数据结构与算法

线性表

张晓平

数学与统计学院

Email: xpzhang.math@whu.edu.cn

Homepage: http://staff.whu.edu.cn/show.jsp?n=Zhang%20Xiaoping

目录

1 2 线性表

2.1 线性表的基本概念

线性表是一种典型的线性结构，其中的数据元素是有序且有限，并且

- 有唯一首元；
- 有唯一末元；
- 除首元外，每个元素均有唯一的直接前驱；
- 除末元外，每个元素均有唯一的直接后继.

定义 (线性表 Link List)

由n个数据元素$a_1, a_2, \cdots, a_n$组成的有限序列($n \geq 0$, 数据元素又称结点).

- a_i的数据类型相同;
- n称为线性表的长度.

定义 (线性表 Link List)

由n个数据元素$a_1, a_2, \cdots, a_n$组成的有限序列($n \geq 0$, 数据元素又称结点).

- a_i的数据类型相同;
- n称为线性表的长度.

1. $n = 0$为空表;
2. $n > 0$为非空的线性表, 记为$(a_1, a_2, \cdots, a_n)$.
   - a_1称为首结点, a_n称为尾结点.
   - $a_1, a_2, \cdots, a_{i-1}$都是$a_i$的前驱, 其中$a_{i-1}$是$a_i$的直接前驱.
   - $a_{i+1}, a_{i+2}, \cdots, a_n$都是$a_i$的后继, 其中$a_{i+1}$是$a_i$的直接后继.

- 结点可以是单值元素

例

字母表：$(A, B, C, \cdots, Z)$

例

扑克点数：$(2, 3, 4, \cdots, J, Q, K, A)$

- 结点可以是记录型元素.

  每个元素可含多个数据项, 每一项称为结点的一个域. 每个元素有一个可以唯一标识每个结点的域, 称为关键字(key word).

例

Table: 某班2014级同学的基本情况

学号	姓名	性别	出生日期
20140212001	张强	男	06/24/1992
20140212002	王明	男	08/22/1992
...	...	...	...
20140212030	李娟	女	09/12/1992

- 若结点按值从小到大（或从大到小）排列, 称线性表是有序的.
- 线性表的长度可根据需要增长或缩短.
- 可对结点进行访问、插入和删除操作.

```
ADT List{
  Data:
    数据对象集合为(a1,a2,···,an)，各元素类型均为DataType. 其
    中，除首元素外，每个元素有且仅有一个直接前驱，除最后一个元素
    外，每个元素有且仅有一个直接后继. 数据元素之间为一对一的关
    系.
  Operation:
    InitList(&L):      初始化操作，建立一个空的线性表L.
    ListEmpty(L):      若线性表为空，返回true，否则返回false.
    ClearList(&L):     将线性表清空.
    GetElem(L,i,&e):   将线性表L中的第i个位置的元素返回给e.
    LocateElem(L,e):   在线性表L中查找与给定值e相等的元素
    ListInsert(&L,i,&e): 在线性表L中的第i个位置插入新元素e.
    ListDelete(&L,i,&e):
          删除线性表L中第i个位置的元素，并用e返回该值.
    ListLength(L):          返回线性表L的元素个数.
    ...
} ADT List
```

例

将两个线性表A和B合并，即把B中存在而A中不存在的数据元素插入到A中.

例

将两个线性表A和B合并，即把B中存在而A中不存在的数据元素插入到A中.

```
void union(List *La,List Lb){
  int La_len,Lb_len,i;
  ElemType e;
  La_Len=ListLength(La);
  Lb_Len=ListLength(Lb);
  for(i=1;i<=Lb_len;i++){
    GetElem(Lb,i,e);
    if(!LocateElem(La,e))
      ListInsert(La,++La_len,e);
  }
}
```

定义 (顺序存储 Sequence List)

把结点按逻辑顺序依次存放在一组地址连续的存储单元里. 以这种方式存储的线性表简称顺序表.

2.2 线性表的顺序存储

定义 (顺序存储 Sequence List)

把结点按逻辑顺序依次存放在一组地址连续的存储单元里. 以这种方式存储的线性表简称顺序表.

特点

- 逻辑顺序与物理顺序一致;
- 数据元素之间的关系是以元素在计算机内“物理位置相邻”来体现的.

2.2 线性表的顺序存储

设有非空线性表$(a_1, a_2, \cdots, a_n)$, l_i表示a_i的存储位置, k为每个元素需占用的存储单元.

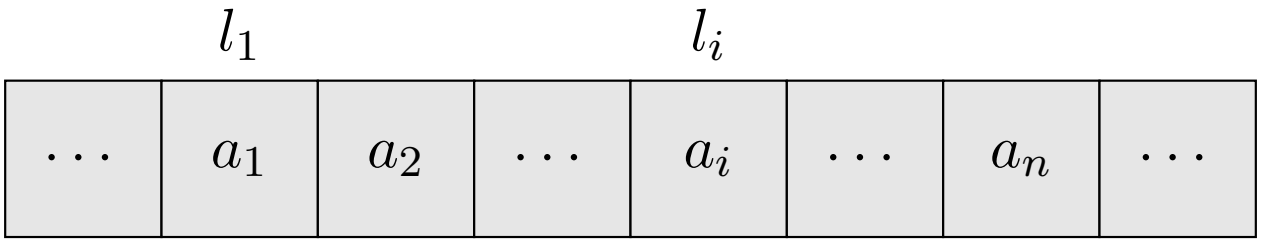


图: 线性表的顺序存储

$$l_{i+1} = l_i + k$$

$$l_i = l_1 + (i-1)k$$

顺序存储的结构代码

```
#define MAX_SIZE 100
typedef int ElemType;
typedef struct sqlist {
    ElemType data[MAX_SIZE];
    int length;
}
```

注 (数据长度与线性表长度的区别)

