2024 考研 408

数据结构

复习笔记

【考查目标】

1. 掌握数据结构的基本概念、基本原理和基本方法。

2. 掌握数据的逻辑结构、存储结构及基本操作的实现，能够对算法进行基本的时间复杂度与空间复杂度的分析。

3. 能够运用数据结构基本原理和方法进行问题的分析与求解，具备采用 C 或 C++ 语言设计与实现算法的能力。

2023 年 10 月 28 日

目录

1 线性表

1.1 线性表的基本概念

1.2 线性表的实现

1. 顺序存储 2. 链式存储

```
typedef struct Node { int data; struct Node *next; } Node;
void listHeadInsert(Node *l, int val) { /* 头插法 */
  Node *p = (Node *) malloc(sizeof(Node)); p->data = val;
  p->next = l->next;
  l->next = p;
}
void listTailInsert(Node *l, int val) { /* 尾插法 */
  Node *p = (Node *) malloc(sizeof(Node)); p->data = val; p->next = NULL;
  Node *tail = l;
  while (tail->next != NULL) tail = tail->next;
  tail->next = p;
}
```

1.3 线性表的应用

1. 逆置
2. 双指针

...

2 栈、队列和数组

2.1 栈和队列的基本概念

2.2 栈和队列的顺序存储结构

1. 顺序栈:

```
typedef struct { int data[M], top; } Stack;
Stack *stackInit() {
  Stack *s = (Stack *) malloc(sizeof(Stack)); s->top = -1;
  return s;
}
int stackPush(Stack *s, int val) {
  if (s->top == M - 1) return 0;
  s->data[++s->top] = val;
  return 1;
}
int stackPop(Stack *s, int *val) {
  if (s->top == -1) return 0;
  *val = s->data[s->top--];
  return 1;
}
```

2. 循环队列:

```
typedef struct { int data[M], front, rear; } Queue;
Queue *queueInit() {
  Queue *q = (Queue*) malloc(sizeof(Queue)); q->front = q->rear = 0;
  return q;
}
int queueEnqueue(Queue *q, int val) {
  if (q->front == (q->rear + 1) % M) return 0;
  q->data[q->rear] = val; q->rear = (q->rear + 1) % M;
  return 1;
}
int queueDequeue(Queue *q, int *val) {
  if (q->front == q->rear) return 0;
  *val = q->data[q->front]; q->front = (q->front + 1) % M;
  return 1;
}
```

2.3 栈和队列的链式存储结构

2.4 多维数组的存储

二维数组行下标与列下标的范围分别为 $[0,h_1]$ 与 $[0,h_2]$，每个数组元素所占的存储单元为 L，
行优先：$LOC(a_{i,j})=LOC(a_{0,0})+[i\times(h_2+1)+j]\times L$.
列优先：$LOC(a_{i,j})=LOC(a_{0,0})+[j\times(h_1+1)+i]\times L$.

2.5 特殊矩阵的压缩存储

1. 对称矩阵：

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1, i\geqslant j & \text{（下三角区和主对角元素）}\\ \dfrac{j(j-1)}{2}+i-1, i<j & \text{（上三角区元素）}\end{cases}$$

2. 上三角矩阵：

$$k=\begin{cases}\dfrac{(i-1)(2n-i+2)}{2}+j-i, & i\leqslant j \quad \text{(上三角区和主对角元素)}\\ \dfrac{n(n+1)}{2}, & i>j \quad \text{(下三角区元素)}\end{cases}$$

3. 下三角矩阵：

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1, & i\geqslant j \quad \text{(下三角区和主对角元素)}\\ \dfrac{n(n+1)}{2}, & i<j \quad \text{(上三角区元素)}\end{cases}$$

4. 三对角矩阵：

$$k=2i+j-3$$

5. 稀疏矩阵的三元组既可以采用数组存储，也可以采用十字链表法存储。

2.6 栈、队列和数组的应用

栈在递归中的应用：
通常需要借助栈将递归程序转化为非递归程序，**但不是必须的**。计算斐波那契数列仅需一个迭代过程。

3 树与二叉树

3.1 树的基本概念

3.2 二叉树

1. 二叉树的定义及其主要特征
2. 二叉树的顺序存储结构和链式存储结构

```
typedef struct BNode { int data; struct BNode *left, *right; } BNode;
```

