数据结构实验四

姓名：刘俊傲　　学号：**U201617047**　　班级：软工**1603**班

1. Radix sort

1. 问题描述：

    1. 实现桶式排序
    2. 实现基于桶式排序的基数排序

2. 问题分析与算法设计

    1. 桶式排序：把数组中的所有元素分为若干个数据块，也就是若干个桶，然后对每个桶里的数据进行排序，最后将所有桶里的数据依次排列即可
    2. 基数排序：将整数按位数切割成不同的数字，然后按每个位数分别比较

3. 算法实现：

    桶式排序：

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

typedef struct node {
    int key;
    struct node * next;
}KeyNode;

void inc_sort(int keys[], int size, int bucket_size) {
    KeyNode **bucket_table = (KeyNode **)malloc(bucket_size * sizeof(KeyNode *));
    for (int i = 0; i < bucket_size; i++) {
        bucket_table[i] = (KeyNode *)malloc(sizeof(KeyNode));
        bucket_table[i]->key = 0; //记录当前桶中的数据量
        bucket_table[i]->next = NULL;
    }
    for (int j = 0; j < size; j++) {
        KeyNode *node = (KeyNode *)malloc(sizeof(KeyNode));
        node->key = keys[j];
        node->next = NULL;
        //映射函数计算桶号
        int index = keys[j] / 10;
        //初始化P成为桶中数据链表的头指针
        KeyNode *p = bucket_table[index];
        //该桶中还没有数据
        if (p->key == 0) {
            bucket_table[index]->next = node;
            (bucket_table[index]->key)++;
        }
        else {
            //链表结构的插入排序
            while (p->next != NULL&&p->next->key <= node->key)
                p = p->next;
            node->next = p->next;
            p->next = node;
            (bucket_table[index]->key)++;
        }
    }
    //打印结果
    for (int b = 0; b < bucket_size; b++)
        for (KeyNode *k = bucket_table[b]->next; k != NULL; k = k->next) {
            printf("%d\n", k->key);
        }

