湖南大學

数据结构

课程实验报告

题　　目：特殊二叉树的应用

学生姓名　杨兰馨

学生学号　201708020305

专业班级　通信 1703 班

完成日期　2019/5/18

一、 需求分析

问题描述

使用二叉查找树（BST）来实现静态查找表，要求只做查找操作。在查找表中查找时，要输出关键字比较的次数。

1. 问题分析

(1) 使用二叉链表来实现二叉查找树，设计一个节点的抽象基类，包括分别指向左右孩子的指针域和关键值域。
(2) 设计 BST 的 ADT，继承节点类，建立静态查找表。
(3) 实现静态查找表的查找功能，判断查找是否成功。
(4) 如果查找成功，计算查找次数。

2. 输入数据

exe 程序运行文件中有提示输入语句，输入分为两个部分，第一部分的输入为输入静态查找表的构建，第二部分的输入为要查找的元素：

（1） 第一次输入一个数列，要求数列中元素均为整数，元素之间用空格间隔，以-1 作为输入结束标志，例如：27 32 3 2 4 -1。输入的数字序列将按顺序插入到一棵 BST 中。

（2） 第二次输入要查找的元素值。

3. 输出数据

对于每一次查找操作，若查找成功，则输出“查找成功，查找次数为：n”；若查找不成功，则输出“查找失败，该树中没有这个元素”。

4. 测试样例设计

测试样例 1	
设计目的	构建一棵普通 BST，要查找的元素为树中根节点
样例输入（蓝色字体为提示输入语句）	下面请输入您要构建的 BST 中的元素序列： 27 32 3 2 4 -1 请输入要查找的元素值（输入-1 时停止查找）：27
说明	要查找的元素为 BST 的树的根节点，只需要查找一次即可查找成功。
样例输出	查找成功，查找次数为：1

测试样例 2	
设计目的	构建一棵普通 BST，要查找的元素为树中叶子结点

样例输入（蓝色字体为提示输入语句）	下面请输入您要构建的 BST 中的元素序列： 27 32 3 2 4 -1 请输入要查找的元素值（输入-1 时停止查找）：2
说明	叶子结点的查找次数等于树的高度
样例输出	查找成功，查找次数为：3

测试样例 3	
设计目的	构建一棵普通的 BST 树，要查找的元素不是树中的元素
样例输入（蓝色字体为提示输入语句）	下面请输入您要构建的 BST 中的元素序列： 27 32 3 2 4 -1 请输入要查找的元素值（输入-1 时停止查找）：9
说明	递归查找，直到查找到叶子结点依旧没有查找到，则说明树中没有这个结点
样例输出	查找失败，该树中没有这个元素

测试样例 4	
设计目的	树中有两个相等的元素的情况
样例输入（蓝色字体为提示输入语句）	下面请输入您要构建的 BST 中的元素序列： 27 32 32 3 2 4 -1 请输入要查找的元素值（输入-1 时停止查找）：32
说明	当树中有两个相等的元素时，而且要查找这个元素，默认要输出更小的查找次数，因为递归是从根节点开始递归查找，当节点元素值与要查找的元素值相等时递归终止，所以会输出更小的查找次数。
样例输出	查找成功，查找次数为：2

测试样例 5	
设计目的	创建一棵空树
样例输入（蓝色字体为提示输入语句）	下面请输入您要构建的 BST 中的元素序列： -1 请输入要查找的元素值（输入-1 时停止查找）：2
说明	空树的建立说明根节点就为 NULL，在该树中查找任何元素均查找失败

样例输出	查找失败，该树中没有这个元素

二、概要设计

1. 抽象数据类型

为实现上述程序的功能，创建静态查找表（静态 BST），并假定静态查找表中的数据都为整数。用二叉链表来构建一棵 BST，每一个结点有三个域来存储数据。两个指针域分别指向左右孩子，关键值域存储元素值，数据类型为 int。

抽象数据类型设计：

数据对象：一个节点（树中的每一个节点都有三个域）

数据关系：每个节点都有一个指针指向左子节点，一个指针指向右子节点。

基本操作：

1. 定义一个 insert 函数，实现 BST 树的创建。
2. 定义一个查找函数，判断查找某一个元素是否成功，同时实现计算查找次数功能。

StaticSearchTable {

数据对象： D = { key | key∈N, i = 1, 2, 3, ……, n, 1≤n≤1000 }

数据关系： R = { key | key∈BST , key's leftchild < key < key's rightchild}

基本操作：

void clear(); //清空静态查找表中的元素

void insert(int k); //向静态查找表中插入元素 k

void find(int k) ; //在静态查找表查找元素 k

}

2. 算法的基本思想

1. **建立 BST 的算法思想：**如果根节点为空，则新建一个节点，如果根节点不为空，则递归插入，如果要插入的元素值小于根节点，则将该节点的左孩子作为根节点再调用插入函数，如果要插入的元素值大于或等于根节点，则将该节点的右孩子作为根节点再调用插入函数，直到遍历到一个叶子节点，则可以新建节点并插入到其中。

