《数据结构》课程实践报告

院、系	计算机学院	年级专业	21 计科	姓名	方浩楠	学号	2127405048
实验布置日期	2022.11.1		提交日期	2022.11.20		成绩	

课程实践实验 7：BST 的实现和测试

一、问题描述及要求

对 BST 进行基本的操作，并且对 BST 进行查找，删除，插入等操作。

二、概要设计

（1） 对实验内容的理解

实验要求实现 BST，其中 BST 的每个节点通过 BST_Node 存储。BST_Node 中包括了该节点存储的数据，左孩子，右孩子以及双亲。

（2） BST 的功能列表

1. 构造函数
2. 析构函数
3. 前序遍历
4. 中序遍历
5. 后续遍历
6. 层序遍历
7. 找出 BST 的最大值
8. 找出 BST 的最小值
9. 在 BST 中查找某个值是否存在
10. 在 BST 中求某个节点的前驱(Predecessor)
11. 在 BST 中求某个节点的后驱(Successor)
12. 求出 BST 的高度，节点数，叶节点数
13. 在 BST 中删除某个节点

（3）程序的界面设计

```
中序遍历:
2       3       4       6       7       9       13      15      17      18      20

后续遍历:
2       4       3       9       13      7       6       17      20      18      15

前序遍历:
15      6       3       2       4       7       13      9       18      17      20

层序遍历:
第1层:  15
第2层:  6       18
第3层:  3       7       17      20
第4层:  2       4       13
第5层:  9

BST的高度:      5
BST的节点数:    11
BST叶节点数:    5

在BST中查找data为4的节点:
找到

在BST中查找data为12的节点:
未找到

拷贝BST
```

```
拷贝BST

遍历拷贝之后的BST:
第1层:  15
第2层:  6       18
第3层:  3       7       17      20
第4层:  2       4       13
第5层:  9

在BST中删除data为3的节点

删除后的BST:
中序遍历:
2       4       6       7       9       13      15      17      18      20

后续遍历:
2       4       9       13      7       6       17      20      18      15

前序遍历:
15      6       4       2       7       13      9       18      17      20

层序遍历:
第1层:  15
第2层:  6       18
第3层:  4       7       17      20
第4层:  2       13
第5层:  9

Release

进程已结束,退出代码0
```

（4）总体设计思路

BST 的每个节点都是 BST_Node 类型的，BST_Node 的定义如下:

```
struct BST_Node
{
    //Node->left->data < Node->data < Node->right->data
    int data;
    BST_Node *left;
    BST_Node *right;
    BST_Node *parent;
};
```

其中 Node->left->data < Node->data < Node->right->data 是 BST 的定义，即每个节点左子树中的每个数均小与这个节点的值，而每个节点的右子树中的每个数均大于这个节点的值

对 BST 多种操作存储在了 class BST 中，BST 的 public 中的定义如下:

```
class BST()
{
public:
    BST();                                  //构造函数
    BST(const vector<BST_Node *> &v);       //构造函数
    ~BST()                                  //析构函数
    BST_Node* GetRoot()                     //获取 BST 的 root
    void Insert(BST_Node *z);               //向 BST 中插入一个节点 z
    void PreorderTreeWalk()                 //先序遍历 BST
    void InorderTreeWalk()                  //中序遍历 BST
    void PostTreeWalk()                     //后序遍历 BST
    void LevelTreeWalk();                   //层序遍历 BST
    void LevelTreeWalk2();                  //层序遍历 BST
    BST_Node *Maximum()                     //求 BST 的最大值
    BST_Node *Minimum()                     //求 BST 的最小值
    BST_Node *TreeSearch(int k)             //在 BST 中查找值为 k 的节点
    BST_Node *IterativeTreeSearch(int k)    //迭代方式对 BST 进行查找
    BST_Node *TreeSuccessor(BST_Node *x);   //求 BST 中某个节点的后继
    BST_Node *TreePredecessor(BST_Node *x); //求 BST 中某个节点的前驱
    int GetTreeSize()                       //获取 BST 的节点数
    int GetLeafSize()                       //获取 BST 的叶节点数
    int GetHeight()                         //获取 BST 的高度
    void TreeDelete(BST_Node* z);           //删除 BST 中的节点 z
}
```

（5）程序结构设计

1. utility.h　声明头文件
2.BST.h　声明 BST 类
3.main.cpp　主函数

三、详细设计

（1）构造函数 BST(const vector<BST_Node *> &v)

通过一个 BST_Node 类型的 vector 来构造一棵 BST, 构造的方式为先构造一棵空二叉树，然后遍历 vector v, 对 v 中的每个 BST_Node 类型的节点不断的调用 void Insert(BST_Node *z), 从而通过 vector<BST_Node *>来构造一棵二叉树

（2）析构函数

需要调用 private 中的 Release()函数。Release()的作用为后续遍历 BST，并且不断的去 free 每个节点.

