线性表——数据结构

线性表的定义

线性表(List)：零个或多个数据元素的有限序列。

元素之间是有序的，若存在多个元素，则第一个元素无前驱，最后一个元素无后继，其他每个元素都有且只有一个前驱和后继。

线性表的顺序存储

指的是用一段地址连续的存储单元依次存储线性表的数据元素。

顺序存储结构

```
#define MAXSIZE 20 //初始存储空间分配量
typedef int Type;
typedef struct
{
    Type data[MAXSIZE]; //数组存储元素
    int length; //线性表当前长度
}SqList;
```

顺序结构的操作

获取元素操作

即返回线性表L中第i个位置的元素，只要i在数组下标的范围内，就是把数组第i-1个下标的值返回。

```
#define TRUE 1
#define FALSE 0
//获取元素操作
int GetElem(SqList L, int i, Type *e)
{
    if (L.length == 0)//线性表为空
        return FALSE;
    if (i < 1 || i > L.length)//i不在范围内
        return FALSE;
    *e = L.data[i - 1];//数组下标从0开始，线性表第i个位置元素的下标为i-1
    return TRUE;
}
```

插入操作

即在线性表L的第i个位置插入新元素e，只要插入的位置合理，从最后一个元素开始向前到第i个元素位置，分别将它们都向后移动一个位置，将插入元素填入位置i处。

```
//插入操作
int ListInsert(SqList *L, int i, Type e)
{
    int j;
```

```
    if (L->length == MAXSIZE)//线性表已满
        return FALSE;
    if (i<1 || i>L->length)//插入位置不合理
        return FALSE;
    if (i <= L->length)//插入位置不在表尾
    {
        for (j = L->length - 1; j >= i - 1; j--)
        {
            L->data[j + 1] = L->data[j];//插入位置后的元素都往后移动一位
        }
    }
    L->data[i - 1] = e;//插入新元素
    L->length++;//表长加一
    return TRUE;
}
```

删除操作

即删除线性表L第i个位置的元素，只要删除位置合理，从删除位置开始到最后一个元素，分别将它们都向前移动一个位置。

```
//删除操作
int ListDelete(SqList *L, int i)
{
    int j;
    if (L->length == 0)//线性表为空
        return FALSE;
    if (i<1 || i>L->length)//删除位置不合理
        return FALSE;
    if (i < L->length)//删除位置不在表尾
    {
        for (j = i; j < L->length; j++)
        {
            L->data[j - 1] = L->data[j];//将删除位置后的元素前移一个位置
        }
    }
    L->length--;//表长减一
    return TRUE;
}
```

线性表顺序存储的优缺点

1. 优点

   不需要为表示表中元素之间的逻辑关系而增加额外的存储空间；

   可以快速的存取表中任意位置的元素；

2. 缺点

   插入和删除操作需要移动大量元素；

   当线性表长度变化较大时，难以确定存储空间的容量；

动态数组实现线性表的顺序存储

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
```

```
4
5  typedef int SLDataType;
6  // 顺序表的动态存储
7  typedef struct SeqList
8  {
9      SLDataType* array; // 指向动态开辟的数组
10     int size;    // 有效数据个数
11     int capicity;  // 容量空间的大小
12 }SeqList;
13
14 // 顺序表初始化
15 void SeqListInit(SeqList* psl)
16 {
17     assert(psl);
18     psl->array = NULL;
19     psl->capicity = 0;
20     psl->size = 0;
21 }
22 // 顺序表销毁
23 void SeqListDestory(SeqList* psl)
24 {
25     assert(psl);
26     free(psl->array);
27     psl->array = NULL;
28     psl->capicity = psl->size = 0;
29 }
30 // 顺序表打印
31 void SeqListPrint(SeqList* psl)
32 {
33     assert(psl);
34     for (int i = 0; i < psl->size; i++)
35     {
36         printf("%d ", psl->array[i]);
37     }
38     printf("\n");
39 }
40 // 检查空间，如果满了，进行增容
41 void CheckCapacity(SeqList* psl)
42 {
43     assert(psl);
44     if (psl->size == psl->capicity)
45     {
46         int newcapicity = psl->capicity * 2;
47         psl->array = (SLDataType*)realloc(psl->array,
   sizeof(SLDataType)*newcapicity);
48         psl->capicity = newcapicity;
49     }
50 }
51 // 顺序表尾插
52 void SeqListPushBack(SeqList* psl, SLDataType x)
53 {
54     assert(psl);
55     SeqListInsert(psl, psl->size, x);
56 }
57 // 顺序表尾删
58 void SeqListPopBack(SeqList* psl)
59 {
60     assert(psl);
```

