Dynamic Tree 个人分报告：Model 层

李举仁 3170104559

1 Introduction

本次项目采用 MVVM 模式实现 Dynamic Tree。MVVM 具有低耦合、可重用性、独立开发和可测试的优点。Dynamic Tree 实现了 Binary Search Tree、AVL Tree 和 Splay Tree 的可视化显示，以及动态演示这三种数据结构的插入、删除、查询等的具体过程，可让初学者更容易理解明白这三种数据结构。本分报告简单介绍了 Model 层的设计原理、工作流程和效果，并简要阐述在此次暑期课程中的感悟。

2 Design Principle

Model 层是整个项目中位于最底层。它提供应用需要的数据类和业务逻辑、提供数据的算法类。当数据类内部数据发生改变时，发出通知。

Model 层提供了 Insert ()、Delete ()、Find () 三个基本操作的公有成员函数，PreOder()、InOder()、PostOrder()三个不同遍历方法的公有成员函数。此外，也提供了 ChangeMode（）函数用于选择数据类型，GetFlashNode（)、GetNowNode（）来访问私有成员。以上九个函数是 ViewModel 层可直接访问的接口函数。通过这些接口函数 ViewModel 层可调用 Model 层的函数实现业务逻辑。

Model 层的九个公有成员函数中，通过对应的逻辑调用其他的私有成员函数。在增加功能或者修改功能时，只需修改私有成员函数，而不需要改变接口，做到了 Model 层与其他各层的独立。因此在确定接口之后，ViewModel 层与 Model 层都可以无需知道对方的具体实现而独立开发，这体现了 MVVM 的低耦合性和独立开发的特点。

3 Module Introduction

Model 层主要由两部分构成，提供给 ViewModel 的公有接口函数和实现业务逻辑的私有函数，业务逻辑主要分为 Binary Tree、AVL Tree、Spaly Tree 三个部分。除此之外，还有四个成员变量，用于储存必要的数据。

3.1 Variables

globalMode：枚举类型，用于记录当前的数据结构类型；

```
typedef enum {BST, AVL, Splay, MinH, MaxH} controlMode;
```

同时，ViewModel 可通过调用 ChangeMode（std::string& Mode）来改变当前的数据结构类型；

```
ModelClass::ModelClass(): globalMode(controlMode::BST), Treeptr(nullptr)
{
}

bool ModelClass::ChangeMode(std::string& Mode)
{
    if(Mode == "Binary Search Tree"){
        globalMode = controlMode::BST;
    }
    else if (Mode == "AVL Tree") {
        globalMode = controlMode::AVL;
    }
    else {
        globalMode = controlMode::Splay;
    }
    Tree_DestroyTree(Treeptr);
    Treeptr = nullptr;
    return true;
}
```

TreePtr: 记录当前 Tree 的头节点；

FlashNode: 用来记录要闪烁的节点；

其对应的 Get 函数：

```
TreeNode* ModelClass::GetFlashNode() const throw()
{
    return FlashNode;
}
```

NowNode: 用来记录当前要操作的节点；

其对应的 Get 函数：

```
TreeNode* ModelClass::GetNowNode() const throw()
{
    return NowNode;
}
```

3.2 Binary Search Tree

BinarySearchTree 是这三个数据结构中最基本的一个数据结构，它的插入、删除、寻找的实现相对来说比较简单。

在插入时，先查找到应当插入的位置，同时通过 Fire_PropertyChanged()函数向上层发送信号，闪烁每一个经过的节点，同时也可以发送信号表示已经插入。

```
1.  TreeNode* ModelClass::BST_Insert(TreeNode* T, ElementType X, int Index)
2.  {
3.      if (T == nullptr)
4.      {
5.
6.          T = new TreeNode(X, Index);
7.
8.          NowNode = T;
9.
10.         Fire_OnPropertyChanged("Node_Insert");
11.
12.     }
13.     else
14.     {
15.         FlashNode = T;
16.         Fire_OnPropertyChanged("Node_Flash");
17.
18.         if (T->Value > X)
19.         {
20.             T->LeftChild = BST_Insert(T->LeftChild, X, Index*2);
21.         }
22.         else
23.             T->RightChild = BST_Insert(T->RightChild, X, Index*2+1);
24.     }
25.     return T;
26. }
```

