南师基教

数据结构上机与实训教程

李橙 编著

DATA STRUCTURE

南京师范大学出版社
NANJING NORMAL UNIVERSITY PRESS

数据结构上机与实训教程

李橙 编著

南京师范大学出版社
NANJING NORMAL UNIVERSITY PRESS

图书在版编目(CIP)数据

数据结构上机与实训教程 / 李橙编著. —南京：南京师范大学出版社，2016.8

ISBN 978-7-5651-2605-5

Ⅰ. ①数… Ⅱ. ①李… Ⅲ. ①数据结构—高等学校—教材 Ⅳ. ①TP311.12

中国版本图书馆 CIP 数据核字(2016)第 081264 号

书　　名	数据结构上机与实训教程
编　　著	李　橙
责任编辑	倪晨娟
出版发行	南京师范大学出版社
地　　址	江苏省南京市宁海路 122 号(邮编: 210097)
电　　话	(025)83598919(总编办)　83598412(营销部)　83598297(邮购部)
网　　址	http: // www. njnup. com
电子信箱	nspzbb@ 163. com
照　　排	南京理工大学资产经营有限公司
印　　刷	启东市人民印刷有限公司
开　　本	787 毫米×960 毫米　1/16
印　　张	11.5
字　　数	181 千
版　　次	2016 年 8 月第 1 版　2016 年 8 月第 1 次印刷
书　　号	ISBN 978-7-5651-2605-5
定　　价	30.00 元

出 版 人　彭志斌

前　言

数据结构是计算机专业最基础，也是最重要的课程之一。它和C程序设计一起为学习计算机学科的其他后续课程奠定了坚实的基础。数据结构同时也是一门实践性很强的课程，要学好数据结构，首先要在学习C程序设计时打好基础，然后要理解教材中的每个知识点，还要编写一定量的代码，这样才能够熟练掌握各种数据结构的实现，并且灵活应用各种数据结构。

本书在简要描述各章知识点的基础上，详尽介绍了各章节涉及的编程实践以及实训拓展。部分章节包括知识要点、学习方法、习题和综合练习四个部分内容。知识要点部分提炼了每章节所涉及的知识点；学习方法部分总结了每章节重点与难点的学习指导办法；习题部分详细介绍了每个章节所涉及的所有程序；综合练习是对习题部分的拓展，介绍了一些教材中没有涉及的，但也会被经常用到的难度稍大的相关数据结构应用。

本书在编写的过程中得到了南京师范大学王琼教授、丁国栋先生的大力支持，同时也得到了南京师范大学泰州学院王婷婷、蒋小玉、王力、孙国庆、沈红威、万洪杰等同学的帮助，在此对他们表示衷心的感谢。

由于笔者水平有限，书中存在不足之处在所难免，敬请广大读者批评指正。

李　橙

2016年5月

目 录

第一章
线性表

线性表是一个数据集合，其中每个数据元素之间仅具有单一的前驱和后继关系。它有两种主要的存储结构：顺序表和链表。

1 顺序表

顺序表是采用顺序存储结构的线性表，即线性表的所有元素，顺序存放在地址连续的一块内存空间中。

有序顺序表是一种数据元素按照一定次序排列的顺序表。在查找、插入等算法中，若能充分利用其数据规律，则可以极大地改善其算法效率。

1.1 知识要点

1.1.1 顺序表的创建与释放算法

顺序表的创建与释放不仅要掌握数据存储的规则，更需要关注表空间的分配与回收问题。空间分配有静态和动态两种，静态分配的空间在不够用时不可追加空间，动态分配的空间在需要的时候可以追加空间并保留原有数据不变。

