核心附加算法
对一颗以孩子一兄弟链表表示的一般树统计其叶子的个数

```
#include
using namespace std;
struct Node{   左孩子，右兄弟表示法
    struct Node child,sibling;
};
typedef Node Tree;
int counts=0;
void NumberOfLeaf(Tree root){ recursive
    if(root==NULL) return;
    if(root-child==NULL) counts++;
    NumberOfLeaf(root-child);
    NumberOfLeaf(root-sibling);
}
```

将单链表 L1 拆开两个链表，其中以 L1 为头的链表保持原来向后链接，
另一个链表头为 L2 其连接方向与 L1 相反，L1 包含奇数序号节点，L2 包含偶数序号节点

```
#include
#include
using namespace std;
struct Node{   单链表
    int data;
    struct Node next;
};
typedef Node List;
void SplitList(List head,Node &head1,Node &head2){
    vector v;
    if(head==NULL) return;
    head=head-next;
    while(head){将头结点以外的所有节点顺序放进去
        v.push_back(head);
        head=head-next;
    }
    head1=new Node,head2=new Node;
    Node tmp1=head1,tmp2=head2;
    for(int i=1;inext=v[i];
        tmp1=tmp1-next;
    }
    tmp1-next=NULL;

    int i=v.size();
    if(i%2==0) i--;
```

```
        i--;
        for( ;i=0;i=i-2){
                tmp2-next=v[i];
                tmp2=tmp2-next;
        }
        tmp2-next=NULL;

}
```

假设二叉树以二叉链表存储，设计一个算法判断一棵二叉树是否为完全二叉树

```
typedef struct BiTNode
{
        char data;
        struct BiTNode lchild,rchild;
}BiTNode,BiTree;
BiTree T;

void    Judge(BiTree T)
{       int     f=0,r=0,s=0;
        BiTree p,q[MAXQSIZE];
        q[r++]=T;根结点入队
        while(fr)
        {
                p=q[f++];根结点出队
                if(p)
                {
                        q[r++]=p-lchild;左孩子入队
                        q[r++]=p-rchild;右孩子入队
                }
                else
                {
                        while(fr)
                        {
                                p=q[f++];
                                if(p)s++;
                        }
                }
        }
        if(s!=0)printf(二叉树不是完全二叉树 n);
        if(s==0)printf(二叉树是完全二叉树 n);
}
```

设计一个含 n 个整数的线性表。设计一个在时空两方面尽可能高效的算法，将表中数据从小到大排序。
写排序算法

已知二叉树 T 采用二叉链表存储结构，每个节点有三个字段，内容、左孩子指针、右孩子指针。
请设计一个计算该二叉树所有叶子节点数目的算法。

```
#include
using namespace std;
struct Node{    二叉树结点
    int data;
    struct Node lchild,rchild;
};
typedef Node Tree;
int counts=0;
void NumberOfLeaf(Tree root){ recursive
    if(root==NULL) return;
    if(root-lchild==NULL&&root-rchild==NULL) counts++;
    NumberOfLeaf(root-lchild);
    NumberOfLeaf(root-rchild);
}
```

将普通链表中值最小的结点提到最前，要求不能申请新的节点

```
void MoveMinToFirst(LinkList& L)
{
    LNode pre = L, p = L-next;
    LNode premin = pre, min = p;
    while (p)
    {
        if (min-data    p-data)
        {
            premin = pre;
            min = p;
        }
        pre = p;
        p = p-next;
    }
    premin-next = min-next;
    min-next = L-next;
    L-next = min;
}
```

试写出一个递归函数，判断两棵树是否相等

```
#include
using namespace std;
struct Node{   二叉树结点
      int data;
      struct Node lchild,rchild;
};
typedef Node Tree;
bool IsTwoTreesEqual(Tree t1,Tree t2){
      if(!t1 && !t2)
            return true;
      if(IsTwoTreesEqual(t1-lchild,t2-lchild) && IsTwoTreesEqual(t1-rchild,t2-rchild)){
            if(t1-data==t2-data)
                  return true;
      }
      return false;
}
```

写算法求二叉树的高度

```
int height(struct node node)
{
    if (node==NULL)
          return 0;
    else
    {
          计算左子树的高度和右子树的高度
          int lHeight = height(node-left);
          int rHeight = height(node-right);
          返回二者较大者加 1
          if (lHeight    rHeight)
                return(lHeight+1);
          else return(rHeight+1);
    }
}
```

写算法，对无头结点的单链表中的元素逆置（不允许申请新的节点空间）

```
void InverseElements(List l)
{
    if(!l!l-next) return ;
    List tmp=l-next;
    List tmp2=tmp-next;
    tmp-next=NULL;
    while(tmp2){
        l-next=tmp2;
        tmp2=tmp2-next;
        l-next-next=tmp;
        tmp=l-next;
    }
}
```

写算法层次顺序遍历二叉树

```
struct Node {
    int data;
    struct Node left, right;
};
void printLevelOrder(Node root) {
    if (root == NULL)    return;
     创建一个空队列
    queueNode    q;
    q.push(root);

    while (q.empty() == false) {
         遍历当前节点
        Node node = q.front();
        cout    node-data     ;
        q.pop();

         左子节点入队
        if (node-left != NULL)
            q.push(node-left);

         右子节点入队
        if (node-right != NULL)
            q.push(node-right);
    }
}
```

写算法对双向循环链表按访问频度自高到底进行排序

```
#includebitsstdc++.h
using namespace std;
typedef struct dnode{
	int data;
	int freq;
	struct dnode next;
	struct dnode prior;
}dinklist;
dinklist h;
void locatenode(dinklist &h,int x)
{
	dinklist p;
	p=h-next;
	int i=0;
	while(p!=NULL&&p-data!=x)
	{
		p=p-next;
		i++;
	}
	p-freq++;
	dinklist q,pre;
	p=h-next-next;
	h-next-next=NULL;
	while(p!=NULL)
	{
		q=p-next;
		pre=h;
		while(pre-next!=NULL&&pre-next-freqp-freq)
		{
			pre = pre-next;
		}
		p-next=pre-next;
		if(pre-next!=NULL)
		{
			pre-next-prior=p;
		}
		pre-next=p;
		p-prior=pre;
		p=q;
	}
}
```

简述折半插入排序的思想、稳定性和时间复杂度，并写出算法

```
void insertSort(int array[], int n){
    int temp;
    for(int i = 1; i    n; i++){
        int low = 0;
        int hight = i-1;
        temp = array[i];

        while(hight=low){
            int mid = ( low + hight )    2;
            if (array[mid]    temp){
                hight = mid - 1;
            }else{
                low = mid + 1;
            }
        }

        for (int j = i-1; j    hight; j--) {
            array[j+1] = array[j];
        }

        array[hight+1] = temp;
    }
}
```