3. 二叉树的遍历

（1）先序遍历（迭代）：

```
void preOrder(BNode *t) {
  int M = 32, top = -1; BNode *stack[M], *p = t;
  while (p || top != -1) {
    if (p) {
      visit(p); stack[++top] = p;
      p = p->left;
    } else {
      p = stack[top--];
      p = p->right;
    }
  }//end while
}
```

（2）中序遍历（迭代）：

```
void inOrder(BNode *t) {
  int M = 32, top = -1; BNode *stack[M], *p = t;
  while (p || top != -1) {
    if (p) {
      stack[++top] = p;
      p = p->left;
    } else {
      p = stack[top--]; visit(p);
      p = p->right;
    }
  }//end while
}
```

（3）后序遍历（迭代）：

```
void postOrder(BNode *t) {
  int M = 32, top = -1; BNode *stack[M], *p = t, *r = NULL;
  while (p || top != -1) {
    if (p) {
      stack[++top] = p;
      p = p->left;
    } else {
      p = stack[top];
      if (p->right && p->right != r) p = p->right;
      else {
        p = stack[top--]; visit(p);
        r = p; p = NULL;
      }
    }
  }//end while
}
```

（4）层次遍历:

```
void levelOrder(BNode *t) {
  int M = 32, front = 0, rear = 0; BNode *queue[M], *p = t;
  queue[rear] = p; rear = (rear + 1) % M;
  while (front != rear) {
    p = queue[front]; front = (front + 1) % M; visit(p);
    if (p->left != NULL) {
      queue[rear] = p->left; rear = (rear + 1) % M;
    }
    if (p->right != NULL) {
      queue[rear] = p->right; rear = (rear + 1) % M;
    }
  }//end while
}
```

（5）二叉树深度:

```
int depth(BNode *t) {
  int M = 32, front = 0, rear = 0, depth = 0;
  BNode *queue[M], *p = t, *last = t, *nlast = NULL;
  queue[rear] = p; rear = (rear + 1) % M;
  while (front != rear) {
    p = queue[front]; front = (front + 1) % M;
    if (p->left != NULL) {
      queue[rear] = p->left; rear = (rear + 1) % M;
      nlast = p->left;
    }
    if (p->right != NULL) {
      queue[rear] = p->right; rear = (rear + 1) % M;
      nlast = p->right;
    }
    if (p == last) { last = nlast; depth++; }
  }//end while
  return depth;
}
```

4. 线索二叉树的基本概念和构造

3.3 树、森林

1. 树的存储结构
2. 森林与二叉树的转换
3. 树和森林的遍历

树的遍历是指用某种方式访问树中的每个结点，且仅访问一次。
1）**先根遍历**：先访问根结点，再依次遍历根结点的每棵子树。与这棵树相应二叉树的**先序序列**相同。
2）**后根遍历**：先依次遍历根结点的每棵子树，再访问根结点。与这棵树相应二叉树的**中序序列**相同。

3.4 树与二叉树的应用

1. 哈夫曼 (Huffman) 树和哈夫曼编码
2. 并查集及其应用

4 图

4.1 图的基本概念

4.2 图的存储及基本操作

1. 邻接矩阵 2. 邻接表

```
typedef struct ArcNode {
  int adjvec;
  struct ArcNode *next;
} ArcNode;
typedef struct VNode {
  int data;
  ArcNode *first;
} VNode;
typedef struct Graph {
  int n, e;
  VNode adjlist[M];
} Graph;
```

3. 邻接多重表、十字链表

> 邻接多重表是**无向图**的一种链式存储结构，十字链表是**有向图**的一种链式存储结构。

4.3 图的遍历

1. 深度优先搜索

```
void _dfs(Graph *g, int v) {
  visit(v); visited[v] = 1;
  for (int w = firstNbr(g, v); w >= 0; w = nextNbr(g, v, w)) {
    if (!visited[w]) _dfs(g, w);
  }
}
void dfs(Graph *g) {
  for (int i = 0; i < g->n; i++) { if (!visited[i]) _dfs(g, i); }
}
```

2. 广度优先搜索

```
void _bfs(Graph *g, int v) {
  visit(v); visited[v] = 1;
  int queue[M], front = 0, rear = 0;
  queue[rear] = v; rear = (rear + 1) % M;
  while (front != rear) {
    v = queue[front]; front = (front + 1) % M;
    for (int w = firstNbr(g, v); w >= 0; w = nextNbr(g, v, w)) {
      if (!visited[w]) {
        visit(w); visited[w] = 1;
        queue[rear] = w; rear = (rear + 1) % M;
      }
    }//end for
  }//end while
}
void bfs(Graph *g) {
  for (int i = 0; i < g->n; i++) { if (!visited[i]) _bfs(g, i); }
}
```

