实验三　链表实验

【实验目的】

（1）掌握链表的存储结构及应用。

（2）掌握用链表表示数据、并进行有关算法设计的方法。

【实验背景】

链接存储的线性表被称为链表。根据链表中结点之间的链接方式不同，链表又包括单链表、单循环链表及双循环链表。

链表的基本运算包括置空表 LLsetnull (L)、求表长 LLlength (L)、按序号取元素 LLget (L，i)、按值查找(定位)LLlocate (L，x)、插入 LLinsert (L，i，x)和删除 LLdelete (L，i)等。下面描述的是单链表的基本运算实现。

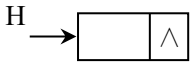
1、 单链表置空表算法

设单链表的头指针为 L

LinkList *L；

当单链表为空表时，表中没有元素，仅有头节点。即：L->next = NULL　(空)

```
#define NULL 0
LinkList *setnull (LinkList *L){
    L->next = NULL；
    return ( L)；
}
```

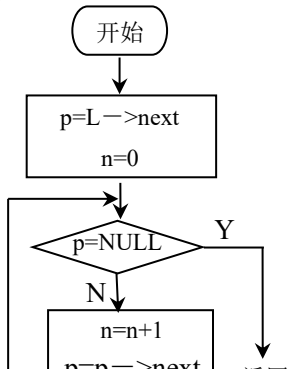


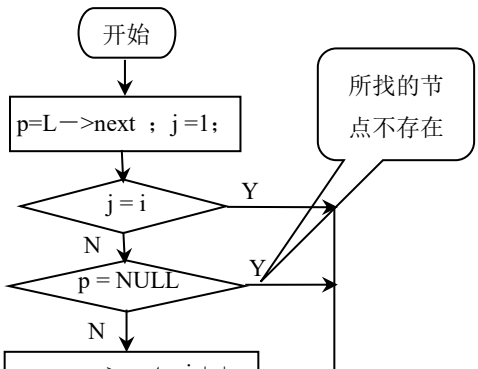
2、 单链表求表长算法

求链表的长度就是求链表中不包括头节点在内的节点个数。与顺序表不同，链表中节点间的线性关系由节点的 next 域中的地址来维系。要确认表中有哪些节点、有多少节点，必须从头节点开始“顺藤摸瓜”，顺着各节点 next 域中指示的地址去寻找下一个节点，直到某节点的 next 域为空为止。

算法：可以定义一个变量 n 记录节点的个数，开始时 n = 0；令 p 等于首元素节点的地址，即 p = L->next，这里 L 为头指针。当 p 不为空时，n 加 1；继续令 p = p->next，并且 p 不为空时，n 加 1，……，如此循环往复，直到 p 为空，这时 n 的值即为表长。图 5.1 为该算法的流程图。

```
int LLlength(LinkList *L){   // 求带头节点的单链表 L 的长度
    LinkList *p；int n=0；// 用来存放单链表的长度
    p=L->next；
    while(p!=NULL){
        p=p->next；n++；
    }
    return   n；
}
```





3、 单链表按序号取元素算法

该运算是求链表中指定序号为 i 的节点的地址。节点的序号从首元素节点开始直到尾节点，依次为 1、2、3、……、m，即尾节点的序号也就是表长 m。这样，要找到第 i 个节点，可以采用与求链表的表长同样的方法，从头节点开始“顺藤摸瓜”。

算法：令变量 n 记录已搜索过的节点的个数；若令 p 等于首元素节点的地址，当 p 不为空时 n＝1；继续令 p＝p->next，且 p 不为空时，n＝n＋1，……，直到 n＝i，则 p 的值即为序号为 i 的节点的地址。若在搜索的过程中 p 为空，则表示链表没有序号为 i 的节点，也就是说 i 的值大于表长 m。图 5.2 为该算法的流程图。

```
LinkList *Get (LinkList *L，int i){
// 在带头节点的单链表 L 中查找第 i 个节点，
// 若找到(1≤i≤n)，则返回该节点的存储位置；否则返回 NULL
    int j；LinkList *p；
    p=L->next；j=1；          // 从首元素节点开始扫描
    while (p! =NULL && j<i){
        p=p->next；    // 扫描下一节点
        j++；       // 已扫描节点计数器
    }
    if(i= = j) return p；          // 找到了第 i 个节点
    else     return NULL；    // 找不到，i≤0 或 i>n
}
```

