顺序表默认结构体：

```
typedef struct{
  int data[maxsize];
  int length;
} Sqlist;
```

1、顺序表递增有序，插入元素 x，仍递增有序

```
int find(Sqlist L, int x){
  int i = 0;
  for (; i < L.length; ++i){
    if (x < L.data[i])
      break;
  }
  return i;
}
void insert(Sqlist &L, int x){
  int j, p;
  p = find(L, x);
  for (j = L.length - 1; j >= p; --j)
    L.data[j + 1] = L.data[j];
  L.data[p] = x;
  ++(L.length); //别遗漏
}
```

2、用顺序表最后一个元素覆盖整个顺序表中最小元素，并返回该最小元素

```
int Del_Min(Sqlist &L){
  int min = L.data[0];
  int pos = 0;
  for (int i = 0; i < L.length; ++i)
    if (L.data[i] < min){
      min = L.data[i];
      pos = i;
    }
  L.data[pos] = L.data[L.length - 1];
  L.length--;
  return min;
}
```

3、将顺序表中的元素逆置

```
void Reverse(Sqlist &L){
  int temp，i=0, j = L.length -1;
  for (; i < L.length / 2; ++i, --j){
    temp = L.data[i];
    L.data[i] = L.data[j];
    L.data[j] = temp;
  }
}
```

4、将（a1,a2,a3……am,b1,b2,……bn）转换成 (b1,b2,……bn,a1,a2,a3,……am)

```
void Reverse(int A[], int m, int n){
  int mid = (m + n) / 2;
  for (int i = m, j = 0; i <= mid; ++i, ++j){
    int temp = A[i];
    A[i] = A[n - j];
    A[n - j] = temp;
  }
}

void change(int A[], int m, int n){
  Reverse(A, 0, m + n - 1);
  Reverse(A, 0, n - 1);
  Reverse(A, n, m + n - 1);
}
```

5、删除顺序表中所有值为 x 的元素(两种方法)

法一：

```
void del(Sqlist &L, int x){
    int k = 0;
    for (int i = 0; i <= L.length - 1; ++i)
        if (L.data[i] != x){
            L.data[k] = L.data[i];
            ++k;
        }
    L.length = k;
}
```

法二：

```
void Del(Sqlist &L, int x){
    int k = 0;
    for (int i = 0; i <= L.length - 1; ++i){
        if (L.data[i] == x)
            ++k;
        else
            L.data[i - k] = L.data[i];
    }
    L.length = L.length - k;
}
```

6、从顺序表中删除给定值在 s 到 t 之间（包含 s 和 t）的所有元素

```
bool del(Sqlist &L, int s, int t){
    int i, k = 0;
    if (L.length == 0 || s >= t)
        return false;
    for (i = 0; i < L.length; ++i){
        if (L.data[i] >= s && L.data[i] <= t)
            ++k;
        else
            L.data[i - k] = L.data[i];
    }
    L.length -= k;
    return true;
}
```

7、从有序表中删除所有值重复的元素

```
bool del(Sqlist &L){
    int i, j;
    for (i = 0, j = 1; j < L.length; ++j)
        if (L.data[i] != L.data[j])
            L.data[++i] = L.data[j];
    L.length = i + 1;
    return true;
}
```

8、两个递增有序表合并成一个递增有序表

```
bool merge(Sqlist A, Sqlist B, Sqlist &C){
    int i = 0, j = 0, k = 0;
    while (i < A.length && j < B.length){
        if (A.data[i] < B.data[j])
            C.data[k++] = A.data[i++];
        else
            C.data[k++] = B.data[j++];
    }
    while (i < A.length)
        C.data[k++] = A.data[i++];
    while (j < B.length)
        C.data[k++] = B.data[j++];
    C.length = k;
    return true;
}
```

9、求两个递增序列合并后的中位数(两种方法)

法一：

```
int find(Sqlist &A, Sqlist &B){
   int i=0, j=0, k=0;
   while (1){
      if (A.data[i] < B.data[j]){
         if (++k == (A.length + B.length) / 2)
            return A.data[i];
         ++i;
      }
      else{
         if (++k == (A.length + B.length) / 2)
            return B.data[j];
         ++j;
      }
   }
}
```

法二：

```
int find(Sqlist &A, Sqlist &B){
   int a0 = 0, b0 = 0, am, bm;
   int an = A.length - 1, bn = B.length - 1;
   while (a0 != an || b0 != bn){
      am = (a0 + an) / 2;
      bm = (b0 + bn) / 2;
      if (A.data[am] == B.data[bm])
         return A.data[am];
      else if (A.data[am] < B.data[bm]){
         a0 = a0 + bn - bm;
         bn = bm;
      }
      else{
         b0 = b0 + an - am;
         an = am;
      }
   }
   if (A.data[a0] > B.data[b0])
      return B.data[b0];
   else
      return A.data[a0];
}
```

10、设计一个时间上尽可能高效的算法，找出数组中未出现的最小正整数

```
int find(int A[], int n){
   int i;
   int *B = new int[n];
   for (int k = 0; k < n; ++k)
      B[k] = 0;
   for (i = 0; i < n; ++i)
      if (A[i] > 0 && A[i] <= n)
         B[A[i] - 1] = 1;
   for (i = 0; i < n; ++i)
      if (B[i] == 0)
         break;
   delete[] B;
   return i + 1;
}
```

11、若一个整数序列中有过半相同元素，则称其为主元素，设计算法找出数组 A(a0,a1……an-1)的主元素。（其中 0<ai<n）若存在主元素则输出，否则返回-1

```
int fun(int A[], int n){
  int *B = new int[n];
  for (int i = 0; i < n; ++i)
    B[i] = 0;
  int i, k, max = 0;
  for (i = 0; i < n; ++i)
    if (A[i] > 0 && A[i] <= n)
      B[A[i] - 1]++;
  for (i = 0; i < n; ++i)
    if (B[i] > max){
      max = B[i];
      k = i;
    }
  delete[] B;
  if (max > n / 2)
    return k + 1;
  else
    return -1;
}
```

单链表默认结构体：

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

1、设计一个递归算法，删除不带头节点的单链表 L 中所有值为 x 的结点

```
void del_x(LNode *&L, int x){
  LNode *p;
  if (L == NULL)
    return;
  if (L->data == x){
    p = L;
    L = L->next;
    free(p);
    del_x(L, x);
  }
  else
    del_x(L->next, x);
}
```

2、删除带头节点单链表中所有值为 x 的结点

```
void del(LNode *&L, int x){
  LNode *p = L->next, *pre = L, *q;
  while (p != NULL){
    if (p->data == x){
      q = p;
      pre->next = p->next;
      p = p->next;
      free(q);
    }
    else{
      pre = p;
      p = p->next;
    }
  }
}//法一
void Del(LNode *&L, int x){
  LNode *p = L;
  while (p->next != NULL){
    if (p->next->data == x){
      LNode *q = p->next;
      p->next = q->next;
      free(q);
    }
    else
      p = p->next;
  }
}//法二
```

3、删除带头节点单链表中第一个值为 x 的结点

```
int finddelete(LNode *&C, int x){
  LNode *p, *q;
  p = C;
  while (p->next != NULL){
    if (p->next->data == x)
      break;
    p = p->next;
  }
  if (p->next == NULL)
    return 0;
  else{
    q = p->next;
    p->next = q->next;
    free(q);
    return 1;
  }
}
```

4、从尾到头反向输出单链表每个结点的值

```
void print(LNode *L)
{
  if (L->next != NULL)
    print(L->next);
  cout << L->data << " ";
}
```

