

第 3 章 栈

习 题 3

习题 3.1 扩充抽象数据类型栈的定义，增加如下栈运算：

(1) $\text{StackSize}(S)$ ：确定栈 S 中的元素个数。

(2) $\text{StackIn}(S)$ ：入栈 S 。

(3) $\text{StackOut}(S)$ ：出栈 S 。

用数组实现扩充后的栈。

解答：

用数组实现扩充后的栈如下。

```
1  int StackSize(Stack S)
2  {
3      return S->top+1;
4  }
5
6  void StackIn(Stack& S)
7  {
8      int sz;
9      printf("输入栈中元素个数 \n");
10     scanf("%d", &sz);
11     if(sz>S->maxtop+1) return;
12     S->top=sz-1;
13     printf("自顶向下输入栈中元素 \n");
14     for(int i=S->top; i>=0; i--) S->data[i]=ReadItem();
15 }
16
17 void StackOut(Stack S)
18 {
19     for(int i=S->top; i>=0; i--) StackShow(S->data[i]);
20     printf("\n");
21 }
```

习题 3.2 扩充栈的定义，增加如下栈运算：

(1) $\text{StackSplit}(S1, S2)$ ：将栈 $S1$ 分为大小相同的两个栈 $S1$ 和 $S2$ 。栈 $S2$ 中含原栈 $S1$ 中上半部分元素，其余元素留在栈 $S1$ 中。

(2) $\text{StackCombine}(S1, S2)$ ：将栈 $S2$ 中的元素合并于栈 $S1$ 顶部，且保持原栈中元素的次序， $S2$ 成为空栈。

用数组实现上述扩充后的栈。

解答：

用数组实现增加了 StackSplit 和 StackCombine 运算的栈如下。

```
1 void StackSplit(Stack& S1, Stack& S2)
2 {
3     int n=S1->top;
4     S1->top=n/2;
5     S2->top=n-n/2-1;
6     for(int i=S2->top; i>=0; i--) S2->data[i]=S1->data[S1->top+i+1];
7 }
8
9 void StackCombine(Stack& S1, Stack& S2)
10 {
11     for(int i=0; i<=S2->top; i++) S1->data[S1->top+i+1]=S2->data[i];
12     S1->top=S1->top+S2->top+1;
13     S2->top=-1;
14 }
```

习题 3.3 用指针实现栈的方法重做习题 3.1。

解答：

用指针实现扩充后的栈如下。

```
1 int StackSize(Stack S)
2 {
3     int n=0;
4     slink p=S->top;
5     while(p) {
6         n++; p=p->next;
7     }
8     return n;
9 }
10
11 void StackIn(Stack& S)
12 {
13     StackFree(S);
14     int sz;
15     printf("输入栈中元素个数 \n");
16     scanf("%d", &sz);
17     if(!sz) return;
18     printf("自顶向下输入栈中元素 \n");
19     S->top=NewStackNode();
20     S->top->element=ReadItem();
21     slink last=S->top;
22     for(int i=2; i<=sz; i++) {
23         slink x=NewStackNode();
24         x->element=ReadItem();
25         last->next=x;
```

```

26     last=x;
27 }
28 last->next=0;
29 }
30
31 void StackOut(Stack S)
32 {
33     slink p=S->top;
34     while(p) {
35         StackShow(p->element);
36         p=p->next;
37     }
38     printf("\n");
39 }

```

习题 3.4 用指针实现栈的方法重做习题 3.2。

解答：

用指针实现增加了 StackSplit 和 StackCombine 运算的栈如下。

```

1 void StackSplit(Stack& S1, Stack& S2)
2 {
3     int n=StackSize(S1)/2;
4     if(n==0) return;
5     StackFree(S2);
6     slink p=S1->top;
7     S2->top=p;
8     for(int i=1; i<n; i++) p=p->next;
9     S1->top=p->next;
10    p->next=0;
11 }
12
13 void StackCombine(Stack& S1, Stack& S2)
14 {
15     slink p=S2->top;
16     if(!p) return;
17     while(p->next) p=p->next;
18     p->next=S1->top;
19     S1->top=S2->top;
20     S2->top=0;
21 }

```

习题 3.5 举两个应用栈的例子。

解答：

(1) 给定一个 C 语言源程序，判断其中"<"与">", "("与")", "["与"]", "{"与"}"符号的匹配情况。用栈实现源程序括号匹配检查的算法如下。

```

1 void MatchPairs(string expr, char ch1, char ch2)

```

```

2  {
3      Stack S=StackInit();
4      int j, len=expr.length();
5      for(int i=0; i<len; i++) {
6          if(expr[i]==ch1)
7              Push(i, S);
8          else if(expr[i]==ch2) {
9              if(StackEmpty(S)) printf("未匹配右括号 %d\n", i);
10             else j=Pop(S);
11         }
12     }
13     while(!StackEmpty(S)) {
14         j=Pop(S);
15         printf("未匹配左括号 %d\n", j);
16     }
17 }

