线性逻辑结构

> 线性表：n（n>0）个数据元素的有限序列

相关概念

直接前驱
直接后继
长度

线性结构特点

在数据元素的非空有限集中：

1. 存在唯一一个被称为“第一个”的数据元素；
2. 存在唯一一个被称为“最后一个”的数据元素；
3. 除第一个元素外，每个数据元素均有唯一一个直接前驱；
4. 除最后一个元素外，每个数据元素均有唯一一个直接后继；

线性表基本操作

增：

- 初始化结构
- 插入元素

删：

- 回收结构
- 清空线性表
- 删除结点

查：

- 线性表是否为空
- 线性表的长度
- 求第i个元素
- 求函数值x在线性表中的位置
- 求函数值x在线性表中的直接前驱（直接后继）
- 遍历线性表

改：

- 在查的基础上进行更改

线性表的存储结构

顺序存储结构

顺序存储：将线性表中的元素相继存放在一个连续的存储空间中

静态分配的顺序表存储

```c
#define ListSize 100
typedef int ListData;

//线性表的结构定义，数组从0开始
typedef struct{
    ListData data[ListSize];
    int length;
} SeqList;

//初始化线性表结构
void initList(SeqList &l){
    l.length = 0;
}

//销毁线性结构
void destroyList(SeqList &l){
    l.length = 0;
}

//清空线性结构中的元素
void clearList(SeqList &l){
    l.length = 0;
}

// 线性表是否为空
int listEmpty(SeqList l){
    return (!l.length);
}

// 求线性表的长度
int listLength(SeqList l){
    return l.length;
}

// 获得线性表中第i个元素的值
ListData getElem(SeqList l, int i){
    if(i>=1&&i<=l.length){
        return l.data[i-1];
    }
    else{
        return NULL;
    }
}

// 定位结点在线性表中的位置序号
// 如果结点存在线性表中，返回结点位置，否则返回-1
int locateElem(SeqList l, ListData data){
    int count = 1;
    while((count<=l.length)&&(l.data[count-1]!=data)){
        count++;
    }
    if(count<=l.length){
        return count;
```

```
    }
    else{
        return -1;
    }
}

// 查看结点是否存在线性表中，若存在，则返回1，构造返回0
int isIn(SeqList l, ListData data){
    int count = 1;
    while((count<=l.length)&&(l.data[count-1]!=data)){
        count++;
    }
    if(count<=l.length){
        return 1;
    }
    else{
        return 0;
    }
}

// 定位元素在线性表中的直接后继
// 若结点的直接后继存在，则返回其直接后继的位置，否则返回-1
int nextElem(SeqList l, ListData data){
    int count = 1;
    while((count<=l.length-1)&&(l.data[count-1]!=data)){
        count++;
    }
    if(count<=l.length-1){
        return (count+1);
    }
    else{
        return -1;
    }
}

// 定位元素在线性表中的直接前驱
// 若结点的直接前驱 存在，则返回其直接前驱额位置，否则返回-1
int priorElem(SeqList l, ListData data){
    int count= 2;
    while((count<=l.length)&&(l.data[count-1]!=data)){
        count++;
    }
    if(count<=l.length){
        return (count-1);
    }
    else{
        return -1;
    }
}

// 将结点插入到线性表中的第i个位置（在原线性表的第i个位置之前插入）
// 插入成功则返回1，不成功返回-1
int listInsert(SeqList &l, int i, ListData data){
    if((i<1)||i>(l.length+1)||(l.length==ListSize)){
```

```
        return -1;
    }
    else{
        for(int j=l.length;j>=i;j--){
            l.data[j] = l.data[j-1];
        }
        l.data[i-1] = data;
        l.length++;
        return 1;
    }
}

// 删除线性表中的第i个元素
// 删除 成功返回1，否则返回-1
int listDelete(SeqList &l, int i){
    if((i<1)||(i>l.length)){
        return -1;
    }
    else{
        for(int j=i+1;j<=l.length;j++){
            l.data[j-2] = l.data[j-1];
        }
        l.length--;
        return 1;
    }
}
```

动态分配的顺序表存储

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

#define LIST_INIT_SIZE 10
#define LISTINCREMENT 5

typedef int ListData;

//定义动态存储结构
typedef struct{
    ListData *data;
    int length;
    int listSize;
}SqList;

