第一部分 练习题及参考答案

1.1 第1章 绪论

1.1.1 练习题

1. 什么是数据结构？有关数据结构的讨论涉及哪三方面？

2. 简述数据结构中运算描述和运算实现的异同。

3. 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。

4. 分析以下算法的时间复杂度。

```
def fun(n):
    x,y=n,100
    while y>0:
        if x>100:
            x=x-10
            y-=1
        else: x+=1
```

5. 分析以下算法的时间复杂度。

```
def fun(n):
    i,k=1,100
    while i<=n:
        k+=1
        i+=2
```

6. 分析以下算法的时间复杂度。

```
def fun(n):
    i=1
    while i<=n:
        i=i*2
```

7. 分析以下算法的时间复杂度。

```
def fun(n):
    for i in range(1,n+1):
        for j in range(1,n+1):
            k=1
            while k<=n:k=5*k
```

1.1.2 练习题参考答案

1. 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，称为一个数据结构。

数据结构涉及以下三方面的内容：

① 数据成员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，即运算。

2. 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，运算描述和运算实现都能完成对数据的“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。

3. 答：通常算法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。它们是在算法开始运算前给予算法的量。这些输入取自特定的对象的集合。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，输出的量是算法运算的结果。

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。

4. 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 $O(1)$。

5. 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 $O(n)$。

6. 答：本算法中的基本操作语句是 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 $O(\log_2 n)$。

7. 答：算法中的基本操作语句为 k=5 * k。对于 while 循环：

```
k=1
while k<=n: k=5 * k
```

其中基本操作语句 k=5 * k 的频度为 $\log_5 n$，再加上外层两重循环，所以本算法的时间复杂度为 $O(n^2\log_5 n)$。

1.2　第 2 章　线性表

1.2.1　练习题

1. 简述顺序表和链表存储方式的主要优缺点。

2. 对单链表设置一个头结点的作用是什么？

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。

4. 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？

5. 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？

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 $L=(1,3,5,6,7)$。

7. 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 $L=(1,2,3)$。

8. 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 $L=(-1,-2,-3,2,3,1,4)$。

9. 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，同一个集合中不存在相同值的元素。

10. 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 $C=(1,2,3,4,5,7)$。

11. 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 $C=(3)$。

12. 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 $C=(3,7)$。

13. 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 $C=(1,2)$。

14. 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 $C=(1,5,7)$。

15. 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 $L=(1,3,1)$。

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 $L=(-1,-2,-3,1,2,3,4)$。

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 $C=(1,3,2,5,4)$。

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 $C=(1,2,3,4,5,7)$。

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 $A=(1,3,2)$,$B=(5,1,4,2)$,差集 $C=(3)$。

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 $C=(3,7)$。

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 $C=(1,2)$。

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 $C=(1,5,7)$。

23. 有一个递增有序的整数双链表 L,其中至少有两个结点,设计一个算法就地删除 L 中所有值重复的结点,即多个相同值的结点仅保留一个。例如,$L=(1,2,2,2,3,5,5)$,删除后 $L=(1,2,3,5)$。

24. 有两个递增有序的整数双链表 A 和 B,分别含有 m 和 n 个整数元素,假设这 $m+n$ 个元素均不相同,设计一个算法求这 $m+n$ 个元素中第 $k(1\leqslant k\leqslant m+n)$ 小的元素值。例如,$A=(1,3)$,$B=(2,4,6,8,10)$,$k=2$ 时返回 2,$k=6$ 时返回 8。

1.2.2 练习题参考答案

1. 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(只需要修改指针属性,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属性，导致存储密度较低。

2. 答：对单链表设置头结点的好处如下。

① 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属性，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操作更简单。

3. 答：各种链表中 p 结点为尾结点的条件如下。

① 单链表：p.next==None。

② 双链表：p.next==None。

③ 循环单链表：p.next==h。

④ 循环双链表：p.next==h。

4. 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，因此无法删除该结点。

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：根据已知结点的位置，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，因此可以删除 p 所指结点。其时间复杂度为 $O(n)$。

5. 答：在带头结点的双链表中，尾结点的后继指针为 None，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。

6. 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：

```
def Insertx(L,x):
    i=0
    while i<L.getsize() and L[i]<x:          #查找刚好≥x的元素序号 i
        i+=1
    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. 解：采用《教程》中例 2.3 的 3 种解法，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：

