CS205 C/C++ Program Design Assignment 2

Name: 陈逸飞 Chen Yifei

Student ID: 12010502

Develop environment: Apple clang version 12.0.5 (clang-1205.0.22.11)

Part 1-Analysis

本次作业共有5个部分

> Part 1. Add Node (20 pts)
>
> Part 2. Judge Child Direction (10 pts)
>
> Part 3. Insert into Binary Search Tree (20 pts)
>
> Part 4. Find Element in Binary Search Tree (20 pts)
>
> Part 5. Splay (30 pts)

本次作业涉及算法结构中树的相关概念，初看作业文档时，感觉非常头大，不知道从何下手。但是通过自学基础知识并理解本次作业的框架，我发现本次的作业是对使用指针的一个深度总结。出题者顾及了我们这样的starter对于构建一个项目没有完整的认识，所以对于使用的数据和异常处理都使用了让我们可以最方便使用的方式。

1.1 Structure

本次作业涉及到的两个structure，一个是tree_node，树的每一个节点（最基本单元）；一个是Binary Search Tree的结构体。

1.1.1 tree_node

```
struct tree_node {
    tree_node *father;
    tree_node *l_child, *r_child;

    uint32_t node_count;
    uint32_t tree_count;

    uint64_t data;
};
```

tree_node为一个节点包括的信息:

1. 关系节点：该节点的父节点（father）和左右子节点 l_child,r_child（结构体指针）

2. 数据：

   node_count: 该节点存储数据的重复次数

   tree_count: 子树（包括根节点）的大小（节点个数）

   data：节点存储的数据

1.1.2 BST

```
struct BST {
    tree_node *root;
    int (*comp)(uint64_t, uint64_t);
};
```

Binary Search Tree

root: 指针结构体tree_node, 作为树的根节点，根据定义，root是没有的father节点，只有左右子树。

comp方法：比较两个uint64_t的值返回大小值。

> @returns (0) if two data fields equal
>
> @returns (int > 0) if the first data field is greater than the second one
>
> @returns (int < 0) if the first data field is smaller than the second one

1.2 Exception （error handling）

本次我们需要实现的方法返回值均为类型为uint32_t的`assign2_exception::exception`

1.2.1 type of exception

NULL_POINTER_EXCEPTION (0x1 << 0) : 访问成员为空指针。

DUPLICATED_LEFT_CHILD_EXCEPTION(0x1 << 1)： 在向父节点的左子节点添加新的子节点时，但父节点已经有了左子节点。

DUPLICATED_RIGHT_CHILD_EXCEPTION(0x1 << 2)： 在向父节点的右子节点添加新的子节点时，但父节点已经有了右子节点

DUPLICATED_FATHER_EXCEPTION(0x1 << 3)： 在向新的父节点添加一个子节点时，而该子节点已经有一个父节点

INVALID_CHILD_DIRECTION_EXCEPTION(0x1 << 4)： 用户使用了无效的**child_direction**字段值（非**CHILD_DIRECTION_LEFT** 0和**CHILD_DIRECTION_RIGHT** 1）

ROOTS_FATHER_EXCEPTION(0x1 << 5)： 用户要访问根节点的父节点

NULL_COMP_FUNCTION_EXCEPTION(0x1 << 6)： the user wants to insert/find/splay a nodes in BST, while there is no `comp` function defined for the BST

