CHAPTER 1

第1章 练习题参考答案

1.1 第1章 绪论

1.1.1 问答题

1. 什么是数据结构？有关数据结构的讨论涉及哪3个方面？

答：按某种逻辑关系组织起来的一组数据元素，按一定的存储方式存储于计算机中，并在其上定义了一个运算的集合，称为数据结构。

数据结构涉及以下3个方面的内容：

(1) 数据元素以及它们相互之间的逻辑关系，也称为数据的逻辑结构。

(2) 数据元素及其关系在计算机存储器内的存储表示，也称为数据的物理结构，简称为存储结构。

(3) 施加于该数据结构上的操作。

2. 简述逻辑结构与存储结构的关系。

答：在数据结构中，逻辑结构与计算机无关，存储结构是数据元素之间的逻辑关系在计算机中的表示。存储结构不仅将逻辑结构中的所有数据元素存储到计算机内存中，而且还要在内存中存储各数据元素间的逻辑关系。通常情况下，一种逻辑结构可以有多种存储结构，例如线性结构可以采用顺序存储结构或链式存储结构表示。

3. 简述数据结构中运算描述和运算实现的异同。

答：运算描述是指逻辑结构施加的操作，而运算实现是指一个完成该运算功能的算法。它们的相同点是都能完成对数据的“处理”或某种特定的操作，不同点是运算描述只是描述处理功能，不包括处理步骤和方法，而运算实现的核心是处理步骤。

4. 简述数据结构和数据类型两个概念之间的区别。

答：数据结构是指所涉及的数据元素的集合以及数据元素之间的关系，包括逻辑结构、存储结构和运算三个方面。数据类型是一组性质相同的值的集合和定义在此集合上的一组操作的总称，如 Java 语言中定义变量 i 为 short 类型，则它的取值范围为－32 768～32 767，可用的操作有＋、－、*、/和％等。可以将数据类型看成是实现了数据结构。

5. 什么是算法？算法的 5 个特性是什么？试根据这些特性解释算法与程序的区别。

答：通常算法被定义为解决某一特定任务而规定的一个指令序列。一个算法应当具有以下特性。

(1) 有穷性：一个算法无论在什么情况下都应在执行有穷步后结束。

(2) 确定性：算法的每一步都应确切地、无歧义地定义。对于每一种情况，需要执行的动作都应严格地、清晰地规定。

(3) 可行性：算法中的每一条运算都必须是足够基本的。也就是说，它们原则上都能精确地执行，甚至人们仅用笔和纸做有限次运算就能完成。

(4) 输入：一个算法必须有 0 个或多个输入。它们是算法开始运算前给予算法的量。这些输入取自于特定的对象的集合。它们可以使用输入语句由外部提供，也可以使用赋值语句在算法内给定。

(5) 输出：一个算法应有一个或多个输出，输出的量是算法计算的结果。

算法和程序不同，程序可以不满足有穷性。例如，一个操作系统在用户未使用前一直处于“等待”的循环中，直到出现新的用户事件为止。这样的系统可以无休止地运行，直到系统停机。

6. 一个算法的执行频度为$(3n^2+2n\log_2 n+4n-7)/(10n)$，其时间复杂度是多少？

答：当 n 足够大时，$T(n)\rightarrow 3n/10=0.3n$，其时间复杂度为 $O(n)$。

7. 某算法的时间复杂度为 $O(n^3)$，当 $n=5$ 时执行时间为 50s，问当 $n=15$ 时其执行时间大致是多少？

答：$T(n)=O(n^3)=m\times n^3$，当 $n=5$ 时 $T(n)=50$s，求得 $m=0.4$。当 $n=15$ 时，$T(n)=m\times n^3=0.4\times 15^3s=1350$s。

1.1.2 算法分析题

1. 分析以下算法的时间复杂度。

```
void fun(int n)
{ int x=n,y=100;
  while(y>0)
  {   if(x>100)
      {   x=x-10;
          y--;
      }
      else x++;
  }
}
```

答：本算法中的基本操作语句是while 循环体内的语句，它的执行次数与问题规模 n 无关，所以算法的时间复杂度为 $O(1)$。

2. 分析以下算法的时间复杂度。

```
void func(int n)
{ int i=1,k=100;
  while(i<=n)
  {   k++;
      i+=2;
  }
}
```

答：设 while 循环语句执行的次数为 $T(n)$，i 从 1 开始递增，最后取值为 $1+2T(n)$，有 $i=1+2T(n)\leqslant n$，即 $T(n)\leqslant(n-1)/2=O(n)$，所以该算法的时间复杂度为 $O(n)$。

3. 分析以下算法的时间复杂度。

```
void fun(int n)
{ int i=1;
  while(i<=n)
      i=i*2;
}
```

答：本算法中的基本操作是语句 $i=i*2$，设其频度为 $T(n)$，则有 $2^{T(n)}\leqslant n$，即 $T(n)\leqslant\log_2 n=O(\log_2 n)$，所以该算法的时间复杂度为 $O(\log_2 n)$。

4. 分析以下算法的时间复杂度。

```
void fun(int n)
{ int i,j,k;
  for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
      {   k=1;
          while(k<=n) k=5*k;
      }
}
```

