CS205 C/ C++ Programming_Assignment2

Name: Lizinan

SID: 12011517

The test frame is from: https://github.com/CutieDeng/cpp_ass2_test

Part 1. Add Node

- ### Analysis

  Because we need to handle all exceptions. I have to write lots of if brach. Child nodes are added only if there are no exceptions. For convenience, I've added a method *maintain* for updating tree_count and avoiding NULL-pointer exception.

- ### Code

```
// update tree_count
void maintain(tree_node *node) {
    node->tree_count = node->node_count;
    if (node->l_child) {
        node->tree_count += node->l_child->tree_count;
    }
    if (node->r_child) {
        node->tree_count += node->r_child->tree_count;
    }
}
```

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

- ### Result & Verification

  Part of test code:

```
{
    tree_node *father = new tree_node;
    tree_node *child = new tree_node;
    tree_node *father_left = new tree_node;
    tree_node *child_father = new tree_node;

    father->l_child = father_left;
    child->father = child_father;

    assign2_exception::exception e = 0;
    e |= add_node(father, child, CHILD_DIRECTION_LEFT);
    try {
        if (e != 10) throw e;
        count++;
    } catch (assign2_exception::exception) {
        std::cout << "\033[41;11m Error in part 1, case2!  \033[0m\n";
    }
    delete father;
    delete child;
    delete father_left;
    delete child_father;
}
```

- ### Difficulties & Solutions

  None

Part 2. Judge Child Direction

- ### Analysis

  All the exception should be hadle and the direction need to be stored in *child_direction.

- ## Code

```
assign2_exception::exception judge_child_direction(tree_node *node, int *child_direction)
{
    assign2_exception::exception e = 0;
    // exception handle
    if (!node || !child_direction) {
        e += NULL_POINTER_EXCEPTION;
    }
    if (node && !node->father) {
        e += ROOTS_FATHER_EXCEPTION;
    }
    // operation
    if (e == 0) {
        if (node->father->l_child == node) {
            *child_direction = CHILD_DIRECTION_LEFT;
        } else {
            *child_direction = CHILD_DIRECTION_RIGHT;
        }
    }
    return e;
}
```

- ## Result & Verification

  Part of test code:

```
{
    tree_node *father = create_tree_node_test(2);
    tree_node *l_child = create_tree_node_test(1);
    tree_node *r_child = create_tree_node_test(3);

    int *child_direction = new int;
    assign2_exception::exception e = 0;

    e |= add_node(father, l_child, CHILD_DIRECTION_LEFT);
    e |= add_node(father, r_child, CHILD_DIRECTION_RIGHT);
    try {
        e |= judge_child_direction(l_child, child_direction);
        if (*child_direction != CHILD_DIRECTION_LEFT) throw e;
        e |= judge_child_direction(r_child, child_direction);
        if (*child_direction != CHILD_DIRECTION_RIGHT) throw e;
        count++;

    } catch (assign2_exception::exception) {
        std::cout << "\033[41;11m Error in part 2, case3! \033[0m";
    }
    delete father;
    delete l_child;
    delete r_child;
    delete child_direction;
}
```

- ### Difficulties & Solutions

  None

Part 3. Insert into Binary Search Tree

- ### Analysis

  In order to increase the simplicity of the code, I create a method to init the tree_node. Through comparing the size of data we can find the correct place to insert node. After insertion the tree_count of nodes are updated from bottom to top.

- ### Code

```
tree_node *init_node(tree_node *father, tree_node *child, uint64_t data) {
    child = new tree_node;
    child->father = father;
    child->data = data;
    child->l_child = NULL;
    child->r_child = NULL;
    child->tree_count = 1;
    child->node_count = 1;
    return child;
}
```

```
assign2_exception::exception insert_into_BST(BST *bst, uint64_t data, tree_node
**inserted_node) {
    assign2_exception::exception e = 0;
    // exception handle
    if (!bst || !inserted_node) {
        e += NULL_POINTER_EXCEPTION;
    }
    if (bst && !bst->comp) {
        e += NULL_COMP_FUNCTION_EXCEPTION;
    }
    // insert
    if (e == 0) {
        tree_node *tmp = bst->root;
        if (tmp) {
            while (1) {
                if (bst->comp(data, tmp->data) == 0) {
                    tmp->node_count++;
                    tmp->tree_count++;
                    break;
                } else if (bst->comp(data, tmp->data) > 0) {
                    if (tmp->r_child) {
                        tmp = tmp->r_child;
                    } else {
                        tmp->r_child = init_node(tmp, tmp->r_child, data);
                        tmp = tmp->r_child;
                        break;
                    }
                } else {
                    if (tmp->l_child) {
                        tmp = tmp->l_child;
                    } else {
                        tmp->l_child = init_node(tmp, tmp->l_child, data);
                        tmp = tmp->l_child;
                        break;
                    }
                }
            }
            *inserted_node = tmp;
            while (tmp->father) {
                tmp = tmp->father;
                maintain(tmp);
            }
        } else {
            bst->root = init_node(NULL, bst->root, data);
            *inserted_node = bst->root;
        }
    }
    return e;
}
```