写算法将二叉树 bt 中每一个结点的左右子树互换

```
typedef struct BiTNode {
    TElemType data;
    BiTNode    lchild, rchild;
} BiTNode, BiTree;
void Exchange(BiTree &bt)
{
    BiTree temp;
    if(bt){
        temp = bt - lchild;
        bt - lchild = bt - rchild;
        bt - rchild = temp;
        Exchange(bt - lchild);
        Exchange(bt - rchild);
    }
}
```

判断链表前 n 个字符是否中心对称

```
int judge(struct node head,int len)
{
	struct node top,p1,p2;
	top = NULL;
	p1 = head-next;
	for(int i = 0 ; i    len2 ; i++)
	{
		p2 = (struct node )malloc(LEN);
		p2-cc = p1-cc;
		p2-next = top;
		top = p2;
		p1 = p1-next;
	}
	if(len%2 == 1)
		p1 = p1-next;
	p2 = top;
	for(i = 0 ; i    len2 ; i++)
	{
		if(p2-cc != p1-cc)
			break;
		top = p2-next;
		p1 = p1-next;
		p2 = top;
	}
	if(!top)
		return 1;
	else
		return 0;
}
```

设计一个算法判断二叉树是否为二叉排序树

```
int prev = MIN;
int flag = true;
bool InOrderTraverse(BiTree T)
{
	if (T-lchild != NULL && flag)
		InOrderTraverse(T-lchild);
	if (T-dataprev)
		flag = false;
	prev = T-data;
	if (T-rchild != NULL && flag)
		InOrderTraverse(T-rchild);
	return flag;
}
```

利用栈的特性，将 a 进制数转换为 b 进制

a-10-b（数组改栈）

```
int toShi(int n,char a[]){
	int len = strlen(a);
	int i,ans=0;
	for(i=0;ilen;i++){
		if(a[i]='0'&&a[i]='9'){
			ans=ansn+a[i]-'0';
		}
		else{
			ans = ansn+a[i]-'A'+10;
		}
	}
	return ans;
}
void toB(int ans,int b){
	char arr[100];
	int i,j=0,tmp;
	while(ans!=0){
		tmp = ans%b;
		if(tmp=10){
			arr[j++] = tmp-10+'A';
		}else{
			arr[j++] = tmp+'0';
		}
		ans = ansb;
	}
	for(i=j-1;i=0;i--){
		printf(%c,arr[i]);
	}
}
```

给定一棵树二叉排序树，从中找到比值 key 小的所有值，并按照从大到小的方式输出

```
void InOrder(BSTree root,int k){
	if(!root)
		return ;
	InOrder(root-rchild,k);
	if(root-data=k){
		visit(root);
	}
	InOrder(root-lchild,k);

}
```

奇数位置尾插法 偶数位置头插法
类似之前 然后使用尾差法和头插法

数据结构的基本概念，熟悉评价算法的标准
线性表
算法 2.1 求线性表 LA 和 LB 的并集

```
void Union(List &La,List Lb) {
    La_len = ListLength(La);
    Lb_len = ListLength(Lb);
    for(i=1;iLb_len;i++) {
        GetElem(Lb,i,e); 取 Lb 中第 i 个数据元素赋给 e
        if(!LocateElem(La,e,equal)){
            La 中不存在和 e 相同的数据元素，则插入之
            ListInsert(La,++La_len.len;e);
        }
    }
}
```

算法 2.2 归并 La 和 Lb 得到新的线性表 Lc，Lc 的数据元素也是按非递减排列

```
void MergeList(List La,List Lb,List &Lc) {
    La 和 Lb 中数据元素按值非递增排列
    InitList(Lc);
    i =j = 1;k=0;
    La_len = ListLength(La);
    Lb_len = ListLength(Lb);
    while((i=La_len) && (j=Lb_len)) {
        GetElem(La,i,ai);
        GetElem(Lb,i,bi);
        if(ai=bi) {
            ListInsert(Lc,++k,ai);++i;
        } else {
            ListInsert(Lc,++k,bi);++j;
        }
    }
    while(i=La_len) {
        GetElem(La,i,ai);
        ListInsert(Lc,++k,ai);
    }
    while(j=Lb_len) {
        GetElem(Lb,i,bi);
        Listinsert(Lc,++k,bj);
    }
}
```

------- 顺序表 --------
线性表的存储结构
```
#define List_Init_Size 100  线性表存储空间  初始化分配量
#define ListIncrement    10   线性表存储空间  分配增量
typedef struct {
    ElemType    elem;   存储空间基址
    int length;     当前长度
    int listsize;  当前分配的存储容量
}SqList;
```

算法 2.3 构造一个空的线性表 L
```
Status InitList_Sq(SqList &L) {
    L.elem = (ElemType )malloc(List_Init_Size    sizeof(ElemType));
    if(! L.elem) {
        exit(OVERFLOW);
    }
    L.length = 0;
    L.listsize = List_Init_Size;
    return OK;
}
```

算法 2.4 在顺序表 L 中第 i 个位置之前插入新的元素 e
```
Status ListInsert_Sq(SqList &L,int i,ElemType e) {
    if(i    1    i    L.length+1) {
        return ERROR;
    }
    if(L.length = L.listsize) {
        newbase = (ElemType )realloc(L.elem,(L.listsize + ListIncrement)sizeof(ElemType));
        if(!newbase) {
            exit(OVERFLOW);
        }
        L.elem = newbase;
        L.listsize += ListIncrement;
    }
    q = &(L.elem[i-1]);
    for(p = &(L.elem[L.length-1]) ; p = q ; --p) {
        (p + 1) = p;  插入位置及之后的元素右移
    }
    q = e;
    ++L.length;
    return OK;
}
```

算法 2.5 在顺序表 L 中删除第 i 个元素，并用 e 返回其值

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e) {
    if((i    1)    (i    L.length)) {
        return ERROR;
    }
    p = &(L.elem[i-1]); p 为被删除元素的位置
    e = p;        被删除元素的值赋给 e
    q = L.elem + L.length - 1;    表尾元素的位置
    for(++p ; p=q ; ++p) {
        (p-1) = p;  被删除元素之后的元素左移
    }
    --L.length;
    return OK;
}
```

算法 2.6 在顺序线性表 L 中查找第 1 个值与 e 满足 compare()的元素的位序

```
int LocateElem_Sq(SqList L,ElemType e,Status(compare)(ElemType,ElemType)) {
    i = 1; i 的初值为第 1 个元素的位序
    p = L.elem;    p 的初值为第 1 个元素存储位置
    while(i=L.length && !(compare(p++ , e))) {
        ++i;
    }
    if(i = L.length) {
        return i;
    } else {
        return 0;
    }
}
```

算法 2.7 归并 La 和 Lb 得到新的线性顺序表 Lc，Lc 元素也按值非递减排列

```
void MergeList_Sq(SqList La, SqList Lb, SqList &Lc) {
    pa = La.elem;
    pb = Lb.elem;
    Lc.listsize = Lc.length = La.length + Lb.length;
    pc = Lc.elem = (ElemType )malloc(Lc.listsizesizeof(ElemType));
    if(!Lc.elem) {
        exit(OVERFLOW);
    }
    pa_last = La.elem + La.length - 1;
    pb_last = Lb.elem + Lb.length - 1;
    while(pa    = pa_last && pb = pb_last) {
        if(pa = pb) {
            pc++ = pa++;
        } else {
```

