

重点算法汇总

顺序表

逆置

```
void Invert()
{
    int i = 0;
    int j = length - 1;
    int mid = (l.length)/2;

    for(; i < mid; ++i, --j)
    {
        int temp = l.data[i];
        l.data[i] = l.data[j];
        l.data[j] = temp;
    }
}
```

删除

- 核心: i 负责遍历整个序列, k 不变、留在满足删除条件的位置
- 删除某个范围: i 从 0 开始遍历, 覆盖时: 先覆盖, 再 k++

```
void Delete(Splist& l, int x)
{
    int i = 0;
    int k = 0;

    for(; i < l.length; i++)
    {
        if( 如果不在XXX范围内 )
        {
            l.data[k] = l.data[i];
            k++;
        }
    }

    l.length = k;
}
```

- 删除重复值: i 从 1 开始遍历, 覆盖时: 先覆盖, 再 k++

```
void Delete(Splist& l, int x)
{
    int i = 1;
```

```

int k = 0;

for(;i<l.length;i++)
{
    // 如果当前不为重复值
    if(l.data[i]!=l.data[k])
    {
        // 先覆盖
        l.data[k] = l.data[i];

        // 再移k
        k++;
    }
}

l.length = k;
}

```

合并

- 保持递增合并

```

Sqlist* Merge(Sqlist& l1,Sqlist& l2,Sqlist& l3)
{
    l3.length = l1.length + l2.length;
    int i = j = k = 0;

    while(i<l1.length && j<l2.length)
    {
        if(l1.data[i]<l2.data[j])
        {
            l3.data[k] = l1.data[i];
            i++;
            k++;
        }
        else
        {
            l3.data[k] = l2.data[j];
            j++;
            k++;
        }
    }

    while(i<l1.length)
    {
        l3.data[k] = l1.data[i];
        k++;
        i++;
    }

    while(j<l2.length)

```

```

{
    l3.data[k] = l2.data[j];
    k++;
    j++;
}

l.length = k;
}

```

- 求中位数：利用上述算法合并，合并完成后，中位数下标为 $\text{mid} = (\text{l1.length} + \text{l2.length} - 1) / 2$

虚拟数组

- 找出数组中未出现的最小正整数
- 如果当前数组的长度为 n ，则当前能够出现的最大最小正整数只能是 $n+1$
 - 比如当前数组长度为 5，元素为 1, 2, 3, 4, 5，则此时最小正整数为 6；
 - 比如当前数组长度为 5，元素为 1, 2, 100, 4, 5，则此时最小正整数为 3；
 - 比如当前数组长度为 5，元素为 100, 200, 300, 400, 500，则此时最小正整数为 1；
- 因此可以用一个长度为 n 的记录数组，用来记录大小在 $1 \sim n$ 这个范围内的元素有没有出现。如果出现，就在对应的下标处将其记为 1。最后只要遍历这个记录数组，找到的第一个值为 0 的元素即为最小正整数；

```

int GetNullMinNumber(int a[], int n)
{
    // 初始化
    int record = new int[n];
    for(int i = 0; i < n; i++)
    {
        record[i] = 0;
    }

    // 遍历，为符合条件的数据设置其对应的虚拟数组
    for(int i = 0; i < n; i++)
    {
        // 由于是返回最小正整数，因此a[i]需要大于0
        // 并且当前n代表的是长度，因此n就代表了当前n个元素可以存在的最大最小整数
        // 比如当n=3时，当前有3个元素，因此最大的最小正整数为3，因此要小于n+1
        if(a[i] < n+1 && a[i] > 0)
        {
            record[a[i]-1] = 1;
        }
    }

    // 找最小
    int minNumber = 0;
    for(; minNumber < n; minNumber++)
    {
        if(record[minNumber] == 0)

```

```

        {
            break;
        }
    }
    Delete[] record;

    return (maxNumber+1);
}

```

链表

结构体

```

typedef struct LNode
{
    struct LNode* next;
    int data;
}LNode,*Linklist;