答：本算法中的基本操作为 $k=5*k$ 语句。对于 while 循环语句：

```
k=1;
while(k<=n) k=5*k;
```

基本操作语句的执行频度为 $\log_5 n$，再加上外层的两重循环，所以本算法的时间复杂度为 $O(n^2\log_5 n)$。

1.1.3　补充的单项选择题

1. 数据结构中处理的数据一般具备某种内在联系，这是指_____。

A. 数据和数据之间存在某种关系

B. 元素和元素之间存在某种关系

C. 元素内部具有某种结构

D. 数据项和数据项之间存在某种关系

答：数据结构中的数据是数据元素的集合，元素之间存在某种关系，逻辑关系的整体构成逻辑结构。答案为 B。

2. 在数据结构中，与所使用的计算机无关的是数据的_____结构。

A. 逻辑　　B. 存储　　C. 逻辑和存储　　D. 物理

答：数据的逻辑结构独立于存储结构，一般由逻辑结构映射成存储结构。答案为 A。

3. 数据结构在计算机中的表示称为数据的_____。

A. 存储结构　　B. 抽象数据类型　　C. 顺序结构　　D. 逻辑结构

答：数据的逻辑结构在计算机中的表示称为存储结构。答案为 A。

4. 在计算机中存储数据时不仅要存储各数据元素的值，而且还要存储_____。

A. 数据的处理方法　　B. 数据元素的类型

C. 数据元素之间的关系　　D. 数据的存储方法

答：一个逻辑结构包含数据元素和元素之间的关系，在设计对应的存储结构时需要保存这两个方面的信息。答案为 C。

5. 在计算机的存储器中表示时，逻辑上相邻的两个元素对应的物理地址也是相邻的，这种存储结构称为_____。

A. 逻辑结构　　B. 顺序存储结构　　C. 链式存储结构　　D. 以上都正确

答：顺序存储结构是线性表的直接映射，逻辑上相邻的元素对应的物理地址也相邻。答案为 B。

6. 当数据采用链式存储结构时，要求_____。

A. 每个结点占用一片连续的存储区域

B. 所有结点占用一片连续的存储区域

C. 结点的最后一个数据域是指针类型

D. 每个结点有多少个后继就设多少个指针域

答：在链式存储结构中，通常用一个结点存放一个元素，这样每个结点占用一片连续的存储区域。答案为 A。

7. 以下关于算法的说法中正确的是_____。

A. 算法最终必须由计算机程序实现　　B. 算法等同于程序

C. 算法的可行性是指指令不能有二义性　　D. 以上几个都是错误的

答：算法不一定必须由计算机程序实现，算法也不同于程序(程序不必满足有穷性)，算法的确定性是指指令不能有二义性。答案为 D。

8. 算法的时间复杂度与_____有关。

A. 问题规模　　B. 计算机硬件性能

C. 编译程序质量　　D. 程序设计语言

答：在进行算法分析时通常采用事前分析估算法，认为一个算法的“运行工作量”的大小只依赖于问题的规模。答案为 A。

9. 算法分析的主要任务之一是分析_____。

A. 算法是否具有较好的可读性

B. 算法中是否存在语法错误

C. 算法的功能是否符合设计要求

D. 算法的执行时间和问题规模之间的关系

答：算法分析包含时间和空间复杂度分析，前者是找到算法的执行时间和问题规模之间的关系，以便改进算法性能。答案为 D。

10. 某算法的时间复杂度为 $O(n^2)$，表明该算法的_____。

A. 问题规模是 n^2　　B. 执行时间等于 n^2

C. 执行时间与 n^2 成正比　　D. 问题规模与 n^2 成正比

答：算法的时间复杂度为 $O(n^2)$，表示执行时间与问题规模呈平方关系，问题规模仍然是 n。答案为 C。

1.2　第 2 章　线性表

1.2.1　问答题

1. 简述顺序表和链表存储方式的主要优缺点。

答：顺序表的优点是可以随机存取元素，存储密度高，结构简单；缺点是需要一片地址连续的存储空间，不便于插入和删除元素（需要移动大量的元素），表的初始容量难以确定。

链表的优点是便于结点的插入和删除（只需要修改指针成员，不需要移动结点），表的容量扩充十分方便；缺点是不能进行随机访问，只能顺序访问，另外每个结点上增加指针成员，导致存储密度较低。

2. 对单链表设置一个头结点的作用是什么？

答：对单链表设置头结点的作用如下。

（1）当带头结点时，空表也存在一个头结点，从而统一了空表与非空表的处理。

（2）在单链表中插入结点和删除结点时都需要修改前驱结点的指针成员，当带头结点时任何数据结点都有前驱结点，这样使得插入和删除结点的操作更简单。

3. 假设均带头结点 h，给出单链表、双链表、循环单链表和循环双链表中 p 所指结点为尾结点的条件。

答：各种链表中 p 结点为尾结点的条件如下。

（1）单链表：p.next==null。

（2）双链表：p.next==null。

（3）循环单链表：p.next==h。

（4）循环双链表：p.next==h。

4. 假设某个含有 $n(n>0)$ 个元素的线性表有如下运算：

Ⅰ. 查找序号为 $i(0\leqslant i\leqslant n-1)$ 的元素

Ⅱ. 查找第一个值为 x 的元素的序号

Ⅲ. 插入第一个元素

Ⅳ. 插入最后一个元素

Ⅴ. 插入第 $i(0\leqslant i\leqslant n-1)$个元素

Ⅵ. 删除第一个元素

Ⅶ. 删除最后一个元素

Ⅷ. 删除第 $i(0\leqslant i\leqslant n-1)$个元素

现设计该线性表的如下存储结构：

① 顺序表

② 带头结点的单链表

③ 带头结点的循环单链表

④ 不带头结点仅有尾结点的循环单链表

⑤ 带头结点的双链表

⑥ 带头结点的循环双链表

给出上述各种运算在各种存储结构中实现算法的时间复杂度。

答：各种存储结构对应运算的时间复杂度如表 1.1 所示。

表 1.1　各种存储结构对应运算的时间复杂度

	Ⅰ	Ⅱ	Ⅲ	Ⅳ	Ⅴ	Ⅵ	Ⅶ	Ⅷ
①	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
②	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
③	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
④	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
⑤	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
⑥	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

5. 在单链表、双链表和循环单链表中，若仅知道指针 p 指向某结点，不知道头结点，能否不通过结点值复制操作将 p 结点从相应的链表中删去？若可以，其时间复杂度各为多少？

答：以下分 3 种链表进行讨论。

(1) 单链表：当已知指针 p 指向某结点时能够根据该指针找到其后继结点，但是由于不知道其头结点，所以无法访问到 p 指针指向的结点的前驱结点，因此无法删去该结点。

(2) 双链表：由于这样的链表提供双向链接，因此根据已知结点可以查找到其前驱和后继结点，从而可以删除该结点。其时间复杂度为 $O(1)$。

(3) 循环单链表：根据已知结点位置可以直接找到其后继结点，又因为是循环单链表，所以可以通过查找得到 p 结点的前驱结点，因此可以删去 p 所指结点。其时间复杂度为 $O(n)$。

