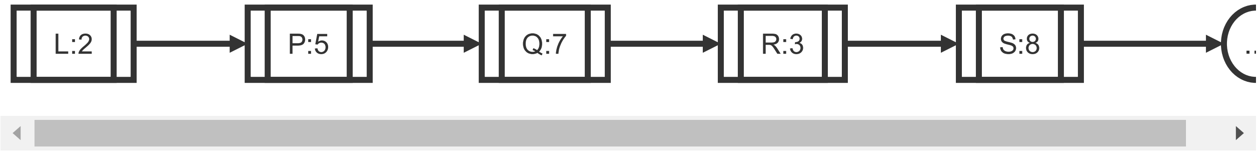
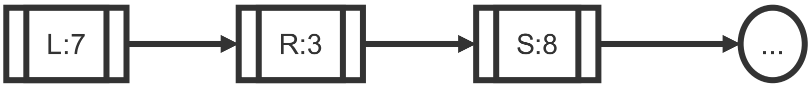
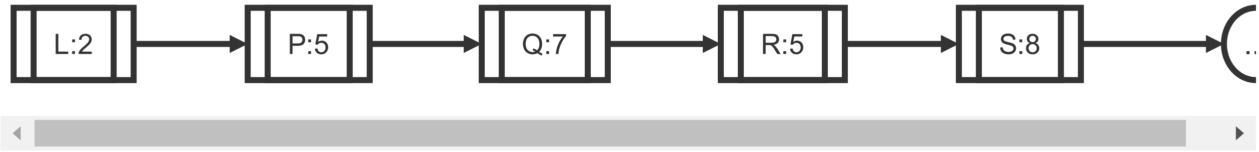
数据结构作业