删除时，先找到要删除的节点，再用其左子树的最大节点将欲删除的节点替代；然后删除左子树的最大节点，同样向 ViewModel 发送对应的信号：

```
1.  TreeNode* ModelClass::BST_Delete(TreeNode* T, ElementType X)
2.  {
3.      if (T == nullptr) {
4.          throw "No such element";    //throw an exception
5.      }
6.
7.      FlashNode = T;
8.      Fire_OnPropertyChanged("Node_Flash");
9.
10.     if (T->Value < X)
11.         T->RightChild = BST_Delete(T->RightChild, X);
12.     else if (T->Value > X)
13.         T->LeftChild = BST_Delete(T->LeftChild, X);
14.     else
15.     {
16.         NowNode = T;
17.         Fire_OnPropertyChanged("Node_Delete");
18.
19.         if (T->LeftChild != nullptr && T->RightChild != nullptr)
```

```
        {
            T->Value = (FindMax(T->LeftChild))->Value;
            T->LeftChild = BST_Delete(T->LeftChild, T->Value);

            NowNode = T;
            Fire_OnPropertyChanged("Node_ChangeValue");

        }
        else
        {
            TreeNode* TempTreeNode = T;
            T = (T->LeftChild != nullptr) ? T->LeftChild : T->RightChild;

            NowNode = TempTreeNode;
            Fire_OnPropertyChanged("Node_RecurDelete");

            if (T != nullptr)
            {
                Tree_UpdateIndex(T, T->Index / 2);   //update the index of the nodes

                NowNode = T;
                Fire_OnPropertyChanged("Node_RecurUpdate");
            }
            delete TempTreeNode;
        }
    }
    return T;
}
```

Find()函数直接利用其树的性质，二叉查找，同时通过 Fire_OnPropertyChanged()函数，向上发送对应的闪烁信号。

```
void ModelClass::BST_Find(TreeNode* T, ElementType X)
{
    if (T == nullptr) {
        throw "No such element";    //throw an excepion
    }
    if (T->Value == X) {

        NowNode = T;
        Fire_OnPropertyChanged("Node_Find");

        return;
    }

    FlashNode = T;
    Fire_OnPropertyChanged("Node_Flash");

    if (T->Value < X)
        BST_Find(T->RightChild, X);
    else
        BST_Find(T->LeftChild, X);
}
```

3.3 AVL Tree

AVL Tree 相对于 Binary Search Tree 增加平衡因子和左右旋转。

```
TreeNode* SingleLeftRotation(TreeNode* A);
TreeNode* SingleRightRotation(TreeNode* A);
TreeNode* DoubleLRRotation(TreeNode* A);
TreeNode* DoubleRLRotation(TreeNode* A);
```

以左旋为例，先发送信号告知上层需要进行左旋了，此时界面应当显示一个旋转符号；然后对节点的部分结构进行更新，交换当前节点 A 与其左儿子 B 的位置；之后向上层发送信号，隐藏左儿子 B 的右子树 BR；然后先隐藏节点 B 及其子树（此时 A 已为 B 的右子树），更新节点 B 节点及其子树的 Index（ModelView 根据数据结构中的 Index 进行解码节点的位置，便于 View 的显示）。随后再次显示 B 节点与其子树，最后更新剩余的结构（将 BR 接到 A 的儿子的位置上），然后向上层发出信号显示 BR。在旋转完成之后再更新平衡因子。

在该部分 Dynamic Tree 的关键就在于何时向上层发怎样的信号：

```
TreeNode* ModelClass::SingleLeftRotation(TreeNode* A)
{
    FlashNode = A;
    Fire_OnPropertyChanged("Left_Rotation");

    TreeNode* B = A->LeftChild;
    TreeNode* TempNode = B->RightChild;

    if(A->Parent!=nullptr){
        if(A == A->Parent->LeftChild){
            A->Parent->LeftChild = B;
        }
        else {
            A->Parent->RightChild = B;
        }
    }
    B->Parent = A->Parent;
    A->Parent = B;
    if(TempNode!= nullptr){
        TempNode->Parent = A;
        NowNode = TempNode;
        Fire_OnPropertyChanged("Node_RecurDelete");
    }

    B->RightChild = nullptr;

    NowNode = A;
    Fire_OnPropertyChanged("Node_RecurDelete");

    B->RightChild = A;
    A->LeftChild = nullptr;
    Tree_UpdateIndex(B, A->Index);

    NowNode = B;
    Fire_OnPropertyChanged("Node_RecurUpdate");

    A->LeftChild = TempNode;
    Tree_UpdateIndex(TempNode, A->Index * 2);

    if(TempNode != nullptr){
        NowNode = TempNode;
        Fire_OnPropertyChanged("Node_RecurUpdate");
    }