```
                pc++ = pb++;
            }
        }
        while(pa = pa_last) {
            pc++ = pa++;
        }
        while(pb = pb_last) {
            pc++ = pb++;
        }
}
```

------ 链式表 --------

算法 2.8 当第 i 个元素存在时，其值赋给 e

```
Status GetElem_L(LinkList L, int i,ElemType &e) {
    p = L-next;
    j = 1;
    while(p && j    i) {
        p = p - next;
        j++;
    }
    if(!p    j    i) {
        return ERROR;
    }
    e = p - data;
    return OK;
}
```

算法 2.9 在带头借点的单链线性表 L 中第 i 个位置之前插入元素 e

```
Status ListInsert_L(LinkList &L, int i, ElemType e) {
    p = L;
    j = 0;
    while(p && j    i-1) {
        p = p - next;  寻找第 i-1 个结点
        j++;
    }
    if(!p    j    i + 1) {    i 小于 1 或大于表长加 1
        return ERROR;
    }
    s = (LinkList)malloc(sizeof(LNode));  生成新结点
    s - data = e;
    s - next = p - next;  插入 L
    p - next = s;
    return OK;
}
```

算法 2.10 在带头结点的单链线性表 L 中，删除第 i 个元素，并由 e 返回其值

```
Status ListDelete_L(LinkList &L, int i, ElemType &e) {
    p = L;
    j = 0;
    while(p - next && j   i-1) { 寻找第 i 个结点，并令 p 指向其前趋
        p = p - next;
        ++j;
    }
    if(!(p - next)   j   i-1) { 删除位置不合理
        return ERROR;
    }
    q = p - next;
    p - next = q - next;   删除并释放结点
    e = q - data;
    free(q);
    return OK;
}
```

算法 2.11 逆位序输入 n 个元素的值，建立带投结点的单链线性表 L

```
void CreateList L(LinkList &L,int n) {
    L = (LinkList)malloc(sizeof(LNode));
    L - next = NULL; 先建立一个头结点的单链表
    for(i=n;i0;--i) {
        p = (LinkList)malloc(sizeof(LNode));
        scanf(& p- data);
        p - next = L - next;
        L - next = p;
    }
}
```

算法 2.12 归并 La 和 Lb 得到新的单链表 Lc

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc) {
    pa = La - next;
    pb = Lb - next;
    Lc = pc = La;     用 La 的头结点作为 Lc 的头结点
    while(pa && pb) {
        if(pa - data = pb - data) {
            pc - next = pa;
            pc = pa;
            pa = pa - next;
        } else {
            pc - next =pb;
            pc = pb;
```

```
                pb = pb - next;
            }
        }
        pc - next = pa    pa    pb;
        free(Lb);
}
```

----- 线性表的静态单链表 ------

用游标（指示器 cur）代替指针指示结点在数组中的相对位置

```
#define MaxSize 1000
typedef struct {
        ElemType data;
        int cur;
}component,SLinkList[MaxSize];
```

算法 2.13 在静态单链线性表 L 中查找第 1 个值为 e 的元素

```
int LocateElem_SL(SLinkList S,ElemType e) {
        i = S[0].cur;
        while(i && S[i].data != e) {
                i = S[i].cur;
        }
        return i;
}
```

算法 2.14 将一维数组 space 中各分量链成一个备用链表，space[0].cur 为头指针。

```
void InitSpace_SL(SLinkList &space) {
        for(i = 0;i    MaxSize;i++) {
                space[i].cur = i+1;
        }
        space[MaxSize-1].cur = 0;
}
```

算法 2.15 若备用空间链表非空，则返回分配的结点下标，否则返回 0

```
int Malloc_SL(SLinkList &space) {
        i = space[0].cur;
        if(space[0].cur){
                space[0].cur = space[i].cur;
        }
        return i;
}
```

算法 2.16 将下标为 k 的空闲结点回收到备用链表

```
void Free_SL(SLinkList &space, int k) {
        space[k].cur = space[0].cur;
```

```
    space[0].cur = k;
}
```

算法 2.17（A-B）∪ (B-A)

```
void Difference(SLinkList &space, int &s) {
    InitSpace_SL(space);
    S = Malloc_SL(space);
    r = S;
    scanf(m, n);
    for(j = 1;j = m;++j) {
        i = Malloc_SL(space);
        scanf(space[i].data);
        space[r].cur = i;
        r = i;
    }
    space[r].cur = 0;
    for(j = 1;j = n;++j) {
        scanf(b);
        p = S;
        k = space[S].cur;
        while(k != space[r].cur && space[k].data != b) {
            p = k;
            k = space[k].cur;
        }
        if(k == space[r].cur) {
            i = Malloc_SL(space);
            space[i].data = b;
            space[i].cur = space[r].cur;
            space[r].cur = i;
        } else {
            space[p].cur = space[k].cur;
            Free_SL(space,k);
            if(r == k) {
                r = p;
            }
        }
    }
}
```

---- 双向链表 ------

```
 typedef struct DuLNode {
      ElemType data;
      struct DuLNode prior;
```

```
    struct DuLNode next;
}DuLNode,DuLinkList;
```

算法 2.18 在带头结点的双链循环线性表 L 中第 i 个位置之前插入元素 e

```
Status ListInsert_Dul(DuLinkList &L, int i, ElemType e) {
    if(!(p = GetElemP_DuL(L, i))) {
        return ERROR;
    }
    if(!(s = (DuLinkList)malloc(sizeof(DuLNode)))) {
        return ERROR;
    }
    s - data = e;
    s - prior = p - prior;  先接前，再接后
    p - prior - next = s;
    s - next = p;
    p - prior = s;
    return OK;
}
```

算法 2.19 删除带头结点的双链循环线性表 L 的第 i 个元素，i 的合法值为 1=i=表长

```
Status ListDelete_DuL() {
    if(!(p = GetElemP_DuL(L, i))) {
        return ERROR;
    }
    e = p - data;
    p - prior - next = p - next;
    p - next - prior = p - prior;
    free(p);
    return OK;
}
```

算法 2.20 在带头结点的单链线性表 L 的第 i 个元素之前插入元素 e

```
Status ListInsert_L(LinkList &L, int i, ElemType e) {
    if(!LocatePos(L, i - 1, h)) { i 值不合法
        return ERROR;
    }
    if(!MakeNode(s, e)) {
        return ERROR;
    }
    InsFirst(h, s);对于从第 i 个结点开始的链表，第 i-1 个
    return OK;
}
```