```
def Delminus1(L):                            #算法 1
    k=0
    for i in range(0,L.getsize()):
        if L[i]>=0:                          #将元素值≥0的元素插入 data 中
            L[k]=L[i]
            k+=1                             #累计插入的元素个数
    L.size=k;                                #重置长度

def Delminus2(L):                            #算法 2
```

```
    k=0
    for i in range(0,L.getsize()):
        if L[i]>=0:                          #将元素值≥0 的元素前移 k 个位置
            L[i-k]=L[i]
        else:                                #累计删除的元素个数
            k+=1
    L.size-=k                                #重置长度

def Delminus3(L):                            #算法 3
    i=-1
    j=0
    while j<L.getsize():                     #j 遍历所有元素
        if L[j]>=0:                          #找到元素值≥0 的元素 a[j]
            i+=1                             #扩大元素值≥0 的区间
            if i!=j:                         #i 不等于 j 时将 data[i]与 data[j]交换
                L[i],L[j]=L[j],L[i]
        j+=1                                 #继续遍历
    L.size=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 $O(1)$。

8. 解:采用《教程》中例 2.3 的解法 3(即区间移动法),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:

```
def Move(L):
    i=-1
    j=0
    while j<L.getsize():                     #j 扫描所有元素
        if L[j]<0:                           #找到需要前移的元素 a[j]
            i+=1                             #扩大负元素的区间
            if i!=j:                         #i 不等于 j 时将 data[i]与 data[j]交换
                L[i],L[j]=L[j],L[i]          #将序号为 i 和 j 的两个元素交换
        j+=1                                 #继续遍历
```

上述算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$。

9. 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:

```
def Union(A,B):
    C=SqList()
    for i in range(0,A.getsize()):           #将 A 中的所有元素添加到 C 中
        C.Add(A[i])
    for j in range(0,B.getsize()):           #将 B 中不属于 A 的元素添加到 C 中
        e=B[j]
        if A.GetNo(e)==-1:                   #时间为 O(n)
            C.Add(e)
    return C                                 #返回 C
```

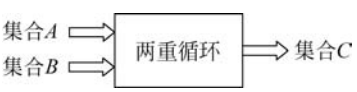

图 1.1 两个无序集合求并集

上述算法的时间复杂度均为 $O(m\times n)$,空间复杂度为 $O(m+n)$,其中 m 和 n 分别表示 A 和 B 中的元素个数。

10. 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，再将没有归并完的其余元素均添加到 C 中，最后返回 C，如图 1.2 所示。对应的算法如下：

```
def Union(A ,B):
    C=SqList()
    i=j=0
    while i<A.getsize() and j<B.getsize():
        if A[i]<B[j]:                    #将较小元素 A[i]添加到 C 中
            C.Add(A[i])
            i+=1
        elif B[j]<A[i]:                  #将较小元素 B[j]添加到 C 中
            C.Add(B[j])
            j+=1
        else:                            #公共元素只添加一个
            C.Add(A[i])
            i+=1
            j+=1
    while i<A.getsize():                 #若 A 未遍历完,将余下的所有元素添加到 C 中
        C.Add(A[i])
        i+=1
    while j<B.getsize():                 #若 B 未遍历完,将余下的所有元素添加到 C 中
        C.Add(B[j])
        j+=1
    return C
```

上述算法的时间复杂度均为 $O(m+n)$，空间复杂度为 $O(m+n)$，其中 m 和 n 分别表示 A 和 B 中的元素个数。

图 1.2　两个有序集合求并集

11. 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：

```
def Diff(A,B):
    C=SqList()
    for i in range(0,A.getsize()):       #将 A 中不属于 B 的元素添加到 C 中
        if B.GetNo(A[i])==-1:            #时间为 O(n)
            C.Add(A[i])
    return C                             #返回 C
```

上述算法的时间复杂度均为 $O(m\times n)$，空间复杂度为 $O(m)$，其中 m 和 n 分别表示 A 和 B 中的元素个数。

12. 解：差集 C 中的元素是属于 A 但不属于 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：

```
def Diff(A,B):
    C=SqList()
    i=j=0
    while i<A.getsize() and j<B.getsize():
```

```
        if A[i]<B[j]:              #将较小元素 A[i]添加到 C 中
            C.Add(A[i])
            i+=1
        elif B[j]<A[i]:            #忽略较小元素 B[j]
            j+=1
        else:                      #忽略公共元素
            i+=1
            j+=1
    while i<A.getsize():           #若 A 未遍历完,将余下的所有元素添加到 C 中
        C.Add(A[i])
        i+=1
    return C
```

上述算法的时间复杂度均为 $O(m+n)$,空间复杂度为 $O(m)$,其中 m 和 n 分别表示 A 和 B 中的元素个数。

13. 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:

```
def Inter(A,B):
    C=SqList()
    for i in range(0,A.getsize()):          #将 A 中属于 B 的元素添加到 C 中
        if B.GetNo(A[i])!=-1:               #时间为 O(n)
            C.Add(A[i])
    return C                                #返回 C
```

