CS205 C/C++ Programming - Lab Assignment2 Report

Name: 邱逸伦 （Qiu Yilun）

SID: 12013006

Part 1 - Analysis

树，作为一种非连续的重要结构，在计算机科学中有着重要的应用。树作为数据结构往往是由节点和连接两个节点之间的边组成，每个节点都会通过边与至多一个父节点和不定数个子节点相连。二叉树是最为广泛运用的树结构，其最主要的特征是每个节点都至多只有两个子节点，这两个子节点根据方向分别作为该节点的左子节点和右子节点。二叉树的种类繁多，使用最为广泛的当属二叉搜索树和哈夫曼树。在本次作业中将聚焦于二叉搜索树的性质与实现。

在此先列举出关于二叉搜索树涉及的一些概念的定义：

祖先节点：某节点u是节点v的祖先节点当且仅当①$u=v$②u是v的父节点③u是v的祖先的父节点。

根节点：树中所有节点的祖先节点。

节点的值：节点中存储的值。不同节点间值的大小比较遵从该树的定义。

节点的深度：当前节点到根节点的边数，此处定义为H。

二叉搜索树是一种高效对大量统计数据进行处理的数据结构。二叉搜索树最为显著地性质在于，对于一个二叉树中的节点，该节点的值比它的左子节点的值大且比它的右子节点的值小。基于此性质，使其与数组相比，其插入与删除操作都是在O(1)的时间里得以实现；与链表相比，它的查找效率为O(H)。故而其在存储需要多次修改和查找的数据时是一个很优秀的数据结构。

但对于普通的有N个节点的二叉搜索树，如果存在每次插入都只在一个方向插入，此时再查找二叉搜索树的节点会使时间复杂度退化到O(N)，这并不是一个很好的实现。为解决此问题，平衡树应运而生。在本次作业中涉及的splay树即为平衡树的一种，它是在1985年由Daniel Sleator和Robert Endre Tarjan发明的，它的主要思想是：对于查找频率较高的点，使其处于离根节点较近的位置。splay树的思想使其在M此查询中会使得总时间复杂度为O(MlogN)，意味着单次查询操作的最坏时间复杂度可能为O(N)，但均摊时间复杂度能够优化到了O(logN)。标准的splay树应当在查找树中的节点即将当前查找到的节点通过zig/zag操作迁移到根节点的位置，在此作业中未实现，在此作业中实现了splay函数用于把传入的节点迁移到根节点的位置。

Part 2 - add_node(tree_node *father, tree_node *child, int child_direction)

在此函数中，传入的参数为tree_node指针`father`和`child`以及一个int数值`child_direction`。在处理完所有可能出现的异常后，该函数模拟了二叉树中插入节点的操作，根据`child_direction`的值，若`child_direction == CHILD_DIRECTION_LEFT`则将`child`作为`father`的左子节点，`father`作为`child`的父节点；`child_direction == CHILD_DIRECTION_RIGHT`则将`child`作为`father`的右子节点，`father`作为`child`的父节点。在成功建立`father`和~的关系后，通过循环向上更新`child`的所有祖先节点的`tree_count`，更新时调用了maintain函数。maintain函数是在此作业中另外定义的一个函数，该函数的具体实现如下所示。调用该函数能对当前节点`node`的`tree_count`进行相应修改，对应的`node->tree_count = node->node_count + node->l_child->tree_count + node->r_child->tree_count`（此时`node->l_child`和`node->r_child`都存在，若左子节点或右子节点不存在，则将其对应的`node->l_child->tree_count`或`node->r_child->tree_count`置为0）。

```
assign2_exception::exception maintain(tree_node *node) {
    if (!node)
        return NULL_POINTER_EXCEPTION;
    uint32_t count = node->node_count;
    if (node->l_child)
        count = count + node->l_child->tree_count;
    if (node->r_child)
        count = count + node->r_child->tree_count;
    node->tree_count = count;
    return 0;
}
```

Part 3 - judge_direction(tree_node *node, int *child_direction)

