算法设计-AVL

目录

1. 采用比较的方法，从 n 个元素中， 找出最大的元素和次最大的元素。

建堆找出最大元素 (n-1)；调整堆选出次最大元素($\log_2 n-1$)，总的时间 n+ $\log_2 n-2$

2. 采用比较的方法，从 n 分元素中，找出最大值和最小值。

将顺序存储的 n 个元素对称比较，即第一个元素与最后一个元素比较，第二个元素与倒数第二个元素比较，……，比较中的小者放前半部，大者放后半部，用了$\lceil n/2\rceil$次比较。再在前后两部分中分别简单选择最小和最大元素，各用$\lceil n/2\rceil-1$ 次比较。总共用了 $3*\lceil n/2\rceil-2$ 次比较。

3. BST 删除 data 值小于等于 x 的结点。

利用二叉排序树的性质，从根结点开始查找，若根结点的值小于等于 x，则根结点及其左子树均应删除，然后以右子树的根结点为树根，重新开始查找。若根结点的值大于 x，则顺左子树向下查找，直到某结点的值小于等于 x，则该结点及其左子树均应删除，同时将指向被删结点的指针（即双亲指向被删结点的指针）置空。

删除以某结点为根的子树，采用后序遍历：删除其左子树，删除其右子树，删除根结点。

4. 求二叉树中每个结点的平衡因子。

先用后序遍历的顺序求树中以每个结点为根的子树的高度，暂存在 bf 域中。然后采用层次遍历，计算每个结点的平衡因子。

```
// 计算以各结点为根的子树的高度，暂存到 bf 中
int getDepth(BitTree T){
    if (T == NULL)
        return 0;
    else {
        int m = getDepth(T->lchild);
        int n = getDepth(T->rchild);
        T->bf = m > n ? m+1 : n+1;
        return T->bf;
    }
}

// 计算各结点的平衡因子
void getBf(BitTree T)
{
    BitTree Queue[50], b;
    int front, rear;
    front = rear = 0;
    if (T)
    {
        Queue[rear++] = T;
        while (front != rear)
        {
            b = Queue[front++];
```

```
            // 计算平衡因子
            if (b->bf == 1) // 叶子结点平衡因子为0
                b->bf = 0;
            else if (!b->lchild && b->rchild) // 只有右孩子
                b->bf = 0 - b->rchild->bf;
            else if (b ->lchild && !b->rchild) // 只有左孩子
                b->bf = b->lchild->bf;
            else // 左右孩子都有
                b->bf = b->lchild->bf - b->rchild->bf;
            // 层次遍历结点入队
            if (b->lchild != NULL)
                Queue[rear++] = b->lchild;
            if (b->rchild != NULL)
                Queue[rear++] = b->rchild;
        }//while
    }//if
}
```

5. 在BST中，每个结点增设一个count成员，保存以该结点为根的子树上的结点个数，查找第k小值，返回其结点指针。

```
BiTree Search_samll_k(BiTree t, int k){
    if (k < 1 || k > t ->count) // 非法的k
        return NULL;
    if (t ->lchild == NULL){
        if (k == 1)
            return t;
        else
            return Search_samll_k(t ->rchild, k-1);
    } else {
        if (t ->lchild ->count == k - 1) // 查找成功
            return t;
        if (t ->lchild ->count > k - 1) // 在左子树中
            return Search_samll_k(t ->lchild, k);
        if (t ->lchild ->count < k - 1) // 在右子树中
            return Search_samll_k(t ->rchild, k-(t ->lchild ->count + 1));
    }
}
```

6. 求指定结点在二叉树中的层次。

```
int level(BiTree bt, BSTNode *p){
    int n = 0; // 统计查找次数
    Bitree t = bt;
    if (bt != NULL){
        n++;
        while (t ->data != p ->data){
```

```
            if (p ->data > t ->data) // 右子树搜索
                t = t ->rchild;
            else // 左子树搜索
                t = t ->lchild;
            n++; // 层次加一
        }
    }
    return n;
}
```