5、试编写算法将单链表就地逆置

```
void reverse(LNode *&L){
  LNode *p = L->next, *r;
  L->next = NULL;
  while (p != NULL){
    r = p->next;
    p->next = L->next;
    L->next = p;
    p = r;
  }
}//法一

void Reverse(LNode *&L){
  LNode *p = L->next, *r = p->next;
  LNode *pre;
  p->next = NULL;
  while (r != NULL){
    pre = p;
    p = r;
    r = r->next;
    p->next = pre;
  }
  L->next = p;
}//法二
```

6、从链表中删除给定值在 s 到 t 之间（不包含 s 和 t）的所有元素

```
void del(LNode *&L, int min, int max){
  LNode *p = L;
  while (p->next != NULL){
    if (p->next->data > min
            && p->next->data < max){
      LNode *u = p->next;
      p->next = u->next;
      free(u);
    }
    else
      p = p->next;
  }
}
```

7、试编写在带头结点的单链表 L 中删除最小值点的高效算法（已知最小值唯一）

```
void del_min(LNode *&L){
  LNode *p = L->next;
  LNode *minp = L;
  while (p->next != NULL){
    if (p->next->data <minp->next->data)
      minp = p;
    p = p->next;
  }
  LNode *u = new LNode;
  u = minp->next;
  minp->next = u->next;
  free(u);
}
```

8、试编写在不带头结点的单链表 L 中删除最小值点的高效算法（已知最小值唯一）

```
void del_min(LNode *&L){
  LNode *minp = L;
  LNode *p = L->next;
  while (p != NULL){
    if (p->data < minp->data)
      minp = p;
    p = p->next;
  }
  if (L == minp){
    L = L->next;
    free(minp);
    return;
  }
  p = L;
  while (p->next != minp)
    p = p->next;
  p->next = minp->next;
  free(minp);
}
```

9、给定一个单链表，按递增排序输出的单链表中各结点的数据元素，并释放节点所占空间

```
void del_min(LNode *&head){
  while (head->next != NULL){
    LNode *pre = head;
    LNode *p = head->next;
    while (p->next != NULL){
      if (p->next->data< pre->next->data)
        pre = p;
      p = p->next;
    }
    cout << pre->next->data << " ";
    LNode *u = pre->next;
    pre->next = u->next;
    free(u);
  }
  free(head);
}
```

10、将一个带头节点的单链表 A 分解成两个带头节点的单链表 A 和 B，使 A 中含奇数位置元素，B 中含偶数位置元素，且相对位置不变

```
Linklist create(LNode *&A){
  LNode *B = new LNode;
  B->next = NULL;
  LNode *ra = A, *rb = B, *p = A->next;
  A->next = NULL;
  while (p != NULL){
    ra->next = p;
    ra = p;
    p = p->next;
    rb->next = p;
    rb = p;
    p = p->next;
  }
  ra->next = NULL;
  rb->next = NULL;
  return B;
}
```

11、将一个单链表{a1,b1,a2,b2……an,bn}拆分成{ a1,a2……an }和{ bn,bn-1,……b1 }

```
Linklist create(LNode *&A){
   LNode *B = new LNode;
   B->next = NULL;
   LNode *ra = A, *p = A->next, *q;
   A->next = NULL;
   while (p != NULL){
      ra->next = p;
      ra = p;
      p = p->next;
      q = p;
      if (q == NULL)
         break;
      p = p->next;
      q->next = B->next;
      B->next = q;
   }
   ra->next = NULL;
   return B;
}
```

12、删除递增链表中重复的元素

```
void del(LNode *&L){
   LNode *p = L->next;
   LNode *q;
   if (p == NULL)
      return;
   while (p->next != NULL){
      q = p->next;
      if (p->data == q->data){
         p->next = q->next;
         free(q);
      }
      else
         p = p->next;
   }
}
```

13、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 fun(LNode *&A, LNode *&B){
   LNode *p = A->next, *q = B->next;
   A->next = NULL;   B->next = NULL;
   LNode *ra = A;
   while (p != NULL && q != NULL){
      if (p->data <= q->data){
         ra->next = p;
         p = p->next;
         ra = ra->next;
      }
      else{
         ra->next = q;
         q = q->next;
         ra = ra->next;
      }
   }
   if (p != NULL)
      ra->next = p;
   if (q != NULL)
      ra->next = q;
}
```

14、两个递增有序的单链表，设计算法成一个非递增有序的链表

```
void fun(LNode *&A, LNode *&B){
  LNode *p = A->next, *q = B->next, *s;
  A->next = NULL;   B->next = NULL;
  while (p != NULL && q != NULL){
    if (p->data <= q->data){
      s = p;   p = p->next;
      s->next = A->next;
      A->next = s;
    }
    else{
      s = q;   q = q->next;
      s->next = A->next;
      A->next = s;
    }
  }
  while (p != NULL){
    s = p;   p = p->next;
    s->next = A->next;
    A->next = s;
  }
  while (q != NULL){
    s = q;   q = q->next;
    s->next = A->next;
    A->next = s;
  }
}
```

15、A, B 两个单链表递增有序，从 A，B 中找出公共元素产生单链表 C，要求不破坏 A，B 结点

```
Linklist common(LNode *A, LNode *B){
  LNode *p = A->next;
  LNode *q = B->next;
  LNode *C = new LNode;
  LNode *r = C, *s;
  while (p != NULL && q != NULL){
    if (p->data < q->data)
      p = p->next;
    else if (p->data > q->data)
      q = q->next;
    else{
      s = new LNode;
      s->data = p->data;
      r->next = s;
      r = s;
      p = p->next;
      q = q->next;
    }
  }
  r->next = NULL;
  return C;
}
```

16、A, B 两个单链表递增有序，从 A，B 中找出公共元素并存放于 A 链表中

```
void Union(LNode *&A, LNode *&B){
  LNode *p = A->next, *q = B->next;
  LNode *ra = A, *u;
  while (p != NULL && q != NULL){
    if (p->data < q->data){
      u = p;   p = p->next;   free(u);
    }
    else if (p->data > q->data){
      u = q;   q = q->next;   free(u);
    }
    else{
      ra->next = p;   ra = p;
      p = p->next;   u = q;
      q = q->next;   free(u);
    }
  }
  while (p != NULL){
    u = p;   p = p->next;   free(u);
  }
  while (q != NULL){
    u = q;   q = q->next;   free(u);
  }
  ra->next = NULL;
  free(q);
}
```

17、两个序列分别为 A、B，将其存放到链表中，判断 B 是否是 A 的连续子序列

```
int seq(LNode *A, LNode *B){
   LNode *p = A->next;
   LNode *pre = p;
   LNode *q = B->next;
   while (p != NULL && q != NULL){
      if (p->data == q->data){
         p = p->next;
         q = q->next;
      }
      else{
         pre = pre->next;
         p = pre;
         q = B->next;
      }
   }
   if (q == NULL)
      return 1;
   else
      return 0;
}
```

18、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

//法一标准解

```
int find(LNode *head, int k){
   LNode *q = head->next;
   LNode *p = head;
   int i = 1;
   while (q->next != NULL){
      q = q->next;
      ++i;
      if (i >= k)
         p = p->next;
   }
   if (p == head)
      return 0;
   else
   {
      cout << p->data;
      return 1;
   }
}
```

//法二暴力解

```
int len(LNode *L){
   LNode *p = L->next;
   int n = 0;
   while (p != NULL){
      p = p->next;
      ++n;
   }
   return n;
}
int Find(LNode *L, int k){
   int i = 0, m;
   m = len(L) - k + 1;
   if (m <= 0)
      return 0;
   while (i < m) {
      L = L->next;
      ++i;
   }
   cout << L->data;
   return 1;
}
```

19、用单链表保存 m 个整数，并且|data|≤n，要求设计时间复杂度尽可能高效的算法，对于 data 绝对值相等的点，仅保留第一次出现的点