SPLAY_NODE_NOT_IN_TREE_EXCEPTION(0x1 << 7)： 在将节点扩展到BST的根节点时，而该节点在该BST中不存在

1.2.2 handle exception

记录异常情况：`exception |= NULL_POINTER_EXCEPTION`

检查异常情况：`if (exception & NULL_POINTER_EXCEPTION)`

实现方法

```
/*Judge whether this is a NULL pointer*/
    if (father == nullptr || child == nullptr) // This is a NULL pointer
    {
        exception |= NULL_POINTER_EXCEPTION;
    }
/*Check whether the left_child is NULL or not, if there is already have value,
then return exception*/
    if (father != nullptr && child_direction == CHILD_DIRECTION_LEFT)
    {
        if (father->l_child != nullptr)
            exception |= DUPLICATED_LEFT_CHILD_EXCEPTION;
    }
/*Check whether the right_child is NULL or not, if there is already have
value, then return exception*/
    else if (father != nullptr && child_direction == CHILD_DIRECTION_RIGHT)
    {
        if (father->r_child != nullptr)
            exception |= DUPLICATED_RIGHT_CHILD_EXCEPTION;
    }
/*Check whether the child already have a father*/
    if (child != nullptr && child->father != nullptr)
    {
        exception |= DUPLICATED_FATHER_EXCEPTION;
    }
/*check the child_direction is valid or not*/
    if ((child_direction != CHILD_DIRECTION_LEFT) && (child_direction !=
CHILD_DIRECTION_RIGHT))
    {
        exception |= INVALID_CHILD_DIRECTION_EXCEPTION;
    }
/*Check the existing of comp in bst*/
    if (bst != nullptr && (bst->comp) == nullptr)
    {
        exception |= NULL_COMP_FUNCTION_EXCEPTION;
    }
/*Check the existing of comp in bst*/
    if (bst != nullptr && node != nullptr&&!(judge_node_bst(bst, node)))
    {
        exception |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
    }

/*使用judge_node_bst来判断node是否在二叉树中*/
int judge_node_bst(BST *bst, tree_node *node)
{
```

```
    tree_node *Move = node;
    while (Move->father != nullptr)
    {
        Move = Move->father;
    }
    if (Move == bst->root)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

在测试这个地方的时候，对于判断异常发生的逻辑一开始出现了问题。在判断`DUPLICATED_RIGHT_CHILD_EXCEPTION`和`DUPLICATED_LEFT_CHILD_EXCEPTION`时需要在`father`为不为空和`child_direction`对应正确的条件下在进行判断：`father->l_child`不为空，则是**DUPLICATED**情况。这一点的逻辑关系是跟方同学讨论时，我才发现的情况，没有排除这样简单的情况感觉很尴尬。

还需注意的是判断`SPLAY_NODE_NOT_IN_TREE_EXCEPTION`时使用自己写的**judge_node_bst**，需要向上遍历节点，查看是否存在。这里需要新建一个**tree_node**指针来表示遍历的当前值**move**。最后通过返回1或0来判断node是否在二叉树中。

在每个方法的逻辑上，我选择先判断涉及到的所有`exception`情况，然后再整体判断是否发生`exception`，若发生，返回`exception`终止方法进程。我在`judge_child_direction`的判断exception.

```
    uint32_t exception = 0;
    if (node == nullptr || child_direction == nullptr)
    {
        exception |= NULL_POINTER_EXCEPTION;
    }
    if (node != nullptr && node->father == nullptr)
    {
        exception |= ROOTS_FATHER_EXCEPTION;
    }

    if (exception != 0)
    {
        return exception;
    }
```

1.3 Functions

1.3.1 Add_Node (20 pts)

> add node @p child to @p father 's corresponding child according to the @p child_direction
>
> if @p child_direction equals @link{CHILD_DIRECTION_LEFT}, add @p child to @p father 's left child
>
> if @p child_direction equals @link{CHILD_DIRECTION_RIGHT}, add @p child to @p father 's right child

这个方法的功能就是依据**child_direction**添加**child**到**father**的左/右侧，如`child_direction == CHILD_DIRECTION_LEFT`时，执行`father->l_child = child`

这里要注意的是在add_node之后需要更新父节点的tree_count，来保证对于整个树的结构记录是正确的。我在这里使用自己写的方法new_all_tree_count(在后面会有对这个方法的解释)。

1.3.2 Judge Child Direction (10 pts)

> judge whether the node @p node is the left or right child of its father, and store the result to @p child_direction

判断子节点的方向，是要比较该节点的地址和`father`的`l_child`和`r_child`，而不能是比较数据。

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

在判断出结果后，将结果存储在`*child_direction`(指针的指针)中，作为添加节点的依据。

1.3.3 Insert into Binary Search Tree (20 pts)

> insert specific @p data into @p bst, according to the @link{BST#comp(uint64_t, uint64_t)} function

这个方法把数据插入到Binary Search Tree，若已经存在节点，就直接把data存储到该节点；若节点不存在，就要通过`BST->comp()`来比较节点的data和要插入的data。根据**Binary Search Tree**的性质：

1）它的左子树不空，则左子树上所有结点的值均小于它的根结点的值。

2）若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

所以循环内的判断逻辑是：

1. 插入的data比节点的data大，则向节点的右子树走；
2. 插入的data比节点的data小，则向节点的左子树走；
3. 插入的data相等，则更新所有涉及到的**node_count**

因为这个方法涉及增加新的节点，所以需要更新**tree_node**中的**tree_count** 数据。

为了使用方便，我创造了**new_all_tree_count**方法来更新**tree_count**.通过递归的方法来遍历所有的节点。