//初始化动态存储结构
int initList(SqList &L){
    L.data = (ListData*)malloc(LIST_INIT_SIZE*sizeof(ListData));
    if(!L.data){
        exit(0);
    }
    L.length = 0;
    L.listSize = LIST_INIT_SIZE;
    return 1;
}

//销毁动态存储结构
int destroyList(SqList &L){
    free(L.data);
    L.length = 0;
    L.listSize = 0;
    return 1;
}

//清空动态存储结构
int clearList(SqList &L){
    L.length = 0;
    return 1;
}

//是否为空
int listEmpty(SqList L){
    return (L.length == 0);
}

//返回长度
int listLength(SqList L){
    return L.length;
}

//得到第i个元素
ListData getElem(SqList L, int i){
    if(i<1||i>L.length){
```

```
        return -1;
    }
    return *(L.data+i-1);
}

//获得元素的位置
int locateElem(SqList L, ListData data){
    int count = 1;
    while((count<=L.length)&&(*(L.data+count-1)!=data)){
        count++;
    }
    if(count==L.length+1){
        return -1;
    }
    return count;
}

//判断是否在内
int isIn(SqList L, ListData data){
    int count = 1;
    while((count<L.length+1)&&(*(L.data+count-1)!=data)){
        count++;
    }
    if(count==L.length+1){
        return -1;
    }
    return 1;
}

//求直接后继
int nextElem(SqList L, ListData data){
    int count = 1;
    while((count<L.length)&&(*(L.data+count-2)!=data)){
        count++;
    }
    if(count==L.length){
        return -1;
    }
    else{
        return count;
    }
}

//求直接前驱
int priorElem(SqList L, ListData data){
    int count = 1;
    while((count < L.length-1)&&(*(L.data+count)!=data)){
        count++;
    }
    if(count==L.length-1){
        return -1;
    }
    else{
        return count;
```

```
    }
}

//插入元素
int listInsert(SqList &L, int i, ListData data){
    if(i<1||i>L.length+1){
        return -1;
    }
    if(L.length == L.listSize){
        int *newList = (ListData*)realloc(L.data,(L.listSize+LISTINCREMENT)*s
        if(!newList){
            exit(0);
        }
        L.data = newList;
        L.listSize = L.listSize + LISTINCREMENT;
    }
    for(int j = L.length;j>i-1;j--){
        *(L.data+j) = *(L.data+j-1);
    }
    *(L.data+i-1) = data;
    L.length++;
    return 1;
}

//删除元素
int deleteElem(SqList &L, int i){
    if(i<1||i>L.length){
        return -1;
    }
    for(i;i<L.length;i++){
        *(L.data+i-1) = *(L.data+i);
    }
    L.length--;
    return 1;
}
```

线性表的链式存储结构

- 单链表
- 静态链表
- 循环链表
- 双向链表

单链表

单链表：用一组地址任意的存储单元存放线性表中的数据元素
单链表结构：每个元素由结点构成，包括两个域：数据域Data和指针域next

不带头结点的单链表

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

typedef int ListData;

typedef struct node{
    ListData data;
    node* next;
}ListNode;

typedef ListNode* LinkList;

//初始化单链表
void initList(LinkList &first){
    first = NULL;
}

//销毁单链表
void destroyList(LinkList &first){
    ListNode* tem;
    while(first){
        tem = first;
        first = first->next;
        free(tem);
    }
}

//清空单链表
void clearList(LinkList &first){
    ListNode* tem;
    while(first){
        tem = first;
        first = first->next;
        free(tem);
    }
}

//链表是否为空
int listEmpty(LinkList &first){
    return (first==NULL);
}

//求链表的长度
int listLength(LinkList first){
    int count = 1;
    ListNode* tem = first;
    while(tem){
        tem = tem->next;
        count++;
    }
    return count-1;
}

