注意

重要算法

- Prim
- Kruskal
- Floyd
- TopologicalSort
- 堆排序
- 快速排序
- 插入排序
- 选择排序
- 二分查找
- huffmanTree

矩阵

- 矩阵旋转90
- 矩阵翻转180
- 矩阵水平镜像（见矩阵旋转90
- 矩阵转置（见矩阵旋转90
- 矩阵垂直镜像（见矩阵旋转180
- 矩阵乘法

链表

- 循环链表的基本操作 ☆
- 反转链表
- 相交链表
- 删除链表倒数第N个节点 ☆
- 链表中第K个节点
- 合并两个有序链表为递减链表
- 求有序链表(集合)A与B的并集 ☆
- 求有序链表(集合)A与B的交集
- 链表的直接插入排序

栈和队列

- 两个栈实现一个队列
- 包含min函数的栈
- 括号匹配
- 共享栈 ☆

线性表

- 删除数组中的最小元素
- 反转线性表
- 移除线性表中等于x的元素 ☆
- 删除有序线性表大于s小于t的元素 ☆
- 删除递增有序表的相同元素 ☆
- 将两个有序表合并为一个有序表
- 寻找有序表中x的位置，如果不存在则插入元素x，且仍然保持线性表有序i
- 循环左移
- 数组中重复的数字
- 有序递增数列查找两个和为s的数
- 将顺序表分成偶数和奇数两部分
- ###
- ###
- ###

字符串

- 只出现一次的字符

二叉树

- 平衡二叉树
- 镜像二叉树 ☆
- 二叉树的高度
- 求二叉树叶子节点的个数
- 中序先序后序递归遍历
- 层次遍历
- 查找二叉树中值为X的结点
- 以上题为基础并输出该结点的父亲节点
- 求先序/中序/后序历序列中第K个结点
- 二叉树中最小值的结点
- 统计所有结点数
- 统计值为X的结点个数
- 寻找值为X结点的所有祖先结点
- 寻找结点值为X的双亲结点
- 判断是否为满二叉树
- ###
- ###

哈希表

- C++STL中的哈希集合

HashSet

HashMap

注意

1. 如无特殊说明，有序即为由小到大

重要算法

Prim

```
struct CloseEdge{
    VerTexType adjvex;
    ArcType lowcost;
}
CloseEdge closeedge[MVNum];

void MiniSpanTree_Prim(AMGraph G, VerTexType vex){
    k = locateVex(G, vex);
    for (int i=0;i<G.vexnum;i++)
        if (i =/= k) closeedge[i] = {vex, G.arcs[k][i]};
    closeedge[k].lowcost=0;
    for (int i=1;i<G.vexnum;i++){
        k=Min(closeedge);
        v0=closeedge[k].adjvex;
        v1=G.vexs[k];
        cout << v0 << v1 << endl;
        closeedge[k].lowcost=0;
        for (int j=0;j<G.vexnum;j++){
            if (closeedge[j].lowcost>G.arcs[k][j])
                closeedge[j]={G.vexs[k],G.arcs[k][j]};
        }
    }
```

```
}
```

Kruskal

```
struct Edge{
    VerTexType s; // start
    VerTexType e; // end
    ArcType lowcost;
}

Edge edges[arcnum];
int vexset[MVNum];

void MiniSpanTree_kruskal(AMGraph G){
    sort(edges); // min to max
    for (int i=0;i<G.vexnum;i++)
        vexset[i]=i;
    for(int i=0;i<G.arcnum;i++){
        v0=LocateVex(G, edges[i].s);
        v1=LocateVex(G, edges[i].e);
        v0s=vexset[v0];
        v1s=vexset[v1];
        if (v1s≠v0s){
            cout<<edges[i].s<<edges[i].e<<endl;
            for(int j=0;j<vexnum;++j)
                if (vexset[j]==v1s)
                    vexset[j]=v0s;
        }
    }
}
```