```
void fun(LNode *&head, int n){
  LNode *p = head;
  LNode *r;
  int *q = new int[n + 1];
  int m;
  for (int i = 0; i < n + 1; ++i)
    q[i] = 0;
  while (p->next != NULL){
    if (p->next->data > 0)
      m = p->next->data;
    else
      m = -p->next->data;
    if (q[m] == 0){
      q[m] = 1;
      p = p->next;
    }
    else{
      r = p->next;
      p->next = r->next;
      free(r);
    }
  }
  free(q);
}
```

20、判断带头结点的循环双链表是否对称

```
int fun(DNode *L){
  DNode *p = L->next;
  DNode *q = L->prior;
  while (p != q && q->next != p)
    if (p->data == q->data){
      p = p->next;
      q = q->prior;
    }
    else
      return 0;
  return 1;
}
```

21、有两个循环单链表，链表头指针分别为 h1,h2,试编写函数将 h2 链表接到 h1 之后，要求链接后仍保持循环链表形式

```
void link(LNode *&h1, LNode *&h2){
  LNode *p, *q;
  p = h1, q = h2;
  while (p->next != h1)
    p = p->next;
  while (q->next != h2)
    q = q->next;
  p->next = h2;
  q->next = h1;
}
```

22、设有一个带头结点的循环单链表，其结点值为正整数，设计算法反复找出链表内最小值并不断输出，并将结点从链表中删除，直到链表为空，再删除表头结点

```
void del(LNode *&L){
  LNode *p, *minp, *u;
  while (L->next != L){
    p = L->next;
    minp = L;
    while (p->next != L){
      if(p->next->data<minp->next->data)
        minp = p;
      p = p->next;
    }
    cout << minp->next->data << endl;
    u = minp->next;
    minp->next = u->next;
    free(u);
  }
  free(L);
}
```

23、判断单链表是否有环

```
int findloop(LNode *L){
   LNode *fast = L, *slow = L;
   while (fast && fast->next){
      slow = slow->next;
      fast = fast->next->next;
      if (slow == fast)
         return 1;
   }
   return 0;
}
```

24、给定一个单链表 L(a_1, a_2, a_3 ,..., a_n),将其重新排列为(a_1, a_n, a_2, a_{n-1} ,...)

```
Linklist divrev(LNode *&L){
   LNode *p = L, *q = L;
   while (q != NULL && q->next != NULL){
      p = p->next;
      q = q->next->next;
   }
   LNode *L1 = new LNode;
   L1->next = p->next;
   p->next = NULL;
   LNode *p1 = L1->next, *r;
   L1->next = NULL;
   while (p1 != NULL){
      r = p1->next;
      p1->next = L1->next;
      L1->next = p1;
      p1 = r;
   }
   return L1;
}

void merge(LNode *&L)
{
   LNode *r, *s;
   LNode *L1 = divrev(L);
   LNode *p = L->next, *q = L1->next;
   L1->next = NULL;
   while (q != NULL)
   {
      r = p->next;
      s = q->next;
      p->next = q;
      q->next = r;
      p = r;
      q = s;
   }
}
```

栈的结构体：

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

队列的结构体

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

1、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 reverse(stack &s, queue &q)
{
   while (q.front != q.rear)
      push(s, dequeue(q));
   while (s.top != -1)
      enqueue(q, pop(s));
}
```

2、判断单链表的全部 n 个字符是否中心对称

```
int fun(LNode *L){
    int n = 0, j;
    stack s;   Init(s);
    LNode *p = L->next;
    while (p != NULL){
        ++n;
        p = p->next;
    }
    p = L->next;
    for (j = 0; j < n / 2; ++j){
        push(s, p->data);   p = p->next;
    }
    if (n % 2 != 0)
        p = p->next;
    while (p != NULL){
        if (pop(s) == p->data)
            p = p->next;
        else
            return 0;
    }
    return 1;
}
```

3、两个栈 s1,s2 都采用顺序存储，并共享一个存储区[0,...,maxsize-1]。采用栈顶相向，迎面增长的存储方式，设计 s1,s2 入栈和出栈的操作。

```
bool push(stack &s, int i, int x){
    if (i < 0 || i > 1)
        return false;
    if (s.top[1] - s.top[0] == 1)
        return false;
    switch (i){
    case 0:
        s.data[++s.top[0]] = x;   break;
    case 1:
        s.data[--s.top[1]] = x;   break;
    }
    return true;
}
```

```
int pop(stack &s, int i){
    if (i < 0 || i > 1)
        return -1;
    switch (i){
    case 0:
        if (s.top[0] == -1)
            return -1;
        else
            return s.data[s.top[0]--];
        break;
    case 1:
        if (s.top[1] == maxsize)
            return -1;
        else
            return s.data[s.top[1]++];
        break;
    }
    return 0;
}
```

4、判断一个表达式中括号是否配对(假设只包含圆括号)

```
bool fun(stack &s, string str){
  int i = 0;
  while (str[i] != '\0'){
    switch (str[i]){
    case '(':
      push(s, '(');   break;
    case ')':
      if (pop(s) != '(')
        return false;
      break;
    }
    ++i;
  }
  if (s.top == -1)
    return true;
  else
    return false;
}
```

5、假设一个序列为 HSSHHHS，运用栈的知识，编写算法将 S 全部提到 H 之前，即为 SSSHHHH

```
void fun(string S){
  stack s;
  Init(s);
  string newS = "";
  int i = 0, j = 0;
  while (S[i] != '\0'){
    if (S[i] == 'H')
      push(s, S[i++]);
    else
      newS[j++] = S[i++];
  }
  while (s.top != -1)
    newS[j++] = pop(s);
  i = 0;
  while (newS[i] != '\0')
    cout << newS[i++];
}
```

6、利用一个栈实现以下递归函数的非递归计算

$$P_n(x) = \begin{cases} 1 & n=0 \\ 2x & n=1 \\ 2xP_{n-1}(x)-2(n-1)P_{n-2}(x) & n>1 \end{cases}$$

```
struct stack{
  int no;
  double val;
} st[maxsize];//结构体

double fun(int n, double x){
  int top = -1, i;
  double fv1 = 1, fv2 = 2 * x;
  for (i = n; i >= 2; i--){
    top++;
    st[top].no = i;
  }
  while (top >= 0){
st[top].val=2*x*fv2-2*(st[top].no-1)*fv1;
    fv1 = fv2;
    fv2 = st[top].val;
    top--;
  }
  if (n == 0)
    return fv1;
  return fv2;
}
```

7、KMP 算法

```
void Next(char pattern[], int next[], int n){
    next[0] = 0;
    int i = 1, len = 0;
    while (i < n){
        if (pattern[i] == pattern[len])
            next[i++] = ++len;
        else{
            if (pattern[i] == pattern[0]){
                next[i++] = 1;
                len = 1;
            }
            else{
                next[i++] = 0;
                len = 0;
            }
        }
    }
    for (int j = n - 1; j > 0; --j)
        next[j] = next[j - 1];
    next[0] = -1;
}
```

```
void kmp(char pattern[], char test[]){
    int n = len(pattern);
    int m = len(test);
    int *next = new int[n];
    get_next(pattern, next, n);
    int i = 0, j = 0;
    while (i < m){
        if (j == n - 1 && test[i] == pattern[j]){
            cout << "匹配成功!,位置为：" ;
            cout<< i - j << endl;
            return;
        }
        if (test[i] != pattern[j]){
            j = next[j];
            if (j != -1)
                continue;
        }
        ++i;
        ++j;
    }
    cout << "匹配失败" << endl;
}
```

