第 1 章　常用算法和数据结构

大纲要求：

- 排序算法。
- 查找算法。
- 数据结构(线性表、栈、队列、数组、树、图)。

1.1　排 序 算 法

1.1.1　考点辅导

1. 选择排序

若设 R[1...n]为待排序的 n 个记录，R[1...i−1]已按照主关键字由小到大排序，且任意 x∈R[1...i−1]，y∈R[i...n]满足 x.key≤y.key，则选择排序的主要思路如下。

(1) 反复从 R[i...n]中选出关键字最小的结点 R[k]。

(2) 若 i≠k，则将 R[i]与 R[k]交换，使得 R[1...i]有序且保持原来的性质。

(3) i 增 1，直到 i 为 n。

为方便描述，被查找的顺序表 C 类型定义如下：

```
#define MAXSIZE 1000                /*顺序表的长度*/
typedef int KeyType;                /*关键字类型为整数类型*/
typedef struct{
    KeyType key;                    /*关键字项*/
    InfoType otherinfo;             /*其他数据项*/
}RecType;                           /*记录类型*/
typedef struct {
    RecType  r[MAXSIZE+1];          /*r[0]空作为哨兵*/
    int length;                     /*顺序表长度*/
}SqList;                            /*顺序表类型*/
```

顺序存储线性表的选择排序算法如下：

```
void Sqsort(SqList &q)
{
    int i, j, k, temp;
    for(i=0;i<q.length-1;i++)
   {
        k=i;
        for(j=i+1;j<q.length;j++)
            if(q.r[j].key<q.r[k].key)k=j;  /*选择关键字最小的记录*/
            if(k!=i)
            {
                temp=q.r[k];
                q.r[k]=q.r[i];
                q.r[i]=temp;
```

```
            }
        }
    }
```

可见，选择排序不管原先序列是否有序，其排序需要比较的次数均为 n(n−1)/2；同时，由于相等的两个元素，位置相对在前的可能被交换到后面，故该选择排序是不稳定的。

2. 直接插入排序

若设 R[1...n]为待排序的 n 个记录，R[1...i−1]已按照主关键字由小到大排序，则直接插入排序的主要思路如下。

(1) 寻找 R[i]在 R[1…i−1]中的插入位置，确保 R[i]插入后保持有序。

(2) i 增 1，若 i 小于等于 n，则转到(1)执行，否则结束。

顺序线性存储结构下的直接插入排序算法如下：

```
void Dinsert(SqList &q)
{
  int i,j,k;
  for(i=1;i<q.length;i++)                              /*q.r[0]为有序*/
  {
      for(t=q.r[i],j=i-1;j>=0 && t.key<q.r[j].key;j--)/*找到插入的位置*/
          q.r[j+1]=q.r[j];
      q.r[j+1]=t;
  }
}
```

【点评】

(1) 对 n 个结点的线性表采用直接插入排序，当线性表已是从小到大排序时，内循环每执行一次，只需要进行 1 次比较，整个排序过程只进行 n−1 次比较。当线性表已是从大到小排序时，对外循环执行 i 次，内循环要进行 i 次比较，整个排序过程需要进行 n(n−1)/2 次比较。可见，对 n 个结点的线性表采用直接插入排序，最少比较次数为 n−1，最多比较次数为 n(n−1)/2。

(2) 直接插入排序是在有序表的基础上进行的，所以排序效率较高且比较稳定。

3. 希尔排序

希尔排序(Shell Sort)的基本思路为：把直接插入方法分成插入步长由大到小不同的若干趟来进行，一开始步长较大，相当于把序列分成几个子表。对每个子表来说，因为其结点少，直接插入排序的效率会很高。以后各趟逐步减小步长，子表的结点也越来越多，但是子表中的结点已经进行过前一趟的大步长的直接插入排序，有相当多的结点已基本有序。这使得后一趟的插入排序能充分利用前一趟的排序结果。当步长降到 1 时，只要对基本有序的线性表进行一趟直接插入排序即可。

初次取线性表的一半长度为步长，以后每次减半，直到步长为 1，希尔排序的算法如下：

```
void Shsort(SqList &q)
{
 int j,k,h,y;                          /*h 为步长*/
 for(h=q.length/2;h>0;h/=2)
   for(j=h;j<n;j++)                    /*对每个子表进行直接插入排序*/
     {
```

```
    y=q.r[j];
        for(k=j-h;k>=0&&y.key<q.r[k].key;k-=h)
        q.r[k+h]=q.r[k];            /*找到插入的位置并移动*/
    q.[k+h]=y;
    }
}
```

4. 冒泡排序

若设 R[1...n]为待排序的 n 个记录，且假设为从上至下纵向排列，则要求将 n 个给定记录由小到大排序的冒泡排序(Bubble Sort)的基本思路如下。

(1) 对当前还未排好序的、指定范围内的全部结点，自上而下对相邻的两个结点依次进行比较和调整，让关键字较大的结点往下沉，关键字较小的结点往上冒。即若 R[j].key>R[j+1].keg，则将 R[j].key 与 R[j+1].key 交换。

(2) 初始化时，排序范围是 R[1...n]，以后的排序范围由前一遍的扫描结果确定：当自上而下将当前排序范围内的结点执行一遍比较之后，若最后往下沉的结点是 R[j]，R[j]下沉到 R[j+1]的位置，R[j+1]以下的结点比较后会发现它们都不再需要交换。因此，下一趟的排列范围可以缩减为从 R[1]到 R[j]。

(3) 整个排列过程最多执行 n−1 趟。若某一趟的比较没有结点交换，所有相邻结点的排列顺序都与排序要求一致，则线性表为有序的。

顺序线性存储结构下的冒泡排序算法如下：

```
void bubblesort(SqList &q)
 {
int j,p=q.length-1,h,t;         /*p 用来记录一趟排序最后下沉的结点位置*/
for(h=1;h<=p;)
  for(j=0;j<p-1;j++)
    if(q.r[j].key>q.r[j+1].key){
      temp=q.r[j]; q.r[j]=q.r[j+1];q.r[j+1]=temp;
      p=j;
      }
}
```

可见，对 n 个结点的线性表采用冒泡排序，外循环最多执行 n−1 趟，每一趟最多执行 n−1 次比较；第 2 趟最多执行 n−2 次比较；依次类推，第 n−1 趟最多执行 1 次比较。因此，整个排序过程最多执行 n(n−1)/2 次比较。由于关键字相等的结点不交换，所以冒泡排序算法是稳定的。

5. 快速排序

快速排序(Quick Sort)的主要思路为：通过对线性表序列的一趟扫描使某个结点移到中间的某个位置，且使其左边序列各结点的关键字都比该结点的关键字小，而其右边序列各结点的关键字都不比该结点的关键字小，常称这样的一次扫描为“划分”。然后，对左、右序列进行同样的处理，直到所有序列均只包含一个结点为止，这样便可将原线性表排好序。

快速排序可以看作是冒泡排序的改进，冒泡排序可以看作是快速排序的退化，即每趟划分总是在同一端进行。

若设待排序的记录序列{ R1，R2，…，Rn }为 R[1...n]，则对其按关键值的非递减序列

进行快速排序的算法如下：

```
void QuickSort1(SqList &R, int s, int t)
{
    /*对 n 个元素的数组 R 进行由小到大排序*/
    int low, high;
    low=s;
    high=t;
    privotkey=R.r[s].key;
    R.r[0]=R.r[s];
    while(low<high)
    {
        while(high>low && R.r[high].key>privotkey)
            high--;
        if (low<high)
        {
            R.r[low]=R.r[high];     /*将比枢轴小的记录移动到低端*/
            low++;
        }
        while(low<high && R.r[low].key<=privotkey)
            ++low;
        if(low<high)
        {
            R.r[high]=R.r[low];     /*将比枢轴大的记录移动到高端*/
            high--;
        }
    }
    R.r[low]=R.r[0];                /*枢轴记录到位，即完成一趟排序*/
    QuickSort1(R, s, i-1);          /*对左区间递归排序*/
    QuickSort1(R, i+1, t);          /*对右区间递归排序*/
}                                   /*完成快速排序*/
```

可见，快速排序可能会破坏两个相等记录的原来次序，因而快速排序算法是不稳定的。

6. 将顺序存储结构上的排序算法移植到链表上

很多优秀的算法都是建立在顺序存储结构上的，如何在链式存储结构上实现这些优秀算法，是考生应注意的问题。近年来，在程序员水平考试中也出现了这样的题目。这里，我们通过在顺序存储结构和链式存储结构两种存储结构上实现快速排序来说明。顺序存储结构下的算法 QuickSort1 可以拓展到链式存储结构下的算法 QuickSort2。

下面的算法就是将算法 QuickSort1 拓展到链式存储结构下的算法 QuickSort2：

```
typedef  struct  node {
    int data;                   /*结点的数据域*/
    struct  node  *next;        /*结点的指针域或链域*/
} Slink;
QuickSort2(Slink *head, Slink *tail)
                                /*将不带头结点的单链表 head 进行由小到大排序*/
                                /*主程序调用时，tail=NULL*/
{
    Slink *p=head->next , *mid, *midpre, *pre, *r;
    int temp;
    midpre=head;
    mid=p;
```

```
    if(!p)return 1;
    pre=p;
    p=p->next;
    while(p!=tail)
    {
        r=p->next;
        if(p->data<=head->next->data)
        {
            pre->next=r;              /*断开与 p 结点的链接*/
            midpre->next=p;           /*将 p 结点插入到 mid 之前*/
            p->next=mid;
            midpre=p;
            p=r ;
        }
        else
        {                             /*结点的位置保持不变*/
            pre=p;
            p=r;
       }
    }
    QuickSort2(head, midpre);         /*对前一部分链表进行快速排序*/
    QuickSort2(mid->next,tail);       /*对后一部分链表进行快速排序*/
}                                     /*QuickSort2*/
```