算法 2.21 归并 La 和 Lb 得到新的线性链表 Lc

```
Status MergeList_L(LinkList &La,LinkList &Lb, LinkList &Lc) {
    if(!InitList(Lc)) {存储空间分配失败
        return ERROR;
    }
    ha = GetHead(La);
    hb = GetHead(Lb);
    pa = NextPos(La,ha);
    pb = NextPos(Lb,hb);
    while(pa && pb) {
        a = GetCurElem(pa);
        b = GetCurElem(pb);
        if((compare)(a, b) = 0) {
            DelFirst(ha, q);
            Append(Lc, q);
            pa = NextPos(La, ha);
        } else {
            DelFirst(hb, q);
            Append(Lc, q);
            pb = NextPos(Lb, hb);
        }
    }
    if(pa) {
        Append(Lc, pa);
    } else {
        Append(Lc, pb)
    }
    FreeNode(ha);
    FreeNode(hb);
    return OK;
}
```

抽象数据类型 polynomial 的实现

```
typedef struct {
    float coef;    系数
    int expn;       指数
}term, ElemType; term 用于本 ADT，Elemtype 为 LinkList 的数据对象名
```

算法 2.22 输入 m 项的系数和指数，建立表示一元多项式的有序链表 P

```
void CreatePolyn(polynomail &P, int m) {
    InitList(P);
    h = GetHead(P);
    e.coef = 0.0;
    e.expn = -1;
    SetCurElem(h,e);
```

```
    for(i = 1 ; i = m ; i++) {
        scanf(e.coef,e.expn);
        if(!LocateElem(P,e,q,(cmp))) { 当前链表中不存在该指数项
            if(MakeNode(s, e)) {
                InsFirst(q, s); 生成结点并插入链表
            }
        }
    }
}
```

算法 2.23 多项式加法：Pa = Pa + Pb 利用两个多项式结点构成“和多项式”

```
void AddPolyn(polynomail &Pa, polynomail &Pb) {
    ha = GetHead(Pa);
    hb = GetHead(Pb);
    qa = NextPos(Pa,ha);
    qb = NextPos(Pb,hb);
    while(qa && qb) {
        a = GetCurElem(qa);
        b = GetCurElem(qb);
        switch(cmp(a, b)) {
            case -1 多项式
                ha = qa;
                qa = NextPos(Pa,qa);
                break;
            case 0
                sum = a.coef + b.coef;
                if(sum != 0.0) {
                    SetCurElem(ha, qa);
                    ha = qa;
                } else {
                    DelFirst(ha, qa);
                    FreeNode(qa);
                }
                DelFirst(hb,qb);
                FreeNode(qb);
                qb = NextPos(Pb,hb);
                qa = NextPos(Pa,ha);
                break;
            case 1
                DelFirst(hb, qb);
                InsFirst(ha, qb);
                qb = NextPos(Pb, hb);
                ha = NextPos(Pa, ha);
                break;
```

```
            }
        }
        if(!ListEmpty(Pb)) {
            Append(Pa,qb);
        }
        FreeNode(hb);
}
```

顺序栈的定义

```
#define Stack_Int_Size 100;  存储空间初始化分配
#define StackIncrement 10;   存储空间分配增量
typedef struct {
        SElemType base;  存储空间初始分配量
        SElemType top;
        int stacksize;
}SqStack;
```

构造一个空栈 S

```
Status InitStack(SqStack &S) {
        S.base = (SElemType )malloc(Stack_Int_Sizesizeof(SElemType));
        if(!S.base) {
            exit(OVERFLOW);
        }
        S.top = S.base;
        S.stacksize = Stack_Int_Size;
        return OK;
}
```

若栈不空，则用 e 返回 S 的栈顶元素，并返回 OK；否则返回 ERROR

```
Status GetTop(SqStack S,SElemType &e) {
        if(S.top == S.base) {
            return ERROR;
        }
        e = (S.top - 1);
        return OK;
}
```

插入元素 e 为新的栈顶元素

```
Status Push(SqStack &S, SElemType e) {
        if(S.top - S.base = S.stacksize) {栈满，追加存储空间
            S.base = (SElemType)realloc(S.base,(S.stacksize +    StackIncrement)sizeof(SElemType));
            if(!S.base) {
                exit(OVERFLOW);
            }
```

```
            S.top = S.base + S.stacksize;
            S.stacksize += StackIncrement;
        }
        S.top++ =e;
        return OK;
}
```

若栈不空，则删除 S 的栈顶元素，用 e 返回其值，并返回

```
Status Pop(SqStack &S, SElemType &e) {
        if(S.top == S.base) {
            return ERROR;
        }
        e = --S.top;
        return OK;
}
```

栈的应用

算法 3.1 对于输入的任意一个非十进制整数，打印输出与其等值的八进制数

```
void conversion() {
        InitStack(S);
        scanf(%d,N);
        while(N) {
            Push(S , N % 8);
            N = N8;
        }
        while(!StackEmpty(s)) {
            Pop(S,e);
            printf(%d,e);
        }
}
```

算法 3.2 利用字符栈 S，从中端接受一行并传送至调用过程的数据区

```
void LineEdit() {
        InitStack(S);
        ch = getchar();
        while(ch != EOF) {
            while(ch! = EOF && ch != 'n') {
                switch(ch) {
                    case '#' 仅当栈非空时退栈
                        Pop(S,c);
                        break;
                    case '@'
                        ClearStack(S);
                        break;
```

```
                default
                    Push(S,ch);
                    break;
            }
            ch = getchar();
        }
        ClearStack(S);
        if(ch! = EOF) {
            ch = getchar();
        }
    }
    DestroyStack(S);
}
```

算法 3.3 若迷宫 maze 中存在从入口 start 到出口 end 的通道，则求得一条存放在栈中（从栈底到栈顶），并返回 true；否则 false

```
Status MazePath(MakeType maze, PosType start, PosType end) {
    InitStack(S);
    curpos = start;
    curstep = 1;
    do {
        if(Pass(curpos)) {
            FootPrint(curpos);
            e = (curstep, curpos, 1);
            Push(S,e);
            if(curpos == end) {
                return TRUE;
            }
            curpos = NextPos(curpos, 1);
            curstep++;
        } else {
            if(!StackEmpty(S)) {
                Pop(S,e);
                while(e.di === 4 && !StackEmpty(S)) {
                    MarkPrint(e.seat);
                    Pop(S,e);
                }
                if(e,di    4) {
                    e.di ++;
                    Push(S, e);
                }
            }
        }
    } while(!StackEmpty(S));
```

```
    return FlASE;
}
```

算法 3.4 算术表达式求值的算符优先算法，设 OPTR 和 OPND 分别为运算符栈和运算数栈

```
OperandType EvaluateExpression() {
    InitStack(OPTR);
    Push(OPTR,'#');
    InitStack(OPND);
    c = getchar();
    while(c != '#'    GetTop(OPTN, c) != '#') {
        if(!In(c, OP)) {  不是运算符则进栈
            Push(OPND , c);
            c = getchar();
        } else {
            switch(Precede(GetTop(OPTR),c)) {
                case ''  栈顶元素优先级低
                    Push(OPTR, c);
                    c = getchar();
                    break;
                case '='  脱括号并接收下一字符
                    Pop(OPTR, x);
                    c = getchar();
                    break;
                case ''  退栈并将运算结果入栈
                    Pop(OPTR, theta);
                    Pop(OPND, b);
                    Pop(OPND, a);
                    Push(OPND, Operate(a, theta, b));
                    break;
            }
        }
    }
    return GetTop(OPND);
}
```