树的结构体：

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

1、计算二叉树中所有结点个数

法一：

```
int count(BTNode *p){
    int n1, n2;
    if (p == NULL)
        return 0;
    else{
        n1 = count(p->lchild);
        n2 = count(p->rchild);
        return n1 + n2 + 1;
    }
}
```

法二：

```
void count2(BTNode *p, int &n){
    if (p != NULL){
        ++n;
        count2(p->lchild, n);
        count2(p->rchild, n);
    }
}
```

2、计算二叉树中所有叶子节点的个数

法一：

```
int count(BTNode *p){
   int n1, n2;
   if (p == NULL)
      return 0;
   if( !p->lchild&& !p->rchild)
      return 1;
   else{
      n1 = count(p->lchild);
      n2 = count(p->rchild);
      return n1 + n2;
   }
}
```

法二：

```
void count2(BTNode *p, int &n){
   if (p != NULL){
  if(p->lchild==NULL&& p->rchild==NULL)
         ++n;
      count2(p->lchild, n);
      count2(p->rchild, n);
   }
}
```

3、计算二叉树中所有双分支的节点个数

```
int count(BTNode *p){
   int n1, n2;
   if (p == NULL)
      return 0;
   if (p->lchild && p->rchild){
      n1 = count(p->lchild);
      n2 = count(p->rchild);
      return n1 + n2 + 1;
   }
   else{
      n1 = count(p->lchild);
      n2 = count(p->rchild);
      return n1 + n2;
   }
}
```

4、计算二叉树的深度

法一：

```
int getDepth(BTNode *p){
   int LD, RD;
   if (p == NULL)
      return 0;
   else{
      LD = getDepth(p->lchild);
      RD = getDepth(p->rchild);
      return (LD > RD ? LD : RD) + 1;
   }
}
```

法二：

```
void getDepth2(BTNode *p, int &n, int
&max){
   if (p != NULL){
      ++n;
      if (n >= max)
         max = n;
      getDepth2(p->lchild, n, max);
      getDepth2(p->rchild, n, max);
      --n;
   }
}
```

5、(a-(b+c))*(d/e)存储在二叉树，遍历求值

```
int comp(BTNode *p){
   int A, B;
   if (p == NULL)
      return 0;
   if (p != NULL){
      if(p->lchild&&p->rchild){
         A = comp(p->lchild);
         B = comp(p->rchild);
         return op(A, B, p->data);
      }
      else
         return p->data - '0';
   }
   return -100;
}
```

6、找出二叉树中最大值的点

```
void Max ( BTNode *p, int &max ){
   if ( p !=NULL ){
      if ( p->data>max )
         max=p->data ;
      Max ( p->lchild, max ) ;
      Max ( p->rchild, max ) ;
   }
}
```

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）

```
int fun(BTNode *T1, BTNode *T2){
   int left, right;
   if (T1 == NULL && T2 == NULL)
      return 1;
   if (T1 == NULL || T2 == NULL)
      return 0;
   else{
      left = fun(T1->lchild, T2->lchild);
      right = fun(T1->rchild, T2->rchild);
      return left && right;
   }
}
```

8、把二叉树所有节点左右子树交换

```
void swap(BTNode *p){
   if (p != NULL){
      swap(p->lchild);
      swap(p->rchild);
      BTNode *temp = p->lchild;
      p->lchild = p->rchild;
      p->rchild = temp;
   }
}
```

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

```
void fun(BTNode *p,BTNode *&q,int key)
{
   if (p != NULL){
      if (p->data == key)
         q = p;
      else{
         fun(p->lchild, q, key);
         fun(p->rchild, q, key);
      }
   }
}
```

10、输出先序遍历第 k 个结点的值

```
void trave(BTNode *p, int k, int &n){
   if (p != NULL){
      ++n;
      if (k == n){
         cout << p->data;
         return;
      }
      trave(p->lchild, k, n);
      trave(p->rchild, k, n);
   }
}
```

11、求二叉树中值为 x 的层号

```
void fun(BTNode *p, char x, int L){
  if (p != NULL){
    if (p->data == x)
      cout << "所在层数为:" << L;
    fun(p->lchild, x, L + 1);
    fun(p->rchild, x, L + 1);
  }
}//或者不用 L 传参，使用++L，--L
```

12、树中元素为 x 的结点，删除以它为根的子树

```
void del(BTNode *&bt, char key){
  if (bt != NULL){
    if (bt->data == key){
      bt = NULL;
      return;
    }
    del(bt->lchild, key);
    del(bt->rchild, key);
    return;
  }
}
```


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）

```
void link(BTNode *p, BTNode *&head, BTNode *&tail){
  if (p != NULL){
    if (!p->lchild && !p->rchild){
      if (head == NULL){
        head = p;
        tail = p;
      }
      else{
        tail->rchild = p;
        tail = p;
      }
    }
    link(p->lchild, head, tail);
    link(p->rchild, head, tail);
  }
}
```

14、输出根节点到每个叶子结点的路径

```
void allpath(BTNode *p, char pathstack[], int top){
  if (p != NULL){
    pathstack[top] = p->data;
    if (!p->lchild && !p->rchild)
      for (int i = 0; i <= top; ++i)
        cout << pathstack[i] << " ";
    cout << endl;
    allpath(p->lchild, pathstack, top + 1);
    allpath(p->rchild, pathstack, top + 1);
  }
}
```

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

```
void change(char pre[],int L1,int R1,char post[],int L2,int R2){
  if (L1 <= R1){
    post[R2] = pre[L1];
    change(pre, L1+1, (L1+R1+1)/2, post, L2, (L2+R2-1)/2);
    change(pre,(L1+R1+1)/2+1,R1,post,(L2+R2-1)/2+1,R2-1);
  }
}
```

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

```
BiTree create(char A[], char B[], int L1, int R1, int L2, int R2){
  BTNode *root = new BTNode;   root->data = A[L1];   int i;
  for (i = L2; B[i] != root->data; i++);
  if (i > L2)
    root->lchild = create(A, B, L1 + 1, i - L2 + L1, L2, i - 1);
  else
    root->lchild = NULL;
  if (i < R2)
    root->rchild = create(A, B, i - L2 + L1 + 1, R1, i + 1, R2);
  else
    root->rchild = NULL;
  return root;
}
```

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

```
BiTree create(char A[], int i, int n){
  if (i < n){
    BTNode *t = new BTNode;   t->data = A[i];
    t->lchild = create(A, 2 * i + 1, n);
    t->rchild = create(A, 2 * i + 2, n);
    return t;
  }
  return NULL;
}
```

18、增加一个指向双亲节点的 parent 指针，输出所有节点到根节点的路径

```
typedef struct BTNode{
    char data;
    struct BTNode *lchild, *rchild, *parent;
} BTNode, *BiTree;
void fun ( BTNode *p, BTNode *q ){
    if ( p !=NULL ){
        p->parent=q ;
        q=p ;
        fun ( p->lchild, q ) ;
        fun ( p->rchild, q ) ;
    }
}
void printpath(BTNode *p){
    while (p != NULL){
        cout << p->data << " ";
        p = p->parent;
    }
}
void allpath(BTNode *p){
    if (p != NULL){
        printpath(p);
        allpath(p->lchild);
        allpath(p->rchild);
    }
}
```

19、先序非递归遍历二叉树

法一：

```
void Nonpre(BTNode *bt){
    BTNode *Stack[maxsize];
    int top = -1;
    if (bt != NULL){
        Stack[++top] = bt;
        while (top != -1){
            bt = Stack[top--];
            cout << bt->data;
            if (bt->rchild != NULL)
                Stack[++top] = bt->rchild;
            if (bt->lchild != NULL)
                Stack[++top] = bt->lchild;
        }
    }
}
```

