数据结构作业

第三章

3.3

`stack`

3.7

定义: G = B*C ， H = G/D , I = A-H, J = E^F , K = I+J

步骤	**OPTR**栈	**OPND**栈	输入字符	主要操作
1	#		A-B*C/D+E^F#	PUSH(OPND, A)
2	#	A	-B*C/D+E^F#	PUSH(OPTR, -)
3	#-	A	B*C/D+E^F#	PUSH(OPND, B)
4	#-	AB	*C/D+E^F#	PUSH(OPTR, *)
5	#-*	AB	C/D+E^F#	PUSH(OPND, C)
6	#-*	ABC	/D+E^F#	OPERATE(B, *, C)
7	#-	AG	/D+E^F#	PUSH(OPTR, /)
8	#-/	AG	D+E^F#	PUSH(OPND, D)
9	#-/	AGD	+E^F#	OPERATE(G, /, D)
10	#-	AH	+E^F#	OPERATE(A, -, H)
11	#	I	+E^F#	PUSH(OPTR, +)
12	#+	I	E^F#	PUSH(OPND, E)
13	#+	IE	^F#	PUSH(OPTR, ^)
14	#+^	IE	F#	PUSH(OPND, F)
15	#+^	IEF	#	OPERATE(E, ^, F)
16	#+	IJ	#	OPERATE(I, +, J)
17	#	K	#	RETURN(K)

3.10

```
void test(int &sum){
    int x;
    while(1){
        scanf(x);
        if (x==0){
            break;
        }
    }
    sum = 0;
    printf(sum);
}
```

3.17

```
#include "stack.h"          //附在本章作业最后

void main(){
    //生成字符串
    Stack S = stack_create();
    TYPE str[] = "566dsa&asd665@213";
    printf("The string is:\n%s", str);
    //减一去掉字符串末尾的\000
    int len = sizeof(str) / sizeof(char) - 1;
    //对字符串进行判断
    //设置标志tag, 当读到&自增
    //tag == 0, push, tag == 1, pop, tag > 1, stop
    int tag = 0;
    for (int i=0; i<len; i++){
        //读到标志符
        if (str[i] == '&'){
            tag++;
        }
        //结束
        else if (str[i] == '@'){
            break;
        }
        //压栈
        else if (tag == 0){
            stack_push(&S, str[i]);
        }
        //退栈, 保证栈非空
        else if (tag == 1 && !stack_empty(S)){
            if (stack_get_top(&S) == str[i]){
                stack_pop(&S);
            }
            else{
                break;
            }
        }
        else{
            //为了避免栈空时形如 str&rts**** 的串符合要求, 额外压入一个空字符
            stack_push(&S, ' ');
            break;
        }

    }
    //结果判断
    if (stack_empty(S)){
        printf("The string meets the requirements.\n");
    }
    else{
        printf("The string does not meet the requirements.\n");
    }
    return;
}
```

3.20

```c
#include <stdio.h>
#include <time.h>           //提供time()函数的原型
#include <stdlib.h>         //提供rand(),srand()函数的原型

#define COLOR_RANGE 2
#define WIDTH 10
#define HEIGHT 20
//生成随机数
#define random rand() % COLOR_RANGE

void trans_color(int x, int y, int color_before, int color_trans, int* graph);
void create_graph(int* graph);
void show_graph(int* graph);
int rand_int(int);

void main(){
    //生成图片
    int* graph = (int*)malloc(sizeof(int) * WIDTH * HEIGHT);
    create_graph(graph);
    printf("The graph before:\n");
    show_graph(graph);

    //设定点(x,y)
    int x = 0;
    int y = 3;
    int color_before = *(graph + x*HEIGHT + y);
    int color_trans;
    if (color_before == COLOR_RANGE - 1){
        color_trans = 0;
    }
    else{
        color_trans = color_before + 1;
    }

    //改变颜色
    trans_color(x, y, color_before, color_trans, graph);

    //显示修改后图片
    printf("The graph after trans:\n");
    show_graph(graph);
    return;
}

void trans_color(int x, int y, int color_before, int color_trans, int* graph){
    //判断边界
    if (x >= 0 && x < WIDTH && y >=0 && y < HEIGHT){
        //判断颜色
        if (*(graph + x*HEIGHT + y) == color_before){
            *(graph + x*HEIGHT + y) = color_trans;
            //递归吧少年
            trans_color(x+1, y, color_before, color_trans, graph);
            trans_color(x-1, y, color_before, color_trans, graph);

            trans_color(x, y+1, color_before, color_trans, graph);
```

```
            trans_color(x, y-1, color_before, color_trans, graph);
        }
    }
}

void create_graph(int* graph){
    //随机给像素点赋值
    srand(time(NULL));
    for (int i=0; i < WIDTH; i++){
        for (int j=0; j < HEIGHT; j++){
            *(graph + i*HEIGHT + j) = random;
        }
    }
}

void show_graph(int* graph){
    for (int i=0; i < WIDTH; i++){
        for (int j=0; j < HEIGHT; j++){
            printf("%d ", *(graph + i*HEIGHT + j));
        }
        printf("\n");
    }
}
```