可见，只要充分领会顺序存储结构下的算法思想，熟悉链表存储结构就可以通过掌握顺序存储结构下的算法得到链表存储结构下的相应算法。

7. 堆排序

首先，要认真掌握堆的定义，然后才能进一步理解建堆的算法。堆的定义为：n 个关键字序列 k_1，k_2，k_3，…，k_n 称为堆，当且仅当该序列满足如下性质(也称堆性质)：①$k_i \leqslant k_{2i}$ 且 $k_i \leqslant k_{2i+1}$ 或 ②$k_i \geqslant k_{2i}$ 且 $k_i \geqslant k_{2i+1}$($1 \leqslant i \leqslant n/2$)。满足第①种情况的堆称为小根堆，满足第②种情况的堆称为大根堆。这里仅讨论第①种情况。

本质上，堆排序在排序过程中，是将顺序表中存储的数据看成一棵完全二叉树，利用完全二叉树中双亲结点和孩子结点之间的内在关系来选择关键字最小记录。堆排序分建堆和堆调整两个过程。

建堆过程为：堆排序的过程是一个不断从堆顶到叶子的调整过程(又称“筛选”)。从一个无序序列建堆的过程就是一个反复筛选的过程。若将此序列看成是一个完全二叉树，则最后一个非终端结点是第 n/2 个元素，因此筛选只需要从第 n/2 个元素开始。

堆调整过程为：将该完全二叉树中最后一个元素替代已输出的结点。若新的完全二叉树的根结点小于左右子树的根结点，则直接输出。反之，则比较左右子树根结点的大小。若左子树的根结点小于右子树的根结点(或右子树的根结点小于左子树的根结点)，则将左子树(或右子树)的根结点与该完全二叉树的根结点进行交换。重复上述过程，调整左子树(或右子树)，直至叶子结点，则新的二叉树满足堆的条件。

堆排序的算法如下：

```
void HeapSort(Sqlist &R)              /*R.r[1...R.length]看成是完全二叉树的*/
                                      /*顺序存储结构*/
{                                     /*建立小根堆*/
```

```
    for(i=R.length/2; i>0; i--)    /*从最后一个内部结点开始调整*/
        HeapAdjust(R, i, R.length);
}
void HeapAdjust(Sqlist & R, int i, int m)
{                          /*i 为堆调整的位置，m 为堆的大小，该函数建立小根堆*/
    R.r[0]=R.r[i];
    for(j=2*i; j<=m; j*=2)
    {
        if(j<m && R.r[j].key>R.r[j+1].key)++j;
                                   /*找孩子结点中关键字最小的*/
        if(R.r[0].key<=R.r[j].key) /*调整结束*/
            break;
        R.r[i]=R.r[j];      /*使得根结点的关键字小于等于左右孩子的关键字*/
        i=j;
    }
    R.r[s]=R.r[0];
}
```

堆排序算法的时间复杂度为 $O(n\log_2 n)$，这是一个不稳定的排序方法。

8. 合并排序

合并排序(Merging Sort)的基本思想是：将两个或者两个以上的有序子表合并成一个新的有序表。对于两个有序子表合并成一个有序表的 2-路合并排序来说，初始时，把含有 n 个结点的待排序序列看作由 n 个长度都为 1 的有序子表所组成，将它们依次两两合并得到长度为 2 的若干有序子表，再对它们做两两合并，直到得到长度为 n 的有序表，排序即告完成。

2-路合并排序算法如下：

```
void mergesort(SqList &q)
{
    SqList r;
    r.length=q.length;
    int len=1,f=0;
    while(len<q.length)
    {                                  /*交替地在 q.r[]和 r.r[]之间来回合并*/
        if(!f)
            mergepass(&q,&r,len);
        else
            mergepass(&r,&q,len);
        len*=2;                        /*一趟合并后，有序段结点数加倍*/
        f=1-f;                         /*控制交替合并*/
        if(f)
            q.r[0...q.length-1]=r.r[0...r.length-1]; }
}
```

其中，一趟合并和相邻两个有序段合并的函数如下：

```
void mergerpass(SqList &q, SqList &r, int len)
{
    int start, end;
    start=0;
    while(start+len<q.length)
    {                                  /*至少还有两个有序段*/
        end=start+2*len-1;
```

```
        if(end>=q.length)
        {
            end=q.length-1;             /*最后一个段可能不足 len 个结点*/
            mergestep(&q,&r,start, start+len-1, end);
                                        /*相邻有序段合并*/
            start=end+1;
        }
    }
    if(start<n)                 /*还剩下一个有序段时将其从 q.r[]复制到 r.r[]*/
        for(;start<q.length;start++)
            r.r[start]=q.r[start];
}

void mergestep(SqList &q,SqList &r, int start, int middle, int end)
                                                    /*将两个相邻段合并成一个段*/
{
    int j,k,l;
    k=l=start;
    j=middle+1;
    while(l<=middle&&j<=end)
    {                                               /*当两个有序段都未结束时循环*/
        if(q.r[l]<=q.r[j])r.r[k++]=q.r[l++];    /*取其中小的元素复制*/
        else r.r[k++]=q.r[j++];
        while(l<=middle)
            r.r[k++]=q.r[l++];                  /*复制还未合并完的剩余部分*/
        while(j<=end)
            r.r[k++]=q.r[j++];
    }
}
```

可以看出，合并排序是一种稳定的排序方法，但需要和待排序序列一样多的辅助存储空间。

9. 如何在 r 进制下运用基数排序

如何依据实际情况，对任意 r 进制运用基数排序是基数排序算法设计的关键。下面通过例子来说明和理解基数排序。

若下列 C 程序的功能是：用基数排序法对读入的 n 个无符号整数进行排序(排成从小到大的次序)，请在程序空缺处填上适当字句，使其能正确执行。具体如下：

```
#define  M 1000
#define  Radix 16
#define  Bits 4
rsort(int a[], int n)
{
    int Bit,divisor, i;
    int count[Radix];
    divisor=1;
    for(Bit=1;Bit<=Bits; Bit++)
    {
        for(i=0;i<=Radix-1;i++)
            count[i]=0;
        for(i=1;i<=n;i++)
            count[(a[i]/divisor)%Radix]=count[(a[i]/divisor)%Radix]+1;
        for(i=0;i<=Radix-1;i++)
```

```
            count[i]=count[i]+count[i-1];
        for(i=n;i>=1;i--)
        {
            t[(1)]=a[i];
            count[(a[i]/ divisor)%Radix]:=(2);
        }
        for(i=1;i<=n;i++)
            (3);
        divisor=(4);
    }
}
```

基数排序的两种主要方法是：最高位优先 MSD(Most Significant Digit first)和最低位优先 LSD(Least Significant Digit first)。这里的程序采用最低位优先 LSD 方法对输入的 n 个整数进行排序，采用的主要思想是把各整数看成由 4 个 16 位数组成；若从低到高分别称为 L1、L2、L3、L4，则执行如下步骤。

(1) 初始化计数器 count[0...15]，使各个 count[i]=0(0≤i≤15)。

(2) 统计 L1 为 0、1、 …、15 的各个整数个数分别到 count[0...15]。

(3) 将 count[0]、count[1]、…、count[15]合并起来得到 L1 为 j(0≤j≤15)的整数在一趟排序后的起始位置；如 a[j]在 count[(a[j]> divisor%) Radix]到 count[(a[j]/divisor)%Radix+1]−1 之间；同时 L1 相同的整数的前后位置没有明显的界限，只需要根据读入的次序来定。

(4) 按照得到的各个整数 a[i]的位置 count[j]将该元素登记在相应的位置上，同时 count[j]增加 1，以便存放下一个和整数 a[i]的 L1 相同的元素。

(5) 对 L2、L3、L4 重复上述过程。

通过仔细分析，不难得到如下答案。

(1) count[(a[i] / divisor)% Radix]

(2) count[(a[i] / divisor)% Radix]+1

(3) a[i]=t[i]

(4) divisor*16

进一步推广，若把整数看成八进制或其他进制，同样也能得到问题的答案。

10. 败者树

采用败者树的目的是在进行最小键值的查找时减少比较次数。

败者树是一棵完全二叉树，其中每个结点的键值都取其两个子结点的键值中的较小者，因此，根结点的键值是这棵树中所有结点的键值中最小的。这就像 k 个参加淘汰赛的球队，胜者(值较小者)进入下一轮的比赛，根结点为冠军(值最小者)。

败者树的构造过程是：对具有 k 个记录的序列，首先用这 k 个记录作为叶结点，然后把相邻的两个结点进行比较，把键值小的记录(优胜者)作为这两个结点的父结点，按此方法自下而上一层一层地产生败者树的结点。为了节约内存空间，非叶子结点可不包含整个记录，只要存放记录的键值及指向该记录的指针即可。