法二：

```
void Nonpre2(BTNode *bt){
    BTNode *Stack[maxsize];
    int top = -1;
    while (bt || top != -1){
        if (bt != NULL){
            cout << bt->data;
            Stack[++top] = bt;
            bt = bt->lchild;
        }
        else{
            bt = Stack[top--];
            bt = bt->rchild;
        }
    }
}
```

20、中序非递归遍历二叉树

```
void Nonin(BTNode *bt){
   BTNode *Stack[maxsize];
   int top = -1;
   while (top != -1 || bt){
      if (bt != NULL){
         Stack[++top] = bt;
         bt = bt->lchild;
      }
      else{
         bt = Stack[top--];
         cout << bt->data << " ";
         bt = bt->rchild;
      }
   }
}
```

21、后序非递归遍历二叉树

法一：

```
void Nonpost(BTNode *bt){
   BTNode *St[maxsize], *tag = NULL;
   int top = -1;
   while (bt || top != -1){
      if (bt != NULL){
         St[++top] = bt;
         bt = bt->lchild;
      }
      else{
         bt = St[top];
         if (bt->rchild && bt->rchild != tag)
            bt = bt->rchild;
         else{
            bt = St[top--];
            cout << bt->data << " ";
            tag = bt;
            bt = NULL;
         }
      }
   }
}
```

法二：

```
void Nonpost2(BTNode *bt){
   if (bt != NULL){
      BTNode *Stack1[maxsize];
      BTNode *Stack2[maxsize];
      int top1 = -1, top2 = -1;
      Stack1[++top1] = bt;
      while (top1 != -1){
         bt = Stack1[top1--];
         Stack2[++top2] = bt;
         if (bt->lchild != NULL)
            Stack1[++top1] = bt->lchild;
         if (bt->rchild != NULL)
            Stack1[++top1] = bt->rchild;
      }
      while (top2 != -1){
         bt = Stack2[top2--];
         cout << bt->data << " ";
      }
   }
}
```

22、在二叉树中查找值为 x 的结点，打印出值为 x 的所有祖先

```
void Nonpost(BTNode *bt, char x){
  BTNode *St[maxsize], *tag = NULL;
  int top = -1;
  while (bt || top != -1){
    if (bt != NULL){
      St[++top] = bt;
      bt = bt->lchild;
    }
    else{
      bt = Stack[top];
      if (bt->rchild && bt->rchild != tag)
        bt = bt->rchild;
      else{
        bt = St[top--];
        if (bt->data == x)
          while (top != -1)
            cout << St[top--]->data;
        tag = bt;
        bt = NULL;
      }
    }
  }
}
```

23、找到 p 和 q 最近公共祖先结点 r

```
BiTree Nonpost(BTNode *bt, BTNode *p,
BTNode *q){
  BTNode *St[maxsize], *tag = NULL;
  BTNode *s1[maxsize], *s2[maxsize];
  int top = -1, top1 = -1, top2 = -1;
  while (bt || top != -1){
    if (bt != NULL){
      St[++top] = bt;
      bt = bt->lchild;
    }
    else{
      bt = St[top];
      if (bt->rchild && bt->rchild != tag)
        bt = bt->rchild;
      else{
        bt = St[top--];
        if (bt == p){
          int temp = top;
          while (temp != -1)
            s1[++top1] = St[temp--];
        }
        if (bt == q){
          int temp = top;
          while (temp != -1)
            s2[++top2] = St[temp--];
        }
        tag = bt;
        bt = NULL;
      } //else
    } //else
  } //while
  for (int i = 0; i < top1; ++i)
    for (int j = 0; j < top2; ++j)
      if (s1[i] == s2[j])
        return s1[i];
  return NULL;
}
```

24、层次遍历

```
void level(BTNode *p){
  int front = 0, rear = 0;
  BTNode *que[maxsize];
  if (p != NULL){
    que[++rear] = p;
    while (front != rear){
      p = que[++front];
      cout << p->data << " ";
      if (p->lchild != NULL)
        que[++rear] = p->lchild;
      if (p->rchild != NULL)
        que[++rear] = p->rchild;
    }
  }
}
```

25、试给出自下而上从右到左的层次遍历

```
void level(BTNode *p){
  BTNode *stack[maxsize];
  int top = -1;
  int front = 0, rear = 0;
  BTNode *que[maxsize];
  if (p != NULL){
    que[++rear] = p;
    while (front != rear){
      p = que[++front];
      stack[++top] = p;
      if (p->lchild != NULL)
        que[++rear] = p->lchild;
      if (p->rchild != NULL)
        que[++rear] = p->rchild;
    }
  }
  for (int i = top; i > -1; --i)
    cout << stack[i]->data << " ";
}
```

26、求解二叉树的宽度

```
typedef struct{
  BTNode *p;   int lno;
} St[maxsize];
int width2(BTNode *boot){
  St que;
  int front = 0, rear = 0;
  int Lno, i = 1, max = 0;
  if (boot != NULL){
    que[++rear].p = boot;
    que[rear].lno = 1;
    while (front != rear){
      BTNode *q = que[++front].p;
      Lno = que[front].lno;
      if (q->lchild != NULL){
        que[++rear].p = q->lchild;
        que[rear].lno = Lno + 1;
      }
      if (q->rchild != NULL){
        que[++rear].p = q->rchild;
        que[rear].lno = Lno + 1;
      }
    }
    while (i <= rear){
      int n = 0;
      int k = que[i].lno;
      while (i<=rear && que[i].lno == k){
        ++i;
        ++n;
      } //while
      if (n > max)
        max = n;
    } //while
  } //if
  return max;
} //法一
void LevelWidth(BiTree T, int a[], int h){
  if (T != NULL){
    a[h] += 1;
    LevelWidth(T->lchild, a, h + 1);
    LevelWidth(T->rchild, a, h + 1);
  }
}
int width(BiTree T){
  int a[maxsize], h = 1;
  for (int i = 0; i <= maxsize; i++)
    a[i] = 0;
  LevelWidth(T, a, h);
  int wid = a[0];
  for (int i = 1; i <= maxsize; i++)
    if (a[i] > wid)
      wid = a[i];
  return wid;
} //法二
```

27、用层次遍历求解二叉树的高度

```
int depth(BTNode *boot){
  if (boot == NULL)
    return 0;
  int front = 0, rear = 0;
  int last = 1, level = 0;
  BTNode *q[maxsize];
  q[++rear] = boot;
  BTNode *p;
  while (front < rear){
    p = q[++front];
    if (p->lchild)
      q[++rear] = p->lchild;
    if (p->rchild)
      q[++rear] = p->rchild;
    if (front == last){
      level++;
      last = rear;
    }
  }
  return level;
}
```

28、判断二叉树是否为完全二叉树

```
bool fun(BTNode *p){
  BTNode *q[maxsize];
  int front = 0, rear = 0;
  if (p == NULL)
    return true;
  q[++rear] = p;
  while (front != rear){
    p = q[++front];
    if (p != NULL){
      q[++rear] = p->lchild;
      q[++rear] = p->rchild;
    }
    else
      while (front != rear){
        p = q[++front];
        if (p != NULL)
          return false;
      }
  }
  return true;
}
```