```

(2) 表达式的后缀形式不需要用括号来表示运算的优先级，只要依次从左到右计算。对于给定的后缀表达式，计算其值。用栈实现计算后缀表达式值的算法如下。

```

1  int suff(string a)
2  {
3      int x, y, len=a.length();
4      Stack S=StackInit();
5      for(int i=0; i<len; i++) {
6          if(a[i]=='+') {
7              x=Pop(S); y=Pop(S); Push(y+x, S);
8          }
9          if(a[i]=='-') {
10             x=Pop(S); y=Pop(S); Push(y-x, S);
11         }
12         if(a[i]=='*') {
13             x=Pop(S); y=Pop(S); Push(y*x, S);
14         }
15         if(a[i]=='/') {
16             if(x==0) printf("除 0\n");
17             x=Pop(S);
18             y=Pop(S);
19             Push(y/x, S);
20         }
21         if(a[i]=='%'){
22             if(x==0) printf("除 0\n");
23             x=Pop(S);
24             y=Pop(S);
25             Push(y%x, S);
26         }
27         if((a[i]>='0') && (a[i]<='9')) Push(0, S);
28         while((a[i]>='0') && (a[i]<='9')) {

```

```

29         x=Pop(S);
30         Push(10*x +(a[i++]-'0'), S);
31     }
32 }
33 return StackTop(S);
34 }

```

习题 3.6 设有编号为 1~4 的 4 辆列车，顺序进入一个栈式结构的站台。试写出这 4 辆车开出车站的所有可能的顺序。

解答：

在一般情况下，计算所有可能的栈排列如下。

```

1 void back(int i)
2 {
3     int m;
4     if(i==n) {
5         outs(); cnt++; return;
6     }
7     if(!StackEmpty(s2)) {
8         m=Pop(s2); ss[i]=m;
9         back(i+1);
10        Push(m, s2);
11    }
12    if(!StackEmpty(s1)) {
13        m=Pop(s1); Push(m, s2);
14        back(i);
15        m=Pop(s2); Push(m, s1);
16    }
17 }

```

其中，outs()函数输出当前找到的栈排列，cnt 记录找到的栈排列数。事实上，全部的栈排列数就是卡特兰数

$$\frac{1}{n+1} \begin{bmatrix} 2n \\ n \end{bmatrix}$$

当 $n=4$ 时，共 14 个栈，排列如下：1234, 1243, 1324, 1342, 1432, 2134, 2143, 2314, 2341, 2431, 3214, 3241, 3421, 4321。

习题 3.7 试证明，借助栈，若输入序列 $1, 2, \dots, n$ 得到的输出序列为 $p(1), p(2), \dots, p(n)$ (输入序列的一个排列)，则在输出序列中不可能出现这样的情形，即存在 $i < j < k$ ，使 $p(j) < p(k) < p(i)$ 。

解答：

当 $i < j < k$ 时，表明 $p(i)$ 最先出栈， $p(j)$ 次之， $p(k)$ 最后出栈。如果 $p(j) < p(k) < p(i)$ 成立，则 $p(j) < p(i)$ 。

① 当 $p(j)$ 输出时，如果 $p(k)$ 已在栈中，则 $p(k)$ 比 $p(j)$ 先入栈，由输入序列知，此时有 $p(k) < p(j)$ 。

② 当 $p(j)$ 输出时, 如果 $p(k)$ 还没有入栈, 由输入序列知, 此时有 $p(k) > p(i) > p(j)$ 。

由此可见, 输出序列中不可能出现 $p(j) < p(k) < p(i)$ 的情形。这种情形称为 231 模式。借助栈得到的输出序列称为一个栈排列。事实上, 一个排列是栈排列的充分必要条件是排列不含 231 模式。长度为 n 的不含 231 模式的排列总数为卡特兰数

$$\frac{1}{n+1} \begin{bmatrix} 2n \\ n \end{bmatrix}$$

习题 3.8 用数组实现栈的一个缺点是需要预先估计栈的大小 maxtop 。克服这个缺点的一个方法是在初始化时先将 maxtop 置为 0, 插入一个元素时, 如果栈已满, 就重新分配一个大小为 $2 \times \text{maxtop} + 1$ 的数组, 将栈中元素复制到新数组中, 并删除老数组。类似地, 在删除栈顶元素后, 如果栈的大小已降至 $\text{maxtop}/4$, 就重新分配一个大小为 $\text{maxtop}/2$ 的新数组, 将栈中元素复制到新数组中, 并删除老数组。

(1) 用上述思想重新设计用数组实现栈的结构。

(2) 设 $p_1, p_2, \dots, p_n$ 是从空栈开始的 n 个由 Push 和 Pop 运算组成的栈运算序列, 如果教材中用数组实现栈的方法执行此运算序列需要 $F(n)$ 计算时间, 试证明用本题实现栈的方法执行此运算序列最多需要 $C \times F(n)$ 计算时间, 其中 C 是一个常数。

解答:

(1) 按此思想设计用数组实现栈如下。

```
1 Stack StackInit()
2 {
3     Stack S=(Stack)malloc(sizeof *S);
4     S->data=(StackItem *)malloc(sizeof(StackItem));
5     S->maxtop=0;
6     S->top=-1;
7     return S;
8 }
```