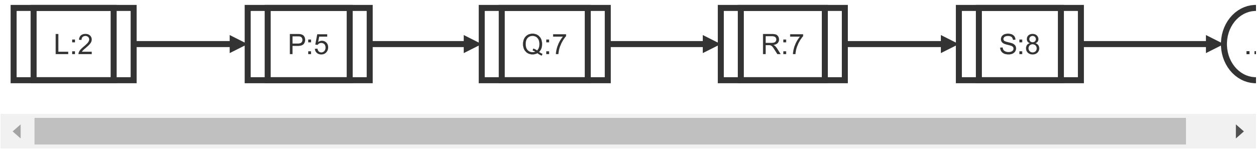
第二章

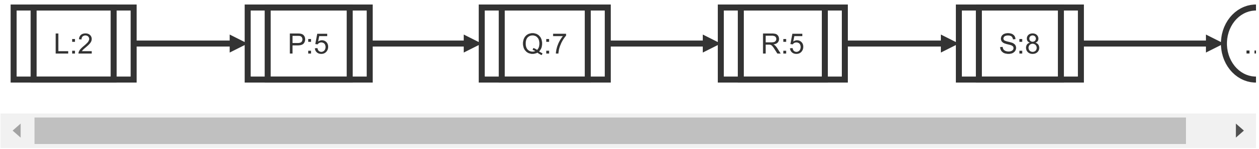
2.4

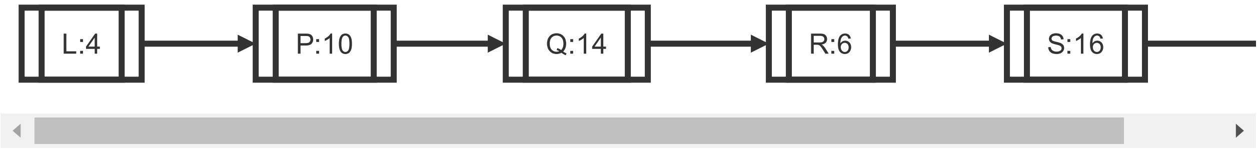


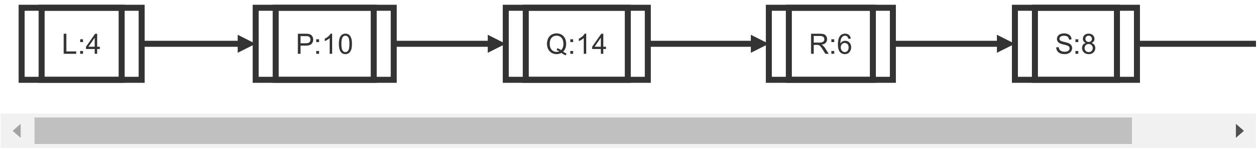






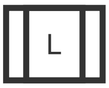




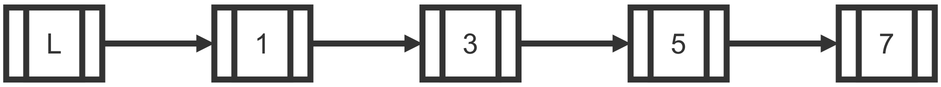


2.5

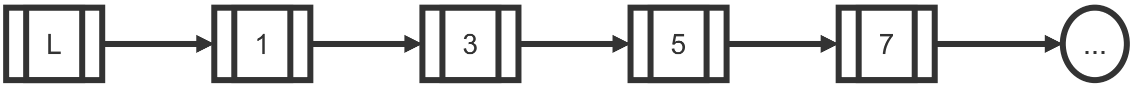
第一行后



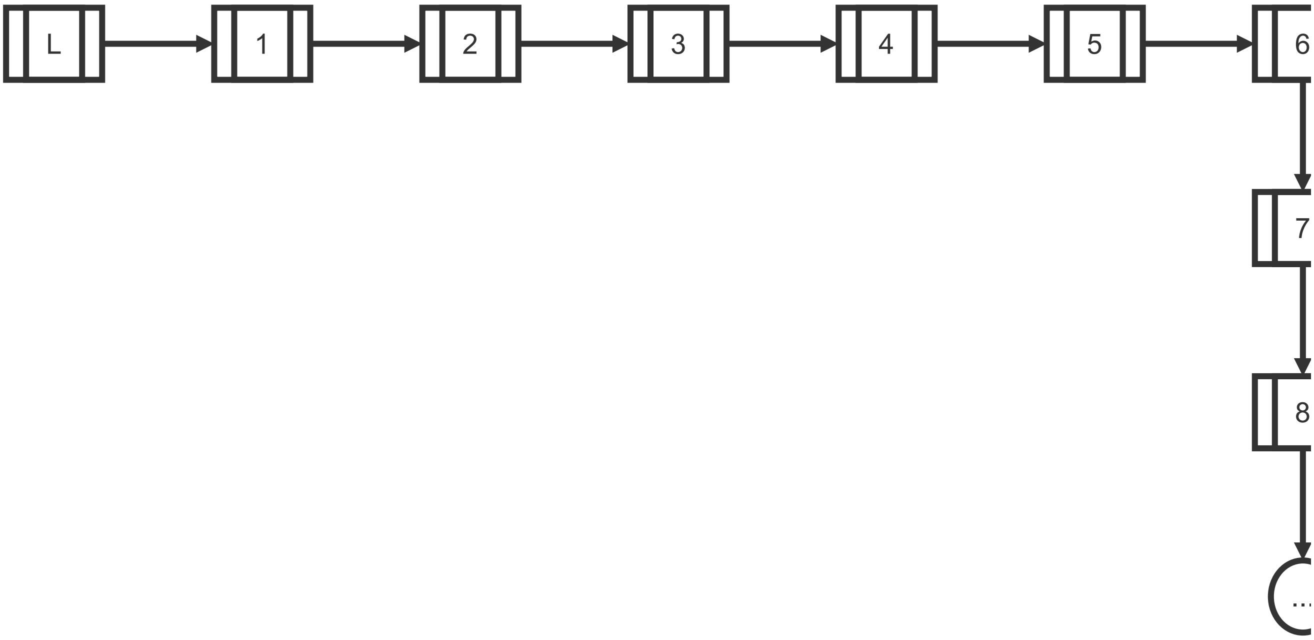
第二行后



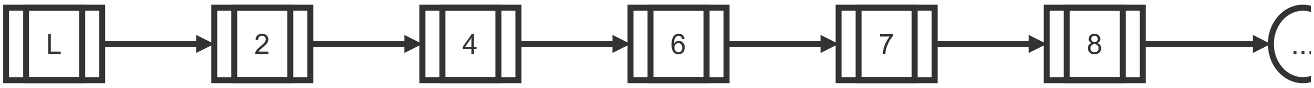
第三行后



第四行后



第五行后



2.9

1. 若`L`非空且长度大于2, 把`L`的头结点变为尾节点
2. 把单循环链表在`pa`, `pb`处断开, 形成两个单循环链表

2.11

```c
#include <stdio.h>
#include "list.h"       //附在本章作业的最后

void list_order_insert(List*, TYPE);

void main(){
    //生成测试列表并显示
    int array_size = 6;
    TYPE array[] = {3, 5, 8, 235, 555, 1996};
    int list_max_size = 10;
    List L = list_create_from_array(list_max_size, array, array_size);
    list_print(L);              //3 5 8 235 555 1996
    //插入并显示结果
    list_order_insert(&L, 18);
    list_print(L);              //3 5 8 18 235 555 1996
    return;
}

void list_order_insert(List* L, TYPE data){
    Node* p = L->head;

    while(p->next != NULL){
        if (p == L->head) {;}
        else if (p->data <= data && p->next->data >= data){
            break;
        }
        p = p->next;
    }

    Node* insert;
    insert = (Node*)malloc(sizeof(Node));
    //当内存分配失败，报错
    if (insert == NULL){
        printf("Fail to create List, storage allocation error\n");
        exit(MALLOC);       //MALLOC定义在list.h中，为-1
    }
    insert->data = data;
    insert->next = p->next;
    p->next = insert;
    L->size ++;
}
```

2.12

```c
#include <stdio.h>
#include "list.h"    //附在本章作业的最后

int list_compare(List, List);

void main(){
    //生成测试列表A, B
    TYPE array_A[] = {1, 2, 3, 4, 666};
    TYPE array_B[] = {1, 2, 3, 4, 666, 7};
    int array_size_A = sizeof(array_A) / sizeof(TYPE);
    int array_size_B = sizeof(array_B) / sizeof(TYPE);
    int list_max_size = 20;
    List L_A = list_create_from_array(list_max_size, array_A, array_size_A);
    List L_B = list_create_from_array(list_max_size, array_B, array_size_B);
    printf("L_A:");
    list_print(L_A);