Floyd

```
void ShortestPath_Floyd(AMGraph G){
    // 初始化路径信息
    for (int i=0;i<G.vexnum;++i){
        for (int j=0;j<G.vexnum;++j){
            D[i][j]=G.arcs[i][j];
            if (D[i][j]≠MaxInt) Path[i][j]=i; // 有弧 前驱置为i
            else Path[i][j]=-1;               // 没有弧 前驱置为-1
        }
    }
```

```
    for (int k=0;k<G.vexnum;++k){
        for (int i=0;i<G.vexnum;++i){
            for (int j=0;j<G.vexnum;++j){
                if (D[i][k]+D[k][j] < D[i][j]){ // i经k到j有更短的路径
                    D[i][j]=D[i][k]+D[k][j];
                    Path[i][j]=Path[k][j];
                }
            }
        }
    }
}
```

TopologicalSort

```
Status TopologicalSort(Graph G, int topo[]){
    FindInDegree(G, indegree); // 求出各顶点的入度存入数组
    InitStack(S); // 初始化栈
    for(int i=0;i<G.vexnum;++i) // 入度为0进栈
        if (indegree[i]==0) Push(S, i);
    m=0;
    while(!StackEmpty(S)){
        Pop(S, i);
        topo[m]=i;
        ++m;        // 出栈保存当前序号
        p=G.vertices[i].firstarc; // p指向vi的第一个邻接点
        while(p≠NULL){
            k=p→adjvex;
            --indegree[k];      // 入度减一
            if (indegree[k]==0) Push(S, k); // 度数为0的节点进栈
            p=p→nextarc;
        }
    }
    if (m<G.vexnum) return ERROR;
    else return OK;
}
```

###

堆排序

```
void maxHeapify(int arr[], int s, int e){
    int dad=s;
    int son=dad*2+1;
```

```
    while(son≤e){
        if (son+1≤e&&arr[son]<arr[son+1])
            ++son;
        if (arr[dad]>arr[son])
            return ;
        else{
            swap(arr[dad],arr[son]);
            dad=son;
            son=dad*2+1;
        }
    }
}

void heapSort(int arr[], int len){
    for(int i=len/2-1;i≥0;i--)
        maxHeapify(arr, i, len-1);
    for(int i=len-1;i>0;i--){
        swap(arr[0], arr[i]);
        maxHeapify(arr,0,i-1);
    }
}
```

快速排序

```
void QuickSort(int arr[], int s, int e){
    // if不能忘
    if (s<e){
        int pivot=arr[s];
        int low=s, high=e;
        while(low<high){
            // 此处为≥
            while (low<high&&arr[high]≥pivot) --high;
            if (low<high) arr[low]=arr[high];
            // 此处为≥
            while (low<high&&arr[low]≤pivot) ++low;
            if (low<high) arr[high]=arr[low];
        }
        arr[low]=pivot;
        QuickSort(arr, s, low-1);
        QuickSort(arr, low+1, e);
    }
}
```

插入排序

```
// 取出无序序列的头元素在已排好的序列中找到位置插入
```

选择排序

```
// 从无序序列中选择最大或最小的元素放在已排好序列的末尾
```

###

```
阿斯弗
```

二分查找

```
// little to big
int binarySearch(int arr[], int s, int e, int target){
    int mid=0;
    // 注意如果跳出循环 则s指向最小的大于target的数
    while(s≤e){
        mid=s+(e-s)/2;
        if (arr[mid]==target)
            return mid;
        else if (arr[mid]>target)
            e=mid-1;
        else
            s=mid+1;
    }
    return -1;
}
```

huffmanTree

```

```

###

```

```

矩阵

矩阵旋转90

```
// nxn
const int n=4;
void Transpose(int arr[n][n]){
    int tmp;
    // 转置
    for(int i=0;i<n;++i)
        // 此处为i即可
        for(int j=0;j<i;++j){
            tmp = arrp[i][j];
            arr[i][j]=arr[j][i];
            arr[j][i]=tmp;
        }

    // 水平镜像
    for (int i=0;i<n;++i)
        for (int j=0;j<n/2;j++){
            tmp=arr[i][j];
            arr[i][j]=arr[i][n-1-j];
            arr[i][n-1-j]=tmp;
        }
}
```

