数据结构作业 第三周

霍斌 PB24111627

1 2.25

输入list1和list2, 返回一个新链表newList, 完整头文件见附录.

```
LinearList mergeListsByIncrease(LinearList *list1, LinearList *list2) {
    LinearList newList;
    initList(&newList);
    // 仍假设两个链表均为递增有序排列, 返回一个newList
    if (!list1 || !list2 || !list1->head || !list2->head) return newList;
    if (list1->head->data == 0 && list2->head->data == 0) return newList; // 两个链表均为空

    // 获取两个链表的第一个节点
    Node *cur1 = list1->head->next;
    Node *cur2 = list2->head->next;

    while(cur1 != NULL && cur2 != NULL) {
        if(cur1->data > cur2->data) {
            cur2 = cur2->next;
        } else if(cur1->data < cur2->data) {
            cur1 = cur1->next;
        } else {
            addToEnd(&newList, cur1->data);
            cur1 = cur1->next;
            cur2 = cur2->next;
        }
    }

    return newList;
}
```

2 2.38

LocateNode函数会搜索第一个匹配的节点, 并按照频率排序. 完整头文件见附录.

```
Node *LocateNode(Node* head, int data) {
    Node *cur = head->next;
    while (cur != head) {
        if (cur->data == data) {
            cur->freq++; // 更新访问频率

            // 每访问一次, 按频率对链表进行排序
            while(cur->prev != head && cur->freq > cur->prev->freq) {
                // 交换一次节点(冒泡排序)
                Node *prevNode = cur->prev;
                Node *nextNode = cur->next;

                nextNode->prev = prevNode;
                prevNode->next = nextNode;
                cur->prev = prevNode->prev;
```

```
                cur->next = prevNode;
                prevNode->prev->next = cur;
                prevNode->prev = cur;
            }
            return cur;
        }
        cur = cur->next;
    }
    return NULL;
}
```

3 3.6

根据栈后进先出的特性,不可能出现输入序列后进的数比先进的数先出栈的情况,所以不存在这样的$i < j < k$.

4 3.19

本函数用于判断是否匹配,完整头文件见附录.

```
// 提取出一个字符串的括号部分，并判断括号是否匹配
void matchPairs(Stack *stack, char* str) {
    if(!isEmpty(stack)) {
        perror("Stack is not empty, cannot deal.");
        return;
    }

    // 遇遇到() [] {} 就入栈，如果匹配到了对应的反括号则Pop，如果括号是匹配的，那么最内层会被消掉，进一步的外层括号也会逐层被消掉
    for(char* p = str; *p != '\0'; p++) {
        if(*p == '(' || *p == '[' || *p == '{') {
            push(stack, *p);
        } else if(*p == ')') {
            if(!isEmpty(stack) && peek(stack) == '(') {
                pop(stack);
            } else {
                printf("失败啦！ ");
                return;
            }
        } else if(*p == ']') {
            if(!isEmpty(stack) && peek(stack) == '[') {
                pop(stack);
            } else {
                printf("失败啦！ ");
                return;
            }
        } else if(*p == '}') {
            if(!isEmpty(stack) && peek(stack) == '{') {
                pop(stack);
            } else {
                printf("失败啦！ ");
                return;
            }
        }
    }

    // 如果栈为空，则说明匹配成功
```

```
    if(isEmpty(stack)) {
        printf("成功啦！");
        return;
    } else {
        printf("失败啦！");
    }
}
```

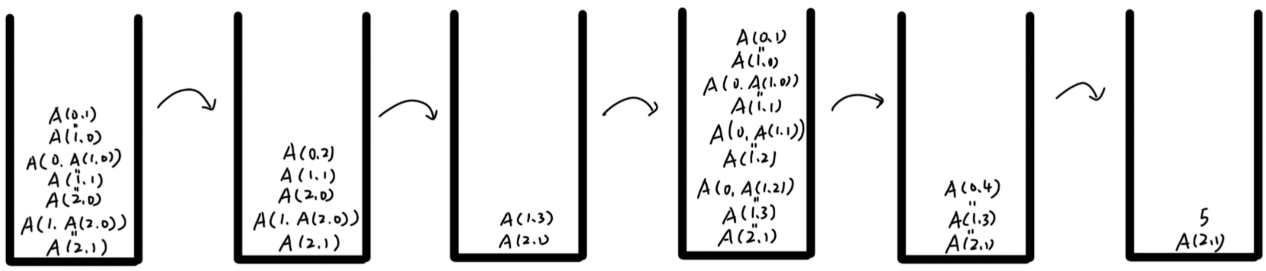
5 3.27(1)