在此函数中，传入的参数tree_node指针`node`和int指针`direction`。在处理完所有可能出现的异常后，该函数将查找到`node`的父节点，并判断`node`是它的父节点的左子节点还是右子节点。若是左子节点，则将`direction`指针的值置为`CHILD_DIRECTION_LEFT`；若是右子节点，则将direction指针的值置为`CHILD_DIRECTION_RIGHT`。具体操作如下所示。

```
tree_node *father = node->father;
if (node == father->l_child)
    (*child_direction) = CHILD_DIRECTION_LEFT;
if (node == father->r_child)
    (*child_direction) = CHILD_DIRECTION_RIGHT;
```

Part 4 - insert_into_BST(BST *bst, uint64_t data, tree_node **inserted_node)

在此函数中传入的参数为BST指针`bst`，uint64_t类型的变量`data`和tree_node的二级指针`inserted_node`。该函数是用于将变量data存储的数据插入到BST（二叉搜索树）中。在处理了所有可能出现的异常后，若`bst->root == nullptr`，意味着`bst`是一棵空树，此时直接用`data`数据新建了一个tree_node指针`node`并将`bst->root`指针指向该新节点，最后令`*inserted_node = node`并直接结束函数，具体操作如下所示。

```
if (!bst->root) {
        auto *node = new tree_node{nullptr, nullptr, nullptr, 1, 1, data};
        bst->root = node;
        *inserted_node = node;
        return 0;
}
```

若传入的`bst`并不为一棵空树，则在函数中定义了一个初始值为`nullptr`的tree_node指针`pre`和初始值指向`bst->roo`t的tree_node指针`temp`。在处理时使用了一个while循环，该循环终止的条件为temp指针为`nullptr`。在循环外预先定义了int变量`direction`，while循环中先使`pre = temp`，再调用`bst`的`comp`函数指针让函数传入的参数`data`和`temp->data`进行比较。若结果小于0，则`data`小于`temp->data`，此时执行`temp = temp->l_child`和`direction = CHILD_DIRECTION_LEFT`；若结果大于0，则`data`小于`temp->data`，此时执行`temp = temp->r_child`和`direction = CHILD_DIRECTION_RIGHT`；若结果等于0，则`data`等于`temp->data`，此时则直接让`temp`节点的`node_count++`，通过循环向上更新`temp`的所有祖先节点的`tree_count`，更新时调用了maintain函数，最后令`*inserted_node = temp`并直接结束函数。

上述的while循环除强制退出的情况，中止的条件为`temp`指针不为`nullptr`。当上述的while循环退出时，根据`data`的值新建了一个tree_node节点（和上方代码操作相同，用`temp`指针存储）。此时根据`direction`的值进行判断，若`direction == CHILD_DIRECTION_LEFT`则令`pre`指针指向节点的左子节点指向`temp`；若`direction == CHILD_DIRECTION_RIGHT`则令`pre`指针指向节点的右子节点指向`temp`，最后让`temp`指针指向节点的父子节点指向`pre`。通过循环向上更新`temp`的所有祖先节点的`tree_count`，更新时调用了maintain函数，最后令`*inserted_node = temp`并直接结束函数。

Part 4 - find_in_BST(BST *bst, uint64_t data, tree_node **target_node)

在此函数中传入的参数为BST指针`bst`，uint64_t类型的变量`data`和tree_node的二级指针`target_node`。该函数是用于查找`data`的值是否有存储于`bst`中。在处理完所有可能出现的异常后，该函数的具体实现与insert_into_BST类似，定义了一个tree_node指针指向`bst->root`仍是在while循环中比较`data`和`temp->data`的值，若`data`的值等于`temp->data`的值，执行`(*target_node) = temp`操作并退出循环结束函数调用；若`data`的值大于`temp->data`的值，执行`temp = temp->r_child`操作；若`data`的值小于`temp->data`的值，执行`temp = temp->l_child`操作。若在循环中`temp`指针为`nullptr`，此时`bst`已经遍历到了叶子节点，未在`bst`中查找到`data`，则执行`(*target_node) = nullptr`操作并退出循环结束函数调用。

Part 5 - splay(BST *bst, tree_node *node)