- 数组长度是存放线性表存储空间的长度，存储分配后一般不变；
- 线性表长度是线性表中数据元素的个数，随着线性表插入和删除操作的进行而发生变化；
- 线性表长度总小于等于数组长度。

- 初始化
- 赋值
- 查找
- 修改
- 插入
- 删除
- 求长度
- ...

```c
#define OK 1
#define ERROR 0
#define TRUE 1
typedef int Status;
```

```
Status SqListInit(SqList *L) {
  L->data=(ElemType *) malloc(MAX_SIZE*sizeof(ElemType
  ));
  if(!L->data) return ERROR;
  L->length=0;
  return OK;
}
```

目标

在$L=(a_1,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$中的第$i$个位置上插入新结点$e$, 使其成为

$$L=(a_1,\cdots,a_{i-1},e,a_i,a_{i+1},\cdots,a_n)$$

目标

在$L=(a_1,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$中的第$i$个位置上插入新结点$e$, 使其成为

$$L=(a_1,\cdots,a_{i-1},e,a_i,a_{i+1},\cdots,a_n)$$

实现步骤

- 如果插入位置不合理，抛出异常；
- 如果线性表长度大于等于数组长度，抛出异常或动态增加容量；
- 从最后一个元素开始向前遍历到第i个位置，分别将它们向后移动一个位置；
- 将元素e填入位置i处；
- 线性表长度加1.

```
Status SqListInsert(SqList *L,int i,ElemType e){
  int j;
  if (L->length>=MAX_SIZE) { //线性表已满
    printf("线性表溢出! \n"); return ERROR;
  }
  if (i<0||i>L->length-1)    //i不在范围内
    return ERROR;
  for (j=L->length-1; j>=i-1; j--)
    L->data[j+1]=L->data[j];
  L->data[i-1]=e;
  L->length++;
  return OK;
}
```

在L的第i个元素之前插入新结点, 其时间主要耗费在结点的移动上. 因此, 可用结点的移动来估计算法的时间复杂度.

设在L的第i个元素之前插入结点的概率为p_i. 不失一般性, 设各位置插入等概率, 则$p_i = \frac{1}{n+1}$, 而插入时移动结点的次数为$n-i+1$, 故总的平均移动次数为

$$E_{insert} = \sum_{i=1}^{n} p_i(n-i+1) = \frac{n}{2}.$$

这表明, 在顺序表上做插入运算, 平均要移动表上一半的结点. 当表长n较大时, 算法效率相当低. 因此算法的平均时间复杂度为$O(n)$.

目标

在

$$L = (a_1, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$$

中删除结点a_i, 使其成为

$$L = (a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n)$$

目标

在

$$L = (a_1, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$$

中删除结点a_i, 使其成为

$$L = (a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n)$$

实现步骤

- 若删除位置不合理，跑出异常；
- 取出删除元素；
- 从删除元素的位置开始遍历到最后一个元素的位置，分别将它们向前移动一个位置；
- 表长减1.

```
Status SqListDelete(SqList *L,int i,ElemType *e) {
  int j;
  if (L->length==0) {
    printf("线性表为空！\n"); return ERROR;
  }
  if (i<0||i>L->length)  {
    printf("要删除的数据元素不存在！\n");
    return ERROR;
  }
  else {
    *e=L->data[i-1];
    for (j=i; j<L->length; j++)
      L->data[j-1]=L->data[j];
    L->length--;
    return OK;
  }
}
```

删除L的第i个元素, 其时间主要耗费在表中结点的移动操作上. 因此, 可用结点的移动来估计算法的时间复杂度.

设在L中删除第i个元素的概率为p_i, 不失一般性, 设各个位置插入等概率, 则$p_i = \frac{1}{n}$, 而插入时移动结点的次数为$n - i$, 故总的平均移动次数为

$$E_{delete} = \sum_{i=1}^{n} p_i(n-i) = \frac{n-1}{2}.$$

这表明, 在顺序表上做删除运算, 平均要移动表上一半的结点. 当表长n较大时, 算法效率相当低. 因此算法的平均时间复杂度为$O(n)$.

目标

在$L=(a_1,a_2,\cdots,a_n)$中删除值为x的第一个结点.

目标

在$L=(a_1,a_2,\cdots,a_n)$中删除值为x的第一个结点.

实现步骤

(1) 在L中查找值为x的第一个数据元素；

(2) 将从找到的位置至最后一个结点依次向前移动一个位置；

(3) 线性表长度减1.

```
Status SqListLocateDelete(SqList *L,ElemType x) {
  int i=0,k;
  while (i<L->length){
    if (L->data[i]!=x) i++;
    else {
      for (k=i+1;k<L->length;k++)
        L->data[k-1]=L->data[k];
      L->length--; break;
    }
  }
  if (i>L->length) {
    printf("要删除的数据元素不存在！\n");
    return ERROR;
  }
  return OK;
}
```

时间主要耗费在数据元素的比较和移动操作上.

设在L中删除数据元素的概率为p_i, 不失一般性, 设各个位置等概率, 则$p_i = \frac{1}{n}$.