```

删除

- 递归删除

```

void Delete(LNode* l,int x)
{
    if(l==null)
    {
        return;
    }
    if(l->data == x)
    {
        LNode* p = l;
        l = l->next;
        free(p);
        Delete(l,x);
    }
    else
    {
        Delete(l->next,x);
    }
}

```

- 非递归删除
- 判断 p 指针指向的下一个结点的值是否等于 x，如果等于，就进行删除操作，让 p 指向 `p->next` 指向的下一个结点；

```

void Delete(LNode* l,int x)
{
    LNode* p = l;

```

```

while(p->next!=null)
{
    if(p->next->data == x)
    {
        LNode* temp = p->next;
        p->next = temp->next;
        free(temp);
    }
    else
    {
        p = p->next;
    }
}
}

```

- 删除链表最小值

```

void DeleteMin(LNode* &l)
{
    LNode* p = l;
    LNode* minP = l;

    while(p->next!=null)
    {
        // 注意min指向的是要删除结点的前一个结点
        if(p->next->data < min->next->data)
        {
            min = p;
        }
        p = p->next;
    }

    if(minP->next!=null)
    {
        LNode* q = minP->next;
        minP->next = q->next;
        free(q);
    }
}

```

- 删除重复值

```

void DeleteRepeatNumber(LNode* l)
{
    if(l==null)
    {
        return;
    }
}

```

```

LNode* pre = l->next;
LNode* p = null;

while(pre->next!=null)
{
    p = pre->next;
    if(pre->data == p->data)
    {
        pre->next = p->next;
        free(p);
    }
    else
    {
        pre = pre->next;
    }
}
}

```

逆置

- 头插法：遍历链表，将每一个元素插到当前的头指针的前面；

```

void Invert(LNode*& l)
{
    LNode* p = l->next;

    // 需要将头指针断开，这个别忘了
    l->next = null;

    while(p!=null)
    {
        LNode* q = p->next;
        p->next = l->next;
        l->next = p;
        p = q;
    }
}

```

拆分

- 将链表按照下标，拆成奇偶两列；
- 每两个元素是一对，两个元素中的第一个为奇元素，要留在 A 中；两个元素中的第二个为偶元素，要加入到 B 中；
- 因此需要定义三根指针，p 指针用于遍历整个链表，ra 指针用于指向一对元素中的第一个，rb 用于指向一对元素中的第二个；
- ra、rb 相当于 A、B 两个链表的链尾，利用链尾执行插入操作，将当前 p 结点插入到 A 和 B 中；

```

// 返回值为B链表
LNode* SplitLink(LNode* &A)
{
    LNode* B = new LNode*;
    LNode* p = A->next;
    LNode* ra = A;
    LNode* rb = B;

    A->next = null;
    B->next = null;

    while(p!=null)
    {
        // 当前p处于一对元素中的第一个位置
        // 将当前p加入到A序列中
        ra->next = p;
        ra = p;

        // p进入到第二个位置
        p = p->next;

        // 当前p处于一对元素中的第二个位置
        rb->next = p;
        rb = p;

        if(p!=null)
        {
            p = p->next;
        }
    }

    ra->next = null;

    return B;
}

```

- 其他:

11、将一个单链表{a1,b1,a2,b2……an,bn}拆分成

- { a1,a2……an }和{ bn,bn-1,……b1 }

- 思想：将链表的奇数列和偶数列分别放入 A、B 中，只不过偶数列采用头插法，每次将高序号的元素插入到 B 链表的头部，保证在 B 中插入的是倒置的序列；

合并

- 两个递增有序的单链表，设计算法成一个非递减有序的链表

```

void Merge(LNode* &A,LNode* &B)
{
    LNode* pa = A->next;

```

```

LNode* pb = B->next;

// p代表当前A链表的头元素
LNode* p = A;
A->next = null;

while(pa!=null&&pb!=null)
{
    // pa的元素更小，将pa结点插入到A结点中
    if(pa->data<pb->data)
    {
        p->next = pa;
        pa = pa->next;
        p = p->next;
    }
    else
    {
        p->next = pb;
        pb = pb->next;
        p = p->next;
    }
}
if(pa!=null)
{
    p->next = pa;
}
if(pb!=null)
{
    p->next = pb;
}
}