1.1.2 顺序表的遍历、查找算法

遍历是对顺序表中每个数据元素访问且仅访问一次。教材中对数据元素的“访问”一般指输出数据元素。实际上比较查找也是一种“访问”运算。

1.1.3 顺序表的插入、删除算法

在顺序表中插入、删除数据元素，必然引起一些数据元素的移位运算。初学者不仅应注意移位方向，对表空间的管理也应关注。

1.1.4 有序顺序表的查找算法

在有序顺序表中查找某一个元素，从前到后拿该元素与顺序表中的元素一一比较，如果比顺序表中的当前元素大，则继续比较下一个；如果相等则找到，返回当前元素的下标；否则查找失败。

1.1.5 多个顺序表之间的集合运算

顺序表常常用来表示集合。集合的一些运算，如求交集、并集、差集，应能够利用顺序表实现。

1.2 学习方法

初学者缺少内存结构的认识，大多数学生不能仔细观察数据对象的内存结构，常常混淆逻辑结构与存储结构的差别，这不仅造成了对顺序表算法学习的困难，更对后继内容的学习形成了障碍。

1. 在顺序表的创建算法中，调试观察顺序表的内存空间，及其中的每个数据元素的内存结构及地址。

2. 在顺序表的查找算法中，调试观察算法的流程、当前数据元素的变化，及查找成功、失败的返回值。

3. 在顺序表的插入、删除算法中，调试观察数据的移位操作。

4. 在两个顺序表的求交集算法中，调试观察算法流程、两个线性表中的当前元素、新表的创建过程。

1.3 习题

1.3.1 顺序表的结构定义

为了便于学习，下文将数据元素的类型设定为 int。

```
#define LIST_INIT_SIZE 20
#define LISTINCREMENT 10
Typedef struct SqList
{   int *elem;                  // 存储空间的基址
    int length;                 // 当前长度
```

```
    int listsize;                    // 当前分配的存储容量
}Sqlist;
```

1.3.2　顺序表的创建

```
Sqlist *Sqlist_build()
{   int i=1, *element;
    s=(Sqlist *)malloc(sizeof(Sqlist));
    s->elem=(int *)malloc(LIST_INIT_SIZE *sizeof(int));
    s->listsize =LIST_INIT_SIZE;
    element=s->elem;
    for(s->length=0;i++<=s->listsize && *(element-1)!=0;)
    {   printf("input a number: \n");
        scanf("%d",element);
        element++; s->length++;
    }/*当输入的数据不为0并且没有超过存储空间时,执行循环语句*/
    if(--i<=s->listsize)// 当输入的数据为0时,表长减1
     s->length--;
    return s;
}
```

1.3.3　顺序表的释放

```
void *Sqlist_free(Sqlist *s)
{   free(s->elem);
    free(s);
    s->listsize =0;
    s->length=0;
}
```

1.3.4 顺序表的插入

```
void Sqlist_insert(Sqlist * s,int position,int element)
{  int * newbase;
   if(position<1 || position>s -> length + 1) return 0;
   if(s -> length> = s -> listsize)
     newbase = (int * )(realloc(s -> elem,(s -> listsize +
LISTINCREMENT) * sizeof(size)));
   s -> elem = newbase;
   s -> listsize + = LISTINCREMENT;
   q = &(s -> elem[position - 1]);// q 为插入位置
   for(p = &(s -> elem[s -> length - 1]);p> = q; -- p)
      * (p + 1) = * p;              // 插入位置及之后的元素右移
   * q = element;                   // 插入元素
   ++ s -> listsize;                // 表长增 1
}
```

1.3.5 顺序表的删除

```
void Sqlist_delete(Sqlist * s,int position,int element)
{  if(position<1 || position>s -> length + 1) return 0;
   p = &(s -> elem[position - 1]);// p 为被删除元素的位置
   element = * p;                  // 被删除元素的值赋给 element
   q = s -> elem + s -> length - 1; // 表尾元素的位置
   for( ++ p;p< = q; ++ p)         // 被删除元素之后的元素左移
     * (p - 1) = * p;
   -- s -> length;                 // 表长减 1
}
```