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

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

```
def Inter(A,B):
    C=SqList()
    i=j=0
    while i<A.getsize() and j<B.getsize():
        if A[i]<B[j]:                       #忽略较小元素 A[i]
            i+=1
        elif B[j]<A[i]:                     #忽略较小元素 B[j]
            j+=1
        else:                               #仅将公共元素添加到 C 中
            C.Add(A[i])
            i+=1
            j+=1
    return C
```

上述算法的时间复杂度均为 $O(m+n)$,空间复杂度为 $O(\mathrm{MIN}(m,n))$,其中 m 和 n 分别表示 A 和 B 中的元素个数。

15. 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 结点值不为 x，pre 和 p 同步后移一个结点。

对应的算法如下：

```
def Delx(L,x):
    pre=L.head
    p=pre.next                          #p 指向首结点
    while p!=None:                      #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均为 $O(n)$，空间复杂度为 $O(1)$。

16. 解法 1：删除插入法。若单链表 L 的首结点是负整数结点，先跳过单链表 L 开头的连续负整数结点，让 last 指向其最后一个负整数结点；若单链表 L 的首结点不是负整数结点，则让 last 指向头结点。将 pre 置为 last，p 指向 pre 结点的后继结点，用(pre，p)同步指针遍历余下的结点：

① 若 p 结点值＜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：

```
def Move1(L):                           #解法 1
    last=L.head
    p=L.head.next
    while p!=None and p.data<0:         #跳过开头的负整数结点
        last=last.next                  #last、p 同步后移一个结点
        p=p.next
    pre=last;
    while p!=None:                      #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                #last、p 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 之后，再将 L 的头结点作为新单链表的头结点。对应的算法如下：

```
def Move2(L):                           #解法 2
    p=L.head.next
```

```
A=LinkNode()                          #建立单链表 A 的头结点
B=LinkNode()                          #建立单链表 B 的头结点
ta,tb=A,B                             #ta、tb 分别为 A 和 B 的尾结点
while p!=None:
    if p.data<0:                      #找到负整数结点 p
        ta.next=p                     #将 p 结点链接到 A 的末尾
        ta=p
        p=p.next
    else:                             #不是负整数结点 p
        tb.next=p                     #将 p 结点链接到 B 的末尾
        tb=p
        p=p.next
ta.next=tb.next=None                  #两个单链表的尾结点的 next 置为 None
ta.next=B.next                        #将 B 链接到 A 的后面
L.head.next=A.next                    #重置 L
```

17. 解：本题先将集合 A 中的所有元素复制到 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，算法执行后 A、B 集合没有被破坏。对应的算法如下：

```
def Union(A,B):
    C=LinkList()
    t=C.head                          #t 指向 C 的尾结点(采用尾插法建立 C)
    p=A.head.next
    while p!=None:                    #将 A 复制到 C 中
        s=LinkNode(p.data)
        t.next=s; t=s;
        p=p.next
    q=B.head.next;
    while q!=None:                    #将 B 中不属于 A 的元素复制后添加到 C 中
        p=A.head.next;
        while p!=None and p.data!=q.data:
            p=p.next;
        if p==None:                   #结点 q 值不出现在 A 中
            s=LinkNode(q.data)
            t.next=s; t=s;
        q=q.next
    t.next=None;
    return C                          #返回 C
```

上述算法的时间复杂度均为 $O(m\times n)$，空间复杂度为 $O(m+n)$，其中 m 和 n 分别表示 A 和 B 中的元素个数。

18. 解：本题采用二路归并+尾插法建立并集 C 单链表，由 A、B 集合的相关结点重组成 C，算法执行后 A、B 集合被破坏。对应的算法如下：

```
def Union(A,B):
    C=LinkList()
    t=C.head                          #t 指向 C 的尾结点
    p=A.head.next
    q=B.head.next
    while p!=None and q!=None:
```

```
        if p.data < q.data:                    #将较小结点 p 添加到 C 中
            t.next=p
            t=p
            p=p.next
        elif q.data < p.data:                  #将较小结点 q 添加到 C 中
            t.next=q
            t=q
            q=q.next
        else:                                  #公共结点只添加一个
            t.next=p
            t=p
            p=p.next
            q=q.next
    t.next=None
    if p!=None: t.next=p                       #若 A 未遍历完,将余下的所有结点链接到 C 中
    if q!=None: t.next=q                       #若 B 未遍历完,将余下的所有结点链接到 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 中的元素个数。

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. 解:本题将集合 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:

```
def Diff(A,B):
    C=LinkList()
    t=C.head                                   #t 指向 C 的尾结点(采用尾插法建立 C)
    p=A.head.next;
    while p!=None:                             #将 A 中不属于 B 的元素添加到 C 中
        q=B.head.next
        while q!=None and q.data!=p.data:
            q=q.next
        if q==None:                            #结点 p 不在 B 中
            s=LinkNode(p.data)
            t.next=s; t=s;
        p=p.next
    t.next=None;
    return C                                   #返回 C
```

上述算法的时间复杂度均为 $O(m\times n)$,空间复杂度为 $O(m)$,其中 m 和 n 分别表示 A 和 B 中的元素个数。

20. 解:采用二路归并+尾插法建立差集 C 单链表,差集 C 中的元素是属于 A 但不属于 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:

```
def Diff(A,B):
    C=LinkList()
```

```
    t=C.head                          #t 指向 C 的尾结点
    p=A.head.next
    q=B.head.next
    while p!=None and q!=None:
        if p.data<q.data:             #将较小结点 p 添加到 C 中
            t.next=p
            t=p
            p=p.next
        elif 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=None
    if p!=None: t.next=p              #若 A 未遍历完,将余下的所有结点链接到 C 中
    return C
```

上述算法的时间复杂度均为 $O(m+n)$,空间复杂度为 $O(m)$,其中 m 和 n 分别表示 A 和 B 中的元素个数。

21. 解:本题将集合 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:

```
def Inter(A,B):
    C=LinkList()
    t=C.head                          #t 指向 C 的尾结点(采用尾插法建立 C)
    p=A.head.next;
    while p!=None:                    #将 A 中属于 B 的元素添加到 C 中
        q=B.head.next;
        while q!=None and q.data!=p.data:
            q=q.next
        if q!=None:                   #结点 p 值在 B 中出现
            s=LinkNode(p.data)
            t.next=s; t=s;
        p=p.next
    return C                          #返回 C
```

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

22. 解:采用二路归并+尾插法建立交集 C 单链表,在归并中仅将 A、B 的相同值元素添加到 C 中,最后返回 C。对应的算法如下:

```
def Inter(A,B):
    C=LinkList()
    t=C.head                          #t 指向 C 的尾结点
    p=A.head.next
    q=B.head.next
    while p!=None and q!=None:
        if p.data<q.data:             #忽略较小结点 p
            p=p.next
        elif q.data<p.data:           #忽略较小结点 q
```

```
            q=q.next
        else:                                  #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均为 $O(m+n)$，空间复杂度为 $O(\text{MIN}(m,n))$，其中 m 和 n 分别表示 A 和 B 中的元素个数。

23. 解：在递增有序的整数双链表 L 中，值相同的结点一定是相邻的。首先让 pre 指向首结点，p 指向其后继结点，然后循环，直到 p 为空为止：

① 若 p 结点值等于前驱结点（pre 结点）值，通过 pre 结点删除 p 结点，置 $p=$pre.next。

② 否则，pre、p 同步后移一个结点。

对应的算法如下：

```
def Delsame(L):
    pre=L.dhead.next
    p=pre.next
    while p!=None:                         #p 遍历所有其他结点
        if p.data==pre.data:               #p 结点是重复的要删除的结点
            pre.next=p.next                #通过 pre 结点删除 p 结点
            if p.next!=None:
                p.next.prior=pre
            p=pre.next
        else:                              #p 结点不是重复的结点
            pre=p                          #pre、p 同步后移一个结点
            p=pre.next
```

24. 解：本算法仍采用二路归并的思路。若 $k<1$ 或者 $k>A.\text{getsize}()+B.\text{getsize}()$，表示参数 k 错误，抛出异常；否则，用 p、q 分别扫描有序双链表 A、B，用 cnt 累计比较次数（从 0 开始），处理如下：

① 当两个顺序表均没有扫描完时，比较它们的当前元素，每比较一次，cnt 增 1，当 $k==$cnt 时，较小的元素就是返回的最终结果。

② 否则，如果没有返回，让 p 指向没有比较完的结点，继续遍历并且递增 cnt，直到找到第 k 个结点 p 后返回其值。

对应的算法如下：

```
def topk(A,B,k):
    assert k>=1 and k<=A.getsize()+B.getsize()
    p=A.dhead.next
    q=B.dhead.next
    cnt=0
    while p!=None and q!=None:             #A、B 均没有遍历完
        cnt+=1                             #比较次数增加 1
        if p.data<q.data:                  #p 结点较小
```

```
            if cnt==k: return p.data      #已比较 k 次,返回 p 结点值
            p=p.next
        else:                             #q 结点较小
            if cnt==k:return q.data       #已比较 k 次,返回 q 结点值
            q=q.next
    if q!=None: p=q                       #p 指向没有比较完的结点
    cnt+=1                                #从 p 开始累计 cnt
    while cnt!=k and p!=None:             #遍历剩余的结点
        p=p.next
        cnt+=1
    return p.data
```

1.3 第 3 章 栈和队列

1.3.1 练习题

1. 简述线性表、栈和队列的异同。

2. 有 5 个元素,其进栈次序为 abcde,在各种可能的出栈次序中,以元素 c、d 最先出栈(即 c 第一个且 d 第二个出栈)的次序有哪几个?