4.4 图的基本应用

1. 最小（代价）生成树

当带权连通图的任意一个环中所包含的边的权值均不相同时，其 MST 是唯一的。

2. 最短路径 3. 拓扑排序 4. 关键路径

1. **事件 v_k 的最早发生时间** $ve(k)$：$ve(\text{源点}) = 0, \quad ve(k) = \text{Max}\{ve(j) + \text{Weight}(v_j, v_k)\}$
2. **事件 v_k 的最迟发生时间** $vl(k)$：$vl(\text{汇点}) = ve(\text{汇点}), \quad vl(k) = \text{Min}\{vl(j) - \text{Weight}(v_k, v_j)\}$
3. **活动 a_i 的最早开始时间** $e(i)$：该活动弧的起点所表示的事件的最早发生时间。
若边 $< v_k, v_j >$ 表示活动 a_i, 则有 $e(i) = ve(k)$。
4. **活动 a_i 的最迟开始时间** $l(i)$：该活动弧的终点所表示事件的最迟发生时间与该活动所需时间之差。
若边 $< v_k, v_j >$ 表示活动 a_i, 则有 $l(i) = vl(j) - \text{Weight}(v_k, v_j)$。
5. $l(i) = e(i)$ 的活动 a_i 是关键活动。

表 1: 图算法的时间复杂度总结

图算法	时间复杂度	
	邻接矩阵	邻接表
深度优先搜索	$O(V^2)$	$O(V+E)$
广度优先搜索	$O(V^2)$	$O(V+E)$
拓扑排序	$O(V^2)$	$O(V+E)$
Prim	$O(V^2)$	$O(V^2)$
Dijkstra	$O(V^2)$	$O(V^2)$

5 查找

5.1 查找的基本概念

5.2 顺序查找法

表 2: 顺序查找的平均查找长度

		查找成功	查找失败
顺序查找	无序表	$\frac{n+1}{2}$	$n+1$
	有序表		$\frac{n}{2}+\frac{n}{n+1}$

5.3 分块查找法

将长度为 n 的查找表均匀分成 b 块，每块有 s 个记录，在等概率情况下，若在块内和索引表中均采用顺序查找，平均查找长度为：ASL $= L_1 + L_S = \frac{s^2+2s+n}{2s}$.

5.4 折半查找法

折半查找选取中间结点时，可以向上或向下取整。

5.5 树型查找

1. 二叉树搜索树

删除并插入叶结点，得到的二叉树与原来的相同；删除并插入中间结点，得到的二叉树与原来的必不相同。

2. 平衡二叉树

1. 以 n_h 表示深度为 h 的平衡树中含有的最少结点树，则 $n_0 = 0, n_1 = 1, n_2 = 2, n_h = n_{h-1} + n_{h-2} + 1$.
2. 删除并插入叶结点或中间结点，得到的平衡树与原来的可能相同。

3. 红黑树

5.6 B 树及其基本操作、B+ 树的基本概念

m 阶 B 树:
1）每个结点至多 m 棵子树，至多含 $m-1$ 个关键字。
2）若根结点不是叶节点，则至少有 2 棵子树。
3）除根结点外所有非叶结点至少有 $\lceil m/2 \rceil$ 棵子树，至少含有 $\lceil m/2 \rceil - 1$ 个关键字。
4）若含 n 个关键字，高度为 h，则 $\log_m(n+1) \leqslant h \leqslant \log_{\lceil m/2 \rceil}((n+1)/2) + 1$.

