### 上机实验1：链表

### 班级：2022211305 学号：2022211119 姓名：赵宇鹏 email:zhaoyupeng@bupt.edu.cn

1. 实验内容：完成了用带头结点的单向链表实现实验所要求的所有函数功能，同时完成了用不带头节点的双向循环链表实现以上功能。
2. 算法与源程序结构：

对带头结点的单向链表：创建一个带头结点的单向链表，通过strcmp函数来进行相应的指令操作，以头节点为哨兵节点，此后从哨兵节点之后开始对链表进行依次操作。

插入函数从小到大从前往后进行遍历，直到为NULL或遇到第一个比需插入数据大的数，将储存需被插入的数据的节点插入进去，即可完成。

翻转函数需要先保存下一个节点的指针，然后将当前节点的next指针反转，更新prev为当前节点，更新下一个节点，往后遍历实现。

其余函数依此类推，皆为从头到尾进行遍历进行操作。

对双向循环链表：由于为双向循环链表，所以进行奇数次翻转时可以直接由后往前进行操作，因此打印操作和翻转操作可直接从后往前运行，其余函数与单向链表类似。

1. 实验总结：完成此作业大概花费了四个小时。在完成单向链表时，对翻转函数的构想花费较多时间，同时，完成翻转操作后也会把哨兵节点一起翻转，导致输出结果错误，后面的sum函数也一起发生错误，后续直接增设一个新的哨兵头结点，原来那个头节点直接跳过，达到翻转目的。此处还有一个问题：在翻转链表后如果再插入数据应该如何插入，是仍然按照从小到大放，还是从大到小放？由于题目没有明确指出，代码按照仍然从小到大放。

完成双向循环链表时，本人没有设置哨兵头结点，导致插入和删除功能在遍历到最后时跳出界限没有分清，产生了如删除节点后前置节点没有更改，插入节点时会把后方数据覆盖等bug后增设了许多条件判断语句以改善。

### 单向链表

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

struct Node{

    int data;

    struct Node\* next;

};

//创建链表

struct Node\* create(){

    struct Node\* head = (struct Node\*)malloc(sizeof(struct Node));

    head->next = NULL;

    head->data = -10000;

    return head;

}

//插入节点

void insert(struct Node\* head, int num){

    struct Node\* temp,\* current;

    temp = (struct Node\*)malloc(sizeof(struct Node));

    temp->data = num;

    if(head->next == NULL){

        head->next = temp;

        temp->next = NULL;

    }else{

        current = head;

        if(num > current->next->data){

            while(current->next !=NULL && current->next->data < num){

                current = current->next;

            }

            temp->next = current->next;

            current->next = temp;

        }else{

            temp->next = current->next;

            current->next = temp;

        }

    }

}//此处存在一个问题，若在翻转的时候再插入一个数据，应该如何插入？是按从大到小还是继续从小到大

//删除节点

void delete(struct Node\* head, int num){

    struct Node\* temp;

    temp = head;

    int delete\_flag = 1;

    while (temp->next != NULL){

        if(temp->next->data == num){

            temp->next = temp->next->next;

            delete\_flag = 0;

        }else{

            temp = temp->next;

        }

    }

    if(delete\_flag == 1){

        printf("ERROR\n");

    }

}

// 翻转单向链表,由于设置了哨兵节点，此处翻转的时候注意不能翻转该节点，由于本函数算法限制，改为直接在头节点加一个哨兵节点。

struct Node\* revert(struct Node\* head){

    struct Node\* pre = NULL;

    struct Node\* current = head->next;

    struct Node\* next = NULL;

    while (current != NULL) {

        next = current->next;  // 保存下一个节点的指针

        current->next = pre;  // 将当前节点的next指针反转

        pre = current;        // 更新prev为当前节点

        current = next;        // 更新current为下一个节点

    }

    struct Node\* temp;

    temp = (struct Node\*)malloc(sizeof(struct Node));

    temp->data = -10000;

    temp->next = pre;

    return temp; // 返回新的头节点

}

//打印

void dump(struct Node\* head){

    struct Node\* current;

    current = head->next;

    if(head->next == NULL){

        printf("()");