在此函数中传入的参数为BST指针`bst`和tree_node指针`node`，该函数是用于实现将`bst`上的`node`节点迁移到`bst`根节点的位置，当`node`即为`bst`的根节点时，直接结束函数调用。在处理完所有可能的异常后，函数使用了一个while循环，循环中判断了当前节点`temp`的两种情况：节点的父亲是`bst->root`和其他情况。

当节点的父亲是`bst->root`时，先使用int指针`direction`调用了judge_direction(temp, direction)函数判断了当前节点是父节点的左子节点还是右子节点，`direction`指针用于存储了结果的值。若当前`(*direction) == CHILD_DIRECTION_LEFT`，则对当前节点进行zig操作；若当前`(*direction) == CHILD_DIRECTION_RIGHT`，则对当前节点进行zag操作。最后将`bst->root`更新为`temp`当前节点，并释放direction所占的内存空间结束函数调用。

与上述情况相反时，因初始化`temp`是否为`bst->root`的情况已经在循环外判断完，故而此时可以对当前节点`temp`和`temp->father`分别调用judge_direction方法，并用`up_dir`和`down_dir`的int指针分别存储了`temp`和`temp->father`是父子节点的左子节点还是右子节点。随后分了四种情况讨论：
①`(*up_dir) == CHILD_DIRECTION_LEFT && (*down_dir) == CHILD_DIRECTION_LEFT`，
②`(*up_dir) == CHILD_DIRECTION_LEFT && (*down_dir) == CHILD_DIRECTION_RIGHT`，
③`(*up_dir) == CHILD_DIRECTION_RIGHT && (*down_dir) == CHILD_DIRECTION_LEFT`，
④`(*up_dir) == CHILD_DIRECTION_RIGHT && (*down_dir) == CHILD_DIRECTION_RIGHT`。情况①先对`temp->father`执行zig操作，再对`temp`执行zig操作；情况②先对`temp`执行zag操作，再对`temp`执行zig操作；情况③先对`temp`执行zig操作，再对`temp`执行zag操作；情况④先对`temp->father`执行zag操作，再对`temp`执行zag操作。上述的操作的结果都使用assign2_exception::exception变量`e`存储。最后先释放`up_dir`和`down_dir`占用的内存空间，并判断`e`是否等于`ROOTS_FATHER_EXCEPTION`，若`e`等于`ROOTS_FATHER_EXCEPTION`，说明当前节点已经迁移到了`bst`的根节点位置，最后将`bst->root`更新为`temp`当前节点，结束函数调用。

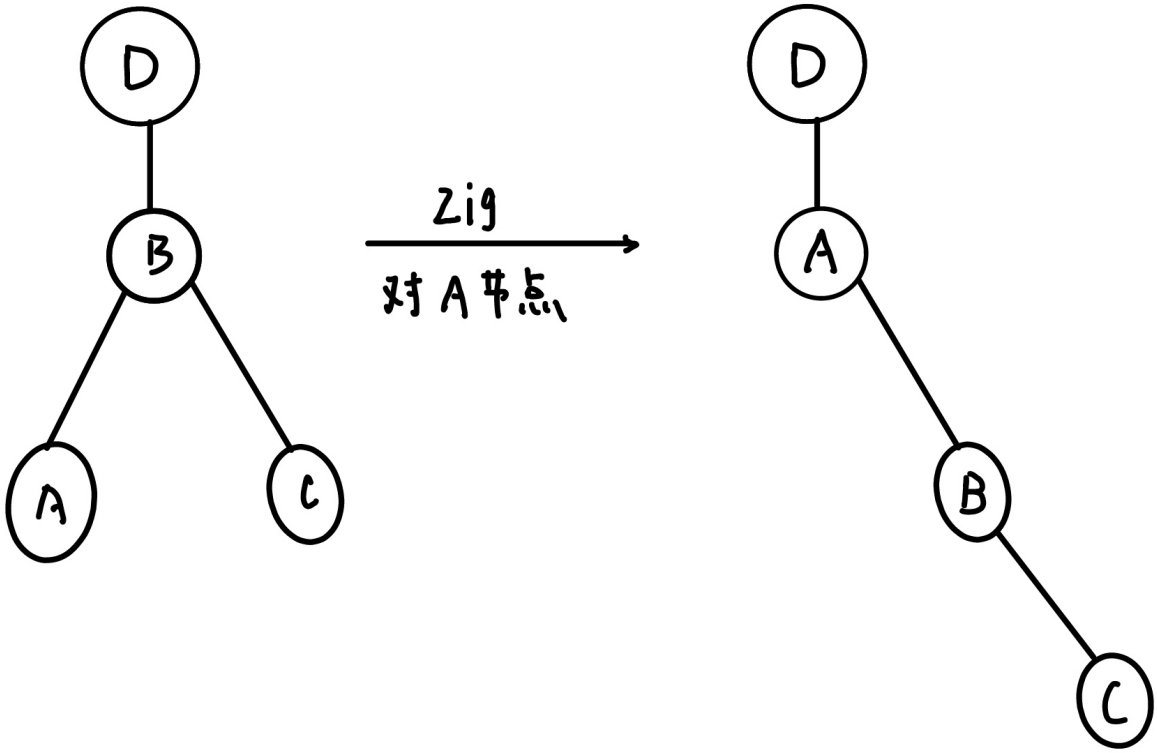
在该函数中使用到了zig和zag函数，两函数传入的参数都为tree_node指针变量，并对该指针变量做了相应的操作，此两函数的具体实现如下所示。