4、 单链表按值查找算法

从单链表的头指针指向的头节点出发(或者从首元素节点出发)，顺着链逐个将节点的值和给定值 x 作比较。若有节点值等于 x 的节点，则返回首次找到的其值为 x 的节点的存储位置，否则返回 NULL。

```
LinkList *LLlocate(LinkList *L，DataType x){
// 在带头节点的单链表 L 中查找其节点值等于 X 的节点，
// 若找到则返回该节点的位置 p；否则返回 NULL
    LinkList *p；
    p=L->next；                    // 从表中第一个节点比较
    while (p!=NULL)
        if (p->data!=x)    p=p->next；
        else break；                // 找到值为 x 的节点，退出循环
    return p；
}
```

该算法与按序号取元素 LLget(L，i)算法一样，while 语句执行次数最多。最坏的情况下，

while 语句执行 n 次;若查找各位置上元素的概率相等,则 while 语句平均执行次数为(n+1)/2。这样，该算法的平均时间复杂度也为$O(n)$。

5、 单链表插入算法

在单链表 L 的第 i 个位置插入值为 x 的节点，首先要动态生成一个数据域为 x 的节点 S，然后插入在单链表中。

为保证插入后节点间的线性逻辑关系，需要修改第 i-1 个节点 a_{i-1} 的 next 域中的值，使其为新节点的地址；同时，对于新节点来说，插入后，其直接后继节点应为原链表中的第 i 个节点 a_i，则应将节点 a_i 的地址存放在新节点的 next 域中。如图 5.3 所示。

上述的操作过程可以描述为：

① S =(LinkList *) malloc(sizeof(LinkList))；

② S->data = x；

③ S->next = P->next；

④ P->next = S；

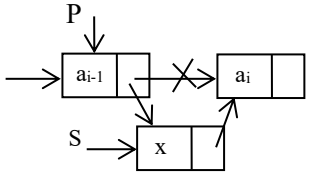


图 5.3 插入算法示意图

关于第 i-1 个节点 a_{i-1} 地址 P 的获得，可采用与按序号取元素算法相同的算法。所不同的是，由于插入操作可以在第一个位置进行(在首元素节点前插入一个节点)，所以本算法中搜索节点地址的操作需要从头节点开始，而不是同按序号取元素 LLget(L，i)算法一样从从首元素节点开始扫描。这一功能的实现可描述为：

```
P = L；j = 0；
while( P != NULL && j < i-1 ){
    P = P->next；j++；
}
```

可以看出，若单链表中有 n 个节点(除头节点外)，则插入操作可以在首元素节点前进行(i=1)，或者在任意两个节点之间(1＜i≤n)，也可以在尾节点之后(i = n+1)进行；即插入序号 i 应满足条件 1≤i≤n+1。

但单链表与顺序表不同，其长度 n 并没有直接给出，要实现对上述条件的判断，必须调用求表长算法 LLlength (L)。显然，这种方法很费时。另一种方法是利用搜索到的 a_{i-1} 节点的地址 P：若 P->next = NULL，表示*P 是尾节点 a_n，可以在尾节点后插入，而且这是最后一个插入位置，等价于 i = n+1；若继续 P = P->next，这时 P = NULL，就不可以在*P 节点后插入了。由此可见，条件 1≤i≤n+1 等价于 P != NULL(P 为 a_{i-1} 节点的地址)。

综合以上分析，单链表的插入算法如下：

```
void LLinsert(LinkList *L，int i，DataType x){
//在带头节点的单链表 L 中第 i 个位置插入值为 x 的新节点
    LinkList *P，*S；
    P = L；j = 0；
    while( P != NULL && j < i-1 ){
        P = P->next；j++；
    }
    if (P = =NULL ) printf( “序号错”)；
    else{
        S = (LinkList *)malloc(sizeof(LinkList))；
```

```
            S->data = x；
            S->next = P->next；
            P->next = S；
        }
    }
```

6、 单链表删除算法

删除单链表 L 中的第 i 个节点 a_i，就是让其后继节点 a_{i+1} 变为其前驱节点 a_{i-1} 的直接后继，即让节点 a_{i-1} 的 next 域获得节点 a_{i+1} 的地址。如图 5.4 所示。