```

公共元素

- 从 A、B 中找出公共元素，放入 C 中

```

LNode* CommonElement(LNode* A,LNode* B)
{
    LNode* C = new LNode*;
    LNode* pc = C;

    LNode* pa = A->next;
    LNode* pb = B->next;

    while(pa!=null&&pb!=null)
    {
        if(pa->data==pb->data)
        {
            LNode* newNode = new LNode*;
            newNode->data = pa->data;
            pc->next = newNode;

```

```

        pc = pc->next;
        pa = pa->next;
        pb = pb->next;
    }
    else if(pa->data<pb->data)
    {
        pa = pa->next;
    }
    else if(pa->data>pb->data)
    {
        pb = pb->next;
    }
}

pc->next = null;
return C;
}

```

查找

- 查找单链表中倒数第 k 个结点，若成功则输出该节点的 data，并返回 1，否则返回 0
- 思路：
 - 使用双指针来遍历单链表。两个指针间的距离为 k，当 p 指针遍历完链表、到达 `null` 时，当前 q 指针的位置即为倒数第 k 个结点；
 - 为了维持两指针的距离，需要先让 p 指针走 k 次。当 p 指针走了 k 次之后，p 和 q 一起后移；

```

int ReturnLast(LNode* l,int k)
{
    LNode* p = l->next;
    LNode* q = l;
    int i = 1; // 两指针间的距离
    while(p!=null)
    {
        // 比如k=3,i=2时，i会自加到3，下一次循环时执行else，此时距离为3
        if(i<k)
        {
            i++;
            p = p->next;
        }
        else
        {
            p = p->next;
            q = q->next;
        }
    }

    // 如果当前链表只有k-1个元素，但却要返回第k个结点，则此时q指针一直不会移动，返回失败
    if(q == l)
    {

```

```

        return 0;
    }
    else
    {
        std::cout<<q->data;
        return 1;
    }
}

```

循环双链表

- 判断是否对称
- 两根指针，分两个方向同时走，判断它们值是否相等，如果相等则继续走，一直走到 `p->next==q` 或者是 `p==q` 时结束；

```

bool IsSymmetry(LNode* l)
{
    LNode* p = l->next;
    LNode* q = l->prior;

    // p和q没走到一起
    while(p->next!=q&&p!=q)
    {
        if(p->data==q->data)
        {
            p = p->next;
            q = q->next;
        }
        else
        {
            return false;
        }
    }

    // 跳出循环时，两者data都相等，则返回真
    return true;
}

```

综合操作

24、给定一个单链表 $L(a_1, a_2, a_3, \dots, a_n)$, 将其重新

- 排列为 $(a_1, a_n, a_2, a_{n-1}, \dots)$
- 分成几个小步骤：
 - 第一步：将单链表中间拆分为两个链表
 - 利用双指针（`low` 和 `fast`）遍历整个链表，`low` 走一步时，`fast` 走两步，当 `fast` 走到 `null` 时，此时 `low` 在链表的中间，将 `low` 及其左边的元素保留在 `L` 中，后续元素放入新链表 `L1` 中；
 - 第二步：对 `L1` 链表进行逆置

- 第三步：将 **L1** 链表中结点逐个插入 L 中

栈与队列

结构体

```
// 栈
typedef struct stack
{
    char data[maxsize];
    int top;
}stack;

// 队列
typedef struct queue
{
    int data[maxsize];
    int rear,front;
}queue;
```

基础操作

- Q 是一个队列，S 是一个空栈，编写算法使得队列中元素逆置

```
void Invert(queue* &queue,stack* &stack)
{
    while(queue.front!=queue.rear)
    {
        // front表示当前队头，在数组的低位置。出队时、front往前移动，front也要自加1
        int data = queue.data[front++];

        // stack默认为空，因此top默认指向-1，如果要入栈，需要先对top自加，然后将元素放到
        // 对应位置
        stack.data[++top] = data;
    }

    while(stack.top!=-1)
    {
        // 取出栈顶元素
        int data = stack.data[top--];

        // 将栈顶元素入队
        queue.data[rear++] = data;
    }
}
```