7. 判断一个序列是否是 BST 的搜索序列。

二叉排序树的查找走了一条从根结点到子孙结点的路径。查找开始时首先和根结点的值比较，若根结点的值大于待查结点的值，则到左子树中查找，否则到右子树中查找。子树根结点的值和待查结点值的比较，也遵循如上规律。在查找过程中逐渐缩小查找范围。我们用 high 表示查找的上界，用 low 表示查找的下界，若 low>high，则不是二叉排序树的查找序列。[illegible]

```
// a[]是搜索序列，n 是数据个数
bool check(int a[], int n){
    // low 表示搜索的下界，top 表示搜索的上界
    int prelow, pretop, low, top;
    low = prelow = -MAX, top = pretop = MAX;
    int i = 0; // 数组下标
    while (low < top && i < n){
        // 相当于沿右子女方向走，在某结点左转时退出
        while (a[i] > low && i < n){ // 逐渐增大
            prelow = low;
            low = a[i++];
        }
        if (i < n){ / 左转
            pretop = low;
            low = prelow;
            top = a[i++];
        }
        // 相当于沿左子女方向走，在某结点右转时退出
        while (a[i] < top && i < n){ / 逐渐减小
            pretop = top;
            top = a[i++];
        }
        if (i < n){ // 右转
            prelow = top;
            top = pretop;
            low = a[i++];
        }
```

```
    } // while
    return low < top;
}
```

8. 判断一个二叉树是否是 AVL。

设置二叉树的平衡标记 balance，以标记返回二叉树 bt 是否为平衡二叉树，若为平衡二叉树，则返回 1，否则返回 0；h 为二叉树 bt 的高度。采用后序遍历的递归算法：

1）若 bt 为空，则高度为 0，balance=1。

2）若 bt 仅有根结点，则高度为 1，balance=1。

3）否则，对 bt 的左、右子树执行递归运算，返回左、右子树的高度和平衡标记，bt 的高度为最高子树的高度加 1。若左、右子树的高度差大于 1，则 balance=0；若左、右子树的高度差小于等于 1，且左、右子树都平衡时，balance=1，否则 balance=0。

```
void judge_AVL(BiTree bt, int &balance, int &h){
    // 左右子树的平衡标记和高度
    int bl = 0, br = 0, hl = 0, hr = 0;
    if (bt == NULL){ // 空树，高度为 0
        h = 0;
        balance = 1;
    } else if (bt ->lchild == NULL && bt ->rchild == NULL){
        h = 1;
        balance = 1;
    } else {
        judge_AVL(bt ->lchild, bl, hl); // 递归判断左子树
        judge_AVL(bt ->rchild, br, hr); // 递归判断右子树
        h = (hl > hr ? hl : hr) + 1;
        if (abs(hl-hr) < 2)//若子树高度差的绝对值<2，则看左、右子树是否都平衡
            balance = bl && br;
        else
            balance = 0;
    }
}
```

9. 利用平衡因子求平衡二叉树的高度。

根结点的层次为 1，每下一层，层次加 1，直到层数最大的叶子结点。当结点的平衡因子 b 为 0 时，任选左、右一分支向下查找，若 b 不为 0，则沿左（当 b=1 时）或右（当 b=-1 时）子树向下查找。

递归：

```
int height(BiTree t){
    if (t == NULL)
        return 0;
    else if (t ->bf == 1 || t ->bf == 0)
        return height(t ->lchild) + 1;
    else
        return height(t ->rchild) + 1;
}
```

非递归：

```
int height(BiTree t){
    BiTree p = t;
    int h = 0;
    while (p){
        h++;
        if (p ->bf < 0)
            p = p ->rchild;
        else
            p = p ->lchild;
    }
    return h;
}
```

10. 快速排序算法代码 ｜ 单层快速排序的思想