中国科学院大学
数据结构实习报告

姓名 蓝宇舟 学号 2022K8009918005
姓名 韦欣池 学号 2022K8009907004
姓名 付博宇 学号 2022K8009937008
实习项目名称 马踏棋盘和平衡二叉树操作的演示

一、 马踏棋盘

1 算法介绍

a 实验具体要求

【问题描述】
设计一个国际象棋的马踏遍棋盘的演示程序。
【基本要求】
将马随机放在国际象棋的 8×8 棋盘 Board[8][8]的某个方格中，马按走棋规则进行移动。要求每个方格只进入一次，走遍棋盘上全部 64 个方格。编制非递归程序，求出马的行走路线，并按求出的行走路线，将数字 1，2，…，64 依次填入一个 8×8 的方阵，输出之。
【选作内容】
(1) 求出从某一起点出发的多条以至全部行走路线。
(2) 探讨每次选择位置的“最佳策略”，以减少回溯的次数。
(3) 演示寻找行走路线的回溯过程。

需要写出一个非递归的程序使马在以“日”字行走的过程中遍历一个 8×8 棋盘上的每一个点，下面讲解具体实现。

b 关键数据结构说明

/* 坐标 */ typedef struct int x, y; int block[8]; /* 下一步不可走的方向 */ Coord;
/* 历史记录 */ typedef struct Coord path[BOARD_SIZE*BOARD_SIZE]; /* 历史路径 */ int pointer; /* 栈指针 */ History;
其中 `Coord` 用来记录坐标，包括两个成员 `x` 和 `y`；`History` 用来记录到目前位置的所有历史记录，从第 0 步(初始坐标)开始计数，有两个成员：`path` 数组(历史路径)和 `pointer`(栈指针)，本质上是一个栈。

c 贪心算法

使用贪心算法，旨在求出马所在这个格子周围八个方向分别有几个下一步，将这 8 个具体的数值存放在 `next[8]` 数组中。每次优先选择下一步最少且合法(在棋盘内的空格子)的方向对应的点进行入栈 `push()` 操作。

入栈操作 `push()` 与教材相同，都是通过指针的移动实现。

Listing 1: 实现 `getnext()` 操作

```
/* 分别得到如果走向周围的 8 个点，那么将会有多少个下一步 */
static void
getNext(Coord coord, int* next, Point innerBoard[BOARD_SIZE][BOARD_SIZE])
{
    Coord coord_t, coord_tt;
    for (int i = 0; i < 8; i++) {
        UpdateDirection(&coord, &coord_t, i);
        if (isValid(coord_t, innerBoard)) {
            for (int j = 0; j < 8; j++) {
                UpdateDirection(&coord_t, &coord_tt, j);
                if (isValid(coord_tt, innerBoard))
                    next[i]++;
            }
        }
    }
}
```

利用 `for` 循环, 求出每个方向周围都有几个合法的点可以走, 存在 `next[8]` 中。

Listing 2: 对 `next` 中值实现从小到大的排序操作

```
/*
** 由贪心算法，我们要给 next 数组进行非递减排序
** 并将具体方向对应地编号写到 seq 数组中
** 例如 next[3] = {5, 8, 2}
** 那么 seq [3] = {2, 0, 1}
*/
static void
getSeq(int* next, int* seq)
{
    int n[8];
    for (int i = 0; i < 8; i++)
        n[i] = next[i];

    for (int i = 0; i < 7; i++) {
        for (int j = 0; j < 7 - i; j++) {
            if (n[j] > n[j + 1]) {
                Swap(&n[j], &n[j + 1]);
                Swap(&seq[j], &seq[j + 1]);
            }
        }
    }
}
```

排在前面的即是优先级高的某一个方向, 注意 `next` 为 0 的会排在前面, 但是不会产生影响, 因为在贪心算法总体实现的 `getNextStep()` 中该方向会被 `if(isValid)` 筛出去。

Listing 3: 贪心算法的总实现

```
/* 更新 history，包括数组元素和栈指针 */
void
GetNextStep(Point innerBoard[BOARD_SIZE][BOARD_SIZE], History* history_ptr)
{
    Coord coord = history_ptr->path[history_ptr->pointer];

    int next[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    int  seq[8] = {0, 1, 2, 3, 4, 5, 6, 7};

    getNext(coord, next, innerBoard);
    getSeq(next, seq);

    for (int i = 0; i < 8; i++) {
        Coord coord_t;
        UpdateDirection(&coord, &coord_t, seq[i]);
        if (isValid(coord_t, innerBoard)) {
            push(coord_t, history_ptr);
            break;
        }
    }
}
```

2 棋盘设计说明

棋盘使用 UTF-8 字符实现，马的位置用 UTF-8 字符“马(国际象棋)”表示，棋盘也用 UTF-8 字符“田”表示，空白处用空格表示。不同类型字符用不同颜色表示，以区分。

此外，为了方便观察，我们在棋盘边上标出了每个格子的坐标，在顶部显示当前马的坐标，以及当前步数，以便于观察马的移动路径。

3 时空性能分析

a 时间复杂度

因为无论棋盘大小如何，遍历的时候都是只有 8 个不同的方向，所以对于单独一步的判断，均只需要 64 步，时间复杂度为 $\mathcal{O}(1)$，对于 $n \times n$ 大小的棋盘，所需时间复杂度为 $\mathcal{O}(n^2)$。

b 空间复杂度

需要用 $\mathcal{O}(n^2)$ 的栈来记录历史路径，贪心算法所占用空间可复用，所以总空间复杂度为 $\mathcal{O}(n^2)$。

4 实例演示及优越性分析

a 成果展示

图 1: 标题界面

图 2: 动画展示中某一时刻的截图

b KTP 完成度总结

由于 8×8 棋盘的对称性, 将棋盘以“田”字四等分, 我们只需要令左上角那一个 4×4 小块的一个上三角区中所有点作为初始点进行验证, 就可以得到是否以棋盘上 64 个点中任意一点作为起始点, 这个马踏棋盘的过程都有解。

通过我们的验证得知, 这个答案是肯定的, 这也说明了通过贪心算法, 我们可以找到最优路径(不需要悔棋), 既实现了选做 2 的要求, 又在某种程度上实现了选做 3。

c 亮点总结

1. 单步打印, 可以清晰看到马走完整个棋盘的全过程, 且每一步停留的延时可以自己调节
2. 健壮性:对于预期范围之外的输入能正确检测并进行处理, 不会出现死循环或者异常退出的情况
3. 通过纯 `C` 语言实现, 在命令行终端打印结果, 同时支持 `Linux` 和 `Windows` 环境, 资源占用小, 兼容性强
4. 可以通过修改宏改变棋盘大小和打印策略(延迟时间, 是否手动跟进下一步等)
5. 虽然在终端打印, 但并不简陋, 包含颜色丰富的标题界面和棋盘, 用 `UTF-8` 字符实现棋盘和马的打印