29、计算二叉树的带权路径长度（叶子节点）

```
int fun(BTNode *p){
  BTNode *que[maxsize];
  int front = 0, rear = 0;
  int wpl = 0, last = 1, deep = 0;
  que[++rear] = p;
  while (front != rear){
    p = que[++front];
    if (!p->lchild && !p->rchild)
      wpl += deep * (p->data - '0');
    if (p->lchild != NULL)
      que[++rear] = p->lchild;
    if (p->rchild != NULL)
      que[++rear] = p->rchild;
    if (front == last){
      deep++;
      last = rear;
    }
  }
  return wpl;
}//法一
```

```
int fun2(BTNode *p, int deep){
    int A, B;
    if (p == NULL)
        return 0;
    if (!p->lchild && !p->rchild)
        return deep * (p->data - '0');
    A = fun2(p->lchild, deep + 1);
    B = fun2(p->rchild, deep + 1);
    return A + B;
}//法二

int fun3(BTNode *p, int deep){
    if (p == NULL)
        return 0;
    int A, B;
    if (!p->lchild && !p->rchild)
        return (p->data - '0') * deep;
    ++deep;
    A = fun3(p->lchild, deep);
    B = fun3(p->rchild, deep);
    --deep;
    return A + B;
}//法三
```

30、将给定的二叉树转化为等价的中缀表达式(具体细节图在视频中会提到)

```
//法一
void fun(BTNode *p, int deep){
    if (p != NULL)
    {
        if ((p->lchild||p->rchild)&&deep>1)
            cout << "(";
        if (p->lchild != NULL)
            fun(p->lchild, deep + 1);
        cout << p->data;
        if (p->rchild != NULL)
            fun(p->rchild, deep + 1);
        if ((p->lchild||p->rchild)&&deep>1)
            cout << ")";
    }
}
```

//法二，同样可以用++deep 和--deep，具体细节见视频或者运行代码

31、建立中序线索二叉树

```
typedef struct TNode{
    char data;
    struct TNode *lchild, *rchild;
    int ltag = 0, rtag = 0;
} TNode, *iTree; //线索二叉树结构体

void InTh(TNode *&p,TNode *&pre){
    if (p != NULL){
        InTh(p->lchild, pre);
        if (p->lchild == NULL){
            p->lchild = pre;
            p->ltag = 1;
        }
        if (pre && pre->rchild == NULL){
            pre->rchild = p;
            pre->rtag = 1;
        }
        pre = p;
        InTh(p->rchild, pre);
    }
}
```

32、中序遍历线索二叉树

```
TNode *First(TNode *p){
  while (p->ltag == 0)
    p = p->lchild;
  return p;
}

TNode *Next(TNode *p){
  if (p->rtag == 0)
    return First(p->rchild);
  else
    return p->rchild;
}

void In(TNode *p){
  TNode *q = First(p);
  for (; q != NULL; q = Next(q))
    cout << q->data << " ";
}
```

33、先序建立二叉搜索树并先序遍历

```
void preTh(TNode *&p, TNode *&pre){
  if (p != NULL){
    if (p->lchild == NULL){
      p->lchild = pre;   p->ltag = 1;
    }
    if (pre && pre->rchild == NULL){
      pre->rchild = p;   pre->rtag = 1;
    }
    pre = p;
    if (p->ltag != 1)
      preTh(p->lchild, pre);
    if (p->rtag != 1)
      preTh(p->rchild, pre);
  }
}//先序建立线索二叉树

TNode *Next(TNode *p){
  if (p->ltag != 1)
    return p->lchild;
  return p->rchild;
}
void qq(TNode *p){
  TNode *q = p;
  for (; q != NULL; q = Next(q))
    cout << q->data << " ";
}//先序遍历线索二叉树
```

34、寻找中序线索二叉树的前驱结点

```
TNode *fun(TNode *T, TNode *p){
  TNode *q;
  if (p->rtag == 0)
    q = p->rchild;
  else if (p->ltag == 0)
    q = p->lchild;
  else if (p->lchild == NULL)
    q = NULL;
  else{
    while (p->ltag == 1 && p->lchild)
      p = p->lchild;
    if (p->ltag == 0)
      q = p->lchild;
    else
      q = NULL;
  }
  return q;
}
```

35、用孩子兄弟表示法求树所有叶子结点个数

```
int fun(BTNode *p){
    if (p == NULL)
        return 0;
    if (p->lchild == NULL)
        return fun(p->rchild) + 1;
    return fun(p->lchild)+fun(p->rchild);
}
```

36、用孩子兄弟表示法求树的高度

```
void fun(BTNode *T, int n, int &max){
    if (max < n)
        max = n;
    if (T->lchild != NULL)
        fun(T->lchild, n + 1, max);
    if (T->rchild != NULL)
        fun(T->rchild, n, max);
}
```

37、已知一棵树的层次序列和每个节点的度，编写算法构造此树的孩子兄弟链表。

```
void create(CBNode *&T, char e[], int degree[], int n){
    CBNode *p = new CBNode[maxsize];
    int i, j, d, k = 0;
    for (i = 0; i < n; ++i){
        p[i].data = e[i];
        p[i].lchild = NULL;
        p[i].rchild = NULL;
    }
    for (i = 0; i < n; ++i){
        d = degree[i];
        if (d){
            p[i].lchild = &p[++k];
            for (j = 2; j <= d; ++j){
                ++k;
                p[k - 1].rchild = &p[k];
            }
        }
    }
    T = p;
}
```

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

```
void search(Sqlist &L, int x){
    int low = 0, high = L.length - 1;
    int mid, temp, i;
    while (low <= high){
        mid = (low + high) / 2;
        if (L.data[mid] == x)
            break;
        else if (L.data[mid] > x)
            high = mid - 1;
        else
            low = mid + 1;
    }
    if(L.data[mid]==x&&mid!=L.length-1){
        temp = L.data[mid];
        L.data[mid] = L.data[mid + 1];
        L.data[mid + 1] = temp;
    }
    if (low > high){
        for (i = L.length - 1; i > high; --i)
            L.data[i + 1] = L.data[i];
        L.data[i + 1] = x;
        L.length++;
    }
}
```

2、找到单链表中值为 key 的元素，将其与前一个结点位置互换。

```
void fun(LNode *&L, int k){
  LNode *l = L;
  if (l->next->data == k)
    return;
  while (l->next->next)
    if (l->next->next->data != k)
      l = l->next;
    else
      break;
  if (l->next->next != NULL){
    LNode *p = l->next;
    LNode *q = p->next;
    p->next = q->next;
    l->next = q;
    q->next = p;
  }
}
```

3、在顺序表中二分查找值为 key 的元素

```
int bi(Sqlist L, int key, int low, int high){
  if (low > high)
    return -1;
  int mid = (low + high) / 2;
  if (L.data[mid] < key)
    bi(L, key, mid + 1, high);
  else if (L.data[mid] > key)
    bi(L, key, low, mid - 1);
  else
    return mid + 1;
}//法一，递归

int binsearch(Sqlist L, int key){
  int low = 0, high = L.length - 1, mid;
  while (low <= high){
    mid = (low + high) / 2;
    if (L.data[mid] == key)
      return mid + 1;
    else if (L.data[mid] < key)
      low = mid + 1;
    else
      high = mid - 1;
  }
  return -1;
}//法二
```

4、判断给定二叉树是否是二叉(搜索)排序树

```
int pre = -INT_MAX;
int JudgeBST(BTNode *root){
  if (root == NULL)
    return 1;
  int a = JudgeBST(root->lchild);
  if (root->data <= pre || a == 0)
    return 0;
  else
    pre = root->data;
  int b = JudgeBST(root->rchild);
  return b;
}
```

5、寻找二叉排序树中最大值和最小值

```
BiTree Min(BTNode *bt){
  while (bt->lchild != NULL)
    bt = bt->lchild;
  return bt;
}
BiTree Max(BTNode *bt){
  while (bt->rchild != NULL)
    bt = bt->rchild;
  return bt;
}
```

6、设求出指定结点在给定二叉排序树的层次

```
int level(BTNode *bt, char k){
  int n = 1;
  BTNode *t = bt;
  if (bt != NULL){
    while (t->data != k){
      if (t->data < k)
        t = t->rchild;
      else
        t = t->lchild;
      ++n;
    }
  }
  return n;
}
```