```

```
//获得链表的第i个元素
ListData getElem(LinkList &first, int i){
    ListNode *tem = first;
    int count = 1;
    while(tem&&count<i){
        tem=tem->next;
        count++;
    }
    if(!tem||count > i){
        return NULL;
    }
    return tem->data;

}

//定位结点
int locateElem(LinkList first, ListData data){
    int count = 1;
    ListNode* tem = first;
    while(tem&&tem->data!=data){
        tem = tem->next;
        count++;
    }
    if(!tem){
        return -1;
    }
    return count;
}

//是否在内
int isIn(LinkList first, ListData data){
    ListNode* tem = first;
    while(tem&&tem->data!=data){
        tem=tem->next;
    }
    if(!tem){
        return -1;
    }
    return 1;
}

//求直接后继
ListData nextElem(LinkList first, ListData data){
    ListNode* tem = first;
    while(tem&&tem->next&&tem->data!=data){
        tem = tem->next;
    }
    if(!tem||!tem->next){
        return NULL;
    }
    return tem->next->data;
}

//求直接前驱
```

```
ListData priorElem(LinkList first, ListData data){
    ListNode* tem = first;
    while(tem&&tem->next&&tem->next->data!=data){
        tem = tem->next;
    }
    if(!tem||!tem->next){
        return NULL;
    }
    return tem->data;
}

//插入结点
int listInsert(LinkList &first, int i, ListData data){
    int count = 1;
    ListNode* tem = first;
    if(i == 1){
        ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
        newNode->data = data;
        newNode->next = first;
        first = newNode;
        return 1;
    }
    while(tem&&count<i-1){
        tem = tem->next;
        count++;
    }
    if(!tem||count>i){
        return -1;
    }
    ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->data = data;
    newNode->next = tem->next;
    tem->next = newNode;
    return 1;
}

//删除结点
int listDelete(LinkList &first, int i){
    if(first&&i == 1){
        ListNode* freeNode = first;
        first = first->next;
        free(freeNode);
        return 1;
    }
    int count = 1;
    ListNode* tem = first;
    while(tem&&count<i-1){
        tem = tem->next;
        count++;
    }
    if(!tem||!(tem->next)||i<1){
        return -1;
    }
    ListNode* freeNode = tem->next;
```

```
162.     tem->next = freeNode->next;
163.     free(freeNode);
164.     return 1;
165. }
```

带头结点的单链表

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

typedef int ListData;

typedef struct node{
    ListData data;
    node* next;
}ListNode;

typedef ListNode* LinkList;

void initList(LinkList &first){
    first = (LinkList)malloc(sizeof(ListNode));
    if(!first){
        exit(0);
    }
    first->next = NULL;
}

void destroyList(LinkList &first){
    ListNode* tem = first;
    while(first){
        tem = first;
        first = first->next;
        free(tem);
    }
}

void clearList(LinkList &first){
    ListNode* tem = first->next;
    ListNode* freeNode = first->next;
    while(tem){
        freeNode = tem;
        tem = freeNode->next;
        free(freeNode);
    }
    first->next = NULL;
}

int listEmpty(LinkList first){
    return (first->next==NULL);
}

int listLength(LinkList first){
    int count = 1;
    ListNode* node = first->next;
    while(node){
        node=node->next;
        count++;
    }
    return (count-1);
}
```

```
54.
55.  ListData getElem(LinkList first, int i){
56.      int count = 1;
57.      ListNode* node = first->next;
58.      while(node&&count<i){
59.          node=node->next;
60.          count++;
61.      }
62.      if(!node||count>i){
63.          return NULL;
64.      }
65.      return node->data;
66.  }
67.
68.  int locateElem(LinkList first, ListData data){
69.      int count = 1;
70.      ListNode* node = first->next;
71.      while(node&&node->data!=data){
72.          node = node->next;
73.          count++;
74.      }
75.      if(!node){
76.          return NULL;
77.      }
78.      return count;
79.  }
80.
81.  int isIn(LinkList first, ListData data){
82.      ListNode *node = first->next;
83.      while(node&&node->data!=data){
84.          node = node->next;
85.      }
86.      if(!node){
87.          return -1;
88.      }
89.      return 1;
90.  }
91.
92.  ListData nextElem(LinkList first, ListData data){
93.      ListNode* node = first->next;
94.      while(node&&node->next&&node->data!=data){
95.          node = node->next;
96.      }
97.      if(!node||!node->next){
98.          return NULL;
99.      }
100.     return node->next->data;
101. }
102.
103. ListData priorElem(LinkList first, ListData data){
104.     ListNode* node = first->next;
105.     while(node&&node->next&&node->next->data!=data){
106.         node = node->next;
107.     }
```

