数据结构作业 第二周

霍斌 PB24111627

我将所有代码封装在了$LinearList.h$中, 完整头文件请见附录

1 2.19

时间复杂度分析:

1. **第一个循环**: 时间复杂度为 $O(n_1)$
2. **第二个循环**: 时间复杂度为 $O(n_2)$
3. **删除节点循环**: 时间复杂度为$O(n_2 - n_1)$

综上, 时间复杂度为 $\boldsymbol{O(n)}$

```
bool deleteFromMin2Max(LinearList *list, int mink, int maxk) {
    //已知链表中元素以值递增有序排列
    if (list->head->data == 0 ||
         maxk < mink ||
        (list->head->next != NULL && list->head->next->data >= maxk)
    )
        return false; //处理异常情况
    Node *leftGapNode = list->head->next;

    while ( leftGapNode->next != NULL && leftGapNode->next->data <= mink) {
        leftGapNode = leftGapNode->next;
    }
    //如果一直到链表末端仍然没有人大于mink, 那么也没了
    if (leftGapNode->next == NULL) {
        return false;
    }

    Node *rightGapNode = leftGapNode->next;

    while (rightGapNode != NULL && rightGapNode->data < maxk) {
        rightGapNode = rightGapNode->next;
    }

    //删除leftGapNode和rightGapNode之间的所有节点
    Node *curNode = leftGapNode->next;
    while (curNode != rightGapNode) {
        Node *nodeToDelete = curNode;
        curNode = curNode->next;
        free(nodeToDelete);
        list->head->data--;
    }

    //最后弥合东西分裂
    leftGapNode->next = rightGapNode;
    return true;
}
```

2 2.21

```
bool reverse(LinearList *list) {
    if ( list->head->data <= 1 ) return false;


    Node *prev = NULL;
    Node *current = list->head->next; // 跳过头结点
    Node *next = NULL;

    while (current != NULL) {
        next = current->next; // 保存下一个节点
        current->next = prev; // 反转当前节点的指针
        prev = current;       // 移动prev和current指针
        current = next;
    }

    list->head->next = prev; // 更新头结点的next指针
    return true;
}
```

3 2.24

```
bool mergeListsByDecrease(LinearList *list1, LinearList *list2) {
    // 仍假设两个链表均为递增有序排列，合并后链表使用list1->head，并释放list2->head和list2
    if (!list1 || !list2 || !list1->head || !list2->head) return false;
    if (list1->head->data == 0 && list2->head->data == 0) return false; // 两个链表均为
空

    Node *dummy = (Node *)malloc(sizeof(Node));
    dummy->next = NULL;
    Node *tail = dummy;
    Node *cur1 = list1->head->next;
    Node *cur2 = list2->head->next;

    while (cur1 && cur2) {
        if (cur1->data <= cur2->data) {
            tail->next = cur1;
            cur1 = cur1->next;
        } else {
            tail->next = cur2;
            cur2 = cur2->next;
        }
        tail = tail->next;
    }
    if (cur1) tail->next = cur1;
    if (cur2) tail->next = cur2;

    // 更新list1的头结点
    list1->head->next = dummy->next;
    list1->head->data = list1->head->data + list2->head->data;

    // 释放dummy节点和list2的头结点及结构体
    free(dummy);
    // free(list2->head);
```

```
    // free(list2);
    reverse(list1);
    return true;
}
```

4 2.29

时间复杂度分析:

不妨设三个链表的长度分别为n_1, n_2, n_3.

1. **第一个循环**: 最坏情况是遍历整个2, 3链表, 也即$O(n_2 + n_3)$.
2. **第二个循环**: 无论如何最后也要遍历整个链表1, 但是总共只需遍历一次(并非和第一次循环倍乘, 而是叠加), 因此复杂度为$O(n_1)$.