- 比较的平均次数为：

$$E_{compare} = \sum_{i=1}^{n} p_i i = \frac{n+1}{2}$$

- 删除时的平均移动次数为

$$E_{delete} = \sum_{i=1}^{n} p_i (n-i) = \frac{n-1}{2}.$$

平均时间复杂度为

$$E_{compare} + E_{delete} = n$$

优点

- 无须为表示表中元素之间的逻辑关系而增加额外的存储空间
- 可以快速地存取表中任一位置的元素

优点

- 无须为表示表中元素之间的逻辑关系而增加额外的存储空间
- 可以快速地存取表中任一位置的元素

缺点

- 插入和删除操作需要移动大量元素
- 当线性表变化较大时，难以确定存储空间的容量
- 造成存储空间的“碎片”

1. 2 线性表
   - 2.1 线性表的基本概念
   - 2.2 线性表的顺序存储
   - 2.3 线性表的链式存储

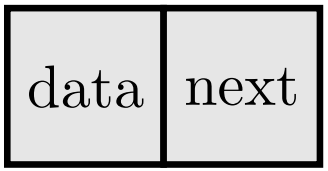
定义 (链式存储)

用一组任意的存储单元存储线性表中的数据元素。 用这种方法存储的线性表简称线性链表。

定义 (链式存储)

用一组任意的存储单元存储线性表中的数据元素。 用这种方法存储的线性表简称线性链表。

- 存储链表中结点的存储单元可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。
- 链表中的逻辑顺序和物理顺序不一定相同。



data: 数据域，存放结点的值

next: 指针域，存放结点的直接后继的地址

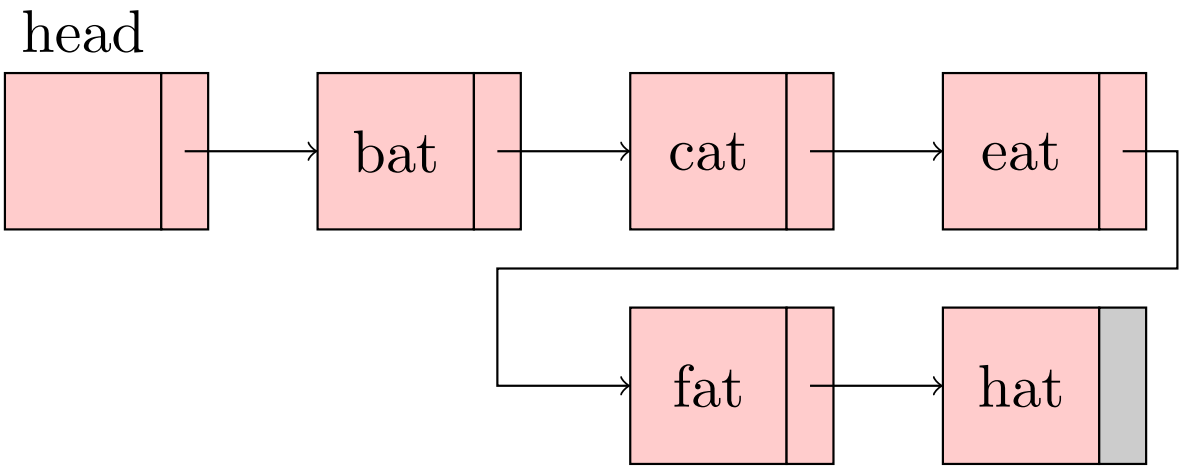
- 链表是通过每个结点的指针域将线性表的n个结点按其逻辑次序链接在一起的。
- 每一个结点只包含一个指针域的链表，称为单链表。
- 为操作方便，总在链表的第一个结点之前附设一个头结点（头指针）head指向第一个结点。头结点的数据域可以不存储任何信息（或链表长度等信息）。
- 单链表由表头唯一确定，因此单链表可用头指针的名字来命名。

