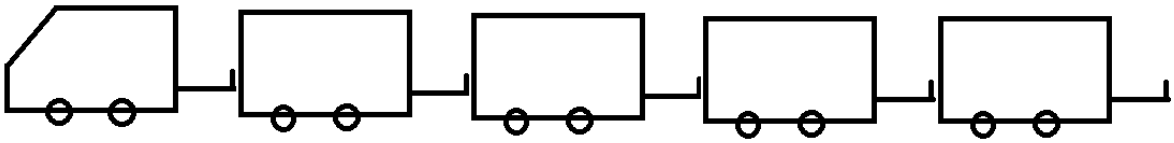
数据结构第二讲

一、链表基本知识

1.1 为啥学习链表

顺序表有个致命缺点：存储元素有上限，一旦到达数组的容量，就不能继续存储数据了

此时可以用链表有效解决该问题



1.2 基本概念

1> 链表：链式存储的线性表

2> 分类：单向链表、双向链表、单向循环链表、双向循环链表

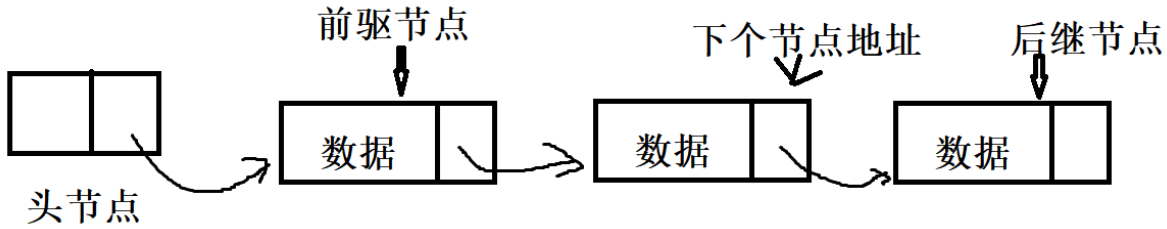
3> 基本单位：结点--->由数据域和指针域两部分组成的基本单元

4> 前驱：当前结点前面的节点称为前驱节点

5> 后继：当前结点的后面的节点称为后继节点

6> 头指针：定义一个节点类型的指针，指向第一个结点的地址，可以表示整条链表

7> 头节点：虚设的一个结点，其指针域指向第一个结点的地址，数据域可以空着不用，也可以记录链表结点个数



二、单向链表

只能从前向后进行访问，不能从后向前访问的链表

2.1 结点结构体类型

结点中由两部分组成：数据（要存储的单个数据）和指针（记录下一个结点的地址）

```
typedef int datatype;

typedef struct Node
{
    union
    {
        int len;            //存储头节点数据域，表示结点个数
        datatype data;            //数据域，普通结点要存放的值
    };
    struct Node *next;          //指针域，记录下个结点的地址
}linkList;


```

2.2 创建链表

申请出来的结点，是一个头节点，讲数据域len赋值为0 指针域next为空

```
linkList *list_create()
{
    linkList *L = (linkList*)malloc(sizeof(linkList));
    if(NULL==L)
    {
        printf("创建失败\n");
        return NULL;
    }

    //给节点初始化
    L->len = 0;    //头结点数据域为0
    L->next = NULL;       //指针域为空

    printf("创建成功\n");
    return L;                 //将创建好的头结点返回
}
```

2.3 判空

对于链表而言，无需进行判满，除非自己电脑内存不足，才会满

对于判空而言，最好使用L->next == NULL,不要使用L->len==0因为在后期增删中，可能会忘记对长度的变化

也可以两个条件同时判断

```
int list_empty(linkList *L)
{
    return L->next==NULL ? 1:0;    //1表示空   0表示非空
}
```