算法 3.5 汉诺塔，将塔座 x 上按直径由小到大且自上而下编号 1 至 n 个原盘规则搬到塔座 z 上，y 可作辅助塔座。

搬动操作 move(x, n, z)可定义为(c 是初值为 0 的全局变量，对搬动计数)

```
void hanoi(int n, char x, char y, char z) {
    if(n === 1) {
        move(x, 1, z);  将编号为 1 的圆盘从 x 移动到 z
    } else {
        hanoi(n-1, x, z, y);将 x 上编号为 1 至 n-1 的圆盘移到 y，z 作铺助塔
        move(x, n ,z);          将编号为 n 的圆盘从 x 移到 z
```

```
            hanoi(n-1, y, x, z);将 y 上编号为 1 至 n-1 的圆盘移到 z
        }
}

 ----- 单链队列   队列的链式存储结构 ------
 typedef struct QNode {
        QElemType data;
        struct QNode next;
 }QNode, QueuePtr;
typedef struct {
        QueuePtr front;   队头指针
        QueuePtr rear;
}LinkQueue;

构造一个空队列 Q
Status InitQueue(LinkQueue &Q) {
        Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode));
        if(!Q.front) {
                exit(OVERFLOW);存储分配失败
        }
        Q.front - next = NULL;
        return OK;
}

销毁队列 Q
Status DestroyQueue(LinkQueue &Q) {
        while(Q.front) {
                Q.rear = Q.front - next;
                free(Q.front);
                Q.front = Q.rear;
        }
        return OK;
}

插入元素 e 为 Q 的新的队列元素
Status EnQueue(LinkQueue &Q, QElemType e) {
        p = (QueuePtr)malloc(sizeof(QNode));
        if(!p) {
                exit(OVERFLOW);     存储分配失败
        }
        p - data = e;
        p - next = NULL;
        Q.rear - next = p;
        Q.rear = p;
```

```
    return OK;
}

若队列不空，则删除 Q 的队头元素，用 e 返回其值，并返回 OK
Status DeQueue(LinkQueue &Q, QElemType &e) {
    if(Q.front == Q.rear) {
        return ERROR;
    }
    p = Q.front - next;
    e = p - data;
    Q.front - next = p - next;
    if(Q.rear == p) {
        Q.rear = Q.front;
    }
    free(p);
    return OK;
}

------ 循环队列 ------
#define MaxQSize 100
typedef struct {
    QElemType base; 初始化的动态分配储存空间
    int front;          头指针，若队列不空，指向队列头元素
    int rear;           尾指针，若队列不空，指向队列尾元素的下一个位置
}SqQueue;

构造一个空队列 Q
Status InitQueue(SqQueue &Q) {
    Q.base = (QElemType)malloc(MaxQSizesizeof(QElemType));
    if(!Q.base) {
        exit(OVERFLOW);
    }
    Q.front = Q.rear = 0;
    return OK;
}

返回 Q 的元素个数，即队列的长度
int QuereLength(SqQueue Q) {
    return(Q.rear - Q.front + MaxQSize) % MaxQSize;
}

插入元素 e 为 Q 的新队尾元素
Status EnQueue(SqQueue &Q, QElemType e) {
    if((Q.rear + 1) % MaxQSize == Q.front) {
```

```
        return ERROR;
    }
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear + 1) % MaxQSize;
    return OK;
}
```

若队列不空，则删除 Q 的队头元素，用 e 返回其值，并返回 OK

```
Status DeQueue(SqQueue & Q, QElemType &E) {
    if(Q.front == Q.rear) {
        return ERROR;
    }
    e = Q.base[Q.front];
    Q.front = (Q.front + 1) % MaxQSize;
    return OK;
}
```

算法 3.6 银行业务模拟，统计一天内客户在银行逗留的平均时间

```
void Bank_Simulation(int CloseTime) {
    OpenForDay();
    while(MoreEvent) {
        EventDrived(OccurTime, EventType); 事件驱动
        switch(EventType) {
            case 'A'
                CustomerArrived();  处理客户到达事件
                break;
            case 'D'
                CustomerDeparture(); 处理客户离开事件
                break;
            default
                Invalid();
        }
    }
    CloseForDay()  计算平均逗留时间
}
```

二叉树的存储结构

```
#define Max_Tree_Size 100;
typedef TElemType SqBiTree[Max_Tree_Size];
SqBiTree bt;
```

算法 6.1 先序遍历二叉树 T 的递归算法，对每个数据元素调用方式 Visit

```
Status PreOrderTraverse(BiTree T.Status( Visit)(TElemType e)) {
    Status PrintElement(TElemType e) {
```

```
            printf(e);
            return OK;
        }
    if(T) {
        if(Visit(T - data)) {
            if(PreOrderTraverse(T - lchild.Visit)) {
                if(PreOrderTraverse(T - rchild.Visit)) {
                    return OK;
                }
            }
        }
        return ERROR;
    } else {
        return OK;
    }
}
```

算法 6.2 采用二叉链表存储结构，Visit 是对数据元素操作的应用函数

```
Status InOrderTraverse(BiTree T.Status(Visit)(TElemType e)) {
    InitStack(S);
    Push(S,T);
    while(!StackEmpty(S)) {
        while(GetTop(S,p) && p) {
            Push(S, p - lchild); 向左走到尽头
        }
        Pop(S, p);                    空指针退栈
        if(!StackEmpty(S)) {          访问结点，向右一步
            Pop(S, p);
            if(!Visit(p - data)) {
                return ERROR;
            }
            Push(S, p - rchild);
        }
    }
    return OK;
}
```

算法 6.3 采用二叉链表存储结构，visit 是对数据元素操作的应用函数中序遍历二叉树 T 的非递归算法，对每个元素调用函数 Visit

```
Status InOrderTraverse(BiTree T.Status( Visit)(TElemType e)) {
    IninStack(S);
    p = T;
    while(p    !StackEmpty(S)) {
        if(p) {
```

```
            Push(S, p);
            p = p - lchild;     根指针进栈，遍历左子树
        } else {                        根指针退栈，访问根结点，遍历右子树
            Pop(S, p);
            if(!Visit(p - data)) {
                return ERROR;
            }
            p = p - rchild;
        }
    }
    return OK;
}

算法 6.4  按先序次序输入二叉树中结点的值，空格字符表示空树
Status CreateBiTree(BiTree &T) {
    scanf(&ch);
    if(ch == '') {
        T = null;
    } else {
        if(!(T = (BiTNode    )malloc(sizeof(BiTNode)))) {
            exit(OVERFLOW);
            T - data = ch;
            CreateBiTree(T - lchild);
            CreateBiTree(T - rchild);
        }
    }
    return OK;
}

-----二叉树的二叉线索存储表示------
typedef enum PointerTag { Link, Thread };
typedef struct BiTrNode {
    TElemType data,
    struct BiTreNode lchild, rchild;
    PointerTag LTag, Rtag;
}BiThrNode, BiThrTree;

算法 6.5  中序遍历二叉线索树 T 的非递归算法，对每个数据元素调用函数 Visit
Status InOrderTraverse_Thr(BiThrTree T.Status(Visit)(TElemType e)) {
    p = T - lchild;     p 指向根结点
    while(p != T) {       空树或遍历结束时，p == T
        while(p - LTag == Link) {
            p = p - lchild;
        }
```

```
            if(!Visit(p - data)) {
                return ERROR;
            }
            while(p - RTag == Thread && p - rchild != T) {
                p = p - rchild;
                Visit(p - data);
            }
            p = p - rchild;
        }
        return OK;
}
```

