分配排序

桶排序和基数排序均属于分配排序。分配排序的基本思想：排序过程无须比较关键字，而是通过用额外的空间来"分配"和"收集"来实现排序，它们的时间复杂度可达到线性阶：O(n)。简言之就是：用空间换时间，所以性能与基于比较的排序才有数量级的提高！

桶排序(Bucket Sort)，也称箱排序

基本思想：设置若干个箱子，依次扫描待排序的记录 array[0]，array[1]，…，array[n - 1]，把关键字等于 k 的记录全都装入到第 k 个箱子里(分配)，然后按序号依次将各非空的箱子里的记录收集起来，从而完成排序。

桶排序所需要的额外空间取决于关键字的个数，若 array[0..n - 1] 中关键字的取值范围是 0 到 m - 1 的整数，则必须设置 m 个箱子。因此箱排序要求关键字的类型是有限类型，否则可能要无限个箱子。一般情况下每个箱子中存放多少个关键字相同的记录是无法预料的，故箱子的类型应设计成链表为宜。
在实现上，桶排序一般采用另外的变种。即：把 [0， m) 划分为 m 个大小相同的子区间，每一子区间是一个桶。然后将 n 个记录分配到各个桶中。因为关键字序列是均匀分布在[0，m)上的，所以一般不会有很多个记录落入同一个桶中。由于同一桶中的记录其关键字不尽相同，所以必须采用关键字比较的排序方法(通常用插入排序)对各个桶进行排序，然后依次将各非空桶中的记录收集起来即可。
C 代码实现：

```
struct bucket_node …{
    int key;
    bucket_node* next;
};

 // 取得数组中最大数的位数
 int get_max_digital_count(int* array, int length)
…{
    assert(array && length > 0);

    int i = 0;
    int max = array[0];
    int maxDigitalCount = 1;

```

```
    for (i = 1; i < length; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }

    while ((max / 10) > 0) {
        max %= 10;
        ++maxDigitalCount;
    }

    return maxDigitalCount;
}

// 取得数 num 中从低到高第 n 位上的数字
int get_ditital_at(int num, int n)
{
    while (--n > 0) {
        num /= 10;
    }

    return (num % 10);
}

// 箱/桶排序
//
void bucket_sort(int* array, int length)
{
    assert(array && length >= 0);

    if (length <= 1) {
        return;
    }
```

```
    int i, index;
    bucket_node* temp = NULL;
    bucket_node bucket[10] = {0, };    // 根据数字个数 0 ~ 9 建立 10 个桶

    int count = get_max_digital_count(array, length);

    // 建立数据节点
    bucket_node* data = (bucket_node*)malloc(length * sizeof(bucket_node
));
    if (!data) {
        printf("Error: out of memory!/n");
        return;
    }

    for (i = 0; i < length; i++) {
        data[i].key = array[i];
        data[i].next = NULL;
    }

    // 分配
    for (i = 0; i < length; i++) {
        index = get_ditital_at(data[i].key, count);
        if (bucket[index].next == NULL) {
            bucket[index].next = &data[i];
        }
        else {
            temp = &bucket[index];

            while (temp->next != NULL && temp->next->key < data[i].key) {
                temp = temp->next;
            }
```

```
            data[i].next = temp->next;
            temp->next = &data[i];
        }
    }

    // 收集
    index = 0;
    for (i = 0; i < 10; i++) {
        temp = bucket[i].next;
        while (temp != NULL) {
            array[index++] = temp->key;
            temp = temp->next;
        }
    }


    free(data);
}
```

时间复杂度分析：
桶排序的平均时间复杂度是线性的，即 O(n)。但最坏情况仍有可能是 O(n ^ 2)。

空间复杂度分析：
桶排序只适用于关键字取值范围较小的情况，否则所需箱子的数目 m 太多导致浪费存储空间和计算时间。

基数排序(Radix Sort)

