大于或等于)num**的数字，找到返回该数字的地址，不存在则返回end。

upper_bound(begin,end,num)：从数组的begin位置到end-1位置二分查找**第一个大于num**的数字，找到返回该数字的地址，不存在则返回end。

通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。

算法杂项

对于EOF的新写法**~scanf("%d",&a);**

位运算

if(x&1)... 等价于 if(x%2==0)... 判断x是否是奇数。

n&1 与运算　可以判断n是否为偶数　如果是**偶数**，n&1返回**0**；否则返回**1**，为**奇数**。

x>>=1 等价于 x/=2

x<<=1 等价于 x*=2

1<<x 等价于 2的x次方，常用于状压枚举、状压dp。

x&(-x) 为 x 的最低位的 '1' 对应的值，常用于树状数组。

l+r>>1 等价于 (l+r)/2，常用于二分。

差分

差分介绍：

一个数组，有多次操作，**每次对某个区间 [l,r]上每个数加上 x**，问最终的数组是什么样子。

我们不妨假设初始的数组全是0。当我们对 [l,r] 区间上每个数加上 x 以后，这个数组就变成了 0,0,...,0,x,x,...,x,0,0,0

如果我们只对两个位置的数进行操作：**a[l]+=x, a[r+1]-=x**，当我们做一次**前缀和**以后，我们发现，这个数组就变成了我们想要的样子。（前缀和是指，对于新数组 sum，sum[i]代表a数组中前i项之和。sum数组可以通过 sum[i]=sum[i-1]+a[i]得出）。

因此我们可以先对每个区间修改只修改两个数：**a[l]+=x,a[r+1]-=x**，在最后做一次前缀和

a[i]+=a[i-1]就可以了

排列组合

C(m,n)=n!/m!(n-m)!

A(m,n)=n!/(n-m)!

可以使用阶乘计算

```
int sum=0;
for(int i=1;i<=n;i++){
    sum*=i
}
```

快速幂

情景：a,b,c均为大数

$$a^b \% c$$

思路：

b不为0时不断循环。

当b为偶数时，b除以二，同时a乘上平方。
当b为奇数时，ans先乘以一个底数a，然后再进行同样的操作。

在运算的时候不断进行取模。

```
ll fast_power(ll a,ll b,ll c){
    ll ans=1;
    a%=c;
    while(b){
        if(b%2==1){
            ans=(ans*a)%c; // 答案先乘底数
        }
        a=(a*a)%c; // 底数平方 指数除二
        b/=2;
    }
    return ans;
}
```

加速(位运算)：

```
ll fast_power_wei(ll a,ll b,ll c){
    ll ans=1;
    a%=c;
    while(b){
        if(b&1){ //注意
            ans=(ans*a)%c;
        }
        a=(a*a)%c;
        b>>=1; //向右移动一位是除以2
    }
    return ans;
}
```