数据结构作业

第10章

10.1

原始序列: 503, 087, 512, 061, 908, 170, 897, 275, 653, 426

1. 插入排序
    1. 087, 503, 512, 061, 908, 170, 897, 275, 653, 426
    2. 087, 503, 512, 061, 908, 170, 897, 275, 653, 426
    3. 061, 087, 503, 512, 908, 170, 897, 275, 653, 426
    4. 061, 087, 503, 512, 908, 170, 897, 275, 653, 426
    5. 061, 087, 170, 503, 512, 908, 897, 275, 653, 426
    6. 061, 087, 170, 503, 512, 897, 908, 275, 653, 426
    7. 061, 087, 170, 275, 503, 512, 897, 908, 653, 426
    8. 061, 087, 170, 275, 503, 512, 653, 897, 908, 426
    9. 061, 087, 170, 275, 426, 503, 512, 653, 897, 908
2. 希尔排序($d[1] = 5$)
    1. 170, 087, 275, 061, 426, 503, 897, 512, 653, 908
    2. 170, 087, 275, 061, 426, 503, 897, 512, 653, 908
    3. 061, 087, 275, 170, 426, 503, 897, 512, 653, 908
    4. 061, 087, 275, 170, 426, 503, 653, 512, 897, 908
    5. 061, 087, 170, 275, 426, 503, 512, 653, 897, 908
3. 快速排序
    1. 426, 087, 275, 061, 170, 503, 897, 908, 653, 512
    2. 170, 087, 275, 061, 426, 503, 512, 653, 897, 908
    3. 061, 087, 170, 275, 426, 503, 512, 653, 897, 908
4. 堆排序
    1. 061, 087, 170, 275, 426, 512, 897, 503, 653, 908
    2. 061, 087, 275, 170, 503, 426, 512, 897, 908, 653
    3. 061, 087, 170, 275, 512, 503, 426, 653, 897, 908
    4. 061, 087, 170, 275, 426, 512, 503, 908, 653, 897
    5. 061, 087, 170, 275, 426, 503, 512, 897, 908, 653
    6. 061, 087, 170, 275, 426, 503, 653, 512, 897, 908
    7. 061, 087, 170, 275, 426, 503, 512, 653, 908, 897
    8. 061, 087, 170, 275, 426, 503, 512, 653, 897, 908
5. 归并排序
    1. 087, 503, 061, 512, 170, 908, 275, 897, 426, 653
    2. 061, 087, 503, 512, 170, 275, 897, 908, 426, 653
    3. 061, 087, 170, 275, 503, 512, 897, 908, 426, 653
    4. 061, 087, 170, 275, 426, 503, 512, 653, 897, 908

6. 基数排序
    1. 170, 061, 512, 503, 653, 275, 426, 087, 897, 908
    2. 503, 908, 512, 426, 653, 061, 170, 275, 087, 897
    3. 061, 087, 170, 275, 426, 503, 512, 653, 897, 908

10.3

快速排序, 希尔排序, 堆排序是不稳定的, 其他是稳定的.

不稳定例子:

1. 快速排序: 3, 2, 1, 2
2. 希尔排序: 3, 2, 1, 2(d=3)
3. 堆排序: 3, 2, 1, 2

10.15

1. 将n个元素两两配对, 进行$\lfloor n/2 \rfloor$次比较;
2. 每组较大的进入较大组, 较小的进入较小组;
3. 若较大组元素不唯一, 执行步骤4, 否则执行步骤5;
4. 较大组的元素两两配对, 较大的再构成新的较大组, 返回执行步骤3;
5. 若较小组元素不唯一, 执行步骤6, 否则执行步骤7;
6. 较小组的元素两两配对, 较小的再构成新的较小组;
7. 较大组的唯一元素为最大元, 较小组的唯一元素为最小元, 输出;

最坏情况下运行$\lfloor \frac{3}{2}n \rfloor$次

10.21

(好难画啊)