（3）void Insert(BST_Node *z);

Insert 操作是将一个新节点 z 插入 BST 中。Insert 操作与 Search 操作类似，Insert 操作需要一个指针 x，该指针记录了一条向下的简单路径，并且查找到一个 nullptr，该 nullptr 需要用来存放需要插入 BST 的新节点 z。该函数同时还需要一个遍历指针(trailing pointer)y，指针 y 需要作为 x 的双亲(parent)。在函数中需要利用 while 循环使得指针 x 和指针 y 下移，两个指针是向左移动还是向右移动是通过比较 z->data 和 x->data。While 循环在 x 变成 nullptr 后终止。此时这个 nullptr 就是新节点 z 需要插入的位置。

（4）PreorderTreeWalk(),InderTreeWalk(),PostorderWalk(),LevelTreeWalk()

分别是对 BST 进行前序遍历，中序遍历，后续遍历和层序遍历，与二叉树中的遍历相同

（5）LevelTreeWalk2()

对 BST 进行层序遍历，但是输出时会根据每一层来输出。该函数需要调用 private 中的 vector<vector<int>> LevelTreeWalk2(BST_Node *x);

vector<vector<int>> LevelTreeWalk2(BST_Node *x)函数用来对 BST 进行层序遍历，并且将遍历结果存储在一个二维向量 vector<vector<int>> ans 中。该函数的实现方式为首先使用一个 std::queue<BST_Node*> q 来存储节点，在使用一个 std::vector<int> v 来存储每一层的节点中的数据。首先将 BST 根节点插入队列 q，然后只要 q 不为空，就用 for 循环进行 q.size()次循环，每次 for 循环时都将 q 的队头元素弹出，然后将队头元素中的 data push_back 进 std::vector<int> v，并且将队头元素的左右孩子插入队列的尾部（前提:队头元素的左孩子右孩子不是 nullptr，若是 nullptr 就不插入队列）。每次 for 循环结束后都将向量 v push_back 进二维向量 ans 中。最终当队列 q 中没有元素之后，返回二维向量 ans。

（6）BST_Node *Maximum()

求 BST 中的最大值所在的节点。根据 BST 性质:每个节点的右子树中的每个节点的值均大于这个节点的值，从而得知 BST 中最大节点一定是沿着根节点向右移动，直到某个节点 node，其中 node->right == nullptr 时，node 节点中的值就是整个 BST 中最大的值。

（7）BST_Node *Minimum()

求 BST 中的最小值所在的节点。求最小值的方式与求最大值是对称的。根据 BST 的性质:每个节点的左子树中的每个节点的值均小与这个节点的值，从而得知 BST 中的最小节点一定是沿着根节点向左移动，直到某个节点 node，其中 node->left == nullptr 时，node 节点中的值就是整个 BST 中的最小的值。

（8）BST_Node *TreeSearch(int k)

利用递归的方式在 BST 中查找是否存在值为 k 的节点。该函数需要调用 private 中的 BST_Node *TreeSearch(BST_Node *x, int k)函数。

其中 BST_Node *TreeSearch(BST_Node *x, int k)函数是从根节点开始查找，并沿着这个 BST 的一条简单路径向下。对于需要的每个节点 x，都会比较 k 和 x->data 的大小，若两个值相等，则终止查找。若 k < x->data，则在 x 的左子树中查找，因为根据 BST 的性质(每个节点的右子树中的每个节点的值均大于这个节点的值)，因此 k 不可能在节点 x 的右子树中。对称的，如果 k > x->data ，则查找在右子树中进行。

（9）BST_Node *IterativeTreeSearch(int k)