这里 P 为的第 i-1 个节点 a_{i-1} 地址，则删除单链表 L 中的第 i 个节点 a_i 的基本操作可描述为：

P->next = P->next->next；

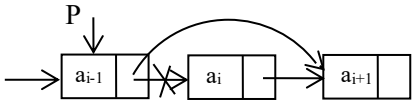


图 5.4 删除算法示意图

如果被删除的节点不再使用，为了不浪费存储空间，我们可以释放所删除节点所占用的存储空间：

free(P->next)；

另一个需要讨论的是删除操作可以进行的条件问题。若单链表中有 n 个节点(除头节点外)，则可以删除的节点应该包括从首元素节点到尾节点的所有节点，即可以被删除的节点的序号 i 满足条件 1≤i≤n；若利用搜索到的 a_{i-1} 节点的地址 P，若 P->next != NULL，

而 P->next->next = NULL，则表示*P 是节点 a_{n-1}(如图 5.5 所示)，这时可以进行最后一次删除操作，删除尾节点 a_n；若继续 P＝P->next，这时 P->next＝NULL，删除操作结束。所以，删除操作可以进行的条件是 P->next != NULL(P 为 a_{i-1} 节点的地址)。

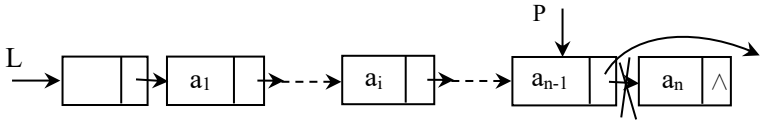


图 5.5 删除尾节点

综合以上分析，单链表的删除算法如下。

```
void LLdelete(LinkList *L，int i){
//在带头节点的单链表 L 中删除第 i 个节点
    LinkList *P，*U；
    int j；
    P = L；j = 0；
    while( P != NULL && j < i-1 ){
            P = P->next；j++；
     }                先找到第 i-1 个节点的存储位置，使指针 P 指向它
     if (P!=NULL && P->next !=NULL){   /第 i-1 个节点和第 i 个节点均存在
            U= P->next；P->next=U->next；free(U) ；
     }
}
```

【实验任务】

1、程序验证 建立含有若干个元素的链表，并实现链表的插入、删除、查找等操作

2、算法填空

（1）下面建立以 head 为头指针的单链表，完善该算法，并输出表中元素。

已知单链表节点类型为：

```
typedef struct node{
    int data;
    struct  node  *next;
}LinkList;
LinkList *create ( ____ ){
    LinkList *p，*q;
    int k；q=head;
    scanf (“%d”，&k);
    while( k>0){
        _____;
        _____;
        _____;
        _____;
        scanf (“%d”，&k);
    }
    q->next = NULL;
}
```

（2）已给如下关于单链表的类型说明：

```
typedef struct node{
    int data;
    struct node *next;
}LinkList;
```

以下程序采用链表合并的方法，将两个已排序的单链表合并成一个链表而不改变其排序性(升序)，这里两链表的头指针分别为 p 和 q。完善该算法，输出合并后表中的元素。

```
LinkList *mergelink(LinkList *p，LinkList *q){
    LinkList *h，LinkList *r;
    _____;
    h->next= NULL；r=h;
    while( p!=NULL&&q!=NULL){
        if (p->data<=q->data){
            _____;
            r=p;
            p=p->next;
        }
        else{ _____  ;
            r=q;
            q=q->next;
        }
    }
```

```
    if (p= =NULL) r->next=q;
    else     _____;
    return   h;
}
```

（3）la 为指向带头节点的单链表的头指针，本算法在表中第 i 个元素之前插入元素 b。完善该算法，并通过运行来验证。

```
LinkList *insert (LinkList *la，int i，datatype b){
    LinkList *p，*s；int j；
    p= _____ ；j= _____；
    while ( p!=NULL &&_____ ) {
        p = _____;
        j=j+1；
    }
    if (p= =NULL || _____ ) error (‘No this position’)
    else{
        s = malloc ( sizeof (LinkList) )；
        s->data=b；s->next=p->next；p->next=s；
    }
    return   la；
 }
```

（4）已知双链表中节点的类型定义为：

```
typedef struct Dnode{
    int data；
    struct Dnode *prior，*next；
}DLinkList ；
```