矩阵翻转180

```
const int n=4;
void Rotation(int arr[n][n]){
    int tmp;
    // 水平镜像
    for (int i=0;i<n;++i)
        for(int j=0;j<n/2;++j){
            tmp=arr[i][j];
            arr[i][j]=arr[i][n-j-1];
            arr[i][n-j-1]=tmp;
        }
    // 垂直镜像
    for (int i=0;i<n/2;++i)
        for (int j=0;j<n;++j){
```

```
            tmp=arr[i][j];
            arr[i][j]=arr[n-i-1][j];
            arr[n-i-1][j]=arr[i][j];
        }
}
```

矩阵水平镜像（见矩阵旋转90

矩阵转置（见矩阵旋转90

矩阵垂直镜像（见矩阵旋转180

矩阵乘法

```
// nxn
void MatrixMultiply(int **A, int **B, int **res, int n){
    for(int i=0;i<n;++i){
        for (int j=0;j<n;++j){
            res[i][j]=0;
            for(int k=0;k<n;++k)
                res[i][j]=A[i][k]*B[k][j];
        }
    }
}
```

###

```
阿斯弗
```

链表

循环链表的基本操作 ☆

```

```

反转链表

```
// 假设有头结点
void reverse(ListNode *head){
    if (head->next==nullptr)
        return ;
    ListNode *pre=head->next,*work=head->next->next, *cur=nullptr;
    while(work!=nullptr){
        pre->next=cur;
        cur=pre;
        pre=work;
        work=work->next;
    }
    pre->next=cur;
    head->next=pre;
}
```

相交链表

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB){
    if (headA==nullptr||headB==nullptr)
        return nullptr;
    ListNode *A=headA, *B=headB;
    while(A!=B){
        A->next==nullptr?A=headB:A=A->next;
        B->next==nullptr?B=headA:B=B->next;
    }
    return A;
}
```

删除链表倒数第N个节点 ☆

```
// 双指针
ListNode *removeNthFromEnd(ListNode *head, int n){
    // 构造一个临时头结点对付特殊情况——删除头结点
    ListNode firt;
    first.next=head;
    ListNode *front=&first, *rear=&first;
    int i=0;
    while(i<n){
        rear=rear->next;
        ++i; // !!!!!!!
```

```
    }
    while(rear->next){
        front=front->next;
        rear=rear->next;
    }
    rear=front->next;
    front->next=front->next->next;
    delete rear;
    return first.next;
}
```

链表中第K个节点

```
// 双指针
ListNode *getKthFormEnd(ListNode *head, int k){
    ListNode *front=head, *rear=head;
    while(k>0){
        rear=rear->next;
        k--;
    }
    while(rear){
        rear=rear->next;
        front=front->next;
    }
    return front;
}
```

合并两个有序链表为递减链表

```
// 假设两个链表没有头结点
// 采用头插法 优先插入小的
ListNode *mergeList(ListNode *A, ListNode *B){
    ListNode dummy;dummy.next=nullptr;
    ListNode *head=&dummy;
    ListNode *tmp;
    while (A&&B){
        if (A->val>B->val){
            tmp=head->next;
            head->next=B;
            B=B->next;
            head->next->next=tmp;
        }else{
            tmp=head->next;
```

```
            head->next=A;
            A=A->next;
            head->next->next=tmp;
        }
    }
    // A不为空
    while(A){
        tmp=head->next;
        head->next=A;
        A=A->next;
        head->next->next=tmp;
    }
    // B不为空
    while(B){
        tmp=head->next;
        head->next=B;
        B=B->next;
        head->next->next=tmp;
    }
    return head->next;
}
```

求有序链表(集合)A与B的并集 ☆

```
// 带头结点
ListNode *Union(ListNode *ha, ListNode *hb){
    ListNode *head=ha, *pa=ha->next, *pb=hb->next, *tmp;
    while(pa&&pb){
        if (pa->val==pb->val){
            head->next=pa;
            pa=pa->next;
            head=head->next;
            tmp=pb;
            pb=pb->next;
            delete tmp;
        }else if (pa->val>pb->val){
            head->next=pb;
            pb=pb->next;
            head=head->next;
        }else{
            head->next=pa;
            pa=pa->next;
            head=head->next;
        }
```