算法在第 5 行将数组大小 maxsize 初始化为 0。

在元素入栈时, 可能重新分配一个大小为 $2 \times \text{maxsize} + 1$ 的数组。

```
1 void Push(StackItem x, Stack S)
2 {
3     if(S->top==S->maxtop) {
4         S->maxtop=2*S->maxtop+1;
5         reSize1D(S->data, S->top+1, S->maxtop);
6     }
7     S->data[++S->top]=x;
8 }
```

算法在第 3~6 行重新分配一个大小为 $2 \times \text{maxsize} + 1$ 的数组。其中, $\text{reSize1D}()$ 函数用于调整数组大小, 需要 $O(n)$ 时间。

```
1 void reSize1D(StackItem * &a, int n, int newSize)
2 {
```

```

3     if(n>newSize) return;
4     StackItem *tmp=new StackItem [newSize];
5     for(int i=0; i<n; i++) tmp[i]=a[i];
6     delete []a;
7     a=tmp;
8 }

```

在元素出栈时，可能重新分配一个大小为 $(1/2)\text{maxsize}$ 的数组。

```

1 StackItem Pop(Stack S)      /* 前提：栈非空 */
2 {
3     if(StackEmpty(S)) exit(1);
4     StackItem x=S->data[S->top--];
5     if((S->top+1<=(S->maxtop+1)/4) && S->maxtop) {
6         S->maxtop=(S->maxtop-1)/2;
7         reSize1D(S->data, S->top+1, S->maxtop+1);
8     }
9     return x;
10 }

```

(2) 在用 `reSize1D()` 函数调整数组大小时，需要 $O(n)$ 时间。然而，这些时间可以均摊到每个运算。在入栈调整时，将调整所要的时间均摊到此前的每个新入栈运算。在出栈调整时，将调整所需的时间均摊到此前的每个新出栈运算。由此可知，每个运算均摊到 $O(1)$ 时间。因此，若教材中用原数组实现栈的方法执行此运算序列需要 $F(n)$ 计算时间，则用本题实现栈的方法执行此运算序列最多需要 $O(1) \times F(n)$ 计算时间，即 $C \times F(n)$ 计算时间，其中 C 是一个常数。

习题 3.9 当两个栈共享一个数组 `stack[0:n]` 时，试设计在第 k 个栈中加入元素 x 的算法 `Push(k, x, S)`，以及删除并返回第 k 个栈的栈顶元素的算法 `Pop(k, S)`。

解答：

两个栈共享一个数组实现的栈描述如下。

```

1 typedef struct astack *Stack; /* 栈指针类型 */
2 typedef struct astack {      /* 栈结构 */
3     int top1, top2,          /* 栈顶 */
4     maxtop;                  /* 栈空间上界 */
5     StackItem *data;         /* 存储栈元素的数组 */
6 } Astack;
7
8 Stack StackInit(int size)
9 {
10     Stack S=(Stack)malloc(sizeof *S);
11     S->data=(StackItem *)malloc(size*sizeof(StackItem));
12     S->maxtop=size;
13     S->top1=-1;
14     S->top2 = size;
15     return S;
16 }
17

```

```

18 int StackSize(int k, Stack S)
19 {
20     if(k==1) return S->top1+1;
21     else return S->maxtop-S->top2;
22 }
23
24 int StackEmpty(int k, Stack S)
25 {
26     if(k==1) return S->top1<0;
27 }
28
29 int StackFull(int k, Stack S)
30 {
31     return S->top1+1==S->top2;
32 }

```

其中，两个栈分别从数组的两端向中间扩展，可以充分利用已经分配的数组空间。

在第 k 个栈中加入元素 x 的算法 $\text{Push}(k, x, S)$ ，以及删除并返回第 k 个栈的栈顶元素的算法 $\text{Pop}(k, S)$ ，分别实现如下。

```

1  StackItem StackTop(int k, Stack S)      /* 前提：栈非空 */
2  {
3      if(StackEmpty(k, S)) exit(1);
5      if(k==1) return S->data[S->top1];
7      else return S->data[S->top2];
9  }
10
11 void Push(int k, StackItem x, Stack S)  /* 前提：栈未满 */
12 {
13     if(StackFull(k, S)) return;
15     if(k==1) S->data[++S->top1]=x;
17     else S->data[--S->top2]=x;
19 }
20
21 StackItem Pop(int k, Stack S)           /* 前提：栈非空 */
22 {
23     if(StackEmpty(k, S)) exit(1);
25     if(k==1) return S->data[S->top1--];
27     else return S->data[S->top2++];
29 }

```

习题 3.10 图 3-1 所示的数据结构是在一个数组中保存 3 个栈的方法。

类似地，可将 k 个栈存于一个数组中，若第 i 个栈的入栈运算 $\text{Push}(i, x, S)$ 使 $\text{Top}(i) = \text{Bottom}(i-1)$ ，则必须先移动所有的栈，使它们两两之间留有适当的空隙。例如，使所有栈的上方留有相等的空隙，或者使各栈上方的空隙与栈的当前长度成正比。