栈的应用

- 题目：

2、判断单链表的全部 n 个字符是否中心对称

- 总结:

- 第一步: 遍历单链表, 确定链表长度, 获得链表的中心位置 $n/2$
- 第二步: 将前一半的元素入栈, 同时指针停留在链表的中心位置
- 第三步: 将栈内元素出栈的同时、指向中心位置的指针往后移动, 逐个比较是否相同, 如果不相同则为 `false`;

- 题目:

4、判断一个表达式中括号是否配对(假设只包含圆括号)

- 总结:

- 使用栈进行匹配, 分为两个动作;
- 左括号: 啥也不用管, 入栈就完事了;
- 右括号: 将栈顶弹出, 看是否和右括号匹配。如果匹配则继续, 如果不匹配就配对失败;

```
bool isMatch(string str, stack& stack)
{
    int i = 0;
    while(str[i]!='\0')
    {
        switch(str[i])
        {
            case '(':
                push(stack, '(');
                break;
            case ')':
                char tmp = pop(stack);
                if(tmp == '(')
                {
                    // 没事
                }
                else
                {
                    return false;
                }
            }

            i++;
        }

        if(stack.top == -1)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

```
}  
}
```

树

结构体

```
typedef struct BTreeNode  
{  
    char data;  
    struct BTreeNode* lchild,rchild;  
}BTreeNode,*BiTree
```

基本操作

- 操作型

```
int CalNumber(BTreeNode* bt,int& n)  
{  
    if(p==null)  
    {  
        return 0;  
    }  
    n++;  
    CalNumber(bt->lchild,n);  
    CalNumber(bt->rchild,n);  
}
```

- 计算型

```
int CalNumber(BTreeNode* bt)  
{  
    if(bt==0)  
    {  
        return 0;  
    }  
  
    int n1 = CalNumber(bt->lchild);  
    int n2 = CalNumber(bt->rchild);  
    return n1+n2+1;  
}
```

常见操作

- 计算二叉树中所有双分支的节点个数

```
int count(BTreeNode* p)  
{  
    if(p==null)
```

```

{
    return 0;
}
if(p->lchild!=null && p->rchild!=null)
{
    // 左右结点都不为空时，return的时候多加1
    int n1 = count(p->lchild);
    int n2 = count(p->rchild);
    return n1+n2+1;
}
else
{
    // 左右结点有一个为空，return的时候直接求和左右结点的返回值
    int n1 = count(p->lchild);
    int n2 = count(p->rchild);
    return n1+n2;
}
}

```

- 计算深度

```

//操作型
void depth(BTNode* bt,int& n,int& max)
{
    if(bt==null)
    {
        return;
    }
    n++;
    if(n>max)
    {
        max = n;
    }

    depth(bt->lchild);
    depth(bt->rchild);

    n--;
}

```

- 题目：

- $(a-(b+c))*(d/e)$ 存储在二叉树，遍历求值

```

int CalValue(BTNode* root)
{
    if(root == null)
    {
        return 0;
    }
}

```

```

// 当前节点类型为叶节点，因此直接返回值
if(root->lchild==null && root->rchild==null)
{
    return root->data;
}

// 当前节点类型为分支节点，按照符号进行运算
if(root->lchild!=null && root->rchild!=null)
{
    int n1 = CalValue(root->lchild);
    int n2 = CalValue(root->rchild);

    // 返回n1和n2按照当前符号的运算结果
    return op(n1,n2,p->data);
}
}

```

- 题目：

7、判断两个二叉树是否相似（指都为空或者都只有一个根节点，或者左右子树都相似）

- 总结：

- 用两根指针，对两棵树同步进行遍历，判断它们结构是否相等；

```

int fun(BTNode* p1,BTNode* p2)
{
    // 两节点都会空，相似
    if(p1==null&&p2==null)
    {
        return 1;
    }

