阅读：56,351　　作者：解学武

快速排序算法（QSort，快排）及C语言实现

上节介绍了如何使用起泡排序的思想对无序表中的记录按照一定的规则进行排序，本节再介绍一种排序算法——快速排序算法（Quick Sort）。

> C语言中自带函数库中就有快速排序——qsort函数，包含在 <stdlib.h> 头文件中。

快速排序算法是在起泡排序的基础上进行改进的一种算法，其实现的基本思想是：通过一次排序将整个无序表分成相互独立的两部分，其中一部分中的数据都比另一部分中包含的数据的值小，然后继续沿用此方法分别对两部分进行同样的操作，直到每一个小部分不可再分，所得到的整个序列就成为了有序序列。

例如，对无序表 {49，38，65，97，76，13，27，49} 进行快速排序，大致过程为：

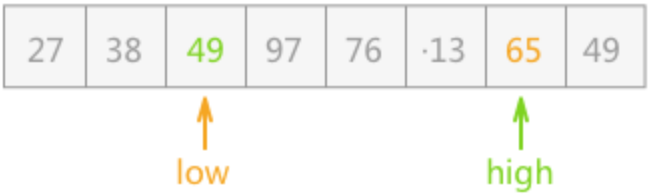
1. 首先从表中选取一个记录的关键字作为分割点（称为“枢轴”或者支点，一般选择第一个关键字），例如选取 49；
2. 将表格中大于 49 个放置于 49 的右侧，小于 49 的放置于 49 的左侧，假设完成后的无序表为：{27，38，13，49，65，97，76，49}；
3. 以 49 为支点，将整个无序表分割成了两个部分，分别为 {27，38，13} 和 {65，97，76，49}，继续采用此种方法分别对两个子表进行排序；
4. 前部分子表以 27 为支点，排序后的子表为 {13，27，38}，此部分已经有序；后部分子表以 65 为支点，排序后的子表为 {49，65，97，76}；
5. 此时前半部分子表中的数据已完成排序；后部分子表继续以 65为支点，将其分割为 {49} 和 {97，76}，前者不需排序，后者排序后的结果为 {76，97}；
6. 通过以上几步的排序，最后由子表 {13，27，38}、{49}、{49}、{65}、{76，97} 构成有序表：{13，27，38，49，49，65，76，97}；

整个过程中最重要的是实现第 2 步的分割操作，具体实现过程为：

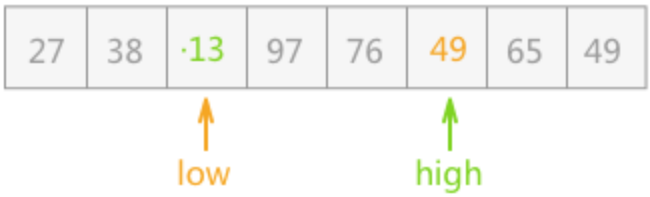
- 设置两个指针 low 和 high，分别指向无序表的表头和表尾，如下图所示：

- 先由 high 指针从右往左依次遍历，直到找到一个比 49 小的关键字，所以 high 指针走到 27 的地方停止。找到之后将该关键字同 low 指向的关键字进行互换：

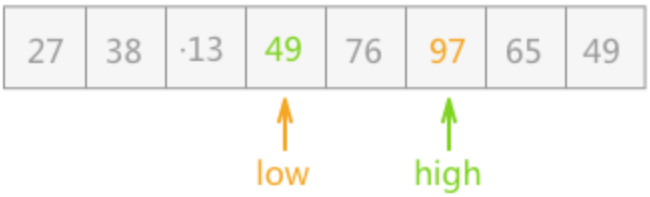
- 然后指针 low 从左往右依次遍历，直到找到一个比 49 大的关键字为止，所以 low 指针走到 65 的地方停止。同样找到后同 high 指向的关键字进行互换：