1.3.6　两个有序顺序表合并

```
Sqlist *Sqlist_merge(Sqlist *s,Sqlist *q)
{  Sqlist *new1 = (Sqlist *)malloc(sizeof(Sqlist));
   int *p1, *p2, *p3;p1 = s -> elem;p2 = q -> elem;
   new1 -> elem = (int *)malloc((q -> length + s>length) * sizeof
(int));
   p3 = new1 -> elem;
   while(s -> length && q -> length)
   {  if( *p2> *p1)
      {  *p3 = *p1;
         p1 ++ ;p3 ++ ;
         s -> length -- ;
      }
      else if( *p2 == *p1)
      {  *p3 = *p1;
         p2 ++ ;p1 ++ ;p3 ++ ;
         q -> length -- ;
         s -> length -- ;
      }
      else
      {  *p3 = *p2;
         p2 ++ ;p3 ++ ;
         q -> length -- ;
      }
   }
   while(s -> length)
   {  *p3 = *p1;
      p1 ++ ;p3 ++ ;
      s -> length -- ;
   }
```

```
    while(q -> length)
    {   * p3 = * p2;
        p2 ++ ;p3 ++ ;q -> length -- ;
    }
        return new1;
}
```

2 链　　表

链表是链式存储结构的线性表。每个数据元素占据一个内存区域(称为结点),整个链表的结点是一组内存单元,可以相邻,也可以不相邻。因此,链表结点中不仅应包含自身数据,也必须包含表示线性表中先后次序的附加数据。

若链表结点的附加数据是后继结点的内存地址,则该链表被称为单向链表。单向链表中,尾结点的后继为空。

若单向链表的尾结点的后继被设为头结点的地址,则该链表被称为单向循环链表。

若链表结点的附加数据是后继结点、前驱结点的内存地址,则该链表被称为双向链表。双向链表中,尾结点的后继为空,头结点的前驱为空。

若双向链表中,尾结点的后继被设为头结点的地址,头结点的前驱为尾结点的地址,则该链表被称为双向循环链表。

2.1 知识要点

2.1.1 简单单向链表的创建、遍历算法

利用 3～4 个结点链接成一个简单链表,建立对链表存储结构的认识。利用对链表的遍历过程,建立对当前结点指针及当前结点的认识。

2.1.2 单向链表的查找算法

根据指定值,查找结点地址。在单向链表中的查找只能从首元结点开始一

个一个向后查找，找到则返回该值所在结点的地址。

目标：强化利用指针变量操作链表的能力。

2.1.3　单向链表的插入算法及创建算法

头插入算法：在单向链表的头结点之前插入新结点。

尾插入算法：在单向链表的尾结点之后插入新结点。

有序插入算法：在有序的单向链表中，在保持原有数据次序的前提下，插入新结点。

链表创建算法 1：多次调用头插入算法创建链表。特点是数据的输入次序与链表的数据次序相反。

链表创建算法 2：多次调用尾插入算法创建链表。特点是数据的输入次序与链表的数据次序相同，但时间复杂度高。

链表创建算法 3：为改善链表创建算法 2 的效率缺陷，将尾插入一个新结点的算法扩展为尾插入多个新结点的算法。

链表创建算法 4：多次调用有序插入算法创建链表。特点是目标链表为有序链表。

2.1.4　单向链表的删除算法及释放算法

删除单向链表的头结点。

删除单向链表的尾结点。

删除单向链表指定数据的结点。

利用第一种删除算法，实现链表的释放算法。

2.1.5　单向循环链表的练习

在单向循环链表中查找插入和删除某一元素应从首元结点开始一个一个进行查找与比对，由于链表的单向性，定位到该结点后的操作通常需要通过该结点前驱的指针域进行，所以在查找元素的过程中需定位当前元素的前驱。