    // 两个结点只有一个为空，不相似
    if(p1==null||p2==null)
    {
        return 0;
    }

    // 两个节点都不为空时，进入到它下一层、继续比较
    int left = fun(p1->lchild,p2->lchild);
    int right = fun(p1->rchild,p2->rchild);
    return (left&&right);
}

```

- 题目：

8、把二叉树所有节点左右子树交换

- 总结:
 - 采用后根遍历，先将左右子树交换，确保左右子树已经符合交换定义后、再对整个树进行交换

```
void Invert(BTNode* p)
{
    if(p==null)
    {
        return;
    }
    else
    {
        // 先将子树的子树倒转，最后交换自己的子树
        Invert(p->lchild);
        Invert(p->rchild);

        BTNode* temp = p->lchild;
        p->lchild = p->rchild;
        p->rchild = temp;
    }
}
```

查找

- 题目:

9、查找二叉树中 data 域等于 key 的结点是否存在，若存在，将 q 指向它，否则 q 为空
- 总结:

```
void Find(BTNode* bt,int x,LNode* q)
{
    if(bt==null)
    {
        return;
    }
    else
    {
        if(bt->data == x)
        {
            q = p;
            return;
        }
        Find(p->lchild,data,q);
        Find(p->rchild,data,q);
    }
}
```

- 题目:

• 11、求二叉树中值为 x 的层号

- 总结:

```
void Find(BTNode* bt,int x,int& n)
{
    if(bt==null)
    {
        return;
    }
    n++;
    if(bt->data == x)
    {
        cout<<n;
        return;
    }
    else
    {
        Find(p->lchild,x,n);
        Find(p->rchild,x,n);
    }
}
```

叶结点操作

- 题目:

• 13、利用结点的右孩子指针将一个二叉树的叶子节点从左向右连接成一个单链表（head 指向第一个，tail 指向最后一个）

- 总结:

- 将 tail 的右孩子指向新发现的叶节点;

```
void fun(BTNode* p,BTNode* head,BTNode* tail)
{
    if(p==null)
    {
        return;
    }
    else
    {
        // 如果为叶节点
        if(p->lchild==null&&p->rchild==null)
        {
            // 如果是第一次到达叶节点，则此结点为单链表的头
            if(head==null)
            {
```

```

        head = p;
        tail = p;
    }
    else
    {
        tail->rchild = p;
        tail = p;
    }
}
else
{
    fun(p->lchild,head,tail);
    fun(p->rchild,head,tail);
}
}
}

```

- 题目:

14、输出根节点到每个叶子结点的路径

- 总结:

- 利用一个栈存储当前路径上的所有结点，进入下一层结点后，入栈。离开下一层结点、返回本层时，出栈；
- 如果左右孩子都为空，打印当前栈中所有值；

```

void PrintPath(BTNode* p,char* pathStack,int& top)
{
    if(p == null)
    {
        return;
    }

    // 将当前元素压入栈
    pathStack[++top] = p->data;

    // 如果是叶节点，输出栈元素
    if(p->lchild==null&&p->rchild==null)
    {
        for(int i = 0;i<=top;i++)
        {
            std::cout<<pathStack[i];
        }
    }
    else
    {
        PrintPath(p->lchild,pathStack,top);
        PrintPath(p->rchild,pathStack,top);
        --top;
    }
}

```

```
}  
}
```

- 题目:

29、计算二叉树的带权路径长度（叶子节点）

- 总结:

- 当左右为空时，当前结点为叶子节点，将深度和权值相乘，返回上一层；

```
int GetWPL(BTNode* bt,int& deep)  
{  
    if(bt==null)  
    {  
        return 0;  
    }  
    deep++;  
    if(bt->lchild==null&&bt->rchild==null)  
    {  
        return deep*(bt->data);  
    }  
    int A = GetWPL(bt->lchild,deep);  
    int B = GetWPL(bt->rchild,deep);  
    return A+B;  
}
```

创建

- 题目:

17、二叉树以顺序方式存在于数组 A 的中，设计算法以二叉链表表示

- 总结:

- 递归的创建

```
// A为数组，i为当前结点位置，n为数组长  
BiTree create(int A[],int i,int n)  
{  
    if(i<n)  
    {  
        BTNode* root = new BTNode;  
        root->data = A[i];  
        // 当前结点的左元素在序列中的下标为 i*2+1  
        root->lchild = create(A[],i*2+1,n);  
  