- ## Result & Verification

Part of test code:

```
{
    assign2_exception::exception e = 0;
    tree_node *(*node) = new tree_node *;
    BST *bst = new BST;
    bst->root = nullptr;
    bst->comp = nullptr;
    try {
        e |= insert_into_BST(bst, 1, node);
        if (e != 64) throw e;
        count++;
    } catch (assign2_exception::exception) {
        std::cout << "\033[41;11m Error in part 3, case5! \033[0m\n";
    }
    delete node;
    delete_tree(bst->root);
    delete bst;
}
```

- ## Difficulties & Solutions

I met memory leakage problem at first. But soon I fix the problem.

Part 4. Find Element in Binary Search Tree

- ## Analysis

We can easily find the location of target_node through coparing until the pointer become NULL. If there is no such node in bst, the target_node pointer should be NULL.

- ## Code

```
assign2_exception::exception find_in_BST(BST *bst, uint64_t data, tree_node
**target_node) {
    assign2_exception::exception e = 0;
    if (!bst || !target_node) {
        e += NULL_POINTER_EXCEPTION;
    }
    if (bst && !bst->comp) {
        e += NULL_COMP_FUNCTION_EXCEPTION;
    }
    if (e == 0) {
        tree_node *tmp = bst->root;
        *target_node = NULL;
        while (tmp) {
            if (bst->comp(data, tmp->data) == 0) {
                *target_node = tmp;
                break;
            } else if (bst->comp(data, tmp->data) > 0) {
                tmp = tmp->r_child;
            } else {
                tmp = tmp->l_child;
            }
        }
    }
    return e;
}
```

- ## Result & Verification

Part of test code:

```
{
    assign2_exception::exception e = 0;
    try {
        BST *bst = new BST;
        tree_node *node = new tree_node;
        node->data = 50;
        node->l_child = NULL;
        node->r_child = NULL;
        node->father = NULL;
        node->node_count = 1;
        node->tree_count = 1;
        bst->root = node;
        bst->comp = comp;
        tree_node *targetnode;
        // std::cout<<"wrong\n";
        insert_into_BST(bst, 55, &targetnode);
        insert_into_BST(bst, 32, &targetnode);
        insert_into_BST(bst, 90, &targetnode);
        insert_into_BST(bst, 61, &targetnode);
        insert_into_BST(bst, 29, &targetnode);
        insert_into_BST(bst, 36, &targetnode);
        insert_into_BST(bst, 100, &targetnode);
        tree_node **target_node = new tree_node *;
        find_in_BST(bst, 1, target_node);
        if (*target_node != NULL) e = 1, throw e;  // judgement step 1.
        count++;
        find_in_BST(bst, 50, target_node);
        if (*target_node == NULL || (*target_node)->data != 50) delete_tree(bst->root),
delete bst, e = 2, throw e;  // judgement step 2.
        count++;
        find_in_BST(bst, 55, target_node);
        if (*target_node == NULL || (*target_node)->data != 55) delete_tree(bst->root),
delete bst, e = 3, throw e;  // judgement step 3.
        count++;
        find_in_BST(bst, 90, target_node);
        if (*target_node == NULL || (*target_node)->data != 90) delete_tree(bst->root),
delete bst, e = 4, throw e;  // judgement step 4.
        count++;
        find_in_BST(bst, 61, target_node);
        if (*target_node == NULL || (*target_node)->data != 61) delete_tree(bst->root),
delete bst, e = 5, throw e;  // judgement step 5.
        count++;
        find_in_BST(bst, 29, target_node);
        if (*target_node == NULL || (*target_node)->data != 29) delete_tree(bst->root),
delete bst, e = 6, throw e;  // judgement step 6.
        count++;
        find_in_BST(bst, 36, target_node);
        if (*target_node == NULL || (*target_node)->data != 36) delete_tree(bst->root),
delete bst, e = 7, throw e;  // judgement step 7.
        count++;
        find_in_BST(bst, 32, target_node);
        if (*target_node == NULL || (*target_node)->data != 32) delete_tree(bst->root),
delete bst, e = 8, throw e;  // judgement step 8.
        count++;
        delete_tree(bst->root);
        delete bst;
    } catch (assign2_exception::exception) {
        std::cout << "\033[41;11m Error in part 4, case1! Happen in judgement step " << e
<< ". \033[0m\n";
    }
}
```

- ### Difficulties & Solutions

  None

Part 5. Splay

- ### Analysis

For convienience, I create two functions named l_rotate & r_rotate. Using these functions we can easily complete all the operations. We need to pay extra attention to the judge of root node, it's easy to meet null pointer exception here.