败者树的根结点的值是构成败者树的元素中最小的，在后面的应用中，往往把根结点的值输出并用一个新的元素替换，要求构成新的败者树，这时只要在原来的败者树的基础上进行调整即可。调整仅在从根到新加入的叶子结点的树枝上的结点及其兄弟结点之间进行，自下而上进行比较并调整其父结点。

11. k 路归并法

有了 m 个初始归并段(都是有序段)，便可进行 k 路归并了，即将 k 个初始归并段采用某种方法进行归并产生一个段，这样 m 个初始归并段便产生多个更大的段，然后对这些段再进行归并，如此下去，直到只生成一个段为止，这个段就是最后生成的归并段。

在内存里进行 k 路归并的方法很多。当归并路数 k 较大时，为了减少合并时的比较次数，常采用败者树进行合并的方法，其合并过程如下。

(1) 用参加合并的 k 个有序段的第一个记录构造一棵初始败者树，该树中的根结点就是这 k 个记录中具有最小键值的记录。

(2) 把败者树根结点所代表的记录送到输出缓冲区。

(3) 输出记录所在的有序段的下一个记录代替输出记录的位置，调整败者树。

(4) 重复步骤(2)和(3)，直到 k 个有序段的所有记录都输出为止。

对于总共有 n 个记录的 k 个有序段的合并过程，如果采用败者树进行合并，那么要选取键值最小的记录，在建立败者树时需要进行 k−1 次比较，此后每次调整败者树只要进行 $\log_2 k$ 次比较即可(因为树中保留了以前的比较结果)。所需总的比较次数为 $k+n\log_2 k$，当 n>>k 时，总的比较次数约为 $n\log_2 k$。

1.1.2　典型例题分析

阅读以下说明和 C 语言函数，将应填入__(n)__处的字句写在答题纸的对应栏内。(2007 年上半年试题四)

【说明】

函数 sort(NODE *head)的功能是：用冒泡排序法对单链表中的元素进行非递减排序。对于两个相邻结点中的元素，若较小的元素在后面，则交换这两个结点中的元素值。其中，head 指向链表的头结点。排序时，为了避免每趟都扫描到链表的尾结点，设置一个指针 endptr，使其指向下趟扫描需要到达的最后一个结点。例如，对于如图 1-1(a)所示的链表进行一趟冒泡排序后，得到如图 1-1(b)所示的链表。

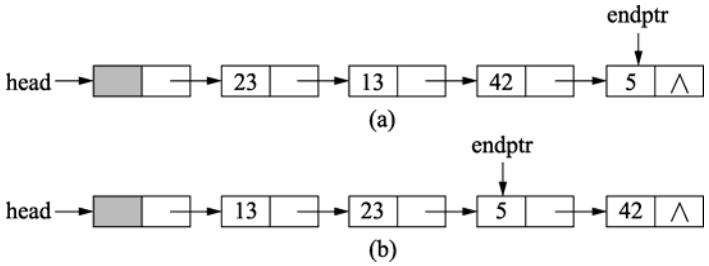


图 1-1　排序

链表的结点类型定义如下：

```
typedef struct Node {
   int data;
   struct Node *next;
}NODE;
```

【C 语言函数】

```
void sort(NODE *head)
{
    NODE *ptr,*preptr,*endptr;
    int tempdata;
    ptr = head -> next;
    while (1)                                    /*查找表尾结点*/
        ptr = ptr -> next;
    endptr = ptr;                                /*令 endptr 指向表尾结点*/
    ptr =(2) ;
    while(ptr != endptr)
    {
        while((3) )
        {
            if (ptr->data > ptr->next->data)
            {
                tempdata = ptr->data;           /*交换相邻结点的数据*/
                ptr->data = ptr->next->data;
                ptr->next->data = tempdata;
            }
            preptr =(4) ;ptr = ptr -> next;
        }
        endptr =(5) ;ptr = head->next;
    }
}
```

分析：从(1)处代码中可知 ptr 最后应该指向表尾结点。所以(1)处应为 ptr -> next。进行冒泡排序时，不断调整元素的位置，最终使最大元素放到表的最后，所以(2)处应为 head->next。(3)处的循环条件应该是扫描的结点，不是最后一个结点，所以(3)处应为 ptr！= endptr。ptr 每向后修改一次，preptr 就要修改一次，所以(4)处应为 ptr，(5)处应为 preptr。

答案：

(1) ptr –> next

(2) head–>next

(3) ptr！=endptr，或其他等价形式

(4) ptr

(5) preptr

1.1.3 同步练习

1. 下面是一个链接存储线性表的直接插入排序函数。把未排序序列中的第一个结点插入到已排序序列中。排序完毕，链表中的结点按结点值从小到大链接，请在空缺处填上适当内容，每个空缺只填一个语句。

```
typedef struct  node{
    char data;
    struct node *link;
}NODE;
NODE *insertsort (NODE *h)
{
    NODE *t, *s, *u, *v;
```

```
    s=h->link;
    h->link=NULL;
    while(s!=NULL)
    {
        for(t=s, v=h; v!=NULL && v->data<t->data;  (1),  (2));
        s=s->link;
       if(v==h)  (3);
        else   (4)   ;
        (5)   ;
    }
    return h;
}
```

2. 请仔细阅读下面的堆排序算法。待排序记录存储在一维数组中，说明如下：

```
typedef struct node{
      int key;
      datatype info;
}Node;
typedef Node[n+1] heaptype;
```

函数heapsort的功能是将数组heap中的前n个记录按关键码值递减的次序排序。heapsort调用 sift 函数，sift 函数的参数 heap、h 和 r 具有如下含义：调用 sift 函数时，以 heap[h+1], heap[h+2], …, heap[r/2]为根的子树已经成为堆；sift 函数执行后，以 heap[h], heap[h+1], heap[h+2], …, heap[r/2]为根的子树都成为堆。

```
sift(heaptype &heap, int h, int r)
{
    int i, j, finish;
    Node x;
    i=h;
    x=heap[i];
    j=2*i;
    finish=0;
    while( (1) )
    {
        if(j<r && heap[j].key>heap[j+1].key)
            j++;
        if(x.key>heap[j].key)
        {
            (2);
        }
        else
            finish=1;
    }
    (3);
}
heapsort(heaptype &heap, int n)
{
    int h, r, i, j;
    Node x;
    for(h=n/2; h>=1; h--)
        (4);
    for(r=n; r>=2; r--)
    {
```

```
            x=heap[1];
            heap[1]=heap[r];
            heap[r]=x;
            (5);
        }
    }
```

(1) 请在 sift 函数和 heapsort 函数的空缺处填入适当内容，使它们能正确执行。

(2) 如果调用 heapsort 函数的参数值 n=10，那么在 heapsort 的执行过程中 sift 函数被调用了多少次？

3. 下面是一改进了的快速排序算法，试补充其中的空白语句，并分析该算法所需的最大递归空间是多少。

```
qsort(Sqlist &R, int m, int n)
/*R.r[m].key<R.r[n].key*/
{
    int i, j, k;
    while(m<n)
    {
        i=m;
        j=n+1;
        k=R.r[m].key;
        while(i<i)
        {
            while(R.r[i].key<=k)i++;
            while(R.r[j].key>=k)j--;
            if(i<j) R.r[i]<-->R.r[j];
        }
        R.r[m]<-->R.r[i];
        if(n-j>=j-m)
        {
            qsort(R, (1) , (2) );
            (3) ;
        }
        else{
            qsort(R, (4) , (5) );
            (6) ;
        }
    }
}
```

4. 下面的排序算法的思想是：第一趟比较将最小的元素放在 r[1]中，最大的元素放在 r[n]中；第二趟比较将次小的放在 r[2]中，将次大的放在 r[n−1]中；依次类推，直到待排序列递增有序(注：<-->代表两个变量的数据交换)。

```
void sort(sqlist &r, int n)
{
    i=1;
    While( (1) )
    {
        min=max=i;
        for(j=i+1; (2) ; ++j)
        {
            if( (3) )min=j;
```

```
            else if(r[j].key>r[max].key)max=j;
        }
        if(__(4)__)r[min]<-->r[i];
        if(max!=n-i+1)
        {
            if(__(5)__)r[min]<-->r[n-i+1];
            else __(6)__;
        }
        i++;
    }
}//sort
```

5. 选择排序，试补充其中的空白语句。

```
void selectionsort( datatype A[], int N)
{
    for(int last=__(1)__; last>=1; last--)
        int L=indexoflargest(A, last+1);
    datatype temp;
    temp=__(2)__;
    A[L]=A[last];
    A[last]=__(3)__;
}
int indexoflargest(datatype A[], int size)
{
    int indexsofar=__(4)__;
    for(int currentindex=1; currentindex<size; currentindex++)
    { if(A[currentindex]>A[indexsofar])
        __(5)__;
    }
    return __(6)__;
}
```

1.1.4　同步练习答案

1.

分析：本题的主要思路是：将当前结点插入到该结点前的有序单链表中，直到当前结点为空。将当前结点插入到该结点前的有序单链表的过程类似顺序表的策略，从单链表的表头开始查找，直到找到该结点应插入的位置，然后完成插入任务。

答案：

(1) u=v

(2) v=v−>link

(3) h=t

(4) u−>link=t

(5) t−>link=v

2.