    }else{

        printf("( ");

        while(current != NULL){

            printf("%d ",current->data);

            current = current->next;

        }

        printf(")\n");

    }

}

//检验是否为从小到大顺序

int verify(struct Node\* head){

    struct Node\* current;

    if(head->next == NULL){

        return 0;  //返回0代表是从小到大

    }else{

        current = head->next;

        while(current != NULL && current->next != NULL){

            if(current->data <= current->next->data){

                current = current->next;

            }else{

                return 1;

            }

        }

        return 0;

    }

}

//计数功能

int count(struct Node\* head){

    struct Node\* current;

    if(head->next == NULL){

        return 0;

    }else{

        int num = 0;

        current = head->next;

        while(current != NULL){

            num++;

            current = current->next;

        }

        return num;

    }

}

//求和功能

int sum(struct Node\* head){

    if(head->next == NULL){

        return 0;

    }else{

        struct Node\* current;

        current = head->next;

        int num = 0;

        while(current != NULL){

            num = num + current->data;

            current = current->next;

        }

        return num;

    }

}

void destroy(struct Node\* head){

    struct Node\* current;

    struct Node\* temp;

    current = head->next;

    temp = head;

    while(current != NULL){

        free(temp);

        temp = current;

        current = current->next;

    }

}

int main()

{

    int num;

    char s[10] = {0};

    scanf("%s",s);

    while(strcmp(s,"create") != 0){

        printf("ERROR\n");

        scanf("%s",s);

    }

    struct Node\* head = create();

    scanf("%s",s);

        while(strcmp(s,"bye")){

            if(strcmp(s,"insert") == 0){

                scanf("%d",&num);

                insert(head,num);

            }

            else if(strcmp(s,"delete") == 0){

                scanf("%d",&num);

                delete(head,num);

            }

            else if(strcmp(s,"revert") == 0){

                head = revert(head);

            }

            else if(strcmp(s,"dump") == 0){

                dump(head);

            }

            else if(strcmp(s,"verify") == 0){

                num = verify(head);

                if(num == 0){

                    printf("YES\n");

                }else{

                    printf("NO\n");

                }

            }

            else if(strcmp(s,"count") == 0){

                num = count(head);

                printf("%d\n",num);

            }

            else if(strcmp(s,"sum") == 0){

                num = sum(head);

                printf("%d\n",num);

            }

            else if(strcmp(s,"destroy") == 0){

                destroy(head);

            }

            else if(strcmp(s,"creat") == 0){

                printf("ERROR\n");

            }

            else{

                printf("ERROR\n");

            }

            scanf("%s",s);

        }

    return 0;

}

### 双向循环链表

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

struct Node{

    int data;

    struct Node\* pre;

    struct Node\* next;

};

//创建双向循环链表

struct Node\* create(){

    struct Node\* head = (struct Node\*)malloc(sizeof(struct Node));

    head->next = head;

    head->pre = head;

    head->data = -10000;

    return head;

}

//插入节点,由于没带头节点，所以情况分类比较多

void insert(struct Node\* head, int num){

    struct Node\* temp,\* current,\* prev;

    temp = (struct Node\*)malloc(sizeof(struct Node));

    temp->data = num;

    if(head->data == -10000){

        head->data = num;

    }else{

            if(head->next == head){

                    head->next = temp;

                    temp->next = head;

                    head->pre = temp;

                    temp->pre = head;

                    if(num < head->data){

                            temp->data = head->data;

                            head->data = num;

                    }

            }else{

                    if(head->data > num){

                            temp->next = head->next;

                            temp->pre = head;

                            head->next->pre = temp;

                            head->next = temp;

                            temp->data =head->data;

                            head->data=num;

                    }else{

                            current = head;

                            while(current->next != head && current->next->data < num){

                                    current = current->next;

                            }

                            temp->next = current->next;

                            current->next->pre = temp;

                            current->next = temp;

                            temp->pre = current;