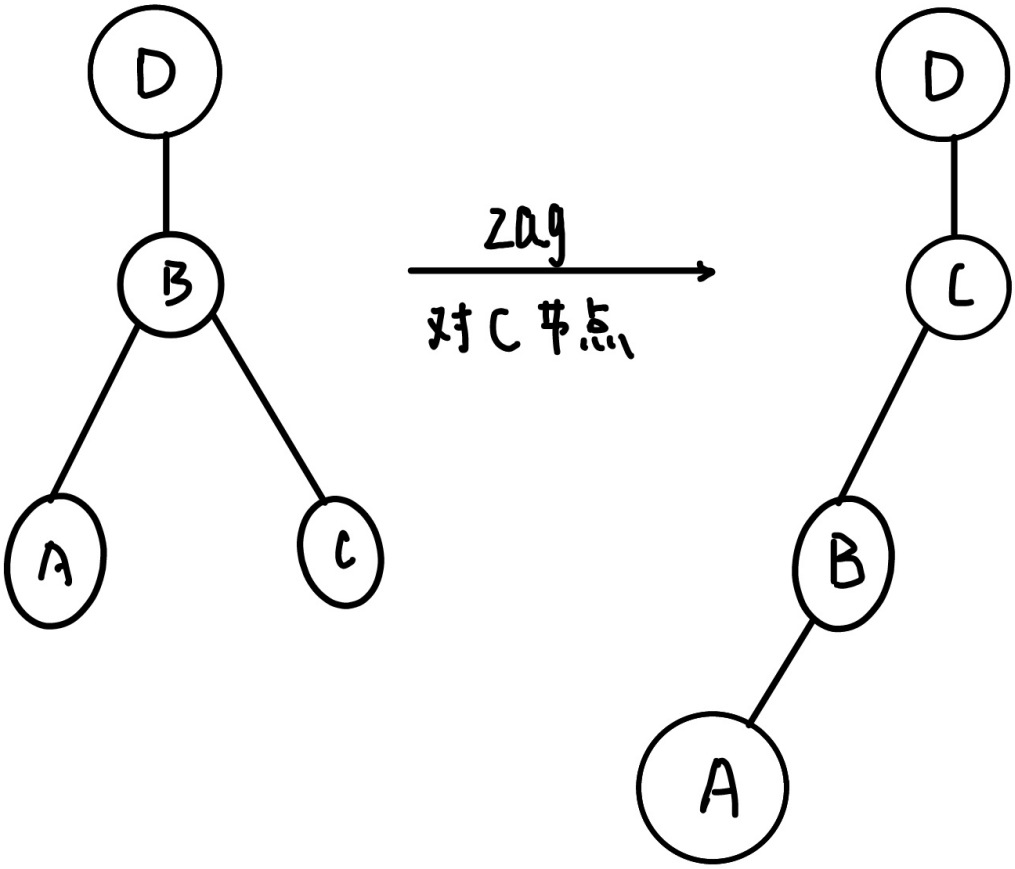
```
assign2_exception::exception zig(tree_node *node) {//Right Rotation
    if (!node || !(node->father))
        return NULL_POINTER_EXCEPTION;
```

```
    tree_node *temp = node->father;
    int *direction = new int;
    assign2_exception::exception e = 0;
    e |= judge_child_direction(temp, direction);
    temp->l_child = node->r_child;
    if (node->r_child)
        node->r_child->father = temp;
    node->r_child = temp;
    tree_node *grand = temp->father;
    temp->father = node;
    maintain(temp);
    maintain(node);
    if (e == ROOTS_FATHER_EXCEPTION) {
        node->father = nullptr;
        delete direction;
        return e;
    }
    if ((*direction) == CHILD_DIRECTION_LEFT)
        grand->l_child = node;
    if ((*direction) == CHILD_DIRECTION_RIGHT)
        grand->r_child = node;
    node->father = grand;
    delete direction;
    return 0;
}

assign2_exception::exception zag(tree_node *node) {//Left Rotation
    if (!node || !(node->father))
        return NULL_POINTER_EXCEPTION;
    tree_node *temp = node->father;
    int *direction = new int;
    assign2_exception::exception e = 0;
    e |= judge_child_direction(temp, direction);
    temp->r_child = node->l_child;
    if (node->l_child)
        node->l_child->father = temp;
    node->l_child = temp;
    tree_node *grand = temp->father;
    temp->father = node;
    maintain(temp);
    maintain(node);
    if (e == ROOTS_FATHER_EXCEPTION) {
        node->father = nullptr;
        delete direction;
        return e;
    }
    if ((*direction) == CHILD_DIRECTION_LEFT)
        grand->l_child = node;
    if ((*direction) == CHILD_DIRECTION_RIGHT)
        grand->r_child = node;
    node->father = grand;
    delete direction;
    return 0;
}
```