2.3 线性表的链式存储

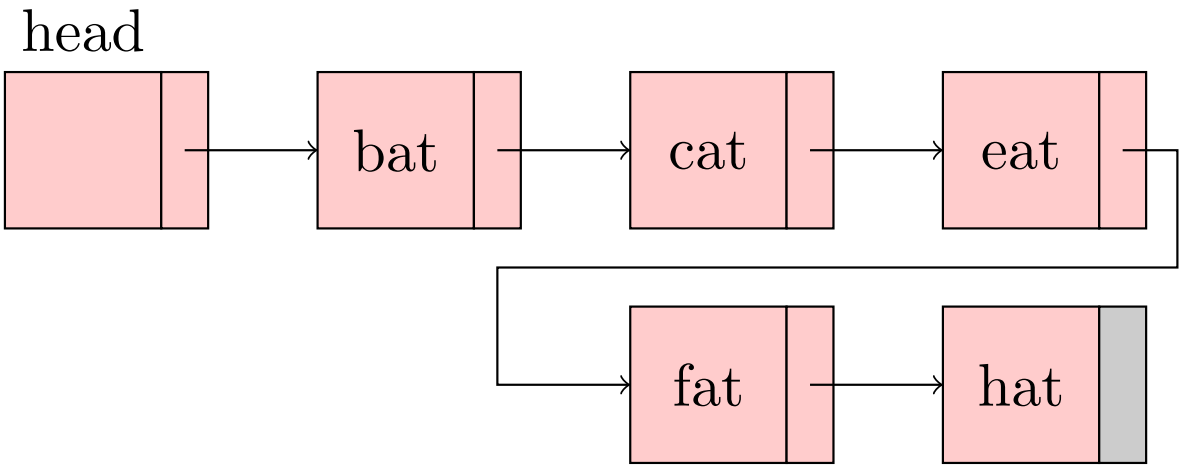
例1：线性表$L = (bat, cat, eat, fat, hat)$

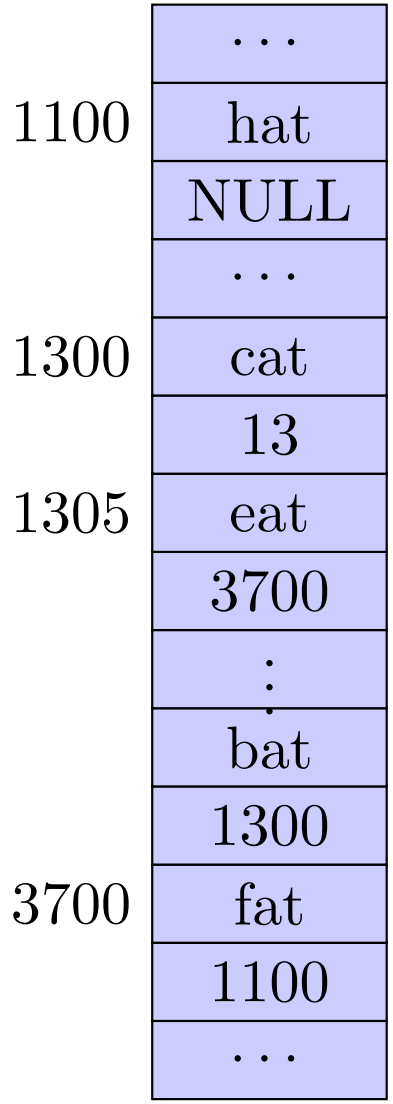
2.3 线性表的链式存储

例1：线性表$L = (bat, cat, eat, fat, hat)$



例1：线性表$L = (bat, cat, eat, fat, hat)$





结点的描述与实现

```
typedef struct LNode{
   ElemType data;
   struct LNode *next;
} LNode;
typedef struct LNode *LinkList;
```

结点的实现

结点是通过动态分配和释放来实现的，即需要时分配，不需要时释放。

```
malloc(), realloc(), sizeof(), free();
```

结点的实现

结点是通过动态分配和释放来实现的，即需要时分配，不需要时释放。

```
malloc(), realloc(), sizeof(), free();
```

- 动态分配

```
p=(LNode*)malloc(sizeof(LNode));
```

分配了一个类型为LNode的结点变量的空间，并将其首地址放入指针变量p中。

- 动态释放

```
free(p);
```

系统回收由指针变量p指向的内存区。

(2) 常见的指针操作

图: q=p

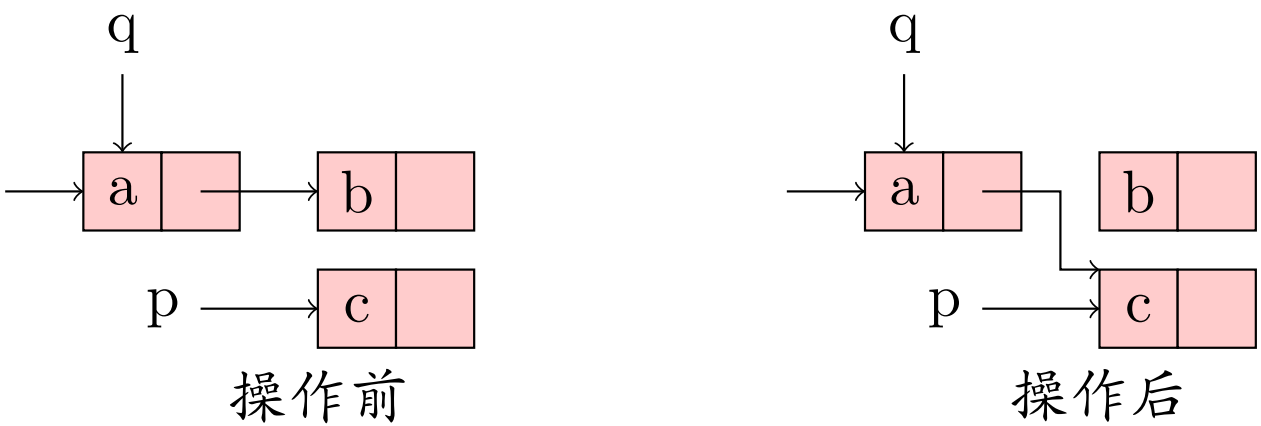
(2) 常见的指针操作

图：q=p

图：q=p->next

图: p=p->next;

图：p=p->next;



图：q->next=p;