2.4 头插

1> 判断是否合法
2> 申请结点，将要插入的数据，封装到结点中
3> 插入逻辑：最好自己画图分析
4> 表长变化
5> 注意：头插法插入的数据是逆序的

```
int list_insert_head(linkList *L, datatype e)
{
    //判断逻辑
    if(NULL==L)
    {
        printf("所给链表不合法\n");
        return -1;
    }

    //申请节点封装数据
    linkList *p = (linkList*)malloc(sizeof(linkList));
    if(NULL==p)
    {
        printf("空间申请失败\n");
        return -2;
    }
    p->data = e;       //将数据封装到节点内
    p->next = NULL;

    //插入逻辑
    p->next = L->next;
    L->next = p;

    //表的变化
```

```
    L->len++;
    printf("插入成功\n");
    return 0;
}
```

2.5 遍历

定义遍历指针，从第一个结点出发，不断向后访问，只要访问的结点不是空，就输出该结点的数据域

```
void list_show(linkList *L)
{
    //判断逻辑
    if(NULL==L || list_empty(L))
    {
        printf("遍历失败\n");
        return;
    }

    //遍历逻辑
    linkList *q = L->next;      //定义遍历指针从第一个节点出发
    printf("当前链表元素分别是：");
    while(q!=NULL)
    {
        printf("%d\t", q->data);    //输出当前结点的数据域

        q = q->next;                //重新指向下个节点
    }
}
```

2.6 尾插法

1> 判断表的合法性
2> 申请结点，将要插入的数据，封装到结点中
3> 插入逻辑：最好自己画图分析
4> 表长变化
5> 注意：头插法插入的数据是正序的

```
int list_insert_tail(linkList *L, datatype e)
{
    //判断逻辑
    if(NULL == L)
    {
        printf("所给链表不合法\n");
```

```
        return -1;
    }

    //申请节点封装数据
    linkList *p = list_create();
    p->data = e;


    //定义遍历指针找到最后一个节点
    linkList *s = L;
    while(s->next != NULL)
    {
        s = s->next;
    }

    //插入逻辑
    s->next = p;
    L->len++;
    printf("插入成功\n");
    return 0;
}

```

2.7 任意位置插入

1> 判断逻辑：表的合法性、位置合法性
2> 申请结点，将要插入的数据，封装到结点中
3> 插入逻辑：最好自己画图分析
4> 表长变化

```
int list_insert_pos(linkList *L,int pos, datatype e)
{
    //判断逻辑
    if(NULL==L || pos<1 || pos>L->len+1)
    {
        printf("插入失败\n");
        return -1;
    }

    //申请节点封装数据
    linkList *p = list_create();
    p->data = e;

    //定义遍历指针指向要插入节点的前一个节点位置

```

```
    linkList *q = L;
    for(int i=1; i<=pos-1; i++)
    {
        q = q->next;
    }

    //插入逻辑
    p->next = q->next;
    q->next = p;

    //表的变化
    L->len++;
    printf("插入成功\n");
    return 0;
}
```

2.8 头删法

1> 判断逻辑：表的合法性、是否为空
2> 记录要删除的结点
3> 孤立要删除的结点
4> 释放要删除的结点
5> 表的变化

```
int list_delete_head(linkList *L)
{
    //  判断逻辑：表的合法性、是否为空
    if(NULL==L || list_empty(L))
    {
        printf("删除失败\n");
        return -1;
    }
    //  记录要删除的结点
    linkList *p = L->next;

    //  孤立要删除的结点
    L->next = p->next;

    //  释放要删除的结点
    free(p);
    p = NULL;

    //  表的变化
    L->len--;
```

```
    printf("删除成功\n");
    return 0;
}
```

2.9 按位置查找返回查找到的节点

1> 判断逻辑
2>定义遍历指针从头结点出发
3>循环找到要查找位置的节点
4>将该节点返回

```
linkList *list_search_pos(linkList *L, int pos)
{
    //判断逻辑
    if(NULL==L || list_empty(L) || pos<1 || pos>L->len)
    {
        printf("查找失败\n");
        return NULL;
    }

    //定义遍历指针从头结点出发
    linkList *q = L;

    //循环找到要查找位置的节点
    for(int i=1; i<=pos; i++)
    {
        q = q->next;
    }

    //将该节点返回
    return q;                //将找到的节点地址返回
}