这两函数分别对应了对节点的右旋和左旋操作，目的是一步步将当前节点旋转到根节点的位置。两函数之间对节点的处理几乎镜像，具体的操作图示如下所示：





在代码中实现了这两种的旋转方式（zig，zag）。在此特殊之处在于在使用zig或zag函数时可能会出现上述旋转时D节点不存在的情况，换句话说，可能出现旋转前B节点是二叉搜索树的根节点的情况。在zig和zag函数中对这两种情况的处理为：对传入节点的父节点调用judge_direction函数来判断当前节点的父节点是其父节点的左子节点还是右子节点，因judge_direction函数处理了`ROOTS_FATHER_EXCEPTION`的异常，在调用时进行了异常的判断，若是出现了`ROOTS_FATHER_EXCEPTION`的异常证明当前节点的父节点是根节点，将异常在zig函数或zag函数中处理，将zig函数旋转后的A节点的父节点设为`nullptr`或将zag函数旋转后的C节点的父节点设为`nullptr`。在此两函数中，执行完旋转的操作也调用了maintain函数来更新传入节点`node`和原传入节点的父节点`temp`两节点的`tree_count`，在这里因为`node`被旋转为了`temp`的父节点，故而在函数中

先对`temp`调用了maintain函数，再对`node`调用了maintain函数。对于`node`节点的祖父节点的`tree_count`并没有受到`node`和`temp`更新的影响，故而此处不需要向上递归更新祖父节点的`tree_count`。