                    }

                }

    }

}//此处存在一个问题，若在翻转的时候再插入一个数据，应该如何插入？是按从大到小还是继续从小到大

//删除节点

struct Node\* delete(struct Node\* head, int num){

    struct Node\* temp;

    if(head->next == head && head->data == num)

    {

        head->data = -10000;

        return head;

    }

    int delete\_flag = 1;

        while(head->data == num){

                head = head->next;

        }

        temp = head;

    do{

        while(temp->next->data == num){

            temp->next = temp->next->next;

            temp->next->pre = temp;

            delete\_flag = 0;

        }//遍历，把头节点处应该删除的节点全部删除

        if(temp->next->data != num){

            temp = temp->next;

        }

    }while (temp != head);

    if(delete\_flag == 1){

        printf("ERROR\n");

    }

    return head;

}

// 翻转双向循环链表，此处可以取巧，若翻转则直接返回标识1，然后后面的dump以及verufy函数根据返回的1来进行反向操作，但此方法有个缺点，若DUMP后再进行插入，则verify无法正常工作

int revert(int isrevert\_flag){

        if(isrevert\_flag == 1){

                return 0;

        }

        if(isrevert\_flag == 0){

                return 1;

        }

}

//打印

void dump(struct Node\* head,int isrevert\_flag){

    struct Node\* current;

        if(isrevert\_flag == 0){

                current = head->next;

                if(head->data == -10000){

                        printf("()\n");

                }else{

                        printf("( %d ",head->data);

                        while(current != head){

                                printf("%d ",current->data);

                                current = current->next;

                        }

                        printf(")\n");

                }

        }else{

                current = head->pre->pre;

                if(head->data == -10000){

                        printf("()\n");

                }else{

                        printf("( %d ",head->pre->data);

                        while(current != head->pre){

                                printf("%d ",current->data);

                                current = current->pre;

                        }

                        printf(")\n");

                }

        }

}

//检验是否为从小到大顺序

int verify(struct Node\* head,int isrevert\_flag){

    struct Node\* current;

    if(head->data == -10000 || head->next == head){

        return 0;  //返回0代表是从小到大

    }else{

             if(isrevert\_flag == 0){

                    current = head;

                    while(current->next != head){

                            if(current->data <= current->next->data){

                                    current = current->next;

                            }else{

                                    return 1;

                            }

                        }

                    return 0;

                }//若不翻转

                else{

                    current = head->pre;

                    while(current->pre != head->pre){

                            if(current->data <= current->pre->data){

                                        current = current->pre;

                            }else{

                                    return 1;

                            }

                        }

                    return 0;

                }//若已经翻转

    }

}

//计数功能

int count(struct Node\* head){

    struct Node\* current;

    if(head->data == -10000){

        return 0;

    }else{

        int num = 1;

        current = head;

        while(current->next != head){

            num++;

            current = current->next;

        }

        return num;

    }

}

//求和功能

int sum(struct Node\* head){

    if(head->data == -10000){

        return 0;

    }

        else if(head->next == head){

                return head->data;

        }else{

        struct Node\* current;

        current = head;

        int num = head->pre->data;

        while(current->next != head){

            num = num + current->data;

            current = current->next;

        }

        return num;

    }

}

void destroy(struct Node\* head) {

    if (head == NULL) {

        return;

    }

    struct Node\* current = head;

    struct Node\* temp;

    // 从头节点开始遍历链表

    do {

        temp = current;

        current = current->next;

        free(temp);

    } while (current != head);

}

int main()

{

    int num;

        int isrevert\_flag = 0;

    char s[10] = {0};

    scanf("%s",s);

    while(strcmp(s,"create") != 0){

        printf("ERROR\n");

        scanf("%s",s);

    }

    struct Node\* head = create();

    scanf("%s",s);

        while(strcmp(s,"bye")){

            if(strcmp(s,"insert") == 0){

                scanf("%d",&num);

                insert(head,num);

            }

            else if(strcmp(s,"delete") == 0){

                scanf("%d",&num);

                head = delete(head,num);