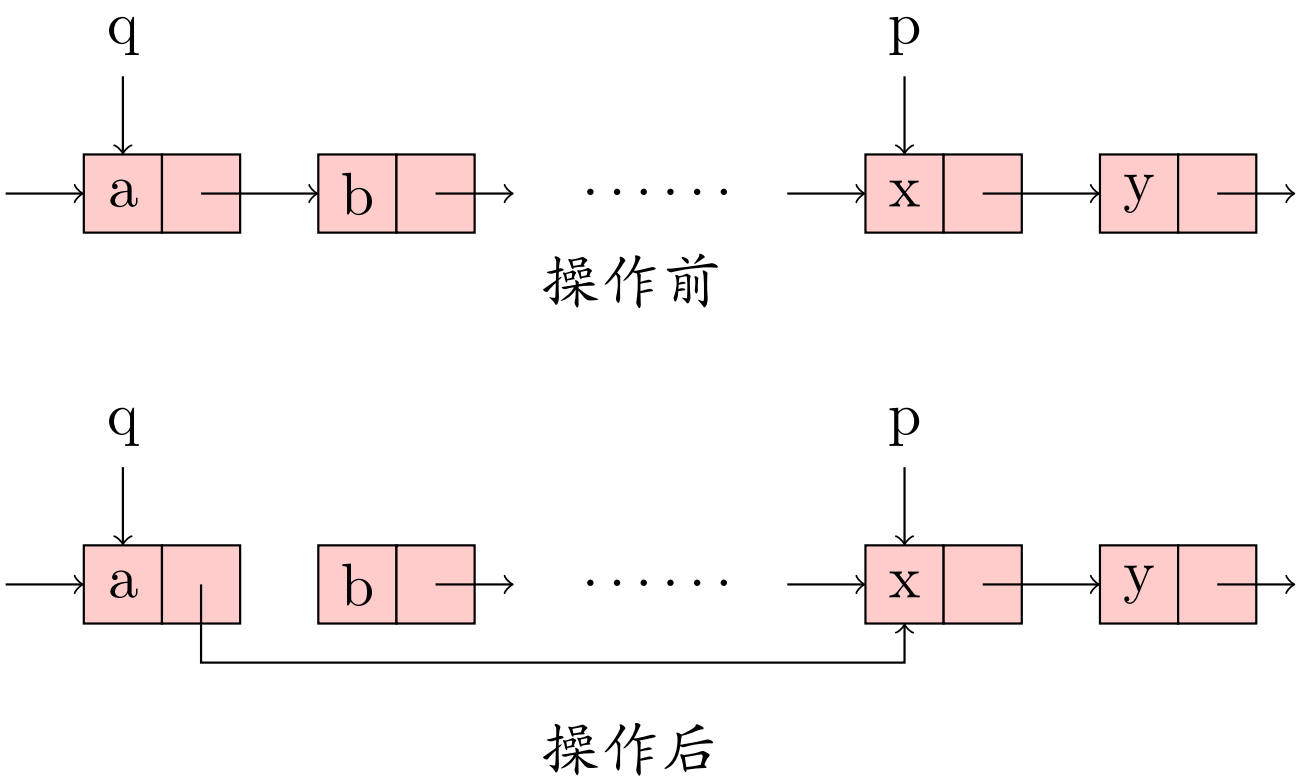
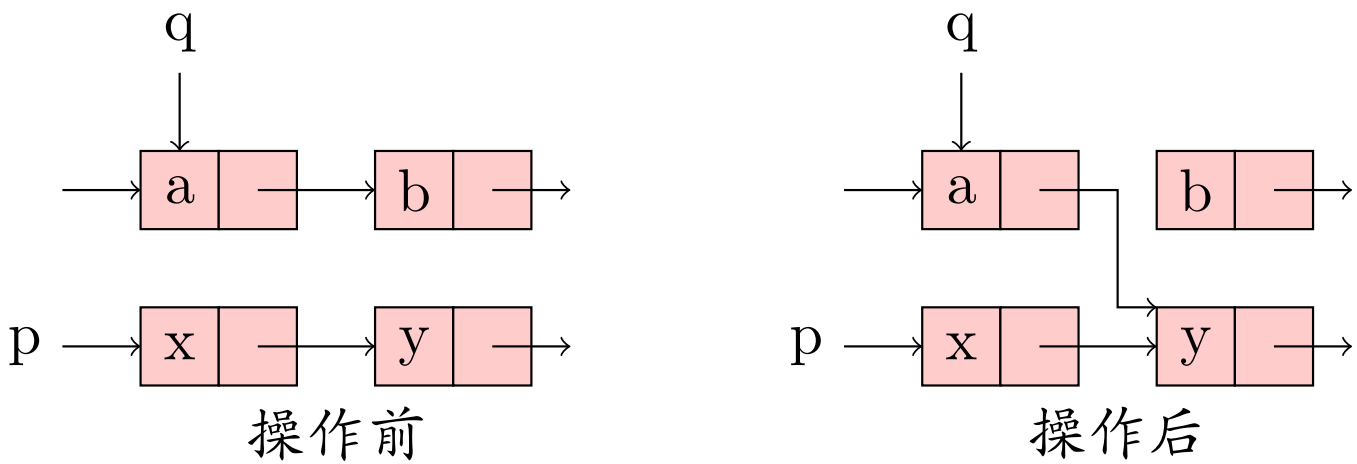


图: q->next=p;



图：q->next=p->next;

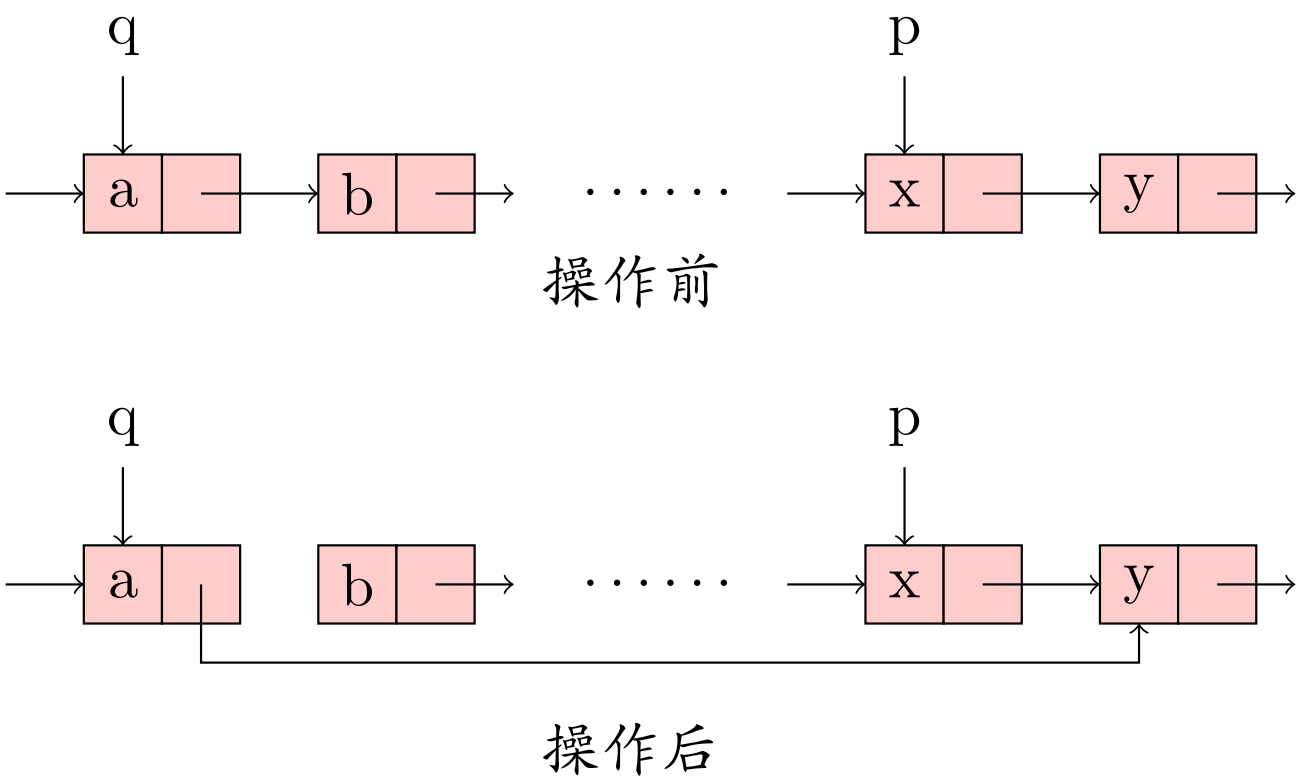


图: q->next=p->next;