        // 当前结点的右元素在序列中的下标为 i*2+2  
        root->rchild = create(A[],i*2+2,n);  
  
        return root;  
    }  
}
```

```
return null;
}
```

一维数组与二叉树

- 题目:

- 15、已知满二叉树先序序列存在于数组中，设计算法将其变成后序序列

- 总结:

- 由满二叉树性质可知:
- 当前先序序列的第一个结点为根节点;
- 当前先序序列的 $L1+1$ 到 $(L1+R1+1)/2$ 为左子树;
- 当前先序序列的 $(L1+R1+1)/2+1$ 到 $R1$ 为右子树;

```
//pre存储先序序列，post存储后序序列
// L1、R1分别指向先序序列的第一个以及最后一个元素
// L2、R2分别指向后续序列的第一个以及最后一个元素
void change(char pre[],int L1,int R1,char post[],int L2,int R2)
{
    if(L1 <= R1 && L2 <= R2)
    {
        // 把当前先序序列的第一个结点，即二叉树的根节点，放到后序序列中根所处的位置
        post[R2]= pre[L1];

        // 对当前结点的左子树递归此过程
        change(pre, L1+1,(L1+R1+1)/2, post, L2,(L2+R2-1)/2);

        // 对当前结点的右子树递归此过程
        change(pre, (L1+R1+1)/2+1,R1,post, (L2+R2-1)/2+1,R2-1);
    }
}
```

- 题目:

- 16、先序与中序遍历分别存在两个一维数组 A, B 中，试着建立二叉链表

- 总结:

- 第一步：取出当前先序序列第一个值，用其值建立新结点
- 第二步：在中序序列中找到对应值的下标
- 第三步：分别对中序序列的左边、右边递归此操作
 - 当前 i 的左边: $A1+1$ 到 $i + Ar-A1$
 - 当前 i 的右边: $i+Ar-A1+1$ 到 Ar

```
BiTree create(char A[],char B[],int A1,int Ar,int B1,int Br)
{
    BTreeNode* root = new BTreeNode;
```

```

// 当前A序列的第一个元素为二叉树的根结点
root->data = A[Al];

// 根据根节点的值，在中序序列中找到根节点的下标
int i = 0;
for(; i<Br; i++)
{
    if(B[i]==root->data)
    {
        break;
    }
}

// 当前i为B序列中、根节点元素的位置
// B系列的左子树不为空
if(i>Bl)
{
    root->lchild = create(A[], B[], Al+1, i+Ar-Al, Bl, i-1);
}
else
{
    root->lchild = null;
}

// B系列的右子树不为空
if(i<Br)
{
    root->rchild = create(A[], B[], i+Ar-Al+1, Ar, i+1, Br);
}
else
{
    root->rchild = null;
}

return root;
}

```

层次遍历

```

void fun(BTNode* bt)
{
    BTNode* p = bt;
    BTNode* queue[maxsize];

    int front = 0;
    int rear = 0;

    if(bt!=null)
    {
        queue[rear++] = p;
    }
}

```

```

while(rear !=front)
{
    // 将当前队列的队头出队，并分析其是否有左右结点
    p = queue[front++];
    std::cout<<p->data;

    // 如果当前p的左右有子节点，将其入队
    if(p->lchild!=null)
    {
        queue[rear++] = p->lchild;
    }
    if(p->rchild!=null)
    {
        queue[rear++] = p->rchild;
    }
}
}
}

```

• 题目：

28、判断二叉树是否为完全二叉树

• 总结：

- 完全二叉树如果按照层次遍历，将所有结点存储在数组中，则符合二叉树在数组中的存储规则，并且因为是完全二叉树，因此在数据中应当在最后一个结点之前、无空缺位置，即位置全部填满；
- 所以可以在出现空结点时判断，当前队列后背是否都是空结点。如果后背还有结点，则肯定不是完全二叉树。如果后背都是空结点，则此时此结点是最右下角的结点；