5.7 散列 (Hash 表)

散列表的平均查找长度:
长度为 n 的散列表，散列函数为 $H(key) = key\%m, (设m < n)$，采用线性探查法解决冲突。
查找成功的平均查找长度：按插入的关键字序列计算即可。
查找失败的平均查找长度：查找失败时的地址可能有 m 个，若填入的关键字序列最高位地址为 $t(t > m - 1)$，则地址为 0 的关键字比较次数为 $1 + (t + 1)$(线性探查) 次，以此类推。

5.8 字符串模式匹配

5.9 查找算法的分析及应用

6 排序

6.1 排序的基本概念

6.2 直接插入排序

```
void insertSort(int *a, int n) {
  int j;
  for (int i = 1; i < n; i++) {
    if (a[i] < a[i - 1]) {
      int tmp = a[i];
      for (j = i - 1; a[j] > tmp && j >= 0; j--) a[j + 1] = a[j];
      a[j + 1] = tmp;
    }
  }//end for
}
```

6.3 折半插入排序

6.4 起泡排序 (bubble sort)

```
void bubbleSort(int *a, int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = n - 1; j > i; j--) {
      /* swap a[j-1] and a[j] */
      if (a[j - 1] > a[j]) { int tmp = a[j-1]; a[j-1] = a[j]; a[j] = tmp; }
    }
  }//end for
}
```

6.5 简单选择排序

```
void selectSort(int *a, int n) {
  for (int i = 0; i < n - 1; i++) {
    int min = i; /* select min */
    for (int j = i + 1; j < n; j++) { if (a[j] < a[min]) min = j; }
    /* swap a[i] and a[min] */
    if (min != i) { int tmp = a[i]; a[i] = a[min]; a[min] = tmp; }
  }//end for
}
```

6.6 希尔排序 (shell sort)

```
void shellSort(int *a, int n) {
  int j;
  for (int dk = n / 2; dk >= 1; dk /= 2) {
    for (int i = dk; i < n; i++) {
      if (a[i] < a[i - dk]) {
        int tmp = a[i];
        for (j = i-dk; j >= 0 && tmp < a[j]; j -= dk) a[j+dk] = a[j];
        a[j + dk] = tmp;
      }
    }//end for
  }//end for
}
```

6.7 快速排序

```
int partition(int *a, int lo, int hi) {
  int pivot = a[lo];
  while (lo < hi) {
    while (lo < hi && a[hi] >= pivot) hi--;
    a[lo] = a[hi];
    while (lo < hi && a[lo] <= pivot) lo++;
    a[hi] = a[lo];
  }
  a[lo] = pivot;
  return lo;
}
void _quick_sort(int *a, int lo, int hi) {
  if (lo < hi) {
    int pivot = partition(a, lo, hi);
    _quick_sort(a, lo, pivot - 1);
    _quick_sort(a, pivot + 1, hi);
  }
}
void quick_sort(int *a, int n) { _quick_sort(a, 0, n - 1); }
```

6.8 堆排序

6.9 二路归并排序 (merge sort)

```
void merge(int a[], int lo, int mi, int hi) {
  int ls = mi - lo, rs = hi - mi, l[ls], r[rs];
  for (int i = 0; i < ls; i++) l[i] = a[i + lo]; /* copy [lo, mi) */
  for (int i = 0; i < rs; i++) r[i] = a[i + mi]; /* copy [mi, hi) */
  int i = lo, j = mi, k = lo;
  while (i < mi && j < hi) {
    if (l[i - lo] < r[j - mi]) a[k++] = l[i++ - lo];
    else a[k++] = r[j++ - mi];
  }
  while (i < mi) a[k++] = l[i++ - lo];
  while (j < hi) a[k++] = r[j++ - mi];
}
void _mergeSort(int a[], int lo, int hi) {
  if (hi - lo > 1) {
    int mi = (lo + hi) / 2;
    _mergeSort(a, lo, mi);
    _mergeSort(a, mi, hi);
    merge(a, lo, mi, hi);
  }
}
void mergeSort(int a[], int n) { _mergeSort(a, 0, n); }
```

6.10 基数排序

6.11 外部排序

6.12 排序算法的分析和应用

选取排序方法需要考虑的因素：数据规模，元素本身信息量的大小，数据初始状态，算法的稳定性，存储结构

表 3: 各种排序算法的性质总结

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
2 路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	是

7 往年代码题

7.1 2009-42

查找带有表头结点的单链表中倒数第 k 个位置上的结点 (k 为正整数)。若查找成功，算法输出该结点的 $data$ 域的值，并返回 1；否则，只返回 0。

```
typedef struct Node { int data; struct Node *link; } Node;
int findLastK(Node *list, int k) {
  Node *p = list->link, *q = list->link; int count = 0;
  while (q != NULL) {
    if (count < k) count++;
    else p = p->link;
    q = q->link;
  }
  if (count < k) return 0;
  else { printf("%d", p->data); return 1; }
}
```

