数据结构习题 第 3 章

3-1

- 题目描述

编写算法Reverse (A[ ], n)，将顺序存储的线性表A=(a_0, a_1, …, a_{n-1})转换为A=(a_{n-1},…, a_1, a_0)，要求转换过程中用尽可能少的辅助空间。

A[0]	A[1]	…	A[$n-2$]	A[$n-1$]
a_0	a_1	…	a_{n-2}	a_{n-1}
a_{n-1}	a_{n-2}	…	a_1	a_0

- 只需从线性表的第1个数据元素开始，将第i个数据元素与第n-i-1个数据元素相交换即可。在这个过程中，i的变化范围是0到 $\lfloor (n-1)/2 \rfloor$ 。

参考答案

算法Reverse(A, n. A)

FOR i=0 TO $\lfloor (n-1)/2 \rfloor$ **DO** (

A[i] ↔ A[n-i-1].

)▌

3-3

- 已知长度为n的线性表A采用顺序结构存储，请写一算法，找出该线性表中值最小的元素的位置。

参考答案

算法FMIN(A, n. pos)

/*找顺序表A中最小元素的位置*/

FM1[初始化]

pos ←**0**

FM2 [循环]

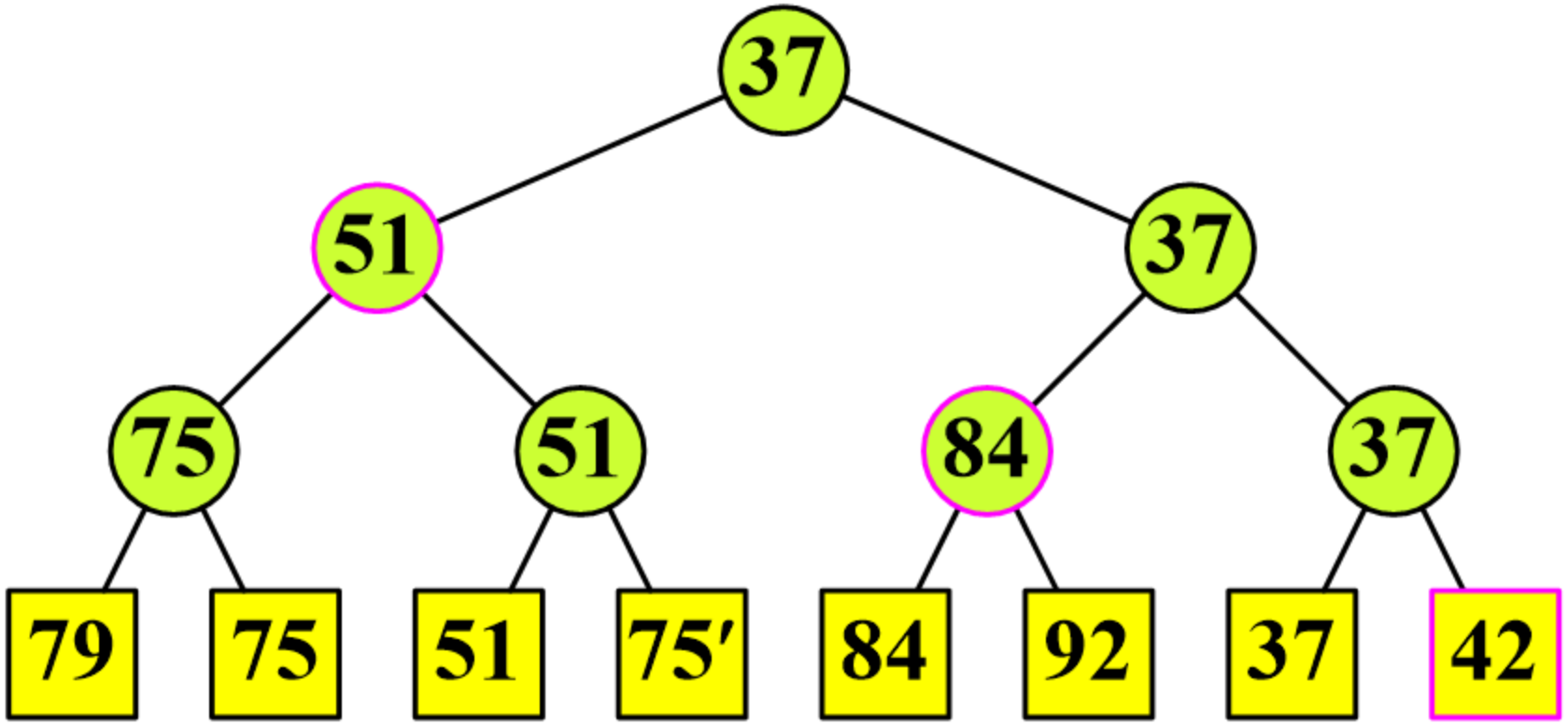
FOR i ←1 TO n-1 **DO**

IF A[i] < A[pos] **THEN** pos ←i.

扩展

- 找第2小的元素

思路



3-6

- 分析单链表中删除操作的时间复杂度

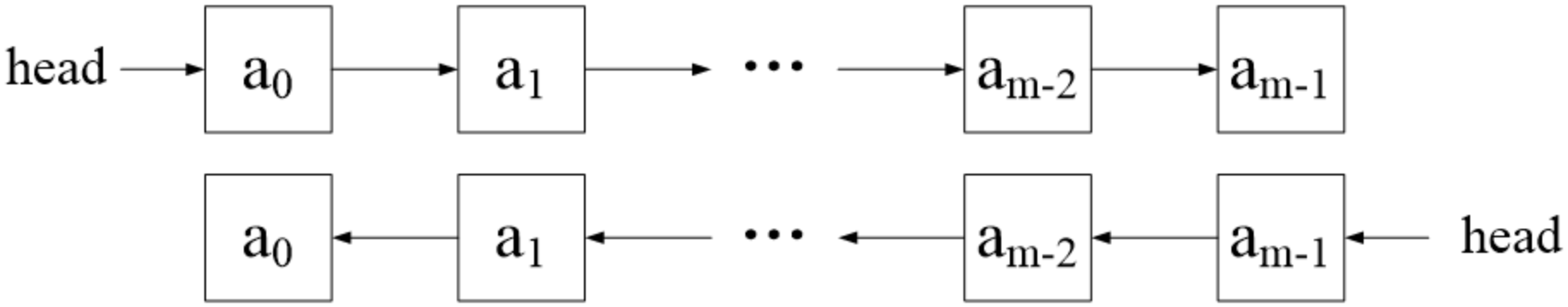
参考答案

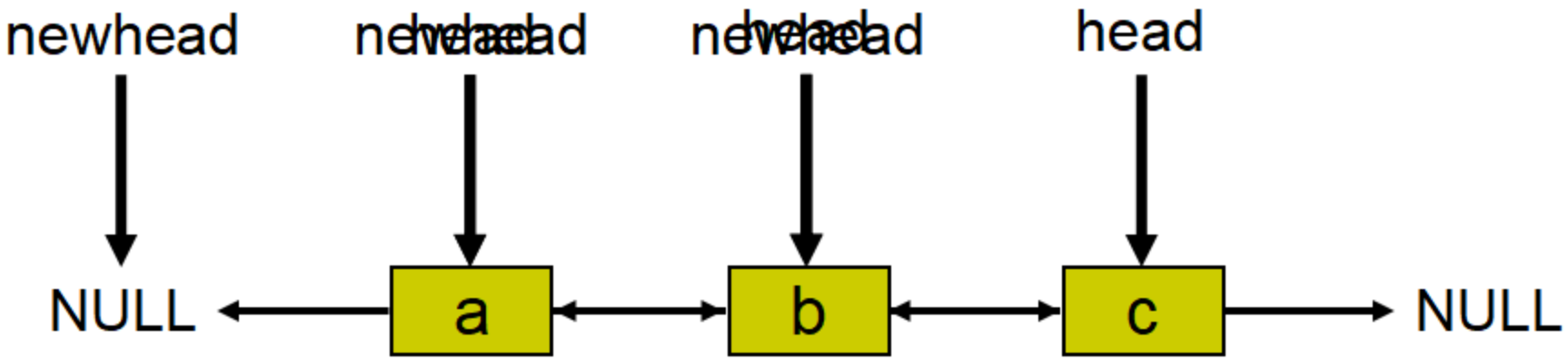
- 算法delete(k, item)，需要从前往后找到第k个元素。
- 关键操作：指针调整
  - 最好情况k<=0，不用调整，O(1)
  - 最坏情况k>n-1，调整n次，O(n)
  - 一般情况，假定k<0, k=0, …, k=n-1, k>n-1的概率相同，调整次数为

