# CS205 C/ C++ Programming - Lab Assignment TWO

**Name:** Chenyu Gu

**SID:** 12011123

## Introduction

In computer science, binary search tree (BST) is a sorted data structure, which can achieve dynamic data managing in average time complexity O(logn)O(logn).

In Assignment 2, you will be firstly required to implement some functions about tree nodes and binary search tree. At last, we will introduce a manipulation called *Splay*, which can adjust the structure of the tree when accessing specific element in the tree, bubbling the target to the root of the tree, so that the most frequently-accessed data will be close to the root – which can reduce the access time to the most frequently-used data.

If you are not major in computer science, you can briefly check some key concepts from:

- Tree [Wiki] [Baidu]
- Binary Tree [Wiki] [Baidu]
- Binary Search Tree [Wiki] [OI-Wiki]
- Splay [Wiki] [OI-Wiki]
- Tips: We cannot gurantee the academic correctness and authority of these materials. Please check the definitions from professional textbooks if mind.

These concept is very important for you to finish tasks in this assignment. Please make sure you have had general idea about them before you start.

## Task 1 - Add Node

1. For a binary tree, one of the most basic operation is adding node. We define function

   ```
   exception add_node(tree_node *father, tree_node *child, int child_direction)
   ```

   as adding node `{child}` to `{father}`'s corresponding child according to the `{child_direction}`. If `{child_direction}` equals `{CHILD_DIRECTION_LEFT}`, add `{child}` to `{father}`'s left child; and if `{child_direction}` equals `{CHILD_DIRECTION_RIGHT}`, add `{child}` to `{father}`'s right child.

   Exceptions you **SHOULD** handle:

   - `NULL_POINTER_EXCEPTION`: Handle this exception when {father} or {child} is nullptr.
   - `DUPLICATED_LEFT_CHILD_EXCEPTION`: Handle this exception when {father}'s left child is not nullptr while {child_direction} is CHILD_DIRECTION_LEFT.
   - `DUPLICATED_RIGHT_CHILD_EXCEPTION`: Handle this exception when {father}'s right child is not nullptr while {child_direction} is CHILD_DIRECTION_RIGHT.
   - `DUPLICATED_FATHER_EXCEPTION`: Handle this exception when {child}'s father is not nullptr.
   - `INVALID_CHILD_DIRECTION_EXCEPTION`: Handle this exception when {child_direction} is neither

CHILD_DICRETION_LEFT nor CHILD_DIRECTION_RIGHT.

What you **SHOULD** do:

- Implement

```
exception add_node(tree_node *father, tree_node *child, int child_direction)
```

in assign2.cpp

- Link {father} and {child} according to {child_direction}.
- Update {node_count} and {tree_count} from {father} to ***the root of the tree***.

# Part 1 - Analysis

> Because the return value of this function is the exception that contains all the exceptions, I initialize the e=0 for the exception. then use a series of the if judgment to judge whether it is a null pointer of the duplicate child.
>
> After the exception is handled, if the input is safe then we do the add node operation. We link the child to the father's child according to the child_direction, then let the child's father = father, after that we update the tree count and node count.

# Part 2 - Code

```
assign2_exception::exception add_node(tree_node *father, tree_node *child, int
child_direction)
{
    assign2_exception::exception e = 0;
    if (father == nullptr)
    {
        e |= NULL_POINTER_EXCEPTION;
    }
    if (child == nullptr)
    {
        e |= NULL_POINTER_EXCEPTION;
    }
    else if (child->father != nullptr)
    {
        e |= DUPLICATED_FATHER_EXCEPTION;
    }
    if (child_direction != CHILD_DIRECTION_LEFT && child_direction !=
CHILD_DIRECTION_RIGHT)
    {
        e |= INVALID_CHILD_DIRECTION_EXCEPTION;
    }
    if (father != nullptr && child != nullptr)
    {
```

```
        if (child->father != nullptr)
        {
            e |= DUPLICATED_FATHER_EXCEPTION;
        }
        else
        {
            child->father = father;
        }
        switch (child_direction)
        {
        case CHILD_DIRECTION_LEFT:
            if (father->l_child == nullptr)
            {
                father->l_child = child;
                father->tree_count += child->tree_count;
            }
            else
                e |= DUPLICATED_LEFT_CHILD_EXCEPTION;
            break;
        case CHILD_DIRECTION_RIGHT:
            if (father->r_child == nullptr)
            {
                father->r_child = child;
                father->tree_count += child->tree_count;
            }
            else
                e |= DUPLICATED_RIGHT_CHILD_EXCEPTION;
            break;
        default:
            e |= INVALID_CHILD_DIRECTION_EXCEPTION;
            break;
        }
    }
    return e;
}
```