3. 假设以 I 和 O 分别表示进栈和出栈操作,则初态和终态为栈空的进栈和出栈的操作序列可以表示为仅由 I 和 O 组成的序列,称可以实现的栈操作序列为合法序列(例如 IIOO 为合法序列,IOOI 为非法序列)。试给出区分给定序列为合法序列或非法序列的一般准则。

4. 什么叫“假溢出”? 如何解决假溢出?

5. 假设循环队列的元素存储空间为 data[0..$m-1$],队头指针 f 指向队头元素,队尾指针 r 指向队尾元素的下一个位置(例如 data[0..5],队头元素为 data[2],则 front=2,队尾元素为 data[3],则 rear=4),则在少用一个元素空间的前提下表示队空和队满的条件各是什么?

6. 在算法设计中,有时需要保存一系列临时数据元素,如果先保存的后处理,应该采用什么数据结构存放这些元素? 如果先保存的先处理,应该采用什么数据结构存放这些元素?

7. 给定一个字符串 str,设计一个算法采用顺序栈判断 str 是否为形如“序列 1@序列 2”的合法字符串,其中序列 2 是序列 1 的逆序,在 str 中恰好只有一个@字符。

8. 设计一个算法利用一个栈将一个循环队列中的所有元素倒过来,队头变队尾,队尾变队头。

9. 对于给定的正整数 n($n>2$),利用一个队列输出 n 阶杨辉三角形,5 阶杨辉三角形如图 1.3(a)所示,其输出结果如图 1.3(b)所示。

```
        1                          1
      1   1                        1 1
    1   2   1                      1 2 1
  1   3   3   1                    1 3 3 1
1   4   6   4   1                  1 4 6 4 1
```

(a) n=5时的杨辉三角形　　(b) 输出结果

图 1.3　5 阶杨辉三角形及其生成过程

10. 有一个整数数组 a，设计一个算法将所有偶数位元素移动到所有奇数位元素的前面，要求它们的相对次序不改变。例如，$a=\{1,2,3,4,5,6,7,8\}$，移动后 $a=\{2,4,6,8,1,3,5,7\}$。

11. 设计一个循环队列，用 data[0..MaxSize−1]存放队列元素，用 front 和 rear 分别作为队头和队尾指针，另外用一个标志 tag 标识队列可能空(False)或可能满(True)，这样加上 front==rear 可以作为队空或队满的条件，要求设计队列的相关基本运算算法。

1.3.2　练习题参考答案

1. 答：线性表、栈和队列的相同点是它们元素的逻辑关系都是线性关系；不同点是运算不同，线性表可以在两端和中间任何位置插入和删除元素，而栈只能在一端插入和删除元素，队列只能在一端插入元素、在另外一端删除元素。

2. 答：abc 进栈，c 出栈，d 进栈，d 出栈，下一步的操作如下。

① ba 出栈，再 e 进栈 e 出栈，得到出栈序列 cdbae。

② b 出栈，e 进栈 e 出栈，再 a 出栈，得到出栈序列 cdbea。

③ e 进栈 e 出栈，再 ba 出栈，得到出栈序列 cdeba。

可能的次序有 cdbae、cdeba、cdbea。

3. 答：合法的栈操作序列必须满足以下两个条件。

① 在操作序列的任何前缀(从开始到任何一个操作时刻)中，I 的个数不得少于 O 的个数。

② 整个操作序列中 I 和 O 的个数相等。

4. 答：在非循环顺序队中，当队尾指针已经到了数组的上界，不能再做进队操作，但其实数组中还有空位置，这就叫“假溢出”。解决假溢出的方式之一是采用循环队列。

5. 答：在一般教科书中设计循环队列时，让队头指针 f 指向队头元素的前一个位置，让队尾指针 r 指向队尾元素，这里是队头指针 f 指向队头元素，队尾指针 r 指向队尾元素的下一个位置。这两种方法本质上没有差别，实际上最重要的是能够方便设置队空、队满的条件。