- 指针 high 继续左移，到 13 所在的位置停止（13<49），然后同 low 指向的关键字进行互换：



- 指针 low 继续右移，到 97 所在的位置停止（97>49），然后同 high 指向的关键字互换位置：



- 指针 high 继续左移，此时两指针相遇，整个过程结束；

该操作过程的具体实现代码为：

```
#define MAX 8
typedef struct {
    int key;
}SqNote;

typedef struct {
    SqNote r[MAX];
    int length;
}SqList;
//交换两个记录的位置
void swap(SqNote *a,SqNote *b){
    int key=a->key;
    a->key=b->key;
    b->key=key;
}
//快速排序，分割的过程
int Partition(SqList *L,int low,int high){
    int pivotkey=L->r[low].key;
    //直到两指针相遇，程序结束
    while (low<high) {
```

```
        //high指针左移，直至遇到比pivotkey值小的记录，指针停止移动
        while (low<high && L->r[high].key>=pivotkey) {
            high--;
        }
        //交换两指针指向的记录
        swap(&(L->r[low]), &(L->r[high]));
        //low 指针右移，直至遇到比pivotkey值大的记录，指针停止移动
        while (low<high && L->r[low].key<=pivotkey) {
            low++;
        }
        //交换两指针指向的记录
        swap(&(L->r[low]), &(L->r[high]));
    }
    return low;
}
```

该方法其实还有可以改进的地方：在上边实现分割的过程中，每次交换都将支点记录的值进行移动，而实际上只需在整个过程结束后（low==high），两指针指向的位置就是支点记录的准确位置，所以无需每次都移动支点的位置，最后移动至正确的位置即可。

所以上边的算法还可以改写为：

```
//此方法中，存储记录的数组中，下标为 0 的位置时空着的，不放任何记录，记录从下标为 1 处开始依次存放
int Partition(SqList *L,int low,int high){
    L->r[0]=L->r[low];
    int pivotkey=L->r[low].key;
    //直到两指针相遇，程序结束
    while (low<high) {
        //high指针左移，直至遇到比pivotkey值小的记录，指针停止移动
        while (low<high && L->r[high].key>=pivotkey) {
            high--;
        }
        //直接将high指向的小于支点的记录移动到low指针的位置。
        L->r[low]=L->r[high];
        //low 指针右移，直至遇到比pivotkey值大的记录，指针停止移动
        while (low<high && L->r[low].key<=pivotkey) {
            low++;
        }
        //直接将low指向的大于支点的记录移动到high指针的位置
        L->r[high]=L->r[low];
    }
    //将支点添加到准确的位置
    L->r[low]=L->r[0];
    return low;
}
```

快速排序的完整实现代码（C语言）

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 9
//单个记录的结构体
typedef struct {
    int key;
}SqNote;
//记录表的结构体
typedef struct {
    SqNote r[MAX];
    int length;
}SqList;
//此方法中，存储记录的数组中，下标为 0 的位置时空着的，不放任何记录，记录从下标为 1 处开始依次存放
int Partition(SqList *L,int low,int high){
    L->r[0]=L->r[low];
    int pivotkey=L->r[low].key;
    //直到两指针相遇，程序结束
    while (low<high) {
        //high指针左移，直至遇到比pivotkey值小的记录，指针停止移动
        while (low<high && L->r[high].key>=pivotkey) {
            high--;
        }
        //直接将high指向的小于支点的记录移动到low指针的位置。
        L->r[low]=L->r[high];
        //low 指针右移，直至遇到比pivotkey值大的记录，指针停止移动
        while (low<high && L->r[low].key<=pivotkey) {
            low++;
        }
        //直接将low指向的大于支点的记录移动到high指针的位置
        L->r[high]=L->r[low];
    }
    //将支点添加到准确的位置
    L->r[low]=L->r[0];
    return low;
}
void QSort(SqList *L,int low,int high){
    if (low<high) {
        //找到支点的位置
        int pivotloc=Partition(L, low, high);
        //对支点左侧的子表进行排序
        QSort(L, low, pivotloc-1);
        //对支点右侧的子表进行排序
        QSort(L, pivotloc+1, high);
    }
```

```
45.  }
46.  void QuickSort(SqList *L){
47.      QSort(L, 1,L->length);
48.  }
49.  int main() {
50.      SqList * L=(SqList*)malloc(sizeof(SqList));
51.      L->length=8;
52.      L->r[1].key=49;
53.      L->r[2].key=38;
54.      L->r[3].key=65;
55.      L->r[4].key=97;
56.      L->r[5].key=76;
57.      L->r[6].key=13;
58.      L->r[7].key=27;
59.      L->r[8].key=49;
60.      QuickSort(L);
61.      for (int i=1; i<=L->length; i++) {
62.          printf("%d ",L->r[i].key);
63.      }
64.      return 0;
65.  }
```

运行结果：

```
13 27 38 49 49 65 76 97
```

总结

快速排序算法的时间复杂度为 O(nlogn)，是所有时间复杂度相同的排序方法中性能最好的排序算法。