《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

## 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。

## 概要设计

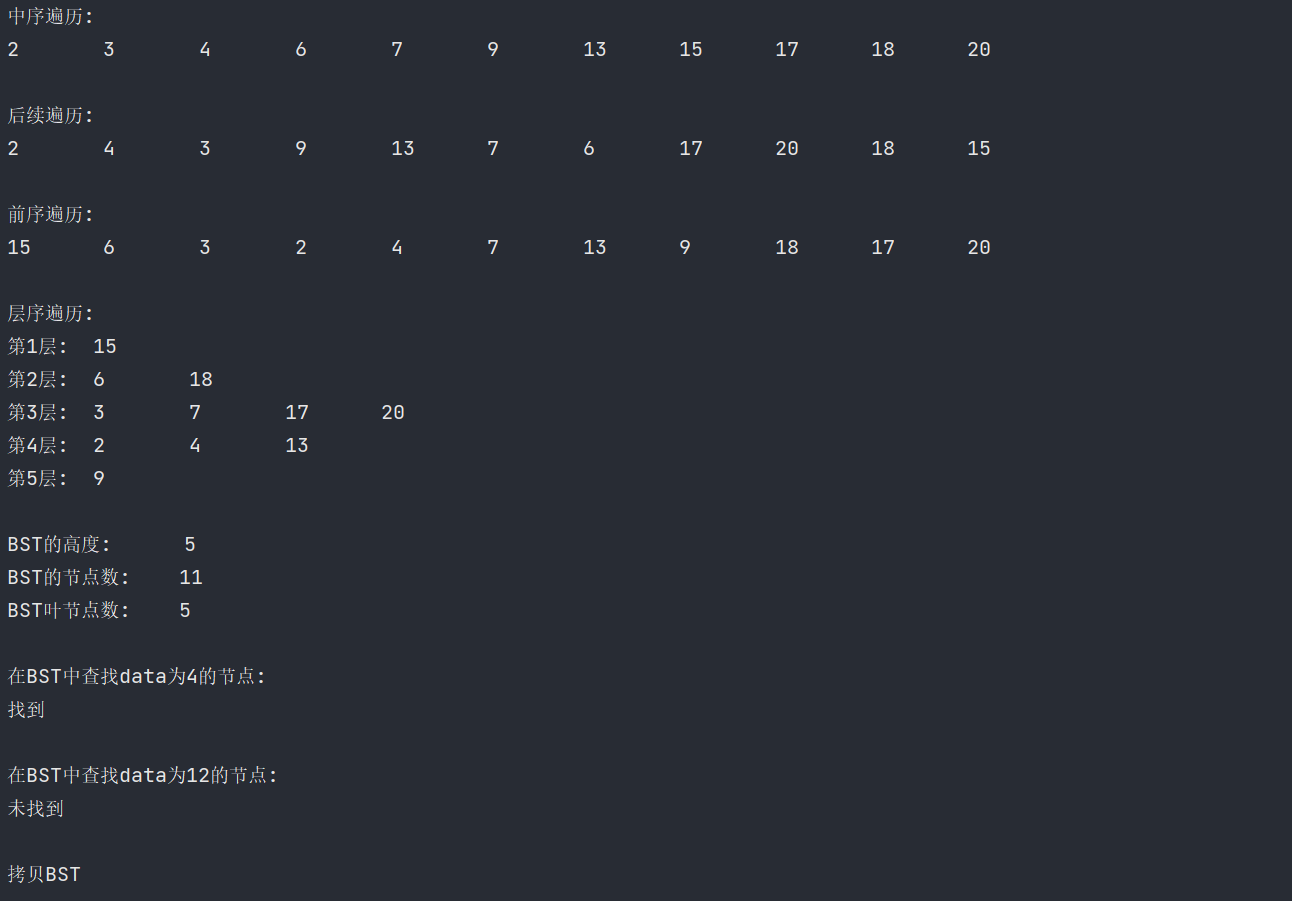
### 对实验内容的理解

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。