```
void new_all_tree_count(tree_node *node)
{
    if (node == nullptr)
    {
        return;
    }
    else
    {
        new_all_tree_count(node->l_child);
        new_all_tree_count(node->r_child);
        int i = 0, j = 0;
        if (node->l_child != nullptr)
            i = (node->l_child)->tree_count;
        if (node->r_child != nullptr)
            j = (node->r_child)->tree_count;
        node->tree_count = node->node_count + i + j;
    }
}
```

在这一环节我感受到了本次结构体这样设置的便利性，根据我自己对二叉搜索树的学习，对于遍历二叉树的方法最基本的有前序、中序、后序三种方式。根据树结构性的不同选择不同的遍历方式。但是在本次作业中，我们就可以直接经过**tree_node**中的关系指针来进行遍历。

1.3.4 Find Element in Binary Search Tree (20 pts)

> find specific data in the BST, and storing the target tree node into ans

这个方法是在Binary Search Tree中找到指定的数据。第四部分和第三部分原理相似，都是通过遍历的方式来找到data，并存储在`**target_node`中。

```
/*No exceptions about bst and target_node*/
    if (bst->root == nullptr)
    {
        *target_node = 0x0;
    }
    else
    {
        tree_node *Move = bst->root;
        while (true)
        {
            int comp = bst->comp(data, Move->data);
            if (comp == 0)
            {
                *target_node = Move;
                break;
            }
            else if (comp > 0)
            {
                if (Move->r_child == nullptr)
```

```
                {
                    *target_node = 0x0;
                    break;
                }
                else
                {
                    Move = Move->r_child;
                }
            }
            else
            {
                if (Move->l_child == nullptr)
                {
                    *target_node = 0x0;
                    break;
                }
                else
                {
                    Move = Move->l_child;
                }
            }
        }
    }
```

这里我注意到对于**target_node**的存储，需要随时注意更新。

1.3.5 Splay (30 pts)

> splay tree node @p node to the root of binary search tree if @p node is in @p bst

这个方法是关于平衡二叉树（AVL）进行左右旋的调整。左旋右旋的目的是减小树的高度，用最合适的方式来存储数据。在经过学习之后，我总结左旋右旋的调整方式和AVL基本原则性质：

- 左右子树深度之差的绝对值不超过1。`最高层-最底层<=1`，所以平衡二叉树只是相对平衡。
- 左右子树仍然为平衡二叉树。

左旋右旋就是通过变化根节点的方式来使树平衡

单独编写的左旋右旋方法：Zig，Zag

```
/*单右旋*/
void Zig(BST *bst, tree_node *node)
{
    tree_node *father = node->father;
    tree_node *grandfather = father->father;
    father->l_child = node->r_child;
    if (node->r_child != nullptr)
    {
        node->r_child->father = father;
    }
    node->r_child = father;
```

```
    node->father = grandfather;
    father->father = node;
    if (grandfather != nullptr)
    {
        if (grandfather->l_child == father)
        {
            grandfather->l_child = node;
        }
        else if (grandfather->r_child == father)
        {
            grandfather->r_child = node;
        }
    }
    else
    {
        bst->root = node;
    }
}