$$\frac{0+0+\ldots+n-1+n}{n+2}=\frac{n(n+1)}{2(n+2)}=O(n)$$

作业3-8

- 设计一个算法，将链表的链接完全颠倒。




newhead
head
head
head
NULL
a
b
c
NULL

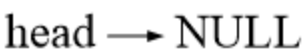
- 算法 Reverse (head. head)

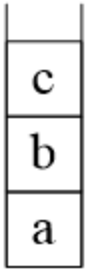
```
/*将指针 head 所指向的链表倒置*/
RV1[空链表]
    IF head = NULL THEN RETURN.
RV2[反转链表]
   newhead ← NULL.
   WHILE next(head) ≠ NULL DO (
      tmp ← head.
      head ← next(head). //移到head指向其后继节点
      next(tmp) ← newhead.  //反转节点指针
      newhead ← tmp.
   )
   next(head) ← newhead.
```

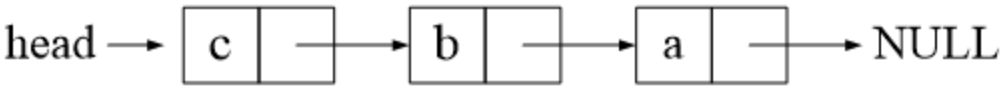
其它方法

- 使用堆栈将元素倒置

- 从表头删除，再插入表尾之后





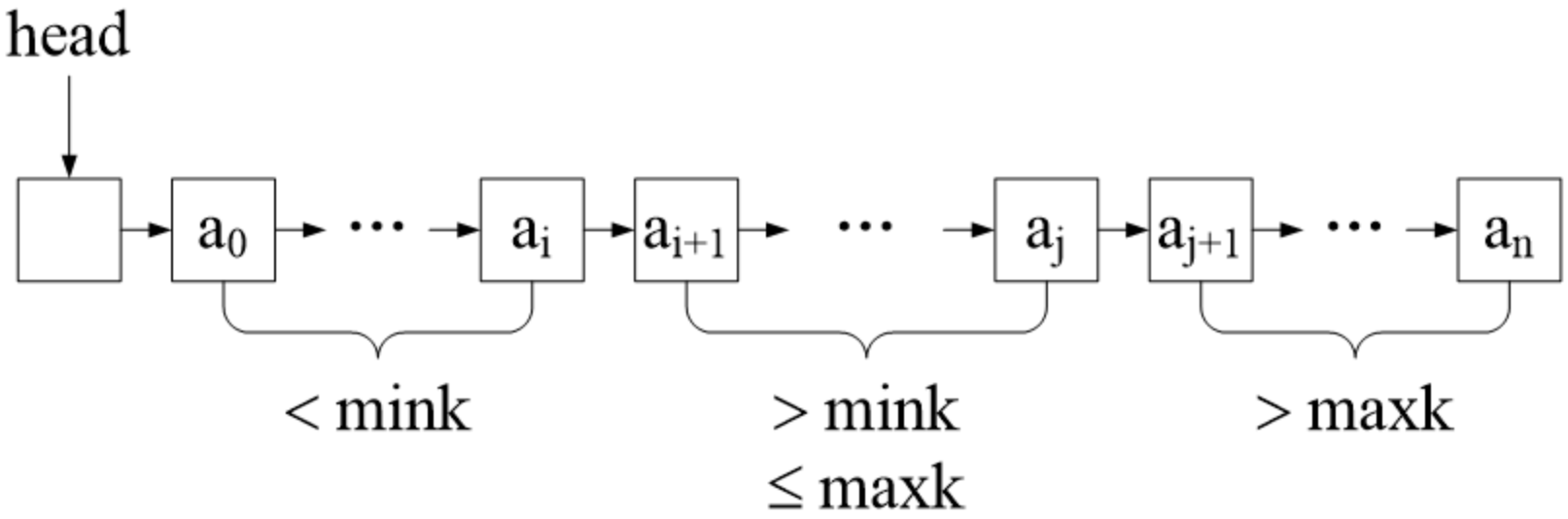


3-11

- 已知线性表中的元素以data值递增有序排列，并以单链表做存储结构。试写一高效算法，删除表中所有值大于mink且小于maxk的元素（若表中存在这样的元素），同时释放被删节点空间，并分析该算法的时间复杂度。(注意，mink和maxk是给定的两个参数，它们的值可以跟表中的元素相同，也可以不同)