算法 6.6 中序遍历二叉树 T，并将其中序线索化，Thrt 指向头结点

```
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T) {
        if(!(Thrt = (BiThrTree)malloc(sizeof(BiThrTree)))) {
            exit(OVERFLOW);
        }
        Thrt - LTag = Link;     建头结点
        Thrt - RTag = Thread;
        Thrt - rchild = Thrt;   右指针回指
        if(!T) {
            Thrt - lchild = Thrt; 若二叉树空，则左指针回指向
        } else {
            Thrt - lchild = T;
            pre = Thrt;
            InThreading(T);   中序遍历进行中序线索化
            pre - rchild = Thrt; 最后一个结点线索化
            pre - RTag = Thread;
            Thrt - rchild = pre;
        }
        return OK;
}
```

算法 6.7

```
void InThreading(BiThrTree p) {
        if(p) {
            InThreading(p - lchild); 左子树线索化
            if(!p - lchild) {  前驱线索
                p - LTag = Thread;
                p - lchild = pre;
            }
            if(!pre - rchild) {  后继线索
                pre - RTag = Thread;
                pre - rchild = p;
```

```
            }
            pre = p;
            InThreading(p - rchild);
        }
}

------- 树和森林  ---------》
#define Max_Tree_Size 100
typedef struct PTNode {  结点机构
        TElemType data;
        int parent;      双亲位置域
}PTNode;
typedef struct {  树结构
        PTNode nodes[Max_Tree_Size];
        int r,n;   根的位置和结点树
}PTree;

----树的孩子链表存储表示-------
  typedef struct CTNode {
          int child;
          struct CTNode next;
  }ChildPtr;
typedef struct {
        TElemType datal
        ChildPtr firstchild;   孩子链表头指针
}CTBox;
typedef struct {
        CTBox node[Max_Tree_Size];
        int n,r;      结点数和根的位置
}CTree;

  -----树的二叉链表（孩子-兄弟）存储表示------
  typedef struct CSNode {
          ElemType data;
          struct CSNode firstchild, nextsibling;
  }CSNode, CSTree;

----ADT MFSet  的树的双亲表存储表示------
typedef PTree MFSet;
算法 6.8
int find_mfset(MFSet S, int i) {
        找集合 S 中 i 所在子集的根
        if(i    1    i    S.n) {
                return -1;
```

```
    }
    for(j = i ; S.node[j].parent    0 ; j = S.node[j].parent);
    return j;
}
```

算法 6.9 求并集 Si U Sj

```
Status merge_mfset(MFSet &S, int i, int j) {
    S.nodes[i]和 S.nodes[j]分别为 S 互不相交的两个子集 Si 和 Sj 的根结点
    if(i   1   i   S.n   j   1   j   S.n) {
        return ERROR;
    }
    S.nodes[i].parent = j;
    return OK;
}
```

6.10 求并集

```
void mix_mfset(MFSet &S, int i, int j) {
     if(i   1   i   S.n   j   1   j   S.n) {
        return ERROR;
    }
    if(S.nodes[j].parent    S.nodes[j].parent) {
        S.nodes[j].parent += S.nodes[i].parent;
        S.nodes[i].parent = j;
    } else {
        S.nodes[i].parent += S.nodes[j].parent;
        S.nodes[j].parent = i;
    }
}
```

6.11 确定 i 所在子集，
并将从 i 至根路径上所有结点变成根的孩子结点。

```
int fix_mfset(MFSet &S,int i) {
    if(i   1   i   S.n) {
        return -1;
    }
    for(j = i;S.nodes[j].parent    0; j = S.nodes[j].parent);
    for(k = i;k != j;k = t) {
        t = S.nodes[k].parent;
        S.nodes[k].parent = j;
    }
    return j;
}
```

求含 n 个元素的集合 A 的幂集 p(A)。进入函数时已对 A 中前 i-1 个元素作了取舍处理

现从第 i 个元素起进行取舍处理。若 in,则求得幂集的一个元素，并输出之
初始调用：PowerSet(1 , n)

```
void PowerSet(int i, int n) {
    if(i   n)   输出幂集的一个元素
    else {
        PowerSet(i + 1, n);  取第一个元素
        PowerSet(i + 1, n);  舍第一个元素
    }
}
```

线性表 A 表示集合 A，线性表 B 表示幂集 P(A)的一个元素
局部量 k 为进入函数时表 B 的当前长度。第一次调用函数时，B 为空表，i = 1

```
void GetPowerSet(int i, List A, List &B) {
    if(i   ListLength(A)) {
        Output(B);
    } else {
        GetElem(A, i, x);
        k = ListLength(B);
        ListInsert(B, k+1 , x);
        GetPowerSet(i+1, A, B);
        ListDelete(B, k+1 ,x);
        GetPowerSet(i+1, A, B)
    }
}
```

进入本函数时，在 n X n 棋盘前 i-1 行已放置了互不攻击的 i-1 个棋子
现从第 i 行起继续为后续棋子选择适合位置
当 in 时，求得一个合法布局，输出之

```
void Trial(int i, int n) {
    if(i     n)  输出棋盘当前布局  n 为 4 即为 4 皇后问题
    else {
        for(j = 1; j = n ; ++j) {
            if(当前布局合法) {
                Trial(i+1 ,n);
            }
        }
    }
}
```