## Part 3 - Result & Verification

Verification is used and thanks to CutieDeng, the verification test is in https://github.com/CutieDeng/cpp_ass2_test

```
正在进行第 0 组测试:
开始 PART 1 测试。
构造两个 tree_node 并执行 ADD NODE LEFT 方法。
测试通过!
====================

正在进行第 1 组测试:
```

```
构造两个 tree_node 并执行 ADD NODE RIGHT 方法。
测试通过!
====================

正在进行第 2 组测试:
构造三个 tree_node 并将其构建成一个小二叉树。
测试通过!
====================

正在进行第 3 组测试:
构建一个 tree_node 并添加左节点 nullptr.
测试通过!
====================

正在进行第 4 组测试:
构建一个 tree_node 并添加右节点 nullptr.
测试通过!
====================

正在进行第 5 组测试:
构建一个 tree_node, 并调用 ADD NODE LEFT(nullptr, node) 方法。
测试通过!
====================

正在进行第 6 组测试:
构建一个 tree_node, 并调用 ADD NODE RIGHT(nullptr, node) 方法。
测试通过!
====================

正在进行第 7 组测试:
构建一个 tree_node 并为其增加父节点后调用 ADD NODE LEFT(nullptr, node) 方法。
测试通过!
====================

正在进行第 8 组测试:
构建一个 tree_node 并调用 ADD NODE LEFT(nullptr, node) 为其设置父节点。
测试通过!
====================

正在进行第 9 组测试:
直接调用 add_node(nullptr, nullptr, 2).
测试通过!
====================

正在进行第 10 组测试:
构建 tree_node 并执行 add_node(node, node, 2).
测试通过!
====================
```

```
正在进行第 11 组测试：
构建 tree_node 并为其添加一个父节点，并执行 add_node(other_node, node, 53).
测试通过!
====================

正在进行第 12 组测试：
构建 tree_node parent and l_child, 建立关系后再次执行 ADD_NODE_LEFT(parent, l_child).
测试通过!
====================

正在进行第 13 组测试：
构建 tree_node parent and r_child, 建立关系后再次执行 ADD_NODE_RIGHT(parent, r_child).
测试通过!
====================

正在进行第 14 组测试：
构建一棵三层的二叉树。
测试通过!
====================

正在进行第 15 组测试：
构建一棵普通二层二叉树，并测试点的相连情形。
测试通过!
====================

本次测试到此结束。
====================

Program ended with exit code: 0
```

## Part 4 - Difficulties & Solutions

**Difficulties:** Exception is hard to contain all and without a throw, it may affect the operation, node count and tree count are easily omitted.

**Solutions:** Apply multiple if function and before doing the operation, use an if function:

```
    if (father != nullptr && child != nullptr)
```

and then pay attention to update node count and tree count:

```
    father->tree_count += child->tree_count;
```

# Task 2 - Judge Child Direction

- In this part, you are required to implement a function to judge the direction of a child to its father, i.e., whether a child is the left or right child of its father. We define function

  ```
  exception judge_child_direction(tree_node *node, int *child_direction)
  ```

  as judging whether the `{node}` is the left or right child of its father, and store the result to the address of `{child_direction}`.

  Exceptions you **SHOULD** handle:

  - `NULL_POINTER_EXCEPTION`: Handle this exception when {node} or {child_direction} is nullptr.
  - `ROOTS_FATHER_EXCEPTION`: Handle this exception when {node}'s father is nullptr, which means an anomaly-the user wants to judge the {child_direction} of a root of a tree.

  What you **SHOULD** do:

  - Implement `exception judge_child_direciton(tree_node *node, int *child_direction)` in `assign2.cpp`

# Part 1 - Analysis

> Because the return value of this function is the exception that contains all the exceptions, I initialize the e=0 for the exception. then use a series of the if judgment to judge whether it is a null pointer or roots father exception. And finally return the exception.
>
> After the exception is handled, if the input is safe then we do the operation. We split this into two situations and use multiply if judgements to handle this.