分析：本题的堆排序的功能是将数组 heap 中的前 n 个记录按关键码值递减的次序排序，因此要构造一个“小根堆”，需要先选择一个关键字作为最小的记录，并与序列中最后一个记录交接，然后对序列中前 n−1 个记录进行筛选，重新将其调整为一个“小根堆”，如此反复直至排序结束。

答案：依据上述分析，本题的答案如下。

(1) j≤r && !finish

(2) h[i]=h[j];h[j]=x; i=j

(3) j=2*j

(4) sift(h,k,n)

(5) sift(h,1,r−1)

若 n=10，那么 sift 共被调用了 5+9=14 次。

3.

分析：本题修改的快速排序算法相对 1.1.1 节中的快速排序算法而言，多了对划分后所得两个子序列的长度进行判断的 if 复合语句，进而先对长度较短的子序列进行快速排序，此目的是将快速排序的运行栈空间降为 $O(\log_{10} n)$。

答案：依据上述分析，本题的答案如下。

(1) m

(2) j−1

(3) m=j+1

(4) j+1

(5) n

(6) n=j−1

4.

分析：仔细分析，可以看出本题采用的算法是选择排序算法的变种，传统的选择排序算法是每次从未排序序列中找一个最小关键字值的记录；这里的算法是每次找到一个最小关键字值的记录，同时找到一个最大关键字值的记录，使两端有序。

答案：依据上述分析，本题的答案如下。

(1) I<=n/2

(2) j<=n−i+1

(3) r[j].key<r[min].key

(4) min!=i

(5) max==min

(6) r[max]<-->r[n−i+1]

5.

分析：本题采用的算法是选择排序算法的变形，第一次从所有的元素中找一个最大的元素与 A[N−1]交换；第二次从剩下的元素中找一个最大的元素与 A[N−2]交换；依次类推，直到所有的元素都递增有序。

答案：依据上述分析，本题的答案如下。

(1) N−1

(2) A[L]

(3) temp

(4) 0

(5) indexsofar=currentindex

(6) indexsofar

1.2　查 找 算 法

1.2.1　考点辅导

1. 查找的基本概念

所谓“查找”(检索)就是在一个含有众多数据元素(记录)的查找表中找出某个“特定的”数据元素。查找表是一种非常重要的数据结构，是由同一类型的数据元素(记录)构成的集合。在查找表中，通常通过查找其关键字是否等于给定值来确定查找是否成功。若查到其关键字等于给定值的记录，则称“查找成功”，否则，称“查找失败”。

对查找表而言，除了按关键字查找外，查找表的插入和删除是对查找表进行的另外两个基本操作。

关键字是数据元素(记录)中某个数据项的值，可以标识一个记录，若能唯一标识，则称为主关键字，否则，称为次关键字。

考生特别要注意的是：在查找中是和关键字进行比较，而不是和数据元素进行比较，因而在算法描述中要体现关键字比较的特征。同时，考生要特别注意查找在顺序存储结构和链式存储结构上的区别。

查找的效率通过平均检索长度(ASL)和所需的辅助空间来确定。在查找其关键字等于给定值的过程中，需要与给定值进行比较的关键字个数的期望值称为检索成功时的平均检索长度。

2. 静态查找

静态查找的算法有顺序查找、折半查找、分块查找和静态树查找等，其中顺序表上的顺序查找、折半查找是静态查找的重点。被查找的顺序表 C 类型定义如下：

```
#define MAXSIZE 1000            /*顺序表的长度*/
typedef int KeyType;            /*关键字类型为整数类型*/
typedef struct{
     KeyType key;               /*关键字项*/
InfoType otherinfo;             /*其他数据项*/
      }RecType;                 /*记录类型*/
typedef struct {
     RecType  r[MAXSIZE+1];     /*r[0]空作为哨兵*/
     int length;                /*顺序表长度*/
}SqList;                        /*顺序表类型*/
```

1)　顺序查找

顺序查找的基本思想是：从表中第 n 个(第 1 个)记录开始，逐个进行记录的关键字与给定值的比较，若某个记录的关键字与给定值比较相等，则查找成功，找到所查记录；反之，若直到第 1 个(第 n 个)记录，其关键字与给定值比较都不等，则表明该表中没有所查的记录，查找不成功。

顺序查找的算法如下：

```
int SeqSearch(SqList R, KeyType k)      /*在 R.r[1..R.length]中查找关键字*/
                                        /*为 k 的记录*/
```

```
{                                          /*从后往前查找*/
    int i;
    R.r[0].key=k;                          /*查找结束*/
    for(i=R.length; i>=0&&R.r[i].key!=k; i--);
    if(i<=0)
        return -1;                         /*查找不成功*/
    else
        return i;
}
```

可见，在查找中与关键字的比较次数 c_i 取决于所查记录在表中的位置，如查找表中第 n 个记录，仅需要比较 1 次；而查找表中第 1 个记录时，需要比较 n 次，即 $c_i=n-i+1$。因此，在等概率的情况下，成功查找时的顺序查找的平均查找长度为：

$$ASL_{sq}=\sum_{i=n}^{1}p_ic_i=\frac{1}{n}\sum_{n}^{1}(n-i+1)=\frac{n+1}{2}$$

即查找成功时的平均比较次数约为表长的一半。若 k 值不在表中，则需要进行 n+1 次比较才能确定查找失败。

2) 折半查找

若顺序表中元素已按关键字有序排列，检索时不必顺序检索，而是用待查关键字与中间位置的元素的关键字进行比较，若相等，则检索成功；若待查关键字较小，则在由中间位置分开的左子表中查找，否则，在右子表中进行查找。如此下去，直至检索成功或检索失败。

折半查找的算法如下：

```
int BinSearch(SqList R, KeyType k)
                        /*在有序表 R.r[1...R.length]中检索关键字为 k 的记录*/
                        /*若检索成功，返回该记录的位置；否则返回-1*/
{
    int l=1; h=R.length;
    while (l<=h)
    {
        i=(l+h)/2;      /*取中间位置*/
            if (R.r[i].key==k)
                return(i);                  /*算法结束*/
            else if (R.r[i].key>k)
                h=i-1;                      /*在左子表中查找*/
            else l=i+1;                     /*在右子表中查找*/
    }
    return(-1);
}
```

折半查找的过程可用二叉树来描述，中间结点是二叉树的根，左子表相当于左子树，右子表相当于右子树，由此得到的二叉树便为描述折半查找的判定树。折半查找的过程是走了一条从根结点到叶子结点的过程，不论检索成功与失败，查找长度均不超过树的高度，其平均性能为 $h=[\log_2(n+1)]$。

折半查找速度快，但表必须有序，且频繁插入和删除不方便。它适合表中元素很少变化而检索频繁的情况；顺序表检索适用于检索少而表中元素频繁变化的情况。

3) 分块查找

分块查找综合了顺序查找和折半查找的优点，既有动态结构，又适于快速查找。分块

查找的基本思想是：将待查找文件等长地分为若干个子文件(最后一个子文件长度可能会小)。子文件内的元素无序，但子文件之间有序，即第一个子文件的最高关键字小于第二个子文件的所有记录的关键字，第二个子文件的最高关键字小于第三个子文件的所有记录的关键字，依次类推。再建立一个索引表(文件)，文件中的每个元素含有各个子文件的最高关键字和各个子文件中第一个元素的地址(下标)，索引文件按关键字有序。

分块查找过程分两步：第一步是在索引表中确定待查记录所在的块，可以顺序查找或折半查找；第二步是在块内顺序查找。

设待检索文件有 n 个记录，平均分成 b 块，每块有 s 个记录。若只考虑检索成功的概率，且在块内和索引表中均用顺序检索，则平均查找长度为：

$$ASL_{bs}=L_b+L_w=\frac{n/s+1}{2}+\frac{s+1}{2}=\frac{n+s}{2s}+\frac{s^2+s}{2s}=\frac{s^2+2s+n}{2s}=(s^2+2s+n)/(2s)$$

其中，L_b 为确定块的平均查找长度；L_w 为块内查找次数。

若 $s=\sqrt{n}$，则平均查找长度取最小值：$\sqrt{n}+1$。

若对索引表采用二分法检索，则平均查找长度为：

$$E(n)=E_b+E_w=\log_2(b+1)+\frac{s+1}{2}$$

3. 动态查找

动态查找也称树表查找，可执行动态查找的数据结构有二叉排序树、B−树和 B+树等。动态查找表的特点是：表结构本身是在查找过程中动态生成的，即对于给定值 key，若表中存在其关键字等于 key 的记录，则在查找成功后返回，否则插入关键字等于 key 的记录。考生重点掌握二叉排序树，因此这里主要介绍二叉排序树。

二叉排序树(简称 BST)或者是一棵空树，或者是具有下列性质的二叉树。

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值。
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值。
- 它的左、右子树也分别为二叉排序树。

从 BST 的性质可推出二叉排序树的另一个重要性质：按中序遍历该树所得到的中序序列是一个递增有序序列。

二叉排序树的查找方法：类似于折半查找，当二叉排序树不空时，首先将给定值 k 与根结点的关键字进行比较，若相等则查找成功；否则依 k 的大小在左子树或右子树上查找。可见，二叉排序树的查找是一个递归过程。

二叉排序树的构造：二叉排序树由依次输入的数据元素的序列构造而成。每读入一个元素，建立一个新的结点，并按下列原则插入结点。

- 若二叉排序树为空树，则新结点为二叉排序树的根结点。
- 若二叉排序树非空，则新结点的值与根结点比较，若小于根结点，则插入到左子树；否则插入到右子树。

二叉排序树的结点删除主要有以下 3 种情况。

- 若删除的结点为*p，其双亲为*f，如果*p 没有左右孩子，则可直接删除*p，修改*f 的指针即可。
- 若*p 结点只有左子树或只有右子树，此时只要令其左子树或右子树直接成为其双

