3.2 单元测试

一、单项选择题

1. B　2. D　3. D　4. C　5. D　6. B　7. A　8. D　9. A　10. B　11. B　12. A　13. A　14. A
15. D　16. B　17. C　18. C　19. A　20. D　21. A　22. C　23. C　24. C　25. C　26. C
27. C　28. A　29. B　30. A/C　31. B　32. A　33. B　34. D　35. C　36. A　37. D　38. C
39. B

二、填空题

1. 顺序　2. O(n)；O(1)　3. p->next；s->data；t　4. 1　5. 0；n　6. O(n)　7. n-i+1
8. O(1)；O(n)　9. O(1)；O(n)　10. p->next!=NULL　11. py->next=px->next；px->next=py
12. L->next->next=L　13. m-1　14. lq->front->next==lq->rear　15. 先进后出；先进先出
16. 2,3；100C　17. data[++top]=x　18. (rear-front+m)%m　19. r==f；(r+1)%m==f　20. 19

三、应用题

1.【解析】因为栈受限制在栈顶输入或输出，而且有“先进后出”的特点，所以有如下几种情况：（1）a 进 a 出 b 进 b 出 c 进 c 出，产生输出序列为 abc；（2）a 进 a 出 b 进 c 进 c 出 b 出，产生输出序列为 acb；（3）a 进 b 进 b 出 a 出 c 进 c 出，产生输出序列为 bac；（4）a 进 b 进 b 出 c 进 c 出 a 出，产生输出序列为 bca；（5）a 进 b 进 c 进 c 出 b 出 a 出，产生输出序列为 cba；不可能产生输出序列 cab。

2.【解析】入队与出队的队列变化如图所示，当元素 D,E,B,G,H 入队后，rear=6，front=1；元素 D,E 出队，rear=6，front=3；元素 I,J,K,L,M 入队，rear=0，front=3；元素 B 出队后，rear=0，front=4；此时再让元素 N,O,P 入队，当 Q 入队时，由于 rear=3，front=4，有(rear+1+11)%11=front，因此不能入队。