---图---
理解图的基本概念，掌握图的存贮结构，图的遍历、最小生成树和拓扑排序
图的数组存储方式

```
# define INFINITY INT_MAX    最大值
# define Max_Vertex_Num 20  最大顶点树
```

```
typedef enum {DG, DN, UDG, UDN} GraphKind; {有向图，有向网，无向图，无向网}
typedef struct ArcCell {
    VRType adj;     VRType 是顶点关系类型。对无权图用 1 或 0
    InfoType info;   该弧相关信息的指针
}ArcCell, AdjMatrix[Max_Vertex_Num][Max_Vertex_Num];
typedef struct {
    VertexType vexs[Max_Vertex_Num]; 顶点向量
    AdjMatrix arcs; 邻接矩阵
    int vexnum, arcnum;   图的当前顶点数和弧数
    GraphKind kind;   图的种类标志
}MGraph;

采用数组（邻接矩阵）表示法，构造图 G
Status CreateGraph(MGraph &G) {
    scanf(&G.kind);
    switch (G.kind) {
        case DG return CreateDG(G); 构造有向图 G
        case DN return CreateDN(G); 构造有向网 G
        case UDG return CreateNDG(G); 构造无向图 G
        case UDN return CreateUDN(G); 构造无向图 G
        default return ERROR;
    }
}

---图的邻接表存储表示---
#define Max_Vertex_Num 20
typedef struct ArcNode {
    int adjvex; 该弧所指向的顶点的位置
    struct ArcNode nextarc; 指向下一条弧的指针
    InfoType info; 该弧相关信息的指针
}ArcNode;

typedef struct VNode {
    VertexType dataA;   顶点信息
    ArcNode firstarc;   指向第一条依附顶点的弧的指针
}VNode, AdjList[Max_Vertex_Num];
typedef struct {
    AdjList vertices;
    int vexnum, arcnum; 图的当前顶点数和弧数
    int kind;   图的种类
}ALGraph;

深度优先遍历
Boolean visited[MAX]; 访问标志数组
```

```
Status (VisitFunc)(int v);  函数变量

void DFSTraverse(Graph G, Status (Visit)(int v)) {
    VisitFunc = Visit;
    for(v = 0 ; v    G.vexnum ; ++v) {
        visited[v] = FALSE;    访问标志数组初始化
    }
    for(v = 0 ; v    G.vexnum ; ++v) {
        if(!visited[v]) {
            DFS(G, v);
        }
    }
}

void DFS(Graph G, int v) {
    visited[v] = TRUE;
    VisitFunc(v);
    for(w = FirstAdjVex(G, v); w=0 ; w=NextAdjVex(G, v, w)) {
        if(!visited[w]) {
            DFS(G, w);
        }
    }
}

按广度优先非递归遍历 G
void BFSTraverse(Graph G, Status(Visit)(int v)) {
    for(v = 0 ; v    G.vexnum ; ++v) {
        visited[v] = FALSE;
    }
    InitQueue(Q);
    for(v = 0; v    G.vexnum; ++v) {
        if(!visited[v]) {
            visited[v] = TRUE;
            Visit(v);
            EnQueue(Q, v);
            while(!QueueEmpty(Q)) {
                DeQueue(Q, u);
                for(w = FirstAdjVex(G, u); w=0; w=NextAdjVex(G, u, w)) {
                    if(!Visited[w]) {
                        Visited[w] = TRUE;
                        Visit(w);
                        EnQueue(Q, W);
                    }
                }
```

```
                }
            }
        }
}
```

最小生成树

```
void MiniSpanTree_PRIM(MGraph G, VertexType u) {
    用普里姆算法从第 u 个顶点出发构造网 G 的最小生成树 T，输出 T 的各条边
    记录从顶点集 U 到 V-U 的代价最小的边的辅助数组定义
     struct {
          VertexType adjvex;
          VRType lowcost;
     }closedge[Max_Vertex_Num];
    k = LocateVex(G, u);
    for(j = 0;j    G.vexnum ; ++j) {  辅助数组初始化
        if(j!=k) {
            closedge[j] = {u, G.arcs[k][j].adj};
        }
    }
    closedge[k].lowcost = 0;  初始  U={u}
    for(i = 1;i    G.vexnum; ++i) {  选择其余 G.vexnum - 1 个顶点
        k = minimum(closedge);  求出 T 的下一个结点：第 k 顶点
        printf(closedge[k].adjvex, G.vexs[k]);   输出生成树的边
        closedge[k].lowcost = 0;    第 k 顶点并入 U 集
        for(j = 0; j    G.vexnum ; ++j) {
            if(G.arcs[k][j].adj    closedge[j].lowcost) {
                新顶点并入 U 后重新选择最小边
                closedge[j] = {G.vexs[k], G.arcs[k][j].adj};
            }
        }
    }
}
```

拓扑排序

```
Status TopologicalSort(ALGraph G) {
    有向图 G 采用邻接表存储结构
    若 G 无回路，则输出 G 的顶点的一个拓扑序列并返回 OK，否则 ERROR。
    FindInDegree(G, indegree);
    InitStack(S);
    for(i = 0 ; i    G.vexnum ; ++i) {  建零入度顶底栈 S
        if(!indegree[i]) {    入度为 0 者进栈
            Push(S, i);
        }
        count = 0;  对输出顶点计数
```

```
            while(!StackEmpty(S)) {
                Pop(S, i);
                printf(i, G.vertices[i].data);
                ++count; 输出 i 号顶点并计数
                for(p = G.vertices[i].firstarc; p=p - nextarc) {
                    k = p - adjvex;
                    if(!(--indegree[k])) {
                        Push(S, k);
                    }
                }
            }
            if(count    G.vexnum) {
                return ERROR;
            } else {
                return OK;
            }
        }
}
```

---查询---

静态查找的顺序存储结构

```
typedef struct {
    ElemType elem;
    int length;
}SSTable;
```

在顺序表 ST 中顺序查找其关键字等于 key 的数据元素。若找到，则函数值为该元素在表中的位置，否则为 0

顺序查询

```
int Search_Seq(SSTable ST, KeyType Key) {
    ST.elem[0].key = key; 哨兵
    for(i = ST.length; !EQ(ST.elem[i].key, key) ; --i);
    return i;
}
```

在有序表 ST 中折半查找其关键字等于 key 的数据元素。该元素在表中的位置，否则为 0

折半查找

```
int Search_Bin(SSTable ST, KeyType key) {
    low = 1;
    high = ST.length;
    while(low = high) {
        mid = (low + high)    2;
        if(EQ(key, ST.elem[mid].key)) {
```

```
            return mid;
        } else if(LT(key, ST.elem[mid].key)) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return 0;
}

---哈希表存储结构---
int hashsize[] = {998,...};
typedef struct {
    ElemType elem;
    int count;
    int sizeindex;
}HashTable;

在开放定址哈希表 H 中查找关键码为 K 的元素，若查找成功，以 p 指示
元素在表中位置返回 SUCCESS；否则，以 p 指示插入位置，并返回 UNSUCCESS
c 用以计冲突次数，其初值置 0,拱建表插入时参考
Status SearchHash(HashTable H, KeyType K, int &p, int &c) {
    p = Hash(K);    求得哈希地址
    while(H.elem[p].key != NULLKEY &&   该位置中填有记录
    !EQ(K,H.elem[p].key)) {    并且关键字不想等
        collision(p, ++c)   求得下一探查地址 p
    }
    if(EQ(K, H.elem[p].key)) {
        return SUCCESS;
    } else {
        return UNSUCCESS;
    }
}

查找不成功时插入数据元素 e 到开放定址哈希表 H 中，并返回 OK
若冲突次数过大，则重建哈希表
Status InsertHash(HashTable &H, ElemType e) {
    c = 0;
    if(SearchHash(H, e.key, p, c)) {
        return DUPLICATE;
    } else if(c     hashsize[H.sizeindex]2) {
        H.elem[p] = e;
        ++H.count;
        return OK;
```

```
    } else {
        RecreateHashTable(H);
        return UNSUCCESS;
    }
}
```