```
108.     if(!node||!node->next){
109.         return NULL;
110.     }
111.     return node->data;
112. }
113. 
114. int listInsert(LinkList first, int i, ListData data){
115.     ListNode* node = first;
116.     int count = 0;
117.     while(node&&count < i-1){
118.         node = node->next;
119.         count++;
120.     }
121.     if(!node||count>i-1){
122.         return -1;
123.     }
124.     ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
125.     newNode->data = data;
126.     newNode->next = node->next;
127.     node->next = newNode;
128.     return 1;
129. }
130. 
131. 
132. int listDelete(LinkList first, int i){
133.     int count = 0;
134.     ListNode* node = first;
135.     while(node&&node->next&&count<i-1){
136.         node = node->next;
137.         count++;
138.     }
139.     if(!node||!node->next||i<1){
140.         return -1;
141.     }
142. 
143.     ListNode* freeNode = node->next;
144.     node->next = freeNode->next;
145.     free(freeNode);
146.     return 1;
147. }
```

静态链表

```
const int MaxSize = 100;
typedef int ListData;
typedef struct node{
    ListData data;
    int link;
}SNode;

typedef struct{
    SNode Nodes[MaxSize];
    in newptr;
}SLinkList;

void initSpace(SLinkList &space){
    space.newptr = 0;
    for(int i=0;i<MaxSize-1;i++){
        space.Nodes[i].link = i+1;
    }
    space.Nodes[MaxSize-1].link = -1;
}

void initList(SLinkList &space, int SLink){
    if(space.newptr!=-1){
        SLink = space.newptr;
        space.newptr = space.Nodes[space.newptr].link;
        space.Nodes[SLink].link = -1;
    }
}

void destroyList(SLinkList &space, int SLink){
    int tem = SLink;
    while(space.Nodes[tem].link!=-1){
        tem = space.Nodes[tem].link;
    }
    space.Nodes[tem].link = space.newptr;
    space.newptr = SLink;
    SLink = -1;
}

void clearList(SLinkList &space, int SLink){
    int tem = space.Nodes[SLink].link;
    while(space.Nodes[tem].link!=-1){
        tem = space.Nodes[tem].link;
    }
    space.Nodes[tem].link = space.newptr;
    space.newptr = space.Nodes[SLink].link;

    space.newptr[SLink].link = -1;
}

int listEmpty(SLinkList &space, int SLink){
    return (space.Nodes[SLink].link==-1);
}

```

```
54.
55.  int listLength(SLinkList space, int SLink){
56.      int count = 0;
57.      int tem = SLink;
58.      while(space.Nodes[tem].link!=-1){
59.          tem = space.Nodes[tem].link;
60.          count++;
61.      }
62.      return count;
63.  }
64.
65.  ListData getElem(SLinkList space, int SLink, int i){
66.      int count = 0;
67.      int tem = SLink;
68.      if(SLink<0||SLink>MaxSize-1){
69.          return NULL;
70.      }
71.      while(space.Nodes[tem].link!=-1&&count<i){
72.          tem = space.Nodes[tem].link;
73.          count++;
74.      }
75.      if(space.Nodes[tem].link==-1||count>i){
76.          return NULL;
77.      }
78.      return space.Nodes[count].data;
79.  }
80.
81.
82.  //特殊，返回的不是元素在链表中的位置，而是元素在静态数组中的位置
83.  int locateElem(SLinkList space, int SLink, ListData data){
84.      int count = 0;
85.      int tem = SLink;
86.      while(space.Nodes[tem].link!=-1&&space.Nodes[tem].data!=data){
87.          tem = space.Nodes[tem].link;
88.          count++;
89.      }
90.      if(space.Nodes[tem].link==-1){
91.          return NULL;
92.      }
93.      return tem;
94.  }
95.
96.  int isIn(SLinkList space, int SLink, ListData data){
97.      int tem = SLink;
98.      while(tem!=-1&&space.Nodes[tem].data!=data){
99.          tem = space.Nodes[tem].link ;
100.     }
101.     if(tem==-1){
102.         return -1;
103.     }
104.     return tem;
105. }
106.
107. int nextElem(SLinkList space, int SLink, ListData data){
```