Part 5 - Final Code

```
#include "assign2.hpp"


assign2_exception::exception maintain(tree_node *node);


assign2_exception::exception zig(tree_node *node);


assign2_exception::exception zag(tree_node *node);


assign2_exception::exception add_node(tree_node *father, tree_node *child, int
child_direction) {
    assign2_exception::exception e = 0;
    if (!father || !child)
        e |= NULL_POINTER_EXCEPTION;
    if (child_direction != CHILD_DIRECTION_LEFT && child_direction !=
CHILD_DIRECTION_RIGHT)
        e |= INVALID_CHILD_DIRECTION_EXCEPTION;
    if (child && child->father)
        e |= DUPLICATED_FATHER_EXCEPTION;
    if (father && child_direction == CHILD_DIRECTION_LEFT && father->l_child)
        e |= DUPLICATED_LEFT_CHILD_EXCEPTION;
    if (father && child_direction == CHILD_DIRECTION_RIGHT && father->r_child)
        e |= DUPLICATED_RIGHT_CHILD_EXCEPTION;
    if (e != 0)
        return e;
    if (child_direction == CHILD_DIRECTION_LEFT)
        father->l_child = child;
    if (child_direction == CHILD_DIRECTION_RIGHT)
        father->r_child = child;
    child->father = father;
    tree_node *temp = father;
    while (temp) {
        (temp->tree_count)++;
        temp = temp->father;
    }
    return e;
}


assign2_exception::exception judge_child_direction(tree_node *node, int
*child_direction) {
    if (!node || !child_direction)
        return NULL_POINTER_EXCEPTION;
    if (!node->father)
        return ROOTS_FATHER_EXCEPTION;
    tree_node *father = node->father;
    if (node == father->l_child)
```

```
        (*child_direction) = CHILD_DIRECTION_LEFT;
    if (node == father->r_child)
        (*child_direction) = CHILD_DIRECTION_RIGHT;
    return 0;
}


assign2_exception::exception insert_into_BST(BST *bst, uint64_t data, tree_node
**inserted_node) {
    if (!bst)
        return NULL_POINTER_EXCEPTION;
    if (!bst->comp)
        return NULL_COMP_FUNCTION_EXCEPTION;
    if (!bst->root) {
        auto *node = new tree_node{nullptr, nullptr, nullptr, 1, 1, data};
        bst->root = node;
        *inserted_node = node;
        return 0;
    }
    tree_node *pre = nullptr;
    tree_node *temp = bst->root;
    auto comp_func = bst->comp;
    int direction = 0;
    while (temp) {
        pre = temp;
        int comp = comp_func(data, temp->data);
        if (comp < 0) {
            temp = temp->l_child;
            direction = CHILD_DIRECTION_LEFT;
        } else if (comp > 0) {
            temp = temp->r_child;
            direction = CHILD_DIRECTION_RIGHT;
        } else {
            assign2_exception::exception e = 0;
            (temp->node_count)++;
            (*inserted_node) = temp;
            while (temp) {
                e |= maintain(temp);
                temp = temp->father;
            }
            return e;
        }
    }
    temp = new tree_node{nullptr, nullptr, nullptr, 1, 1, data};
    if (direction == CHILD_DIRECTION_LEFT)
        pre->l_child = temp;
    if (direction == CHILD_DIRECTION_RIGHT)
        pre->r_child = temp;
    temp->father = pre;
    while (pre) {
        maintain(pre);
        pre = pre->father;
    }
    (*inserted_node) = temp;
    return 0;
}
```

```
assign2_exception::exception find_in_BST(BST *bst, uint64_t data, tree_node
**target_node) {
    if (!bst)
        return NULL_POINTER_EXCEPTION;
    if (!(bst->comp))
        return NULL_COMP_FUNCTION_EXCEPTION;
    tree_node *temp = bst->root;
    auto comp_func = (bst->comp);
    while (true) {
        if (!temp) {
            (*target_node) = nullptr;
            break;
        }
        int comp = comp_func(data, temp->data);
        if (comp == 0) {
            (*target_node) = temp;
            break;
        } else if (comp > 0)
            temp = temp->r_child;
        else
            temp = temp->l_child;
    }
    return 0;
}


assign2_exception::exception maintain(tree_node *node) {
    if (!node)
        return NULL_POINTER_EXCEPTION;
    uint32_t count = node->node_count;
    if (node->l_child)
        count = count + node->l_child->tree_count;
    if (node->r_child)
        count = count + node->r_child->tree_count;