### 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



图片包含 图形用户界面

描述已自动生成

### （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

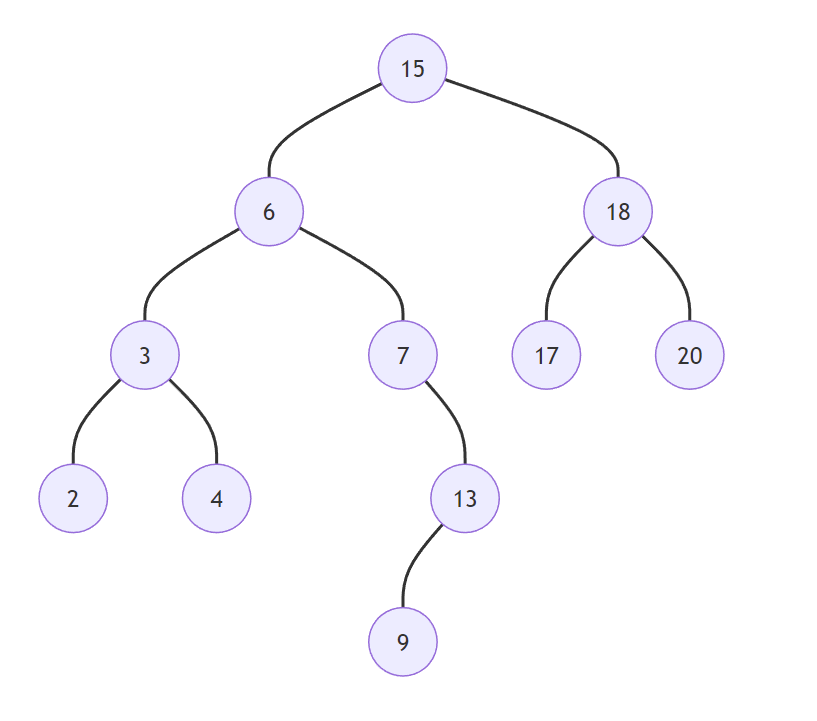
### 第一组测试 测试BST的构造以及BST的遍历

测试输入:

文本

描述已自动生成

该输入构建的BST为:



正确输出:

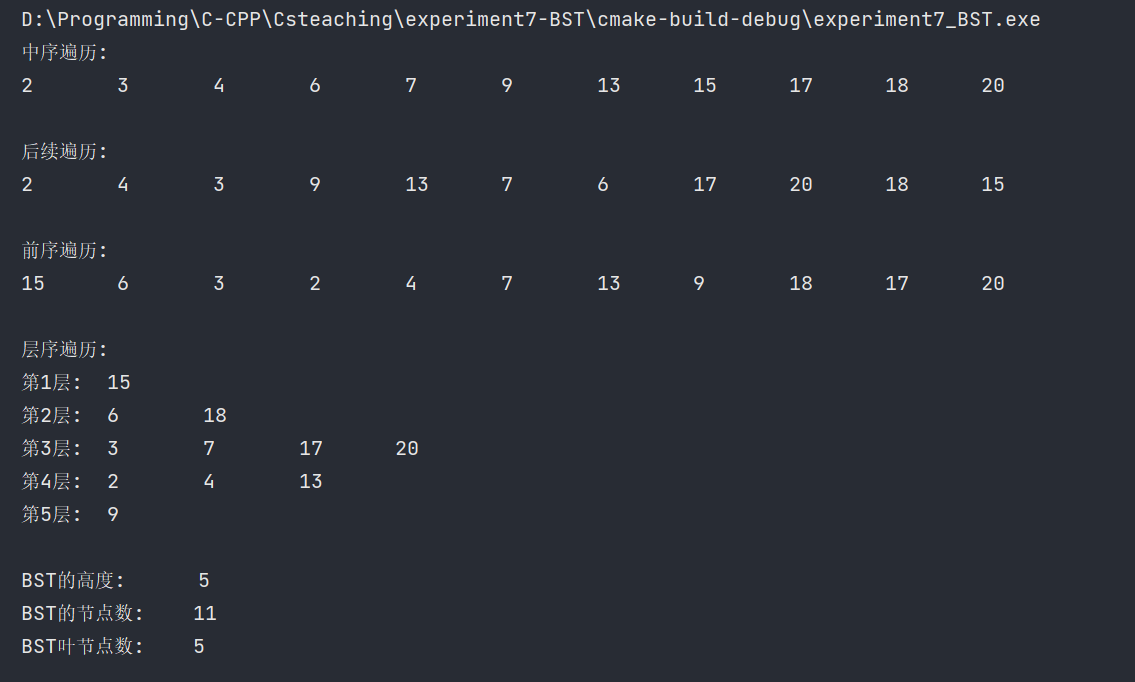
前序遍历: 15 6 3 2 4 7 13 9 18 17 20

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：

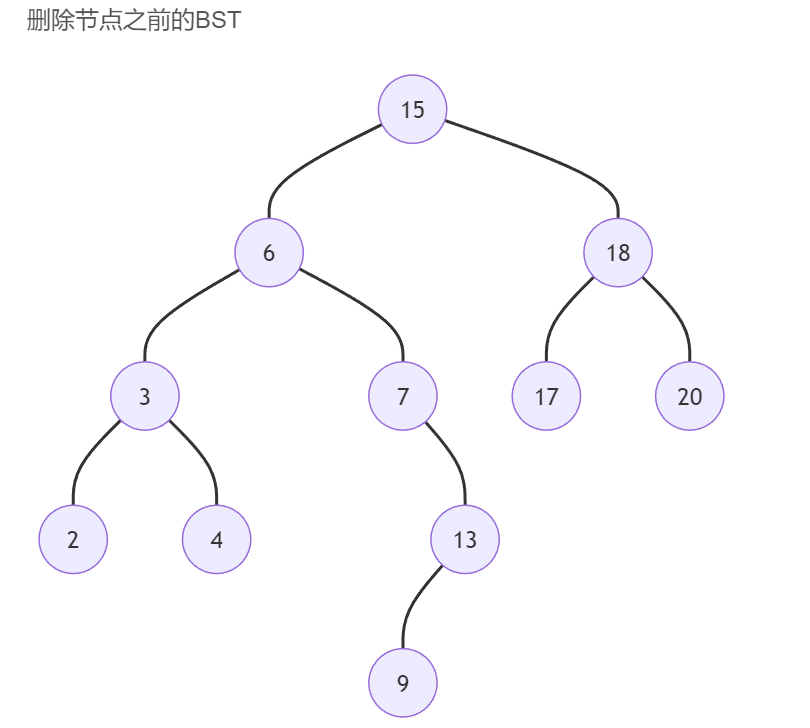


测试结果:正确

### 第二组测试 测试BST的查找以及删除

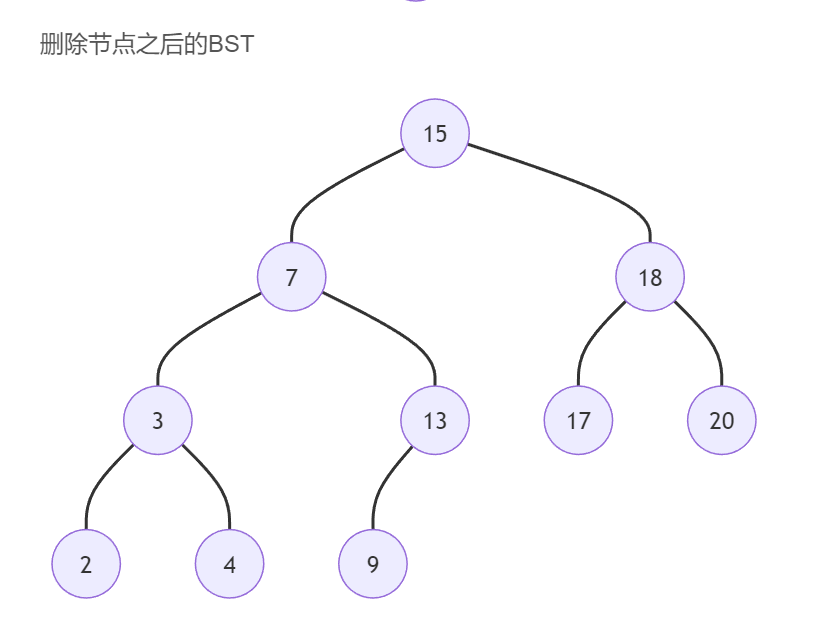
测试数据:在刚刚构建的BST中查找data为4的节点，在刚刚构建的BST中查找data为12的节点，在BST中删除data为6的节点

正确输出:

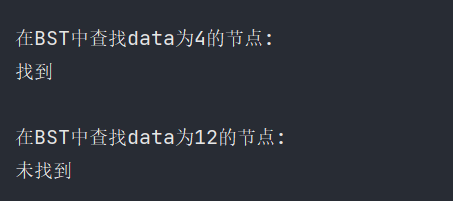


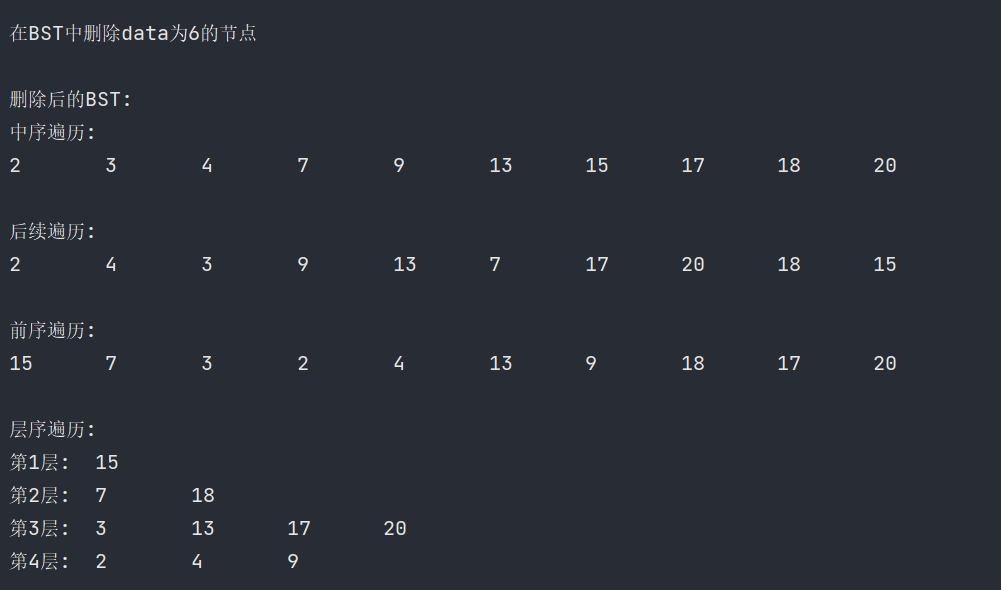
该BST中存在data为4的节点，不存在data为12的节点，因此4能够找到，但是12不能找到。

删除data为6的节点后，BST应为:



实际输出:





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。

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，其中为树的高度。在BST是随机构建的情况下，二叉树的期望高度为，因此上述所说的操作的评价运行时间为。

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

### 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

### 源代码

### utility.h

#include<iostream>  
#include<queue>  
#include<stack>  
#include<string>  
#include<vector>  
#include<cstdlib>  
using namespace std;

### BST.h

#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;  
}