    printf("L_B:");
    list_print(L_B);
    //比较链表
    int compare_result = list_compare(L_A, L_B);
    //输出结果
    if (compare_result == 0){
        printf("L_A = L_B\n");
    }
    else if (compare_result == 1){
        printf("L_A > L_B\n");
    }
    else if (compare_result == -1){
        printf("L_A < L_B\n");
    }
    return;
}

int list_compare(List L_A, List L_B){
    Node* pa = L_A.head->next;
    Node* pb = L_B.head->next;

    while(pa != NULL && pb != NULL){
        //当A的值较大
        if (pa->data > pb->data){
            return 1;
        }
        //当B的值较大
        else if (pa->data < pb->data){
            return -1;
        }
        else{
            //当A, B同时为空
            if (pa->next == NULL && pb->next == NULL){
                return 0;
            }
            //当A空

            else if (pa->next == NULL){
                return -1;
            }
            //当B空
            else if (pb->next == NULL){
                return 1;
            }
            //继续迭代
            else{
                pa = pa->next;
                pb = pb->next;
            }
        }
    }
}
```

2.19

```
#include <stdio.h>
#include "list.h"   //附在本章作业的最后

void list_range_delete(List* L, TYPE min, TYPE max);

void main(){
    //生成并暂时测试列表
    TYPE array[] = {3, 5, 8, 235, 555, 1996};
    int array_size = sizeof(array) / sizeof(TYPE);
    int list_max_size = 10;
    List L = list_create_from_array(list_max_size, array, array_size);
    printf("L before:\n");
    list_print(L);

    //删除指定范围
    int min = 7;
    int max = 300;
    list_range_delete(&L, min, max);
    //展示结果
    printf("L after delete:\n");
    list_print(L);
    return;
}

void list_range_delete(List* L, TYPE min, TYPE max){
    Node* p = L->head;
    Node* temp;

    while(p->next != NULL){
        //当符合要求, 删除
        if (p->next->data > min && p->next->data < max){
            temp = p->next;
            p->next = temp->next;
            free(temp);
            L->size --;
        }
        //当不符合要求, 指针后移一位, 继续寻找
        else{
            p = p->next;
        }
    }
}
```