```c
    SeqListErase(psl, psl->size);
}
// 顺序表头插
void SeqListPushFront(SeqList* psl, SLDataType x)
{
    assert(psl);
    SeqListInsert(psl, 0, x);
}
// 顺序表头删
void SeqListPopFront(SeqList* psl)
{
    assert(psl);
    SeqListErase(psl, 0);
}
// 顺序表查找
int SeqListFind(SeqList* psl, SLDataType x)
{
    assert(psl);
    for (int i = 0; i < psl->size; i++)
    {
        if (psl->array[i] == x)
        {
            return i;
        }
    }
    return -1;
}
// 顺序表在pos位置插入x
void SeqListInsert(SeqList* psl, int pos, SLDataType x)
{
    assert(psl);
    assert(pos<=psl->size);
    CheckCapacity(psl);//检查线性表是否已满，若满了则进行扩容操作
    int end = psl->size;
    while (end > pos)
    {
        psl->array[end] = psl->array[end - 1];
        end--;
    }
    psl->array[pos] = x;
    psl->size++;
}
// 顺序表删除pos位置的值
void SeqListErase(SeqList* psl, int pos)
{
    assert(psl);
    assert(pos<psl->size);
    int start = pos + 1;
    while (start < psl->size)
    {
        psl->array[start - 1] = psl->array[start];
        start++;
    }
    psl->size--;
}
```

线性表的链式存储结构

为了表示每个数据元素与其直接后继数据元素之间的逻辑关系，除了存储本身的信息之外，还需存储一个指示其直接后继的信息(即直接后继的存储位置)。

我们把存储数据元素信息的位置称为**数据域**，把存储其直接后继信息的位置称为**指针域**。这两部分组成数据元素的**结点(Node)**。

头结点和头指针

1. 头结点

   头结点是为了操作的统一和方便而设立的，放在第一元素的节点之前，其数据域一般无意义；

   有了头结点，对在第一个元素结点前插入和删除第一个结点的操作就与其他结点的操作统一了；

   头结点不是链表的必须要有的；

2. 头指针

   头指针是指指向链表第一个结点的指针，若链表有头结点，则是指向头结点的指针；

   无论链表是否为空，头指针均不为空。头指针是链表的必要元素；

链式存储结构

单链表

```
typedef int DateType;
//链表结点
typedef struct ListNode
{
    DateType date;
    struct ListNode* next;
}ListNode;
```

尾插

思路：申请一个新节点newnode，判断单链表是否为空，若为空则直接插入*pplist=newnode，若不为空则寻找单链表的表尾(定义一个tail指针遍历链表，若tail->next=NULL则表示tail是单链表的表尾)再将新节点插入。

```
void ListPushBack(ListNode** pplist, DateType x)
{
    //申请空间，创建一个新结点
    ListNode* newnode = (ListNode*)malloc(sizeof(ListNode));
    //数据域、指针域赋值
    newnode->date = x;
    newnode->next = NULL;
    //判断链表是否为空
    if (*pplist == NULL)
    {
        *pplist = newnode;//若为空，则直接插入
    }
    else//若不为空，寻找链表位进行插入操作
    {
        //定义tail寻找链表尾
        ListNode* tail = *pplist;
        while (tail->next != NULL)
```

```
        {
            tail = tail->next;
        }
        //尾部插入新结点
        tail->next = newnode;
    }
}
```

头插

思路：创建一个新节点newnode ，让newnode的next指向原来的单链表头ppList，ppList指向新创建的结点。

```
void ListPushFront(ListNode** pplist, DateType x)
{
    //申请一个新节点
    ListNode* newnode = (ListNode*)malloc(sizeof(ListNode));
    //数据域、指针域赋值
    newnode->date = x;
    newnode->next = NULL;
    //插入新节点
    newnode->next = *pplist;
    *pplist = newnode;
    }
}
```

删除操作

尾删

思路：定义两个指针，一个pre初始化为NULL用来在遍历时找到单链表的前一个结点，另一个tail初始化指向单链表的表头用来在遍历时找到单链表的表尾；判断单链表的结点个数，若单链表为空或者只有一个结点，则直接*pplist=NULL，若单链表有多个结点，则遍历找到表尾和表尾的前一个指针，将新的表尾置空，释放原来的表尾结点。