综上, 复杂度为$O(n_1 + n_2 + n_3)$, 即$O(n)$.

```
bool deleteFrom1WhoBothAppearIn2and3(LinearList *list1, LinearList *list2, LinearList *list3) {
    //从表1中删去表2和表3中共有的值
    //使用双指针逐步得到BC的重复值，然后再删除A中对应的人
    if( list1->head->data == 0 || list2->head->data == 0 || list3->head->data == 0 ) return false; //有一个链表为空，那么就没有交集

    Node *prev = list1->head;
    Node *cur1 = list1->head->next;
    Node *next = NULL;
    Node *cur2 = list2->head->next;
    Node *cur3 = list3->head->next;


    int duplicate = 0;

    while (cur2 && cur3) {
        if (cur2->data < cur3->data) {
            cur2 = cur2->next;
        } else if (cur2->data > cur3->data) {
            cur3 = cur3->next;
        } else {    //如果找到重复值了，去往表1中删除对应元素
            duplicate = cur2->data;
            while(cur1 != NULL && cur1->data < duplicate) {
                prev = cur1;
                cur1 = cur1->next;

            } //找到了重复值
            if (cur1->data == duplicate) { //如果发现了重复值，那么删掉这个人
                next = cur1->next;
                free(cur1);
                cur1 = next;
                prev->next = cur1;
                list1->head->data--;
            }
            //如果cur1已经超过了重复值，那么需要使cur2或者cur3再动一步防止死循环
            if(cur1->data > duplicate) cur2 = cur2->next;
        }
    }
```

```
    return true;
}
```