7、输出二叉搜索树中所有值大于 key 的结点

```
void fun(BTNode *T, char key){
  if (T != NULL){
    if (T->lchild != NULL)
      fun(T->lchild, key);
    if (T->data >= key)
      cout << T->data;
    if (T->rchild != NULL)
      fun(T->rchild, key);
  }
}
```

8、判断一个二叉树是否为平衡二叉树

```
void fun(BTNode *bt, int &balance, int &h){
  int bl = 0, br = 0, hl = 0, hr = 0;
  if (bt == NULL){
    h = 0;
    balance = 1;
  }
  else if (!bt->lchild && !bt->rchild){
    h = 1;
    balance = 1;
  }
  else{
    fun(bt->lchild, bl, hl);
    fun(bt->rchild, br, hr);
    h = (hl > hr ? hl : hr) + 1; //不可去掉！
    if (abs(hl - hr) < 2)
      balance = bl && br;
    else
      balance = 0;
  }
}
```

图的结构体

邻接表存储：

```
typedef struct ArcNode
{
  int adjvex; //边所指向节点的位置
  struct ArcNode *nextarc;
} ArcNode, *Node; //边结点结构体

typedef struct
{
  int data;
  ArcNode *firstarc;
} Vnode; //顶点结构体

typedef struct
{
  Vnode adjlist[maxsize];
  int numver, numedg;
} AGraph;
```

邻接矩阵存储：

```
typedef struct
{
  char verticle[maxsize];
  int Edge[maxsize][maxsize];
  int numver, numedg;
} mgraph;
```

头插法建立图(邻接表结构体)

```
AGraph *aaaa(int v, int e){
   AGraph *G = new AGraph;
   G->numver = v;
   G->numedg = e;
   for (int i = 0; i < v; ++i)
      G->adjlist[i].firstarc = NULL;
   for (int i = 0; i < e; ++i){
      int v1, v2;
      cin >> v1;
      cin >> v2;
      ArcNode *p = new ArcNode;
      p->adjvex = v2;
      p->nextarc = G->adjlist[v1].firstarc;
      G->adjlist[v1].firstarc = p;
      ArcNode *q = new ArcNode;
      q->adjvex = v1;
      q->nextarc = G->adjlist[v2].firstarc;
      G->adjlist[v2].firstarc = q;
   }
   return G;
}
```

1、已知无向连通图 G 由顶点集 V 和边集 E 组成，|E|>0,当 G 中度为奇数的顶点个数为不大于 2 的偶数时，G 存在包含所有边且长度为|E|的路径（称为 EL 路径）。设图 G 采用邻接矩阵存储，设计算法判断图中是否存在 EL 路径，若存在返回 1，否则返回 0。

```
int IsExistEL(mgraph G){
   int degree, i, j, count = 0;
   for (i = 0; i < G.numver; ++i){
      degree = 0;
      for (j = 0; j < G.numver; ++j)
         degree += G.Edge[i][j];
      if (degree % 2 != 0)
         count++;
   }
   if (count == 0 || count == 2)
      return 1;
   else
      return 0;
}
```

2、图的广度优先遍历（BFS）

```
void BFS(AGraph *G, int v, int visit[]){
   for (int i = 0; i < G->numver; ++i)
      visit[i] = 0;
   int que[maxsize];
   int front = 0, rear = 0;
   cout << v << " ";
   visit[v] = 1;
   que[++rear] = v;
   while (front != rear){
      int v = que[++front];
      ArcNode *p = G->adjlist[v].firstarc;
      while (p != NULL){
         if (visit[p->adjvex] == 0){
            cout << p->adjvex << " ";
            visit[p->adjvex] = 1;
            que[++rear] = p->adjvex;
         }
         p = p->nextarc;
      }
   }
}
```

3、利用 BFS 求无向图的最短路径

```
void BFSmin(AGraph *G, int v, int d[]){
    int visit[maxsize];
    for (int i = 0; i < G->numver; ++i)
        d[i] = INT16_MAX;
    for (int i = 0; i < G->numver; ++i)
        visit[i] = 0;
    int que[maxsize];
    int front = 0, rear = 0;
    visit[v] = 1;
    d[v] = 0;
    que[++rear] = v;
    while (front != rear){
        int v = que[++front];
        ArcNode *p = G->adjlist[v].firstarc;
        while (p != NULL){
            if (visit[p->adjvex] == 0){
                d[p->adjvex] = d[v] + 1;
                visit[p->adjvex] = 1;
                que[++rear] = p->adjvex;
            }
            p = p->nextarc;
        }
    }
}
```

4、图的深度优先遍历（DFS）

```
void DFS(AGraph *G, int v, int visit[]){
    visit[v] = 1;
    cout << v << " ";
    ArcNode *p = G->adjlist[v].firstarc;
    while (p != NULL)
    {
        if (visit[p->adjvex] == 0)
            DFS(G, p->adjvex, visit);
        p = p->nextarc;
    }
}
```

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）

```
bool DFS1(AGraph *G, int i, int j){
    int k, visit[maxsize];
    for (k = 0; k < G->numver; ++k)
        visit[k] = 0;
    DFS(G, i, visit);
    if (visit[j] == 1)
        return true;
    else
        return false;
}
```

6、设计算法判断无向图是否是一棵树

```
void fun(AGraph *G, int v, int &vn, int
&en, int visit[]){
    visit[v] = 1;
    ++vn;
    ArcNode *p = G->adjlist[v].firstarc;
    while (p != NULL){
        ++en;
        if (visit[p->adjvex] == 0)
            fun(G, p->adjvex, vn, en, visit);
        p = p->nextarc;
    }
}

bool GisTree(AGraph *G){
    int vn = 0, en = 0;
    int visit[maxsize];
    for (int i = 0; i < G->numver; ++i)
        visit[i] = 0;
    fun(G, 1, vn, en, visit);
    if (vn == G->numver && (G->numver -
1) == en / 2)
        return true;
    else
        return false;
}
```

7、设计算法，求无向连通图距顶点 v 最远的一个结点（即路径长度最大）

```
int BFS(AGraph *G, int v){
    ArcNode *p;
    int j, que[maxsize];
    int front = 0, rear = 0;
    int visit[maxsize];
    for (int i = 0; i < G->numver; ++i)
        visit[i] = 0;
    que[++rear] = v;
    visit[v] = 1;
    while (front != rear){
        j = que[++front];
        p = G->adjlist[j].firstarc;
        while (p != NULL){
            if (visit[p->adjvex] == 0){
                visit[p->adjvex] = 1;
                que[++rear] = p->adjvex;
            }
            p = p->nextarc;
        }
    }
    return j;
}
```

8、写出深度优先遍历的非递归算法

```
void DFS1(AGraph *G, int v){
    int visit[maxsize];
    for (int i = 0; i < G->numver; ++i)
        visit[i] = 0;
    int stack[maxsize];
    int top = -1;
    cout << v << " ";
    visit[v] = 1;
    stack[++top] = v;
    while (top != -1){
        int k = stack[top];
        ArcNode *p = G->adjlist[k].firstarc;
        while (p && visit[p->adjvex] == 1)
            p = p->nextarc;
        if (p == NULL)
            --top;
        else{
            cout << p->adjvex << " ";
            visit[p->adjvex] = 1;
            stack[++top] = p->adjvex;
        }
    }
}
```