# Part 2 - Code

```
assign2_exception::exception judge_child_direction(tree_node *node, int
*child_direction)
{
    assign2_exception::exception e = 0;

    if (node == nullptr || child_direction == nullptr)
    {
        e |= NULL_POINTER_EXCEPTION;
    }

    if (node != nullptr && node->father == nullptr)
    {
        e |= ROOTS_FATHER_EXCEPTION;
    }
    if (node != nullptr && child_direction != nullptr && node->father != nullptr)
    {
        if (node->father->l_child == node)
        {
```

```
            *child_direction = CHILD_DIRECTION_LEFT;
        }
        if (node->father->r_child == node)
        {
            *child_direction = CHILD_DIRECTION_RIGHT;
        }
//        if (node->data > node->father->data)
//            *child_direction = CHILD_DIRECTION_RIGHT;
//        if (node->data < node->father->data)
//            *child_direction = CHILD_DIRECTION_LEFT;
    }
    return e;
}
```

## Part 3 - Result & Verification

Verification is used and thanks to CutieDeng, the verification test is in https://github.com/CutieDeng/cpp_ass2_test

```
正在进行第 0 组测试:
Part II 测试开始。
构建一个 tree_node 并询问它的 child direction.
测试通过!
====================

正在进行第 1 组测试:
询问 nullptr 的 child direction.
测试通过!
====================

正在进行第 2 组测试:
询问 nullptr 的 child direction, 并且不设置对应的答复内存空间。
测试通过!
====================

正在进行第 3 组测试:
构造节点 node 且询问 judge_child_direction(node, nullptr) 时的回答。
测试通过!
====================

正在进行第 4 组测试:
构造节点 node 为一个左子节点, 并调用 judge_child_direction(node, nullptr).
测试通过!
====================

正在进行第 5 组测试:
构造一棵二层二插树。
询问左节点的 child direction.
```

```
获得结果，正在比较该返回值。
测试通过!
====================

正在进行第 6 组测试:
构造一棵二层二叉树。
询问右节点的 child direction.
获得结果，正在比较该返回值。
测试通过!
====================

正在进行第 7 组测试:
构造一棵三层二叉树。
询问根的左节点的 child direction.
获得结果，正在比较该返回值。
测试通过!
====================

正在进行第 8 组测试:
构造一棵三层二叉树。
询问左节点的右子节点的 child direction.
获得结果，正在比较该返回值。
测试通过!
====================

本次测试到此结束。
====================

Program ended with exit code: 0
```

# Part 4 - Difficulties & Solutions

**Difficulties:**

- Exception is hard to contain all and without a throw, it may affect the operation.
- The pointer operation need to be careful.

**Solutions:**

- Solve the exception problem using this:

```
    if (node != nullptr && child_direction != nullptr && node->father != nullptr)
```

- Use pointer to make assignment:

```
            *child_direction = CHILD_DIRECTION_LEFT;
```

# Task 3 - Insert into Binary Search Tree

We define function

```
exception insert_into_BST(BST *bst, uint64_t data, tree_node **inserted_node)
```

as inserting specific `{data}` into `{bst}`; and store the inserted tree node to the address of `{inserted_node}`. Because we use the return value as exception handling, we must use another pointer in parameter to specify where to store the inserted tree node. We use `{inserted_node}` here, a pointer to `{tree_node*}` to specify the address, in which the function should store the inserted tree node. If you insert a *new* node, just store the address of the new node to this address; or if you update an existing node, please store the address of updated node to this address.

Exceptions you **SHOULD** handle:

- `NULL_POINTER_EXCEPTION`: Handle this exception when {bst} or {inserted_node} is nullptr.
- `NULL_COMP_FUNCTION_EXCEPTION`: Handle this exception when {bst->comp} is nullptr.
- All the exceptions should be handled in Part One if you use {add_node()} in this function. (**NOT RECOMMENDED**)

What you **SHOULD** do:

- Implement `exception insert_into_BST(BST *bst, uint64_t data, tree_node **inserted_node)` in `assign2.cpp`
  - If {bst->root} is nullptr, which means it is an empty binary search tree, set {bst->root} to the new inserted node; else insert the new node to suitable position
  - Update {node_count} and {tree_count} from ***inserted node*** to ***the root of the tree***.
  - If there has been specific {node} in {bst}, in which {bst->data} is the same as {data} compared by {bst->comp()}, increase {node->node_count} instead of inserting a new node.