基本思想：基数排序是对桶排序的改进和推广。如果说桶排序是一维的基于桶的排序，那么基数排序就是多维的基于桶的排序。我这么说，可能还不是太清楚。比方说：用桶排序对 [0, 30] 之间的数进行排序，那么需要 31 个桶，分配一次，收集一次，完成排序；那么基数排序则只需要 0 - 9 总共 10 个桶（即关键字为数字 0 - 9），依次进行个位和十位的分配和收集从而完成排序。

C 代码实现：

```
// 基数排序
//
void radix_sort(int* array, int length)
{…
    assert(array && length >= 0);

    if (length <= 1) …{
        return;
    }

    const int buffer_size = length * sizeof(int);

    int i, k, count, index;
    int bucket[10] = …{0, };    // 根据数字个数 0 ~ 9 建立 10 个桶

    int* temp = (int*)malloc(buffer_size);
    if (!temp) …{
        printf("Error: out of memory!/n");
        return;
    }

    count = get_max_digital_count(array, length);

    for (k = 1; k <= count; ++k) …{
        memset(bucket, 0, 10 * sizeof(int));

        // 统计各桶中元素的个数
        for (i = 0; i < length; ++i) …{
            index = get_ditital_at(array[i], k);
            ++bucket[index];
        }

        // 为每个记录创建索引下标
```

```
        for (i = 1; i < 10; ++i) {
            bucket[i] += bucket[i - 1];
        }

        // 按索引下标顺序排列
        for (i = length - 1; i >= 0; --i) {
            index = get_ditital_at(array[i], k);
            assert(bucket[index] - 1 >= 0);
            temp[--bucket[index]] = array[i];
        }

        // 一趟桶排序完毕，拷贝结果
        memcpy(array, temp, buffer_size);

#ifdef DEBUG_SORT
        debug_print(" 第 %d 趟排序：", k);
        for (i = 0; i < length; ++i) {
            debug_print("%d ", array[i]);
        }

        debug_print("/n");
#endif
    }

    free(temp);
}
```

时间复杂度分析：
基数排序的时间负责度为 O(n)。

空间复杂度：
基数排序所需的辅助存储空间为 O(n + r * d)，其中 r 为记录中关键字分量的最大个数，d 为关键字的个数。比如说：待排序为 0 - 999，那么分量的最大个数为 3，关键字的个数为 10（0 - 9）。

补充：
基数排序是稳定的。

若排序文件不是以数组 array 形式给出，而是以单链表形式给出(此时称为链式的基数排序)，则可通过修改出队和入队函数使表示箱子的链队列无须分配结点空间，而使用原链表的结点空间。人队出队操作亦无需移动记录而仅需修改指针。虽然这样一来节省了一定的时间和空间，但算法要复杂得多，且时空复杂度就其数量级而言并未得到改观。
==========
==========
测试：
在前文《排序算法之插入排序》测试代码的基础上添加两行代码即可：
```
{"桶/箱排序",    bucket_sort},
{"基数排序",     radix_sort},
```

测试结果：

```
=== 桶/箱排序 ===
original: 65 32 49 10 8 72 27 42 18 58 91
  sorted: 8 10 18 27 32 42 49 58 65 72 91




original: 10 9 8 7 6 5 4 3 2 1 0
  sorted: 0 1 2 3 4 5 6 7 8 9 10
=== 基数排序 ===
original: 65 32 49 10 8 72 27 42 18 58 91
第 1 趟排序：10 91 32 72 42 65 27 8 18 58 49
第 2 趟排序：8 10 18 27 32 42 49 58 65 72 91
  sorted: 8 10 18 27 32 42 49 58 65 72 91


original: 10 9 8 7 6 5 4 3 2 1 0
第 1 趟排序：10 0 1 2 3 4 5 6 7 8 9
第 2 趟排序：0 1 2 3 4 5 6 7 8 9 10
  sorted: 0 1 2 3 4 5 6 7 8 9 10
```

原文地址

http://www.cnblogs.com/kesalin/archive/2011/03/18/2338351.html