6. 已知一个单链表 L 存放 n 个元素，其中结点类型为(data,next)，p 结点是其中的任意一个数据结点。请设计一个平均时间复杂度为 $O(1)$的算法删除 p 结点，并且说明为什么该算法的平均时间复杂度为 $O(1)$。

答：对于给定的带头结点的单链表 L，其头结点是 L.head，并且结点类型是确定的。删除 p 结点的操作如下：

(1) 若 p 结点不是尾结点,将 p 结点的后继结点的 data 值复制到 p 结点中,再删除 p 结点的后继结点。

(2) 若 p 结点是尾结点,通过其头结点找到尾结点的前驱结点 pre,再通过 pre 结点删除 p 结点。

对应的算法如下:

```
public static void Delpnode(LinkListClass < Integer > L, LinkNode < Integer > p)
{ LinkNode < Integer > q, pre;
  if(p.next!=null)                          //p 结点不是尾结点
  {   q=p.next;                             //q 指向 p 结点的后继结点
      p.data=q.data;                        //复制结点值
      p.next=q.next;                        //删除 q 结点
      q=null;
  }
  else                                      //p 结点是尾结点
  {   pre=L.head;
      while(pre.next!=p)                    //查找 p 结点的前驱结点 pre
          pre=pre.next;
      pre.next=p.next;                      //通过 pre 结点删除 p 结点
      p=null;
  }
}
```

对于含 n 个结点的单链表 L,上述算法仅仅在 p 结点为尾结点时的执行时间为 $O(n)$,其他 $n-1$ 种情况(p 结点不是尾结点)的执行时间均为 $O(1)$,所以算法的平均时间复杂度为 $\dfrac{(n-1)\times O(1)+1\times O(n)}{n}=O(1)$。

7. 带头结点的双链表和循环双链表相比有什么不同?在何时使用循环双链表?

答:在带头结点的双链表中,尾结点的后继指针为 null,头结点的前驱指针不使用;在带头结点的循环双链表中,尾结点的后继指针指向头结点,头结点的前驱指针指向尾结点。当需要快速找到尾结点时可以使用循环双链表。

1.2.2　算法设计题

1. 有一个整数顺序表 L,设计一个算法找最后一个值最小的元素的序号,并给出算法的时间和空间复杂度。例如 $L=(1,5,1,1,3,2,4)$,返回结果是 3。

解:用 min 表示最小的元素值(初始时为序号 0 的元素值),mini 表示最后一个值最小的元素的序号(初始为 0)。i 从 1 开始遍历 L 中的所有元素,若序号 i 的元素值小于或等于 minv,置 minv 为该元素值,mini=i。最后返回 mini。对应的算法如下:

```
public static int Lastmin(SqListClass < Integer > L)
{ Integer minv=L.GetElem(0);
  int mini=0;
  for(int i=1;i<L.size();i++)
      if(L.GetElem(i)<=minv)
      {   minv=L.GetElem(i);
```

```
            mini=i;
        }
    return mini;
}
```

该算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$。

2. 有一个整数顺序表 L,设计一个尽可能高效的算法删除其中所有值为负整数的元素(假设 L 中值为负整数的元素可能有多个),删除后元素的相对次序不改变,并给出算法的时间和空间复杂度。例如,$L=(1,2,-1,-2,3,-3)$,删除后 $L=(1,2,3)$。

解:采用《教程》中例 2.4 的 3 种解法,仅仅将保留元素的条件改为"元素值≥0"即可。对应的 3 种算法如下:

```
public static void Delminus1(SqListClass<Integer> L)
{ int i,k=0;
  for(i=0;i<L.size();i++)
      if(L.GetElem(i)>=0)                    //将元素值≥0 的元素插入 data 中
      {   L.SetElem(k,L.GetElem(i));
          k++;                               //累计插入的元素的个数
      }
  L.Setsize(k);                              //重置长度
}
public static void Delminus2(SqListClass<Integer> L)
{ int i,k=0;
  for(i=0;i<L.size();i++)
      if(L.GetElem(i)>=0)                    //将元素值≥0 的元素前移 k 个位置
          L.SetElem(i-k,L.GetElem(i));
      else                                   //累计删除的元素的个数
          k++;
  L.Setsize(L.size()-k);                     //重置长度
}
public static void Delminus3(SqListClass<Integer> L)
{ int i=-1,j=0;
  while(j<L.size())                          //j 遍历所有元素
  {   if(L.GetElem(j)>=0)                    //找到元素值≥0 的元素 a[j]
      {   i++;                               //扩大元素值≥0 的区间
          if(i!=j)
              L.swap(i,j);                   //将 a[i]与 a[j]交换
      }
      j++;                                   //继续遍历
  }
  L.Setsize(i+1);                            //重置长度
}
```

上述 3 种算法的时间复杂度均为 $O(n)$,空间复杂度均为 $O(1)$。

3. 有一个整数顺序表 L,设计一个尽可能高效的算法删除表中值大于或等于 x 且小于或等于 y 的所有元素($x\leqslant y$),删除后元素的相对次序不改变,并给出算法的时间和空间复杂度。例如,$L=(4,2,1,5,3,6,4)$,$x=2$,$y=4$,删除后 $L=(1,5,6)$。

解:采用《教程》中例 2.4 的 3 种解法,仅仅将保留元素的条件设置为"元素值$<x$ ‖ 元素值$>y$"