5.1 算法

分别实现akm函数的递归和非递归(栈)实现, akm函数使用了IntStack.h, 完整头文件见附录.

```
// 递归算法
int akm_recursion(int m, int n) {
    if(m == 0) {
        return n + 1;
    } else if(m != 0 && n == 0) {
        return akm_recursion(m - 1, 1);
    } else if(m != 0 && n != 0) {
        return akm_recursion(m - 1, akm_recursion(m, n - 1));
    }

    return -1;
}

// 非递归算法
int akm_human(int m, int n) {
    IntStack* s = createIntStack();
    push(s, m);

    while(!isEmpty(s)) {
        m = peek(s);
        pop(s);

        if (m == 0) {
            n = n + 1;
        } else if (n == 0) {
            push(s, m - 1);
            n = 1;
        } else {
            push(s, m - 1);
            push(s, m);
            n = n - 1;
        }
    }
    return n;
}
```

5.2 绘图



如图所示, 需要依次计算从栈顶到栈底的Akm函数, 直到栈清空, 最后会得到A(2,1) = 5.

附录

1 LinearList

```
#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;
} LinearList;


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

    list->head = (Node *)malloc(sizeof(Node));
    if (!list->head) {
        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) {
        return false;
    }

    if (!isValidIndex(list, index, true)) {
        return false;
    }

    Node *newNode = (Node *)malloc(sizeof(Node));
    if (!newNode) {
        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) {
        return false;
    }

    if (!isValidIndex(list, index, false)) {
        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;
```

```c
}

// 获取指定位置的元素
int get(const LinearList *list, int index) {
    if (!list || !list->head) {
        return -1;
    }

    if (!isValidIndex(list, index, false)) {
        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) {
        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)
    )
```

```c
        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;
}

LinearList mergeListsByIncrease(LinearList *list1, LinearList *list2) {
    LinearList newList;
    initList(&newList);
    // 仍假设两个链表均为递增有序排列，返回一个newList
    if (!list1 || !list2 || !list1->head || !list2->head) return newList;
    if (list1->head->data == 0 && list2->head->data == 0) return newList; // 两个链表
均为空

    // 获取两个链表的第一个节点
    Node *cur1 = list1->head->next;
    Node *cur2 = list2->head->next;

    while(cur1 != NULL && cur2 != NULL) {
        if(cur1->data > cur2->data) {
            cur2 = cur2->next;
        } else if(cur1->data < cur2->data) {
            cur1 = cur1->next;
        } else {
            addToEnd(&newList, cur1->data);
            cur1 = cur1->next;
            cur2 = cur2->next;
        }
    }

    return newList;
```

```
}

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;
}


#endif // LINEARLIST_H

```

2 DoublyCircularLinkedList

```
#ifndef DOUBLY_CIRCULAR_LINKED_LIST_H
#define DOUBLY_CIRCULAR_LINKED_LIST_H

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

```c
#include <stdbool.h>

typedef struct Node {
    int data;
    int freq; // 访问频率
    struct Node *prev;
    struct Node *next;
} Node;

// 创建链表头节点
Node *CreateList() {
    Node *head = (Node *)malloc(sizeof(Node));
    head->next = head;
    head->prev = head;
    head->data = 0; // 用于存储链表长度
    head->freq = 0; // 用于存储访问频率
    return head;
}

void InsertNode(Node* head, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->freq = 0; // 访问频率初始化为0
    newNode->prev = head->prev;
    newNode->next = head;
    head->prev->next = newNode;
    head->prev = newNode;
    head->data++;
}

Node *LocateNode(Node* head, int data) {
    Node *cur = head->next;
    while (cur != head) {
        if (cur->data == data) {
            cur->freq++; // 更新访问频率

            // 每访问一次，按频率对链表进行排序
            while(cur->prev != head && cur->freq > cur->prev->freq) {
                // 交换一次节点(冒泡排序)
                Node *prevNode = cur->prev;
                Node *nextNode = cur->next;

                nextNode->prev = prevNode;
                prevNode->next = nextNode;
                cur->prev = prevNode->prev;
                cur->next = prevNode;
                prevNode->prev->next = cur;
                prevNode->prev = cur;
            }
            return cur;
        }
        cur = cur->next;
    }
    return NULL;
}

```

```c
void PrintList(Node* head) {
    Node* cur = head->next;
    while (cur != head) {
        printf("Data: %d, Freq: %d\n", cur->data, cur->freq);
        cur = cur->next;
    }
}

#endif // DOUBLY_CIRCULAR_LINKED_LIST_H
```

3 CharStack(括号匹配)

```c
#ifndef STACK_H
#define STACK_H

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

#define MAXSIZE 100
typedef struct Stack {
    int top;
    char data[MAXSIZE];
} Stack;