二、 平衡二叉树操作的演示

1 算法介绍

- 实验具体要求。

6.4⑤ 平衡二叉树操作的演示

【问题描述】

利用平衡二叉树实现一个动态查找表。

【基本要求】

实现动态查找表的三种基本功能：查找、插入和删除。

【测试数据】

由读者自行设定。

【实现提示】

(1) 初始,平衡二叉树为空树,操作界面给出查找、插入和删除三种操作供选择。每种操作均要提示输入关键字。每次插入或删除一个结点后,应更新平衡二叉树的显示。

(2) 平衡二叉树的显示可采用如 6.3 题要求的凹入表形式,也可以采用图形界面画出树形。

(3) 教科书已给出查找和插入算法,本题重点在于对删除算法的设计和实现。假设要删除关键字为 x 的结点。如果 x 不在叶子结点上,则用它左子树中的最大值或右子树中的最小值取代 x。如此反复取代,直到删除动作传递到某个叶子结点。删除叶子结点时,若需要进行平衡变换,可采用插入的平衡变换的反变换(如,左子树变矮对应于右子树长高)。

【选作内容】

(1) 合并两棵平衡二叉树。

(2) 把一棵平衡二叉树分裂为两棵平衡二叉树,使得在一棵树中的所有关键字都小于或等于 x,另一棵树中的任一关键字都大于 x。

图 3: 平衡二叉树操作的具体要求

我们需要实现的是平衡二叉树结点的插入、删除、查找, 以及选作内容中的合并两棵平衡二叉树和将一棵平衡二叉树分裂为两颗平衡二叉树并满足相应的要求。下面介绍各种算法的具体实现。

- 平衡二叉树的概念介绍。

平衡二叉树满足如下条件：左子树和右子树深度之差的绝对值不大于 1；左子树和右子树也都是平衡二叉树。特别地，空树也为平衡二叉树。

结点的平衡因子：该结点的左子树的深度减去其右子树深度。因此，平衡二叉树上每个结点的平衡因子只可能是 1、0 和-1。

在本实验中，我们使用的用来表示平衡二叉树的数据结构为：

Listing 4: 平衡二叉树数据结构

```
typedef int ElemType;
typedef struct AVLTreeNode {
    ElemType data;
    int bf; // 平衡因子
    struct AVLTreeNode *lchild, *rchild;
} *AVLTree;
```

与教材上的相同。

- 平衡化旋转

如果在一棵平衡的二叉排序树中插入一个新结点，造成了不平衡，那么必须调整树的结构，使之平衡化。每插入/删除一个新结点时，AVL 树中相关结点的平衡状态会发生改变。因此，在插入/删除一个新结点后，需要检查各结点的平衡因子，如果在某一结点发现不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点，进行平衡化旋转。

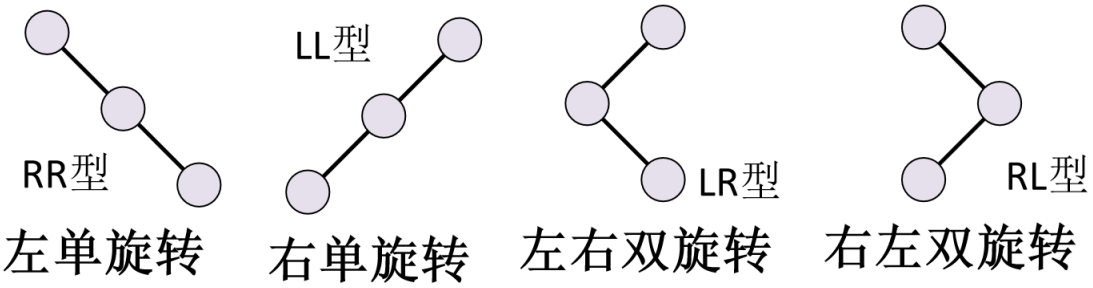

图 4: 平衡化分类

如上图所示，共有四种平衡化类型，但是由于后两种类型的平衡化是由前两种类型组合而成的，因此只需要实现前两种旋转即可：

Listing 5: 左单旋转和右单旋转

```
// 左旋转
void L_Rotate(AVLTree *T) {
    AVLTree rc;
    rc = (*T)->rchild;
    (*T)->rchild = rc->lchild;
    rc->lchild = *T;
    *T = rc;
}
```

```c
// 右旋转
void R_Rotate(AVLTree *T) {
    AVLTree lc;
    lc = (*T)->lchild;
    (*T)->lchild = lc->rchild;
    lc->rchild = *T;
    *T = lc;
}
```

具体的操作过程和上课介绍的一样, 在这里就不详细介绍了。

- 平衡二叉树的插入。

在向一棵本来是平衡的 AVL 树中插入一个新结点时, 需从插入结点沿通向根的路径向上回溯, 如果某个结点的平衡因子的绝对值 $|bf| > 1$, 那么需从这个结点出发, 使用平衡旋转方法进行平衡化处理。

如果一开始的平衡二叉树是空树, 那么我们需要获取一定的空间, 将待插入的关键字插入到根节点:

Listing 6: 空树的插入

```c
if (*T == NULL) {
    *T = (AVLTree)malloc(sizeof(struct AVLTreeNode));
    if (*T == NULL) return FALSE;
    (*T)->data = e;
    (*T)->lchild = (*T)->rchild = NULL;
    (*T)->bf = EH;
    *taller = TRUE;
}
```

如果待插入的关键字已经存在, 则无法插入:

Listing 7: 待插入的关键字已存在

```c
else if (e == (*T)->data) {
    *taller = FALSE;
    return FALSE;
}
```

排除以上特殊情况之后, 设新结点 p 的平衡因子为 0, 其父结点为 pr。插入新结点后, pr 的平衡因子值有三种情况:

结点 pr 的平衡因子为 0, 说明刚才是在 pr 的较矮的子树上插入了新结点, 此时不需做平衡化处理, 返回主程序, 子树的高度不变。

结点 pr 的平衡因子的绝对值 $|bf| = 1$, 说明插入前 pr 的平衡因子是 0, 插入新结点后, 以 pr 为根的子树不需平衡化旋转。但该子树高度增加, 还需从结点 pr 向根方向回溯, 继续考查结点 pr 双亲 $(pr = Parent(pr))$ 的平衡状态。

