Lesson03---二叉树进阶

[本节目标]

- **1. 内容安排说明**
- **2. 二叉搜索树实现**
- **3.二叉树搜索树应用分析**
- **4. 二叉树进阶面试题**

1. 内容安排说明

> 二叉树在前面C数据结构阶段已经讲过，本节取名二叉树进阶是因为：
>
> 1. **map和set特性需要**先铺垫二叉搜索树，而二叉搜索树也是一种树形结构
> 2. **二叉搜索树的特性了解，有助于更好的理解map和set的特性**
> 3. **二叉树中部分面试题稍微有点难度**，在前面讲解**大家不容易接受**，且时间长容易忘
> 4. **有些OJ题使用C语言方式实现比较麻烦，比如有些地方要返回动态开辟的二维数组，非常麻烦。**
>
> 因此本节借二叉树搜索树，对二叉树部分进行收尾总结。

2. 二叉搜索树

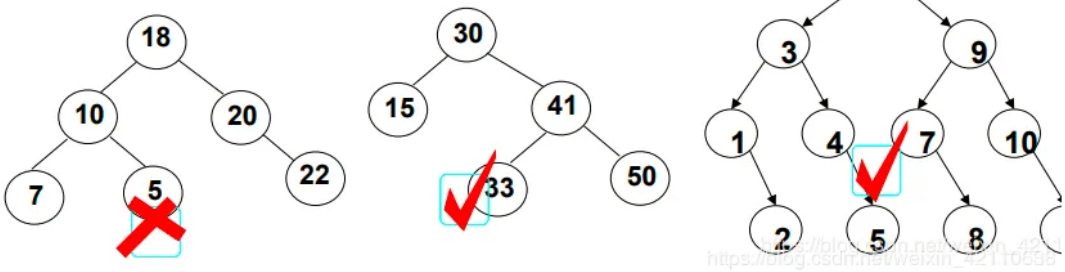
2.1 二叉搜索树概念

> **二叉搜索树又称二叉排序树，它或者是一棵空树**，或者是具有以下性质的二叉树:
>
> - **若它的左子树不为空，则左子树上所有节点的值都小于根节点的值**
> - **若它的右子树不为空，则右子树上所有节点的值都大于根节点的值**
> - **它的左右子树也分别为二叉搜索树**

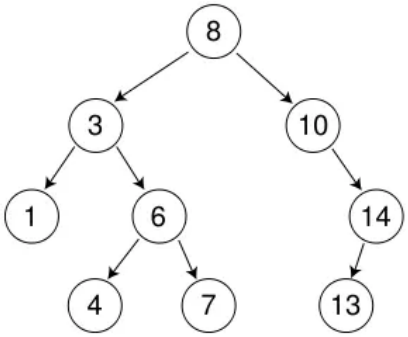
二叉搜索树（BST，Binary Search Tree），
也称二叉排序树或二叉查找树