Stack* createStack();
bool isFull(Stack *stack);
bool isEmpty(Stack *stack);
void push(Stack *stack, char value);
void pop(Stack *stack);
char peek(Stack *stack);

Stack* createStack() {
    Stack *stack = (Stack*)malloc(sizeof(Stack));
    stack->top = -1;
    return stack;
}

bool isFull(Stack *stack) {
    return stack->top == MAXSIZE - 1;
}

bool isEmpty(Stack *stack) {
    return stack->top == -1;
}

void push(Stack *stack, char value) {
    if(!isFull(stack)) {
        stack->data[stack->top + 1] = value;
        stack->top++;
    } else {
        perror("StackOverflow");
    }
}

void pop(Stack *stack) {
```

```c
    if(!isEmpty(stack)) {
        stack->top--;
    } else {
        perror("Stack is Empty");
    }
}

char peek(Stack *stack) {
    if(!isEmpty(stack)) {
        return stack->data[stack->top];
    } else {
        perror("Stack is Empty");
        return '\0';
    }
}

// 提取出一个字符串的括号部分，并判断括号是否匹配
void matchPairs(Stack *stack, char* str) {
    if(!isEmpty(stack)) {
        perror("Stack is not empty, cannot deal.");
        return;
    }

    // 遇遇到() [] {} 就入栈，如果匹配到了对应的反括号则Pop，如果括号是匹配的，那么最内层会被消掉，进一步的外层括号也会逐层被消掉
    for(char* p = str; *p != '\0'; p++) {
        if(*p == '(' || *p == '[' || *p == '{') {
            push(stack, *p);
        } else if(*p == ')') {
            if(!isEmpty(stack) && peek(stack) == '(') {
                pop(stack);
            } else {
                printf("失败啦！");
                return;
            }
        } else if(*p == ']') {
            if(!isEmpty(stack) && peek(stack) == '[') {
                pop(stack);
            } else {
                printf("失败啦！");
                return;
            }
        } else if(*p == '}') {
            if(!isEmpty(stack) && peek(stack) == '{') {
                pop(stack);
            } else {
                printf("失败啦！");
                return;
            }
        }
    }

    // 如果栈为空，则说明匹配成功
    if(isEmpty(stack)) {
        printf("成功啦！");
        return;
```

```c
    } else {
        printf("失败啦！ ");
    }
}
#endif // STACK_H
```

4 IntStack(Ackerman使用了该文件)

```c
#ifndef INTSTACK_H
#define INTSTACK_H

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

#define MAXSIZE 10000
typedef struct IntStack {
    int top;
    int data[MAXSIZE];
} IntStack;

IntStack* createIntStack();
bool isFull(IntStack *stack);
bool isEmpty(IntStack *stack);
void push(IntStack *stack, int value);
void pop(IntStack *stack);
int peek(IntStack *stack);

IntStack* createIntStack() {
    IntStack *stack = (IntStack*)malloc(sizeof(IntStack));
    stack->top = -1;
    return stack;
}

bool isFull(IntStack *stack) {
    return stack->top == MAXSIZE - 1;
}

bool isEmpty(IntStack *stack) {
    return stack->top == -1;
}

void push(IntStack *stack, int value) {
    if(!isFull(stack)) {
        stack->data[stack->top + 1] = value;
        stack->top++;
    } else {
        perror("StackOverflow");
    }
}

void pop(IntStack *stack) {
    if(!isEmpty(stack)) {
        stack->top--;
    } else {
        perror("Stack is Empty");
```

```c
    }
}

int peek(IntStack *stack) {
    if(!isEmpty(stack)) {
        return stack->data[stack->top];
    } else {
        perror("Stack is Empty");
        return '\0';
    }
}

#endif // STACK_H
```

5 Ackerman(递归&非递归)

```c
#ifndef ACKERMAN_H
#define ACKERMAN_H

#include "IntStack.h"
#include <stdio.h>

// 递归法
int akm_recursion(int m, int n) {
    if(m == 0) {
        return n + 1;
    } else if(m != 0 && n == 0) {
        return akm_recursion(m - 1, 1);
    } else if(m != 0 && n != 0) {
        return akm_recursion(m - 1, akm_recursion(m, n - 1));
    }

    return -1;
}

// 非递归(用栈模拟)
int akm_human(int m, int n) {
    IntStack* s = createIntStack();
    push(s, m);

    while(!isEmpty(s)) {
        m = peek(s);
        pop(s);

        if (m == 0) {
            n = n + 1;
        } else if (n == 0) {
            push(s, m - 1);
            n = 1;
        } else {
            push(s, m - 1);
            push(s, m);
            n = n - 1;
        }
    }
    return n;
```

```
}

#endif // ACKERMAN_H
```