附录

```
#ifndef LINEARLIST_H
#define LINEARLIST_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
//
typedef struct Node {
    int data;
    struct Node *next;
} Node;

typedef struct {
    Node *head;     // use head node to store length of list
} LinearList;


void initList(LinearList *list) {
    if (!list) return;

    list->head = (Node *)malloc(sizeof(Node));
    if (!list->head) {
        fprintf(stderr, "Memory allocation failed\n");
        return;
    }
    list->head->data = 0;
    list->head->next = NULL;
}

int getSize(const LinearList *list) {
    if (!list || !list->head) return 0;
    return list->head->data;
}

bool isValidIndex(const LinearList *list, int index, bool allowEnd) {
    if (!list || !list->head) return false;
    int size = list->head->data;
    if (allowEnd) {
        return index >= 0 && index <= size;
    } else {
        return index >= 0 && index < size;
    }
}

// 在指定位置插入元素
bool insert(LinearList *list, int index, int value) {
    if (!list || !list->head) {
        fprintf(stderr, "List not initialized\n");
        return false;
```

```
    }

    if (!isValidIndex(list, index, true)) {
        fprintf(stderr, "Index out of bounds\n");
        return false;
    }

    Node *newNode = (Node *)malloc(sizeof(Node));
    if (!newNode) {
        fprintf(stderr, "Memory allocation failed\n");
        return false;
    }

    newNode->data = value;
    Node *curNode = list->head;


    for (int i = 0; i < index; i++) {
        curNode = curNode->next;
    }

    newNode->next = curNode->next;
    curNode->next = newNode;
    list->head->data++;

    return true;
}


bool extract(LinearList *list, int index) {
    if (!list || !list->head) {
        fprintf(stderr, "List not initialized\n");
        return false;
    }

    if (!isValidIndex(list, index, false)) {
        fprintf(stderr, "Index out of bounds\n");
        return false;
    }

    Node *curNode = list->head;

    for (int i = 0; i < index; i++) {
        curNode = curNode->next;
    }

    Node *nodeToDelete = curNode->next;
    curNode->next = nodeToDelete->next;
    free(nodeToDelete);
    list->head->data--;

    return true;
}

// 获取指定位置的元素
int get(const LinearList *list, int index) {
```

```
    if (!list || !list->head) {
        fprintf(stderr, "List not initialized\n");
        return -1;
    }

    if (!isValidIndex(list, index, false)) {
        fprintf(stderr, "Index out of bounds\n");
        return -1;
    }

    Node *curNode = list->head->next;

    for (int i = 0; i < index; i++) {
        curNode = curNode->next;
    }

    return curNode->data;
}

// 打印链表
void printList(const LinearList *list) {
    if (!list || !list->head) {
        printf("List not initialized\n");
        return;
    }

    printf("List (size: %d): ", list->head->data);
    Node *curNode = list->head->next;

    while (curNode != NULL) {
        printf("%d", curNode->data);
        if (curNode->next != NULL) {
            printf(" -> ");
        }
        curNode = curNode->next;
    }
    printf("\n");
}

bool addToEnd(LinearList *list, int value) {
    if (!list || !list->head) return false;
    return insert(list, list->head->data, value);
}

bool addToFront(LinearList *list, int value) {
    return insert(list, 0, value);
}

// 删除最小值到最大值之间的所有元素(不包括端点)
bool deleteFromMin2Max(LinearList *list, int mink, int maxk) {
    //已知链表中元素以值递增有序排列
    if (list->head->data == 0 ||
         maxk < mink ||
        (list->head->next != NULL && list->head->next->data >= maxk)
    )
        return false; //处理异常情况
```

```
    Node *leftGapNode = list->head->next;

    while ( leftGapNode->next != NULL && leftGapNode->next->data <= mink) {
        leftGapNode = leftGapNode->next;
    }
    //如果一直到链表末端仍然没有人大于mink，那么也没了
    if (leftGapNode->next == NULL) {
        return false;
    }

    Node *rightGapNode = leftGapNode->next;

    while (rightGapNode != NULL && rightGapNode->data < maxk) {
        rightGapNode = rightGapNode->next;
    }

    //删除leftGapNode和rightGapNode之间的所有节点
    Node *curNode = leftGapNode->next;
    while (curNode != rightGapNode) {
        Node *nodeToDelete = curNode;
        curNode = curNode->next;
        free(nodeToDelete);
        list->head->data--;
    }

    //最后弥合东西分裂
    leftGapNode->next = rightGapNode;
    return true;
}

bool reverse(LinearList *list) {
    if ( list->head->data <= 1 ) return false;


    Node *prev = NULL;
    Node *current = list->head->next; // 跳过头结点
    Node *next = NULL;

    while (current != NULL) {
        next = current->next; // 保存下一个节点
        current->next = prev; // 反转当前节点的指针
        prev = current;       // 移动prev和current指针
        current = next;
    }

    list->head->next = prev; // 更新头结点的next指针
    return true;
}

bool mergeListsByDecrease(LinearList *list1, LinearList *list2) {
    // 仍假设两个链表均为递增有序排列，合并后链表使用list1->head，并释放list2->head和list2
    if (!list1 || !list2 || !list1->head || !list2->head) return false;
    if (list1->head->data == 0 && list2->head->data == 0) return false; // 两个链表均
为空

    Node *dummy = (Node *)malloc(sizeof(Node));
```

```
    dummy->next = NULL;
    Node *tail = dummy;
    Node *cur1 = list1->head->next;
    Node *cur2 = list2->head->next;

    while (cur1 && cur2) {
        if (cur1->data <= cur2->data) {
            tail->next = cur1;
            cur1 = cur1->next;
        } else {
            tail->next = cur2;
            cur2 = cur2->next;
        }
        tail = tail->next;
    }
    if (cur1) tail->next = cur1;
    if (cur2) tail->next = cur2;

    // 更新list1的头结点
    list1->head->next = dummy->next;
    list1->head->data = list1->head->data + list2->head->data;

    // 释放dummy节点和list2的头结点及结构体
    free(dummy);
    // free(list2->head);
    // free(list2);
    reverse(list1);
    return true;
}

bool deleteFrom1WhoBothAppearIn2and3(LinearList *list1, LinearList *list2, LinearList
*list3) {
    //从表1中删去表2和表3中共有的值
    //使用双指针逐步得到BC的重复值，然后再删除A中对应的人
    if( list1->head->data == 0 || list2->head->data == 0 || list3->head->data == 0 )
return false; //有一个链表为空，那么就没有交集

    Node *prev = list1->head;
    Node *cur1 = list1->head->next;
    Node *next = NULL;
    Node *cur2 = list2->head->next;
    Node *cur3 = list3->head->next;


    int duplicate = 0;

    while (cur2 && cur3) {
        if (cur2->data < cur3->data) {
            cur2 = cur2->next;
        } else if (cur2->data > cur3->data) {
            cur3 = cur3->next;
        } else {    //如果找到重复值了，去往表1中删除对应元素
            duplicate = cur2->data;
            while(cur1 != NULL && cur1->data < duplicate) {
                prev = cur1;
                cur1 = cur1->next;
```

```
271
272             } //找到了重复值
273             if (cur1->data == duplicate) { //如果发现了重复值，那么删掉这个人
274                 next = cur1->next;
275                 free(cur1);
276                 cur1 = next;
277                 prev->next = cur1;
278                 list1->head->data--;
279             }
280             //如果cur1已经超过了重复值，那么需要使cur2或者cur3再动一步防止死循环
281             if(cur1->data > duplicate) cur2 = cur2->next;
282         }
283     }
284
285     return true;
286 }
287 #endif // LINEARLIST_H
```