二叉搜索树：一棵二叉树，可以为空；如果不为空，满足以下性质：

1. 非空左子树的所有键值小于其根结点的键值。
2. 非空右子树的所有键值大于其根结点的键值。
3. 左、右子树都是二叉搜索树。



2.2 二叉搜索树操作




```
int a[] = {8, 3, 1, 10, 6, 4, 7, 14, 13};
```

1. **二叉搜索树的查找**

   a、从根开始比较，查找，比根大则往右边走查找，比根小则往左边走查找。

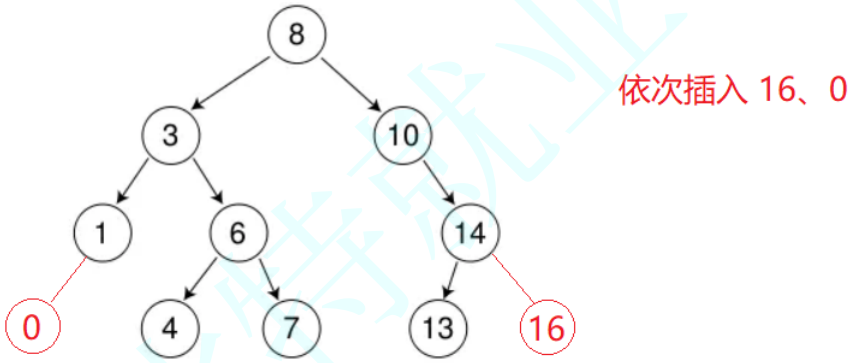
   b、最多查找高度次，走到到空，还没找到，这个值不存在。

2. **二叉搜索树的插入**

   插入的具体过程如下：

   a. **树为空，则直接新增节点，赋值给root指针**

   b. **树不空，按二叉搜索树性质查找插入位置，插入新节点**



1. **二叉搜索树的删除

首先查找元素是否在二叉搜索树中，如果不存在，则返回，否则要删除的结点可能分下面四种情况：

a. **要删除的结点无孩子结点**

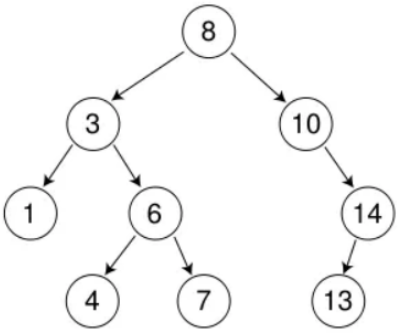
b. **要删除的结点只有左孩子结点**

c. **要删除的结点只有右孩子结点**

d. **要删除的结点有左、右孩子结点**

看起来有待删除节点有4中情况，实际情况a可以与情况b或者c合并起来，因此真正的删除过程如下：

- **情况b：删除该结点且使被删除节点的双亲结点指向被删除节点的左孩子结点--直接删除**
- **情况c：删除该结点且使被删除节点的双亲结点指向被删除结点的右孩子结点--直接删除**
- **情况d：在它的右子树中寻找中序下的第一个结点(关键码最小)，用它的值填补到被删除节点中，再来处理该结点的删除问题--替换法删除**



依次删除7、14、3、8
7和14属于直接删除的场景
3、8属于需要替换法进行删除的场景

2.3 二叉搜索树的实现

```
template<class T>
struct BSTNode
{
    BSTNode(const T& data = T())
        : _pLeft(nullptr) , _pRight(nullptr), _data(data)
    {}
    BSTNode<T>* _pLeft;
    BSTNode<T>* _pRight;
    T _data;
};

template<class T>
class BSTree
{
    typedef BSTNode<T> Node;
    typedef Node* PNode;
public:
    BSTree(): _pRoot(nullptr)
    {}

    // 同学们自己实现，与二叉树的销毁类似
    ~BSTree();
    // 根据二叉搜索树的性质查找：找到值为data的节点在二叉搜索树中的位置
    PNode Find(const T& data);
    bool Insert(const T& data)
    {
        // 如果树为空，直接插入
        if (nullptr == _pRoot)
        {
            _pRoot = new Node(data);
            return true;
        }

        // 按照二叉搜索树的性质查找data在树中的插入位置
        PNode pCur = _pRoot;
        // 记录pCur的双亲，因为新元素最终插入在pCur双亲左右孩子的位置
        PNode pParent = nullptr;
        while (pCur)
        {
            pParent = pCur;
            if (data < pCur->_data)
```

```
                pCur = pCur->_pLeft;
            else if (data > pCur->_data)
                pCur = pCur->_pRight;  // 元素已经在树中存在
            else
                return false;
        }

        // 插入元素
        pCur = new Node(data);
        if (data < pParent->_data)
            pParent->_pLeft = pCur;
        else
            pParent->_pRight = pCur;

        return true;
    }

    bool Erase(const T& data)
    {
        // 如果树为空，删除失败
        if (nullptr == _pRoot)
            return false;

        // 查找在data在树中的位置
        PNode pCur = _pRoot;
        PNode pParent = nullptr;
        while (pCur)
        {
            if (data == pCur->_data)
                break;
            else if (data < pCur->_data)
            {
                pParent = pCur;
                pCur = pCur->_pLeft;
            }
            else
            {
                pParent = pCur;
                pCur = pCur->_pRight;
            }
        }

        // data不在二叉搜索树中，无法删除
        if (nullptr == pCur)
            return false;

        // 分以下情况进行删除，同学们自己画图分析完成
        if (nullptr == pCur->_pRight)
        {
            // 当前节点只有左孩子或者左孩子为空---可直接删除
        }
        else if (nullptr == pCur->_pRight)
        {
            // 当前节点只有右孩子---可直接删除
        }
        else
        {
```

```
            // 当前节点左右孩子都存在，直接删除不好删除，可以在其子树中找一个替代结点，比如：
            // 找其左子树中的最大节点，即左子树中最右侧的节点，或者在其右子树中最小的节点，即右子树中最小的节点
            // 替代节点找到后，将替代节点中的值交给待删除节点，转换成删除替代节点
        }

        return true;
    }

// 同学们自己实现
    void InOrder();
private:
    PNode _pRoot;
};
```