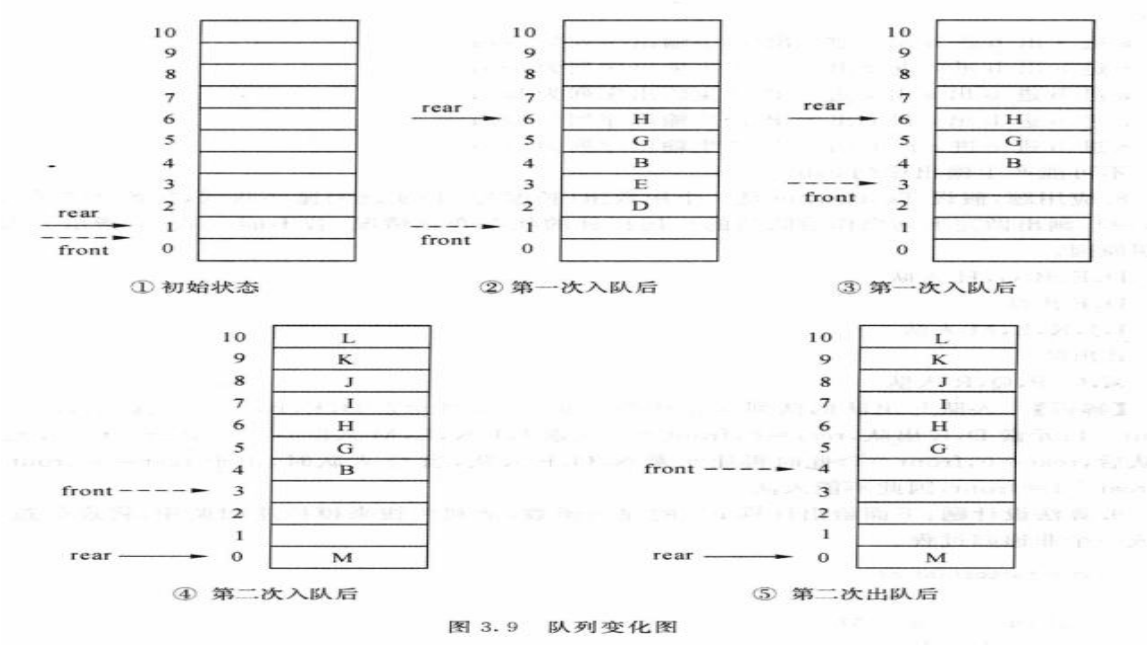


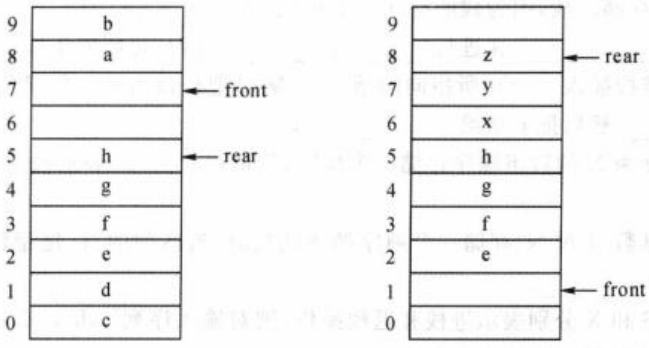
图 3.9 队列变化图

3.【解析】元素个数=(rear-front+Maxsize)%Maxsize，其中，Maxsize 表示队列的容量。故：①元素个数=(19-11+40)%40=8；②元素个数=(11-19+40)%40=32。

在第一种情况下，循环队列中元素个数为 8 个；在第二种情况下，循环队列中元素个数为 32 个。

4.【解析】Push;Pop;Push;Push;Pop;Pop;Push;Push;Pop;Push;Push;Pop;Pop;Pop。

5.【解析】



四、算法设计题

1.【解析】

```
void Insert(List L, int x)
    {
        int i,j;
```

```
        for(i=0;L->Data[i] < x&&i<=L->Last;i++)     ;
        for(j=L->Last; j>=i; j--)
            L->Data[j+1]=L->Data[j];
        L->Data[i] = x;
        L->Last++;
    }
```

2.【解析】

```
void reverse(List L)
{
    int i,j;
    ElementType temp;
    for(i=0,j=L->Last;i<j;i++,j--){
        temp=L->Data[i];
        L->Data[i]=L->Data[j];
        L->Data[i]=temp;
    }
}
```

3.【解析】

```
void deletex(List L,int x)
{
    int i=0,j;
    while(i<=L->Last && L->Data[i]!=x)
        i++;
    for(j=i+1;j<=L->Last;j++)
        if(L->Data[j]!=x)
        {
            L->Data[i]=L->Data[j];
            i++;
        }
    L->Last=i-1;
}
```

4.【解析】

```
void merge(List A ,List B)
{
    int i=A->Last,j=B->Last,k=A->Last+B->Last+1;
    //i,j,k 分别指向 A，B 及新表的最后一个元素的位置
    while(j>=0){
        if(i<0||A->Data[i]<B->Data[j]){
            A->Data[k]=B->Data[j];
            k--;
```

```
            j--;
        }
        else{
            A->Data[k]=A->Data[i];
            k--;
            i--;
        }
    }
    A->Last=A->Last+B->Last+1;
}
```

5.【解析】

```
void delete(List L ,ElementType min,ElementType max)
{   //首先找到要删除的元素位置（从 i 到 j-1），然后删除
    int i=0,j,k,d;
    while(i<=L->Last&&L->Data[i]<=min)
        i++;
    j=i;
    while(j<=L->Last&&L->Data[j]<max)
        j++;
    d=j-i;
    if(d==0)    return;
    for(k=j;k<=L->Last;k++,i++)
        L->Data[i]=L->Data[k];
    L->Last=i;
}
```

6.【解析】

```
int delx(List L, ElementType x)
{
    List pre = L;              //pre 指向 p 的前驱节点
    List p = pre->Next;
    while (p != NULL && p->Data!=x) {
        pre = p;
        p = p->Next;               //pre、p 同步后移一个节点
    }
    if (p != NULL) {               //找到值为 x 的 p 节点
```

```
            pre->Next = p->Next;
            free(p);
            return 1;
        } else {
            return 0;                    //未找到值为 x 的 p 节点
        }
    }
```

7.【解析】

```
    typedef struct Node{
            ElemType data;
            struct Node *next;
        }*LinkList;
        ElemType FindMax(LinkList L){
            if(L->next==NULL)      exit();
            LinkList pmax,p;
            pmax=L->next;
            p=L->next->next;
            while(p!=NULL){
                if(p->data > pmax->data)
                        pmax=p;
                p=p->next;}
            return pmax->data;
        }
```

8.【解析】

```
    void delete(List L,int mink,int maxk){
        p=L->Next;   //首元结点
        while(p&&p->Data<=mink)
        {   pre=p;   p=p->Next;} //查找第一个值大于 mink 的结点
        if(p)
        {    while(p&&p->Data<maxk)
                p=p->Next;   //查找第一个值大于等于 maxk 的结点
            q=pre->Next;
            pre->Next=p; //修改指针
            while(q!=p)
            { s=q->Next; free(q);   q=s;} //释放结点空间
        }
```

```
}
```