/*单左旋*/
void Zag(BST *bst, tree_node *node)
{
    tree_node *father = node->father;
    tree_node *grandfather = father->father;
    father->r_child = node->l_child;
    if (node->l_child != nullptr)
    {
        node->l_child->father = father;
    }
    node->l_child = father;
    node->father = grandfather;
    father->father = node;
    if (grandfather != nullptr)
    {
        if (grandfather->l_child == father)
        {
            grandfather->l_child = node;
        }
        else if (grandfather->r_child == father)
        {
            grandfather->r_child = node;
        }
    }
    else
        bst->root = node;
}
```

左右旋首先定义父节点和祖父节点，然后设置其父节点的孩子为一侧，；若有那一边的孩子则更新这一侧节点的父节点信息。然后依次更新原父节点和祖父节点。

这一部分的zig，zag思路是学习b站上的平衡树课程编写

> https://www.bilibili.com/video/BV15a4y1a7B5?spm_id_from=333.880.my_history.page.click

最基本的实现方式是只通过一个单旋方式。

```
/*Part 5 splay binary search tree*/
assign2_exception::exception splay(BST *bst, tree_node *node)
{
    uint32_t exception = 0;
    if (bst == nullptr || node == nullptr)
    {
        exception |= NULL_POINTER_EXCEPTION;
    }
    if (bst != nullptr && bst->comp == nullptr)
    {
        exception |= NULL_COMP_FUNCTION_EXCEPTION;
    }
    if (bst != nullptr && node != nullptr&&!(judge_node_bst(bst, node)))
    {
        exception |= SPLAY_NODE_NOT_IN_TREE_EXCEPTION;
    }

    if (exception & NULL_POINTER_EXCEPTION ||
        exception & SPLAY_NODE_NOT_IN_TREE_EXCEPTION)
    {
        return exception;
    }

     while (bst->root->l_child != node && bst->root->r_child != node && bst-
>root != node)
    {
        judge_Zig_Zag(bst,node);
    }

    while (node->father != nullptr)
    {
        /*当前节点为根的左孩子，只需一次右旋*/
        if (node->father->l_child == node)
        {
            Zig(bst, node);
        }
        else if (node->father->r_child == node) //当前节点为根的右孩子，只需要进行一
次左旋
        {
            Zag(bst, node);
        }
    }
    new_all_tree_count(bst->root);
    return exception;
```

```
}
```

`while (node->father != nullptr)`内的循环这里实现的是单旋的方式，虽然能够实现目标，但是不能保证**splay**的平衡性，在算法上也不够简洁，会提高时间复杂度。

采用双旋则需要引入factor来判断旋转的次数

```
void judge_Zig_Zag(BST *root, tree_node *node)
{
    tree_node *parent = node->father;
    tree_node *grandparent = parent->father;
    int factor_i, factor_j;
    parent = node->father;
    grandparent = parent->father;

    factor_i = (grandparent->l_child == parent) ? -1 : 1;
    factor_j = (grandparent->r_child == parent) ? -1 : 1;
    if ((factor_i == -1 && factor_j == -1))
    {
        /* Zig-Zig 双右旋 */
        Zig(root, node);
        Zig(root, node);
    }
    else if ((factor_i == -1 && factor_j == 1))
    {
        /* Zig-Zag 右左旋*/
        Zag(root, node);
        Zig(root, node);
    }
    else if ((factor_i == 1 && factor_j == -1))
    {
        /* Zag-Zig 左右旋*/
        Zig(root, node);
        Zag(root, node);
    }
    else
    {
        /* Zig-Zig 左左旋*/
        Zag(root, node);
        Zag(root, node);
    }
    new_all_tree_count(root->root);
}
```

分为RR，RL，LR，LL型

判别类型的方式是把查找点往上递推，查看父节点与祖父节点的关系来判别。

Part Two-Testing

本次作业没有给出官方的测试渠道，我自己尝试写了测试方法，虽然能够对树进行一定的操作，但是终归是没有办法考虑到所有的情况。同时还有一些自己没有学习到和不会利用的语言地方，所以自己的测试没有办法检查完全程序问题。我和几位非计算机系的同学对于这个问题都非常头疼。但是在危难关头，计算机系的大佬们贡献出了自己的测试代码，供大家共同测试。我使用了曾同学和邓同学的测试代码作为自测，也请我的好朋友cqz用他的本地测试类测试我的代码。测试出的类型错误主要有4种：

1. 段错误
2. exception类型判断错误
3. 数据传输错误
4. splay不平衡

2.1 Segmentation Fault