即可。对应的 3 种算法如下：

```
public static void Delinterval1(SqListClass < Integer > L, Integer x, Integer y)
{ int i,k=0;
  Integer e;
  for(i=0;i < L.size();i++)
  {   e=L.GetElem(i);
      if(e < x || e > y)                    //将保留的元素插入 data 中
      {   L.SetElem(k,e);
          k++;                              //累计保留的元素的个数
      }
  }
  L.Setsize(k);                             //重置长度
}
public static void Delinterval2(SqListClass < Integer > L, Integer x, Integer y)
{ int i,k=0;
  Integer e;
  for(i=0;i < L.size();i++)
  {   e=L.GetElem(i);
      if(e < x || e > y)                    //将保留的元素前移 k 个位置
          L.SetElem(i-k,e);
      else                                  //累计删除的元素的个数
          k++;
  }
  L.Setsize(L.size()-k);                    //重置长度
}
public static void Delinterval3(SqListClass < Integer > L, Integer x, Integer y)
{ int i=-1,j=0;
  Integer e;
  while(j < L.size())                       //j 遍历所有元素
  {   e=L.GetElem(j);
      if(e < x || e > y)                    //找到要保留的元素 a[j]
      {   i++;                              //扩大保留元素的区间
          if(i!=j)
              L.swap(i,j);                  //将 a[i]与 a[j]交换
      }
      j++;                                  //继续遍历
  }
  L.Setsize(i+1);                           //重置长度
}
```

上述 3 种算法的时间复杂度均为 $O(n)$，空间复杂度均为 $O(1)$。

4. 有一个整数顺序表 L，设计一个尽可能高效的算法将所有负整数的元素移到其他元素的前面，并给出算法的时间和空间复杂度。例如，$L=(1,2,-1,-2,3,-3,4)$，移动后 $L=(-1,-2,-3,2,3,1,4)$。

解：采用《教程》中例 2.4 的解法 3，即区间划分法，将“元素值!=x”改为“元素值<0”，并且移动后顺序表的长度不变。对应的算法如下：

```
public static void Move(SqListClass < Integer > L)
{ int i=-1,j=0;
```

```
  while(j < L.size())                          //j 扫描所有元素
  {   if(L.GetElem(j)< 0)                      //找到需要前移的元素 a[j]
      {   i++;                                 //扩大负元素的区间
          if(i!=j)
              L.swap(i,j);                     //将 a[i]与 a[j]交换
      }
      j++;                                     //继续扫描
  }
}
```

上述算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$。

5. 有一个整数顺序表 L,假设其中 n 个元素的编号是 1～n,设计一个尽可能高效的算法将所有编号为奇数的元素移到所有编号为偶数的元素的前面,并给出算法的时间和空间复杂度。例如,$L=(1,2,3,4,5,6,7)$,移动后 $L=(1,3,5,7,2,6,4)$。

解:采用《教程》中例 2.4 的解法 3,即区间划分法,在遍历元素 $a[j]$时将"元素值!=x"改为"$j\%2==0$"(序号 j 为偶数对应编号为奇数的元素,是需要前移的元素),并且移动后顺序表的长度不变。对应的算法如下:

```
public static void Move(SqListClass < Integer > L)
{ int i=-1,j=0;
  while(j < L.size())                          //j 扫描所有元素
  {   if(j%2==0)                               //找到需要前移的元素 a[j]
      {   i++;                                 //扩大序号为奇数的元素的区间
          if(i!=j)
            L.swap(i,j);                       //将 a[i]与 a[j]交换
      }
      j++;                                     //继续扫描
  }
}
```

上述算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$。

6. 有一个递增有序的整数顺序表 L,设计一个算法将整数 x 插入适当位置,以保持该表的有序性,并给出算法的时间和空间复杂度。例如,$L=(1,3,5,7)$,插入 $x=6$ 后 $L=(1,3,5,6,7)$。

解:先在有序顺序表 L 中查找有序插入 x 的位置 i(即在 L 中从前向后查找到刚好大于或等于 x 的位置 i),再调用 L.Insert(i,x)插入元素 x。对应的算法如下:

```
public static void Insertx(SqListClass < Integer > L, Integer x)
{ int i=0;
  while(i < L.size() && L.GetElem(i)< x)       //查找刚好≥x 的元素的序号 i
      i++;
  L.Insert(i,x);
}
```

上述算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$。

7. 有一个递增有序的整数顺序表 L,设计一个尽可能高效的算法删除表中值大于或等于 x 且小于或等于 y 的所有元素($x\leqslant y$),删除后元素的相对次序不改变,并给出算法的时间和空间复杂度。例如,$L=(1,2,4,5,7,9)$,$x=2$,$y=5$,删除后 $L=(1,7,9)$。

解:先根据顺序表 L 的有序性查找(即在 L 中从前向后查找到第一个大于或等于 x 的

位置 i),再采用《教程》中例 2.4 的前两种解法实施删除。对应的两种算法如下:

```
public static void Delinterval1(SqListClass < Integer > L, Integer x, Integer y)
{ int i=0,k=0;
  Integer e;
  while(i < L.size() && L.GetElem(i)< x)          //查找第一个>=x 的元素的序号 i
  {   i++;
      k++;
  }
  for(;i < L.size();i++)
  {   e=L.GetElem(i);
      if(e > y)                                   //将保留的元素插入 data 中
      {   L.SetElem(k,e);
          k++;                                    //累计保留的元素的个数
      }
  }
  L.Setsize(k);                                   //重置长度
}
public static void Delinterval2(SqListClass < Integer > L, Integer x, Integer y)
{ int i=0,k=0;
  Integer e;
  while(i < L.size())                             //查找第一个>=x 的元素的序号 i
  {   e=L.GetElem(i);
      if(e >=x && e <=y) k++;                     //累计>=x 且<=y 的元素个数 k
      else if(e > y) break;                       //找到> y 的元素时退出
      i++;
  }
  for(;i < L.size();i++)                          //将后面保留的元素前移 k 个位置
  {   e=L.GetElem(i);
      L.SetElem(i-k,e);
  }
  L.Setsize(L.size()-k);                          //重置长度
}
```

上述两种算法的时间复杂度均为 $O(n)$,空间复杂度均为 $O(1)$。与本节中第 3 题的算法相比,这里的有序顺序表并没有提高算法的效率。

8. 有两个集合采用整数顺序表 A、B 存储,设计一个算法求两个集合的并集 C,C 仍然用顺序表存储,并给出算法的时间和空间复杂度。例如 $A=(1,3,2)$,$B=(5,1,4,2)$,并集 $C=(1,3,2,5,4)$。