- 顺序存储结构的创建，就是一个数组的初始化。而单链表则不同，它可以很散，是一个动态结构。
- 对每个链表而言，它所占用空间的大小和位置不需要预先分配，可根据系统的情况和实际需求即时生成。

所以创建单链表的过程就是一个动态生成链表的过程，即从空表的初始状态起，一次建立各元素结点，并逐个插入链表。

动态地建立单链表的常用方法有两种：

- 头插入法
- 尾插入法

- 声明一结点p和计数器变量i
- 初始化一空链表L
- 让L的头结点的指针指向NULL，即建立一个带头结点的单链表
- 循环
  - 生成一个新结点赋值给p
  - 随机生成一个数赋给p的数据域p->data
  - 将p插入到头结点与前一新结点之间

```
void CreateLinkListHead(LinkList L, int n){
  LinkList p;
  int i;
  L->next=NULL;
  for(i=0;i<n;i++){
    p=(LinkList) malloc(sizeof(LNode));
    p->data=rand()%100+1;
    p->next=L->next;
    L->next=p;
  }
}
```

头插入法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。该方法是将新结点插入到当前链表的表尾，使其成为当前链表的尾结点。

```
void CreateLinkListTail(LinkList L, int n){
  LinkList p,r;
  int i;
  r=L;
  for(i=0;i<n;i++){
    p=(LinkList) malloc(sizeof(LNode));
    p->data=rand()%100+1;
    r->next=p;
    r=p;
  }
  r->next=NULL;
}
```

1. 按序号查找
2. 按值查找

对于单链表，不能像顺序表中那样直接按序号i访问结点，而只能从链表的头结点出发，沿指针域next逐个结点往下搜索，知道搜到第i个结点为止。因此，链表不是随机存储结构。

设单链表长度为n，要查找第i个结点，仅当$1 \leq i \leq n$时，i的值是合法的。

```
Status GetElem(LinkList L,int i,ElemType *e){
  int j;
  LinkList p;
  p=L->next;          /* Point to the first node */
  j=1;
  while(p!=NULL&&j<i){
    p=p->next; j++;
  }
  if(!p||j>i)
    return ERROR;     /* the i-th element DONOT exist */
  *e=p->data;         /* get data of the i-th element */
  return OK;
}
```

移动指针p的频度

$$\begin{cases} 0\text{次}, & i < 1; \\ i-1\text{次}, & i \in [1,n]; \\ n\text{次}, & i > n. \end{cases} \quad \Rightarrow \quad \text{时间复杂度为} O(n).$$

按值查找是在链表中，查找是否有结点值等于给定值key的结点?

- 若有，则返回首次找到的值为key的结点的存储位置；
- 否则返回NULL。

查找时从开始结点出发，沿链表逐个将结点的值和给定值key作比较。

```
LinkList LocateNodeKey(LinkList L,ElemType key){
  LinkList p=L->next;

  while(p!=NULL&&p->data!=key) p=p->next;
  if(p->data==key) return p;
  else{
    printf("The node you find DONOT exist!\n");
    return NULL;
  }
}
```

平均时间复杂度

算法的执行与形参key有关，平均时间复杂度为$O(n)$。

插入运算是指将值为e的新结点插入到表的第i个结点的位置上，即插入到a_{i-1}与a_i之间。因此，必须首先找到a_{i-1}所在的结点p，然后生成一个数据域为e的新结点q，q作为p的直接后继。

```
void InsertLNode(LinkList L,int i,ElemType e){
  int j=0; LinkList p,q;
  p=L->next;                    /* Point to the first node */
  while(p!=NULL&&j<i-1){
    p=p->next; j++;
  }
  if(j!=i-1) printf("i too big or equal 0!\n");
  else {
    q=(LinkList) malloc(sizeof(LNode));
    q->data=e; q->next=p->next;
    p->next=q;
  }
}
```

平均时间复杂度

设链表长度为n，合法的插入位置是$1 \leq i \leq n$。算法的时间主要耗费在移动指针p上，平均时间复杂度为$O(n)$。

- 按序号删除：删除单链表中的第i个结点。
- 按值删除：删除单链表中值为key的第一个结点。

- 为了删除第i个结点a_i，必须找到结点的存储地址。
- 该存储地址在其直接前驱结点a_{i-1}的next域中，因此必须首先找到a_{i-1}的存储位置p，然后令p->next指向a_i的直接后继结点，即把a_i从链上摘下来。
- 最后释放结点a_i的空间，将其归还给“存储池”。

设单链表长度为n，则删去第i个结点仅当$1 \leq i \leq n$时是合法的。
当$i = n + 1$时，虽然被删结点不存在，但其前驱结点却存在，是终端结点。故判断条件之一是p->next!=NULL。

```
void DeleteLNodeIndex(LinkList L,int i){
  int j=1; LinkList p,q;
  p=L; q=L->next;
  while(p->next!=NULL&&j<i){
    p=q; q=q->next; j++;
  }
  if(j!=i) printf("i too big or equal 0!\n");
  else {
    p->next=q->next;
    free(q);
  }
}
```