对于题目中指定的循环队列，f、r 的初始值为 0，仍然以 $f==r$ 作为队空的条件，$(r+1)\%m==f$ 作为队满的条件。

元素 x 进队的操作是 data[r]$=x$；$r=(r+1)\%m$。队尾指针 r 指向队尾元素的下一个位置。

元素 x 出队的操作是 $x=$data[f]；$f=(f+1)\%m$。队头元素出队后，下一个元素成为队头元素。

6. 答：如果先保存的后处理，应该采用栈数据结构存放这些元素。如果先保存的先处理，应该采用队列数据结构存放这些元素。

7. 解：设计一个栈 st，遍历 str，将其中'@'字符前面的所有字符进栈，再扫描 str 中'@'字符后面的所有字符，对于每个字符 ch，退栈一个字符，如果两者不相同，返回 False。当循环结束时，若 str 扫描完毕并且栈空，返回 True，否则返回 False。对应的算法如下：

```
def match(str):
    st=SqStack()                          #定义一个顺序栈
```

```
i=0
while i<len(str) and str[i]!='@':
    st.push(str[i])
    i+=1
if i==len(str):                          #没有找到@,返回 False
    return False;
i+=1                                     #跳过@
while i<len(str) and not st.empty():     #str 没有扫描完毕并且栈不空,循环
    if str[i]!=st.pop():                 #两者不等返回 False
        return False
    i+=1
if i==len(str) and st.empty():           #str 扫描完毕并且栈空返回 True
    return True
else:                                    #其他返回 False
    return False
```

8. 解：设置一个栈 st，先将循环队列 qu 中的所有元素出队并进到 st 栈中，再将栈 st 中的所有元素出栈并进到 qu 队列中。对应的算法如下：

```
def Reverse(qu):
    st=SqStack()                         #定义一个顺序栈
    while not qu.empty():
        st.push(qu.pop())
    while not st.empty():
        qu.push(st.pop())
```

9. 解：由 n 阶杨辉三角形的特点可知，其高度为 n，第 $r(1\leqslant r\leqslant n)$ 行恰好包含 r 个数字，在每行前后添加一个 0(第 r 行包含 $r+2$ 个数字)，采用迭代方式，定义一个队列 qu，由第 r 行生成第 $r+1$ 行。

① 先输出第 1 行，仅输出 1，将 0、1、0 进队。

② 当队列 qu 中包含第 r 行的全部数字时(队列中共 $r+2$ 个元素)，生成并输出第 $r+1$ 行的过程是进队 0，出队元素 s(第 1 个元素为 0)，再依次出队元素 t(共执行 $r+1$ 次)，$e=s+t$，输出 e 并进队，重置 $t=s$，最后进队 0，这样输出了第 $r+1$ 行的 $r+1$ 个元素，队列中含 $r+3$ 个元素。图 1.4 所示为生成前 3 行的过程。

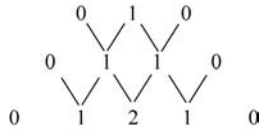

图 1.4　生成前 3 行的过程

对应的算法如下：

```
def YHTriangle(n):
    qu=CSqQueue()                        #定义一个循环队列
    print(1)                             #输出第 1 行
    qu.push(0)                           #第 1 行进队
    qu.push(1)
    qu.push(0)
    for r in range(2,n+1):               #输出第 2~n 行
        qu.push(0)
        s=qu.pop()
```

```
    for c in range(r):                    #输出第 r 行的 r 个数字
        t=qu.pop()
        e=s+t
        print(e,end=' ')
        qu.push(e)
        s=t
    qu.push(0)
    print()
```

10. 解：采用两个队列来实现，先将 a 中的所有奇数位元素进 qu1 队中，所有偶数位元素进 qu2 队中，再将 qu2 中的元素依次出队并放到 a 中，qu1 中的元素依次出队并放到 a 中。对应的算法如下：

```
def Move(a):
    qu1=CSqQueue()                        #存放奇数位元素
    qu2=CSqQueue()                        #存放偶数位元素
    i=0
    while i<len(a):
        qu1.push(a[i])                    #奇数位元素进 qu1 队
        i+=1
        if i<len(a):
            qu2.push(a[i])                #偶数位元素进 qu2 队
            i+=1
    i=0
    while not qu2.empty():                #先取 qu2 队列的元素
        a[i]=qu2.pop()
        i+=1
    while not qu1.empty():                #再取 qu1 队列的元素
        a[i]=qu1.pop()
        i+=1
```

11. 解：初始时 tag=False，front=rear=0，成功的进队操作后 tag=True（任何进队操作后队列不可能空，但可能满），成功的出队操作后 tag=False（任何出队操作后队列不可能满，但可能空），因此这样的队列的四要素如下。