# Part 1 - Analysis

> In this part, we need to insert a specific data into a binary tree, then store the node in the double pointer we passed in.
>
> Because the return value of this function is the exception that contains all the exceptions, I initialize the e=0 for the exception. then use a series of the if judgment to judge whether it is a null pointer or null comp function. And finally return the exception.
>
> After the exception handle is done, use a if judgement to make sure the operation is safe. then i design a recursion to solve this problem in each step it will create a new bst tree then go deeper until the data is perfectly inserted.

## Part 2 - Code

```
assign2_exception::exception insert_into_BST(BST *bst, uint64_t data, tree_node
**inserted_node)
{
    assign2_exception::exception e = 0;
    if (bst == nullptr || inserted_node == nullptr) {
        e |= NULL_POINTER_EXCEPTION;
    }
    if (bst != nullptr && bst->comp == nullptr) {
        e |= NULL_COMP_FUNCTION_EXCEPTION;
    }
    if (bst != nullptr && inserted_node != nullptr && bst->comp != nullptr) {
        tree_node *node = new tree_node;
        node->data = data;
        node->father = node->l_child = node->r_child = nullptr;
        node->node_count = 1;
        node->tree_count = 1;
        if (bst->root == nullptr)
        {
            bst->root = node;
            *inserted_node = node;
            return e;
        }
        delete node;
        if (bst->comp( data , bst->root->data) < 0)
        {
            BST *new_bst = new BST;
            new_bst->comp = bst->comp;
            new_bst->root = bst->root->l_child;
            e |= insert_into_BST(new_bst, data, inserted_node);
//            add_node(bst->root, new_bst->root, CHILD_DIRECTION_LEFT);
//            ADD_NODE_LEFT(bst->root, new_bst->root);
            {
                bst->root->l_child = new_bst->root;
                new_bst->root->father = bst->root;
                bst->root->tree_count ++;
            }

            delete new_bst;
        }
        else if (bst->comp(data , bst->root->data) > 0)
        {
            BST *new_bst = new BST;
            new_bst->comp = bst->comp;
            new_bst->root = bst->root->r_child;
            e |= insert_into_BST(new_bst, data, inserted_node);
//            add_node(bst->root, new_bst->root, CHILD_DIRECTION_RIGHT);
```

```
//            ADD_NODE_RIGHT(bst->root, new_bst->root);
            {
                bst->root->r_child = new_bst->root;
                new_bst->root->father = bst->root;
                bst->root->tree_count ++;
            }
            delete new_bst;
        }
        else if (bst->comp(data , bst->root->data) == 0)
        {
            bst->root->node_count++;
            bst->root->tree_count++;
            *inserted_node = bst->root;
        }
    }

    return e;
}
```

## Part 3 - Result & Verification

Verification is used and thanks to CutieDeng, the verification test is in https://github.com/CutieDeng/cpp_ass2_test

```
正在进行第 0 组测试:
进入 Part III 测试。
向 nullptr(bst) 中插入 data.
测试通过!
====================

正在进行第 1 组测试:
向未初始化的 bst 中插入数据但没有给出答案空间。
测试通过!
====================

正在进行第 2 组测试:
向未初始化的 bst 中插入数据。
测试通过!
====================

正在进行第 3 组测试:
向标准的空 bst 中插入一条数据。
测试通过!
====================

正在进行第 4 组测试:
向标准的空 bst 中插入两条相同数据。
正在检查该 bst 树的大小信息。
```

```
测试通过!
====================

正在进行第 5 组测试:
向标准的 bst 中插入不同数据。
正在检查该 bst 树的大小信息。
测试通过!
====================

正在进行第 6 组测试:
向逆序的 bst 插入一些数据。
正在检查该 bst 的大小信息。
测试通过!
====================

正在进行第 7 组测试:
向全等 bst 中插入一些数据。
正在检查该 bst 的大小信息。
测试通过!
====================

正在进行第 8 组测试:
向个位 bst 中插入若干数据。
遍历树结构并进行检查。
测试通过!
====================

正在进行第 9 组测试:
向虚数 bst 中插入若干数据。
遍历树结构并进行检查。
测试通过!
====================

正在进行第 10 组测试:
向 bst 中插入若干不同数据，后只插入重复数据。
遍历树结构进行检查。
测试通过!
====================

本次测试到此结束。
====================

Program ended with exit code: 0
```

## Part 4 - Difficulties & Solutions

**Difficulties:**

- Exception is hard to contain all and without a throw, it may affect the operation.
- The pointer operation needs to be careful.
- The recursion has to be used
- The previous add node function dose not fit well and needs refactoring.