结点 pr 的平衡因子的绝对值 $|bf| = 2$ 说明新结点在较高的子树上插入, 造成了不平衡, 需要做平衡化旋转。此时可进一步分 2 种情况讨论:若结点 pr 的 bf = -2, 说明右子树高, 结合其右子女 q 的 bf 分别处理:若 q 的 bf 为-1, 执行左单旋转; 若 q 的 bf 为 1, 执行先右后左双旋转。若结点 pr 的 bf = 2, 说明左子树高, 结合其左子女 q 的 bf 分别处理:若 q 的 bf 为 1, 执行右单旋转; 若 q 的 bf 为-1, 执行先左后右双旋转。

Listing 8: 平衡二叉树的插入

```
else if (e < (*T)->data) {
    if (InsertAVL(&(*T)->lchild, e, taller) == FALSE)
        return FALSE;
    if (*taller == TRUE) {
        switch ((*T)->bf) {
            case LH:
                LeftBalance(T);
                *taller = FALSE;
                break;
            case RH:
                (*T)->bf = EH;
                *taller = FALSE;
                break;
            case EH:
                (*T)->bf = LH;
                *taller = TRUE;
                break;
        }
    }
} else {
    if (InsertAVL(&(*T)->rchild, e, taller) == FALSE)
        return FALSE;
    if (*taller == TRUE) {
        switch ((*T)->bf) {
            case RH:
                RightBalance(T);
                *taller = FALSE;
                break;
            case LH:
                (*T)->bf = EH;
                *taller = FALSE;
                break;
            case EH:
                (*T)->bf = RH;
                *taller = TRUE;
                break;
        }
    }
}
```

以上为刚刚讨论的三种情况的具体代码，可以看出主要是利用递归找到对应插入位置之后再完成相应的平衡化操作。注意这里面的 LeftBalance() 和 RightBalance() 函数则是判断平衡化的关键：

Listing 9: 插入后对左子树的平衡化处理

```
// 插入之后树 T 失衡时的平衡函数
void LeftBalance(AVLTree *T) {
    AVLTree lc, rc;
    lc = (*T)->lchild;
```

```
    switch (lc->bf) {
        case LH:
            (*T)->bf = lc->bf = EH;
            break;
        case RH:
            rc = lc->rchild;
            switch (rc->bf) {
                case LH:
                    (*T)->bf = RH;
                    lc->bf = EH;
                    break;
                case EH:
                    (*T)->bf = lc->bf = EH;
                    break;
                case RH:
                    (*T)->bf = EH;
                    lc->bf = LH;
                    break;
            }
            rc->bf = EH;
            L_Rotate(&(*T)->lchild);
            break;
    }
    R_Rotate(T);
}
```

Listing 10: 插入后对右子树的平衡化处理

```
// 插入之后右子树失衡时的平衡函数
void RightBalance(AVLTree *T) {
    AVLTree lc, rc;
    rc = (*T)->rchild;
    switch (rc->bf) {
        case RH:
            (*T)->bf = rc->bf = EH;
            break;
        case LH:
            lc = rc->lchild;
            switch (lc->bf) {
                case RH:
                    (*T)->bf = LH;
                    rc->bf = EH;
                    break;
                case EH:
                    (*T)->bf = rc->bf = EH;
                    break;
                case LH:
                    (*T)->bf = EH;
```

```
                rc->bf = RH;
                break;
            }
            lc->bf = EH;
            R_Rotate(&(*T)->rchild);
            break;
    }
    L_Rotate(T);
}
```

- 平衡二叉树的查找。

Listing 11: 平衡二叉树的查找

```
AVLTree SearchAVLTree(AVLTree T, ElemType key) {
    if (T == NULL)
        return NULL;
    if (T->data == key)
        return T;
    if (T->data > key)
        return SearchAVLTree(T->lchild, key);
    else
        return SearchAVLTree(T->rchild, key);
}
```

如上, 平衡二叉树的查找算法就是根据带查找关键字大小和当前结点关键字大小比较, 然后利用递归进行查找。

- 平衡二叉树的删除

如果被删结点 x 最多只有一个子女, 可做简单地将结点 x 从树中删去。因为结点 x 最多有一个子女, 可以简单地把 x 的双亲中原来指向 x 的指针改指到 x 的这个子女结点。如果结点 x 没有子女, x 双亲原来指向 x 的指针置为 NULL。再将原来以结点 x 为根的子树的高度减 1。

如果被删结点 x 有两个子女, 搜索 x 在中序次序下的直接前驱 y(同样可以找直接后继), 把结点 y 的内容传送给结点 x, 现在问题转移到删除结点 y。把结点 y 当作被删结点 x, 因为结点 y 最多有一个子女, 可以简单地用上述最多只有一个子女的情况进行删除, 但是必须沿结点 x 通向根的路径反向追踪高度的变化对路径上各个结点的影响。

具体分为三种情况:

当前结点 p 的 bf 为 0:如果它的左子树或右子树被缩短, 则它的 bf 改为 1 或-1, 同时, shorter 置为 False, 这种情况不用旋转。

结点 p 的 bf 不为 0 且较高的子树被缩短:则 p 的 bf 改为 0, 同时 shorter 置为 True, 这种情况也不用旋转。

结点 p 的 bf 不为 0, 且较矮的子树又被缩短, 则在结点 p 发生不平衡, 需要进行平衡化旋转来恢复平衡: 如果 q(较高的子树)的 bf 为 0, 执行一个单旋转来恢复结点 p 的平衡, 置 shorter 为 False, 无需检查上层结点的平衡因子;如果 q bf 与 p 的 bf 相同, 则执行一个单旋转来恢复平衡, 结点 p 和 q 的 bf 均改为 0, 同时置 shorter 为 True, 还要继续检查上层结点的平衡因子;如果 p 与 q 的 bf 相反, 则执行一个双旋转来恢复平衡。先围绕 q 转再围绕 p 转, 新根结点的 bf 置为 0, 其他结点的 bf 相应处理, 同时置 shorter 为 True。