亲结点*p 的左子树即可。

- 若*p 结点的左子树和右子树均不空，如果希望中序遍历该二叉树得到的序列的相对位置在删除结点前后不变，则可采用如下方法：其一，令*p 的左子树为*f 的左子树，而*p 的右子树为*s 的右子树。其中*s 为*p 左子树中最右边的一个结点。其二，令*p 的直接前驱(或直接后继)替代*p，然后再从二叉排序树中删除它的直接前驱(或直接后继)。这两种删除方法都能使原二叉排序树的中序遍历结果中结点的先后次序保持不变。

就平均时间性能而言，二叉排序树上的查找和折半查找相似。但就维护表的有序性而言，前者更有效，因为无须移动记录，只需修改指针即可完成对二叉排序树的插入和删除操作，而且其平均的执行时间均为 $O(\log_2 n)$。

有关平衡二叉树、B−树和 B+树的知识，考生可参考相关参考文献。

4. 散列查找(或哈希查找)

在记录的存储位置与它的关键字之间建立一个确定的对应关系 f，通过这个对应关系 f 找给定值 k 的像 f(k)，进行查找的方法为散列方法。称这个关系 f 为哈希函数，按此建立的表为哈希表。

设哈希表是一个地址为 0～(n−1)的向量，冲突是指由关键字得到的哈希地址为 j∈[0, n−1]的位置上已存有记录，而冲突处理就是为该关键字的记录找到另一个空的哈希地址。

1) 哈希函数的构造方法

哈希函数的构造方法有如下几种。

- 直接定址法：取关键字或关键字的某个线性函数值为哈希地址，即：H(key)=key 或 H(key)=a*key+b；其中 a 和 b 为常数。
- 数字分析法：假设关键字是以 r 为基的数，并且哈希表中可能出现的关键字都是事先知道的，则可取关键字的若干数位组成哈希地址。
- 平方取中法：取关键字平方后的中间几位为哈希地址。
- 折叠法：将关键字分割成位数相同的几部分，然后取这几部分的叠加和。
- 除余数法：取关键字被某个不大于哈希表表长 m 的数 p 除后所得余数为哈希地址，即：H(key)=key mod p，p≤m(注：p 的选择很重要)。
- 随机数法：选择一个随机函数，取关键字的随机函数值为它的哈希地址。即：H(key)=random(key)。

2) 处理冲突的方法

处理冲突的方法有如下几种。

- 开放地址法：$H_i=(H(key)+di) \bmod m$，i=1, 2, 3, …, n，其中，H(key)为哈希函数，m 为哈希表表长，di 为增量序列。若 di=1, 2, …, n，则称线性探测再散列；di=1^2, -1^2, 2^2, -2^2, …, n^2, $-n^2$，则称二次探测再散列；di=伪随机数序列，则称伪随机探测再散列。
- 再哈希法：$H_i=RH_i(key)$, i=1, 2, …, n，其中，RH_i 均是不同的哈希函数，即在同义词产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生。这种方法不易产生聚集，但增加了计算时间。

- 链地址法：将所有关键字为同义词的记录存储在同一线性链表中。
- 建立一个公共溢出区：关键字和基本表中关键字为同义词的记录，一旦发生冲突，就将其填入溢出表。

3)　哈希表查找的性能分析

给定 k 值，根据造表时设定的哈希函数求得哈希地址，若表中此位置没有记录，则查找不成功；否则比较关键字，若和给定值相等，则查找成功；若不等，则根据造表时设定的处理冲突的方法查找下一个地址，直至某个位置上表为空或关键字比较相等为止。可见，比较关键字的个数取决于 3 个因素：哈希函数、处理冲突的方法和哈希表的装填因子。

哈希表的平均查找长度不是对象个数 n 的函数，而是装填因子α(α=n/m)的函数。线性探测法成功查找的平均查找长度为(1+1/(1−α))/2，而不成功查找的平均查找长度为(1+1/(1−α)^2)/2；二次探测再散列法的成功查找的平均查找长度为$-\log e^{(1-\alpha)/\alpha}$，而不成功查找的平均查找长度为 1/(1−$\alpha$)；链地址法成功查找的平均查找长度为 1+$\alpha$/2，而不成功查找的平均查找长度为$\alpha+e^{-\alpha}\approx\alpha$。

1.2.2　典型例题分析

例 1　阅读以下说明和 C 函数，将应填入 (n) 处的字句写在答题纸的对应栏内。(2009 年下半年试题二)

【说明 1】

函数 Counter (int n , int w[ ])的功能是计算整数 n 的二进制表示形式中 1 的个数，同时用数组 w 记录该二进制数中 1 所在位置的权。

例如，十进制数 22 的二进制表示为 10110。对于该二进制数，1 的个数为 3，在 w[0]中存入 2 (即 2^1)，w[1]中存入 4 (即 2^2)，w[2]中存入 16 (即 2^4)。

【C 函数 1】

```
int Counter ( int n , int w[ ] )
{
    int i = 0 , k = l ;
    while ( (1) )
    {
        if ( n % 2 )
            w [ i + + ] = k ;
        n = n / 2 ;
        (2) ;
    }
    return  i;
}
```

【说明 2】

函数 Smove (int A[] , int n)的功能是将数组中所有的奇数都放到所有偶数之前。其过程为：设置数组元素下标索引 i(初值为 0)和 j(初值为 n−1)，从数组的两端开始检查元素的奇偶性。若 A[i]、A[j]都是奇数，则从前往后找出一个偶数，再与 A[j]进行交换；若 A[i]、A[j]都是偶数，则从后往前找出一个奇数，再与 A[i]进行交换；若 A[i]是偶数而 A[j]是奇数，则交换两者，直到将所有的奇数都排在所有偶数之前为止。

【C 函数 2】

```
void Smove (int A[] , int n)
{
    int temp , i= 0 , j = n-l ;
    if ( n < 2 )
        return;
    while ( i< j )
    {
        if ( A [i]%2 == l & & A[j] % 2 ==l )
        {
            (3) ;
        }
        else if ( A[i] % 2 == 0 & & A[j]% 2==0 )
        {
            (4) ;
        }
        else {
            if ( (5) )
            {
                temp = A[i] ; A[i] = A[j] ; A[j]= temp ;
            }
            i++,j++ ;
        }
    }
}
```

分析：由说明 1 中可知函数 Counter (int n , int w[ ])的功能是计算整数 n 的二进制表示形式中 1 的个数，同时用数组 w 记录该二进制数中 1 所在位置的权。所以(1)处循环判断条件应为 n!=0，即当 n 不为 0 时进行操作，每执行一次 k 的值就增加一倍，所以(2)处应为 k=k*2 或 k*=2 。由函数 Smove (int A[] , int n)的功能可知，若 A [i]、A[j]都是奇数(A [i]%2 == l & & A[j] % 2 ==l)，则从前往后找出一个偶数，再与 A[j]进行交换，所以此时应将 A [i]继续向后查找(即 i++)，而 A[j]保持不变，留待 A [i]为偶数时与之交换，同理，若 A [i]、A[j]都是偶数，则 A[j]向前查找(j--)，A[i]保持不变，留待 A[j]为奇数时与之交换，若 A [i]是偶数而 A[j]是奇数，则交换两者，所以(3)处应为 i++，(4)处应为 j--，(5)处应为 a[i]%2==0&& a[j]% 2==1。

答案：

(1) n 或 n!=0

(2) k=k*2、k*=2 或其他等价表达式

(3) i++或其他等价表达式

(4) j--或其他等价表达式

(5) a[i]%2==0&&a[j]%2==1 或其他等价表达式

例 2 阅读以下说明和 C 函数，将解答填入答题纸的对应栏内。(2009 年下半年试题四)

【说明】

函数 del_substr (S , T)的功能是从头至尾扫描字符串 S，删除其中与字符串 T 相同的所有子串，其处理过程为：首先从串 S 的第一个字符开始查找子串 T，若找到，则将后面的

字符向前移动，将子串 T 覆盖掉，然后继续查找子串 T，否则从串 S 的第二个字符开始查找，依次类推，重复该过程，直到串 S 的结尾为止。该函数中字符串的存储类型 SString 定义如下：

```
typedef struct {
char * ch ;   /*串空间的首地址*/
int length ;  /*串长*/
} SString ;
```

【C 函数】

```
void del _ substr ( SString * S , SString T )
{
    inti , j ;
    if ( S->length < l || T . length < l || S->length < T . length )
        return;
    i=0 ;                                   /*i 为串 S 中字符的下标*/
    for ( ; ; )
    {
        j = 0 ;                             /*j 为串 T 中字符的下标*/
        While ( i < S->length & & j < T . length )
        {                                   /*在串 S 中查找与 T 相同的子串*/
            if ( S->ch[i]== T . ch[j] )
            {
                i++; j++;
            }else
            {
                i= (1) ;  j = 0 ;           /*i 值回退，为继续查找 T 做准备*/
            }
        }
        If( (2) )
        {                                   /*在 S 中找到与 T 相同的子串*/
            i= (3) ;                        /*计算 S 中子串 T 的起始下标*/
            for ( k = i + T. length ; k < s->length ; k + + )
                                            /*通过覆盖子串 T 进行删除*/
            S-> ch[ (4) ] =S-> ch[k] ;
            S->length = ( 5 ) ;             /*更新 S 的长度*/
        }
        else break ;                        /*串 S 中不存在子串 T*/
    }
}
```

