算法基础

第三次作业（DDL：2024 年 10 月 xx 日 23:59）

解答过程中请写出必要的计算和证明过程

Q1.（$5 + 15 = 20$ 分）

(a) 下面的排序算法中哪些是稳定的：插入排序、冒泡排序、希尔排序、堆排序和快速排序？

(b) 给出一个能使任何排序算法都稳定的方法。你所给出的方法带来的额外时间和空间开销是多少？

Solution:

(a) 插入排序和冒泡排序是稳定的，希尔排序、堆排序和快速排序不稳定。

(b) 给输入数组的每个元素增加第二键值 index，为该元素在输入数组中的索引值。修改元素之间的比较为：先比较元素大小，若大小相同则比较 index 大小。如此任何排序算法都变成稳定的排序算法。该方法额外使用了 n 个 index，所以额外空间开销是 $O(n)$。该方法的比较次数没有增多，所以时间复杂度没有变化。

Q2.（20 分）Quicksort 包含了两个对自身的递归调用：

```
Quicksort(A, p, r)
    if p < r then
        q = Partition(A, p, r)
        Quicksort(A, p, q - 1)
        Quicksort(A, q + 1, r)
```

其中第二个递归调用并非必须的。请修改 Quicksort，使得 Quicksort 只包含一个递归调用。（提示：使用一个循环结构来代替其中一个递归调用）

Solution:

```
TAIL-RECURSIVE-Quicksort(A, p, r)
    while p < r
        q = Partition(A, p, r)
        TAIL-RECURSIVE-Quicksort(A, p, q - 1)
        p = q + 1
```

Q3.（20 分）因为在基于比较的排序模型中，完成 n 个元素的排序，其最坏情况下需要 $\Omega(n\log n)$ 时间。试证明：任何基于比较的算法从 n 个元素的任意序列中构造一棵二叉搜索树，其最坏情况下需要 $\Omega(n\log n)$ 的时间。

Solution:
反证法：假设存在一个基于比较的算法 A 构造一棵二叉搜索树的最坏情况的时间 $T(n) < O(n\log n)$，则构建算法 B 为运行一次算法 A，然后输出得到的二叉搜索树的中序遍历。则算法 B 是一个基于比较的排序算法。中序遍历二叉搜索树的时间复杂度是 $\Theta(n)$，故算法 B 的时间复杂度 $T'(n) = T(n) + \Theta(n) < O(n\log n) + \Theta(n)$，所以 $T'(n) < \Omega(n\log n)$，与基于比较的排序模型中，完成 n 个元素的排序，其最坏情况下需要 $\Omega(n\log n)$ 时间矛盾，所以假设不成立。

Q4.（20 分）定义二叉搜索树 T 上节点的深度 $d(x)$ 如下：

$$d(x) = \begin{cases} 1, x = root(T) \\ d(p(x)) + 1, else \end{cases}$$

试证明：以随机的输入构建的二叉搜索树的平均节点深度的期望为 $\Theta(log(n))$

Solution:
设 $S(T)$ 为二叉搜索树 T 上节点的深度之和，则 $S(T) = \Sigma_{x\in T}d(x)$，平均节点深度 $avgd(T) = \frac{1}{n}\Sigma_{x\in T}d(x) = \frac{1}{n}S(T)$。
设 $P(n) = E(S(T))$，因为输入随机，所以每个元素作为根节点的概率相同，故 $P(n) = \frac{1}{n}\Sigma_{i=1}^{n}(P(i-1) + P(n-i) + n)$，因此等价于随机快速排序的平均时间复杂度，所以 $P(n) = \Theta(nlog(n))$。
所以 $E(avgd(T)) = \frac{1}{n}E(S(T)) = \Theta(log(n))$。

Q5.（20 分）在线地求数组中排名为 k 的数是二叉搜索树的应用之一。请修改二叉搜索树，并以此实现 Querykth(T, k)，返回二叉搜索树 T 中第 k 大的数，且时间复杂度为 $O(h)$。(提示：你可能需要申请额外的空间，以此维护更多的信息。)

Solution:

对于每个节点，多维护两个值：

count，代表当前节点包含了多少个 key 值相同的节点；

size，代表当前节点代表的子树的 count 值之和。

```
update(x, count)
    while x != null do
        size[x] = count[x] + size[left[x]] + size[right[x]]
        x = p[x]

NewTreeInsert(T, z)
    y = null
    x = root[T]
    while x != null do
        y = x
        if key[z] < key[x] then
            x = left[x]
        else if key[z] > key[x] then
            x = right[x]
        else
            count[x] ++
            update(x)
            return
    p[z] = y
    count[z] = 1
    size[z] = 1
    if y = null then
        root[T] = z
    else if key[z] < key[y] then
        left[y] = z
    else
        right[y] = z
    update(y)
```

```
NewTreeDelete(T, z)
    if count[z] > 1
        count[z] --
        update(z)
        return null
    if left[z] = null or right[z] = null then
        y = z
    else
        y = TreeSuccessor[z]
    if left[y] != null then
        x = left[y]
    else
        x = right[y]
    if x != null then
        p[x] = p[y]
    if p[y] == null then
        root[T] = x
    else if y = left[p[y]] then
        left[p[y]] = x
    else
        right[p[y]] = x
    if y != z then
        key[z] = key[y]
    count[y] = size[y] = 0
    update(x)
    update(p[y])
    return y
```

```
Querykth(T, k)
    x = root[T]
    if k < 1 or k > size[x]
        return null
    while !(size[left[x]] < k and k <= size[left[x]] + count[x])
        if size[left[x]] >= k
            x = left[x]
        else if size[left[x]] + count[x] < k
            k -= size[left[x]] + count[x]
            x = right[x]
    return x
```