① 队空条件：front==rear and tag==False

② 队满条件：front==rear and tag=True

③ 元素 x 进队：rear=(rear+1)%MaxSize；data[rear]=x；tag=True；

④ 元素 x 出队：front=(front+1)%MaxSize；x=data[front]；tag=False；

设计对应的循环队列类如下：

```
MaxSize=100                               #全局变量，假设容量为 100
class QUEUE:                              #队列类
    def __init__(self):                   #构造方法
        self.data=[None]*MaxSize          #存放队列中的元素
        self.front=0                      #队头指针
        self.rear=0                       #队尾指针
        self.tag=False                    #为 False 表示可能队空，为 True 表示可能队满
    def empty(self):                      #判队空
        return self.front==self.rear and self.tag==False
    def full(self):                       #判队满
        return self.front==self.rear and self.tag==True
```

```
def push(self,x):                          #进队算法
    assert not self.full()                 #检测队满
    self.rear=(self.rear+1)%MaxSize;
    self.data[self.rear]=x
    tag=True                               #至少有一个元素,可能满
def pop(self):                             #出队算法
    assert not self.empty()                #检测队空
    self.front=(self.front+1)%MaxSize
    x=self.data[self.front]
    tag=False                              #出队一个元素,可能空
    return x                               #成功出队
def gethead(self):                         #取队头元素
    assert not self.empty()                #检测队空
    head=(self.front+1)%MaxSize            #求队头元素的位置
    return self.data[head]
```

1.4 第4章 串和数组

1.4.1 练习题

1. 设 s 为一个长度为 n 的串,其中的字符各不相同,则 s 中的互异非平凡子串(非空且不同于 s 本身)的个数是多少?

2. 在 KMP 算法中计算模式串的 next 时,当 $j=0$ 时为什么要取 next[0]=-1?

3. KMP 算法是 BF 算法的改进,是不是说在任何情况下 KMP 算法的性能都比 BF 算法好?

4. 设目标串 s="abcaabbabcabaacbacba",模式串 t="abcabaa",计算模式串 t 的 nextval 函数值,并画出利用改进的 KMP 算法进行模式匹配时每一趟的匹配过程。

5. 为什么数组一般不使用链式结构存储?

6. 如果某个一维整数数组 A 的元素个数 n 很大,存在大量重复的元素,且所有值相同的元素紧跟在一起,请设计一种压缩存储方式使得存储空间更节省。

7. 一个 n 阶对称矩阵存入内存,在采用压缩存储和采用非压缩存储时占用的内存空间分别是多少?

8. 设计一个算法,计算一个顺序串 s 中最大字符出现的次数。

9. 设计一个算法,将一个链串 s 中的所有子串"abc"删除。

10. 假设字符串 s 采用 String 对象存储,设计一个算法,在串 s 中查找子串 t 最后一次出现的位置。例如,s="abcdabcd",t="abc",结果为 4。

(1) 采用 BF 算法求解。

(2) 采用 KMP 算法求解。

11. 设计一个算法,将含有 n 个整数元素的数组 $a[0..n-1]$循环右移 m 位,要求算法的空间复杂度为 $O(1)$。

12. 设计一个算法,求一个 n 行 n 列的二维整型数组 a 的左上角-右下角和右上角-左下角两条主对角线的元素之和。

1.4.2　练习题参考答案

1. 答：由串 s 的特性可知，一个字符的子串有 n 个，两个字符的子串有 $n-1$ 个，3 个字符的子串有 $n-2$ 个，……，$n-2$ 个字符的子串有 3 个，$n-1$ 个字符的子串有两个，所以非平凡子串的个数 $=n+(n-1)+(n-2)+\cdots+2=n(n+1)/2-1$。

2. 答：在 KMP 算法中，当目标串 s 与模式串 t 匹配时，若 $s_i=t_j$，执行 $i++$，$j++$（称为情况 1）；若 $s_i\neq t_j$（失配处），i 位置不变，置 $j=$next[j]（称为情况 2）。若失配处是 $j=0$，即 $s_i\neq t_0$，那么从 s_i 开始的子串与 t 匹配一定不成功，下一趟匹配应该从 s_{i+1} 开始与 t_0 比较，即 $i++$，$j=0$，为了与情况 1 统一，置 next[0]$=-1$（即 $j=$next[0]$=-1$），这样再执行 $i++$，$j++\rightarrow j=0$，从而保证下一趟从 s_{i+1} 开始与 t_0 进行匹配。

3. 答：不一定。例如，$s=$"aaaabcd"，$t=$"abcd"，采用 BF 算法时需要 4 趟匹配，比较字符的次数为 10。采用 KMP 算法，求出 t 对应的 next$=\{-1,0,0,0\}$，同样也需要 4 趟匹配、10 次字符比较，另外还要花时间求 next 数组，所以并非在任何情况下 KMP 算法的性能都比 BF 算法好，只能说在平均情况下 KMP 算法的性能好于 BF 算法。

4. 答：模式串 t 的 nextval 函数值如表 1.1 所示，利用改进 KMP 算法的匹配过程如图 1.5 所示。

表 1.1　模式串 t 的 nextval 函数值

j	0	1	2	3	4	5	6
$t[j]$	a	b	c	a	b	a	a
next[j]	-1	0	0	0	1	2	1
nextval[j]	-1	0	0	-1	0	2	1

