hw2

2.19

忽略鲁棒性，从第一个>mink的节点开始删除到第一个>=maxk的节点结束；

```
while(curr->next != NULL) {
    if (mink < key  < maxk) delete node;
    else curr = curr->next;
}
```

```c
// 对排列好的顺序表，高效删除值在[mink, maxk]区间的节点
struct Node* delete_range(struct Node* head, int mink, int maxk) {
    if (head == NULL) return NULL;

    while (head->data > mink && head->data < maxk) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
    }

    struct Node* curr = head;
    while (curr != NULL && curr->next != NULL) {
        if (curr->next->data > mink && curr->next->data < maxk) {
            struct Node* temp = curr->next;
            curr->next = curr->next->next;
            free(temp);
        } else {
            curr = curr->next;
        }
    }
    return head;
}
```

时间复杂度：O(n)

2.21

双指针，一头一尾互换值

```
// 线性表逆转
void reverse_seq_list(SeqList* list) {
    int left = 0;
    int right = list->size - 1;
    while (left < right) {
        int temp = list->data[left];
        list->data[left] = list->data[right];
        list->data[right] = temp;
        left++;
        right--;
    }
}
```

2.24

两个指针从头开始遍历，每次取较小的那个作为Merged_list的节点，连接到Merged_list的前面；直到有一个表全部被合并，将剩余的绩点全部合并到Merged_list最前面；

```
while(list1 != NULL && list2 != NULL) {
    if (list1<list2) {
        temp = list1-next;
        add_to_head(list1, merged);
        list1 = temp;
    } else {连接list2}
}
while(list1 != NULL) {
    temp = list1-next;
    add_to_head(list1, merged);
    list1 = temp;
}
while(list2 != NULL) {
    同理;
}
```

```
// 合并两个递增链表，合并后链表变为递减
struct Node* merge_and_reverse(struct Node* l1, struct Node* l2) {
    struct Node* merged = NULL;
    while (l1 != NULL && l2 != NULL) {
        if (l1->data < l2->data) {
            struct Node* next = l1->next;
            l1->next = merged;
            merged = l1;
            l1 = next;
        } else {
            struct Node* next = l2->next;
            l2->next = merged;
            merged = l2;
            l2 = next;
        }
    }
    while (l1 != NULL) {
        struct Node* next = l1->next;
        l1->next = merged;
        merged = l1;
        l1 = next;
    }
    while (l2 != NULL) {
        struct Node* next = l2->next;
        l2->next = merged;
        merged = l2;
        l2 = next;
    }
    return merged;
}
```

2.29

假设ABC长度分别为nmk：

暴力求解：

```
// 已知ABC递增，删除A中与B和C中都有的元素
void delete_common_elements(SeqList* A, SeqList* B, SeqList* C) {
    for (int i = 0; i < A->size; i++) {
        if (is_in_list(B, A->data[i]) && is_in_list(C, A->data[i])) {
            // 删除搬移
            for (int j = i; j < A->size - 1; j++) {
                A->data[j] = A->data[j + 1];
            }
            A->size--;
            i--;  // 重新检查当前位置
        }
    }
}
```

时间复杂度： O(n^2 + n (m + k))

优化版：

不搬运元素，仅覆写数据；

三个指针，i遍历A，jk保证 B[j], C[k] >= A[i];

若指针停下后恰好三者相同，则i++；否则写入A[i]到A[w]，i++，w++；

```
void delete_common_elements_sorted(SeqList* A, SeqList* B, SeqList* C) {
    // 三个读指针，一个写指针
    int i = 0, j = 0, k = 0;
    int w = 0;

    while (i < A->size) {
        int x = A->data[i];

        while (j < B->size && B->data[j] < x) j++;
        while (k < C->size && C->data[k] < x) k++;

        int inB = (j < B->size && B->data[j] == x);
        int inC = (k < C->size && C->data[k] == x);

        if (inB && inC) {
            // 跳过：相当于删除 x
            i++;
        } else {
            // 保留：覆写到 A[w]
            if (w != i) A->data[w] = x;
            w++; i++;
        }
    }
    A->size = w;
}
```

时间复杂度O(n+m+k)