```
    }
    /* 没必要再一个一个加入
    while(pa){
        head->next=pa;
        pa=pa->next;
        head=head->next;
    }
    while(pb){
        head->next=pb;
        pb=pb->next;
        head=head->next;
    }
    */
    // 直接复制过来
    if (pa) head->next=pa;
    else head->next=pb;
    return ha;
}
```

求有序链表(集合)A与B的交集

```
ListNode* AUB(ListNode *A, ListNode *B){
    ListNode *pa=A->next, *pb=B->next, *tmp;
    ListNode *work=pa;
    while(pa&&pb){
        if (pa->val==pb->val){
            work->next=pa;
            pa=pa->next;
            work=work->next;
            tmp=pb;
            pb=pb->next;
            delete tmp;
        }
        else if (pa->val>pb->val){
            tmp=pb;
            pb=pb->next; // 不能只移动指针 还要删除当前指针指向的内存块 否则会发生内存泄漏
            delete tmp;
        }
        else{
            tmp=pa;
            pa=pa->next;
            delete tmp;
        }
```

```
    }
    // 将剩下的节点直接挂载到B上，懒得删除了
    if (pa) B->next=pa;
    else B->next=pb;
    work->next=nullptr;
    return A;
}
```

链表的直接插入排序

```
// 147 leetcode
ListNode* insertSortList(ListNode * head){
      if (head==nullptr)
            return head;
        ListNode *dummyHead=new ListNode(0);
        dummyHead->next=head;
        ListNode *curr=head->next, *lastSorted=head;
        while(curr!=nullptr){
            if(lastSorted->val<=curr->val)
                lastSorted=curr;
            else{
                ListNode *prev=dummyHead; // 从头开始
                while(prev->next->val<=curr->val)
                    prev=prev->next;
                lastSorted->next=curr->next;
                curr->next=prev->next;
                prev->next=curr;

            }
            curr=lastSorted->next;
        }
        return dummyHead->next;
}
```

###

```

```

###

```

```

栈和队列

两个栈实现一个队列

```
class MyQueue{
    stack<T> in; // 入队栈
    stack<T> out; // 出队栈
public:
    void enQueue(T v){
        in.push(v);
    }

    bool empty(){
        return in.empty()&&out.empty();
    }

    T deQueue(){
        if (this→empty())
            Error;
        if (out.empty()){
            while (!in.empty()){
                out.push(in.top());
                in.pop();
            }
        }
        T tmp=out.top();
        out.pop();
        return tmp;
    }

}
```

包含min函数的栈

```
class MinStack{
    Stack<T> s;
    Stack<T> min;
public:
    void push(T v){
        s.push(v);
        if (min.empty())
```

```
            min.push(v);
        else{
            if (min.top()>v)
                min.push(v);
        }
    }

    T min(){
        return min.top();
    }

    void pop(){
        if (s.empty())
            return;
        if (s.top()==min.top()){
            s.pop();
            min.pop();
        }
        else{
            s.pop();
        }
    }
}
```

括号匹配

```
bool isValid(string s){
    if (s.size()%2==1)
        return false;
    // 用逗号隔开
    unordered_map<char, char> map={
        {')','('},
        {']','['},
        {'}','{'}
    }
    stack<char> stk;
    for(auto ch:s){
        if (map.count(ch)){
            if (stk.empty() || stk.top()≠map[ch])
                return false;
            stk.pop();
        }else
            stk.push(ch);
    }
```

```
    return stk.empty();
}
```

共享栈 ☆

```
class ShareStack{
    ElemType
public:

}
```

###

```
阿斯弗
```

###

```
阿斯弗
```

###

```
阿斯弗
```

线性表

删除数组中的最小元素

```
bool deleteMinElem(SqList &L, ElemType &v){
    if (L.length==0)
        return false;
    int min=0;
    // 找到最小元素
    for(int i=1;i<L.length;i++)
        if (L.data[i]<L.data[min])
            min=i;
    v=L.data[min];
    // 从min开始逐个前移一位
    for(int i=min+1;i<length;i++)
```