```

2.10 任意位置删除

1> 判断逻辑：合法性、判断位置是否合法
2> 定义遍历指针，找到要删除位置的前一个结点
3> 保存要删除的结点
4> 孤立要删除的结点
5> 释放要删除的结点
6> 表的变化

```
int list_delete_pos(linkList *L, int pos)
```

```c
{
    //1>    判断逻辑：合法性、判断位置是否合法
    if(NULL==L || pos<1 || pos>L->len)
    {
        printf("删除失败\n");
        return -1;
    }

    //2>    定义遍历指针，找到要删除位置的前一个结点
    linkList *q = list_search_pos(L, pos-1);

    //3>    保存要删除的结点
    linkList *p = q->next;

    //4>    孤立要删除的结点
    q->next = p->next;  //q->next = q->next->next;

    //5>    释放要删除的结点
    free(p);

    //6>    表的变化
    L->len--;
    printf("删除成功\n");
    return 0;
}
```

2.11 按位置修改

1> 判断逻辑：判断合法性、判断位置释放合法
2> 查找到要修改的结点
3> 将要修改的结点值进行更新

```c
//按位置修改
int list_update_pos(linkList *L, int pos, datatype new_e)
{
    //1>    判断逻辑：判断合法性、判断位置释放合法
    if(NULL==L || pos<1 || pos>L->len)
    {
        printf("修改失败\n");
        return -1;
    }

    //2>    查找到要修改的结点
```

```
    linkList *p = list_search_pos(L, pos);

    //3>    将要修改的结点值进行更新
    p->data = new_e;

    printf("修改成功\n");
    return 0;
}
```

2.12 按值查找

1> 判断逻辑：表的合法性、是否为空
2> 定义遍历指针，将所有结点遍历一遍
3> 如果某个结点的数据域，与要查找的值相等，则将该结点返回
4> 如果全部遍历结束，还没有找到，则返回NULL

```
linkList * list_search_value(linkList *L, datatype e)
{
    //判断逻辑
    if(NULL==L || list_empty(L))
    {
        printf("查找失败\n");
        return NULL;
    }

    //定义遍历指针从第一个节点出发
    linkList *q = L->next;
    while(q!=NULL)
    {
        if(q->data == e)    //将要查找的数据跟链表中元素比较
        {
            return q;     //将查找到的节点返回
        }
        q = q->next;
    }


    printf("查找失败\n");
    return NULL;
    //如果没找到，返回空
}
```

2.13 链表销毁

1> 不能直接调用free函数将头节点释放，因为只是单纯释放头节点的空间，其余结点没有释放，造成内存泄漏

2> 可以不断调用删除函数，完成将所有普通结点全部删除，直到头节点的指针域为空

3> 最后再将头节点空间释放

```
void list_free(linkList *L)
{
    //判断逻辑
    if(NULL == L)
    {
        printf("释放失败\n");
        return ;
    }

    //删除逻辑
    while(L->next != NULL)
    {
        list_delete_head(L);    //不断调用头删，将后面节点释放
    }

    //将头结点空间释放
    free(L);
    L=NULL;          //防止野指针
}
```

三、全部代码

1> linkList.h

```
#ifndef __LINKLIST_H__
#define __LINKLIST_H__

//定义节点类型
typedef int datatype;
typedef struct Node
{
    union
    {
        int len;          //存储头节点数据域，表示结点个数
```

```
        datatype data;          //数据域，普通结点要存放的值
    };
    struct Node *next;          //指针域，记录下个结点的地址
}linkList;

//操作
//创建链表
linkList *list_create();

//判空
int list_empty(linkList *L);

//头插法
int list_insert_head(linkList *L, datatype e);

//遍历
void list_show(linkList *L);

//尾插法
int list_insert_tail(linkList *L, datatype e);

//任意位置插入
int list_insert_pos(linkList *L,int pos, datatype e);

//头删
int list_delete_head(linkList *L);

//按位置查找返回查找到的节点
linkList *list_search_pos(linkList *L, int pos);

//尾删
int list_delete_tail(linkList *L);

//任意删除
int list_delete_pos(linkList *L, int pos);

//按值修改
int list_update_value(linkList *L, datatype old_e, datatype
new_e);

//按位置修改
int list_update_pos(linkList *L, int pos, datatype new_e);

//按值查找
linkList* list_search_value(linkList *L, datatype e);
```

```
55
56 //销毁
57 void list_free(linkList *L);
58 #endif
```