```
void ListPopBack(ListNode** pplist)
{
    //定义pre指向链表尾的前一个结点
    ListNode* pre = NULL;
    //定义tail寻找链表尾
    ListNode* tail = *pplist;
    //1.链表为空或者只有一个结点
    if (tail == NULL || tail->next == NULL)
    {
        free(tail);
        *pplist = NULL;
    }
    else//2.链表有多个结点
    {
        while (tail->next != NULL)
        {
            pre = tail;
            tail = tail->next;
        }
        //将新的表尾置空
        pre->next = NULL;
```

```
        //释放掉原来的表尾结点
        free(tail);
    }
}
```

头删

思路： 判断链表的结点个数，1、链表为空，直接return，2、链表只有一个结点，删除这个结点，将来链表置空，3、链表有个结点，定义一个指针next指向链表头结点的下一个结点，删除头结点，重新定义链表的头；

```
void ListPopFront(ListNode** pplist)
{
    ListNode* plist = *pplist;
    //1、空
    if (plist == NULL)
    {
        return;
    }
    //2、只有一个结点
    else if (plist->next == NULL)
    {
        free(plist);
        *pplist = NULL;
    }
    //3、正常情况
    else
    {
        ListNode* next = plist->next;
        free(plist);
        *pplist = next;
    }
}
```

查找操作

思路：按值查找，定义一个指针cur遍历单链表进行查找。

```
ListNode* ListFind(ListNode* plist, DateType x)
{
    ListNode* cur = plist;
    while (cur != NULL)
    {
        if (cur->date == x )
        {
            return cur;
        }
        cur = cur->next;
    }
    return NULL;
}
```

循环链表

将单链表中终端节点的指针域由空指针改为指向头结点，就使整个单链表形成一个环，这种头尾相接的单链表称为单循环链表，简称循环链表(circular linked list)。

双向链表

双向链表(doouble linked list)实在单链表的每个结点中，再设置一个指向其前驱结点的指针域。所以再双向链表中有两个指针域，一个指向直接后继，一个指向直接前驱。

```
//带头结点双向循环链表
typedef int DataType;
typedef struct ListNode
{
    DataType data;//数据域
    struct ListNode* pre;//前驱指针域
    struct ListNode* next;//后继指针域
}ListNode;
```

插入操作

尾插

```
void ListPushBack(ListNode* phead, DataType x)
{
    //申请一个新节点
    ListNode* newnode = (ListNode*)malloc(sizeof(ListNode));
    //数据域赋值
    newnode->data = x;
    //指针域赋值
    newnode->pre = NULL;
    newnode->next = NULL;
    //定义tail指针指向链表的表尾结点
    ListNode* tail = phead->pre;
    //插入
    tail->next = newnode;
    newnode->pre = tail;
    newnode->next = phead;
    phead->pre = newnode;
}
```

头插

```
void ListPushFront(ListNode* phead, DataType x)
{
    //申请一个新节点
    ListNode* newnode = (ListNode*)malloc(sizeof(ListNode));
    //数据域赋值
    newnode->data = x;
    //指针域赋值
    newnode->pre = NULL;
    newnode->next = NULL;
    //定义head指针指向链表的头结点
    ListNode* head = phead->next;
    //插入
    phead->next = newnode;
    newnode->pre = phead;
    newnode->next = head;
    head->pre = newnode;
}
```

删除操作

尾删

```
void ListPopBack(ListNode* phead)
{
    //断言，链表若为空，则不删除
    assert(phead->next != phead);
    //tail指针指向链表的尾结点
    ListNode* tail = phead->pre;
    //prev指针指向尾结点的前一个结点
    ListNode* prev = tail->pre;
    //释放尾结点
    free(tail);
    //重新连接
    prev->next = phead;
    phead->pre = prev;
}
```

头删

```
void ListPopFront(ListNode* phead)
{
    //断言，链表若为空，则不删除
    assert(phead->next != phead);
    //定义first指针指向链表的头结点
    ListNode* first = phead->next;
    //定义second指针指向链表的头结点
    ListNode* second = first->next;
    //释放结点
    free(first);
    //重新建立连接
    phead->next = second;
    second->pre = phead;
}
```