9、输出无向图点 u 到点 v 的所有路径

```
void findpath(AGraph *G, int u, int v,
int path[], int d, int visit[]){
    int w, i;
    path[++d] = u;
    visit[u] = 1;
    if (u == v){
        cout << endl;
        for (int i = 0; i <= d; ++i)
            cout << path[i] << " ";
    }
    ArcNode *p = G->adjlist[u].firstarc;
    while (p != NULL){
        w = p->adjvex;
        if (visit[w] == 0)
            findpath(G, w, v, path, d, visit);
        p = p->nextarc;
    }
    visit[u] = 0;
}
```

10、求无向图的连通分量个数：

```
int func(AGraph *G){
   int visit[maxsize];
   for (int i = 0; i < G->numver; ++i)
      visit[i] = 0;
   int count = 0;
   for (int j = 0; j < G->numver; ++j)
      if (visit[j] == 0){
         DFS(G, j, visit);   count++;
      }
   return count;
}
```

11、邻接表转化成邻接矩阵

```
void invert(mGraph G1, AGraph *g2){
     G1.numver= g2.numver ;
     G1.numedg = g2.numedg ;
     ArcNode *p ;
     for (int i = 0; i < g2.numver; i++){
          G1.verticle[i]=g2->adjlist[i] ;
          p = g2.adjlist[i].firstarc ;
          while (p != NULL){
               G1.Edge[i][p->adjvex] = 1 ;
               p = p -> nextarc;
          }
     }
}
```

12、判断无向图是否存在环（并查集）

```
void init(int parent[], int rank[]) {
   for (int i = 0; i < maxsize; ++i){
      parent[i] = -1;   rank[i] = 0; }
}
int find_root(int x, int parent[]){
   int x_root = x;
   while (parent[x_root] != -1)
      x_root = parent[x_root];
   return x_root;
}
int Union(int x, int y, int parent[], int
rank[]){
   int x_root = find_root(x, parent);
   int y_root = find_root(y, parent);
   if (x_root == y_root)
      return 1;
   else if (rank[x_root] > rank[y_root])
      parent[y_root] = x_root;
   else if (rank[x_root] < rank[y_root])
      parent[x_root] = y_root;
   else{
      rank[y_root]++;
      parent[x_root] = y_root;
   }
   return 0;
}
```

13、邻接矩阵转化成邻接表：

```
void invert(MGraph G1, ALGraph g2){
     g2.numver = G1.numver ;
     g2.numedg = G2.numedg ;
     ArcNode *p ;
     for (int i = 0; i < G1.numver; i++)
          g2.adjlist[i].firstarc = NULL ;
     for (int i = 0; i < G1.numver; i++){
     for (int j = 0; j < G1.numver; j++){
          if (G1.Edge[i][j] != 0){
               ArcNode *p= new ArcNode
               p->adjvex = j ;
               p->next=g2.adjlist[i].firstarc ;
               g2.adjlist[i].firstarc = p ;
          }
     }
     }
}
```

1、直接插入排序

```
void InsertSort(int R[], int n){
    int i, j, temp;
    for (i = 2; i <= n; ++i){
        R[0] = R[i];
        for (j = i - 1; R[0] < R[j]; --j)
            R[j + 1] = R[j];
        R[j + 1] = R[0];
    }
}  //顺序存储

void linksort(LNode *&L){
    LNode *p = L->next;
    LNode *r = p->next;
    p->next = NULL;
    p = r;
    while (p != NULL){
        r = p->next;
        LNode *pre = L;
        while (pre->next != NULL &&
                   pre->next->data < p->data)
            pre = pre->next;
        p->next = pre->next;
        pre->next = p;
        p = r;
    }
} //链式存储
```

2、折半插入排序

```
void BinInsert(int A[], int n){
    int i, j;
    int low, high, mid;
    for (i = 2; i <= n; ++i){
        A[0] = A[i];
        low = 1;
        high = i - 1;
        while (low <= high){
            mid = (low + high) / 2;
            if (A[mid] > A[0])
                high = mid - 1;
            else
                low = mid + 1;
        }
        for (j = i - 1; j >= high + 1; --j)
            A[j + 1] = A[j];
        A[high + 1] = A[0];
    }
}
```

3、希尔排序

```
void shellsort(int arr[], int n){
    int temp;
    for (int gap = n / 2; gap > 0; gap /= 2){
        for (int i = gap; i < n; ++i){
            temp = arr[i];
            int j = i;
            while(j>=gap&&arr[j-gap]>temp){
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = temp;
            cout << endl;
        }
    }
}
```

4、冒泡排序

```
void Bubblesort(int R[], int n){
    int i, j, flag, temp;
    for (i = n - 1; i >= 1; --i){
        flag = 0;
        for (j = 1; j <= i; ++j)
            if (R[j - 1] > R[j]){
                temp = R[j];
                R[j] = R[j - 1];
                R[j - 1] = temp;
                flag = 1;
            }
        if (flag == 0)
            return;
    }
}
```

5、快速排序

```
int part(int A[], int low, int high){
    int pivot = A[low];
    while (low < high){
        while (low<high &&A[high] >=pivot)
            --high;
        A[low] = A[high];
        while (low<high && A[low] <= pivot)
            ++low;
        A[high] = A[low];
    }
    A[low] = pivot;
    return low;
}

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

6、选择排序

```
void SelectSort(int R[], int n){
    int i, j, k, temp;
    for (i = 0; i < n; ++i){
        k = i;
        for (j = i + 1; j < n; ++j)
            if (R[k] > R[j])
                k = j;
        temp = R[i];
        R[i] = R[k];
        R[k] = temp;
    }
}
```

7、堆排序

```
void sift(int arr[], int low, int high){
    int i = low, j = 2 * i + 1;
    int temp = arr[i];
    while (j <= high){
        if (j < high && arr[j] < arr[j + 1])
            ++j;
        if (temp < arr[j]){
            arr[i] = arr[j];
            i = j;
            j = 2 * i + 1;
        }
        else
            break;
    }
    arr[i] = temp;
}
void heapSort(int arr[], int n){
    for (int i = n / 2 - 1; i >= 0; --i)
        sift(arr, i, n - 1);
    for (int i = n - 1; i > 0; --i){
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        sift(arr, 0, i - 1);
    }
}
```

8、归并排序

```
void fun(int A[],int low, int mid, int high){
    int *B = new int[high - low + 1];
    int i = low, j = mid + 1;
    for (int k = low; k <= high; ++k)
        B[k] = A[k];
    int k = i;
    for (; i <= mid && j <= high; ++k){
        if (B[i] <= B[j])
            A[k] = B[i++];
        else
            A[k] = B[j++];
    }
    while (j <= high)
        A[k++] = B[j++];
    while (i <= mid)
        A[k++] = B[i++];
}
void MergeSort(int A[],int low,int high){
    if (low < high){
        int mid = (low + high) / 2;
        MergeSort(A, low, mid);
        MergeSort(A, mid + 1, high);
        fun(A, low, mid, high);
    }
}
```

该资料于 2022 年 5 月 1 日更新完毕，相较于去年增加了约 50%的题目，共计 108 道题目。另外今年新增了每道题的代码运行脚本，我已上传到我的 github，链接如下：https://github.com/Petrichor-yin/DS-code/tree/master

由于我个人实习、论文等原因，今年暂时没有私信答疑，但是我还是会定期(3-4 周)进行腾讯会议，大家可以集中在那个时间段进行提问，平时有问题可以在群里互相交流讨论。

这份资料建议搭配相应讲解视频使用。目前（2022 年 5 月 5 日）视频还在更新中，购买这份资料的同学我会拉到一个群里，我每周都会录制一部分题目的讲解视频发到群里。**另外关于代码脚本的运行，我会尽快(2022-5-7 前)出一期视频来讲解如何使用(发到 b 站)。5 月 7 号之后购买的同学直接在 b 站我的频道(我头发还多还能学)观看即可。**最后，感谢大家的支持，期待你们 2023 考研上岸！