时间复杂度

时间复杂度为$O(n)$。

与按值查找相类似，首先要查找值为key的结点是否存在？

- 若存在，则删除；
- 否则返回NULL。

```
void DeleteLNodeKey(LinkList L,ElemType key){
  LinkList p=L,q=L->next;
  while(q!=NULL&&q->data!=key){
    p=q; q=q->next;
  }
  if(q==NULL) {
    printf("The Node you delete DONOT exist!\n");
    return;
  }
  if(q->data==key){
    p->next=q->next; free(q);
  }
  else {
    printf("The Node you delete DONOT exist!\n");
  }
}
```

删除结点

按值删除

时间复杂度

时间复杂度为$O(n)$。

链表实现插入和删除运算，无需移动结点，仅需修改指针，解决了顺序表的插入或删除操作需要移动大量元素的问题。

变形之一

删除单链表中值为key的所有结点。

变形之一

删除单链表中值为key的所有结点。

基本思想

- 从单链表的第一个结点开始，对每个结点进行检查，若结点的值为key，则删除之，
- 然后检查下一个结点，直到所有的结点都检查。

```
void Delete_LinkList_Node(LinkList L, int key) {
  LinkList p=L, q=L->next;
  while (q!=NULL){
    if (q->data!=key){
      p->next=q->next; free(q); q=p->next;
    } else {
      p=q; q=q->next;
    }
  }
}
```

变形之二

删除单链表中所有值重复的结点，使得所有结点的值都不相同。

变形之二

删除单链表中所有值重复的结点，使得所有结点的值都不相同。

基本思想

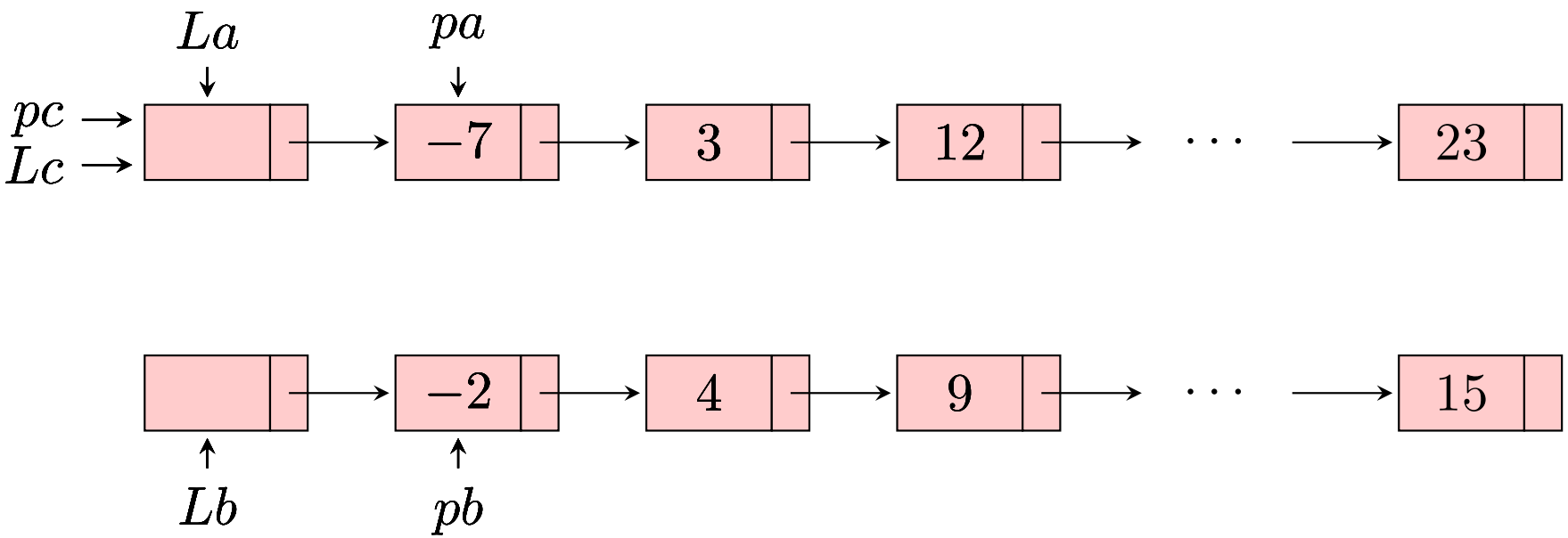
从单链表的第一个结点开始，对每个结点进行检查：

- 检查链表中该结点的所有后继结点，只要有值和该结点的值相同，则删除之；
- 然后检查下一个结点，直到所有的结点都检查。

```
void Delete_LinkList_Value(LNode *L) {
  LinkList p=L->next, q, ptr;
  while (p!=NULL){
    *q=p; *ptr=p->next;
    while(ptr!=NULL){
      if (ptr->data==p->data){
        q->next=ptr->next; free(ptr); ptr=q->next;
      } else {
        q=ptr; ptr=ptr->next;
      }
    }
    p=p->next;
  }
}
```