Listing 12: 平衡二叉树的删除

```
if (e < (*T)->data) {
    if (DeleteAVL(&(*T)->lchild, e, shorter) == FALSE)
        return FALSE;
    if (*shorter == TRUE) {
        switch ((*T)->bf) {
            case LH:
                (*T)->bf = EH;
                *shorter = TRUE;
                break;
            case EH:
                (*T)->bf = RH;
                *shorter = FALSE;
                break;
            case RH:
                *shorter = RightDeleteBalance(T);
                break;
        }
    }
} else if (e > (*T)->data) {
    if (DeleteAVL(&(*T)->rchild, e, shorter) == FALSE)
        return FALSE;
    if (*shorter == TRUE) {
        switch ((*T)->bf) {
            case RH:
                (*T)->bf = EH;
                *shorter = TRUE;
                break;
            case EH:
                (*T)->bf = LH;
                *shorter = FALSE;
                break;
            case LH:
                *shorter = LeftDeleteBalance(T);
                break;
        }
    }
} else {
    AVLTree p;
    p = *T;
    if ((*T)->lchild != NULL && (*T)->rchild != NULL) {
        p = (*T)->lchild;
        while (p->rchild != NULL) {
            p = p->rchild;
        }
        (*T)->data = p->data;
        DeleteAVL(&(*T)->lchild, p->data, shorter);
        if (*shorter == TRUE) {
            switch ((*T)->bf) {
                case LH:
                    (*T)->bf = EH;
```

```
                    *shorter = TRUE;
                    break;
                case EH:
                    (*T)->bf = RH;
                    *shorter = FALSE;
                    break;
                case RH:
                    *shorter = RightDeleteBalance(T);
                    break;
            }
        }
    } else {
        *T = ((*T)->lchild == NULL) ? (*T)->rchild : (*T)->lchild;
        free(p);
        *shorter = TRUE;
    }
}
```

以上为刚刚讨论的三种情况的具体代码，可以看出主要是利用递归找到应该删除的结点对应位置之后再完成相应的平衡化操作。注意这里面的 LeftDeleteBalance() 和 RightDeleteBalance() 函数则是判断平衡化的关键：

Listing 13: 删除后对左子树的平衡化处理

```
// 删除之后树 T 的左子树失衡时的平衡函数
Status LeftDeleteBalance(AVLTree *T) {
    AVLTree lc, rc;
    lc = (*T)->lchild;
    Status result;
    switch (lc->bf) {
        case LH:
            (*T)->bf = lc->bf = EH;
            result = TRUE;
            break;
        case RH:
            rc = lc->rchild;
            switch (rc->bf) {
                case LH:
                    (*T)->bf = RH;
                    lc->bf = EH;
                    break;
                case EH:
                    (*T)->bf = lc->bf = EH;
                    break;
                case RH:
                    (*T)->bf = EH;
                    lc->bf = LH;
                    break;
            }
            rc->bf = EH;
```

```
            L_Rotate(&(*T)->lchild);
            result = TRUE;
            break;
        case EH:
            (*T)->bf = LH;
            lc->bf = RH;
            result = FALSE;
            break;
    }
    R_Rotate(T);
    return result;
}
```

Listing 14: 删除后对右子树的平衡化处理

```
// 删除之后右子树失衡时的平衡函数
Status RightDeleteBalance(AVLTree *T) {
    AVLTree lc, rc;
    rc = (*T)->rchild;
    Status result;
    switch (rc->bf) {
        case RH:
            (*T)->bf = rc->bf = EH;
            result = TRUE;
            break;
        case LH:
            lc = rc->lchild;
            switch (lc->bf) {
                case RH:
                    (*T)->bf = LH;
                    rc->bf = EH;
                    break;
                case EH:
                    (*T)->bf = rc->bf = EH;
                    break;
                case LH:
                    (*T)->bf = EH;
                    rc->bf = RH;
                    break;
            }
            lc->bf = EH;
            R_Rotate(&(*T)->rchild);
            result = TRUE;
            break;
        case EH:
            (*T)->bf = RH;
            rc->bf = LH;
            result = FALSE;
```

```
            break;
    }
    L_Rotate(T);
    return result;
}
```

- 平衡二叉树的合并

关于二叉树的合并，常规思路是将一棵结点数较小的平衡二叉树中的结点一个一个的插入到结点数较大的平衡二叉树中，但这样就没有利用到两棵树都是平衡二叉树的有利条件，从而导致时间、空间复杂度的提高。因此，我们组想到的方法是，先用中序遍历将两棵二叉树存储到数组中，再利用有序数组的排序合并成一个有序数组，然后再根据该有序数组的特性，每次选择中间的关键字进行插入，然后进行递归，从而优化了时间和空间复杂度。

Listing 15: 平衡二叉树的合并

```
// 创建平衡二叉树的辅助函数
AVLTree CreateAVLFromSortedArray(ElemType *arr, int start, int end) {
    if (start > end) return NULL;
    int mid = (start + end) / 2;
    AVLTree node = (AVLTree)malloc(sizeof(struct AVLTreeNode));
    node->data = arr[mid];
    node->bf = EH;
    node->lchild = CreateAVLFromSortedArray(arr, start, mid - 1);
    node->rchild = CreateAVLFromSortedArray(arr, mid + 1, end);
    return node;
}

// 合并两个AVL树
void MergeAVLTree(AVLTree *T1, AVLTree T2) {
    if (T2 == NULL) return;
    int size1 = GetDepth(*T1) * 2;
    int size2 = GetDepth(T2) * 2;
    ElemType *arr1 = (ElemType*)malloc(size1 * sizeof(ElemType));
    ElemType *arr2 = (ElemType*)malloc(size2 * sizeof(ElemType));
    int index1 = 0, index2 = 0;
    InOrderTraversal(*T1, arr1, &index1);
    InOrderTraversal(T2, arr2, &index2);
    int newSize;
    ElemType *mergedArray = MergeSortedArrays(arr1, index1, arr2, index2, &newSize);
    *T1 = CreateAVLFromSortedArray(mergedArray, 0, newSize - 1);
    free(arr1);
    free(arr2);
    free(mergedArray);
}
```

上图为两个平衡二叉树合并的关键函数，CreateAVLFromSortedArray() 函数的主要作用是根据得到的有序数组通过求出中间关键字进行插入。MergeAVLTree() 则是合并的主函数。

- 平衡二叉树的分裂

平衡二叉树的分裂也是如此，常规思路是遍历整个二叉树，与标准比较，小于等于标准的插入一棵新的平衡二

叉树，大于标准的插入另一棵平衡二叉树，但这样也没有充分利用平衡二叉树的有利条件，因此为了降低时间和空间复杂度，我们采用中序遍历平衡二叉树得到相对应的有序数组，然后将数组分成两个有序数组，按照上述合并二叉树中的方式构造两棵新的平衡二叉树。