**Solutions:**

- Exception handled using an if to prevent error:

```
if (bst != nullptr && inserted_node != nullptr && bst->comp != nullptr)
```

- The pointer of inserted node is used to store the node which is inserted in the tree:

```
tree_node *node = new tree_node;
 node->data = data;
 node->father = node->l_child = node->r_child = nullptr;
 node->node_count = 1;
 node->tree_count = 1;
 if (bst->root == nullptr)
 {
     bst->root = node;
     *inserted_node = node;
     return e;
 }
 delete node;
```

- A recursion is used to iterate the elements in the tree:

```
BST *new_bst = new BST;
new_bst->comp = bst->comp;
new_bst->root = bst->root->r_child;
e |= insert_into_BST(new_bst, data, inserted_node);
```

- The add node function is deprecated and a new one is added( for better coping with the counts)

```
{
    bst->root->r_child = new_bst->root;
    new_bst->root->father = bst->root;
    bst->root->tree_count ++;
}
delete new_bst;
```

# Task 4 - Find Element in Binary Search Tree

> We define function
>
> ```
> exception find_in_BST(BST *bst, uint64_t data, tree_node **target_node)
> ```
>
> as finding specific data in the BST, and storing the target tree node into `{target_node}`.
>
> Exceptions you **SHOULD** handle:
>
> - `NULL_POINTER_EXCEPTION`: Handle this exception when {bst} or {target_node} is nullptr.
> - `NULL_COMP_FUNCTION_EXCEPTION`: Handle this exception when {bst->comp} is nullptr.
>
> What you **SHOULD** do:
>
> - Implement `exception find_in_BST(BST *bst, uint64_t data, tree_node **target_node)` in `assign2.cpp`
>   - If there is no such {node} in {bst} satisfying {node->data} is the same as {data} compared by {bst->comp()}, set *{target_node} to nullptr; else set* {target_node} to {node}.

# Part 1 - Analysis

> Similar to the task 3, in this function we will find a specific data in the bst or return a exception.
>
> Because the return value of this function is the exception that contains all the exceptions, I initialize the e=0 for the exception. then use a series of the if judgment to judge whether it is a null pointer or null comp function. And finally return the exception.
>
> After the exception handle is done, use a if judgement to make sure the operation is safe. then i design a recursion to solve this problem in each step it will create a new bst tree then go deeper until the data is found and store the node into the pointer or throw a exception.

# Part 2 -Code

```
assign2_exception::exception find_in_BST(BST *bst, uint64_t data, tree_node
**target_node)
{
    assign2_exception::exception e = 0;
    if (bst == nullptr || target_node == nullptr)
    {
        e |= NULL_POINTER_EXCEPTION;
    }
    if (bst != nullptr && bst->comp == nullptr)
    {
        e |= NULL_COMP_FUNCTION_EXCEPTION;
    }
    if (bst != nullptr && target_node != nullptr && bst->comp != nullptr)
```

```
    {
        if (bst->root == nullptr)
        {
            *target_node = nullptr;
            return e;
        }
        else if (bst->comp(bst->root->data, data) == 0)
        {
            (*target_node) = bst->root;
            return e;
        }
        else if (bst->comp(data, bst->root->data) < 0)
        {
            BST *new_bst = new BST;
            new_bst->comp = bst->comp;
            new_bst->root = bst->root->l_child;
            find_in_BST(new_bst, data, target_node);
            delete new_bst;
        }
        else if (bst->comp(data, bst->root->data) > 0)
        {
            BST *new_bst = new BST;
            new_bst->comp = bst->comp;
            new_bst->root = bst->root->r_child;
            find_in_BST(new_bst, data, target_node);
            delete new_bst;
        }
    }
    return e;
}
```