```
正在进行第 45 组测试：
正在进入 Part V 测试。
调用 splay nullptr nullptr 方法。
测试异常： Segmentation Fault!
====================

正在进行第 46 组测试：
未完全初始化一个 bst，并对其调用 splay nullptr.
测试异常： Segmentation Fault!
====================

正在进行第 47 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.
测试异常： Segmentation Fault!
====================
```

邓同学的测试给出结果非常详细，将传入指针为空的情况总结的非常全面,我发现了很多空指针或者指针更新指向错误地址的问题。

2.2 exception

exception出现的错误主要在**find_bst**和**insert_node**部分中，当出现指针的指针的时候，我会忽略判断存储是否为空指针。

2.3 Data fault

对于**tree_count**的更新问题，在第一部分中和第三部分中，都涉及到在树中添加节点，在添加节点之后，就要考虑到node以上的**tree_count**的更新问题。我在本次使用了方法**new_all_tree_count**的方法。通过递归的方法来遍历以上的所有node中的**tree_count**

，直到达到树根**root**。

我在尝试曾同学的测试时，遇到了**add_node**中节点数据未更新的问题。

```
ericchen@EricdeMacBook-Air cpp_ass2_test-master % g++ -std=c++2a test.cpp assign2.cpp -o test.out && ./test.out
正在进行第 0 组测试：
 Error in part 1, case5!
------------------
The final result is 32/33
If you have patience, you can read my code and add some new test cases.
Hope you a good score!
If you find any bug or any wrong answer, please contact us.
测试通过！
==================
```

2.4 incorrect AVL

对于选择操作单旋还是双旋的问题上，影响是否能调整树为平衡。在邓同学的最后两个测试类中，若是只使用单旋，就会出现AVL树不平衡的情况。但是若考虑双旋的不同类型树，就能解决这个不平衡的问题。

```
正在进行第 51 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。
测试失败，错误原因：超过限制的树高，bst 实际高度：6，限制高度：4.
======================