2> linkList.c

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<malloc.h>
4  #include"linkList.h"
5
6  //创建链表
7  linkList *list_create()
8  {
9      linkList *L = (linkList*)malloc(sizeof(linkList));
10     if(NULL==L)
11     {
12         printf("创建失败\n");
13         return NULL;
14     }
15
16     //给节点初始化
17     L->len = 0;    //头结点数据域为0
18     L->next = NULL;       //指针域为空
19
20     printf("创建成功\n");
21     return L;                //将创建好的头结点返回
22 }
23
24 //判空
25 int list_empty(linkList *L)
26 {
27     return L->next==NULL ? 1:0;    //1表示空   0表示非空
28 }
29
30 //头插法
31 int list_insert_head(linkList *L, datatype e)
32 {
33     //判断逻辑
34     if(NULL==L)
35     {
36         printf("所给链表不合法\n");
37         return -1;
38     }
```

```c

    //申请节点封装数据
    linkList *p = (linkList*)malloc(sizeof(linkList));
    if(NULL==p)
    {
        printf("空间申请失败\n");
        return -2;
    }
    p->data = e;       //将数据封装到节点内
    p->next = NULL;

    //插入逻辑
    p->next = L->next;
    L->next = p;

    //表的变化
    L->len++;
    printf("插入成功\n");
    return 0;
}

//遍历
void list_show(linkList *L)
{
    //判断逻辑
    if(NULL==L || list_empty(L))
    {
        printf("遍历失败\n");
        return;
    }

    //遍历逻辑
    linkList *q = L->next;      //定义遍历指针从第一个节点出发
    printf("当前链表元素分别是：");
    while(q!=NULL)
    {
        printf("%d\t", q->data);    //输出当前结点的数据域

        q = q->next;                //重新指向下个节点
    }
    printf("\n");
}

//尾插法
int list_insert_tail(linkList *L, datatype e)
```

```c
{
    //判断逻辑
    if(NULL == L)
    {
        printf("所给链表不合法\n");
        return -1;
    }

    //申请节点封装数据
    linkList *p = list_create();
    p->data = e;


    //定义遍历指针找到最后一个节点
    linkList *s = L;
    while(s->next != NULL)
    {
        s = s->next;
    }

    //插入逻辑
    s->next = p;

    //表的变化
    L->len++;
    printf("插入成功\n");
    return 0;
}

//任意位置插入
int list_insert_pos(linkList *L,int pos, datatype e)
{
    //判断逻辑
    if(NULL==L || pos<1 || pos>L->len+1)
    {
        printf("插入失败\n");
        return -1;
    }

    //申请节点封装数据
    linkList *p = list_create();
    p->data = e;

    //定义遍历指针指向要插入节点的前一个节点位置
    linkList *q = L;
```

```
    for(int i=1; i<=pos-1; i++)
    {
        q = q->next;
    }

    //插入逻辑
    p->next = q->next;
    q->next = p;

    //表的变化
    L->len++;
    printf("插入成功\n");
    return 0;
}

//头删
int list_delete_head(linkList *L)
{
    //  判断逻辑：表的合法性、是否为空
    if(NULL==L || list_empty(L))
    {
        printf("删除失败\n");
        return -1;
    }
    //  记录要删除的结点
    linkList *p = L->next;

    //  孤立要删除的结点
    L->next = p->next;

    //  释放要删除的结点
    free(p);
    p = NULL;

    //  表的变化
    L->len--;
    printf("删除成功\n");
    return 0;
}

//按位置查找返回查找到的节点
linkList *list_search_pos(linkList *L, int pos)
{
    //判断逻辑
    if(NULL==L || list_empty(L) || pos<0 || pos>L->len)
```

