

线性表

概念： 拥有n个**相同数据类型**的数据元素的**有限序列**，其中n为表长

$L=(a_1,a_2,a_3,\dots,a_n)$

其中 a_n 中“第n个”表示元素的**位序**（从0开始的是**下标**）

a_1 是表头元素， a_n 是表尾元素

除了 a_1 其他元素都有一个**直接前驱**，除了 a_n 其他元素都有**直接后继**

基本操作

InitList(&L)：初始化表

DestroyList(&L)：销毁表

ListInsert(&L,i,e)：插入操作

ListDelete(&L,i,&e)：删除操作

LocateElem(L,e)：按值查找

GetElem(L,i)：按位查找

Length(L)：求表长

PrintList(L)：打印表

Empty(L)：判空操作

顺序表

逻辑上相邻的元素其物理位置上也相邻

静态分配

```
#define MAXSIZE 10
```

```
typedef struct {
```

```
    int data[MAXSIZE];
```

```
}
```

提前定义指定长度的数组，但是这样的话不能确定实际上的数据元素个数。在初始化表的时候，数组内数据可以不用初始化，但是int length需要初始化为0，定义时会不会自动设为0主要看编译器。

动态分配

```
typedef struct{  
    int *data;  
  
}
```

动态申请空间：

C: malloc free

C++: new delete

//加长表

```
addLength(SqList &L,int length){  
    int *p = L.data;  
  
    L.data = (int *)malloc(sizeof(int)x(L.MaxLength + length));  
  
    for(int i = 0;i < L.length;i++){  
        L.data[i] = p[i];  
    }  
  
    L.MaxLength+=length;  
  
    free p;  
  
}
```

顺序表特点： 随机访问、存储密度高、拓展容量不方便、插入删除不方便

```

15  bool ListInsert(SqList &L,int i,int e){
16      if (L.length >= 10)
17          return false;
18      if (i<1 || i>L.length+1)
19          return false;
20      if (i>10)
21          return false;
22      for (int j = L.length; j >= i; j--)
23      {
24          L.data[j] = L.data[j-1];
25      }
26      L.data[i-1] = e;
27      L.length++;
28      return true;
29  }

```

插入方法

```

33  bool ListDelete(SqList &L,int i,int &e){
34      if (i<1 || i>L.length)
35          return false;
36      e = L.data[i-1];
37      for (int j = i; j < L.length; j++)
38      {
39          L.data[j-1] = L.data[j];
40      }
41      L.length--;
42      return true;
43  }

```

删除方法

```
46  int GetElem(SqList &L,int i){
47      if (i<1 || i>L.length)
48          return 0;
49      return L.data[i-1];
50  }
```

按位查找

```
53  int ElemLocal(SqList &L,int e){
54      for (int i = 0; i < L.length; i++)
55      {
56          if (L.data[i] == e)
57          {
58              return i+1;
59          }
60      }
61      return 0;
62  }
```

按值查找

结构体不能直接使用“==”来判断是否相等

单链表

```
typedef struct{
    int data;//数据域
    struct LNode *next;//指针域
}LNode,*LinkList;
```

不带头结点的单链表

L (头指针) -> LNode (存放数据的)

写代码会更麻烦

```
void init(LinkList &L){  
    L = NULL;  
}
```

带头结点的单链表

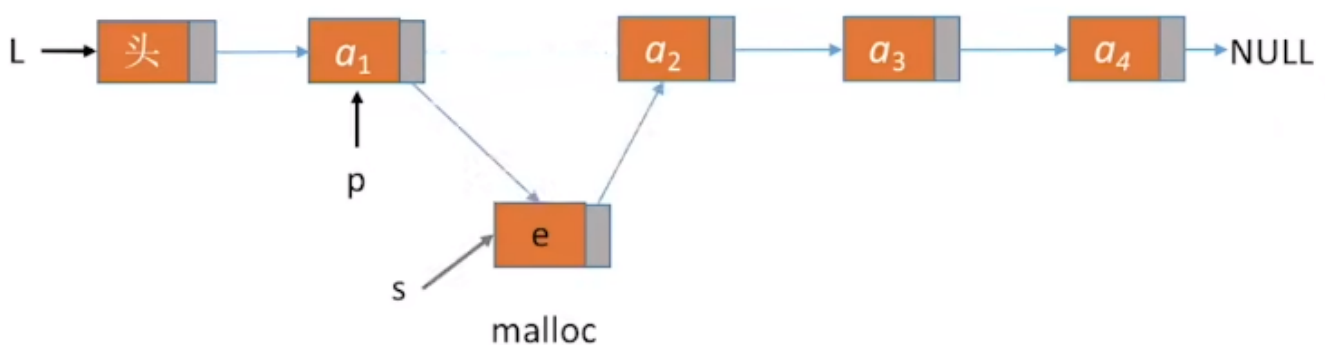
L (头指针) -> LNode (头结点) -> LNode (存放数据的)