利用迭代的方式在 BST 中查找是否存在值为 k 的节点。本函数的原理与 BST_Node *TreeSearch(int k)相同，只是采用的方式不同。本函数使用了迭代的方式，而上一个函数使用了递归的方式。

（10）BST_Node *TreeSuccessor(BST_Node *x)

给定一个节点 x，有时需要按照中序遍历的顺序来查找它的后继，即求 BST 中大于 x->data 的节点中的最小关键字的节点，即求 x 的后继(Successor)。BST 的构造使我们可以不通过比较就确定一个节点的后继。实现方式如下:

如果节点 x 的右子树不为空，则 x 的后继就是 x 的右子树中最靠左的节点

若 x 没有右节点，并且 x 有一个后继 y，则 y 就是 x 的最底层祖先，同时，y 的左孩子也是 x 的一个祖先。为了找到 y，我们需要令节点 x 沿着 BST 向上，直到遇到这样一个节点，这个节点是它的双亲的左孩子，此时，这个节点就是 x 的后继 y。

（11）BST_Node *TreePredecessor(BST_Node *x)

给定一个节点 x，求 x 的前驱。实现方式如下:

如果 x 的左子树不为空，则 x 的前驱就是 x 的左子树中最靠右的节点

若 x 没有左节点，并且 x 有一个前驱 y，那么 y 就是 x 的最底层祖先，同时 y 的右孩子也是 x 的一个祖先。为了找到 y，我们需要令节点 x 沿着 BST 向上，直到遇到这样一个节点，这个节点是它的双亲的右孩子，此时，这个节点就是 x 的前驱 y。

（12）GetTreeSize(),GetLeafSize(),GetHeight()

这三个函数分别求 BST 的节点数，BST 的叶节点数，BST 的高度。这三个函数的实现方式与二叉树相同。

（13）void TreeDelete(BST_Node* z)

该函数用来删除 BST 中的节点 z。删除操作分为三种情况

情况一：如果 z 没有孩子节点，那么就是简单的将它删除，并修改它的父节点，并且用 nullptr 作为孩子来替换 z

情况二：如果 z 只有一个孩子，那么就将这个孩子提升到树中 z 的位置上，并且修改 z 的父节点，用 z 的孩子来替换 z。

情况三：如果 z 有两个孩子，那么找 z 的后继 y（由于 z 有右子树，因此 z 后继一定在 z 的右子树中），并且让 y 占据树中 z 的位置。z 原来的右子树成为 y 的新的右子树，并且 z 的左子树成为 y 的新的左子树。

情况三较为棘手，因为情况三还与 y 是否是 z 的右孩子有关。若 y 是 z 的右孩子，那么就用 y 来替换 z，并且仅留下 y 的右孩子。否则，y 在 z 的右子树中单不是 z 的右孩子。在在这种情况下，先用 y 的右孩子替换 y，然后再用 y 替换 z。

为了在 BST 中移动子树，我们需要调用 private 中的 void Transplant(BST_Node *u, BST_Node *v)函数。该函数是用一棵以 v 为根的子树替换一棵以 u 为根的子树，并成为其双亲的孩子节点。