如下过程将在双链表第 i 个节点(i>=0)之后插入一个元素为 x 的节点。请完善该算法，并通过运行来验证。

```
DLinkList *insert(DLinkList *head，int i，int x){
    DLinkList *s，*p；int j；
    s = malloc ( sizeof (LinkList) )；
    s->data=x；
    if (i= = 0){ //如果 i=0，则将 s 节点插入到表头后返回
        s->next = head；
        _____;
        head=s
    }
    else{
        p = head；_____；//在双链表中查找第 i 个节点，由 p 所指向
        while ( p!= NULL && j<i ){
            j =j+1；        _____ ； }
        if ( p!=NULL )
            if (p->next= =NULL){
                p->next=s；s->next =NULL；_____
            }
```

```
            else{
                s->next=p->next；_____；p->next = s；_____；
            }
        else printf (“can not find node!”)；
    }
}
```

3、算法设计：

（1）在一个非递减有序链表中，插入一个值为 x 的元素，使插入后的链表仍为非递减有序。

（2）设计算法将一个线性链表逆置，即将表(a_1，a_2，……，a_n)逆置为(a_n，a_{n-1}，……，a_1)，要求逆置后的链表仍占用原来的存储空间。

（3）假设有一个循环链表的长度大于 1，且表中既无头节点也无头指针。已知 s 为指向链表某个节点的指针，试编写算法在链表中删除指针 s 所指节点的前驱节点。

试写一算法，在无表头节点的单链表中值为 a 的节点前插入一个值为 b 的节点，如果 a 不存在，则把 b 插在表尾。

（4）假设有一个单向循环链表，其节点包含三个域：data、pre 和 next，其中 data 为数据域，next 为指针域，其值为后继节点的地址，pre 也为指针域，其初值为空(NULL)，试设计算法将此单向循环链表改为双向循环链表。

（5）假设字符串 str 存储在带表头节点的单链表中，编写删除串 str 从位置 i 开始长度为 k 的子串的算法。

4、实例演练：

（1）约瑟夫环

[问题描述]

约瑟夫（Joeph）问题的一种描述是：编号为 1,2,…,n 的 n 个人按顺时针方向围坐一圈，每人持有一个密码（正整数）。一开始任选一个正整数作为报数上限值 m，从第一个人开始按顺时针方向自 1 开始顺序报数，报到 m 时停止报数。报 m 的人出列，将他的密码作为新的 m 值，从他在顺时针方向上的下一个人开始重新从 1 报数，如此下去，直至所有人全部出列为止。试设计一个程序求出出列顺序。

[基本要求]

① 利用单向循环链表存储结构模拟此过程，按照出列的顺序印出各人的编号。

② 设计算法完成问题求解；

③ 分析算法的时间复杂度和空间复杂度。

[测试数据]

m 的初值为 20；密码：3，1，7，2，4，8，4（正确的结果应为 6，1，4，7，2，3，5）。

[实现提示]

程序运行后首先要求用户指定初始报数上限值，然后读取各人的密码。设 n≤30。

[思维扩展]

向上述程序中添加在顺序结构上实现的部分。

（2）长整数运算

[问题描述]

设计一个程序实现两个任意长的整数求和运算。

[基本要求]

① 利用双项循环链表实现长整数的存储，每个结点含一个整型变量。任何整型变量的范围是-（215-1）~（215-1）。输入和输出形式：按中国对于长整数的表示习惯，每四位

一组，组间用逗号隔开。

② 设计算法完成问题求解；

③ 分析算法的时间复杂度。

[测试数据]

① 0；0；应输出“0”。

② -2345,6789；-7654,3211；应输出“-1,0000,0000”。

③ -9999,9999；1,0000,0000,0000；应输出“9999,0000,0001”。

④ 1,0001,000；-1,0001,0001；应输出“0”。

⑤ 1,0001,0001；-1,0001,0000；应输出“1”。

[实现提示]

① 每个结点中可以存放的最大整数为 215-1=32767，才能保证两数相加不会溢出。但若这样存，即相当于按 32768 进制数存，在十进制数与 32768 进制数之间的转换十分不方便。故可以在每个结点中仅存十进制数的 4 位，即不超过 9999 的非负整数，整个链表视为万进制数。

② 可以利用头结点数据域的符号代表长整数的符号。用其绝对值表示元素结点数目。相加过程中不要破坏两个操作数链表。两操作数的头指针存于指针数组中是简化程序结构的一种方法。不能给长整数位数规定上限。

[思维扩展]

修改上述程序，使它在整型量范围是-（2n-1）~（2n-1）的计算机上都能有效地运行。其中，n 是由程序读入的参量。输入数据的分组方法可以另行规定。