7.2 2010-42

将长为 n 的数组循环左移 p 个位置。

```
void reverse(int a[], int lo, int hi) {
  while (lo < hi) {
    int tmp = a[lo]; a[lo] = a[hi]; a[hi] = tmp; /* swap a[lo] and a[hi] */
    lo++; hi--;
  }
}
void shiftLeftP(int a[], int n, int p) {
  reverse(a, 0, p - 1);
  reverse(a, p, n - 1);
  reverse(a, 0, n - 1);
}
```

7.3 2011-42

求两个等长升序序列 a 和 b 的中位数。

```
int findMid(int a[], int b[], int n) {
  int i = 0, j = 0, k = 0;
  while (i < n && j < n) {
    k++;
    if (a[i] == b[j]) return a[i];
    if (a[i] < b[j]) { i++; if (k == n) return a[i - 1]; }
    else { j++; if (k == n) return b[j - 1]; }
  }
}
```

7.4 2012-42

求两个带有头节点的字符类型的单链表的相同后缀位置。

```
typedef struct Node { char data; struct Node *next; } Node;
int listLen(Node *list) {
  int count = 0; Node *p = list->next;
  while (p != NULL) { count++; p = p->next; }
  return count;
}
Node *findSuffix(Node *str1, Node *str2) {
  int m = listLen(str1), n = listLen(str2), ahead = abs(m - n);
  Node *p = str1->next, *q = str2->next;
  if (m > n) { for (int i = 0; i < ahead; i++) p = p->next; }
  else { for (int i = 0; i < ahead; i++) q = q->next; }
  while (p != NULL && q != NULL) {
    if (p->data == q->data) return p;
    p = p->next; q = q->next;
  }
  return NULL;
}
```

7.5 2013-41

查找数组中的主元素。

```
int findMainElem(int a[], int n) {
  int b[n], max = 0; for (int i = 0; i < n; i++) b[i] = 0;
  for (int i = 0; i < n; i++) {
    b[a[i]]++;
    if (b[max] < b[a[i]]) max = a[i];
  }
  if (b[max] <= n / 2) max = -1;
  return max;
}
```

7.6 2014-41

二叉树的带权路径长度。

```
typedef struct Node { int weight; struct Node *left, *right; } Node;
int wpl(Node *root) {
  int M = 32, front = 0, rear = 0, wpl = 0, depth = 0;
  Node *queue[M], *p = root, *l = root, *nl = NULL;
  queue[rear] = p; rear = (rear + 1) % M;
  while (front != rear) {
    p = queue[front]; front = (front + 1) % M;
    if (p->left == NULL && p->right == NULL) wpl += depth * p->weight;
    if (p->left != NULL) {
      queue[rear] = p->left; rear = (rear + 1) % M;
      nl = p->left;
    }
    if (p->right != NULL) {
      queue[rear] = p->right; rear = (rear + 1) % M;
      nl = p->right;
    }
    if (p == l) { l = nl; depth++; }
  }
  return wpl;
}
```

7.7 2015-41

删除带头节点的单链表中第二次出现的绝对值相同的节点。

```
typedef struct Node { int data; struct Node *link; } Node;
void duplicate(Node *head, int n) {
  int count[n]; for (int i = 0; i < n; i++) count[i] = 0;
  Node *p = head->link, *r = head, *q;
  while (p != NULL) {
    if (count[abs(p->data)] == 0) {
      count[abs(p->data)] = 1;
      r = p; p = p->link;
    } else {
      q = p; p = q->link; r->link = p;
      free(q);
    }
  }//end-while
}
```

7.8 2016-43

将长为 n 的数组划分为最小的 $\lfloor n/2 \rfloor$ 个元素和最大的的 $\lceil n/2 \rceil$ 个元素。

```
int partition(int a[], int n) {
  int lo = 0, hi = n - 1, llo = lo, lhi = hi, k = n/2 - 1, pivot = a[lo];
  while (1) {
    while (lo < hi) {
      while (lo < hi && pivot <= a[hi]) hi--;
      if (lo != hi) a[lo] = a[hi];
      while (lo < hi && pivot >= a[lo]) lo++;
      if (lo != hi) a[hi] = a[lo];
    }
    a[lo] = pivot;
    if (lo == k) break;
    if (lo < k) { llo = ++lo; hi = lhi; }
    else { lhi = --hi; lo = llo; }
  }
  int s1 = 0, s2 = 0;
  for (int i = 0; i < lo + 1; i++) s1 += a[i];
  for (int i = lo + 1; i < n; i++) s2 += a[i];
  return s2 - s1;
}
```