说明:这里的集合均指数学意义上的集合,在同一个集合中不存在值相同的元素。

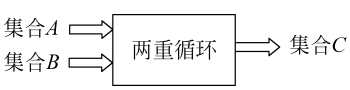

图 1.1　两个无序集合求并集

解:将集合 A 中的所有元素添加到 C 中,再将集合 B 中不属于 A 的元素添加到集合 C 中,最后返回 C,如图 1.1 所示。对应的算法如下:

```
public static SqListClass < Integer > Union(SqListClass < Integer > A, SqListClass < Integer > B)
{ SqListClass < Integer > C=new SqListClass < Integer >();
  int i,j;
  Integer e;
```

```
    for(i=0;i<A.size();i++)                    //将 A 中的所有元素添加到 C 中
        C.Add(A.GetElem(i));
    for(j=0;j<B.size();j++)                    //将 B 中不属于 A 的元素添加到 C 中
    {   e=B.GetElem(j);                        //时间复杂度为 O(1)
        if(A.GetNo(e)==-1)                     //时间复杂度为 O(n)
            C.Add(e);
    }
    return C;                                  //返回 C
}
```

上述算法的时间复杂度为 $O(m\times n)$,空间复杂度为 $O(m+n)$,其中 m 和 n 分别表示 A 和 B 中元素的个数。

9. 有两个集合采用递增有序的整数顺序表 A、B 存储,设计一个在时间上尽可能高效的算法求两个集合的并集 C,C 仍然用顺序表存储,并给出算法的时间和空间复杂度。例如 $A=(1,3,5,7)$,$B=(1,2,4,5,7)$,并集 $C=(1,2,3,4,5,7)$。

解:由于顺序表 A、B 是有序的,采用二路归并方法,当 A、B 都没有遍历完时将较小元素添加到 C 中,相等的公共元素仅添加一个,如图 1.2 所示。再将没有归并完的其余元素都添加到 C 中,最后返回 C。对应的算法如下:

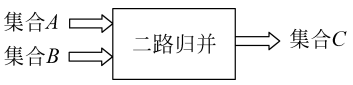

图 1.2　两个有序集合求并集

```
public static SqListClass<Integer> Union(SqListClass<Integer> A,SqListClass<Integer> B)
{ SqListClass<Integer> C=new SqListClass<Integer>();
  Integer ea,eb;
  int i=0,j=0;
  while(i<A.size() && j<B.size())
  {   ea=A.GetElem(i);
      eb=B.GetElem(j);
      if(ea<eb)                                //将较小元素 ea 添加到 C 中
      {   C.Add(ea);
          i++;
      }
      else if(eb<ea)                           //将较小元素 eb 添加到 C 中
      {   C.Add(eb);
          j++;
      }
      else                                     //公共元素只添加一个
      {   C.Add(ea);
          i++; j++;
      }
  }
  while(i<A.size())                            //若 A 未遍历完,将余下的所有元素添加到 C 中
  {   C.Add(A.GetElem(i));
      i++;
  }
  while(j<B.size())                            //若 B 未遍历完,将余下的所有元素添加到 C 中
  {   C.Add(B.GetElem(j));
      j++;
  }
  return C;
}
```

上述算法的时间复杂度为 $O(m+n)$，空间复杂度为 $O(m+n)$，其中 m 和 n 分别表示 A 和 B 中元素的个数。与本节中第 8 题的算法相比，这里利用有序顺序表提高了算法的效率。

10. 有两个集合采用整数顺序表 A、B 存储，设计一个算法求两个集合的差集 C，C 仍然用顺序表存储，并给出算法的时间和空间复杂度。例如 $A=(1,3,2)$，$B=(5,1,4,2)$，并集 $C=(3)$。

解：将集合 A 中所有不属于集合 B 的元素添加到 C 中，最后返回 C。对应的算法如下：

```
public static SqListClass<Integer> Diff(SqListClass<Integer> A,SqListClass<Integer> B)
{  SqListClass<Integer> C=new SqListClass<Integer>();
   Integer e;
   for(int i=0;i<A.size();i++)                    //将 A 中不属于 B 的元素添加到 C 中
   {    e=A.GetElem(i);                           //时间为 O(1)
        if(B.GetNo(e)==-1)                        //时间为 O(n)
            C.Add(e);
   }
   return C;                                      //返回 C
}
```

上述算法的时间复杂度为 $O(m\times n)$，空间复杂度为 $O(m)$，其中 m 和 n 分别表示 A 和 B 中元素的个数。

11. 有两个集合采用递增有序的整数顺序表 A、B 存储，设计一个在时间上尽可能高效的算法求两个集合的差集 C，C 仍然用顺序表存储，并给出算法的时间和空间复杂度。例如 $A=(1,3,5,7)$，$B=(1,2,4,5,9)$，差集 $C=(3,7)$。

解：差集 C 中的元素是属于 A 而不属于 B 的元素，由于顺序表 A、B 是有序的，采用二路归并方法，在归并中先将 A 中小于 B 的元素添加到 C 中，当 B 遍历完后再将 A 中较大的元素（如果存在这样的元素）添加到 C 中，最后返回 C。对应的算法如下：

```
public static SqListClass<Integer> Diff(SqListClass<Integer> A,SqListClass<Integer> B)
{  SqListClass<Integer> C=new SqListClass<Integer>();
   Integer ea,eb;
   int i=0,j=0;
   while(i<A.size() && j<B.size())
   {    ea=A.GetElem(i);
        eb=B.GetElem(j);
        if(ea<eb)                                 //将较小元素 ea 添加到 C 中
        {    C.Add(ea);
             i++;
        }
        else if(eb<ea)                            //忽略较小元素 eb
             j++;
        else                                      //忽略公共元素
        {    i++;
             j++;
        }
   }
   while(i<A.size())                              //若 A 未遍历完，将余下的所有元素添加到 C 中
```

```
    {   C.Add(A.GetElem(i));
        i++;
    }
    return C;
}
```

上述算法的时间复杂度为 $O(m+n)$,空间复杂度为 $O(m)$,其中 m 和 n 分别表示 A 和 B 中元素的个数。与本节中第 10 题的算法相比,这里利用有序顺序表提高了算法的效率。