    node->tree_count = count;
    return 0;
}


assign2_exception::exception zig(tree_node *node) {//Right Rotation
    if (!node || !(node->father))
        return NULL_POINTER_EXCEPTION;
    tree_node *temp = node->father;
    int *direction = new int;
    assign2_exception::exception e = 0;
    e |= judge_child_direction(temp, direction);
    temp->l_child = node->r_child;
    if (node->r_child)
        node->r_child->father = temp;
    node->r_child = temp;
    tree_node *grand = temp->father;
    temp->father = node;
    maintain(temp);
    maintain(node);
    if (e == ROOTS_FATHER_EXCEPTION) {
        node->father = nullptr;
        delete direction;
        return e;
```

```
    }
    if ((*direction) == CHILD_DIRECTION_LEFT)
        grand->l_child = node;
    if ((*direction) == CHILD_DIRECTION_RIGHT)
        grand->r_child = node;
    node->father = grand;
    delete direction;
    return 0;
}


assign2_exception::exception zag(tree_node *node) {//Left Rotation
    if (!node || !(node->father))
        return NULL_POINTER_EXCEPTION;
    tree_node *temp = node->father;
    int *direction = new int;
    assign2_exception::exception e = 0;
    e |= judge_child_direction(temp, direction);
    temp->r_child = node->l_child;
    if (node->l_child)
        node->l_child->father = temp;
    node->l_child = temp;
    tree_node *grand = temp->father;
    temp->father = node;
    maintain(temp);
    maintain(node);
    if (e == ROOTS_FATHER_EXCEPTION) {
        node->father = nullptr;
        delete direction;
        return e;
    }
    if ((*direction) == CHILD_DIRECTION_LEFT)
        grand->l_child = node;
    if ((*direction) == CHILD_DIRECTION_RIGHT)
        grand->r_child = node;
    node->father = grand;
    delete direction;
    return 0;
}


assign2_exception::exception splay(BST *bst, tree_node *node) {
    if (!bst)
        return NULL_POINTER_EXCEPTION;
    assign2_exception::exception e = 0;
    if (!node)
        e |= NULL_POINTER_EXCEPTION;
    if (!(bst->comp))
        e |= NULL_COMP_FUNCTION_EXCEPTION;
    if (node == (bst->root))//exclude the node is the toot itself
        return e;
    tree_node *temp = node;
    while (temp->father)
        temp = temp->father;
    if (temp != (bst->root))
        e |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
    if (e != 0)
        return e;
```

```
    temp = node;
    while (true) {
        if (temp->father == bst->root) {
            int *direction = new int;
            e |= judge_child_direction(temp, direction);
            if (e != 0) {
                delete direction;
                return e;
            }
            if ((*direction) == CHILD_DIRECTION_LEFT)
                e |= zig(temp);//Not an exception to handle
            if ((*direction) == CHILD_DIRECTION_RIGHT)
                e |= zag(temp);//Not an exception to handle
            bst->root = temp;
            delete direction;
            return 0;
        } else {
            int *up_dir = new int;
            int *down_dir = new int;
            e |= judge_child_direction(temp, down_dir);
            e |= judge_child_direction(temp->father, up_dir);
            if (e != 0) {//Not an exception to occur
                delete up_dir;
                delete down_dir;
                return e;
            }
            if ((*up_dir) == CHILD_DIRECTION_LEFT && (*down_dir) ==
CHILD_DIRECTION_LEFT) {
                e = zig(temp->father);
                e = zig(temp);
            } else if ((*up_dir) == CHILD_DIRECTION_LEFT && (*down_dir) ==
CHILD_DIRECTION_RIGHT) {
                e = zag(temp);
                e = zig(temp);
            } else if ((*up_dir) == CHILD_DIRECTION_RIGHT && (*down_dir) ==
CHILD_DIRECTION_LEFT) {
                e = zig(temp);
                e = zag(temp);
            } else {
                e = zag(temp->father);
                e = zag(temp);
            }
            delete up_dir;
            delete down_dir;
            if (e == ROOTS_FATHER_EXCEPTION) {
                bst->root = temp;
                return 0;
            }
        }
    }
}
```