```

```
    A->Factor = Max(GetFactor(A->LeftChild), GetFactor(A->RightChild)) + 1;

    B->Factor = Max(GetFactor(B->LeftChild), GetFactor(B->RightChild)) + 1;

    return B;
}
```

在插入对应的节点后，对平衡因子进行判断，决定是否执行左旋、右旋或者双璇：

```
if(T->Value > X){
    T->LeftChild = AVL_Insert(T->LeftChild, X, Index*2);
    if((GetFactor(T->LeftChild)-GetFactor(T->RightChild)) == 2){
        if(X < T->LeftChild->Value)
            T = SingleLeftRotation(T);
        else
            T = DoubleLRRotation(T);
    }
}
else {
    T->RightChild = AVL_Insert(T->RightChild, X, Index*2+1);
    if((GetFactor(T->LeftChild)-GetFactor(T->RightChild)) == -2){
        if(X >= T->RightChild->Value)
            T = SingleRightRotation(T);
        else
            T = DoubleRLRotation(T);
    }
}
```

同理，当删除一个节点之后也进行判断，执行左旋、右旋或者双璇：

```
T->LeftChild = AVL_Delete(T->LeftChild, X);
if((GetFactor(T->LeftChild) - GetFactor(T->RightChild)) == -2){
    if(T->RightChild->LeftChild == nullptr||
            (GetFactor(T->RightChild->LeftChild) - GetFactor(T->RightChild->
RightChild)) == -1){
        T = SingleRightRotation(T);
    }
    else {
        T = DoubleRLRotation(T);
    }
}
```

3.4 Splay Tree

与 AVL Tree 一样 Splay Tree 在 Binary Tree 的基础上增加了旋转；它的特性是每一次访问一个节点，都会通过 zig、zag（相当于左、右旋转）以及其组合将其旋转到根节点，从而达到减少树的高度的目的。它的单璇与 AVL Tree 一样，因此直接调用的 AVL 的旋转。

```
TreeNode* LLRotation(TreeNode* G);
TreeNode* LRRotation(TreeNode* G);
TreeNode* RRRotation(TreeNode* G);
TreeNode* RLRotation(TreeNode* G);
```

每次访问节点之后（insert、delete、find 操作之后），都会通过 Spaly()函数将其旋转到根节点。因此，在执行插入、删除和寻找时，基本上只需要在 Binary Searche Tree 的基础上调用该函数就行；

```
TreeNode* ModelClass::Splay(TreeNode* T, TreeNode* X)
{
    if (T == nullptr)
        return nullptr;
    TreeNode *P, *G;    //ParentTree, GrandParentTree
    P = X->Parent;
    while (P != nullptr)
    {
        G = P->Parent;
        if (X == P->LeftChild)
        {
            if (G == nullptr)
                X = SingleLeftRotation(P);
            else if (P == G->LeftChild)
                X = LLRotation(G);
            else
                X = RLRotation(G);
        }
        else
        {
            if (G == nullptr)
            {
                X = SingleRightRotation(P);
            }
            else if (P == G->RightChild)
            {
                X = RRRotation(G);
            }
            else
            {
                X = LRRotation(G);
            }
        }
        P = X->Parent;
    }
```