在删除 z 节点时，若 z 节点没有左孩子，就把以 z 的右孩子为根的子树移到 z 的位置。

若 z 节点有左孩子但没有右孩子，就把以 z 的左孩子为根的子树移到 z 的位置。

剩下的情况中，找到 z 的后继 y，让 y 占据树中 z 的位置。z 原来的右子树成为 y 的新的右子树，并且 z 的左子树成为 y 的新的左子树。

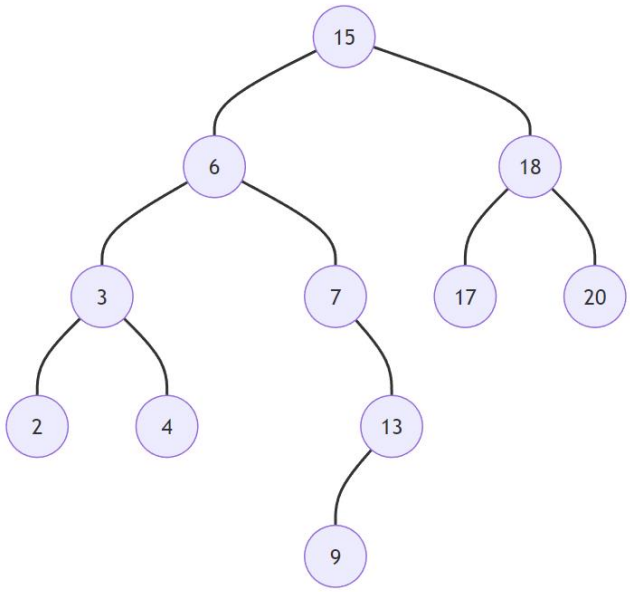
四、实验结果

第一组测试 测试 BST 的构造以及 BST 的遍历

测试输入:

```
auto a = new BST_Node;
auto b = new BST_Node;
auto c = new BST_Node;
auto d = new BST_Node;
auto e = new BST_Node;
auto f = new BST_Node;
auto g = new BST_Node;
auto h = new BST_Node;
auto i = new BST_Node;
auto j = new BST_Node;
auto k = new BST_Node;
a->data = 15;
b->data = 6;
c->data = 3;
d->data = 2;
e->data = 4;
f->data = 7;
g->data = 13;
h->data = 9;
i->data = 18;
j->data = 17;
k->data = 20;
vector<BST_Node*>v{a,b,c,d,e,f,g,h,i,j,k};
BST(Tree){v};
```

该输入构建的 BST 为:



正确输出:

前序遍历: 15　6　3　2　4　7　13　9　18　17　20
中序遍历: 2　3　4　6　7　9　13　15　17　18　20
后续遍历: 2　4　3　9　13　7　6　17　20　18　15
层序遍历:15　6　18　3　7　17　20　2　4　13　9

实际输出:

```
D:\Programming\C-CPP\Csteaching\experiment7-BST\cmake-build-debug\experiment7_BST.exe
中序遍历:
2       3       4       6       7       9       13      15      17      18      20

后续遍历:
2       4       3       9       13      7       6       17      20      18      15

前序遍历:
15      6       3       2       4       7       13      9       18      17      20

层序遍历:
第1层:  15
第2层:  6       18
第3层:  3       7       17      20
第4层:  2       4       13
第5层:  9

BST的高度:      5
BST的节点数:    11
BST叶节点数:    5
```