分析

- 利用有序性减少元素比较次数。



- 时间复杂性T(n)=maxv结点的位置，记为O(n)

算法 Delete(head, mink, maxk. head)

/*删除单链表中值在mink和maxk之间的元素，单链表有哨兵变量*/

D1[特殊情况]

IF mink > maxk **THEN RETURN**.

D2 [定位小于mink的边界]

pre ←head.

WHILE next(pre) ≠ NULL AND data(next(pre)) < mink **DO** pre ← next(pre).

D2 [定位大于maxk的边界并删除]

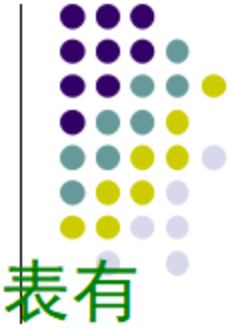
p ← next(p).

WHILE p ≠ NULL AND data(p) <= maxk **DO** (

AVAIL ⇐ p. p ← next(p).

)

next(pre) ← p.



算法 DeleteUnordered(head, mink, maxk. head)

/*删除单链表中值在mink和maxk之间的元素，单链表有哨兵变量*/

D1[特殊情况]

IF mink > maxk **THEN RETURN**.

D2 [一边比较一边删除]

pre ←head. p ←next(head).

WHILE p ≠ NULL **DO** (

IF data(p) < mink **THEN** (pre ←p. p←next(p).).

ELSE IF data(p) <= maxk **THEN** (

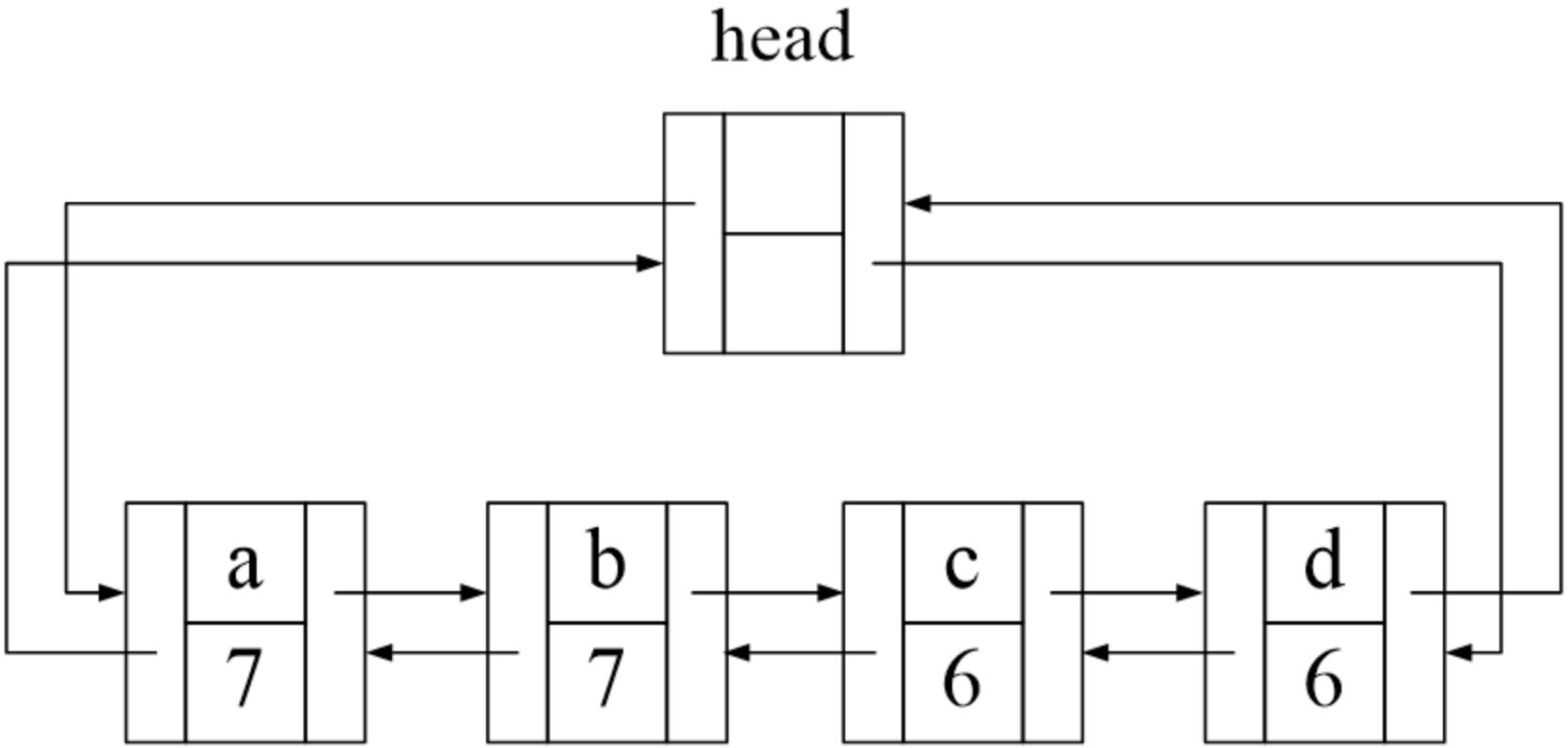
next(pre) ←next(p). AVAIL ⇐p. p ← next(pre).)