带头结点的话后续进行操作的时候会更加方便

```
void init(LinkList &L){  
    L = (LNode *)malloc(sizeof(LNode));  
    L->data = NULL;  
    L->next = NULL;  
}
```

单链表按序插入

【带头结点】 如果要在序号为1的位置插入结点（新结点），则把指针挪动到序号为0的位置（旧结点），把旧结点的next赋给新结点的next，把旧结点的next设置为新结点

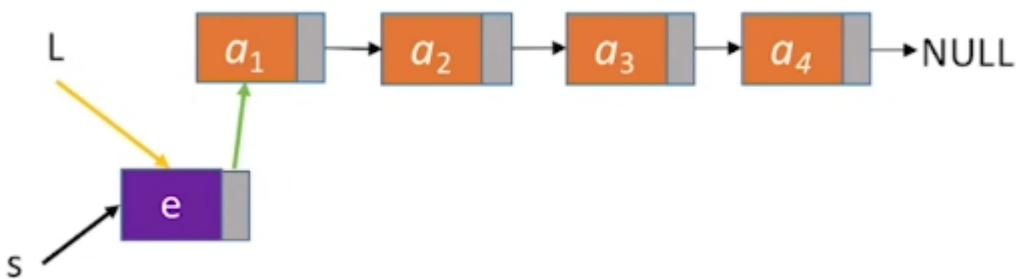


```

void insertHead(LinkList &L,int index,int e){
    if (index < 1)
        return;//插入位置不正确
    LNode *p = L;
    for (int i = 0; i < index - 1; i++)
        p = p->next;//把指针移动到指定位置
    if (p == NULL)
        return;//插入位置超出范围
    LNode *new_node = (LNode *)malloc(sizeof(LNode));
    new_node->data = e;
    new_node->next = p->next;
    p->next = new_node;
}

```

【不带头结点】 没有头结点的单链表在进行插入时，对于第一个结点位置的插入需要单独处理



```

void insert(LinkList &L,int index,int e){
    if (index < 1)
        return;//插入位置不正确
    if (index == 1)
    {
        /* 对于插入位置为1时特殊处理 */
        LNode *s = (LNode *)malloc(sizeof(LNode));
        s->data = e;
        s->next = L;
        L = s;
        return;
    }
    LNode *p = L;
    for (int i = 1; i < index - 1; i++)
        p = p->next;//移动到插入位置-1的结点
    if (p == NULL)
        return;//插入位置不合法
    LNode *s = (LNode*)malloc(sizeof(LNode));
    s->data = e;
    s->next = p->next;
    p->next = s;
}

```

在指定结点后插入结点

给定一个指向某一结点的指针，将在这个结点后插入一个结点

```

void insertNodeNext(LNode *p,int e){
    if (p == NULL)
        return;
    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    s->next = p->next;
    p->next = s;
}

```

删除指定位置的结点

【带头结点】 获取需要删除的结点的前一个结点，然后将前后两个结点相连，并删除指定结点

```

void deleteHead(LinkList &L,int index){
    if(index < 1)
        return;//删除位置不正确
    LNode *p = L;
    for (int i = 0; i < index - 1 && p != NULL; i++)
        p = p->next;//移动到删除位置-1的结点
    if (p->next == NULL)
        return;//如果删除位置的结点为空
    cout<<"delete:"<<p->next->data<<endl;
    LNode *q = p->next;
    p->next = p->next->next;
    free(q);
}

```

单链表的查找

按序查找


```

void getElem(LinkList &L,int index){
    if (index < 1)
        return;//查找位置不合理
    LNode *p = L;
    for (int i = 0; i < index && p->next != NULL; i++)
        p = p->next;
    cout<<index<<": "<<p->data;
}

```

按值查找

```

int LocateElem(LinkList &L,int e){
    LNode *p = L->next;
    int index = 1;
    while (p != NULL)
    {
        if (p->data == e){
            return index;
        }else{
            p = p->next;
            index++;
        }
    }
    return -1;
}

```