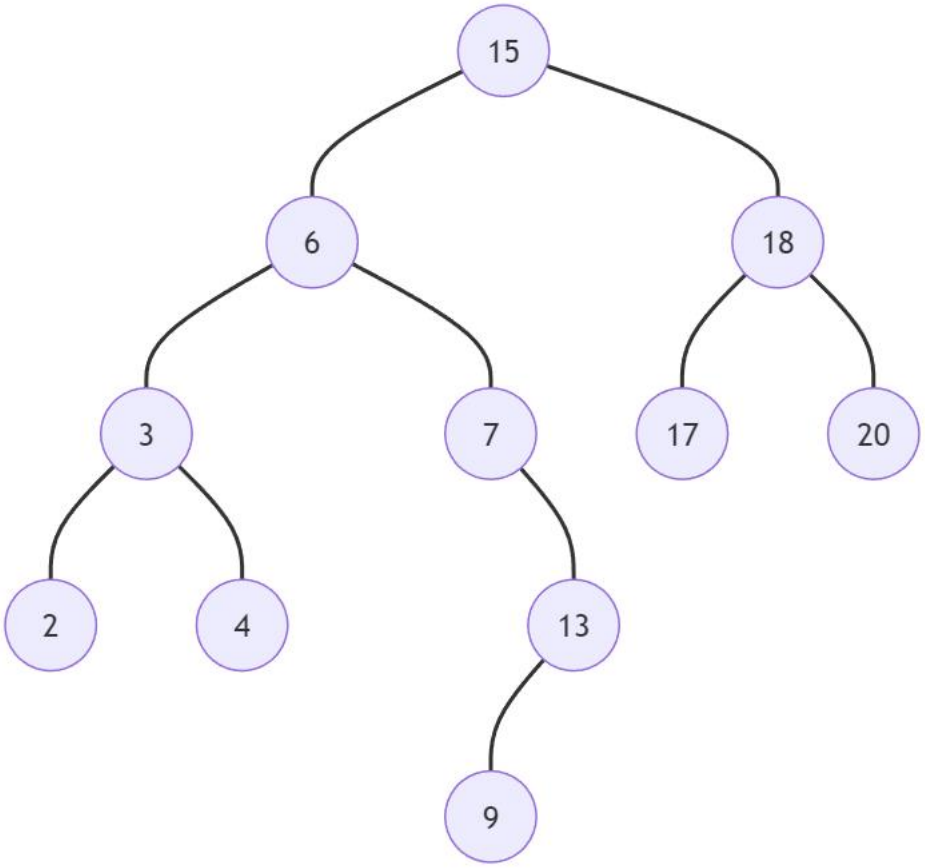
测试结果:正确

第二组测试 测试 BST 的查找以及删除

测试数据:在刚刚构建的 BST 中查找 data 为 4 的节点，在刚刚构建的 BST 中查找 data 为 12 的节点，在 BST 中删除 data 为 6 的节点

正确输出:

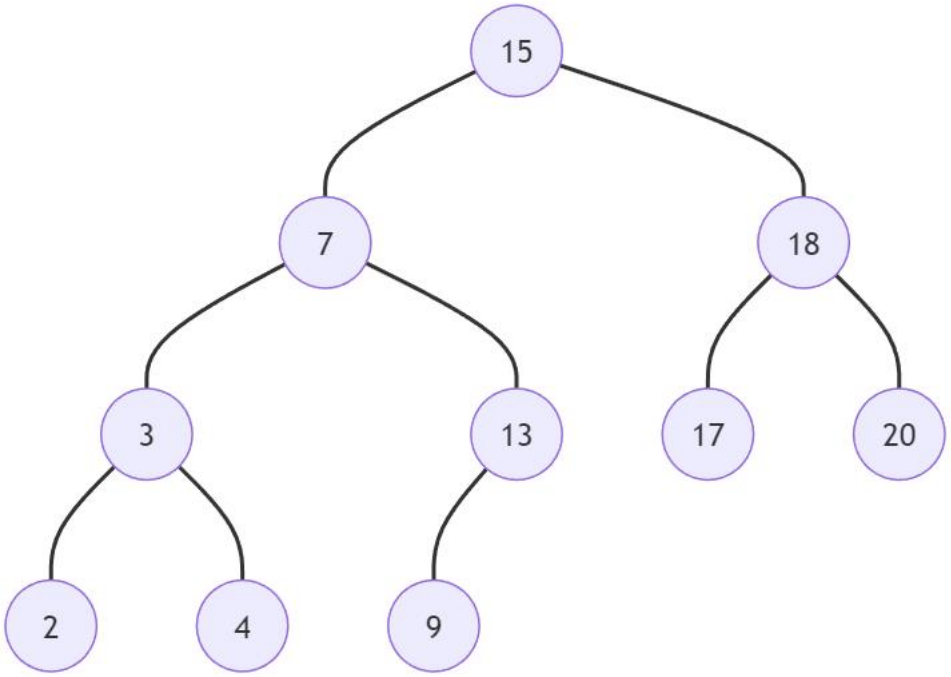
删除节点之前的BST



该 BST 中存在 data 为 4 的节点，不存在 data 为 12 的节点，因此 4 能够找到，但是 12 不能找到。

删除 data 为 6 的节点后，BST 应为:

删除节点之后的BST



实际输出:

```
在BST中查找data为4的节点:
找到

在BST中查找data为12的节点:
未找到
```

```
在BST中删除data为6的节点

删除后的BST:
中序遍历:
2       3       4       7       9       13      15      17      18      20

后续遍历:
2       4       3       9       13      7       17      20      18      15

前序遍历:
15      7       3       2       4       13      9       18      17      20

层序遍历:
第1层:  15
第2层:  7       18
第3层:  3       13      17      20
第4层:  2       4       9
```

测试结果:正确

五、实验分析与探讨

二叉搜索树中，前序，中序和后序遍历的时间复杂度均为$T(n) = O(n)$。

查找，求最大值，求最小值，求前驱，求后继，插入以及删除的时间复杂度均为$T(n) = O(h)$，其中h为树的高度。在 BST 是随机构建的情况下，二叉树的期望高度为$O(logn)$，因此上述所说的操作的评价运行时间为$T(n) = O(logn)$。