```
    {
        printf("查找失败\n");
        return NULL;
    }

    //定义遍历指针从头结点出发
    linkList *q = L;

    //循环找到要查找位置的节点
    for(int i=1; i<=pos; i++)
    {
        q = q->next;
    }

    //将该节点返回
    return q;                //将找到的节点地址返回
}

//尾删
int list_delete_tail(linkList *L);

//任意删除
int list_delete_pos(linkList *L, int pos)
{
    //1>    判断逻辑：合法性、判断位置是否合法
    if(NULL==L || pos<1 || pos>L->len)
    {
        printf("删除失败\n");
        return -1;
    }

    //2>    定义遍历指针，找到要删除位置的前一个结点
    linkList *q = list_search_pos(L, pos-1);

    //3>    保存要删除的结点
    linkList *p = q->next;

    //4>    孤立要删除的结点
    q->next = p->next;  //q->next = q->next->next;

    //5>    释放要删除的结点
    free(p);

    //6>    表的变化
    L->len--;
```

```
    printf("删除成功\n");
    return 0;
}

//按值修改
int list_update_value(linkList *L, datatype old_e, datatype new_e);

//按位置修改
int list_update_pos(linkList *L, int pos, datatype new_e)
{
    //1>    判断逻辑：判断合法性、判断位置释放合法
    if(NULL==L || pos<1 || pos>L->len)
    {
        printf("修改失败\n");
        return -1;
    }

    //2>    查找到要修改的结点
    linkList *p = list_search_pos(L, pos);

    //3>    将要修改的结点值进行更新
    p->data = new_e;

    printf("修改成功\n");
    return 0;
}

//按值查找返回查找到的节点
linkList * list_search_value(linkList *L, datatype e)
{
    //判断逻辑
    if(NULL==L || list_empty(L))
    {
        printf("查找失败\n");
        return NULL;
    }

    //定义遍历指针从第一个节点出发
    linkList *q = L->next;
    while(q!=NULL)
    {
        if(q->data == e)    //将要查找的数据跟链表中元素比较
        {
            return q;     //将查找到的节点返回
```

```
        }
        q = q->next;
    }


    printf("查找失败\n");
    return NULL;
    //如果没找到，返回空
}


//销毁
void list_free(linkList *L)
{
    //判断逻辑
    if(NULL == L)
    {
        printf("释放失败\n");
        return ;
    }

    //删除逻辑
    while(L->next != NULL)
    {
        list_delete_head(L);    //不断调用头删，将后面节点释放
    }

    //将头结点空间释放
    free(L);
    L=NULL;          //防止野指针
}
```

3> main.c

```
#include"linkList.h"
#include<stdio.h>
int main(int argc, const char *argv[])
{
    linkList *L = list_create();
    if(NULL == L)
    {
        return -1;
    }

    //调用头插函数
```

```c
    list_insert_head(L, 6);
    list_insert_head(L, 2);
    list_insert_head(L, 3);
    list_insert_head(L, 5);

    //调用遍历函数
    list_show(L);

    //调用尾插函数
    list_insert_tail(L, 1);
    list_insert_tail(L, 9);
    list_show(L);

    //调用任意位置插入函数
    list_insert_pos(L, 3, 9);
    list_show(L);

    //调用头删函数
    list_delete_head(L);
    list_delete_head(L);
    list_show(L);

    //调用任意位置删除函数
    list_delete_pos(L, 3);
    list_show(L);

    //调用任意位置修改函数
    list_update_pos(L, 1, 7);
    list_show(L);

    //调用按值查找函数
    linkList *res = list_search_value(L, 5);
    if(res != NULL)
    {
        printf("您要找的节点在链表中\n");
    }

    //调用释放函数
    list_free(L);
    L = NULL;
    list_show(L);

    return 0;
}
```

作业

作业1：实现链表尾删函数、按值进行修改函数、按值查找并返回第一次出现的位置函数
作业2：实现链表的反转：例如原链表数据为：1 2 3 4 5 调用反转函数后，该链表数据为：5 4 3 2 1
作业3：将今天学习内容总结到思维导图上