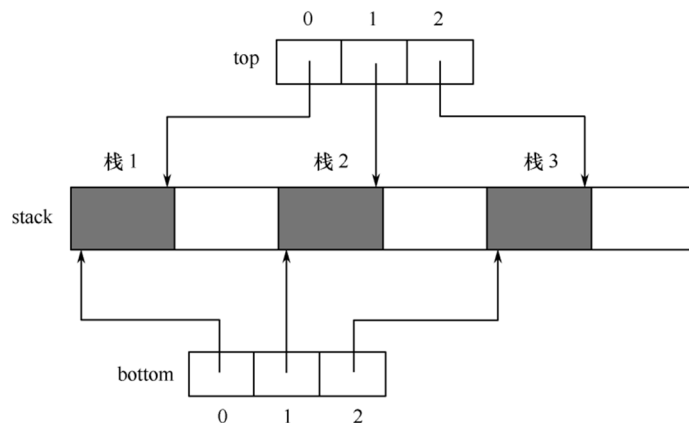


图 3-1 多个栈共享一个数组

- (1) 假设已有函数 **Reorg()** 能在两个栈发生冲突时调整各栈在数组中的位置，试重新设计用数组实现栈的结构。
- (2) 若已有函数 **NewTops()** 可计算出所有新栈顶的位置 **newtop[i]** ($1 \leq i \leq k$)，则如何实现函数 **Reorg()**? (提示：对各栈的位置进行调整时，第 i 个栈可能向上或向下移动。若第 i 个栈的新位置将与第 j ($j \neq i$) 个栈的老位置重合，则必须先移动第 j 个栈，才能移动第 i 个栈。为了调度各栈的移动次序，可以用另一个目标栈，其中的目标就是待移动栈的编号。依次考虑栈号 1, 2, ..., k 。当考虑第 i 个栈时，如果它能够安全移动，则被移到新位置，再处理目标栈顶的栈号；如果第 i 个栈还不能安全地移动，则将 i 推入目标栈。)
- (3) 如何实现 (2) 中的目标栈？是否必须将它作为整数栈？还有更简捷的表示方法吗？
- (4) 如何实现函数 **NewTops()**，使得每个栈上方留出的空隙与该栈的当前长度成正比？

解答：

(1) 将 k 个栈存于一个数组中的结构如下。

```

1  typedef struct astack *Stack;      /* 栈指针类型 */
2  typedef struct astack {            /* 栈结构 */
3      int n,                          /* 栈空间上界 */
4      k;                              /* 栈的个数 */
5      StackItem *data;                /* 存储栈元素的数组 */
6      int *top;                       /* 栈顶 */
7      int *bottom;                   /* 栈底 */
8  } Astack;
9
10 Stack StackInit(int k1, int size)
11 {
12     Stack S=(Stack)malloc(sizeof *S);
13     S->data=(StackItem *)malloc(size*sizeof(StackItem));
14     S->top=new int[k1+1];

```

```

15     S->bottom=new int[size];
16     S->n=size;
17     S->k=k1;
18     for(int i=0; i<=k1; i++) {
19         S->bottom[i]=size/k1*i;
20         S->top[i]=S->bottom[i]-1;
21     }
22     return S;
23 }
24
25 int StackEmpty(int sn, Stack S)
26 {
27     return S->top[sn]<S->bottom[sn];
28 }
29
30 int StackFull(int sn, Stack S)
31 {
32     return S->top[sn]+1>=S->bottom[sn+1];
33 }

```

如果已有 Reorg() 函数能在两个栈发生冲突时调整各栈在数组中的位置，则栈的基本运算实现如下。

```

1 void Push(int sn, StackItem x, Stack S) /* 前提：栈未满 */
2 {
3     if(StackFull(sn, S) && !Reorg(S)) return 0;
4     S->data[++S->top[sn]]=x;
5 }
6
7 StackItem Pop(int sn, Stack S) /* 前提：栈非空 */
8 {
9     if(StackEmpty(sn, S)) return 0;
10    return S->data[S->top[sn]--];
11 }

```

第 3~4 行由 Reorg() 函数调整各栈在数组中的位置，调整成功时返回 true，否则返回 false。

(2) 如果已有 NewTops() 函数可以计算出所有新栈顶的位置 newtop[i]，则 Reorg() 函数可以实现如下。

```

1 bool Reorg(Stack S)
2 {
3     int stk, cnt=0;
4     int* newtop=new int[S->k];
5     if(!NewTops(newtop, S)) {
6         free(newtop);
7         return false;
8     }

```

```

9      for(int i=1; i<S->k; i++) {
10         if(newtop[i]>=S->bottom[i+1]) puts(i, stk, cnt);
11         else {
12             shift(i, newtop[i], S);
13             for(int j=stk+cnt-1; j>=stk; j--) shift(j, newtop[j], S);
14             cnt=0;
15         }
16     }
17     free(newtop);
18     return true;
19 }

```

第 5~8 行用 NewTops()函数计算新的栈顶, 计算成功返回 true, 否则返回 false。其中, shift()函数用于移动栈到新的位置。

```

1 void shift(int sn, int ntop, Stack S)
2 {
3     int len=S->top[sn]-S->bottom[sn]+1;
4     for(int i=0; i<len; i++) S->data[ntop-i]=S->data[S->top[sn]-i];
5     S->top[sn]=ntop;
6     S->bottom[sn]=ntop-len+1;
7 }