折半插入排序最大比较次数为6, 归并排序最大比较次数为5

10.25

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

#define KEY int
#define TYPE int

typedef struct node
{
    TYPE data;
    KEY key;
    struct node *next;
} Node;

typedef struct List
{
    Node *head;
    int len;
} List;

/*********************************************
Function name:  list_insert_sort
Purpose:        将链表L排序, 结果从小到大排列
Params:
    @List       L:  要排序的链表, 头指针为空, 至少有一个元素
Return:         void
*********************************************/
void list_insert_sort(List L)
{
    Node *pre_p, *p, *pre_q, *q; // 4个辅助指针, p, q为主要指针
                                 // pre_p, pre_q为相应的前驱, 用于交换
    pre_p = L.head->next;        // 初始化指针p, pre_p为链表第二个元素
    p = pre_p->next;
    while (p)           // 当指针非空
    {                   //
        pre_q = L.head; // 初始化指针q, 用于循环
        q = pre_q->next;
        while (q != p)
        {
            if (p->key < q->key) // 当p的关键字较小, 交换位置
            {
                pre_p->next = p->next;
                p->next = q;
                pre_q->next = p;
                break;
            } // 否则q指针推向下一个
            pre_q = q;
            q = q->next;
        }
        if (pre_p->next == p) // 若没有发生交换, pre_ 向前推进 否则pre_p保持不变
        {
            pre_p = p;
        }

        p = pre_p->next; // p指针是pre_p的后继
        }
}
```

10.30

```c
#include <stdio.h>
#include <stdlib.h>
#include "sort_list.h"
#include "stack.h"

int partition(List *L, int low, int high, int *flag)
{
    KEY pivotkey = L->r[low].key;
    while (low < high)
    {
        while (low < high && L->r[high].key >= pivotkey)
        {
            high--;
        }
        if (low != high)
        {
            node_swap(&(L->r[low]), &(L->r[high]));
            *flag = 1;
        }
        while (low < high && L->r[low].key <= pivotkey)
        {
            low++;
        }
        if (low != high)
        {
            node_swap(&(L->r[low]), &(L->r[high]));
            *flag = 1;
        }
    }
    return low;
}

void quick_sort(List *L)
{
    Stack S = stack_create();
    int low = 1, high = L->len;
    while (low < high)
    {
        if (high - low < 3)
        {
            for (int i = low; i < high; i++)
            {
                for (int j = i + 1; j <= high; j++)
                {
                    if (L->r[j].key < L->r[i].key)
                    {
                        node_swap(&(L->r[i]), &(L->r[j]));
                    }
                }
            }
        }
        else