```
        L.data[i-1]=L.data[i];
    /* 或者直接把表尾元素复制过来
        L.data[min]=L.data[L.lenght-1];
    */
    L.lenght--; // 非常重要！！！！
    return true;
}
```

反转线性表

```
void ReverseSqList(SqList &L){
    ElemType tmp;
    for(int i=0;i<L.length/2;i++){
        tmp=L.data[i];
        L.data[i]=L.data[L.length-1-i];
        L.data[L.length-1-i]=tmp;
    }
}
```

移除线性表中等于x的元素 ☆

```
void delete1Elem(SqList &L, ElemType x){
    int k=0;
    for(int i=0;i<L.length;++i)
        L.data[i]==x?x++:L.data[i-k]=L.data[i];
    L.length-=k; // 不能忘！！！
}
```

删除有序线性表大于s小于t的元素 ☆

```
bool deleteElemS2T(SqList &L, ElemType s, ElemType t){
    if (s≥t||L.length==0)
        return false;
    int count=0;
    int i=0;
    // 统计介于s到t之间的元素
    for(;i<L.length;)
        if (L.data[i]>s&&L.data[i]<t)
            count++;
        else if (L.data[i]>t)
            break;
    if (count==0)   // 没有找到
```

```
        return false;
    for(;i<L.length;++i)
        L.data[i-count]=L.data[i];
    L.length-=count; // !!!!!!!!!!
    return true;
}
```

```
// 第二版 更精简
void deleteElemS2T(SqList &L, ElemType s, ElemType t){
    int count=0;
    for(int i=0;i<L.length;++i)
        L.data[i]>s&&L.data[i]<t?++count:L.data[i-count]=L.data[i];
    L.lenght-=count;
}
```

删除递增有序表的相同元素 ☆

```
void deleteElem(SqList &L){
    if (L.length==0)
        return ;
    int i=0, j=1;
    while (j<L.length){
        if (L.data[i]≠L.data[j]){
            L.data[i+1]=L.data[j];
            ++i;++j;
        }
        else
            ++j;
    }
    L.length=i+1;
}
// 如果考试无思路或者是无需表 可以使用哈希表来判断是否重复
```

将两个有序表合并为一个有序表

```
bool mergeSqList(SqList L1, SqList L2, SqList &L3){
    if (L1.length+L2.length>MaxSize) // !!!!!!!
        return false;
    int i=0, j=0, k=0;
    while(i<L1.length&&j<L2.length)
        L1.data[i]>L2.data[j]?
L3.data[k++]=L2.data[j++]:L3.data[k++]=L1.data[i++];
    while(i<L1.length)
        L3.data[k++]=L1.data[i++];
    while(j<L2.length)
        L3.data[k++]=L1.data[j++];
    L3.length=k;
    return true;
}
```

寻找有序表中x的位置，如果不存在则插入元素x，且仍然保持线性表有序i

```

```

循环左移

```
bool cycleShiftLeft(ElemType arr[], int len, int num){
    if(num⩽0)
        return false;
    numm=num%len;
    reverse(arr, 0, num-1);
    reverse(arr, num, len-1);
    reverse(arr, 0, len-1);
    return true;
}

bool reverse(ElemType arr[], int s, int e){
    if (s⩾e)
        return false;
    ElemType tmp;
    while(s<e){
        tmp=arr[s];
        arr[s++]=arr[e];
        arr[e--]=tmp;
    }
}
```

数组中重复的数字

```
int findRepeatNumber(vector<int> &arr){
    unordered_set<int> hashset;
    for(auto v:arr)
        if (hashset.count(v)>0)
            return v;
        else
            hashset.insert(v);
    return -1;
}
```

有序递增数列查找两个和为s的数