}

void main() {
    int raw[] = { 49,38,65,97,76,13,27,49 };
    int size = sizeof(raw) / sizeof(int);
    inc_sort(raw, size, 10);
    getchar();
}
```

基数排序：

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define RADIXCOUNT 10  //桶的个数，桶号：0 1 2 ..... 9
#define RANDMAX 100000 //随机数的最大值加1+

struct Node {
    int value;
    struct Node *next;
};

struct Queue {
    struct Node *head;
    struct Node *tail;
};

void getRandArray(int array[], int size);
void radixSort(int array[], int size);
void printArray(int array[], int size);
int getMaxLength(int array[], int size);
void distributeNumbers(int array[], int size, struct Queue bucket[], int dividend);
void rearrangeArray(int array[], int size, struct Queue bucket[]);
void isSorted(int array[], int size);

//利用伪随机数填充数组array
void getRandArray(int array[], int size)
{
    if (array == NULL || size <= 0) {
        printf("error!");
        return;
    }

    srand((unsigned)time(NULL));
    int i = 0;
    for (i = 0; i < size; ++i) {
        //产生RANDMAX以内的伪随机数
        array[i] = rand() % RANDMAX;
    }
}

//基数排序，按从小到大的顺序进行排列
void radixSort(int array[], int size)
{
    if (array == NULL || size <= 0) {
        printf("error!");
        return;
    }

    struct Queue bucket[RADIXCOUNT];
    int i = 0;

    for (i = 0; i < RADIXCOUNT; i++) {
```

```
        bucket[i].head = NULL;
        bucket[i].tail = NULL;
    }

    int maxLength = getMaxLength(array, size);
    int dividend = 1;
    for (i = 0; i < maxLength; ++i) {
        distributeNumbers(array, size, bucket, dividend);
        rearrangeArray(array, size, bucket);
        dividend *= 10;
    }
}

//获取数组array中最大数的长度（位数）
int getMaxLength(int array[], int size)
{
    if (array == NULL || size <= 0) {
        printf("error!");
        return 0;
    }

    int max = array[0];
    int i = 0;
    for (i = 1; i < size; ++i) {
        if (max < array[i]) {
            max = array[i];
        }
    }

    int length = 1;
    while ((max /= 10) != 0) {
        ++length;
    }
    return length;
}

//把数组array中的数放到对应的桶中,桶的底层是用链式队列实现
void distributeNumbers(int array[], int size, struct Queue bucket[], int dividend)
{
    if (array == NULL || size <= 0 || bucket == NULL || dividend <= 0) {
        printf("error!");
        return;
    }

    int radixValue = 0;
    struct Node *node;
    int i = 0;
    for (i = 0; i < size; ++i) {
        //把array[i]放到下标为radixValue的桶中
        radixValue = (array[i] / dividend) % 10;
        node = (struct Node *) malloc(sizeof(struct Node));
        node->value = array[i];

        node->next = NULL;
```

```
            if (bucket[radixValue].head == NULL) {
                bucket[radixValue].head = node;
                bucket[radixValue].tail = node;
            }
            else {
                bucket[radixValue].tail->next = node;
                bucket[radixValue].tail = node;
            }
        }
    }

    //把桶0..9中的数按放入桶中的先后次序放回到数组array中
    void rearrangeArray(int array[], int size, struct Queue bucket[])
    {
        if (array == NULL || size <= 0 || bucket == NULL) {
            printf("error!");
            return;
        }

        struct Node *pointer = NULL;
        int arrayIndex = 0;
        int listIndex = 0;
        for (listIndex = 0; listIndex < RADIXCOUNT; ++listIndex) {
            while (bucket[listIndex].head != NULL) {
                array[arrayIndex++] = bucket[listIndex].head->value;
                pointer = bucket[listIndex].head;
                bucket[listIndex].head = bucket[listIndex].head->next;
                free(pointer);
            }
        }
    }

    void printArray(int array[], int size)
    {
        if (array == NULL || size <= 0) {
            printf("error!");
            return;
        }

        int i = 0;
        for (i = 0; i < size; ++i) {
            printf("%d ", array[i]);
        }
        printf("\n\n");
    }

    //判断数组array是否已经是有序的
    void isSorted(int array[], int size)
    {
        if (array == NULL || size <= 0) {
            printf("error!");
            return;

        }
```

```
    int unsorted = 0;
    int i = 0;
    for (i = 1; i < size; ++i) {
        if (array[i] < array[i - 1]) {
            unsorted = 1;
            break;
        }
    }

    if (unsorted) {
        printf("the array is unsorted!\n");
    }
    else {
        printf("the array is sorted!\n");
    }
}

void main()
{
    int size = 0;
    printf("please enter size: ");
    scanf("%d", &size);
    if (size <= 0)
    {
        printf("error,please enter again: ");
        scanf("%d", &size);
    }

    int *array = (int *)calloc(size, sizeof(int));
    getRandArray(array, size);
    printArray(array, size);

    radixSort(array, size);
    printArray(array, size);
    isSorted(array, size);

    free(array);
    getchar();
    getchar();
}
```

4. 结果分析：

    1. 桶式排序：

        > 首先对数据进行分区块的时间复杂度为O(N)，而对桶中的数据进行排序的平均时间复杂度为O(N*logN)，最好的结果是每个桶中都只有一个数据，此时时间复杂度为零。

    2. 基数排序：

        > 从代码中可以知道，基数排序的时间复杂度为O(k*n),其中k为最大值的位数，n为数据数

2. stack

1. 问题描述：

   > 用C语言设计堆栈，并实现中缀表达式到后缀表达式的转换

2. 问题分析与算法设计：

   1. 问题分析：首先构造出相应的 stack ，然后通过对每个字符的判断，并进行相应的入栈和出栈操作。
   2. 算法设计：

      1. 开始扫描；
      2. ·数字时，加入后缀表达式；
      3. 运算符：
      4. a. 若为 '('，入栈；

         b. 若为 ')'，则依次把栈中的的运算符出栈，加入后缀表达式中，直到出现'('，从栈中删除'(' ；

         c. 若为 除括号外的其他运算符， 当其优先级高于除'('以外的栈顶运算符时，直接入栈。否则从栈顶开始，依次弹出比当前处理的运算符优先级高和优先级相等的运算符，直到一个比它优先级低的或者遇到了一个左括号为止。