由于 BST 比较容易获取最大最小值，因此 BST 有时也可以用来实现优先队列。

六、小结

到此，二叉搜索树的实验便已完成。我较为顺利的完成了实验所要求的内容，同时增强了自己的思维能力以及编程能力

程序的局限性:

虽然 BST 在大多数时候都能有着不错的性能，并且删除，插入，查找等操作的效率较高。但是某些情况下 BST 在最坏情况下的性能十分恶劣，比如 BST 退化成链表的时候。此时，我们需要寻找更加优秀的算法以及数据结构，比如 AVL 树和红黑树，而这也将是我下一步学习的方向。

附录：源代码

（1） 实验环境

编译器:mingw
gcc version 8.1.0 (x86_64-posix-seh-rev0，Built by NinGW-w64 project)
C++版本:C++ 17

（2） 源代码

utility.h

```
#include<iostream>
#include<queue>
#include<stack>
#include<string>
#include<vector>
```

```cpp
#include<cstdlib>
using namespace std;
```

BST.h

```cpp
#include "utility.h"

struct BST_Node
{
    //Node->left->data < Node->data < Node->right->data
    int data;
    BST_Node *left;
    BST_Node *right;
    BST_Node *parent;
};

class BST
{
public:
    BST();

    BST(const vector<BST_Node *> &v);


    ~BST()
    {
        cout<<"Release"<<endl;
        Release(_root);
        exit(0);
    }

    BST_Node* GetRoot()
    {return _root;}

    void Insert(BST_Node *z);

    void PreorderTreeWalk()
    { PreorderTreeWalk(_root); }

    void InorderTreeWalk()
    { InorderTreeWalk(_root); }
```

```
    void PostTreeWalk()
    { PostorderTreeWalk(_root); }

    void LevelTreeWalk();

    void LevelTreeWalk2();

    BST_Node *Maximum()
    { return Maximum(_root); }

    BST_Node *Minimum()
    { return Minimum(_root); }

    BST_Node *TreeSearch(int k)
    { return TreeSearch(_root, k); }

    BST_Node *IterativeTreeSearch(int k)
    { return IterativeTreeSearch(_root, k); }

    BST_Node *TreeSuccessor(BST_Node *x);

    BST_Node *TreePredecessor(BST_Node *x);

    int GetTreeSize()
    { return GetTreeSize(_root); }

    int GetLeafSize()
    { return GetLeafSize(_root); }

    int GetHeight()
    { return GetHeight(_root); }

    void TreeDelete(BST_Node* z);

private:
    void Release(BST_Node* x);

    void PreorderTreeWalk(BST_Node *x);

    void InorderTreeWalk(BST_Node *x);

    void PostorderTreeWalk(BST_Node *x);

    vector<vector<int>> LevelTreeWalk2(BST_Node *x);
```

```
    BST_Node *Maximum(BST_Node *x);

    BST_Node *Minimum(BST_Node *x);

    BST_Node *TreeSearch(BST_Node *x, int k);

    BST_Node *IterativeTreeSearch(BST_Node *x, int k);

    int GetTreeSize(BST_Node *x);

    int GetLeafSize(BST_Node *x);

    int GetHeight(BST_Node *x);

    void Transplant(BST_Node *u, BST_Node *v); // 用一棵以 v 为根的子树替代一棵以 u 为根的子树

    BST_Node *_root{};
};

BST::BST()
{
    _root = nullptr;
}

BST::BST(const vector<BST_Node *> &v)
{
    for (auto i: v)
    {
        Insert(i);
    }
}

void BST::Insert(BST_Node *z)
{
    BST_Node *y = nullptr;
    BST_Node *x = this->_root;
    while (x != nullptr)
    {
        y = x;
        if (z->data < y->data)
        {
            x = x->left;
```

```
        }
        else
        {
            x = x->right;
        }
    }
    z->parent = y;
    if (y == nullptr)
    {
        this->_root = z; //empty tree
    }
    else if (z->data < y->data)
    {
        y->left = z;
    }
    else
    {
        y->right = z;
    }
    z->left = nullptr;
    z->right = nullptr;
}

void BST::PreorderTreeWalk(BST_Node *x)
{
    if (x != nullptr)
    {
        cout << x->data << "\t";
        PreorderTreeWalk(x->left);
        PreorderTreeWalk(x->right);
    }
}

void BST::InorderTreeWalk(BST_Node *x)
{
    if (x != nullptr)
    {
        InorderTreeWalk(x->left);
        cout << x->data << "\t";
        InorderTreeWalk(x->right);
    }
}

void BST::PostorderTreeWalk(BST_Node *x)
```

```
{
    if (x != nullptr)
    {
        PostorderTreeWalk(x->left);
        PostorderTreeWalk(x->right);
        cout << x->data << "\t";
    }
}

BST_Node *BST::Maximum(BST_Node *x)
{
    while (x->right != nullptr)
    {
        return Maximum(x->right);
    }
    return x;
}

BST_Node *BST::Minimum(BST_Node *x)
{
    while (x->left != nullptr)
    {
        return Minimum(x->left);
    }
    return x;
}

BST_Node *BST::TreeSearch(BST_Node *x, int k)
{
    if (x == nullptr or k == x->data)
    {
        return x;
    }