            }

            else if(strcmp(s,"revert") == 0){

                isrevert\_flag = revert(isrevert\_flag);

            }

            else if(strcmp(s,"dump") == 0){

                dump(head,isrevert\_flag);

            }

            else if(strcmp(s,"verify") == 0){

                num = verify(head,isrevert\_flag);

                if(num == 0){

                    printf("YES\n");

                }else{

                    printf("NO\n");

                }

            }

            else if(strcmp(s,"count") == 0){

                num = count(head);

                printf("%d\n",num);

            }

            else if(strcmp(s,"sum") == 0){

                num = sum(head);

                printf("%d\n",num);

            }

            else if(strcmp(s,"destroy") == 0){

                destroy(head);

            }

            else if(strcmp(s,"creat") == 0){

                printf("ERROR\n");

            }

            else{

                printf("ERROR\n");

            }

            scanf("%s",s);

        }

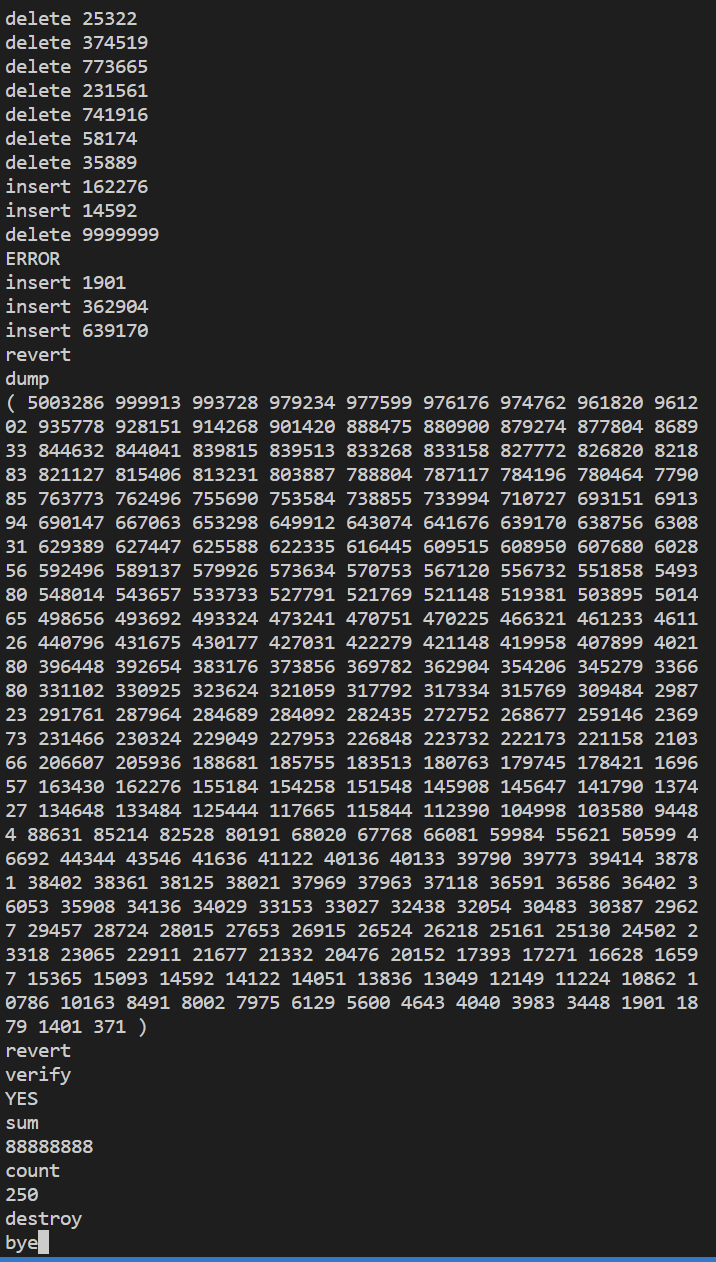
    return 0;