## Part 3 - Result & Verification

Verification is used and thanks to CutieDeng, the verification test is in https://github.com/CutieDeng/cpp_ass2_test

```
正在进行第 0 组测试:
下面进入 Part IV 测试。
构建一棵空 bst, 并执行 find in bst.
测试通过!
====================

正在进行第 1 组测试:
执行 find in bst (nullptr).
测试通过!
====================

正在进行第 2 组测试:
```

```
构造一棵 bst, 但在查询时没有提供答案回传地址。
测试通过!
====================

正在进行第 3 组测试:
构造一棵标准的 bst, 并插入若干数据。
搜索不存在于 bst 中的数据。
测试通过!
====================

正在进行第 4 组测试:
构建一棵 bst, 并向其中插入数据。
搜索存在于 bst 中的数据。
测试通过!
====================

正在进行第 5 组测试:
构建一棵个位 bst, 并向其中插入数据。
询问其中的部分元素个数。
正在比对该元素在 bst 中的个数。
测试通过!
====================

正在进行第 6 组测试:
构建一个全等 bst, 并向其中插入数据。
进行元素个数的询问.
正在比对该元素在 bst 中的个数。
测试通过!
====================

正在进行第 7 组测试:
构建一棵空 bst, 有comp方法, 并执行 find in bst.
测试通过!
====================

本次测试到此结束。
====================

Program ended with exit code: 0
```

# Part 4 - Difficulties & Solutions

---

**Difficulties:**

- Exception is hard to contain all and without a throw, it may affect the operation.
- The pointer operation needs to be careful.
- The recursion has to be used.

**Solutions:**

- Exception handled using an if to prevent error:

```
if (bst != nullptr && target_node != nullptr && bst->comp != nullptr)
```

- The pointer of target node is used to store the node which is inserted in the tree:

```
        if (bst->root == nullptr)
        {
            *target_node = nullptr;
            return e;
        }
        else if (bst->comp(bst->root->data, data) == 0)
        {
            (*target_node) = bst->root;
            return e;
        }
```

- A recursion is used to iterate the elements in the tree:

```
            BST *new_bst = new BST;
            new_bst->comp = bst->comp;
            new_bst->root = bst->root->r_child;
            find_in_BST(new_bst, data, target_node);
            delete new_bst;
```

# Task 5 - Splay

> The target for splay function is to adjust the structure of the BST without changing the in-order traversal, putting the target node to the root. We define function
>
> ```
> exception splay(BST *bst, tree_node *node)
> ```
>
> as *splay* tree node `node` to the root of binary search tree if `node` is in `bst`.
>
> Tips:
>
> - The basic two operations to the node: Zig, Zag
> - The three conbination operations to the node according to the structure: Zig-Zig/Zag-Zag, Zig-Zag/Zag-Zig, Zig/Zag
> - Reuse the code
> - Be aware of **NULL** pointers
>
> Exceptions you **SHOULD** handle:
>
> - `NULL_POINTER_EXCEPTION`: Handle this exception when {bst} or {node} is nullptr.
> - `NULL_COMP_FUNCTION_EXCEPTION`: Handle this exception when {bst->comp} is nullptr.

- `SPLAY_NODE_NOT_IN_TREE_EXCEPTION`: Handle this exception when the furthest ancestors of {node} is not {bst->root}. The furthest ancestors-{furthest_anc} of {node} is defined as {node->father->father->...->father}, and {furthest_anc->father} is nullptr.
- All the exceptions should be handled in Part One if you use {add_node()} in this function. (**NOT RECOMMENDED**)

What you **SHOULD** do:

- Implement `exception splay(BST *bst, tree_node *node)` in `assign2.cpp`
  - ***Splay*** {node} in argument to the root of {bst} if the furthest ancestors of {node} is {bst->root}; else record SPLAY_NODE_NOT_IN_TREE_EXCEPTION.

# Part 1 - Analysis

---

In this task, we needs to do the splay operation in the bst, which to briefly speaking is that make the specific node rotates to the top root.

Before the function begin, i have defined three functions which are two rotate function that either left or right rotate the node and return the rotated node in the root position. Another function is to update the node count and the tree count after rotated.

Because the return value of this function is the exception that contains all the exceptions, I initialize the e=0 for the exception. then use a series of the if judgment to judge whether it is a null pointer or null comp function. And finally return the exception.

After the exception handle is done, use a if judgement to make sure the operation is safe. Then the operation starts:

( we know that the splay tree has the following cases: )

**1) Node is root** We simply return the root, don't do anything else as the accessed node is already root.
**2) Zig: *Node is child of root*** (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation).
T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

**3) *Node has both parent and grandparent***. There can be following subcases.
.....**3.a) Zig-Zig and Zag-Zag** Node is left child of parent and parent is also left child of grand parent (Two right rotations) OR node is right child of its parent and parent is also right child of grand parent (Two Left Rotations).