    if (k < x->data)
    {
        return TreeSearch(x->left, k);
    }
    else
    {
        return TreeSearch(x->right, k);
    }
}

BST_Node *BST::IterativeTreeSearch(BST_Node *x, int k)
```

```
{
    while (x != nullptr and k != x->data)
    {
        if (k < x->data)
        {
            x = x->right;
        }
        else
        {
            x = x->right;
        }
    }
    return x;
}

BST_Node *BST::TreeSuccessor(BST_Node *x)
{
    if (x->right != nullptr)
    {
        return Minimum(x->right);
    }
    auto y = x->parent;
    while (y != nullptr and x == y->right)
    {
        x = y;
        y = y->parent;
    }
    return y;
}

BST_Node *BST::TreePredecessor(BST_Node *x)
{
    if (x->left != nullptr)
    {
        return Maximum(x->left);
    }
    auto y = x->parent;
    while (y != nullptr and x == y->left)
    {
        x = y;
        y = y->parent;
    }
    return y;
}
```

```
int BST::GetTreeSize(BST_Node *x)
{
    if (x == nullptr)
    { return 0; }
    else if (x->left == nullptr and x->right == nullptr)
    {
        return 1;
    }
    return 1 + GetTreeSize(x->left) + GetTreeSize(x->right);
}

int BST::GetLeafSize(BST_Node *x)
{
    if (x == nullptr)
    {
        return 0;
    }
    else if (x->left == nullptr and x->right == nullptr)
    {
        return 1;
    }
    return GetLeafSize(x->left) + GetLeafSize(x->right);
}

int BST::GetHeight(BST_Node *x)
{
    int left_height;
    int right_height;
    if (x == nullptr)
    {
        return 0;
    }
    else
    {
        left_height = GetHeight(x->left);
        right_height = GetHeight(x->right);
        return left_height > right_height ? ++left_height :
++right_height;
    }
}

void BST::LevelTreeWalk()
```

```
{
    queue<BST_Node *> q;
    if (_root != nullptr)
    { q.push(_root); }
    while (!q.empty())
    {
        auto temp = q.front();
        q.pop();
        if (temp->left != nullptr)
        { q.push(temp->left); }
        if (temp->right != nullptr)
        { q.push(temp->right); }
        cout << temp->data << "\t";
    }
}

void BST::LevelTreeWalk2()
{
    vector<vector<int>> v = LevelTreeWalk2(_root);
    for(int count1 = 0;count1<v.size();count1++)
    {
        cout<<"第"<<count1+1<<"层:\t";
        for(int count2 : v[count1])
        {
            cout<<count2<<"\t";
        }
        cout<<endl;
    }
}


vector<vector<int>> BST::LevelTreeWalk2(BST_Node *x)
{
    queue<BST_Node *> q;
    vector<vector<int>> v;
    if (x != nullptr)
    { q.push(x); }
    while (!q.empty())
    {
        vector<int> every_level;
        int length_of_level = q.size();
        for (int i = 0; i < length_of_level; i++)
        {
            auto temp = q.front();
```

```
            q.pop();
            every_level.push_back(temp->data);
            if (temp->left != nullptr)
            { q.push(temp->left); }
            if (temp->right != nullptr)
            { q.push(temp->right); }
        }
        v.push_back(every_level);
    }
    return v;
}

void BST::Transplant(BST_Node *u, BST_Node *v) // 用一棵以 v 为根的子树
替代一棵以 u 为根的子树
{
    if (u->parent == nullptr)
    {
        this->_root = v;  //u 是树根
    }
    else if (u == u->parent->left)
    {
        u->parent->left = v;
    }
    else
    {
        u->parent->right = v;
    }
    if (v != nullptr)
    {
        v->parent = u->parent;
    }
}

void BST::TreeDelete(BST_Node *z)
{
    if(z->left == nullptr)
    {
        Transplant(z,z->right);
    }
    else if(z->right == nullptr)
    {
        Transplant(z,z->left);
    }
    else
```

```
    {
        auto y = TreeSuccessor(z);
        if(y->parent!=z)
        {
            Transplant(y,y->right);
            y->right = z->right;
            y->right->parent = y;
        }
        Transplant(z,y);
        y->left = z->left;
        y->left->parent = y;
    }
}

void BST::Release(BST_Node *x)
{
    if(x == nullptr)
    { return;}
    else
    {
        Release(x->left);
        Release(x->right);
        free(x);
    }
}
```