7.9 2017-41

二叉树转中缀表达式，通过括号反映操作数的计算次序。

```
typedef struct node { char data[10]; struct node *left, *right; } BTree;
/* 非满分代码，但是容易想到 */
void toExp(BTree *root) {
  if (root == NULL) return;
  int flag = 0; char ch = root->data[0];
  if (ch == '+' || ch == '-' || ch == '*' || ch == '/') flag = 1;
  if (flag == 1) printf("(");
  toExp(root->left);
  printf("%s", root->data);
  toExp(root->right);
  if (flag == 1) printf(")");
}
/* 标准答案 */
void _toExp(BTree *root, int depth) {
  if (root == NULL) return;
  if (root->left == NULL && root->right == NULL) {
    printf("%s", root->data);
    return;
  }
  if (depth > 1) printf("(");
  _toExp(root->left, depth + 1);
  printf("%s", root->data);
  _toExp(root->right, depth + 1);
  if (depth > 1) printf(")");
}
void toExp(BTree *root) {
  _toExp(root, 1);
}
```

7.10 2018-41

给定一个含 $n(\geqslant 1)$ 个整数的数组，请设计一个在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。例如，数组 {-5,3,2,3} 中未出现的最小正整数是 1；数组 {1,2,3} 中未出现的最小正整数是 4。

```
int findMin(int a[], int n) {
  int m = a[0]; /* 妈的题目也没说明最大数是 n 啊，不还得找最大数吗 */
  /* select max value in a[n] */
  for (int i = 0; i < n; i++) { if (a[i] > 0 && m < a[i]) m = a[i]; }
  int b[m + 1];
  for (int i = 0; i < m + 1; i++) b[i] = 0;
  /* mark existed value */
  for (int i = 0; i < n; i++) { if (a[i] > 0) b[a[i]] = 1; }
  int min = m + 1;
  /* find min value */
  for (int i = 1; i < m + 1; i++) { if (b[i] == 0) { min = i; break; } }
  return min;
}
```

7.11 2019-41

设线性表 $L=(a_1,a_2,a_3,\ldots,a_{n-2},a_{n-1},a_n)$ 采用带头结点的单链表保存，请设计一个空间复杂度为 $O(1)$ 且时间上尽可能高效的算法，重新排列 L 中的各结点，得到线性表 $L^{'}=(a_1,a_n,a_2,a_{n-1},a_3,an-2,\ldots)$.

```
void changeList(Node *l) {
  Node *p = l, *q = p, *r;
  while (q->next != NULL) { /* 找到中间结点 */
    p = p->next; q = q->next;
    if (q->next != NULL) q = q->next;
  }
  q = p->next; p->next = NULL;
  while (q != NULL) { /* 后半段原地逆置 */
    r = q; q = q->next;
    r->next = p->next; p->next = r;
  }
  q = p->next; p->next = NULL; p = l->next; /* 从中间断开 */
  while (q != NULL) { /* 将后半段隔一个结点插入前半段 */
    r = q; q = q->next;
    r->next = p->next; p->next = r;
    p = r->next;
  }
}
```

7.12 2020-41

定义三元组 (a,b,c)(其中 a,b,c 均为正数) 的距离 $D=|a-b|+|b-c|+|c-d|$。给定 3 个非空整数集合 $S1$、$S2$ 和 $S3$，按升序分别存储在 3 个数组中。设计一个尽可能高效的算法，计算并输出所有可能的三元组 $(a,b,c)(a\in S1,b\in S2,c\in S3)$ 中的最小距离。例如 $S1=\{-1,0,9\},S2=\{-25,-10,10,11\},S3=\{2,9,17,30,41\}$，则最小距离为 2，相应的三元组为 $(9,10,9)$。

```
int findMinDist(int s1[], int n1, int s2[], int n2, int s3[], int n3) {
  int i = 0, j = 0, k = 0, min = 0x7fffffff, d, a, b, c;
  while (i < n1 && j < n2 && k < n3) {
    a = s1[i]; b = s2[j]; c = s3[k];
    d = abs(a - b) + abs(b - c) + abs(c - a);
    if (d < min) { min = d; }
    if (a < b) { a < c ? i++ : k++; }
    else { b < c ? j++ : k++; }
  }
  return min;
}
```