```
108.     int tem = SLink;
109.     while(tem!=-1&&space.Nodes[tem].link!=-1&&space.Nodes[tem].data!=data){
110.         tem = space.Nodes[tem].link;
111.     }
112.     if(tem==-1||space.Nodes[tem].link==-1){
113.         return -1;
114.     }
115.     return space.Nodes[tem].link
116. }
117. 
118. int priorElem(SLinkList space, int SLink, ListData data){
119.     int tem = SLink;
120.     while(tem!=-1&&space.Nodes[tem].link!=-1&&space.Nodes[space.Nodes[tem].li
121.         tem = space.Nodes[tem].link;
122.     }
123.     if(tem==-1||space.Nodes[tem].link==-1){
124.         return -1;
125.     }
126.     return tem;
127. }
128. 
129. int listInsert(SLinkList &space, int SLink, int i, ListData data){
130.     int count = 0;
131.     int tem = SLink;
132.     while(tem!=-1&&count<i-1){
133.         tem = space.Nodes[tem].link;
134.     }
135.     if(tem==-1||count>i-1){
136.         return -1;
137.     }
138.     if(space.newptr!=-1){
139.         int newNode = space.newptr;
140.         space.Nodes[newNode].data = data;
141.         space.Nodes[newNode].link = space.Nodes[tem].link;
142.         space.Nodes[tem].link = newNode;
143.         return 1
144.     }
145.     else{
146.         return -1;
147.     }
148. }
149. 
150. int listDelete(SLinkList &space, int SLink, ListData data){
151.     int count = 0;
152.     int tem = SLink;
153.     while(tem!=-1&&space.Nodes[tem].link!=-1&&space.Nodes[space.Nodes[tem].li
154.         tem = space.Nodes[tem].link;
155.     }
156.     if(tem==-1||space.Nodes[tem].link==-1){
157.         return -1;
158.     }
159.     int freeNode = space.Nodes[tem].link;
160.     space.Nodes[tem].link = space.Nodes[freeNode].link;
161.     space.Nodes[freeNode].link = space.newptr;
```

```
162.         space.newptr = freeNode;
163.         return 1;
164.     }
```

双向循环链表（带头结点）

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

typedef int ListData;
typedef struct DNode{
    ListData data;
    struct DNode *prior, *next;
} DNode;

typedef DNode* DList;

void initList(DList &l){
    l = (DNode*)malloc(sizeof(DNode));
    if(!l){
        printf("存储分配出错！\n");
        exit(1);
    }
    l->prior = l;
    l->next = l;
}

void clearList(DList &l){
    DNode *node = l->next;
    while(node!=l){
        DNode *tem = node;
        node = node->next;
        free(tem);
    }
}

void destroyList(DList &l){
    DNode *node = l->next;
    while(node!=l){
        DNode *tem = node;
        node = node->next;
        free(tem);
    }
    free(l);
}

int listEmpty(DList l){
    return (l->next==l);
}

void listLength(DList l){
    DNode *node = l->next;
    int count = 0;
    while(node!=l){
        node = node->next;
        count++;
    }
}

```

```
ListData getElem(DList l, int i){
    DNode *node = l->next;
    int count = 1;
    while(node!=l&&count<=i){
        node = node->next;
        count++;
    }
    if(count>i||node==l){
        return NULL;
    }
    else{
        return node->data;
    }
}

int locate(DList l, ListData data){
    DNode *node = l->next;
    int count = 0;
    while(node!=l&&node->data!=data){
        count++;
        node = node->next;
    }
    if(node==l){
        return -1;
    }
    else{
        return count;
    }
}

int isIn(DList l, ListData data){
    DNode *node = l->next;
    while(node!=l&&node->data!=data){
        node = node->next;
    }
    if(node==l){
        return -1;
    }
    else{
        return 1;
    }
}

DNode* nextElem(DList l, ListData data){
    DNode *node = l->next;
    while(node!=l&&node->data!=data){
        node = node->next;
    }
    if(node==l||node->next==l){
        return NULL;
    }
    return node->next;
}

```

```
DNode* priorElem(DList l, ListData data){
    DNode *node = l->next;
    while(node!=l&&node->next->data!=data){
        node = node->next;
    }
    if(node==l||node->next==l){
        return NULL;
    }
    return node;
}

int insert(DList l, ListData data, int i){
    int count = 1;
    DNode *node = l->next;
    while(node!=l&&count<i){
        count++;
        node = node->next;
    }
    if(node==l||count>=i){
        return 0;
    }
    else{
        DNode *newNode = (DNode*)malloc(sizeof(DNode));
        newNode->data = data;
        newNode->next = node->next;
        node->next->prior = newNode;
        newNode->prior = node;
        node->next = newNode;
        return 1;
    }
}

int delete(DList l, int i){
    int count=1;
    DNode *node = l->next;
    while(node!=l&&count<=i){
        count++;
        node = node->next;
    }
    if(node == l||count>i){
        return 0;
    }
    node->prior->next = node->next;
    node->next->prior = node->prior;
    free(node);
}
```