```
// 顺序二分查找 O(nlogn)
// 双指针法
vector<int> twoSum(vector<int> arr, int target){
    vector<int> res;
    int left=0, right=arr.size();
    while(left<right){
        if (target-arr[left]==arr[right]){
            res.push_back(arr[left]);
            res.push_back(arr[right]);
            return res;
        }
        else if (target-arr[left]>arr[right])
            ++left;
        else
            --right;
    }
    return res;
}
```

将顺序表分成偶数和奇数两部分

```
// 或者根据用某个数把线性表划分为两部分
// 一次快速排序
// 此处简写为数组
// 假设左侧放奇数
void divideOddEven(int arr[], int len){
    int pivot=arr[0];
    int low=1, high=len-1;
```

```
    while(low<high){
        while(low<high&&arr[high]%2==0) --high;
        arr[low]=arr[high];
        while(low<high&&arr[low]%2==1) ++low;
        arr[high]=arr[low];
    }
    arr[low]=pivot;
}
```

###

```

```

###

```

```

###

```

```

字符串

只出现一次的字符

```
char findUniqueCh(string s){
    unordered_map<char, bool> hashmap;
    for (auto ch:s)
        if (hashmap.count(ch)<=0)
            hashmap[ch]=true;
        else
            hashmap[ch]=false;
    for(int i=0;i<s.size();++i)
        if (hashmap[s[i]])
            return s[i];
    return ' ';
}
```

###

```
```

###

```
```

二叉树

平衡二叉树

```
```

镜像二叉树 ☆

```
// 返回当前二叉树的镜像
TreeNode *mirrorTree(TreeNode *root){
    if (root==nullptr)
        return nullptr;
    TreeNode *tmp=root->left;
    root->left=mirrorTree(root->right);
    root->right=mirrorTree(tmp);
    return root;
}
```

二叉树的高度

```
 int height(TreeNode *root){
     if (root==nullptr)
         return 0;
     int leftH=height(root->left);
     int rightH=height(root->right);
     return leftH>rightH?leftH+1:rightH+1;
 }
```

求二叉树叶子节点的个数

```
int getLeafNum(TreeNode *root){
    if (root==null)
        return 0;
    if (root->left==null&&root->right==null)
        return 1;
    else
        return getLeafNum(root->left)+getLeafNum(root->right)
}
```

中序先序后序递归遍历

```
void Inorder(TreeNode *root){
    if (root==null)
        return ;
    Inorder(root->lchild);
    cout << root->data;
    Inorder(root->rchild);
}

void PreOrder(TreeNode *root){
    if (root==null)
        return ;
    cout << root->data;
    PreOrder(root->lchild);
    PreOrder(root->rchild);
}

void PostOrder(){
    if (root==null)
        return ;
```

```
    PostOrder(root->lchild);
    PostOrder(root->rchild);
    cout << root->data;
}
```

层次遍历

```
void levelTraversal(TreeNode *root){
    if (root==nullptr)
        return ;
    TreeNode *tmp=root;
    queue<TreeNode*> q;
    q.push(tmp);
    while(!q.empty()){
        tmp=q.front();
        q.pop();
        cout << tmp->data;
        if (tmp->left!=nullptr)
            q.push(tmp->lchild);
        if (tmp->right!=nullptr)
            q.push(tmp->rchild);
    }
    return ;
}
```

查找二叉树中值为X的结点

```
TreeNode *findElem(TreeNode *root, ElemType x){
    if (root==nullptr)
        return nullptr;
    if (root->data==x)
        return root;
    // 现在左子树中查找
    TreeNode * tmp=findElem(root->lchild);
    if (tmp==nullptr)
        return findElem(root->rchild);
    return tmp;

}
```

以上题为基础并输出该结点的父亲节点

```
int printFather(TreeNode *root, ElemType x){
    if (root==nullptr)
        return 0;
    if (root->data==x)
        return 1;
    if (root){
        // 只要子节点中有符合条件的，就输出出来
        if (printFather(root->lchild) || printFather(root->rchild)){
            cout << root->data;
            return 1;
        }
    }
    return 0;
}
```

求先序/中序/后序历序列中第K个结点