9.【解析】

```
void Insertx(List L,ElementType x){//L 为带表头结点的有序单链表
    PtrToLNode p=L,s;
    while(p->Next->Data<x)   //寻找插入位置
        p=p->Next;
    s = (PtrToLNode)malloc(sizeof(struct LNode));
    s->Data=x;
    s->Next=p->Next;
    p->Next=s;
}
```

10.【解析】

```
typedef struct SNode
{
    ElementType Data[Maxsize];
    int Top0,Top1;
}DblStack;
```

（1）初始化算法：

```
void InitStack (DblStack* S)
{
    S=(DblStack *)malloc(sizeof(struct SNode));
    S->Top0=-1;
    S->Top1=Maxsize;
}
```

(2)判空：

```
int EmptyStack(DblStack*S, int i)
{
    if(i==0)
        return (S->Top0==-1)
    else
        return (S->Top1==Maxsize)
}
```

（3）入栈算法：

```
void push(DblStack* S,int i,ElementType x)
{
    if(S->Top0+1==S->Top1)
        return(overflow);
    if(i==0)
```

```
        {
            S->Top0++;
            S->Data[S->Top0]=x;
        }
        else
        {
            S->Top1--;
            S->Data[S->Top1]=x;
        }
    }
```

（4）出栈算法：

```
    ElementType pop(DblStack* S,int i)
    {
        if(i==0)
        {
            if(S->Top0==-1)
                exit();
            return S->Data[S->Top0--];
        }
        else
        {
            if(S->Top1==Maxsize)
                exit();
            return S->Data[S->Top1++];
        }
    }
```

11.【解析】

初始化栈：

```
    Stack CreateStack( )
    {
      Stack S=NULL;
      return S;
    }
```

判空：

```
    int IsEmpty ( Stack S )
    { /*  判断堆栈 S 是否为空，若是返回 1；否则返回 0 */
      return ( S == NULL );
    }
```

进栈：

```
    void Push( Stack S, ElementType X )
```

```
{ /* 将元素 X 压入堆栈 S */
    PtrToSNode TmpCell;
    TmpCell = (PtrToSNode)malloc(sizeof(struct SNode));
    TmpCell->Data = X;
    TmpCell->Next = S;
    S = TmpCell;
}
```

出栈：

```
ElementType Pop( Stack S )
{ /* 删除并返回堆栈 S 的栈顶元素 */
    PtrToSNode FirstCell;
    ElementType TopElem;
    if( IsEmpty(S) ) {
        printf("堆栈空");     return ERROR;
    }
    else {
        FirstCell = S;
        TopElem = FirstCell->Data;
        S = FirstCell->Next;
        free(FirstCell);
        return TopElem;
    }
}
```

12.【解析】

```
typedef struct Node *PtrToNode;
struct Node {        /* 队列中的结点 */
    ElementType Data;
    PtrToNode Next;
};
typedef struct QNode *PtrToQNode;
struct QNode {
    PtrToNode    rear;          /* 队列的尾指针 */
};
typedef PtrToQNode Queue;
```

初始化：

```
Queue CreateQueue( )
{
```

```
    Queue Q= (Queue)malloc(sizeof(struct QNode));
    Q->rear=NULL;
    return Q;
}
```

入队：

```
void AddQ( Queue Q, ElementType X )
{ /* 将元素 X 入队 */
    PtrToNode TmpCell;
    TmpCell = (PtrToNode)malloc(sizeof(struct Node));
    TmpCell->Data = X;
    TmpCell->Next=Q->rear->Next;
    Q->rear->Next=TmpCell;
}
```

出队：

```
ElementType DeleteQ( Queue Q )
{   Position  FrontCell;
    ElementType  FrontElem;
    if  ( IsEmpty(Q) ) {
    printf("队列空");      return ERROR;
    }
    else {
    FrontCell = Q->rear->Next;
    if ( Q->rear == Q->rear->Next )  /* 若队列只有一个元素 */
        Q->rear = NULL;  /* 删除后队列置为空 */
    else
        Q->rear->Next = FrontCell->Next;
    FrontElem = FrontCell->Data;
    free( FrontCell );   /* 释放被删除结点空间  */
    return  FrontElem;
    }
}
```

判队空：

```
bool IsEmpty( Queue Q )
{   return ( Q->rear == NULL);
}
```