.....**3.b) Zig-Zag and Zag-Zig** Node is right child of parent and parent is right left of grand parent (Left Rotation followed by right rotation) OR node is left child of its parent and parent is right child of grand parent (Right Rotation followed by left rotation).

Here comes our solution:

- first judge the node lies in the left side or the right side using a if judgement
- then using a recursion to bring the root to either Zig-Zig (Left Left) or. Zig-Zag (Left Right) :(we use the left side as an example and vice versa)
- then we split the rotation into two parts

    - Do first rotation for root or root->l_child
    - second rotation is done after else
- if the node not found then we throw an exception

## Part 2 - Code

```
void update(tree_node *node);
tree_node *left_rotation(tree_node *x);
tree_node *right_rotation(tree_node *x);

assign2_exception::exception splay(BST *bst, tree_node *node)
{
    assign2_exception::exception e = 0;
    if (bst == nullptr || node == nullptr) {
        e |= NULL_POINTER_EXCEPTION;
    }
    if (bst != nullptr && bst->comp == nullptr) {
        e |= NULL_COMP_FUNCTION_EXCEPTION;
    }
    if (bst != nullptr && node != nullptr && bst->comp != nullptr) {
        if (bst->root == nullptr) {
            e |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
        }
        if (bst->root != nullptr) {
            if (bst->root == node) {
                return e;
            }
            if (bst->comp(node->data , bst->root->data)<0) {
                if (bst->root->l_child == nullptr) {
                    e |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
                    return e;
                }
                if (bst->comp(node->data , bst->root->l_child->data)<0) {
                    BST* new_bst = new BST;
                    new_bst->comp = bst->comp;
                    new_bst->root = bst->root->l_child->l_child;
                    e |= splay(new_bst, node);
                    bst->root->l_child->l_child = new_bst->root;
                    bst->root = right_rotation(bst->root);
                    update(bst->root->r_child);
                    update(bst->root);
                    delete new_bst;
                }
                else if (bst->comp(node->data , bst->root->l_child->data)>0){
                    BST* new_bst = new BST;
                    new_bst->comp = bst->comp;
                    new_bst->root = bst->root->l_child->r_child;
                    e |= splay(new_bst, node);
```

```
                if (bst->root->l_child->r_child != nullptr) {
                    bst->root->l_child->r_child = new_bst->root;
                    bst->root->l_child = left_rotation(bst->root->l_child);
                    update(bst->root->l_child->l_child);
                    update(bst->root->l_child);
                }
                delete new_bst;
            }
            if (bst->root->l_child == nullptr || bst->root->l_child != node) {
                e |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
            }
            else{
                bst->root =  right_rotation(bst->root);
            }
            update(bst->root->r_child);
            update(bst->root);
            return e;
        }
        else{
            if (bst->root->r_child == nullptr) {
                e |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
                return e;
            }
            if (bst->comp(node->data , bst->root->r_child->data)<0) {
                BST* new_bst = new BST;
                new_bst->comp = bst->comp;
                new_bst->root = bst->root->r_child->l_child;
                e |= splay(new_bst, node);
                if (bst->root->r_child->l_child != nullptr) {
                    bst->root->r_child->l_child = new_bst->root;
                    bst->root->r_child = right_rotation(bst->root->r_child);
                    update(bst->root->r_child->r_child);
                    update(bst->root->r_child);
                }
                delete new_bst;
            }
            else if (bst->comp(node->data , bst->root->r_child->data)>0){
                BST* new_bst = new BST;
                new_bst->comp = bst->comp;
                new_bst->root = bst->root->r_child->r_child;
                e |= splay(new_bst, node);
                bst->root->r_child->r_child = new_bst->root;
                bst->root = left_rotation(bst->root);
                update(bst->root->l_child);
                update(bst->root);
                delete new_bst;
            }
            if (bst->root->r_child == nullptr || bst->root->r_child != node) {
                e |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
```

```
                }
                else{
                    bst->root =  left_rotation(bst->root);
                }
                update(bst->root->l_child);
                update(bst->root);
                return e;
            }
        }

    }

    return e;
}

tree_node* right_rotation(tree_node* x){
    tree_node* y = x->l_child;
    x->l_child = y->r_child;
    if (x->l_child != nullptr) {
        x->l_child->father = x;
    }
    y->r_child = x;
    y->father = nullptr;
    y->r_child->father = y;
    return y;
}

tree_node* left_rotation(tree_node* x){
    tree_node* y = x->r_child;
    x->r_child = y->l_child;
    if (x->r_child != nullptr) {
        x->r_child->father = x;
    }
    y->l_child = x;
    y->father = nullptr;
    y->l_child->father = y;
    return y;
}

void update(tree_node* node){
    if (node != nullptr) {
        node->tree_count = ((node->l_child != nullptr)?node->l_child->tree_count:0) +
((node->r_child != nullptr)?node->r_child->tree_count:0) + node->node_count;
    }
}
```