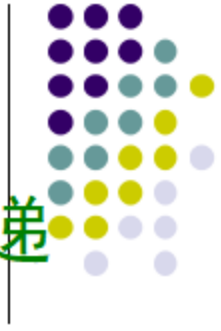
)

3-13

- 设有一个双向循环链表，每个结点中除有prior、data和next三个域外，还增设一个访问频度域freq。我们假定循环链表中无重复元素，且在链表被起用之前，频度域freq的值均初始化为零。而每当对链表进行一次Locate(head,x)的操作后，被访问的结点（即元素值等于x的结点）中的频度域freq的值增1，同时调整链表中结点之间的次序，使其按访问频度非递增的次序顺序排列，以便始终保持被频繁访问的结点总是靠近表头结点。试编写符合上述要求的Locate操作的方法。

例子分析





算法 Locate(head, x. head)

/*在双向循环链中定位值为x结点，修改freq并按递减序调整，删除所有碰到的x，有哨兵变量*/

Loc1[定位值为x的元素,修改freq]

```
p ←next(head).
WHILE p ≠ head(L) DO (
    IF data(p) = x THEN (p ←next(p). Break.)
    ELSE p ← next(p).
)
IF p ≠ head THEN RETURN.
freq(p) ← freq(p) + 1.
t ← next(p).
```

Loc2[找freq大于等于p的结点q]

q ←p

WHILE q ≠ head **AND** freq(p) > freq(prior(q) **DO**

q ←prior(q).

Loc3[从链表中删除p]

prior(next(p))←prior(p).