- ## Code

```
void l_rotate(tree_node *node) {
    tree_node *tmp = node->father;
    tmp->r_child = node->l_child;
    if (node->l_child) {
        node->l_child->father = tmp;
    }
    node->father = tmp->father;
    if (tmp->father) {
        int *dir = new int;
        judge_child_direction(tmp, dir);
        if (*dir == CHILD_DIRECTION_LEFT) {
            tmp->father->l_child = node;
        } else {
            tmp->father->r_child = node;
        }
        delete dir;
    }
    node->l_child = tmp;
    tmp->father = node;
    maintain(tmp);
    maintain(node);
}

void r_rotate(tree_node *node) {
    tree_node *tmp = node->father;
    tmp->l_child = node->r_child;
    if (node->r_child) {
        node->r_child->father = tmp;
    }
    node->father = tmp->father;
    if (tmp->father) {
        int *dir = new int;
        judge_child_direction(tmp, dir);
        if (*dir == CHILD_DIRECTION_LEFT) {
            tmp->father->l_child = node;
        } else {
            tmp->father->r_child = node;
        }
        delete dir;
    }
    node->r_child = tmp;
    tmp->father = node;
    maintain(tmp);
    maintain(node);
}
```

```
assign2_exception::exception splay(BST *bst, tree_node *node) {
```

```
assign2_exception::exception e = 0;
// exception handle
if (!bst || !node) {
    e += NULL_POINTER_EXCEPTION;
}
if (bst && !bst->comp) {
    e += NULL_COMP_FUNCTION_EXCEPTION;
}
if (node && bst) {
    tree_node *tmp = node;
    while (tmp->father) {
        tmp = tmp->father;
    }
    if (tmp != bst->root) {
        e += SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
    }
}
if (e == 0) {
    int *dir1 = new int, *dir2 = new int;
    while (node->father) {
        if (!node->father->father) {
            break;
        }
        judge_child_direction(node, dir1);
        judge_child_direction(node->father, dir2);
        if (*dir1 == *dir2) {
            if (*dir1 == CHILD_DIRECTION_LEFT) {
                r_rotate(node->father);
                r_rotate(node);
            } else {
                l_rotate(node->father);
                l_rotate(node);
            }
        } else if (*dir1 == CHILD_DIRECTION_LEFT) {
            r_rotate(node);
            l_rotate(node);
        } else {
            l_rotate(node);
            r_rotate(node);
        }
    }
    if (node->father) {
        judge_child_direction(node, dir1);
        if (*dir1 == CHILD_DIRECTION_LEFT) {
            r_rotate(node);
        } else {
            l_rotate(node);
        }
    }
    bst->root = node;
    delete dir1;
    delete dir2;
}
return e;
```

```
}
```

- ## Result & Verification

Part of test code:

```
{
    assign2_exception::exception e = 0;
    try {
        BST *bst = new BST;
        tree_node *node = new tree_node;
        node->data = 50;
        node->l_child = NULL;
        node->r_child = NULL;
        node->father = NULL;
        node->node_count = 1;
        node->tree_count = 1;
        bst->root = node;
        bst->comp = comp;
        tree_node *targetnode;
        tree_node *test_node_one;
        tree_node *test_node_two;
        tree_node *test_node_three;
        // std::cout<<"wrong\n";
        insert_into_BST(bst, 55, &targetnode);
        insert_into_BST(bst, 32, &targetnode);
        insert_into_BST(bst, 90, &targetnode);
        insert_into_BST(bst, 61, &targetnode);
        insert_into_BST(bst, 61, &targetnode);
        insert_into_BST(bst, 61, &targetnode);
        insert_into_BST(bst, 61, &test_node_two);
        insert_into_BST(bst, 29, &test_node_one);
        insert_into_BST(bst, 36, &targetnode);
        insert_into_BST(bst, 36, &test_node_three);
        insert_into_BST(bst, 36, &targetnode);
        insert_into_BST(bst, 100, &targetnode);
        insert_into_BST(bst, 100, &targetnode);
        splay(bst, test_node_one);
        int tree_c = 0;
        if (!judge_bst(bst, tree_c) || bst->root != test_node_one) delete_tree(bst-
>root), delete bst, throw e = 1;
        count++;
        std::cout << "------------------\n";
        splay(bst, test_node_two);
        tree_c = 0;
        if (!judge_bst(bst, tree_c) || bst->root != test_node_two) delete_tree(bst-
>root), delete bst, throw e = 2;
        count++;
        splay(bst, test_node_three);
        tree_c = 0;
        if (!judge_bst(bst, tree_c) || bst->root != test_node_three) delete_tree(bst-
>root), delete bst, throw e = 3;
        count++;
        delete_tree(bst->root);
        delete bst;
    } catch (assign2_exception::exception) {
        std::cout << "\033[41;11m Error in part 5, case1! Happen in judgement step " << e << ". \033[0m\n";
    }
}
```

- ## Difficulties & Solutions

  NULL pointer, NULL pointer and NULL pointer. I waste plenty of time in fixing the termination condition of while loop.