Listing 16: 平衡二叉树的分裂

```
// 分裂AVL树
void DivAVLTree(AVLTree R, AVLTree *T1, AVLTree *T2, ElemType e) {
    if (R == NULL) return;
    int size = GetDepth(R) * 2;
    ElemType *arr = (ElemType*)malloc(size * sizeof(ElemType));
    int index = 0;
    InOrderTraversal(R, arr, &index);
    int mid = 0;
    for (int i = 0; i < index; i++) {
        if (arr[i] > e) {
            mid = i;
            break;
        }
    }
    *T1 = CreateAVLFromSortedArray(arr, 0, mid - 1);
    *T2 = CreateAVLFromSortedArray(arr, mid, index - 1);
    free(arr);
}
```

2 时空性能分析

- 插入操作的时空复杂度。

  插入操作在 AVL 树中涉及查找正确的插入位置和可能的旋转操作来保持树的平衡。

  查找位置: 由于 AVL 树是平衡二叉搜索树，其查找时间复杂度为 O(log n)。

  旋转操作: 每次旋转的时间复杂度为 O(1)。

  因此，插入操作的总时间复杂度为 O(log n)。

  插入操作过程中没有额外的空间消耗，空间复杂度主要来源于递归调用的栈空间。递归调用的深度为 O(log n)，所以空间复杂度为 O(log n)。

- 查找操作的时空复杂度。

  因为 AVL 树是平衡的，每次比较都可以将查找空间减半，因此查找操作在 AVL 树中的时间复杂度为 O(log n)。

  查找操作过程中没有额外的空间消耗，空间复杂度主要来源于递归调用的栈空间。递归调用的深度为 O(log n)，所以空间复杂度为 O(log n)。

- 删除操作的时空复杂度

  删除操作在 AVL 树中涉及查找节点、删除节点和可能的旋转操作来保持树的平衡。

  查找节点: 时间复杂度为 O(log n)。

  删除节点: 如果要删除的节点有两个子节点，需要找到前驱或后继节点，操作的时间复杂度为 O(log n)。

  旋转操作: 每次旋转的时间复杂度为 O(1)。

  因此，删除操作的总时间复杂度为 O(log n)。

删除操作过程中没有额外的空间消耗, 空间复杂度主要来源于递归调用的栈空间。递归调用的深度为 O(log n), 所以空间复杂度为 O(log n)。

- 合并操作的时空复杂度

  合并操作包括以下步骤:

  中序遍历两棵树: 每棵树的中序遍历时间复杂度为 O(n1) 和 O(n2), 其中 n1 和 n2 分别是两棵树的节点数。

  合并两个有序数组: 时间复杂度为 O(n1 + n2)。

  从合并后的有序数组创建新的 AVL 树: 时间复杂度为 O(n1 + n2)。

  总时间复杂度为 O(n1 + n2)。

  需要额外的空间存储中序遍历的结果和合并后的有序数组, 空间复杂度为 O(n1 + n2)。

- 合并操作的时空复杂度

  分裂操作包括以下步骤:

  中序遍历树: 时间复杂度为 O(n), 其中 n 是树的节点数。

  分割有序数组: 时间复杂度为 O(n)。

  从分割后的两个有序数组创建新的 AVL 树: 时间复杂度为 O(n)。

  因此, 总时间复杂度为 O(n)。

  需要额外的空间存储中序遍历的结果和分割后的有序数组, 空间复杂度为 O(n)。

3 实例演示及优越性分析

- 插入、删除、查找操作的实例演示。

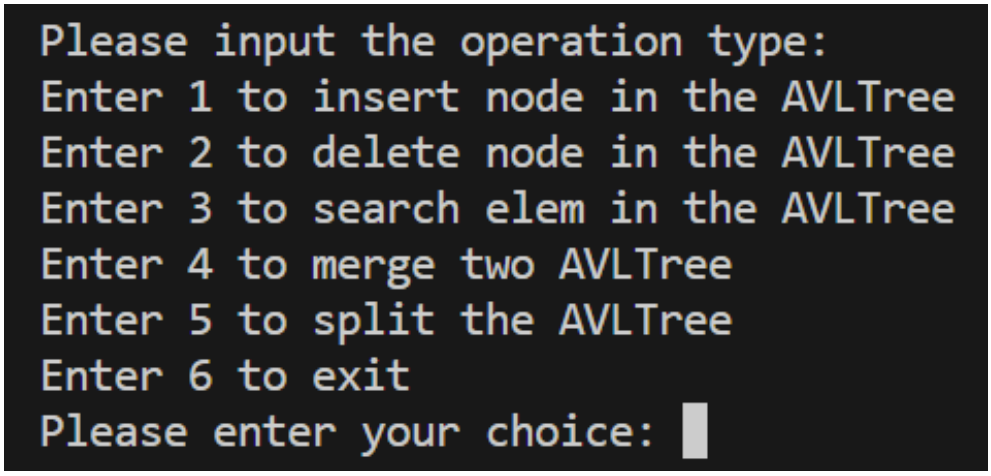

图 5: 终端输入界面

上图为程序执行过程中用户看到的终端输入界面的提示, 用户选择想要进行的操作并输入相应的数字即可继续下一步操作。

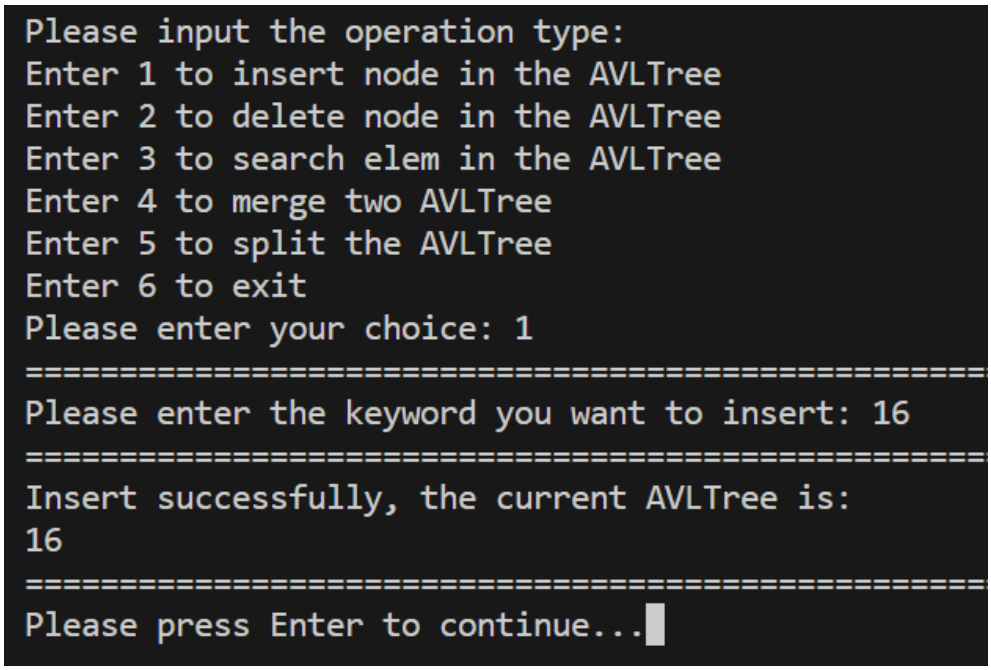

图 6: 用户完成插入操作后的界面

提示用户输入需要插入的关键字后得出相应的平衡二叉树, 然后用户需要按回车键继续下一步的操作。我们提供了清屏功能, 以保持当前界面始终保持整洁、美观, 以免用户难以发现关键信息。