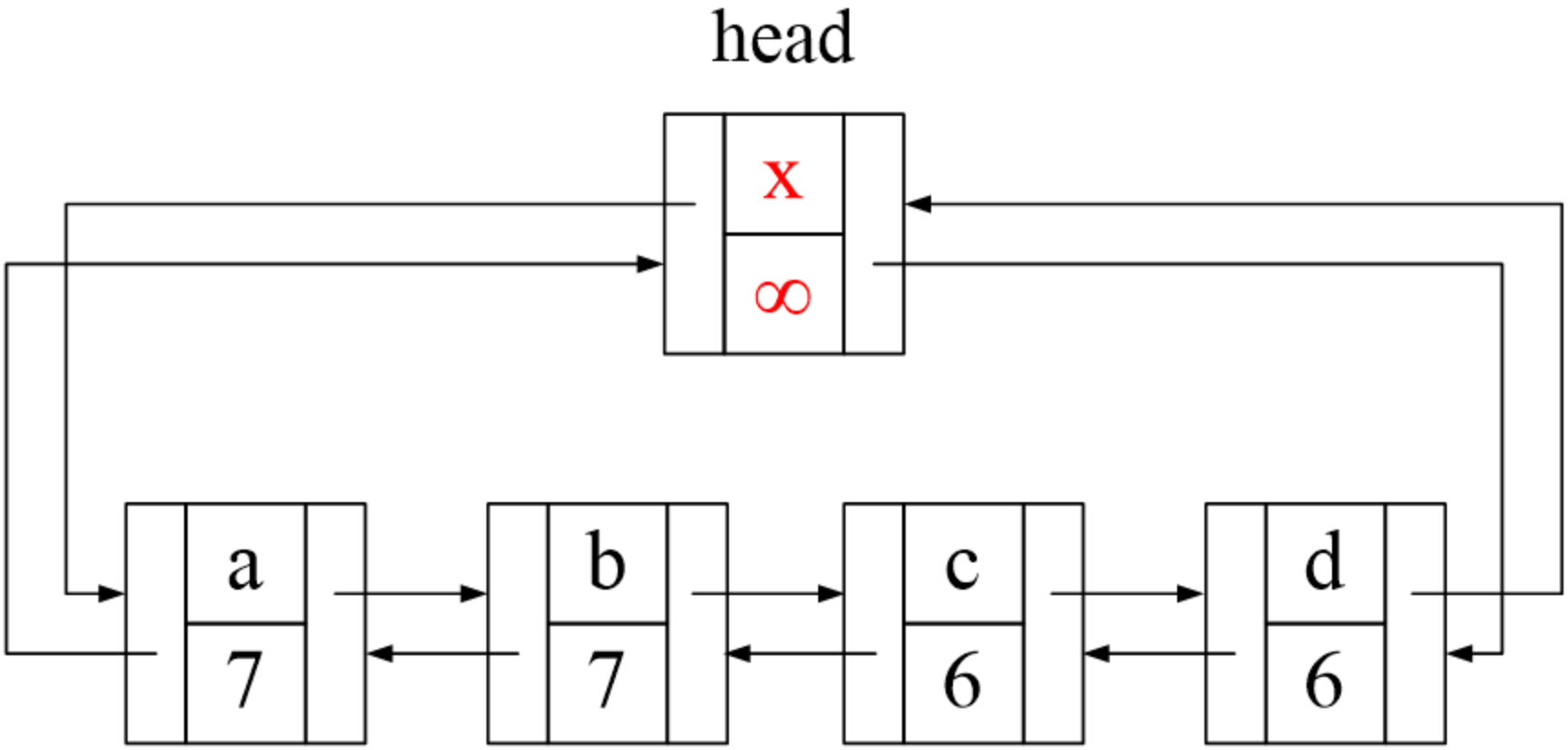
next(prior(p))=next(p).

Loc4[将p插入到q后]

prior(p) ←q. next(p) ←next(q).

prior(next(p)) ←p. next(prior(p)) ←p.

改进思想



算法 Locate-Improved(head, x. head)

/*在双向循环链中定位值为x结点，修改freq并按递减序调整，删除所有碰到的x，有哨兵变量*/

Loc0[修改哨位节点，减少条件判断]

data(head) ← x. freq(head) ← ∞.

Loc1[定位值为x的元素,修改freq]

p ←next(head).

WHILE data(p) = x **DO** p ←next(p).

IF p = head **THEN RETURN** //未找到

freq(p) ← freq(p) + 1.

t ← next(p).

Loc2[找freq大于等于p的结点q]

q ←p.

WHILE freq(p) > freq(prior(q)) **DO** q ←prior(q).

Loc3[从链表中删除p]

prior(next(p))←prior(p). next(prior(p))=next(p).

Loc4[将p插入到q后]

prior(p) ←q. next(p) ←next(q).

prior(next(p)) ←p. next(prior(p)) ←p.