---排序---

```
#define MaxSize 20;
typedef int KeyType;
typedef struct {
    KeyType key;
    InfoType otherinfo;
}RedType;
typedef struct {
    RedType r[MaxSize + 1]; r[0]闲置或哨兵单元
    int length;
}SqList;
```

对顺序表 L 作直接插入排序

```
void InsertSort(SqList &L) {
    for(i = 2; i=L.length; ++i) {
        if(LT(L.r[i].key, L.r[i-1].key)) { ‘’
            L.r[0] = L.r[i];
            L.r[i] = L.r[i - 1];
            for(j = i-2 ; LT(L.r[0].key, L.r[i].key) ; --j) {
                L.r[j+1] = L.r[j];
            }
            L.r[j + 1] = L.r[0];
        }
    }
}
```

插入排序 网上算法

```
void insertion_sort(int arr[], int len){
    int i,j,key;
    for (i=1;ilen;i++){
        key = arr[i];
        j=i-1;
        while((j=0) && (arr[j]key)) {
                arr[j+1] = arr[j];
                j--;
        }
        arr[j+1] = key;
    }
```

```
}

对顺序表 L 作折半插入排序
void BInsertSort(SqList &L) {
    for(i = 2; i = L.length; ++i) {
        L.r[0] = L.r[i];
        low = 1;
        high = i - 1;
        while(low = high) {
            m = (low + high)    2;  折半
            if(LT(L.r[0].key,L.r[m].key)) { ''  插入点在低半区
                high = m - 1；
            } else {        插入点在高半区
                low = m + 1;
            }
        }
        for(j = i -1 ; j = high + 1; --j) {
            L.r[j+1] = L.r[j];
        }
        L.r[high + 1] = L.r[0];
    }
}

对顺序表 L 作一趟希尔插入排序
1.前后记录位置的增量是 dk，而不是 1；
2.r[0]只是暂存单元，不是哨兵。当 j=0 时，插入位置已找到
void ShellInsert(SqList &L, int dk) {
    for(i = dk + 1; i= L.length ; ++i) {
        if(LT(L.r[i].key, L.r[i - dk].key)) {  需将 L.r[i]插入有序增量子表
            L.r[0] = L.r[i];  暂存在 L.r[0]
            for(j = i - dk; j0 && LT(L.r[0].key, L.r[j].key); j-=dk) {
                L.r[j + dk] = L.r[j];
            }
            L.r[j + dk] = L.r[0];
        }
    }
}

按增量序列 dlta[0..t-1]对顺序表 L 作希尔排序
void ShellSort(SqList &L, int dlta[] , int t) {
    for(k = 0;k    t; ++k) {
        ShellInsert(L, dlta[k]);
    }
}
```

快速排序
交换顺序表 L 中子表 r[low..high]的记录
在它之前（后）的记录均不大（小）于它

```
int Partition(SqList &L, int low, int high) {
    L.r[0] = L.r[low];
    pivotkey = L.r[low].key;  记录关键字
    while(low    high) {
        while(low    high && L.r[high].key = pivotkey) {
            --high
        }
        L.r[low] = L.r[high];
        while(low    high && L.r[low].key = pivotkey) {
            ++low;
        }
        L.r[high] = L.r[low];
    }
    L.r[low] = L.r[0];
    return low;
}
```

对顺序表 L 作快速排序

```
void QSort(SqList &L, int low, int high) {
    if(low    high) {
        pivotkey = Partition(L, low, high);
        Qsort(L, low, pivotkey - 1);
        Qsort(L, pivotkey + 1, high);
    }
}

void QuickSort(SqList &L) {
    Qsort(L, 1, L.length);
}
```

对顺序表 L 作简单选择排序

```
void SelectSort(SqList &L) {
    for(i = 1 ; i    L.length ; ++i) {
        j = SelectMinKey(L, i);  在 L.r[i]中选择 key 最小的记录
        if(i != j) {
            k = L.r[i];
            L.r[i] = L.r[j];
            L.r[j] = k;
        }
```

```
    }
}
```

选择排序 网上算法

```
void selection_sort(int arr[], int len)
{
    int i,j;
    for (i = 0 ; i    len - 1 ; i++)
    {
        int min = i;
        for (j = i + 1; j    len; j++)        走訪未排序的元素
            if (arr[j]    arr[min])      找到目前最小值
                min = j;       紀錄最小值
        swap(&arr[min], &arr[i]);       做交換
    }
}
```

堆排序

已知 H.r[s..m]中记录的关键字除 H.r[s].key 之外均满足堆的定义

使 H.r[s..m]成为一个大顶堆

```
void HeapAdjust(HeapType &H, int s, int m) {
    rc = H.r[s];
    for(j = 2s ; j = m ; j = 2) {
        if(j    m && LT(H.r[j].key, H.r[j+1].key)) {
            ++j;
        }
        if(!LT(rc.key, H.r[j].key)) {
            break;
        }
        H.r[s] = H.r[j];
        s = j;
    }
    H.r[s] = rc;
}
```

对顺序表 H 进行堆排序

```
void HeapSort(HeapType &H) {
    for(i = H.length2 ; i    0 ; --i) {
        HeapAdjust(H, i, H.length);
    }
    for(i = H.length; i1; --i) {
        swap(H.r[1], H.r[i]);
        HeapAdjust(H, 1, i-1);
    }
```

```
}

归并排序
将有序的 SR[i..M]
void Merge(RcdType SR[], RcdType &TR[], int i, int m, int n) {
    for(j = m + 1,k = i ; i = m && j = n ; ++k) {
        if(LQ(SR[i].key, SR[j].key)) {
            TR[k] = SR[i++];
        } else {
            TR[k] = SR[j++]
        }
    }
    if(i = m) {
        TR[k..n] = SR[i..m]; 将剩余 SR[i..m]复制到 TR
    }
    if(j = n) {
        TR[k..n] = SR[j..n]; 将剩余 SR[j..n]复制到 TR
    }
}

void MSort(RcdType SR[], RcdType &TR1[], int s, int t) {
    if(s == t) {
        TR1[s] = SR[s];
    } else {
        m = (s + t)2; 将 SR[s..t]平分为 SR[s..m]和 SR[m+1..t]
        MSort(SR, TR2, s, m); 递归将 SR[s..m]归并为有序的 TR2[s..m]
        MSort(SR, TR2, m + 1 ,t); 递归将 SR[m+1..t]归并为有序的 TR2[m+1..t]
        Merge(TR2, TR1, s, m, t);将 TR2[s..m]和 TR2[m+1..t]归并到 TR1[s..t]
    }
}

void MergeSort(SqList &L) {
    MSort(L.r, L.r, 1 ,L.length);
}
```