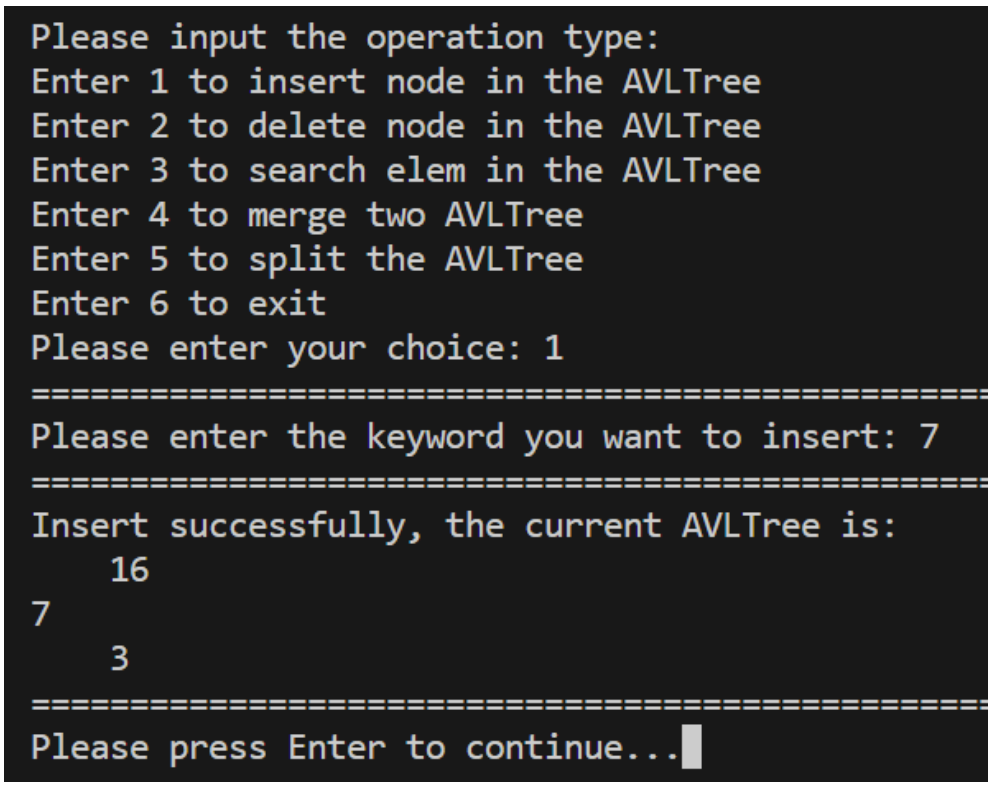

图 7: 完成插入并平衡化之后的界面

如图所示, 当我们依次插入 16, 3, 7 后, 本应得到的二叉树是非平衡的, 然后进行了平衡化处理, 详细过程如下图所示:

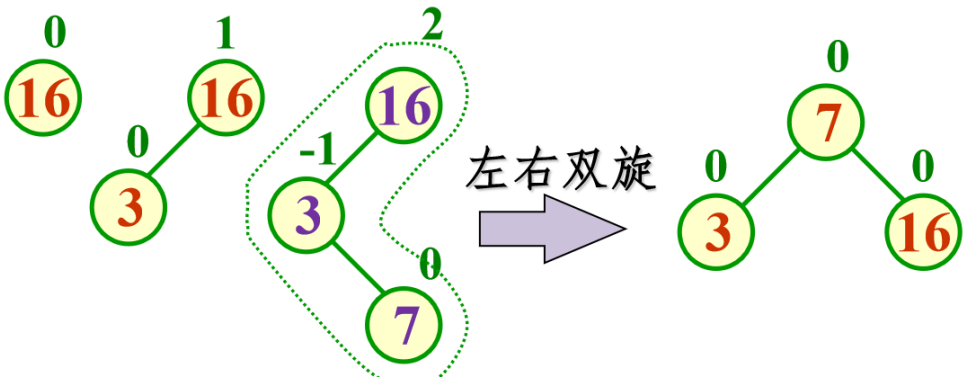

图 8: 平衡化详细过程

然后依次插入 11, 9, 26, 18, 14, 15, 最终得到的结果为:

```
Please input the operation type:
Enter 1 to insert node in the AVLTree
Enter 2 to delete node in the AVLTree
Enter 3 to search elem in the AVLTree
Enter 4 to merge two AVLTree
Enter 5 to split the AVLTree
Enter 6 to exit
Please enter your choice: 1
========================================================
Please enter the keyword you want to insert: 15
========================================================
Insert successfully, the current AVLTree is:
        26
    18
            16
        15
            14
11
        9
    7
        3
========================================================
Please press Enter to continue...
```

图 9: 完成插入所有关键字后的界面

结果和 PPT 上例题中的结果一致。
注意到这里我们对于使用凹入表打印的优化, 在 OJ 中的凹入表打印是如下图所示:

输入样例 1

```
A 2 -1
B 5 3
C 7 4
D -1 -1
E -1 6
F -1 -1
G -1 -1
```

输出样例 1

```
A
-B
--E
--F
-C
--G
-D
```

图 10: 传统凹入表打印样式

虽然“-”的个数表示结点对应的层数, 但是同一层的结点是分开的, 不够直观的显示平衡二叉树的结构。

我们进行的优化是将树状图横置过来打印, 即每一列表示平衡二叉树的每一层, 结点右下方的结点为其左子节点, 右上方的结点为其右子节点, 能够直观的显示出平衡二叉树的特点。