12. 有两个集合采用整数顺序表 A、B 存储,设计一个算法求两个集合的交集 C,C 仍然用顺序表存储,并给出算法的时间和空间复杂度。例如 $A=(1,3,2)$,$B=(5,1,4,2)$,交集 $C=(1,2)$。

解:将集合 A 中所有属于 B 的元素添加到集合 C 中,最后返回 C。对应的算法如下:

```
public static SqListClass<Integer> Inter(SqListClass<Integer> A,SqListClass<Integer> B)
{ SqListClass<Integer> C=new SqListClass<Integer>();
  Integer e;
  for(int i=0;i<A.size();i++)              //将 B 中属于 A 的元素添加到 C 中
  {   e=A.GetElem(i);                      //时间复杂度为 O(1)
      if(B.GetNo(e)!=-1)                   //时间复杂度为 O(n)
          C.Add(e);
  }
  return C;                                //返回 C
}
```

上述算法的时间复杂度为 $O(m\times n)$,空间复杂度为 $O(\mathrm{MIN}(m,n))$,其中 m 和 n 分别表示 A 和 B 中元素的个数。

13. 有两个集合采用递增有序的整数顺序表 A、B 存储,设计一个在时间上尽可能高效的算法求两个集合的交集 C,C 仍然用顺序表存储,并给出算法的时间和空间复杂度。例如 $A=(1,3,5,7)$,$B=(1,2,4,5,7)$,交集 $C=(1,5,7)$。

解:由于顺序表 A、B 是有序的,采用二路归并方法,在归并中仅仅将 A、B 中相同值的元素添加到 C 中,最后返回 C。对应的算法如下:

```
public static SqListClass<Integer> Inter(SqListClass<Integer> A,SqListClass<Integer> B)
{ SqListClass<Integer> C=new SqListClass<Integer>();
  Integer ea,eb;
  int i=0,j=0;
  while(i<A.size() && j<B.size())
  {   ea=A.GetElem(i);
      eb=B.GetElem(j);
      if(ea<eb)                            //忽略较小元素 ea
          i++;
      else if(eb<ea)                       //忽略较小元素 eb
          j++;
      else                                 //仅仅将公共元素添加到 C 中
      {   C.Add(ea);
          i++; j++;
      }
  }
```

```
    return C;
}
```

上述算法的时间复杂度为 $O(m+n)$，空间复杂度为 $O(\text{MIN}(m,n))$，其中 m 和 n 分别表示 A 和 B 中元素的个数。与本节中第 12 题的算法相比，这里利用有序顺序表提高了算法的效率。

14. 有一个整数单链表 L，设计一个算法删除其中所有值为 x 的结点，并给出算法的时间和空间复杂度。例如 $L=(1,2,2,3,1)$，$x=2$，删除后 $L=(1,3,1)$。

解：设置(pre，p)指向 L 中相邻的两个结点，初始时 pre 指向头结点，p 指向首结点。用 p 遍历 L：

(1) 若 p 结点的值为 x，通过 pre 结点删除 p 结点，置 p 结点为 pre 结点的后继结点。

(2) 若 p 结点的值不为 x，pre 和 p 同步后移一个结点。

对应的算法如下：

```
public static void Delx(LinkListClass<Integer> L,Integer x)
{ LinkNode<Integer> pre=L.head,p=pre.next;          //p 指向首结点
  while(p!=null)                                    //遍历所有数据结点
  {   if(p.data==x)                                 //找到值为 x 的 p 结点
      {   pre.next=p.next;                          //通过 pre 结点删除 p 结点
          p=pre.next;                               //置 p 结点为 pre 结点的后继结点
      }
      else                                          //p 结点不是值为 x 的结点
      {   pre=pre.next;                             //pre 和 p 同步后移一个结点
          p=pre.next;
      }
  }
}
```

上述算法的时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

15. 有一个单链表 L，设计一个算法在第 i 个结点之前插入一个元素 x，若参数错误返回 false，否则插入后返回 true，并给出算法的时间和空间复杂度。例如 L=(1,2,3,4,5)，$i=0$，$x=6$，插入后 $L=(6,1,2,3,4,5)$。

解：i 参数的有效范围是 0～$n-1$。从头结点开始查找第 $i-1$ 个结点，若不存在，则返回 false；若存在第 $i-1$ 个结点 pre，在其后插入含值 x 的结点 s，返回 true。对应的算法如下：

```
public static boolean Inserti(LinkListClass<Integer> L,int i,Integer x)
{ LinkNode<Integer> pre=L.head,s;                   //pre 指向头结点
  if(i<0) return false;
  int j=-1;
  while(j<i-1 && pre!=null)                         //查找第 i-1 个结点 pre
  {   j++;
      pre=pre.next;
  }
  if(pre==null)                                     //没有找到第 i-1 个结点
      return false;
  else                                              //找到第 i-1 个结点 pre
```

```
    {   s=new LinkNode<Integer>(x);
        s.next=pre.next;                                    //在 pre 结点之后插入 s 结点
        pre.next=s;
        return true;
    }
}
```

上述算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$。

16. 有一个整数单链表 L,设计一个尽可能高效的算法将所有负整数的元素移到其他元素的前面。例如,$L=(1,2,-1,-2,3,-3,4)$,移动后 $L=(-1,-2,-3,1,2,3,4)$。

解法 1:删除插入法。先跳过单链表 L 开头的负整数结点,last 指向最后一个负整数结点,p 指向其后继结点。pre 置为 last,(pre,p)遍历余下的结点:

(1) 若 p 结点的值<0,通过 pre 结点删除 p 结点,再将 p 结点插入 last 结点之后,然后置 last=p,p 继续指向 pre 结点的后继结点,以此类推,直到 p 为空。