2.1.6　双向循环链表的练习

在双向链表中进行元素的查找、插入、删除操作时，由于双向链表中不仅有指向后继的指针，还有指向前驱的指针，所以找到后进行处理时会比较方便。

2.1.7　多个链表之间的运算

利用链表表示集合，利用链表运算，实现求交集、并集、差集的集合运算。

2.2 学习方法

初学者常常难以建立对链表内存结构的认识。这需要循序渐进的实践、思考、练习的积累。

1. 在简单单向链表的创建算法中，调试观察每个结点的数据结构及地址。

2. 在单向链表的查找算法中，调试观察算法的流程、当前数据结点的变化，及查找成功、失败的返回值。

3. 在单向链表的插入、删除算法中，调试观察结点中指针数据的变化，并分步绘制相应的内存结构图。

4. 在单向链表的创建、释放算法中，调试链表的整体变化，并分步绘制相应的内存结构图。

5. 在双向链表的插入、删除算法中，调试观察结点中指针数据的变化，并分步绘制相应的内存结构图。

6. 在两个链表的求交集算法中，调试观察算法流程、两个链表中的当前元素、新链表的创建过程。

2.3 习题

2.3.1 线性链表的结构定义

为便于学习，下文将数据元素的类型设定为 int。

```
typedef struct node    // 结构体的定义
{   int data;
    struct node * next;
}LNODE, * Linklist;
```

2.3.2 线性链表的初始化

```
void Linklist_init(Linklist L)// 线性链表初始化
{    L = (Linklist)malloc(sizeof(Linklist));
     L -> next = NULL;
}
```

2.3.3　建立线性链表

```
void Linklist_build(Linklist L)
{   Linklist p1,p2;int i;
    p1 = p2 = (Linklist)malloc(sizeof(Linklist));L = p1;
    p1 = (Linklist)malloc(sizeof(Linklist));
    printf("请输入数据:");
    scanf("%d",&p1 -> data);
    while(p1 -> data)
    {   p2 -> next = p1;
        p2 = p1;
          p1 = (Linklist * )malloc(sizeof(Linklist));
          printf("请输入数据:");
          scanf("%d",&p1 -> data);
    }
    free(p1);
    p2 -> next = NULL;
}
```

2.3.4　线性链表的遍历

```
void Linklist_traverse(Linklist L)
{  Linklist P;P = L -> next;
   while(P)
   {   printf("%d\n",P -> data);
       P = P -> next;
   }
}
```

2.3.5 线性链表的查找

```
Linklist Linklist_search(Linklist L,int position)
                                                    // 返回检索到的结点
{   Linklist *P;P = L;
    While(P&&position)
    {   position--;
        P = P->next;
    }
    return P;
}
```

2.3.6 线性链表的插入(在指定位置 position 之前插入)

```
void Linklist_insertfront(Linklist L,int position)
{   Linklist p1,p;
    p = search(L,position - 1);
    p1 = (Linklist)malloc(sizeof(Linklist));
    printf("请输入数据:");
    scanf("%d",&p1->data);
    p1->next = p->next;
    p->next = p1;
}
```

2.3.7 线性链表的插入(在指定位置 position 之后插入)

```
void Linklist_insertrear(Linklist L,int position)
{   Linklist p1,p;
    p = search(L,position);
    p1 = (Linklist)malloc(sizeof(Linklist));
```

```
    printf("请输入数据:");
    scanf("%d",&p1->data);
    p1->next=p->next;
    p->next=p1;
}
```

2.3.8　线性链表的删除(删除 position 位置的结点)

```
void Linklist_delete(Linklist L,int position)
{   int i=0;
    Linklist P,Q;
    P=L;
    P=search(P,position-1);
    Q=P->next;
    P->next=Q->next;
    free(Q);
}
```

2.3.9　线性链表的逆序(有特殊头结点)

```
LNODE * LinkList_reverse(LNODE *head)
{ LNODE *NewHead,*p;
  NewHead=(LNODE *)malloc(sizeof(LNODE));
  NewHead->next=NULL;
    while(head->next!=NULL)
  { p=head->next;          head->next=p->next;
    p->next=NewHead->next;  NewHead->next=p;
  }
  free(Head);
```