3.5 Travers

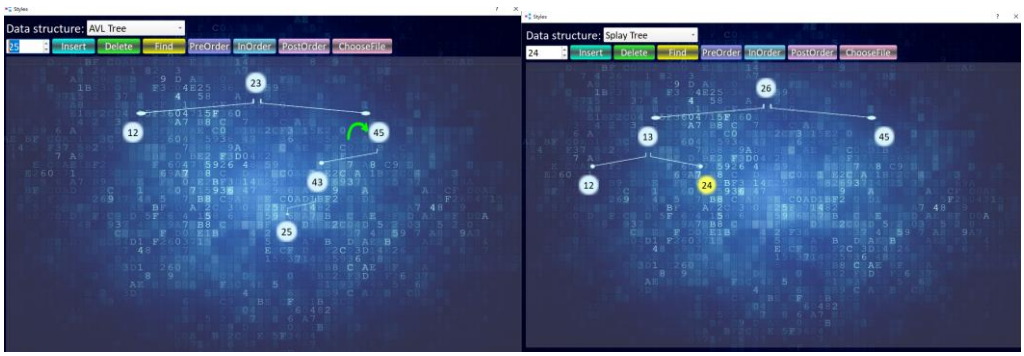
Dinamic Tree 也实现了前序、中序、后序的遍历过程。通过递归的方式实现这三个遍历，同时通过 Fire_OnPropertyChange()向上层发送信号标记闪烁所经过的节点；

```
void ModelClass::Tree_PostOrder(TreeNode* T)
{
    if (T == nullptr)
        return;
    Tree_PostOrder(T->LeftChild);
    Tree_PostOrder(T->RightChild);
    FlashNode = T;
    Fire_OnPropertyChanged("Node_Flash");
}

void ModelClass::Tree_PreOrder(TreeNode* T)
{
```

```
13.     if (T == nullptr)
14.         return;
15.
16.     FlashNode = T;
17.     Fire_OnPropertyChanged("Node_Flash");
18.
19.     Tree_PreOrder(T->LeftChild);
20.     Tree_PreOrder(T->RightChild);
21. }
22.
23. void ModelClass::Tree_InOrder(TreeNode* T)
24. {
25.     if (T == nullptr)
26.         return;
27.     Tree_InOrder(T->LeftChild);
28.
29.     FlashNode = T;
30.     Fire_OnPropertyChanged("Node_Flash");
31.
32.     Tree_InOrder(T->RightChild);
33. }
```

4 运行截图



5 心得体会

此次课程我学到了在应用型程序中工具链的使用，让所学的知识能够更加实用化。学会了基本的 Git、Github 的使用方法，持续集成、单元测试、代码覆盖率测试等的工具的使用。通过 Git 与 Github 使得团队的协作更加高效，虽然在最初磕磕碰碰不太熟练，但经过一个项目的练习，已经掌握了基本的操作。单元测试、代码覆盖率的测试等工具的使用让我进一步了解了自己以及团队的代码质量如何。

在课程的学习中，进一步了解了 C++面向对象编程，初步了解了程序框架一步步从 MVC 到 MVP 最后到 MVVM 的过程。在团队进行项目的一开始，就打算用 MVVM 模式进行项目的开发，但是由于对 MVVM 的理解不够通透，只是在形式上将整个项目分成 View、Model、

View Model 三部分，而实际上却是 MVC 模式。这样在功能比较少的时候还没什么问题，但是当增加功能的时候，往往会出现一个人等待另一个人的模块的完成的情况，因为不同模块之间的耦合程度太大，而不能独立开发。在老师与其他同学的帮助之下，我们团队最终顺利地转向 MVVM。在后续的开发过程中，由于各个模块之间没有直接的联系，可以做到独立开发，从而使得整个团队更加高效。

此次学习，不仅了解了 C++在项目工程中的开发与实践，更加明白了一个合理的程序框架对整个程序的开发的重要性。最后，在验收之时，老师有提到在工业中红黑树的应用最广泛，而我们开发时并没有注意到这一点。这让我感受到，预先的一些调查，与工业相结合也许做出的项目在实用性方面有更大提升。

6 改进建议

此次小学期学到了很多非常使用的知识，让我对工程项目中的 C++编程有了初步的理解，收获满满。

团队最初的跑偏，做成 MVC 模式是我们对 MVVM 的理解不够，也希望老师能够对 MVVM 的具体实现（C++代码）能够进行简要的介绍，这样也许对同学来说更容易理解。

谢谢老师!!!