(2) 否则,pre 和 p 同步后移一个结点。

对应的算法如下:

```
public static void Move1(LinkListClass<Integer> L)
{ LinkNode<Integer> last=L.head,p=L.head.next,pre,q;
  while(p!=null && p.data<0)                //跳过开头的负整数结点
  {   last=last.next;                       //将 last、p 结点同步后移一个结点
      p=p.next;                             //循环结束后 last 结点为前面负整数的尾结点
  }
  pre=last;
  while(p!=null)                            //查找负整数结点 p
  {   if(p.data<0)                          //找到负整数结点 p
      {   pre.next=p.next;                  //删除 p 结点
          p.next=last.next;                 //将 p 结点插入 last 结点之后
          last.next=p; last=p;
          p=pre.next;
      }
      else                                  //p 结点不是负整数结点
      {   pre=pre.next;                     //将 last、p 结点同步后移一个结点
          p=pre.next;
      }
  }
}
```

解法 2:分拆合并法。扫描单链表 L,采用尾插法建立由所有负整数结点构成的单链表 A,采用尾插法建立由所有其他结点构成的单链表 B(A、B 均带头结点)。将 B 链接到 A,再将 L 的头结点作为新单链表的头结点。对应的算法如下:

```
public static void Move2(LinkListClass<Integer> L)
{ LinkNode<Integer> p=L.head.next,pre,q;
  LinkNode<Integer> A=new LinkNode<Integer>();
  LinkNode<Integer> B=new LinkNode<Integer>();
  LinkNode<Integer> ta=A,tb=B;              //ta、tb 分别为 A 和 B 的尾结点
  while(p!=null)
```

```
    {   if(p.data<0)                    //找到负整数结点 p
        {   ta.next=p;                  //将 p 结点连接到 A 的末尾
            ta=p;
            p=p.next;
        }
        else                            //不是负整数结点 p
        {   tb.next=p;                  //将 p 结点连接到 B 的末尾
            tb=p;
            p=p.next;
        }
    }
    ta.next=tb.next=null;               //两个单链表的尾结点的 next 置为 null
    ta.next=B.next;                     //将 B 连接到 A 的后面
    L.head.next=A.next;                 //重置 L
}
```

17. 有两个集合采用整数单链表 A、B 存储，设计一个算法求两个集合的并集 C，C 仍然用单链表存储，并给出算法的时间和空间复杂度。例如 $A=(1,3,2)$，$B=(5,1,4,2)$，并集 $C=(1,3,2,5,4)$。

解：本题直接利用本节中第 8 题的求解思路，只将顺序表改为单链表（两者的时间复杂度相同）。将集合 A 中的所有元素添加到 C 中，再将集合 B 中不属于 A 的元素添加到集合 C 中，最后返回 C。对应的算法如下：

```
public static LinkListClass<Integer> Union(LinkListClass<Integer> A,LinkListClass<Integer> B)
{ LinkListClass<Integer> C=new LinkListClass<Integer>();
  int i,j;
  Integer e;
  for(i=0;i<A.size();i++)               //将 A 中的所有元素添加到 C 中
      C.Add(A.GetElem(i));
  for(j=0;j<B.size();j++)               //将 B 中不属于 A 的元素添加到 C 中
  {   e=B.GetElem(j);                   //时间复杂度为 O(n)
      if(A.GetNo(e)==-1)                //时间复杂度为 O(n)
          C.Add(e);
  }
  return C;                             //返回 C
}
```

上述算法的时间复杂度为 $O(m\times n)$，空间复杂度为 $O(m+n)$，其中 m 和 n 分别表示 A 和 B 中元素的个数。

18. 有两个集合采用递增有序的整数单链表 A、B 存储，设计一个在时间上尽可能高效的算法求两个集合的并集 C，C 仍然用单链表存储，并给出算法的时间和空间复杂度。例如 $A=(1,3,5,7)$，$B=(1,2,4,5,7)$，并集 $C=(1,2,3,4,5,7)$。

解：本题可以直接利用本节中第 9 题的求解思路，只需要将顺序表改为单链表，但在顺序表中 GetElem(i)的时间为 $O(1)$，而单链表中 GetElem(i)的时间为 $O(n)$，为了提高效率，此处改为采用二路归并+尾插法建立并集 C 单链表。对应的算法如下：

```
public static LinkListClass<Integer> Union(LinkListClass<Integer> A,LinkListClass<Integer> B)
{ LinkListClass<Integer> C=new LinkListClass<Integer>();
```

```
    LinkNode<Integer> t=C.head;                    //t 指向 C 的尾结点
    LinkNode<Integer> p=A.head.next;
    LinkNode<Integer> q=B.head.next;
    while(p!=null && q!=null)
    {   if(p.data<q.data)                          //将较小结点 p 添加到 C 中
        {   t.next=p; t=p;
            p=p.next;
        }
        else if(q.data<p.data)                     //将较小结点 q 添加到 C 中
        {   t.next=q; t=q;
            q=q.next;
        }
        else                                       //公共结点只放一个
        {   t.next=p; t=p;
            p=p.next;
            q=q.next;
        }
    }
    t.next=null;
    if(p!=null) t.next=p;                          //若 A 未遍历完,将余下的所有结点连接到 C 中
    if(q!=null) t.next=q;                          //若 B 未遍历完,将余下的所有结点连接到 C 中
    return C;
}
```

上述算法的时间复杂度为 $O(m+n)$,空间复杂度为 $O(1)$,其中 m 和 n 分别表示 A 和 B 中元素的个数。与本节中第 17 题的算法相比,这里利用有序性提高了算法的效率。

说明:这里单链表 C 是利用 A 和 B 单链表的结点重构得到的,并没有新建结点,算法执行后 A 和 B 单链表不复存在。若采用结点复制的方法,不破坏 A 和 B 单链表,对应的时间复杂度为 $O(m+n)$,空间复杂度为 $O(m+n)$。

19. 有两个集合采用整数单链表 A、B 存储,设计一个算法求两个集合的差集 C,C 仍然用单链表存储,并给出算法的时间和空间复杂度。例如 $A=(1,3,2)$,$B=(5,1,4,2)$,并集 $C=(3)$。

解:本题直接利用本节中第 10 题的求解思路,只将顺序表改为单链表(两者的时间复杂度相同),即将集合 A 中所有不属于集合 B 的元素添加到 C 中,最后返回 C。对应的算法如下:

```
public static LinkListClass<Integer> Diff(LinkListClass<Integer> A,LinkListClass<Integer> B)
{ LinkListClass<Integer> C=new LinkListClass<Integer>();
  Integer e;
  for(int i=0;i<A.size();i++)                      //将 A 中不属于 B 的元素添加到 C 中
  {   e=A.GetElem(i);                              //时间复杂度为 O(n)
      if(B.GetNo(e)==-1)                           //时间复杂度为 O(n)
          C.Add(e);
  }
  return C;                                        //返回 C
}
```