```

(3) 算法 Reorg 中的目标栈可以看作一个整数栈。由于这个整数栈的特殊性, 即栈中的栈号是连续整数, 可以用一个很简单的方法实现。在上述算法中用 puts()函数实现如下。

```

1 void puts(int st, int& stk, int& cnt)
2 {
3     if(!cnt) stk=st;
4     cnt++;
5 }

```

其中, stk 是初始栈号, cnt 是连续栈个数。Reorg()函数中的第 10~11 行是连续出栈运算。

(4) 设数组的长度为 n , 栈 $S[i]$ 的当前长度为 $l_i = \text{top}[i] - \text{bottom}[i] + 1$ 。

k 个栈的当前长度之和为

$$m = \sum_{i=0}^{k-1} l_i$$

重新分配后, 每个栈的可用空间至少为 1。剩余可分配空间是 $n - m - k$ 。将这些空间按照当前栈长度 l_i 等比例分配给各栈。每个栈可分配空间为

$$r_i = (n - m - k) \frac{l_i}{m}$$

因此, 栈 $S[i]$ 上方留出的空隙为 $g_i = \lfloor 1 + r_i \rfloor$ 。记 $t_i = \text{newtop}[i]$, 则

$$t_i = \begin{cases} \text{top}[0], & i = 0 \\ t_{i-1} + g_{i-1} + l_i, & i > 0 \end{cases}$$

按此思想实现的算法如下。

```

1 bool NewTops(int* newtop, Stack S)

```

```

2  {
3      int len=0;
4      for(int i=0; i<S->k; i++) {
5          newtop[i]=S->top[i]-S->bottom[i]+1;
6          len+=newtop[i];
7      }
8      if(len+S->k>=S->n) return false;
9      int l=newtop[0];
10     newtop[0]=S->top[0];
11     for(int i=1; i<S->k; i++) {
12         int g=(1+(S->n-len-S->k)*l/len);
13         l=newtop[i];
14         newtop[i]=newtop[i-1]+g+1;
15     }
16     return true;
17 }

```

(5) 用游标模拟指针，将 k 个栈存于一个数组中的空间效率更高，且易于实现如下。

```

1  typedef struct astack *Stack; /* 栈指针类型 */
2  typedef struct astack {      /* 栈结构 */
3      int n,                    /* 栈空间上界 */
4      k,                        /* 栈的个数 */
5      free;                     /* 可用单元链首 */
6      StackItem *data;          /* 存储栈元素的数组 */
7      int *top;                 /* 栈顶 */
8      int *next;                /* 游标 */
9  } Astack;
10
11 Stack StackInit(int k1, int size)
12 {
13     Stack S=(Stack)malloc(sizeof *S);
14     S->data=(StackItem *)malloc(size*sizeof(StackItem));
15     S->top=new int[k1];
16     S->next=new int[size];
17     S->n=size;
18     S->k=k1;
19     S->free=0;
20     for(int i=0; i<S->k; i++) S->top[i]=-1;
21     /* 初始化可用单元 */
22     for(int i=0; i<S->n-1; i++) S->next[i]=i+1;
23     S->next[S->n-1]=-1;        /* -1 表示可用单元链尾 */
24     return S;
25 }
26
27 int StackEmpty(int sn, Stack S)
28 {
29     return S->top[sn]==-1;
30 }

```

```

31
32 int StackFull(Stack S)
33 {
34     return S->free== -1;
35 }

```

该算法的栈的基本运算实现如下。

```

1 void Push(int sn, StackItem x, Stack S) /* 前提: 栈未满 */
2 {
3     if(StackFull(S)) return;
4     int i=S->free;          /* 第一个可用单元 */
5     S->free=S->next[i];      /* 下一个可用单元 */
6     S->next[i]=S->top[sn];    /* 栈 sn 当前栈顶 */
7     S->top[sn]=i;
8     S->data[i]=x;
9 }
10
11 StackItem Pop(int sn, Stack S) /* 前提: 栈非空 */
12 {
13     if(StackEmpty(sn, S)) return 0;
14     int i=S->top[sn];
15     S->top[sn]=S->next[i];    /* 栈 sn 栈顶的下一个单元 */
16     S->next[i]=S->free;      /* 回收单元 i */
17     S->free=i;
18     return S->data[i];
19 }

```

从算法描述容易看出, 基本运算需要 $O(1)$ 时间。

习题 3.11 试设计一个最小栈 **MinStack**, 除了用 $O(1)$ 时间支持所有栈运算, 还支持 $O(1)$ 时间 **find-min()** 运算, 即用 $O(1)$ 时间找出当前栈中最小元素。除了存储栈中元素所需空间, 只能用 $O(1)$ 额外空间。

解答:

(1) 先不考虑只能用 $O(1)$ 额外空间的约束条件。可以用两个栈结构来实现 $O(1)$ 时间 **find-min()** 运算。其中, 栈 **stk** 按常规定义存储栈元素, 辅助栈 **minstk** 用于存储当前栈中最小元素。在任何时刻, 栈 **minstk** 的栈顶元素 **StackTop(minstk)** 始终是当前栈中最小元素。在执行 **Push** 运算时, 如果待入栈元素 x 满足 $x \leq \text{StackTop}(\text{minstk})$, 则元素 x 同时入栈 **stk** 和 **minstk**, 否则元素 x 只入栈 **stk**。

在执行 **Pop** 运算时, 如果待出栈元素 **StackTop(stk)** 满足 **StackTop(stk)=StackTop(minstk)**, 则同时抛栈 **stk** 和 **minstk**, 否则只抛栈 **stk**。

该算法可以保证栈 **minstk** 的栈顶元素在任何时刻均为栈 **stk** 中元素的最小值, 从而可以实现 $O(1)$ 时间 **find-min()** 运算。其他所有运算也都只要 $O(1)$ 时间。

该算法在最坏情况下需要 $O(n)$ 额外空间。

(2) 为了满足用 $O(1)$ 额外空间的约束条件, 可以采用如下技巧。变量 **minE** 存储当前栈中的最小元素。在执行 **Push** 运算时, 如果待入栈元素是 x , 则将 $x - \text{minE}$ 存入栈中。这样, 在任

何时刻， minE 均为栈 stk 中元素的最小值，从而可以实现 $O(1)$ 时间 $\text{find-min}()$ 运算。

如何用 $O(1)$ 时间正确执行其他栈运算？以教材中用数组实现的栈为例，来说明其他栈运算的算法如下。

```
1  typedef struct astack{    /* 栈结构 */
2      int top,              /* 栈顶 */
3      maxtop;              /* 栈空间上界 */
4      StackItem minE;      /* 最小元素 */
5      StackItem *data;     /* 存储栈元素的数组 */
6  }Astack;
```

栈结构中增加了变元 minE 。

```
1  Stack StackInit(int size, StackItem mE)
2  {
3      Stack S=(Stack)malloc(sizeof *S);
4      S->data=(StackItem *)malloc(size*sizeof(StackItem));
5      S->maxtop=size;
6      S->top=-1;
7      S->minE=mE;
8      return S;
9  }
```

由于此时栈元素是有序集中的元素，因此可以比较大小，需要一个表示 ∞ 的元素。当栈空时，用此元素表示最小值。栈初始化时会初始化 minE 。

```
1  StackItem Min(Stack S)    /* 前提：栈非空 */
2  {
3      if(StackEmpty(S)) exit(1);
4      return S->minE;
5  }
```

最小值计算函数 $\text{Min}()$ 显然只要 $O(1)$ 时间。

```
1  void Push(StackItem x, Stack S)    /* 前提：栈未满 */
2  {
3      if(StackFull(S)) exit(1);
4      S->data[++S->top]=x-S->minE;
5      S->minE=min(x, S->minE);
6  }
```

在执行 Push 运算时，如果待入栈元素是 x ，则将 $x - \text{minE}$ 存入栈中。注意当 $x < \text{minE}$ 时，算法是先将 $x - \text{minE}$ 存入栈中，再修改 minE 的值。修改后， $x = \text{minE}$ 。若设此时栈顶元素为 y ，则 $\text{minE} - y$ 的值是修改前的 minE 的值。当 $x \geq \text{minE}$ 时， $\text{minE} - y = 2\text{minE} - x \leq \text{minE}$ 。由此可知，执行 Pop 运算后， minE 的值改变为 $\max\{\text{minE} - y, \text{minE}\}$ 。因此， Pop 运算可实现如下。

```
1  StackItem Pop(Stack S)    /* 前提：栈非空 */
2  {
3      if(StackEmpty(S)) exit(1);
```

```

4      S->minE=max(S->minE-S->data[S->top], S->minE);
5      return max(S->data[S->top--]+S->minE, S->minE);
6  }

```

注意，Pop 运算需返回出栈的栈顶元素，有两种情况。当栈顶元素不是最小元素时，易知此时 $\text{minE} + y = \text{minE} + x - \text{minE} = x$ ；当栈顶元素是最小元素时，有 $\text{minE} = x$ ，且 $\text{minE} + y = \text{minE} + x - \text{minE}' \leq x$ ，其中 minE' 是上一个最小值。由此可见，在任何时刻，总有栈顶元素为 $x = \max\{\text{minE} + y, \text{minE}\}$ 。因此，StackTop 运算可实现如下。

```

1  StackItem StackTop(Stack S)    /* 前提：栈非空 */
2  {
3      if(StackEmpty(S)) exit(1);
4      return max(S->data[S->top]+S->minE, S->minE);
5  }