3. 算法实现：

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define OK 1
#define ERROR -1
#define TRUE 1
#define FALSE 0
#define MAXSIZE 10

typedef int Status;
typedef char element_type;

typedef struct {
    element_type data[MAXSIZE];
    int top;//栈顶指针
}Stack;

//1. 初始化
Status InitStack(Stack *S) {
    int i;
    for (i = 0; i<MAXSIZE; i++)
        S->data[i] = NULL;
    S->top = -1;
    return OK;
}
//2. 创建一个长度为n的堆栈
Status CreateStack(Stack *S, int n) {
    if (n > MAXSIZE || n < 1) {
        printf("输入长度有误! \n");
        return ERROR;
    }
    srand(time(0));
    int i;
    for (i = 0; i<n; i++) {
        S->data[i] = rand() % 100 + 1;
    }
    S->top = n - 1;

    return OK;
}

//3. 压栈操作
Status push(Stack *S, element_type e) {
    if (MAXSIZE - 1 == S->top) {
        printf("栈已满\n");
        return ERROR;
    }
    //栈顶指向的元素有值
    ++(S->top);
    S->data[S->top] = e;

    return OK;
```

```
}

//4. 出栈
Status pop(Stack *S, element_type *e) {
    //将栈顶元素出栈，传给e
    if (-1 == S->top) {
        printf("栈为空！\n");
        return ERROR;
    }
    *e = S->data[S->top];
    --(S->top);
    return OK;
}

//5. 中缀表达式转后缀表达式
void MidToFinal(char *mid, char *final) {
    //中缀表达式为middle，要转换成后缀表达式传给last
    //新建一个栈，来存储符号
    char e;
    Stack S;
    if (OK != InitStack(&S)) {
        printf("初始化栈失败！\n");
    }
    //当带转换的字符串*mid未终止时，循环处理
    while (*mid) {
        //如果是数字，则直接输出
        if (*mid >= '0' && *mid <= '9') {
            *(final++) = *(mid++);
            continue;
        }
        else if (*mid == '+' || *mid == '-' || *mid == '*' || *mid == '/' || *mid
== '(' || *mid == ')') {
            //输入的是合法运算符号,比较之前是否有更高优先级的符号
            if (S.top == -1 || '(' == *mid) {
                //当符号栈为空或遇到左括号时，符号入栈
                push(&S, *(mid++));
                continue;
            }
            if (')' == *mid) {
                //遇到右括号时，栈顶元素依次出栈；直到遇到第一个左括号时结束
                pop(&S, &e);
                *(final++) = e;
                while (pop(&S, &e) && e != '(') {
                    *(final++) = e;
                }
                // printf("%c\n",e);
                mid++;
                continue;
            }
            //后续的处理都要取出临时的栈顶元素，与当前输入的符号*mid相比较；当临时栈顶元
素优先级大于等于输入符号的优先级时，出栈；否则符号入栈（已经弹出一个，记得把弹出的元素也入
栈）

            pop(&S, &e);
```

```
            if ('+' == *mid || '-' == *mid) {
                if (e == '(') {
                    push(&S, '(');
                    push(&S, *(mid++));
                    continue;
                }
                else {
                    *(final++) = e;
                    push(&S, *(mid++));
                    continue;
                }
            }
            else if ('*' == *mid || '/' == *mid) {
                if ('*' == e || '/' == e) {
                    *(final++) = e;
                    push(&S, *(mid++));
                    continue;
                }
                else {
                    push(&S, e);
                    push(&S, *(mid++));
                    continue;
                }
            }
        }
        else {
            printf("输入的字符不合法！%c\n", *mid);
            return;
        }
    }
    //当待转换的字符已经结束时，符号栈至少还有一个元素（中缀表达式的特点：数字结尾；后缀表达式以符号结尾）；将栈中的元素依次出栈
    while (S.top != -1) {
        pop(&S, &e);
        *(final++) = e;
    }
    //字符串的结束符！
    *final = '\0';
}

void main()
{
    char data[] = "3+(5*6-7/1*7)*9";
    char final[] = "";
    MidToFinal(data, final);
    printf("%s\n", final);
}
```