2.22

```c
#include <stdio.h>
#include "list.h"    //附在本章作业的最后

void list_reverse(List L);

void main(){
    //生成并显示测试列表
    TYPE array[] = {3, 5, 8, 235, 555, 1996};
    int array_size = sizeof(array) / sizeof(TYPE);
    int list_max_size = 10;
    List L = list_create_from_array(list_max_size, array, array_size);
    printf("L before:\n");
    list_print(L);

    //逆置链表
    list_reverse(L);
    printf("L after reverse:\n");
    list_print(L);
    return;
}

void list_reverse(List L){
    //取下头节点
    Node* p = L.head->next;
    L.head->next = NULL;
    Node* temp;

    //当节点非空，取下来插到头结点后
    while(p != NULL){
        temp = p->next;
        p->next = L.head->next;
        L.head->next = p;
        p = temp;
    }
}
```

2.29

```c
#include <stdio.h>
#include "list.h"   //附在本章作业的最后

void list_delete_cross(List*, List, List);
int list_cross(List, List, TYPE*);

void main(){
    //生成测试列表A, B
    TYPE array_A[] = {1, 2, 3, 3, 5, 44, 666, 7777};
    TYPE array_B[] = {1, 2, 3, 4, 666, 1117};
    TYPE array_C[] = {1, 2, 3, 4, 666, 1107};
    int array_size_A = sizeof(array_A) / sizeof(TYPE);
    int array_size_B = sizeof(array_B) / sizeof(TYPE);
    int array_size_C = sizeof(array_C) / sizeof(TYPE);
    int list_max_size = 20;
    List L_A = list_create_from_array(list_max_size, array_A, array_size_A);
    List L_B = list_create_from_array(list_max_size, array_B, array_size_B);
    List L_C = list_create_from_array(list_max_size, array_C, array_size_C);
    printf("L_A:\n");
    list_print(L_A);
    printf("L_B:\n");
    list_print(L_B);
    printf("L_C:\n");
    list_print(L_C);

    //删除A中的部分元素
    list_delete_cross(&L_A, L_B, L_C);
    printf("L_A after delete:\n");
    list_print(L_A);

    return;
}

void list_delete_cross(List* L_A, List L_B, List L_C){
    //取B, C中比较长的那个的size构造数组, 用于存储公共元素
    int size = (L_B.size > L_C.size)?L_B.size:L_C.size;
    TYPE result[size];
    int count = list_cross(L_B, L_C, result);

    //删除A中的B, C公共元素.
    int index = 0;
    Node* pa = L_A->head;
    Node* temp;
    while (pa->next != NULL && index < count){
        //当是公共元素, 删除
        if (pa->next->data == result[index]){
            temp = pa->next;
            pa->next = temp->next;
            free(temp);
            L_A->size --;
        }
        //当大于当前比较的公关元, 比较下一个

        else if (pa->next->data > result[index]){
            index ++;
        }
        //当大于当前比较的公关元，指针偏移一位
        else if (pa->next->data < result[index]){
            pa = pa->next;
        }
    }
}

//找出B，C中的公共元素，存在result中，返回公共元素个数
int list_cross(List L_B, List L_C, TYPE* result){
    int count = 0;
    Node* pb = L_B.head->next;
    Node* pc = L_C.head->next;
    while(pb != NULL && pc != NULL){
        if (pb->data == pc->data){
            result[count] = pb->data;
            count ++;
            pb = pb->next;
            pc = pc->next;
        }
        else if (pb->data > pc->data){
            pc = pc->next;
        }
        else if (pb->data < pc->data){
            pb = pb->next;
        }
    }
    //返回公共元素计数器
    return count;
}
```