        {
            int flag = 0;
            int pivot_low = partition(L, low, high, &flag);
            if (flag)
            {
                if (high - pivot_low > pivot_low - low)
                {
                    stack_push(&S, pivot_low + 1);
                    stack_push(&S, high);
                    low = pivot_low - 1;
                    continue;
                }
                else
                {
                    stack_push(&S, low);
                    stack_push(&S, pivot_low - 1);
                    high = pivot_low + 1;
                    continue;
                }
            }
        }
        if (!stack_empty(S))
        {
            high = stack_pop(&S);
            low = stack_pop(&S);
        }
    }
}
```

10.32

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

#define STACK_TYPE Node
#include "stack.h"

typedef enum COLOR {
    RED = 0,
    WHITE = 1,
    BLUE = 2
} COLOR;

void sort_by_color(List *L)
{
    Stack red_s, white_s, blue_s;
    red_s = stack_create();
    white_s = stack_create();
    blue_s = stack_create();
    Node temp_node;

    for (int i = 1; i < L->len; i++) // 建立3个栈，按颜色把节点压进栈
    {
        temp_node = L->r[i];
        if (temp_node.key == RED)
        {
            stack_push(&red_s, temp_node);
        }
        else if (temp_node.key == WHITE)
        {
            stack_push(&white_s, temp_node);
        }
        else if (temp_node.key == BLUE)
        {
            stack_push(&blue_s, temp_node);
        }
        else
        {
            return;
        }
    }

    for (int i = 1; i < L->len; i++) // 按顺序退栈
    {
        if (!stack_empty(red_s))
        {
            temp_node = stack_pop(&red_s);
        }
        else if (!stack_empty(white_s))
        {
            temp_node = stack_pop(&white_s);
        }

        else if (!stack_empty(blue_s))
        {
            temp_node = stack_pop(&blue_s);
        }
        else
        {
            return;
        }
        *&(L->r[i]) = *&(temp_node); // 将对应位置的节点赋值为退栈节点;
    }
}

int main()
{
    List L;
    L.r[1].key = WHITE;
    L.r[2].key = BLUE;
    L.r[3].key = RED;
    L.r[4].key = WHITE;
    L.r[5].key = WHITE;
    L.r[6].key = RED;
    L.len = 7;

    for (int i = 1; i < L.len; i++)
        printf("%d", L.r[i].key);
    sort_by_color(&L);

    printf("\n");
    for (int i = 1; i < L.len; i++)
        printf("%d", L.r[i].key);

    return 0;
}
```

10.41

若选择$O(n\log n)$的高效排序算法, 所需的比较和移动次数约是10000次, 若选择$O(dn)$的基数排序, 所需的约是8000次(4000次压栈, 4000次退栈), 故选择基数排序.

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "sort_list.h"

#define STACK_TYPE Node
#include "stack.h"

/* 获取整数num的倒数第count位 */
#define DIGIT(num, count) (int)(num % (int)pow(10, count + 1)) / (int)pow(10, count)

void radix_sort(List *L)
{
    Node temp_node;
    Stack stacks[10]; // 建立10个栈，并初始化
    for (int i = 0; i < 10; i++)
    {
        stacks[i] = stack_create();
    }
    for (int digit = 0; digit < 4; digit++) // 对位数做循环，小于10000则最多4位
    {
        for (int i = 1; i < L->len; i++) // 按key的倒数第digit位压进栈
        {
            temp_node = L->r[i];
            stack_push(&stacks[DIGIT(temp_node.key, digit)], temp_node);
        }

        int stack_num = 9;
        for (int i = L->len - 1; i > 0; i--) // 逆序出栈
        {
            temp_node = stack_pop(&stacks[stack_num]);
            if (stack_empty(stacks[stack_num]) && stack_num > 0)
            {
                stack_num--;
            }
            *&(L->r[i]) = *&(temp_node); // 赋值
        }
    }
}
```

附: sort_list.h

```c
#include <stdlib.h>

#define MAXSIZE 20
#define KEY int
#define TYPE int

typedef struct Node
{
    KEY key;
    TYPE data;
} Node;

typedef struct List
{
    Node r[MAXSIZE];
    int len;
} List;

void node_swap(Node *a, Node *b)
{
    Node *temp = (Node *)malloc(sizeof(Node));
    *temp = *a;
    *a = *b;
    *b = *temp;
}
```

附: stack.h

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

#define MAX_SIZE 100
#define INCREASE 10

#define MALLOC -1
#define EMPTYSTACK -2

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

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

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

Stack stack_create()
{
    Stack S;
    S.base = (STACK_TYPE *)malloc(sizeof(STACK_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, STACK_TYPE data)
{
    if (S->top - S->base >= S->size)
    {
        S->size += INCREASE;
        S->base = (STACK_TYPE *)realloc(S->base, sizeof(STACK_TYPE) * S->size);
        if (S->base == NULL)
        {
```

```c
            printf("Fail to push, storage allocation error\n");
            exit(MALLOC);
        }
    }
    *((S->top)++) = data;
}

STACK_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;
    }
}

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