```
// 先序
int i=0;
void PreOrderK(TreeNode *root, int k){
    if(root==nullptr)
        return ;
    i++;
    if (i==k){
        cout << root->data;
        return ;
    }
    PreOrderK(root->lchild, k);
    PreOrderK(root->rchild, k);
}

// 中序
int i=0;
void InOrderK(TreeNode *root, int k){
    if (root==nullptr)
        return ;
    InOrderK(root->lchild, k);
    i++;
    if (k==i){
        cout << root->data;
        return ;
    }
    InOrder(root->rchild, k);
}
```

二叉树中最小值的结点

```
int Sum(TreeNode *root){
    if (root==nullptr)
        return 0;
    return (root->data+Sum(root->lchild)+Sum(root->rchild));
}
```

统计所有结点数

```
int all=0;
void countAllNode(TreeNode *root){
    if (root≠nullptr){
        all++;
        countAllNode(root->lchild);
        countAllNode(root->rchild);
    }
}

int CountAllNode(TreeNode *root){
    if (root==nullptr)
        return 0;
    return CountAllNode(root->lchild)+CountAllNode(root->rchild)+1;
}
```

统计值为X的结点个数

```
int CountNumValueX(TreeNode *root, ElemType x){
    if (root==nullptr)
        return 0;
    int lnum=CountNumValueX(root->lchild,x);
    int rnum=CountNumValueX(root->rchild,x);
    if (root->data==x)
        return lnum+rnum+1;
    return lnum+rnum;
}
```

寻找值为X结点的所有祖先结点

```
bool PrintFather(TreeNode *root, ElemType x){
    if (root==nullptr)
        return 0;
    if (root->data==x)
        return 1;
    if (PrintFather(root->rchild)||PrintFather(root->lchild))
    {
        cout << root->data << endl;
        return 1;
    }
    return 0;
}
```

寻找结点值为X的双亲结点

```
TreeNode *parent(TreeNode *root, ElemType x){
    if (root==nullptr)
        return nullptr;
    TreeNode *tmp;
    // 先在左子数中查找
    if (root->lchild)
        tmp=root->lchild->data==x?root:parent(root->lchild,x);
    if (tmp)
        return tmp;
    else{
        if (root->rchild)
            return root->rchild->data==x?root:parent(root->lchild,
x);
    }
    return nullptr;
}
```

判断是否为满二叉树

```
int height(TreeNode *root){
    if (root==nullptr)
        return 0;
    int lh=height(root->lchild);
    int rh=height(root->rchild);
    return max(rh,lh)+1;
}

void count(TreeNode *root, int &num){
```

```
    if (root){
        num++;
        count(root->lchild);
        count(root->rchild);
    }
}

bool isFull(TreeNode *root){
    if (root==nullptr) return 0;
    int num=0;
    count(root,num);
    int h=height(root);
    return num==pow(2, h)-1;
}
```

###

```
```

###

```
```

哈希表

C++STL中的哈希集合

```
#include <unordered_set>
using namespace std;

unordered_set<T> hashset;
// 常用操作
// 插入
hashset.insert(v);
// 删除
hashset.erase(v);
```

```
// 判断某元素是否存在 出现的次数
hashset.count(v)==0
// 遍历
for (auto p=hashset.begin();p≠hashset.end();++p)
    cout << *p << endl;
```

HashSet

```
class MyHashSet{
public:
    vector<T> backets[10000];
    MyHashSet(){}
    int getIndex(T key){
        return H(key); // key%10000;
    }

    void add(T key){
        int index=getIndex(key);
        if (!contain(key))
            backets[index].push_back(key);
    }

    void remove(T key){
        int index=getIndex(key);
        if (backets[index].size()==0)
            return ;
        for(int i=0;i<backets[index].size();++i)
            if (backets[index][i]==key)
                backets[index].earse(backets[index].begin()+i);
    }

    bool contain(T key){
        int index=getIndex(key);
        if (backets[index].size()==0)
            return false;
        else{
            for(auto v:backets[index])
                if (v==key)
                    return true;
        }
        return false;
    }

};
```

HashMap

###

```

```

###

```

```