正在进行第 52 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。
测试失败，错误原因：超过限制的树高，bst 实际高度：5，限制高度：4.
======================
```

Part 3-Conclusion

经过这次作业，我对整个指针的算法有了一个比较清晰的认识，对于结构体指针的使用以及指针在型参的中的操作，都较为熟悉。

对于整个二叉树的概念，因为有了本次作业所搭建的框架，实现理解起来都非常容易。不过在实际应用当中，却没有这么容易。其实，现在的文件系统或者数据系统，比如mysql数据库，都采用了所谓的“树”的数据结构方式（特别是B树），主要是为了排序和检索效率。而二叉树，是一种很基本又很经典的所谓的排序“树”，但是本身因为缺点比较明显所以并不常用，但却是一个很好的教学例子，来了解“树”。

我自己通过网上的学习，也自己编写了二叉搜索树的结构，从而熟悉了最基本的遍历方法（包括递归和非递归的栈结构方法）：

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct treeNode
{
    char data;              //数据域字符表示
    struct treeNode* LChild;
    struct treeNode* RChild;
```

```
}TREE,*LPTREE;
//lp一般指指针别名
/*Create a new node*/
LPTREE createNode(char data)
{
    LPTREE newNode = (LPTREE)malloc(sizeof(TREE));
    newNode->data = data;
    newNode->LChild = NULL;
    newNode->RChild = NULL;
    return newNode;
}
//Random tree
void insertNode(LPTREE father, LPTREE left_child, LPTREE right_child)
{
    father->LChild = left_child;
    father->RChild = right_child;
}

void printCurNodeData(LPTREE curData)
{
    printf("%c\t",curData->data);
}
//递归遍历
void preOrder(LPTREE root)
{
    if(root != NULL)
    {
        printCurNodeData(root); //root
        preOrder(root->LChild); //Left
        preOrder(root->RChild); //Right
    }
}

void midOrder(LPTREE root)
{
    if(root != NULL)
    {
        midOrder(root->LChild); //Left
        printCurNodeData(root); //Root
        midOrder(root->RChild); //Right
    }
}

void lastOrder(LPTREE root)
{
    if(root != NULL)
    {
        lastOrder(root->LChild); //Left
        lastOrder(root->RChild); //Right
```

```
        printCurNodeData(root);  //Root

    }
}

void preOrderBystack(LPTREE root)
{
    if (root == NULL)
    {
        return;
    }
    LPTREE stack[10];   //存储每次打印节点的位置
    int stackTop = -1;      //栈顶标记
    LPTREE pMove = root;//从根节点开始打印
    while (stackTop != -1 || pMove) //没有到栈顶，当前节点不等于空的时候
    {
        //根 左 右
        while(pMove)    //走过的节点入栈
        {
            //路径入栈
            printf("%c\t",pMove->data);
            stack[++stackTop] = pMove;
            pMove = pMove->LChild;
        }
        //无路可走
        if (stackTop != -1)
        {
            pMove = stack[stackTop];
            stackTop--;
            pMove = pMove->RChild;
        }
    }
}
//中序
void midOrderBystack(LPTREE root)
{
    if (root == NULL)
    {
        return;
    }
    LPTREE stack[10];
    int stackTop = -1;
    //定义移动的指针
    LPTREE pMove = root;
    while (stackTop != -1 || pMove)
    {
        /* 走到最左边，把走过的结点入栈 */
        while (pMove)
        {
```

```
            stack[++stackTop] = pMove;
            pMove = pMove->LChild;
        }
        //出栈
        if (stackTop != -1)
        {
            pMove = stack[stackTop--];
            printf("%c\t",pMove->data);
            pMove = pMove->RChild;
        }
    }
}

//后序 左 根 右
void lastOrderBystack(LPTREE root)
{
    if (root == NULL)
    {
        return;
    }
    LPTREE stack[10];
    int stackTop = -1;
    //定义移动的指针
    LPTREE pMove = root;
    //定义最后遍历到的结点.访问标记
    LPTREE pLastVisit = NULL;
    //左 右 根
    while (pMove)
    {
        stack[++stackTop] = pMove;
        pMove = pMove->LChild;
    }
    while (stackTop != -1)
    {
        pMove = stack[stackTop--];
        if (pMove->RChild == NULL || pMove->RChild == pLastVisit)//左边是不是空
的，右边是不是已经被访问了
        {
            //如果被访问就可以打印 左右访问过才能访问根
            printf("%c\t",pMove->data);
            pLastVisit = pMove; //保存位置 标记
        }
        else
        {
            //右边没有被访问
            stack[++stackTop] = pMove;
            pMove = pMove->RChild;
            while (pMove)
            {
```

```
                stack[++stackTop] = pMove;
                pMove = pMove->LChild;
            }
        }
    }
}
int main()
{
    LPTREE A = createNode('A');
    LPTREE B = createNode('B');
    LPTREE C = createNode('C');
    LPTREE D = createNode('D');
    LPTREE E = createNode('E');
    LPTREE F = createNode('F');
    LPTREE G = createNode('G');
    insertNode(A,B,C);
    insertNode(B,D,NULL);
    insertNode(D,NULL,G);
    insertNode(C,E,F);
    preOrder(A);
    printf("\n");
    midOrder(A);
    printf("\n");
    lastOrder(A);
    printf("\n非递归的\n");
    preOrderBystack(A);
    printf("\n中序\n");
    midOrderBystack(A);
    printf("\n后序\n");
    lastOrderBystack(A);
    //system("pause");
    return 0;
}
```

本次二叉树的学习虽然不能真的利用二叉树进行什么应用，但是对数据结构能够有一定的理解和认识。虽然这次作业对于我们难度很大，但是从中也是受益匪浅，通过实践得到了许多知识和经验，因此要感谢出题的各位SA和TA们。这次非常感谢计算机系的同学们的帮助，我们因为不以计算机课程为主修，对数据结构的认识远不如计系的同学们深刻。在这个时候有同学能够开源测试项，真的令人感动。

卷中自有温情在，以文常会友，唯德自成邻。—— 写在这次Assignment的最后。