2.41

```
typedef struct Poly{
    int order;
    double coef;
}Poly;

#define TYPE Poly               //list.h中的TYPE是int，这里需要修改
#include <stdio.h>
#include "list.h"                //附在本章作业的最后

TYPE* create_Poly_from_array(double* array, int size, TYPE* polys);
void poly_equal(TYPE* a, TYPE b);
void poly_list_print(List L);
void poly_diff(List* L);

void main(){
    //生成多项式系数数组
    double array[] = {0, 3, 4, 22.1, 6, 0.5, 20, 77, 50, 28};
    int size = sizeof(array) / sizeof(double) / 2;
    TYPE polys[size];
    create_Poly_from_array(array, size, polys);
    //生成多项式链表
    int max_size = 20;
    List L = list_create_from_array(max_size, polys, size, &poly_equal);
    printf("The polynomial before:\n");
    poly_list_print(L);
    //求导
    poly_diff(&L);
    printf("The polynomial after differentiate:\n");
    poly_list_print(L);
    return;
}

//求偏导
void poly_diff(List* L){
    Node* p = L->head->next;
    while(p != NULL){
        //对于多项式的常数项，去掉这个结点
        //若该结点存在，必定紧接着头结点
        if (p->data.order == 0){
            p = p->next;
            free(L->head->next);
            L->head->next = p;
            L->size --;
        }
        else{
            p->data.coef *= p->data.order;
            p->data.order --;
            p = p->next;
        }
    }
}


//好气啊，好想重载啊，好想用默认参数啊
```

```
TYPE* create_Poly_from_array(double* array, int size, TYPE* polys){
    for (int i=0; i < size; i++){
        Poly temp;
        temp.order = array[2*i];
        temp.coef = array[2*i+1];
        poly_equal(&(polys[i]), temp);
    }
}

//重载(???)结构体poly的等于
void poly_equal(TYPE* a, TYPE b){
    (*a).order = b.order;
    (*a).coef = b.coef;
}

//输出多项式
void poly_list_print(List L){
    Node* p = L.head->next;
    while (p != NULL){
        if (p == L.head->next){
            if (p->data.order == 0){
                printf("%lf", p->data.coef);
            }
            else{
                printf("%lf*x^%d", p->data.coef, p->data.order);
            }
        }
        else{
            printf(" + %lf*x^%d", p->data.coef, p->data.order);
        }
        p = p->next;
    }
    printf("\n");
}
```

附:`list.h`

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

#define MALLOC      -1
//判断是否重定义了TYPE
#ifndef TYPE
    //当未重定义
    #define TYPE        int
    #define RELOADTAG   0
    #define list_create_from_array(max_size, array, array_size) _list_create_from_array(max_size, array, array_size, &useless)
#else
    //当重新定义了TYPE的类型，需要重载运算，使用函数指针和宏定义(虽然讲道理不如重新写一个-_-)
    #define RELOADTAG   1
    #define list_create_from_array(max_size, array, array_size, function) _list_create_from_array(max_size, array, array_size, function)
#endif

//默认使用int类型
typedef struct node{
    TYPE data;
    struct node* next;
}Node;

typedef struct {
    Node* head;
    int max_size;
    int size;
}List;

List list_create(int);
void list_print(List);
List _list_create_from_array(int max_size, TYPE* array, int array_size, void(*function)(TYPE*, TYPE));
//这个函数没什么用，在未重定义TYPE时使用
void useless(TYPE* a, TYPE b){;}


List list_create(int max_size){
    Node *head;
    head = (Node*)malloc(sizeof(Node));
    if (head == NULL){
        printf("Fail to create List, storage allocation error\n");
        exit(MALLOC);
    }
    head->next = NULL;

    List L;
    L.head = head;
    L.max_size = max_size;
    L.size = 0;


    return L;
}

//为了重载，连函数指针都拿出来了
List _list_create_from_array(int max_size, TYPE* array, int array_size, void (*function)(TYPE*,
TYPE)){
    List L = list_create(max_size);
    for (int i=0; i < array_size; i++){
        Node* p;
        p = (Node*)malloc(sizeof(Node));
        //当内存分配失败，报错
        if (p == NULL){
            printf("Fail to create List, storage allocation error\n");
            exit(MALLOC);
        }
        //逆序插入数组
        //TYPE != int
        if (RELOADTAG){
            (*function)(&(p->data), array[array_size-i-1]);
        }
        //TYPE == int
        else{
            p->data = array[array_size-i-1];
        }
        p->next = L.head->next;
        L.head->next = p;
    }
    L.size = array_size;
    return L;
}
```