## Part 3 - Result & Verification

Verification is used and thanks to CutieDeng, the verification test is in https://github.com/CutieDeng/cpp_ass2_test

```
正在进行第 0 组测试:
正在进入 Part V 测试。
调用 splay nullptr nullptr 方法。
测试通过!
====================

正在进行第 1 组测试:
未完全初始化一个 bst, 并对其调用 splay nullptr.
测试通过!
====================

正在进行第 2 组测试:
初始化一个 bst, 并调用 splay, 并传递一个空指针描述待旋转的 tree node.
测试通过!
====================

正在进行第 3 组测试:
初始化一个 bst, 并初始化一个节点 node, 调用 splay bst node.
测试通过!
====================

正在进行第 4 组测试:
初始化两个 bst, 并向其中传入若干相同数据。
查询一棵树上的结果, 对另一棵树调用 splay.
测试通过!
====================

正在进行第 5 组测试:
初始化一个个位识别的 bst, 并向其中插入少量递减数据。
执行 splay 操作。
测试通过!
====================

正在进行第 6 组测试:
初始化一个 bst, 插入数据 15, 12, 7, 5, 2, 4。
执行 splay 操作, 将 2 转至 root 点。
测试通过!
====================

正在进行第 7 组测试:
初始化一个反向 bst, 并向其中插入重复数据。
数据: 16, 22, 23, 47, 27, 22, 16, 47, 36, 36, 32, 31.
执行 splay 操作, 搜索值为 31 的点。
```

```
测试通过!
====================

本次测试到此结束。
====================

Program ended with exit code: 0
```

## Part 4 - Difficulties & Solutions

**Difficulties:**

- Exception is hard to contain all and without a throw, it may affect the operation.
- The pointer operation needs to be careful.
- The recursion has to be used.
- The rotation function also has to handle with the father relation makes it a bit cumbersome
- the Zig-Zag cases needs to be clearly solved
- Node counts and tree counts needs to be updated

**Solutions:**

- Exception handled using an if to prevent error:

```
    assign2_exception::exception e = 0;
    if (bst == nullptr || node == nullptr) {
        e |= NULL_POINTER_EXCEPTION;
    }
    if (bst != nullptr && bst->comp == nullptr) {
        e |= NULL_COMP_FUNCTION_EXCEPTION;
    }
    if (bst != nullptr && node != nullptr && bst->comp != nullptr) {
        if (bst->root == nullptr) {
            e |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
        }
```

- The pointer of each child and father is solved manually with lots of tests
- Recursions are used to iterate the elements in the tree:

```
                    BST* new_bst = new BST;
                    new_bst->comp = bst->comp;
                    new_bst->root = bst->root->r_child->r_child;
                    e |= splay(new_bst, node);
                    bst->root->r_child->r_child = new_bst->root;
                    bst->root = left_rotation(bst->root);
                    update(bst->root->l_child);
                    update(bst->root);
                    delete new_bst;
```

- The rotation function also has to handle with the father relation

```
tree_node* right_rotation(tree_node* x){
    tree_node* y = x->l_child;
    x->l_child = y->r_child;
    if (x->l_child != nullptr) {
        x->l_child->father = x;
    }
    y->r_child = x;
    y->father = nullptr;
    y->r_child->father = y;
    return y;
}

tree_node* left_rotation(tree_node* x){
    tree_node* y = x->r_child;
    x->r_child = y->l_child;
    if (x->r_child != nullptr) {
        x->r_child->father = x;
    }
    y->l_child = x;
    y->father = nullptr;
    y->l_child->father = y;
    return y;
}
```

- the Zig-Zag cases have clearly solved
- After rotation,we needs to update the root->child count first then the root count

```
                    bst->root = right_rotation(bst->root);
                    update(bst->root->r_child);
                    update(bst->root);
                    delete new_bst;
```