```

栈的其他运算保持不变。因此，上述支持最小值运算的栈的所有运算只需要 $O(1)$ 时间，只用了 $O(1)$ 额外空间。

算法实验题 3

算法实验题 3.1 最大全 1 子矩阵问题。

★ **问题描述：**给定一个由数字 0 和 1 组成的二维矩阵，找出它的最大全 1 子矩阵。这里所说的最大是指子矩阵的面积（行数乘以列数）最大。

★ **实验任务：**对于给定的由数字 0 和 1 组成的二维矩阵，计算其最大全 1 子矩阵的面积。例如，二维矩阵如下

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

其最大全 1 子矩阵的面积是 8。

★ **数据输入：**由文件 input.txt 给出输入数据，有多个测试项。每个测试项的第 1 行有两个正整数 n 和 m ($0 < n, m < 9000$)，表示由 0 和 1 组成的二维矩阵的行数和列数。接下来的 n 行，每行有 m 个由 0 和 1 组成的数字。

★ **结果输出：**将计算出的每个测试项的最大面积依次输出到文件 output.txt。每行输出一个数。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	8
0 1 1 0	8
1 1 1 1	
1 1 1 1	
1 1 0 0	
4 4	
0 1 1 0	

1 1 1 1	
1 1 1 1	
1 1 0 0	

解答:

可以利用教材中例 3.2 的求直方图最大矩形面积算法解此问题。将二维矩阵中每一行上方的图形看作一个直方图，求其最大面积。所有面积中最大者即所求。

```

1  int histo(int h[])
2  {
3      Stack stk=StackInit(n);
4      int i=0, mx=0;
5      while(i<n) {
6          if(StackEmpty(stk) || h[StackTop(stk)]<=h[i]) Push(i++, stk);
7          else {
8              int tmp=StackTop(stk);
9              Pop(stk);
10             int a=h[tmp]*(StackEmpty(stk) ? i : i-StackTop(stk)-1);
11             if(mx<a) mx=a;
12         }
13     }
14     /* 计算栈中剩余的非减序列 */
15     while(!StackEmpty(stk)) {
16         int tmp=StackTop(stk);
17         Pop(stk);
18         int a=h[tmp]*(StackEmpty(stk)? i : i-StackTop(stk)-1);
19         if(mx<a) mx=a;
20     }
21     return mx;
22 }

```

教材例 3.2 的求直方图最大矩形面积算法入选。

```

1  int maxrec(int a[][n])
2  {
3      int ret=histo(a[0]);
4      for(int i=1; i<m; i++) {
5          for(int j=0; j<n; j++)
6              if(a[i][j])a[i][j]+=a[i-1][j];
7          ret=max(ret, histo(a[i]));
8      }
9      return ret;
10 }

```

histo()函数的第 5~6 行计算第 i 行上方的直方图，第 7 行计算相应直方图中的最大矩形面积。由于计算直方图中最大矩形面积需要 $O(n)$ 时间，因此上述算法需要的计算时间是 $O(mn)$ 。

算法实验题 3.2 大牌明星问题。

★ **问题描述：** 在一个由 n 个人参加的聚会中，可能有一个大牌明星也参加了聚会。这个大牌明星不认识任何参加聚会者，但参加聚会者都认识他。如果聚会者有大牌明星，可以通过若干“A 认识 B 吗”这种形式的问题，来确定参加聚会者中的大牌明星。

★ **实验任务：** 假定 n 个人的编号为 $0, 1, \dots, n-1$ 。由矩阵 A 给出对于“A 认识 B 吗”这种形式的问题的回答。 $A[i][j]=0$ 表示“ i 不认识 j ”， $A[i][j]=1$ 表示“ i 认识 j ”。设计一个算法，用最少的询问找到大牌明星。

★ **数据输入：** 由文件 input.txt 给出输入数据。第 1 行是参加聚会的人数 n ($0 < n < 8000$)；从第 2 行起的 n 行，每行有 n 个数字 0 或 1。

★ **结果输出：** 将找到的大牌明星的编号输出到文件 output.txt；如果参加聚会者中没有大牌明星，则输出-1。

输入文件示例	输出文件示例
input.txt	output.txt
4	2
0 0 1 0	
0 0 1 0	
0 0 0 0	
0 0 1 0	

解答：

注意：若 A 认识 B，则 A 不是大牌明星，可以排除 A；若 A 不认识 B，则 B 不是大牌明星，可以排除 B。用一次询问就可以排除一个参加聚会者。可以用一个栈来完成排除任务，代码如下。

```
1 int find(int n)
2 {
3     Stack s=StackInit(N);
4     int ret;
5     for(int i=0; i<n; i++) Push(i, s);
6     int A=Pop(s), B=Pop(s);
7     while(!StackEmpty(s)) {
8         if(k(A, B)) A=Pop(s);
9         else B=Pop(s);
10    }
11    if(k(A, B)) ret=B;
12    else ret=A;
13    for(int i=0; i<n; i++)
14        if((i != ret) && (k(ret, i) || !k(i, ret))) return -1;
15    return ret;
16 }
```

第 8 行排除 A，第 9 行排除 B，第 13~15 行最终确认候选者是否为大牌明星。算法需要的计算时间显然是 $O(n)$ 。

算法实验题 3.3 反向字符串输出问题。

★ **问题描述：** 对于给定的由空格分隔的字符串，反向输出其中的每个字符串。

★ **实验任务：**给定的由空格分隔的若干字符串，反向输出其中的每个字符串。例如，设给定字符串是"Hello World"，则反向输出的字符串是"olleH dlroW"。