分析：进行查找时，如果 T 中前 j 个字符和 S 中从下标 i 开始的 j 个字符相同，则 i 都会增加到 i+j，如果 T 中第 j+1 个和 S 中下一个字符不同，则查找结束，此时需要将 i 的值(此时已是 i+j)回退到此遍历进行前的值(即 i)的下一个字符位置即 i−j+1，并开始新一轮的遍历查找，所以空(1)处应为 i−j+1。一轮查找结束后，如果 j 的值等于 T 的长度，表明 S 中存在子字符串和 T 相同，因为只要其中有一个字母不同，j 就会清零一次，最后的值就一定会小于 T 的长度，所以空(2)处应为 j==T.length。找到时，同时 i 的值也已经增加了 T 的长度，所以要计算 S 中子串 T 的起始下标只需要 i 减去 T 的长度即可，所以空(3)处应为 i−T.length 或者 i−j。通过覆盖子串 T 进行删除时，从 i + T. length 到 s−>length，只需要将每个字符替换到其前 j 处，即空(4)处应为 k−j。替换后，将 S 的长度减去 T 的长度即可更新 S 的长度。所以空(5)处应为 s−>length-T.length。

答案：

(1) i−j+1

(2) j==T.length

(3) i−T.length 或 i−j

(4) k−j

(5) s−>length-T.length

例 3　阅读以下说明、C 函数和问题，将解答填入答题纸的对应栏内。(2009 年上半年试题三)

【说明】

二叉查找树又称为二叉排序树，它或者是一棵空树，或者是具有如下性质的二叉树：

- 若它的左子树非空，则其左子树上所有结点的键值均小于根结点的键值；
- 若它的右子树非空，则其右子树上所有结点的键值均大于根结点的键值；
- 左、右子树本身就是二叉查找树。

设二叉查找树采用二叉链表存储结构，链表结点类型定义如下：

```
typedef struct BiTnode {
    int key_value ;                    /*结点的键值为非负整数*/
    struct BiTnode * left , * right ;  /*结点的左、右子树指针*/
} * BSTree ;
```

函数 find_key (root , key)的功能是用递归方式在给定的二叉查找树(root 指向根结点)中查找键值为 key 的结点并返回结点的指针；若找不到，则返回空指针。

【C 函数】

```
BSTree find _ key ( BSTree root , int key)
{
    If( (1) )
        return NULL ;
    else
        if(key==root->key_value)
            return (2) ;
        else if(key<root->key_value)
            return (3) ;
        else
            return (4) ;
}
```

【问题 1】请将函数 find_key 中应填入(1)到(4)处的字句写在答题纸的对应栏内。

【问题 2】若某二叉查找树中有 n 个结点，则查找一个给定关键字时，需要比较的结点个数取决于 (5) 。

分析：依题意，当遍历到叶子结点的左右子树时递归结束，返回 NULL，所以空(1)处应为 root==NULL。在进行查找的过程中，如果 key 等于 root 的键值，则 root 就是所要查找的结点，所以空(2)处应为 root。依题意若它的左子树非空，则其左子树上所有结点的键值均小于根结点的键值；若它的右子树非空，则其右子树上所有结点的键值均大于根结点的键值，如果 key 小于 root 的键值，应查询其左子树，大于 root 的键值则查询其右子树，所以空(3)和空(4)处分别为 find_key(root−>left, key)和 find_key(root−>right, key)。若某二叉查

找树总共有 n 个结点，则查找一个给定关键字时，需要比较的结点个数取决于关键字所在结点的层数和二叉树的高度。因为若结点存在，层次越高，查找次数就越多；若结点不存在，则要遍历整个树。

答案：

(1) root==NULL

(2) root

(3) find_key(root->left, key)

(4) find_key(root->right, key)

(5) 关键字所在结点的层数和二叉树的高度

1.2.3　同步练习

阅读以下说明和流程图(见图 1-2)，填补流程图中的空缺(1)～(5)，将解答填入答题纸的对应栏内。

【说明】

下面的流程图可在正文字符串 T(1 : L)中计算关键词字符串 K(1 : m)出现的次数(用 n 表示)。其中，L 为字符串 T 的长度，m 为字符串 K 的长度(m<L)。为便于模糊查找，关键词中的字符？可以匹配任意一个字符。在该流程图中，先从 T 中取出长度为 m 的子串存入 A 中，再将 A 与 K 进行逐个字符的比较(其中，K 可以包含字符？)。注意：从正文字符串中取出的关键词字符串不允许交叉。

例如，aaaaaa 中有 3 个关键词字符串 aa。

【流程图】

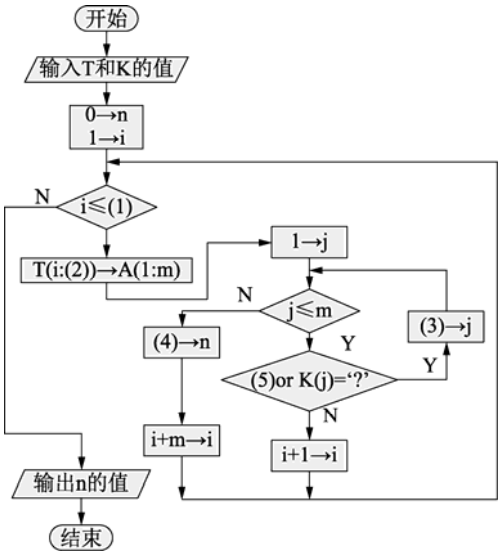


图 1-2　流程图

1.2.4　同步练习答案

分析：1.2.3 节中练习题考查字符串的基本操作。

根据题意，正文字符串中的各个字符依次存放在 T(1)、T(2)、…、T(L)中，关键词字符串中的各个字符依次存放在 K(1)、K(2)、…、K(m)中。从图 1-2 所示流程图可以看出，i 是字符数组 T 的动态下标。为了与关键词字符串进行比较，需要每次从数组 T 中连续取出 m 个元素放在数组 A 中。第一次将 T(1:m)存入 A(1:m)，第 2 次将 T(2:m+1)存入 A(1:m)，依此计算，第 i 次将 T(i:m+i−1)存入 A(1:m)，最后一次应将 T(L:m+L−1)存入 A(1:m)。因此，在该流程图中，空(2)处应填入：m+i−1。另外，由于正文字符串长度不会大于 L，因此，m+i−1<=L，即 i<=L−m+1。所以在该流程图中，空(1)处应填入：L−m+1。

该流程图右下方的部分描述了数组 A 与 K 逐个字符的比较过程。该流程图用 j 表示数组 K 的动态下标，j=1,2,…,m。显然，数组 A 的下标为 i+j−1(j=1,2,…,m)。两个字符数组都从左到右逐个字符地进行比较。如果发现有不一致的字符，则结束比较，将 i 增 1 后继续从数组 T 中取新的子串放置在 A 中。如果一直到比较结束，两个数组中对应的各个字符都是一致的，即找到了一处关键词。关键词的个数用变量 n 表示，找到一个关键词 n 的值应加 1，因此，空(4)处应填入：n+1。

数组 A 与 K 的比较过程是逐个字符 A(j)与 K(j)的比较。由于允许模糊查找，K(j)中的字符?可与任何字符相匹配，因此，比较判断条件可以写成：A(j)=K(j) or K(j)="?"。由此，空(5)处应填入：A(j)=K(j)。如果比较结果为真，则还需要对 j 增 1，准备继续往下比较。因此，空(3)处应填入：j+1。

答案：

(1) L−m+1　　(2) m+i−1　　(3) j+1

(4) n+1　　(5) A(j)=K(j)

1.3 数 据 结 构

1.3.1 考点辅导

1.3.1.1 线性表

1. 线性表的顺序存储结构

1) 顺序表

线性表的顺序存储结构采用一组连续的存储单元依次存储线性表中的各数据元素。建立一个数组 V，线性表的长度为 N，V[i]表示第 i 个分量，第 i 个分量是线性表中第 i 个元素 a_i 在计算机存储器中的映像，即 V[i]= a_i。若线性表的第一个元素的存储地址是 LOC(a_1)，每个元素用 L 个存储单元，则表的第 i 个元素的存储地址为：LOC(a_i)=LOC(a_1)+(i−1)*L。

假设线性表的数据元素的类型为 ElemType(在实际应用中，此类型应根据实际问题中出现的数据元素的特性具体定义，如为 int、float 类型等)，则线性表的顺序表的 C 语言描述如下：

```
#define MAXSIZE                  /*顺序表的长度为对 MAXSIZE 定义的值*/
typedef  struct {
   ElemType  data[MAXSIZE];
   int  len;                     /*线性表数据元素的个数*/
}Sqlist;
```

从中可以看出，顺序表是由数组 data 和 len 两部分组成的。为了反映 data 和 len 之间的关系，上述类型定义中将它们说明为结构体类型 Sqlist 的两个域。这样，Sqlist 类型就完全描述了顺序表的组织。

2)　基本运算在顺序表上的实现

由于 C 语言中数组的下标是从 0 开始的，所以，在逻辑上所指的“第 k 个位置”实际上对应的是顺序表的“第 k−1 个位置”。这里仅给出在顺序表上线性表的插入和删除函数。