3-14

- 请用图来说明对空栈L执行如下操作序列后堆栈的状态：

- Push(10), Push(4), Push(7), Pop, Push(15), Pop, Pop, Push(1)

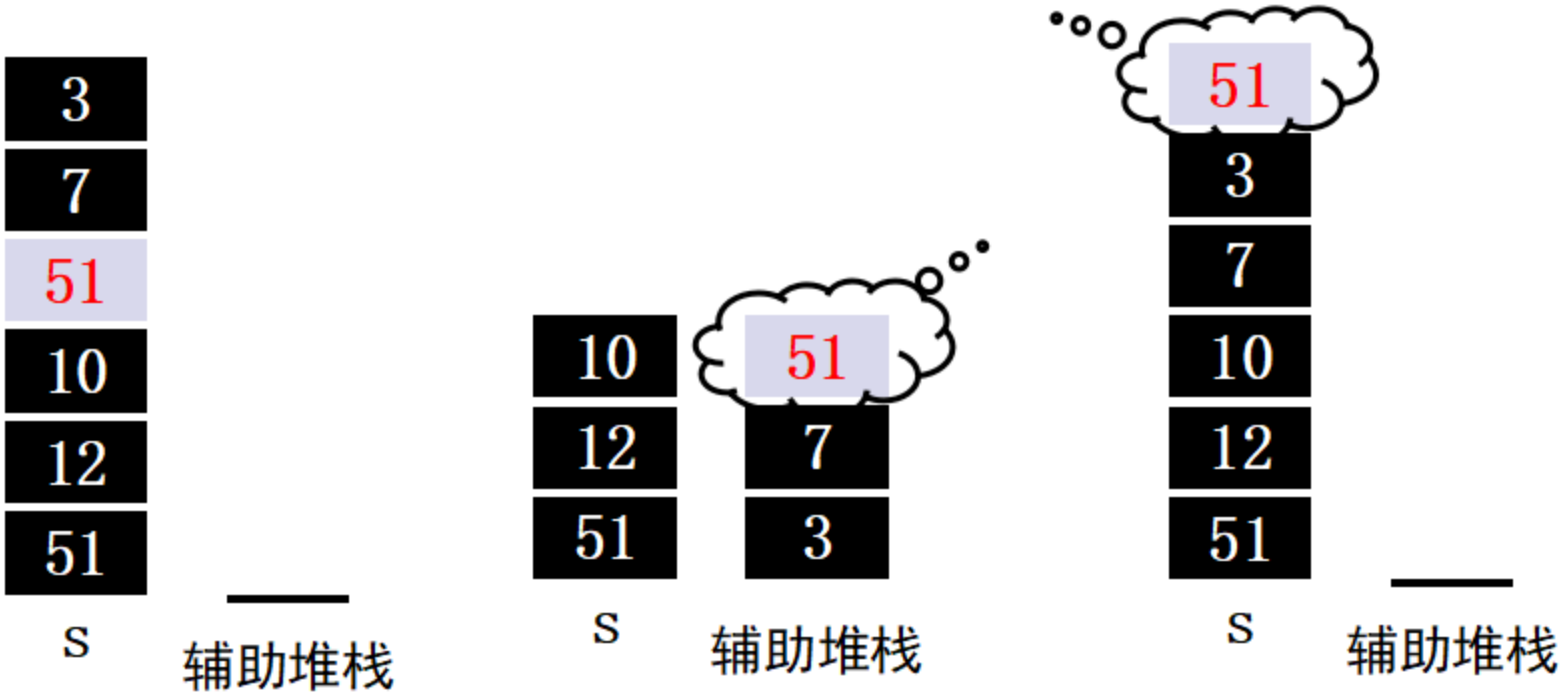
作业3-17

- 对于顺序堆栈和链式堆栈s，分别编写函数SelectItem(Stack & s, int n)，要求在堆栈中查找元素n在栈中第一次出现的位置，并将该位置元素移至栈顶，同时其他元素次序不变。（注意：用int匹配堆栈的模板）

分析

- 解题思路
  - 取栈顶元素，若不匹配，则对 s 进行弹栈操作
  - 找到（或无法找到）后恢复原来的元素次序
    - 关键在于记录弹出的顺序，后弹出的元素能够先被压回原来的栈 s，因此需要使用一个辅助堆栈