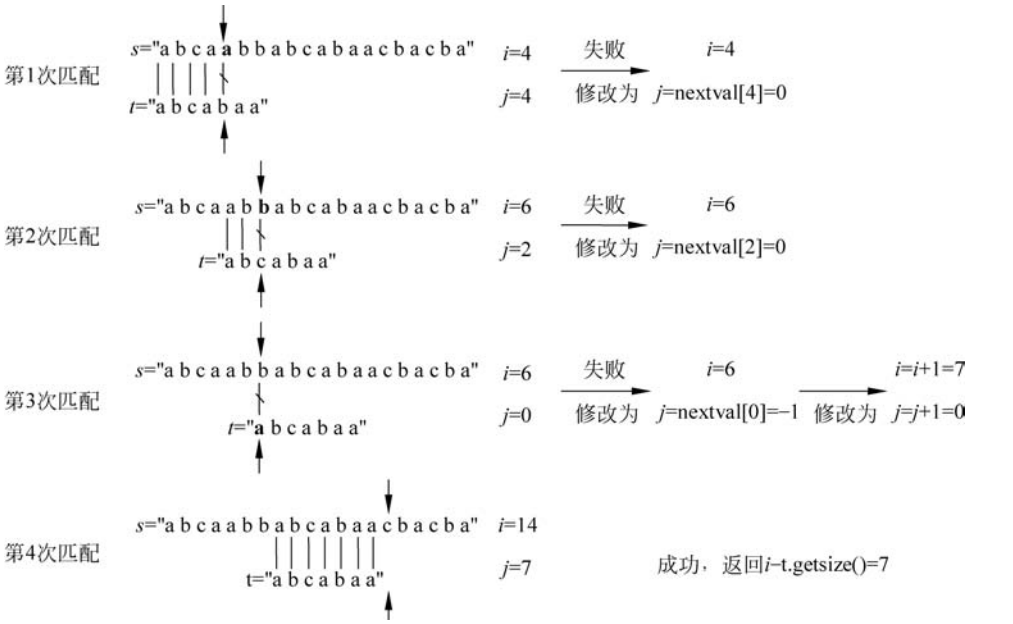

图 1.5　KMP 算法的匹配过程

5. 答：因为数组的主要操作是存取元素，通常没有插入和删除操作，使用链式结构存储时需要额外占用更多的存储空间，而且不具有随机存取特性，使得相关操作更复杂。

6. 答：采用元素类型形如“[整数，个数]”的列表压缩存储，例如数组 A 为{1,1,1,5,5,5,5,3,3,3,3,4,4,4,4,4,4}，共有 17 个元素，对应的压缩存储为[[1,3],[5,4],[3,4],[4,6]]，在压缩列表中只有 8 个整数。从中看出，重复元素越多，采用这种压缩存储方式越节省存储空间。

7. 答：若采取压缩存储，其容量为 $n(n+1)/2$；若不采用压缩存储，其容量为 n^2。

8. 解：置最大字符 mch 为 s 的首字符，mcnt 表示其出现次数，i 扫描其他字符，x 为序号为 i 的字符。

① 若 $x>$mch，则 x 为新的最大字符，置 $x=$mch，mcnt=1。

② 若 $x==$mch，当前最大字符 x 的出现次数 mcnt 增加 1。

最后返回 mcnt。对应的算法如下：

```
def maxcount(s):
    mcnt=1;
    mch=s[0]
    for i in range(s.getsize()):
        if s[i]>mch:
            mch=s[i]
            mcnt=1
        elif s[i]==mch:
            mcnt+=1
    return mcnt
```

9. 解：用 pre、p 一对同步指针遍历链串 s(初始时 pre=s.head，p=pre.next)，当 p 结点及其后继两个结点合起来为"abc"时，通过 pre 结点删除该子串，p=pre.next 继续遍历删除，否则 pre、p 同步后移一个结点。对应的算法如下：

```
def delabc(s):
    pre=s.head
    p=pre.next
    if p==None or p.next==None or p.next.next==None:
        return
    while p!=None and p.next!=None and p.next.next!=None:
        if p.data=='a' and p.next.data=='b' and p.next.next.data=='c':
            pre.next=p.next.next.next       #查找到"abc"子串实施删除
            p=pre.next
        else:
            pre=pre.next                    #pre、p同步后移一个结点
            p=pre.next
```

10. 解：用 lasti 记录串 s 中子串 t 最后一次出现的位置(初始为−1)，修改 BF 和 KMP 算法，在 s 中找到一个 t 后用 lasti 记录其位置，此时 i 指向 s 中 t 子串的下一个字符，将 j 置为 0 继续查找。对应的算法如下：

```
MaxSize=100
#(1)小题的算法
```