关于删除结点, 我们以删除叶子结点和根节点为例:

```
Please input the operation type:
Enter 1 to insert node in the AVLTree
Enter 2 to delete node in the AVLTree
Enter 3 to search elem in the AVLTree
Enter 4 to merge two AVLTree
Enter 5 to split the AVLTree
Enter 6 to exit
Please enter your choice: 2
==================================================
Please enter the keyword you want to delete: 14
==================================================
Delete successfully, the current AVLTree is:
        26
    18
            16
        15
11
        9
    7
        3
==================================================
Please press Enter to continue...
```

图 11: 删除叶子结点后的界面

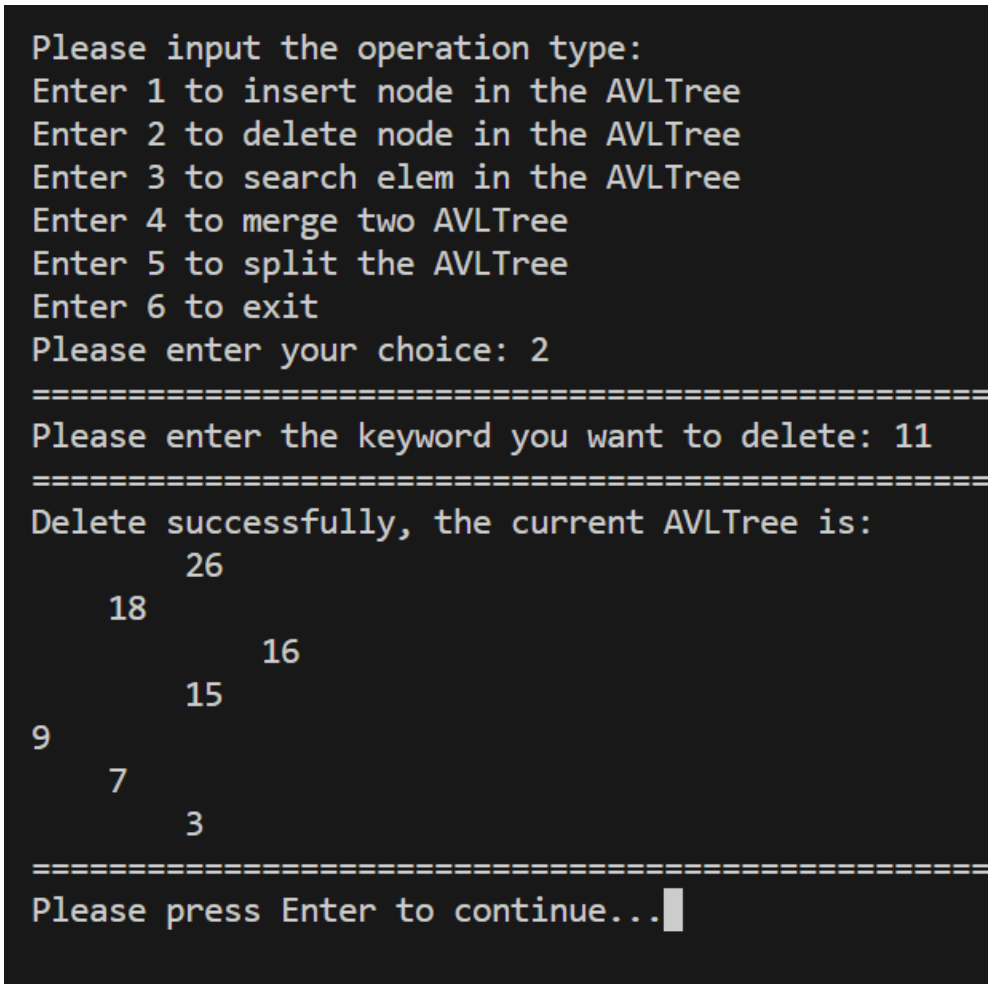

图 12: 删除根结点后的界面

可以看出, 删除叶子节点对于平衡没有影响, 因此无需平衡化处理。

而删除根节点, 则需要搜索其在中序次序下的直接前驱 y, 把结点 y 的内容传送给根节点, 然后将问题转移到删除结点 y, 再进行平衡化处理。

对于查找操作的结果, 则是若查找成功则输出查找成功, 以及查找的关键字, 若查找失败, 则输出查找失败, 关键字不存在的提示。

- 合并、分裂操作的实例演示。

合并操作要求我们输入两棵平衡二叉树的数据, 并不一定要按照平衡二叉树的排序来输入, 可以任意输入, 然后程序会首先将其生成两棵平衡二叉树, 再进行合并。

同时我们会将合并前的两棵平衡二叉树进行输出, 以便用户观察和比较:

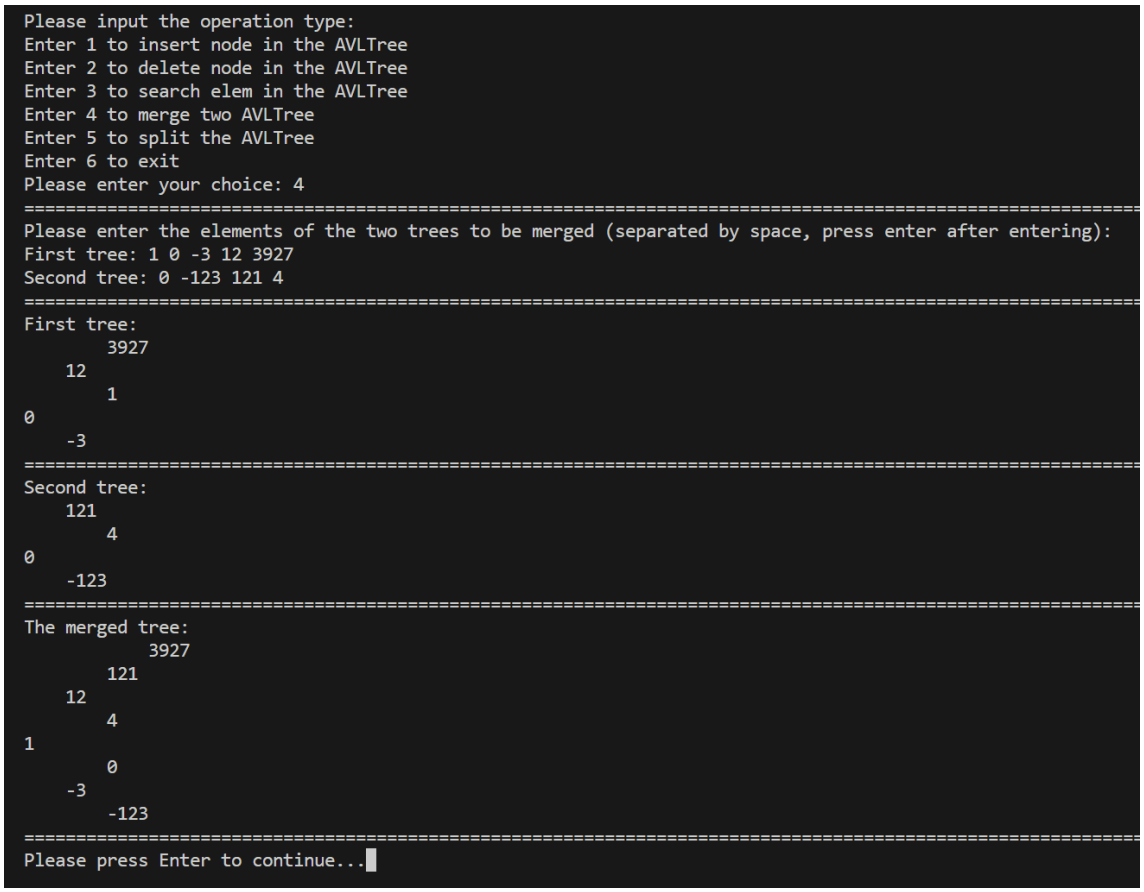

图 13: 合并操作完成后的界面

上述合并操作的结果经验证为正确结果。

对于分裂操作,我们首先输入待分裂的平衡二叉树的数据,也不要求按照平衡二叉树的排序来输入,可以任意输入,然后程序会首先将其生成平衡二叉树,再输入分裂的标准关键字,然后程序根据标准关键字将该平衡二叉树分裂成满足要求的两棵平衡二叉树。

同样的,我们会输出分裂前和分裂后的平衡二叉树供用户观察和比较:

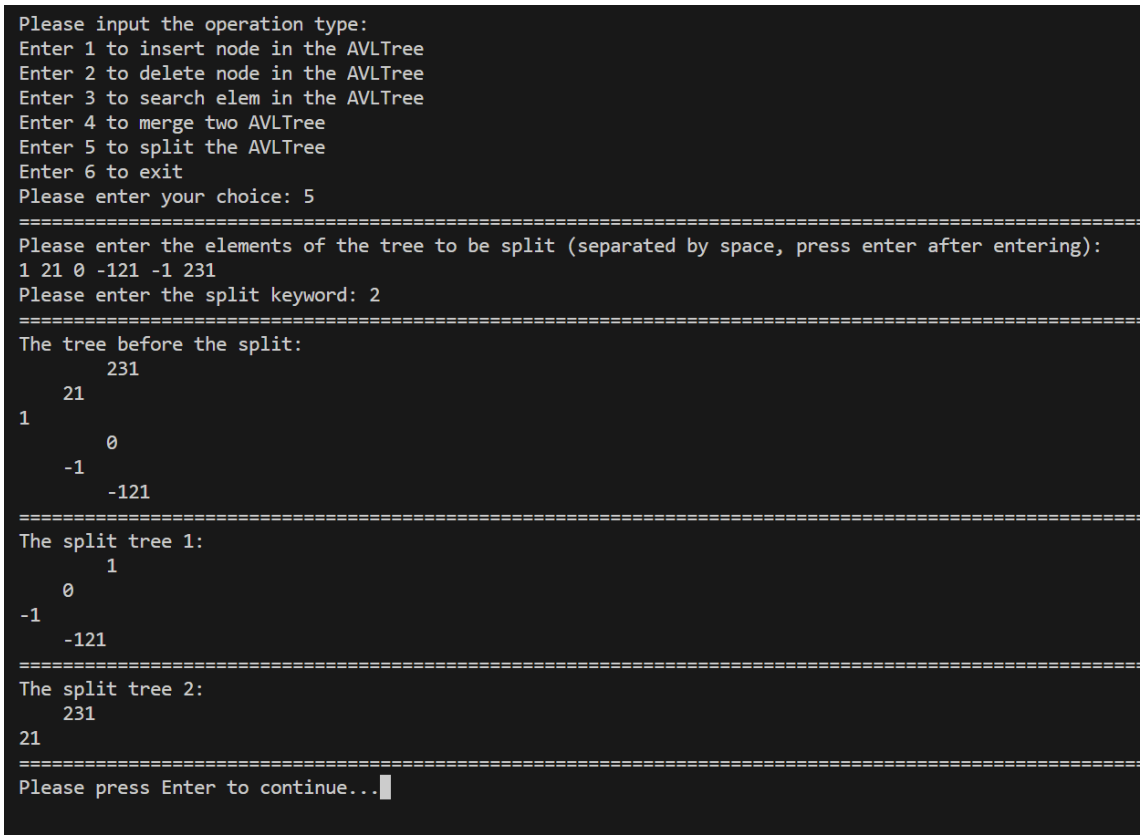

图 14: 分裂操作完成后的界面

上述分裂操作的结果经验证也是正确的。

- 优越性分析

本实验中最大的优化就是降低了合并操作和分裂操作的时间复杂度。

按照原方法的合并操作, 合并操作遍历树 T2 中的每个节点, 并将其插入到 T1 中。T2 中的每个节点插入到 T1 中的时间复杂度为 O(log n1), 其中 n1 是 T1 中的节点数。

因此, 合并操作的时间复杂度为 O(n2*log n1), 其中 n2 是 T2 中的节点数。

合并操作的递归调用栈的深度为 O(log n2), 此外在插入结点时递归调用栈的深度为 O(log n1), 且需要额外增加 O(n2) 个结点。

因此, 总空间复杂度为 O(log n1 + log n2 + n2), 即 O(n) 的空间复杂度。

而优化后的算法时间空间复杂度均为 O(n1 + n2), 在空间上二者相当, 但是在时间上有较大的优化。

同样的, 对于分裂操作, 原方法分裂操作遍历树 R 中的每个节点, 并将其插入到 T1 或 T2 中。R 中的每个节点插入到 T1 或 T2 中的时间复杂度为 O(log n), 其中 n 是 T1 或 T2 中的节点数。

因此, 分裂操作的时间复杂度为 O(n*log n), 其中 n 是 R 中的节点数。

分裂操作的递归调用栈的深度为 O(log n), 此外在插入结点时需要额外增加 O(n) 个结点。

因此, 总空间复杂度为 O(n)。

而优化后的算法时间空间复杂度均为 O(n1 + n2), 在空间上二者相当, 但是在时间上同样有较大的优化。