- 算法思想示例： n = 51



参考答案

```
int SelectItem(Stack<int> &s, int n)
{
#ifdef ARRAY_STACK
   AStack<int> temp(100);  //顺序堆栈
#else
   LStack<int> temp;  //链式堆栈
#endif
   bool flag = false;
   int loc = 0;
```

```
    while (! s.isEmpty() && s.Peek()!=n ) {
        temp.Push(s.Pop());
        loc++;
    }
    if(! s.isEmpty() ) { s.Pop(); flag = true;}
    while(!temp.isEmpty()) s.Push(temp.Pop());
    if (flag) s.Push(n)
    else loc = -1;
    return loc;
}
```

3-25

- 编写并实现双端队列类。双端队列（Deque）是可进行如下操作的线性表。
  (1) push(item)：将元素item插入到双端队列的前端
  (2) pop(item)：从双端队列删除前端元素并赋给item
  (3) inject(item)：将元素item插入到双端队列的尾端
  (4) eject(item)：从双端队列删除尾端元素并赋给item

参考答案

双端队列DQueue的类声明

```
template < class T >
class DQueue
{
private:
    SLNode <T> *front, *rear;  // 队首和队尾指针
    int size;  // 队列中元素个数
public:
    // 构造函数
    DQueue() { front=rear=NULL; size=0; }
    // 析构函数
    ~ DQueue() {  QClear(); }
```

```
// 将元素item插入到双端队列的前端
void push(const T& item)
{
    if (IsEmpty()) {
        front=rear=new SLNode<T>(item, NULL);
        size=1;
    } else {
        SLNode <T> *temp=front;
        front=new SLNode <T>(item, NULL);
        front→next=temp;
        size++;
    }
}
```

```
// 从双端队列中删除前端元素并将该值赋给变量item
 void pop(T& item)
 {
    if ( IsEmpty() ) cout << “Deleting from an empty queue!”<<endl;
    SLNode<T> *temp=front;
    item=temp→data;
    front=front→next;
    size - -;
    delete temp;
    if ( size==0 ) rear=NULL;
 }
```

```
// 将元素item插入到双端队列的尾端
void inject(T& item)
{
    if(IsEmpty()) {
        front=rear=new SLNode<T>(item, NULL);
        size=1;
    } else {
        rear→next=new SLNode <T>(item, NULL);
        rear=rear→next;
        size++;
    }
}
```

```
// 从双端队列中删除尾端元素并将该值赋给变量item
void eject(T& item)
{
    if ( IsEmpty() ) { cout << “Deleting from an empty queue!”<<endl; return; }
    item= rear→data; delete rear;
    if (--size==0)  front=rear=NULL;
    else {
        SLNode<T> *temp=front;
        while(temp→next!=rear) temp=temp→next;
        rear=temp;
        rear→next = NULL;
    }
}
```

```
    // 检测队列状态
    int QLength() const { return size; }
    int IsEmpty() const { return front==NULL; }
    // 清空队列
    void QClear()
    {
        while (!IsEmpty()) {
            rear=front;
            front=front→next;
            delete rear;
            size - -;
        }
        rear=NULL ;
    }
};
```

参考答案2（双向循环链版本）

```
template < class T >
class Deque
{
private:
    DLNode<T>* head ;
public:
     Deque() { head = new DLNode; head->left=head; head->right=head };
    ~Deque ()  { del() ; }
    bool push ( const T& item )
    bool pop ( T & item )  ;
    bool inject ( const T& item )
    bool eject( T & item )  ;
} ;
```

```
template < class T >
bool Deque<T>::push ( const T& item )
{
    p=new DLNode<T>(item,head,head->right);
   if(p==NULL) return false;
   left(right(p))=p;
   right(left(p))=p;
   return true;
}
```

```
template < class T >
bool Deque<T>::pop (T& item )
{
   if(right(head)==head) return false;
   p=right(head);
   item=p->data;
   left(right(p))=left(p);
   right(left(p))=right(p);
   return true;
}
```

```
template < class T >
bool Deque<T>:: inject( const T& item )
{
    p=new DLNode<T>(item,head->left,head);
   if(p==NULL) return false;
   left(right(p))=p;
   right(left(p))=p;
   return true;
}
```

```
template < class T >
bool Deque<T>::pop (T& item )
{
   if(left(head)==head) return false;
   p=left(head);
   item=p->data;
   left(right(p))=left(p);
   right(left(p))=right(p);
   return true;
}
```

THE END