```

bool IsFun(BTNode* bt)
{
    BTNode* queue[maxsize];
    int rear = 0;
    int front = 0;

    if(bt==true)
    {
        return true;
    }
    queue[rear++] = bt;

    while(rear!=front)
    {
        bt = queue[front++];

        if(bt!=null)
        {
            queue[rear++] = bt->lchild;

```

```

        queue[rear++] = bt->rchild;
    }
    else // 当前为一个空结点，一个个出队、检查队列后背是否都是空结点。如果后背还有结
    点，则肯定不是完全二叉树
    {
        while(front!=rear)
        {
            // 从当前空结点继续
            bt = queue[front++];

            if(bt!=null)
            {
                return false;
            }
        }
    }

    return true;
}

```

线索化

- 结构体

```

typedef struct TNode
{
    char data;
    struct TNode* lchild,rchild;
    int ltag = 0;
    int rtag = 0;
}TNode,iTree*;

```

- 中序线索化

```

void Fun(TNode* &p,TNode* &pre)
{
    if(p!=null)
    {
        Fun(p->lchild,pre);

        if(p->lchild == null)
        {
            p->ltage = 1;
            p->lchild = pre;
        }
        if(p->rchild==null&&pre!=null)
        {
            pre->rtag = 1;
            pre->rchild = p;
        }
    }
}

```

```

    }
    pre = p;

    Fun(p->rchild,pre);
}
}

```

查找

- 题目:

1、顺序表递增有序，设计算法在最少的时间
内查找值为 x 的元素。若找到，则将其于后继
元素位置交换，否则按照递增顺序插入顺序表

- 总结:

- 第一步：二分查找到值为 x 的元素的位置
- 第二步：如果找到，交换
- 第三步：如果 `low < high`，则表示没找到，后移

```

void Insert(Sqlist& list,int x)
{
    int low = 0;
    int high = list.lenght-1;
    int mid;

    while(low<=high)
    {
        mid = (low+high)/2;
        if(list.data[mid]==x)
        {
            break;
        }
        if(list.data[mid]<x)
        {
            low = mid+1;
        }
        else
        {
            high = mid-1;
        }
    }

    // 如果当前位置的元素就等于当前要找的值，并且不是最后一个元素
    if(list.data[mid] == x && mid!=list.length-1)
    {
        // 交换前后元素
        int temp = x;
        list.data[mid] = list.data[mid+1];

```

```

        list.data[mid+1] = temp;
        return;
    }

    if(low>high)
    {
        int i
        for(i = list.length-1;i>high;--i)
        {
            list.data[i+1] = list[i];
        }
        list.data[i+1] = x;
        list.length++;
    }
}

```

- 题目:

5、寻找二叉排序树中最大值和最小值

- 总结:

- 最大值: 走到右下角即可

```

int GetMax(BTNode* bt)
{
    if(bt==null)
    {
        return 0;
    }
    while(bt->rchild!=null)
    {
        bt = bt->rchild;
    }
    return bt->data;
}

```

图

结构体

```

typedef struct AGraph
{
    int vertexNum = 0;
    int edgeNum = 0;
    VNode adjlist[maxsize];
}AGraph;

typedef struct VNode
{
    char data;
}

```

```

    ArcNode* firstArc;
}VNode;

typedef struct ArcNode
{
    int adjVertex;
    struct ArcNode* NextArc;
}ArcNode;

typedef struct MGraph
{
    int vertexNum = 0;
    int arcNum = 0;
    int Edge[maxsize][maxsize];
    char vertex[maxsize];
}MGraph;

```

基本操作

- 邻接表（无向图）的插入

```

AGraph* Insert(int vertexNum,int edgeNum)
{
    AGraph* g = new AGraph*;
    g->vertexNum = vertexNum;
    g->edgeNum = edgeNum;
    for(int i = 0;i<g->vertexNum;i++)
    {
        g->adjlist[i].firstArc = null
    }

    for(int i =0;i<edgeNum;i++)
    {
        int v1,v2;
        cin>>v1;
        cin>>v2;
        ArcNode* newNode1 = new ArcNode*;
        newNode1.adjVertex = v2;
        newNode1->firstArc = g->adjlist[v1]->firstArc;
        g->adjlist[i]->firstArc = newNode1;