3.21

```
#include "stack.h"
#include <string.h>

void opnd_do(TYPE, Stack*);
int opnd_value(TYPE);

void main(){
    //生成通常表达式
    TYPE expression[] = "a+b*c-d/e*f";
    Stack opnd_stack = stack_create();
    //用指针来循环表达式
    TYPE* the_char = expression;
    while(*the_char){
        //当是运算符
        if (*the_char == '+' || *the_char == '-' || *the_char == '*' || *the_char == '/'){
            //进行操作并自增指针
            opnd_do(*the_char++, &opnd_stack);
        }
        //当是字母
        else if ((*the_char-'a'>=0 && *the_char-'z'<=0) || (*the_char-'A'>=0 && *the_char-
'Z'<=0)){
            //输出字母并自增指针
            printf("%c", *the_char++);
        }
    }
    //退空算符栈
    while (!stack_empty(opnd_stack)){
        printf("%c", stack_pop(&opnd_stack));
    }
    return;
}

void opnd_do(TYPE the_char, Stack* opnd_stack){
    //当前算符优先级高于栈顶
    if (opnd_value(the_char) > opnd_value(stack_get_top(opnd_stack))){
        stack_push(opnd_stack, the_char);
    }
    else{
        //退栈并输出字符，直至当前算符优先级高于栈顶
        printf("%c", stack_pop(opnd_stack));
        opnd_do(the_char, opnd_stack);
    }
}

int opnd_value(TYPE c){
    if (c == '*' || c == '/'){
        return 2;
    }
    else if (c == '+' || c == '-'){
        return 1;
    }
    else{
        return 0;
```

```
    }
}
```

3.25

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

int func(int n);
int func_no_rec(int n);

void main(){
    int n = 92;
    printf("func with recursion:\t%d\n", func(n));
    printf("func without recursion:\t%d\n", func_no_rec(n));
    return;
}

//会有一定程度的截断
int func(int n){
    if (n < 0){
        printf("input should be a positive number\n");
        exit(-1);
    }
    else if (n == 0){
        return n + 1;
    }
    else{
        return n * func(n/2);
    }
}

int func_no_rec(int n){
    if (n < 0){
        printf("input should be a positive number\n");
        exit(-1);
    }
    int product = 1;
    while (n){
        product *= n;
        n /= 2;
    }
    return product;
}
```

3.31

```c
#include "stack.h"              //附在本章作业最后

void main(){
    //生成字符串
    Stack S = stack_create();
    TYPE str[] = "5dhjk44kjhd5@";
    printf("The string is:\n%s\n", str);
    int len = sizeof(str) / sizeof(char) - 1;
    //求出中点位置
    int half = (len-1)/2;

    for (int i=0; i<len; i++){
        //读到标志符
        if (str[i] == '@'){
            break;
        }
        //前一半压栈
        else if (i < half){
            stack_push(&S, str[i]);
        }
        //刚好中间点，判断奇偶
        else if (i == half){
            if (len % 2){
                stack_pop(&S);
            }
        }
        //退栈，保证栈非空
        else if (i > half && !stack_empty(S)){
            if (stack_get_top(&S) == str[i]){
                stack_pop(&S);
            }
            else{
                break;
            }
        }
        else{
            break;
        }

    }
    //结果判断
    if (stack_empty(S)){
        printf("The string meets the requirements.\n");
    }
    else{
        printf("The string does not meet the requirements.\n");
    }
    return;
}
```

附`stack.h`

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

#define MAX_SIZE        100
#define INCREASE        10

#define MALLOC          -1
#define EMPTYSTACK      -2

#ifndef TYPE
    #define RELOADTAG   0
    #define TYPE char
#else
    #ifndef RELOADTAG
    #define RELOADTAG   1
    #endif
#endif

typedef struct{
    TYPE* base;
    TYPE* top;
    int size;
}Stack;

Stack stack_create();
void stack_push(Stack*, TYPE);
TYPE stack_pop(Stack* S);
int stack_empty(Stack);
TYPE stack_get_top(Stack* S);

Stack stack_create(){
    Stack S;
    S.base = (TYPE*)malloc(sizeof(TYPE) * MAX_SIZE);
    if (S.base == NULL){
        printf("Fail to create List, storage allocation error\n");
        exit(MALLOC);
    }
    S.top = S.base;
    S.size = MAX_SIZE;
    return S;
}

void stack_push(Stack* S, TYPE data){
    if (S->top - S->base >= S->size){
        S->size += INCREASE;
        S->base = (TYPE*)realloc(S->base, sizeof(TYPE)*S->size);
        if (S->base == NULL){
            printf("Fail to push, storage allocation error\n");
            exit(MALLOC);
        }
    }
    *((S->top)++) = data;
}
```

```
TYPE stack_pop(Stack* S){
    if (stack_empty(*S)){
        printf("Fail to pop, empty stack\n");
        exit(EMPTYSTACK);
    }
    return *(--(S->top));
}

int stack_empty(Stack S){
    if (S.top == S.base){
        return 1;
    }
    else{
        return 0;
    }
}

TYPE stack_get_top(Stack* S){
    return *(S->top-1);
}
```