★ **数据输入：**由文件 input.txt 给出输入数据。第 1 行是正整数 n ，表示有 n 个测试项；接下来的 n 行，每行给出一个由空格分隔的字符串。

★ **结果输出：**将输入数据中每个由空格分隔的字符串依次反向输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
1	olleH dlroW
Hello World	

解答：

用一个栈将字符串中所有字符存入栈中，一旦遇到空格，就输出栈中所有字符。

```

1 void reverse(string str)
2 {
3     Stack st=StackInit(N);
4     for(int i=0; i<str.length(); ++i) {
5         if(str[i]!=' ') Push(str[i], st);
6         else {
7             while(!StackEmpty(st)) cout << Pop(st);
8             cout << " ";
9         }
10    }
11    while(!StackEmpty(st)) cout << Pop(st);
12 }
```

算法需要的计算时间显然是 $O(n)$ 。

算法实验题 3.4 亲兄弟问题。

★ **问题描述：**给定 n 个整数 $a_0, a_1, \dots, a_{n-1}$ 组成的序列，序列中元素 a_i 的亲兄弟元素 a_k 定义为 a_i 的右边最靠近它且不小于它的元素，即

$$k = \min_{i < j < n} \{j | a_j \geq a_i\}$$

亲兄弟问题要求给出序列中每个元素的亲兄弟元素的位置。元素 a_i 的亲兄弟元素为 a_k 时，称 k 为元素 a_i 的亲兄弟元素的位置。当元素 a_i 没有亲兄弟元素时，约定其亲兄弟元素的位置为 -1。

例如，当 $n=10$ ，整数序列为 6, 1, 4, 3, 6, 2, 4, 7, 3, 5 时，相应的亲兄弟元素位置序列为：4, 2, 4, 4, 7, 6, 7, -1, 9, -1。

★ **实验任务：**对于给定的 n 个整数 $a_0, a_1, \dots, a_{n-1}$ 组成的序列，试用栈设计一个 $O(n)$ 时间算法，计算相应的亲兄弟元素位置序列。

★ **数据输入：**由文件 input.txt 提供输入数据。文件的第 1 行有 1 个正整数 n ，表示给定 n 个整数；第 2 行是 $a_0, a_1, \dots, a_{n-1}$ 。

★ **结果输出：**程序运行结束时，将计算出的与给定序列相应的亲兄弟元素位置序列输出到 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
10 6 1 4 3 6 2 4 7 3 5	4 2 4 4 7 6 7 -1 9 -1

解答:

最直观的解法是逐个元素扫描找出亲兄弟元素。每次扫描在最坏情况下需要 $O(n)$ 时间, 因此, 用逐个元素扫描的算法需要 $O(n^2)$ 时间。

事实上, 只要用一次线性扫描就可以找出所有元素的亲兄弟元素。注意, 在线性扫描过程中, 一旦找到元素 $a[i+1] \geq a[i]$, 从而确定 $a[i]$ 的亲兄弟元素是 $a[i+1]$ 时, 此前未确定亲兄弟的元素中大于 $a[i]$ 且不超过 $a[i+1]$ 的亲兄弟元素也是 $a[i+1]$ 。例如, 在输入文件示例中, 当 $i=3$ 时, 发现 $a[4]=6$ 是元素 $a[3]=3$ 的亲兄弟元素, 此时尚未确定亲兄弟的元素 $a[2]=4$, 满足条件大于 $a[i]$ 且不超过 $a[i+1]$ 。因此, $a[4]$ 也是 $a[2]$ 的亲兄弟元素。

按照上述思想, 可以用一个单调栈来存储扫描过程中未找到亲兄弟的元素。一旦找到元素的一个亲兄弟元素, 就可以从栈中找到有相同亲兄弟的所有元素。具体算法实现如下。

```

1 void dom(int n)
2 {
3     Stack s=StackInit(n);
4     for(int i=0; i<n; i++) d[i]=-1;
5     Push(0, s);
6     for(int i=1; i<n; i++) {
7         while(!StackEmpty(s) && x[StackTop(s)]<=x[i]) d[Pop(s)]=i;
8         Push(i, s);
9     }
10 }
```

栈 s 用于存储待确定亲兄弟的元素。在扫描到元素 $x[i]$ 时, 依次考察 s 的栈顶元素 (第 7 行), 发现栈顶元素 $\leq x[i]$, 则元素 $x[i]$ 就是他的亲兄弟元素, 并删除栈顶元素, 同时记录该栈顶元素的亲兄弟元素为 $x[i]$ 。算法只需要 $O(n)$ 时间和空间。因此, 该算法可以在最优的线性时间内找到并输出所有元素的亲兄弟元素。

```

1 int main()
2 {
3     int n;
4     scanf("%d", &n);
5     x=new int[n];
6     d=new int[n];
7     for(int i=0; i<n; i++) scanf("%d", &x[i]);
8     dom(n);
9     for(int i=0; i<n; i++) {
10         printf("%d ", d[i]);
11         puts("");

```

```
12     }  
13     return 0;  
14 }
```

电子工业出版社版权所有
盗版必究