        ArcNode* newNode2 = new ArcNode*;
        newNode2.adjVertex = v1;
        newNode2.NextArc = g->adjlist[v2].firstArc;
        g->adjlist[v2].firstArc = newNode2;
    }
    return G;
}

```

BFS

- BFS

```
void BFS(AGraph* g,int v,int visit[])
{
    int queue[maxsize];
    int rear = 0;
    int front = 0;
    for(int i = 0;i<G->vertexNum;i++)
    {
        visit[i] = 0;
    }

    queue[++rear] = v;
    visit[v] = 1;
    cout<<v;

    while(rear!=front)
    {
        v = queue[++front];
        ArcNode* p = g->adjlist[v].firstArc;
        while(p!=null)
        {
            if(visit[p->adjVertex]==0)
            {
                visit[p->adjVertex]=1;
                cout<<p->adjVertex;
                queue[++rear] = p->adjVertex;
            }
            p = p->NextArc;
        }
    }
}
```

- 利用 BFS 求无向图最短路径：将 BFS 中的 cout 语句，改成了更改距离

```
if(visit[p->adjVertex]==0)
{
    visit[p->adjVertex] = 1;
    // 将和当前v结点相连的结点的距离，在v的基础商+1
    dist[p->adjVertex] = dist[v] + 1;
    queue[++rear] = p->adjVertex;
}
```

DFS

- DFS

```
void DFS(AGraph* g,int v,int visit[])
{
    cout<<v;
```

```

visit[v] = 1;
ArcNode* p = g->adjlist[v].firstArc;

while(p!=null)
{
    if(visit[p.adjVertex]==0)
    {
        DFS(g,p.adjVertex,visit[]);
    }
    p = p->nextArc;
}
}

```

- 题目:

5、图采用邻接表存储，设计算法判断 i 和 j 结点之间是否有路径（以下全是邻接表存储）

- 总结:

- 以 i 结点为起始点，进行一次 DFS。如果 j 结点和 i 之间有路径，则 j 结点肯定会被访问，即 `visit[j]==1`，如果 `visit[j]==0`，则表示无路径

```

bool IsPath(AGraph* G,int i,int j)
{
    int visit[maxsize];

    for(int i =0;i<G->vertexNum;i++)
    {
        visit[i] = 0;
    }

    DFS(G,i,visit[]);
    if(visit[j]==1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

```

- 题目:

6、设计算法判断无向图是否是一棵树

- 总结:

- 树的两个条件:
 - (1) 是一个连通图
 - (2) 结点数 - 边数 = 1

- 因此，可以用一个 vn 来计算当前图 DFS 遍历到的节点个数，如果等于结点数，则表示为连通图；
- 题目：
 - **10、求无向图的连通分量个数：**
- 总结：
 - DFS 的次数即为连通分量个数；

```
int Fun()
{
    int count = 0;
    int visit[maxsize];

    for(int i =0;i<G->vertexNum;i++)
    {
        visit[i] = 0;
    }

    for(int i =0;i<G->vertexNum;i++)
    {
        if(visit[i]==0)
        {
            DFS(G,v,visit[]);
            count++;
        }
    }

    return count;
}
```

排序

直接插入

```
void Insert(int R[],int n)
{
    for(int i =2;i<=n;++i)
    {
        R[0] = R[i];
        for(int j = i-1;R[0]<R[j];--j)
        {
            R[j+1] = R[j];
        }

        R[j+1] = R[0];
    }
}
```

快速排序

```
int part(int A[],int low,int high)
{
    int pivot = A[low];

    while(low<high)
    {
        while(A[high]>pivot&&low<high)
        {
            high--;
        }

        A[low] = A[high];

        while(A[low]<pivot&&low<high)
        {
            low++;
        }
        A[high] = A[low];
    }
    A[low] = pivot;
    return low;
}

void QuickSort(int A[],int low,int high)
{
    if(low<high)
    {
        int pos = part(A,low,high);
        QuickSort(A,low,pos-1);
        QuickSort(A,pos+1,high);
    }
}
```

获取笔记