2.4 二叉搜索树的应用

1. **K模型：K模型即只有key作为关键码，结构中只需要存储Key即可，关键码即为需要搜索到的值。**

   比如：**给一个单词word，判断该单词是否拼写正确**，具体方式如下：

   - 以词库中所有单词集合中的每个单词作为key，构建一棵二叉搜索树
   - 在二叉搜索树中检索该单词是否存在，存在则拼写正确，不存在则拼写错误。

2. **KV模型：每一个关键码key，都有与之对应的值Value，即<Key, Value>的键值对。**该种方式在现实生活中非常常见：

   - 比如**英汉词典就是英文与中文的对应关系**，通过英文可以快速找到与其对应的中文，英文单词与其对应的中文<word, chinese>就构成一种键值对；
   - 再比如**统计单词次数**，统计成功后，给定单词就可快速找到其出现的次数，**单词与其出现次数就是<word, count>就构成一种键值对。**

```
// 改造二叉搜索树为KV结构
template<class K, class V>
struct BSTNode
{
    BSTNode(const K& key = K(), const V& value = V())
        : _pLeft(nullptr) , _pRight(nullptr), _key(key), _Value(value)
    {}
    BSTNode<T>* _pLeft;
    BSTNode<T>* _pRight;
    K _key;
    V _value
};

template<class K, class V>
class BSTree
{
    typedef BSTNode<K, V> Node;
    typedef Node* PNode;
public:
    BSTree(): _pRoot(nullptr){}
    PNode Find(const K& key);
    bool Insert(const K& key, const V& value)
    bool Erase(const K& key)
private:
    PNode _pRoot;
```

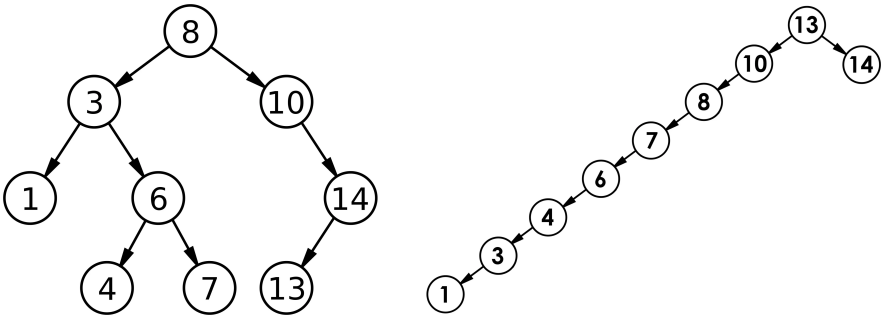
```
};

void TestBSTree3()
{
    // 输入单词，查找单词对应的中文翻译
    BSTree<string, string> dict;
    dict.Insert("string", "字符串");
    dict.Insert("tree", "树");
    dict.Insert("left", "左边、剩余");
    dict.Insert("right", "右边");
    dict.Insert("sort", "排序");
    // 插入词库中所有单词
    string str;
    while (cin>>str)
    {
        BSTreeNode<string, string>* ret = dict.Find(str);
        if (ret == nullptr)
        {
            cout << "单词拼写错误，词库中没有这个单词:" <<str <<endl;
        }
        else
        {
            cout << str << "中文翻译:" << ret->_value << endl;
        }
    }
}

void TestBSTree4()
{
    // 统计水果出现的次数
    string arr[] = { "苹果", "西瓜", "苹果", "西瓜", "苹果", "苹果", "西瓜",
"苹果", "香蕉", "苹果", "香蕉" };
    BSTree<string, int> countTree;
    for (const auto& str : arr)
    {
        // 先查找水果在不在搜索树中
        // 1、不在，说明水果第一次出现，则插入<水果, 1>
        // 2、在，则查找到的节点中水果对应的次数++
        //BSTreeNode<string, int>* ret = countTree.Find(str);
        auto ret = countTree.Find(str);
        if (ret == NULL)
        {
            countTree.Insert(str, 1);
        }
        else
        {
            ret->_value++;
        }
    }

    countTree.InOrder();
}
```

2.5 二叉搜索树的性能分析

> 插入和删除操作都必须先查找，查找效率代表了二叉搜索树中各个操作的性能。

对有n个结点的二叉搜索树，若每个元素查找的概率相等，则二叉搜索树平均查找长度是结点在二叉搜索树的深度的函数，即结点越深，则比较次数越多。

但对于同一个关键码集合，如果各关键码插入的次序不同，可能得到不同结构的二叉搜索树：



最优情况下，二叉搜索树为完全二叉树(或者接近完全二叉树)，其平均比较次数为：$log_2 N$

最差情况下，二叉搜索树退化为单支树(或者类似单支)，其平均比较次数为：$\frac{N}{2}$

问题：如果退化成单支树，二叉搜索树的性能就失去了。那能否进行改进，不论按照什么次序插入关键码，二叉搜索树的性能都能达到最优？那么我们后续章节学习的AVL树和红黑树就可以上场了。

3. 二叉树进阶面试题

这些题目更适合使用C++完成，难度也更大一些

1. 二叉树创建字符串。OJ链接
2. 二叉树的分层遍历1。OJ链接
3. 二叉树的分层遍历2。OJ链接
4. 给定一个二叉树, 找到该树中两个指定节点的最近公共祖先 。OJ链接
5. 二叉树搜索树转换成排序双向链表。OJ链接
6. 根据一棵树的前序遍历与中序遍历构造二叉树。 OJ链接
7. 根据一棵树的中序遍历与后序遍历构造二叉树。OJ链接
8. 二叉树的前序遍历，非递归迭代实现 。OJ链接
9. 二叉树中序遍历 ， 非递归迭代实现。OJ链接
10. 二叉树的后序遍历 ， 非递归迭代实现。OJ链接