上述算法的时间复杂度为 $O(m\times n)$,空间复杂度为 $O(m)$,其中 m 和 n 分别表示 A 和

B 中元素的个数。

20. 有两个集合采用递增有序的整数单链表 A、B 存储，设计一个在时间上尽可能高效的算法求两个集合的差集 C，C 仍然用单链表存储，并给出算法的时间和空间复杂度。例如 $A=(1,3,5,7)$，$B=(1,2,4,5,9)$，差集 $C=(3,7)$。

解：采用二路归并＋尾插法建立并集 C 单链表。差集 C 中的元素是属于 A 而不属于 B 的元素，在归并中先将 A 中小于 B 的元素添加到 C 中，当 B 遍历完后再将 A 中较大的元素（如果存在这样的元素）添加到 C 中，最后返回 C。对应的算法如下：

```
public static LinkListClass<Integer> Diff(LinkListClass<Integer> A,LinkListClass<Integer> B)
{ LinkListClass<Integer> C=new LinkListClass<Integer>();
  LinkNode<Integer> t=C.head;                    //t 指向 C 的尾结点
  LinkNode<Integer> p=A.head.next;
  LinkNode<Integer> q=B.head.next;
  while(p!=null && q!=null)
  {   if(p.data<q.data)                          //将较小结点 p 添加到 C 中
      {   t.next=p; t=p;
          p=p.next;
      }
      else if(q.data<p.data)                     //忽略较小结点 q
          q=q.next;
      else                                       //忽略公共结点
      {   p=p.next;
          q=q.next;
      }
  }
  t.next=null;
  if(p!=null) t.next=p;                          //若 A 未遍历完,将余下的所有结点连接到 C 中
  return C;
}
```

上述算法的时间复杂度为 $O(m+n)$，空间复杂度为 $O(m)$，其中 m 和 n 分别表示 A 和 B 中元素的个数。与本节中第 19 题的算法相比，这里利用有序性提高了算法的效率。

21. 有两个集合采用整数单链表 A、B 存储，设计一个算法求两个集合的交集 C，C 仍然用单链表存储，并给出算法的时间和空间复杂度。例如 $A=(1,3,2)$，$B=(5,1,4,2)$，交集 $C=(1,2)$。

解：本题直接利用本节中第 12 题的求解思路，只将顺序表改为单链表（两者的时间复杂度相同），即将集合 A 中所有属于 B 的元素添加到集合 C 中，最后返回 C。对应的算法如下：

```
public static LinkListClass<Integer> Inter(LinkListClass<Integer> A,LinkListClass<Integer> B)
{ LinkListClass<Integer> C=new LinkListClass<Integer>();
  Integer e;
  for(int i=0;i<A.size();i++)                    //将 B 中属于 A 的元素添加到 C 中
  {   e=A.GetElem(i);                            //时间复杂度为 O(n)
      if(B.GetNo(e)!=-1)                         //时间复杂度为 O(n)
          C.Add(e);
  }
```

```
    return C;                                    //返回 C
}
```

上述算法的时间复杂度为 $O(m\times n)$,空间复杂度为 $O(\text{MIN}(m,n))$,其中 m 和 n 分别表示 A 和 B 中元素的个数。

22. 有两个集合采用递增有序的整数单链表 A、B 存储,设计一个在时间上尽可能高效的算法求两个集合的交集 C,C 仍然用单链表存储,并给出算法的时间和空间复杂度。例如 $A=(1,3,5,7)$,$B=(1,2,4,5,7)$,交集 $C=(1,5,7)$。

解:采用二路归并+尾插法建立并集 C 单链表,在归并中仅仅将 A、B 中值相同的元素添加到 C 中,最后返回 C。对应的算法如下:

```
public static LinkListClass < Integer > Inter(LinkListClass < Integer > A,LinkListClass < Integer > B)
{  LinkListClass < Integer > C=new LinkListClass < Integer >();
   LinkNode < Integer > t=C.head;                //t 结点指向 C 的尾结点
   LinkNode < Integer > p=A.head.next;
   LinkNode < Integer > q=B.head.next;
   while(p!=null && q!=null)
   {    if(p.data < q.data)                      //忽略较小结点 p
            p=p.next;
        else if(q.data < p.data)                 //忽略较小结点 q
            q=q.next;
        else                                     //仅仅将公共结点添加到 C 中
        {    t.next=p; t=p;
             p=p.next;
             q=q.next;
        }
   }
   return C;
}
```

上述算法的时间复杂度为 $O(m+n)$,空间复杂度为 $O(\text{MIN}(m,n))$,其中 m 和 n 分别表示 A 和 B 中元素的个数。与本节中第 21 题的算法相比,这里利用有序性提高了算法的效率。

23. 有两个递增有序的整数单链表 A、B,分别含 m 和 n 个元素,设计一个在时间上尽可能高效的算法将这 $m+n$ 个元素合并到单链表 C 中,使得 C 中的所有整数递减排列,并给出算法的时间和空间复杂度。例如 $A=(1,3,5,7)$,$B=(1,2,8)$,合并后 $C=(8,7,5,3,2,1,1)$。

解:采用二路归并+头插法建立单链表 C。对应的算法如下:

```
public static LinkListClass < Integer > Union(LinkListClass < Integer > A,LinkListClass < Integer > B)
{  LinkListClass < Integer > C=new LinkListClass < Integer >();
   LinkNode < Integer > p=A.head.next;
   LinkNode < Integer > q=B.head.next;
   LinkNode < Integer > tmp;
   while(p!=null && q!=null)
   {    if(p.data < q.data)                      //将较小结点添加到 C 中
        {    tmp=p.next;                         //tmp 临时保存 p 结点的后继结点
             p.next=C.head.next;                 //将 p 结点插入到 C 的表头
```