}

5、

单向链表：

测试数据1： 2、

create

insert 100

insert 200

insert 100

insert 90

insert 150

dump

( 90 100 100 150 200 )

count

5

sum

640

revert

dump

( 200 150 100 100 90 )

verify

NO

revert

verify

YES

delete 250

ERROR

delete 200

dump

( 90 100 100 150 )

sum

440

delete 100

delete 90

delete 150

count

0

verify

YES

revert

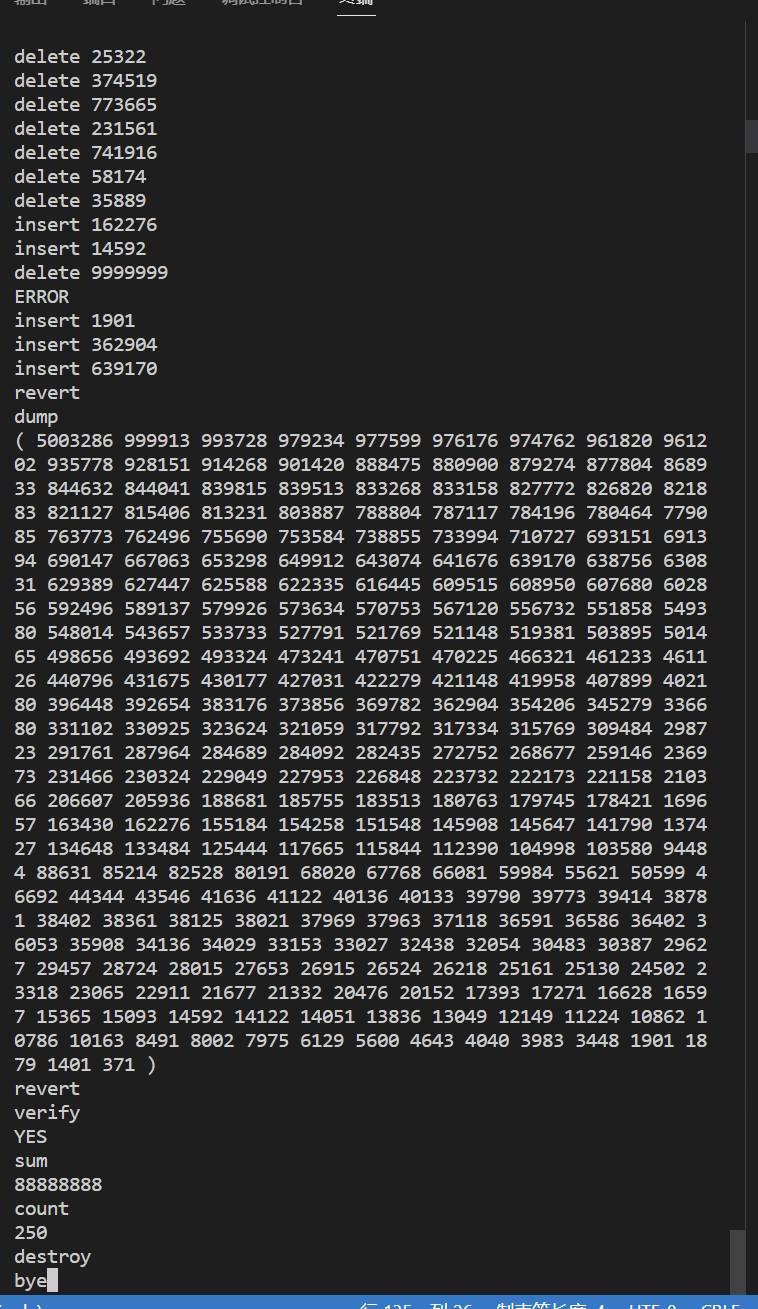
sum

0

destroy

Bye

双向循环链表 2、

1：

create

insert 100

insert 200

insert 100

insert 90

insert 150

dump

( 90 100 100 150 200 )

count

5

sum

640

revert

dump

( 200 150 100 100 90 )

verify

NO

revert

verify

YES

delete 250

ERROR

delete 200

dump

( 90 100 100 150 )

sum

440

delete 100

delete 90

delete 150

count

0

verify

YES

revert

sum

0

destroy

Bye