(1)　插入函数。代码如下：

```
insert(v, n, i, x)          /*该算法在长度为 n 的线性表 L 的第 i 个位置插入元素 x*/
int n, i;
float x, v[];
{
    if ((i<0)||(i>n+1))     /*插入位置非法*/
        printf("error");
    else
        for(j=n;j>=i;j--)
            v[j+1]=v[j];
        v[i]=x; n++;
}
```

(2)　删除函数。代码如下：

```
delete (L，n，i)            /*该算法删除长度为 n 的线性表 L 的第 i 个位置的元素 x*/
int n, i;
float L[];
{
    if ((i>=n)||(i<0))
        printf("error");
    else
    {
        for (j=i;j<n-1;j++)
            v[j]=v[j+1];
        n--;
    }
}
```

3)　插入和删除元素算法的时间复杂度分析

(1)　插入算法的时间复杂度。

$$\sum_{i=1}^{n+1}(pi*ci)=1/(n+1)*\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2}$$

其中，pi 是在第 i 个元素前插入元素的概率，ci 是在第 i 个元素前插入元素时元素移动的次数。

(2)　删除算法的时间复杂度。

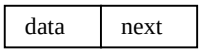
$$\sum_{i=1}^{n}(pi*ci)=1/n*\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$$

其中，pi 是在第 i 个元素前删除元素的概率，ci 是在第 i 个元素前删除元素时元素移动的次数。

可见，插入和删除算法的时间复杂度均为 O(n)。

2．线性表的单链表存储结构

单链表中的每个结点由两部分组成：数据域和指针域。结点形式如下。



其中，data 部分称为数据域，用于存储线性表的一个数据元素(结点)。next 部分称为指针域或链域，用于存放一个指针，该指针指向本结点所含数据域元素的直接后继所在的结点。若数据元素的类型用 ElemType 表示，则单链表的类型定义如下：

```
typedef  struct  node{
    ElemType data;
    struct  node  *next;
} Slink;
```

单链表分为带头结点(其 next 域指向第一个结点)和不带头结点两种类型，由于头指针的设置使得对链表的第一个位置上的操作与在表其他位置上的操作一致，因而可简化运算的实现过程。

在单链表上实现线性表基本运算的函数如下。

1) 初始化函数

初始化函数用于创建一个头结点，由 head 指向它，该结点的 next 域为空，data 域未设定任何值。由于调用该函数时，指针 head 在本函数中指向的内容发生改变，为了返回改变的值，因此使用了应用型参数，其时间复杂度为 O(1)。初始化函数的语句如下：

```
void Initlist(Slink *head)
{
    head=(Slink )malloc(sizeof(Slink)); /*创建头结点*/
    head->next=NULL;
}
```

2) 插入函数 insert(Slink *head, int i , ElemType x)

插入函数的设计思想是：创建一个 data 域值为 x 的新结点*p，然后插入到 head 所指向的单链表的第 i 个结点之前。为保证插入正确有效，必须查找到指向第 i 个结点的前一个结点的指针，主要的时间耗费在查找上，因而在长度为 n 的线性单链表中进行插入操作的时间复杂度为 O(n)。插入函数的语句如下：

```
insert(Slink *head, int i, ElemType x)
{
    Slink  *p,  *pre, *q;
    int j=0;
    p=( Slink *) malloc(sizeof(Slink ));
    p->data=x;                    /*生成 p 结点，x 是元素的值*/
    pre=head;                     /*pre 指向待插入结点的前驱结点*/
    q=head->next;                 /*q 指向当前比较结点*/
    while (q&&j<i-1)              /*查找 p 结点应插入的位置*/
    {
        pre=q;
        q=q->next;
        j++;
    }
    if(j!=i-1||i<1)return 0;      /*插入不成功*/
```

```
    else{
        p->next=q;                  /*将 p 结点插入链表*/
        pre->next=p;
    }
    return 1;                       /*插入成功*/
}
```

3)　删除函数 delete(Slink *head, int i , ElemType x)

删除函数的设计思想是：线性链表中元素的删除要修改被删元素前驱的指针，回收被删元素所占的空间。主要的时间耗费在查找上，因而在长度为 n 的线性单链表中进行删除操作的时间复杂度为 O(n)。删除函数的语句如下：

```
delete(Slink *head,int i,ElemType x)   /*删除第 i 个结点，并通过 x 返回值*/
{
    Slink *p, *q;
    int j=0;
    p=head;
    while(p->next && j<i-1)            /*查找第 i 个结点的前驱位置 p*/
    {
        p=p->next; j++;
    }
    if(!(p->next)|| j>i-1) return 0;   /*删除位置不合适*/
    q=p->next;                         /*删除并释放结点*/
    p->next=q->next;
    x=q->data;
    free(q);
    return 1;
}
```

4)　查找函数 get(Slink *head, int i)

查找函数的设计思想是：在线性链表中查找元素要找元素前驱的指针。在长度为 n 的线性单链表中进行查找操作的时间复杂度为 O(n)。查找函数的语句如下：

```
Slink * get(Slink *head, int i)        /*查找第 i 个结点*/
{
    Slink *p;
    int j=0;
    p=head;
    while(p->next&&j<i-1)          /*查找第 i 个结点的前驱位置 p*/
    {
        p=p->next;
        j++;
    }
    if(!(p->next)|| j>i-1) return NULL;
                                   /*查找位置不合适*/
    return p->next;
}
```

5)　求单链表表长函数 Length(Slink *head)

求单链表表长函数的设计思想是：通过遍历的方法，从头数到尾，即可得到单链表的表长。求单链表表长函数的语句如下：

```
int Length (Slink *head )
```

```
{
    int len=0 ;
    Slink  *p; p=head;              /*设该表有头结点*/
    while(p->next)
    {
        p=p->next;
        len++;
    }
    return len;
}
```

3. 带头结点的单链表和不带头结点的单链表的区别

带头结点的单链表和不带头结点的单链表的区别主要体现在其结构和算法操作上。

在结构上，带头结点的单链表不管链表是否为空，均含有一个头结点；而不带头结点的单链表不含头结点。

在操作上，带头结点的单链表的初始化为申请一个头结点，且在任何结点位置进行的操作算法一致；而不带头结点的单链表让头指针为空，同时其他操作要特别注意空表和第一个结点的处理。下面列举带头结点的单链表插入操作和不带头结点的单链表插入操作的区别。

定义单链表的结点类型如下：

```
typedef  struct  node {
         ElemType data;       /*结点的数据域*/
         struct  node  *next;  /*结点的指针域或链域*/
    } Slink;
```

1) 带头结点的单链表插入函数 insert(Slink *head, int i , ElemType x)

带头结点的单链表插入函数的设计思想是：创建一个 data 域值为 x 的新结点*p，然后插入到 head 所指向的单链表的第 i 个结点之前。为保证插入正确有效，必须查找到指向第 i 个结点的前一个结点的指针，主要的时间耗费在查找上，因而在长度为 n 的线性单链表中进行插入操作的时间复杂度为 O(n)。

2) 不带头结点的单链表插入函数 insert(int i , ElemType x)

不带头结点的单链表插入函数的设计思想是：创建一个 data 域值为 x 的新结点*p，然后插入到单链表的第 i 个结点之前。由于链表不带头结点，所以当 i=1 时插入操作的算法实现，与 i>1 时是有差别的，必须单独处理。为保证插入正确有效，必须查找到指向第 i 个结点的前一个结点的指针，主要的时间耗费在查找上，因而在长度为 n 的线性单链表中进行插入操作的时间复杂度为 O(n)。

可见，带头结点的单链表插入操作和不带头结点的单链表插入操作在算法实现上有很大区别，主要体现在初始化、能否插入成功的判别及插入时的操作上，在带头结点的单链表上插入在任何位置上都是相同的，而在不带头结点单链表的第一个结点和其他结点前插入操作是不同的。

对于带头结点的单链表和不带头结点的单链表在其他操作上的区别可类似得到。

4. 链表的指针修改次序对结果的影响

链表的指针修改必须保持其逻辑结构的次序，否则将违背线性表的特征，尤其是进行插入和删除操作。下面通过双向链表的插入操作来说明，若在如图 1-3 所示的 P 所指向的结

点之前插入一个 S 所指向的结点，则需要进行指针的修改，修改指针的策略有两种，如图 1-4 和图 1-5 所示，指针的修改次序为 1，2，3，4。根据线性表的性质可知，图 1-4 可保证指针修改成功；而图 1-5 中指针修改不成功，主要原因是其首先将 P 的前驱指向 S，这样 P 结点的原前驱结点就不能找到了，因而指针修改步骤 3 和 4 不成立。