单链表的合并

设有两个有序的单链表，它们的头指针分别为La、Lb，将它们合并为以Lc为头指针的有序链表。



图：合并前的示意图

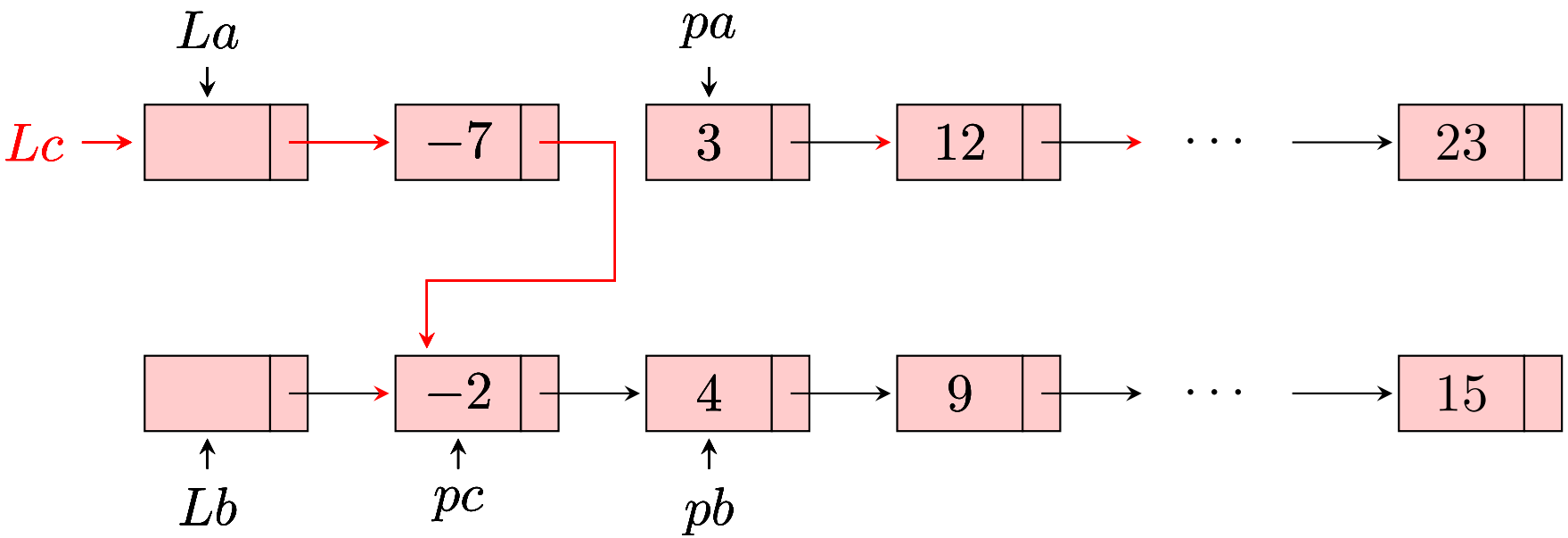


图: 合并了值为$-7,-2$的结点后的示意图

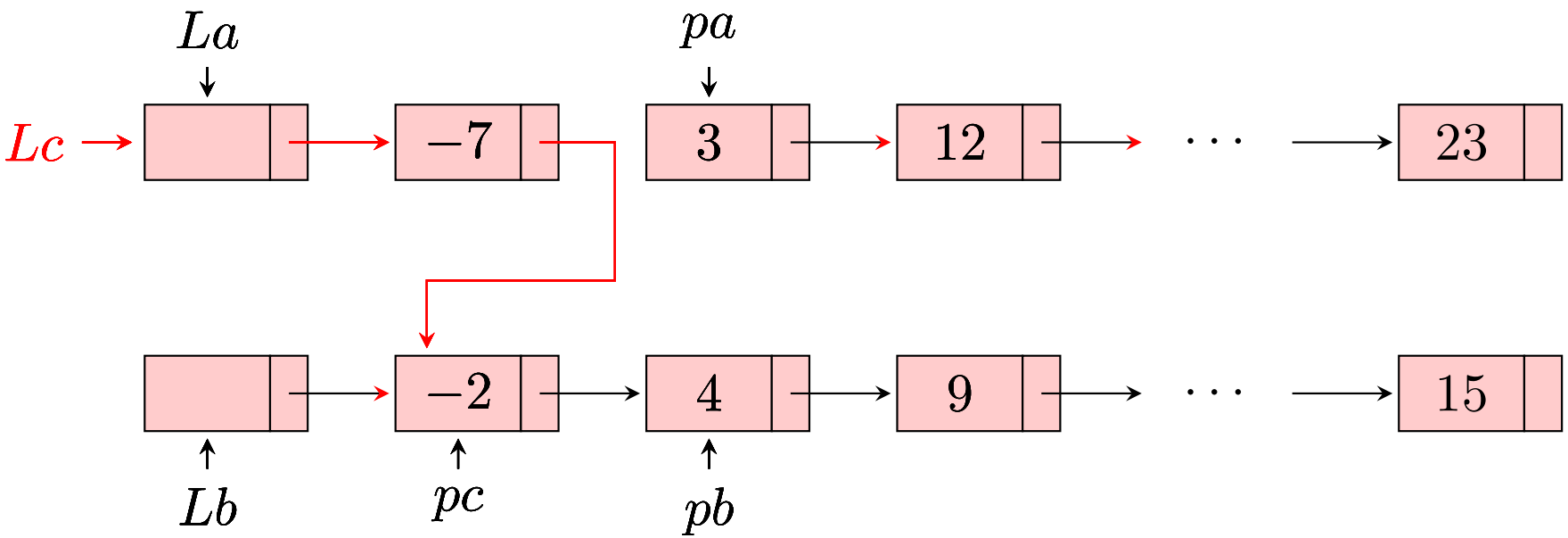


图: 合并了值为$-7,-2$的结点后的示意图

- pa, pb分别是待考察的两个链表的当前结点;
- pc是合并过程中合并的链表的最后一个结点。

```
LinkList Merge_Linklist(LinkList La, LinkList Lb){
  LinkList Lc,pa,pb,pc,ptr;
  Lc=La; pc=La; pa=La->next; pb=Lb->next;
  while(pa!=NULL && pb!=NULL){
    if (pa->data<pb->data){
      pc->next=pa; pc=pa; pa=pa->next; }
    if (pa->data<pb->data){
      pc->next=pb; pc=pb; pb=pb->next; }
    if (pa->data==pb->data){
      pc->next=pa; pc=pa; pa=pa->next;
      ptr=pb; pb=pb->next; free(ptr); }
  }
  if(pa!=NULL) pc->next=pa;
  else pc->next=pb;
  free(Lb);
  return(Lc);
}
```

时间复杂度

若La, Lb两个链表的长度分别为m, n，则链表合并的时间复杂度为$O(m+n)$。