main.cpp

```
#include "utility.h"
#include "BST.h"

void Print(BST Tree)
{
    cout<<"中序遍历:"<<endl;
    Tree.InorderTreeWalk();
    cout<<endl;
    cout<<endl;

    cout<<"后续遍历:"<<endl;
    Tree.PostTreeWalk();
```

```
cout<<endl;
cout<<endl;

cout<<"前序遍历:"<<endl;
Tree.PreorderTreeWalk();
cout<<endl;
cout<<endl;

cout<<"层序遍历:"<<endl;
Tree.LevelTreeWalk2();
cout<<endl;

cout<<"BST 的高度:"<<"\t"<<Tree.GetHeight()<<endl;
cout<<"BST 的节点数:"<<"\t"<<Tree.GetTreeSize()<<endl;
cout<<"BST 叶节点数:"<<"\t"<<Tree.GetLeafSize()<<endl;
cout<<endl;

cout<<"在 BST 中查找 data 为 4 的节点:"<<endl;
if(Tree.TreeSearch(4)!= nullptr){cout<<"找到"<<endl;}
else{cout<<"未找到"<<endl;}
cout<<endl;

cout<<"在 BST 中查找 data 为 12 的节点:"<<endl;
if(Tree.TreeSearch(12)!= nullptr){cout<<"找到"<<endl;}
else{cout<<"未找到"<<endl;}
cout<<endl;

cout<<"拷贝 BST"<<endl;
BST(Tree2){Tree};
cout<<endl;

cout<<"遍历拷贝之后的 BST:"<<endl;
Tree2.LevelTreeWalk2();
cout<<endl;

cout<<"在 BST 中删除 data 为 6 的节点"<<endl;
Tree.TreeDelete(Tree.TreeSearch(6));
cout<<endl;

cout<<"删除后的 BST:"<<endl;

cout<<"中序遍历:"<<endl;
Tree.InorderTreeWalk();
cout<<endl;
```

```
    cout<<endl;

    cout<<"后续遍历:"<<endl;
    Tree.PostTreeWalk();
    cout<<endl;
    cout<<endl;

    cout<<"前序遍历:"<<endl;
    Tree.PreorderTreeWalk();
    cout<<endl;
    cout<<endl;

    cout<<"层序遍历:"<<endl;
    Tree.LevelTreeWalk2();
    cout<<endl;
}

int main()
{
    auto a = new BST_Node;
    auto b = new BST_Node;
    auto c = new BST_Node;
    auto d = new BST_Node;
    auto e = new BST_Node;
    auto f = new BST_Node;
    auto g = new BST_Node;
    auto h = new BST_Node;
    auto i = new BST_Node;
    auto j = new BST_Node;
    auto k = new BST_Node;
    a->data = 15;
    b->data = 6;
    c->data = 3;
    d->data = 2;
    e->data = 4;
    f->data = 7;
    g->data = 13;
    h->data = 9;
    i->data = 18;
    j->data = 17;
    k->data = 20;
    vector<BST_Node*>v{a,b,c,d,e,f,g,h,i,j,k};
    BST(Tree){v};
    Print(Tree);
```

```
    system("pause");
    return 0;
}
```