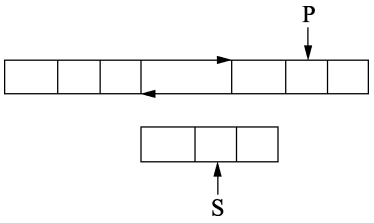


图 1-3　双向链表的结点插入前

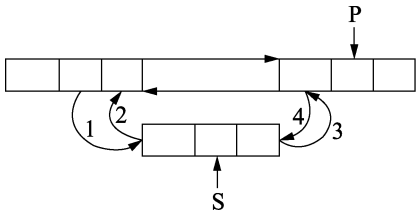


图 1-4　指针的修改策略 1

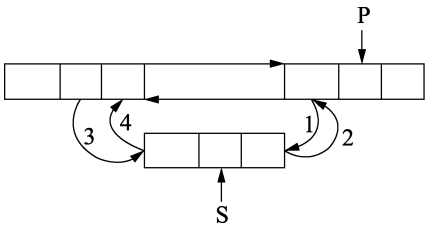


图 1-5　指针的修改策略 2

可见，指针的修改次序是链表插入成功与否的关键因素之一；同理，在进行结点的删除时也同样需要主要指针的次序。

5. 顺序存储结构上的算法如何移植到链式存储结构上

很多优秀的算法都是建立在顺序存储结构上的，如何在链式存储结构上实现这些优秀算法是考生应注意的问题。近年来，在不少程序员水平考试试题中都出现了这样的题目。这里，通过在顺序存储结构和链式存储结构两种存储结构上实现选择排序来进行说明。依据顺序存储结构下的算法 sort1 可以拓展到链式存储结构下的算法 sort2。

算法 sort1 如下：

```
sort1(int R[], int n)                    /*对 n 个元素的数组 R 进行由小到大排序*/
{
    int i, j, k, temp;
    for(i=0; i<n-1; i++)
    {
        k=i;
        for(j=i+1; j<n; j++)             /*在 R[i...n-1]中找最小的元素*/
            if(R[j]<R[k])k=j;
            if(k!=i)
            {
                temp=R[i];
                R[i]=R[k];
                R[k]=temp;
            }
    }
}                                        /*sort1*/
```

下面将算法 sort1 拓展到链式存储结构下的算法 sort2：

```
typedef  struct  node {
         int data;                  /*结点的数据域*/
         struct  node  *next;       /*结点的指针域或链域*/
    } Slink;
sort2(Slink *head)                  /*将带头结点的单链表 head 进行由小到大排序*/
{
    Slink *p=head->next , *q, *r;
    int temp;
    while(p)
    {
        q=p;
        r=p->next;
        while(r)
        {
            if(r->data<q->data)q=r;
                                    /*在以 p 为首结点的单链表中找最小的元素*/
        r=r->next;
        }
        if(q!=p)
        {
            temp=p->data;
            p->data=q->data;
            q->data=temp;
        }
                                    /*找到的结点和 p 结点进行数据交换*/
        p=p->next;
    }
}                                   /*sort2*/
```

可见，只要充分领会顺序存储结构下的算法思想，熟悉链表存储结构，就可以通过掌握顺序存储结构下的算法得到链表存储结构下的相应算法。

1.3.1.2 栈

1. 栈的顺序存储结构

栈的顺序存储用向量作为栈的存储结构，向量 S 表示栈，m 表示栈的大小，用一栈指针 top 指向栈顶位置，S[top]表示栈顶元素，当在栈中进行插入、删除操作时，都要移动栈指针；而当 top=m−1 时，则栈满，当 top=−1 时，则栈空。同时为了避免浪费空间可以采用双栈机制，即向量的两端为栈底。

栈的顺序存储结构的 C 语言描述如下：

```
#define StackSize 100            /*栈的容量*/
typedef struct { ElemType data[StackSize];
                 int top;
                 } SqStack;
SqStack sq;
```

栈的说明如下。

- 由于 C 语言数组下标的范围是从 0 至 StackSize −1，初始化设置为 sq.top= −1。
- 栈空条件为 sq.top== −1，栈满条件为 sq.top== StackSize −1。

- 栈顶元素为 sq.data[sq.top]。
- 元素压栈的规则为：在栈不满时，先改变栈顶指针(top=top+1)，再压栈。出栈时，在栈非空时，先取栈顶元素的值，再修改栈顶指针(top=top −1)。
- 栈中元素的个数为当前栈顶指针加 1。

在顺序栈上实现基本操作的有关函数如下。

1)　初始化 InitStack(SqStack *S)

```
void InitStack(SqStack *S)
{
    S->top=-1;
}
```

2)　判空 StackEmpty(SqStack S)

```
int StackEmpty(SqStack S)
{
  if(S.top==-1) return 1;
  else return 0;
 }
```

3)　压栈 Push(SqStack *S, ElemType e)

```
int Push(SqStack *S, ElemType e)
{
    if(S->top< StackSize-1)
    {
        S->top=S->top+1;
        S->data[S->top]=e;
        return 1;
    }
    else {printf("栈满! \n");
    return 0;
    }
}
```

4)　出栈 Pop(SqStack *S, ElemType *e)

```
int Pop(SqStack *S, ElemType *e)
{
    if(sq->top==-1) return 0;
    else
    {
        *e=sq->data[sq->top];
        sq->top--;
        return 1;
    }
}
```

5)　取栈顶 GetTop(SqStack *S, ElemType *e)

```
int GetTop(SqStack *S, ElemType *e)
{
    if(sq->top==-1) return 0;
    else
    {
        *e=sq->data[sq->top];
```

```
        return 1;
    }
}
```

6) 清栈 ClearStack(SqStack *S)

```
int ClearStack(SqStack *S)
{
S->top=-1;
}
```

2. 栈的链式存储结构

栈的链式存储也叫链栈，我们把插入和删除均在链表表头进行的链表称为链栈。链栈也分带头结点和不带头结点两种。带头结点的链栈操作比较方便。

链栈的结点类型定义如下：

```
typedef struct stnode{
        ElemType data;
        struct stnode *next;
        }LStack;
LStack *S;
```

链栈的约定与说明如下。

- 栈以链表的形式出现，链表(不带头结点)首指针为 S，即栈顶为 S，链表尾结点为栈底。
- 初始化时，S=NULL(不带头结点)；S=(LStack *)，malloc(sizeof(LStack))，S->next=NULL(带头结点)。
- 栈顶指针的引用为 S(不带头结点)或 S->next(带头结点)，栈顶元素的引用为 S->data(不带头结点)或 S->next->data(带头结点)。
- 栈空条件为 S==NULL(不带头结点)或 S->next==NULL(带头结点)。
- 进栈操作和出栈操作与单链表在开始结点的插入和删除操作一致。

对不带头结点的链栈，其基本操作函数如下。

1) 初始化 initstack(LStack *S)

```
void initstack(LStack *S)
{
 S=NULL;
}
```

2) 压栈(入栈)push(LStack *S, ElemType x)

```
int push(LStack *S, ElemType x)
                                        /*将元素 x 压到链栈 S 中*/
{
    p=(LStack *) malloc(sizeof(LStack ));
    p->data=x;
    p->next=NULL;
    if(S==NULL)S=p;
    else
    {
        p->next=S;
        S->next=p;
```

```
    }
    return 1;
}
```

3)　退栈(出栈) pop(LStack *S, ElemType *x)

```
int pop(LStack *S, ElemType *x)
{
    if(S==null)                              /*链栈为空*/
    {
        printf("\n underflow");
        return 0;
    }
    else
    {
        p=S;
        S=S->next;
        x=p->data;
        free(p);
        return 1;
    }
}
```

4)　读栈顶元素 gettop(LStack *S, ElemType *x)

```
int gettop(LStack *S, ElemType *x)
{
    if  (S==null)                            /*链栈为空*/
    {
        printf("\n underflow");
        return 0;
    }
    else
    {
        x=S->data;
        return 1;
    }
}
```

5)　判栈空 isempty(LStack *S)

```
int isempty(LStack *S)
{
    if  (S==null)
      return 0;
    else
      return 1;
}
```

3. 栈的应用

栈具有广泛的应用，如：求表达式的值及递归到非递归等。

1)　表达式求值

在源程序编译中，若要把一个含有表达式的赋值语句翻译成正确求值的机器语言，首先应正确地解释表达式。例如，赋值语句“X=4+8×2−3;”，其正确的计算结果应该是 17，但若在编译程序中简单地按自左向右扫描的原则进行计算，则为：X=12×2−3=24−3=21。这

个结果显然是错误的。因此，为了使编译程序能够正确地求值，必须事先规定求值的顺序和规则。通常采用运算符优先法。

2) 递归到非递归

将一个递归算法转换为功能等价的非递归算法有很多种方法，可以使用栈保存中间结果。其一般形式如下：

```
将初始状态 s0 进栈;
while(栈不为空){
退栈，将栈顶元素赋给 s;
     if(s 是要找的结果)返回;
    else{
         寻找到 s 的相关状态 s1;
         将 s1 入栈;
        }
}
```

例如，求 n!的递归函数如下：

```
int funrec(int n)
{
  if(n==0||n==1) return 1;
  else
  return n*funrec(n-1);
}
```

使用转换成等价的非递归算法如下，其中 st[top][0]用于存放 n 值，st[top][1]用于存放 n!值，在初始时，设置 st[top][1]为 0，表示 n！尚未求出。

```
#define Max 100
int funnonrec(int n)
{
    int st[Max][2], top=-1;             /*栈定义及初始化*/
    top++;
    st[top][0]=n;
    st[top][1]=0;
    do{                                 /*循环求解*/
        if(st[top][0]==1)               /*满足递归出口，给出 st[top][0]值*/
            st[top][1]=1;
        if(st[top][0]>1 && st[top][1]==0)
        {
                                        /*递推入栈*/
            top++;
            st[top][0]=st[top-1][0]-1;
            st[top][1]=0;
        }
        if(st[top][1]!=0)
        {                               /*求值出栈*/
            st[top-1]=st[top][1]*st[top-1][0];
            top--;
        }
    }while(top>0);
    return st[0][1];                    /*返回栈底值*/
}
```