2. **清空一棵 BST 树的算法思想：**采用递归的方法，删除的顺序时从叶子节点开始删，直到将根节点删除，递归终止。

3. **在静态查找表中查找元素的算法思想：**从根节点开始调用递归函数，如果 k 值大于此时的节点的元素值，则以该节点的右子节点作为根节点，继续进行查找，如果 k 值小于此时节点的元素值，则以该节点的左子节点作为根节点，继续进行查找，递归的终止条件有两个，一个是传入的根节点参数为空结点时，此时查找失败；一个是当前节点的值刚好等于要查找的元素值，此时查找成功。

3. 程序的流程

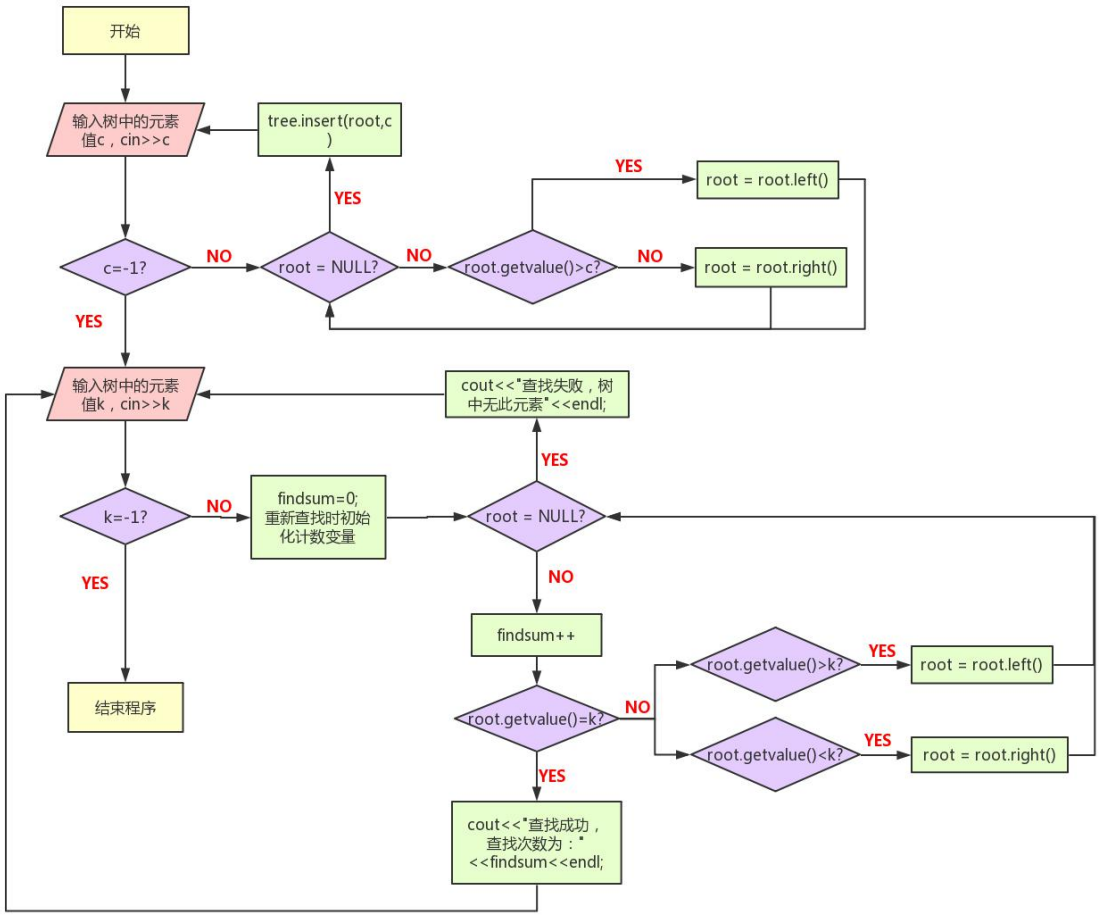
程序由三个模块组成：

（1） 输入模块：有提示语句，提示输入格式以及结束标志，需要输入要构建的树的元素序列，以及需要输入要查找的元素值。

（2） 处理模块：按顺序插入元素到 BST 中，构建出一棵 BST，并实现查找功能，同时计算查找次数。

（3） 输出模块：对查找结果进行格式化输出。

程序流程图如下：



三、详细设计

1. 物理数据类型

本实验采用二叉链表的方式存储 BST，二叉链表的节点类作为抽象基类，其中有三个域，两个指针域分别指向左子节点和右子节点，数据域存储该节点的元素值。其成员函数应该包括返回元素值，返回左子右子节点元素值，设置左子右子元素值等函数。

定义的数据类型的伪代码：

```
class Node {
private:
    int k;
    Node* lc;
    Node* rc;
public:
    Node() { lc = rc = NULL;}
    explicit Node(int k, Node* lc, Node* rc) {
        this->k = k;
        this->lc = lc;
```

```
        this->rc = rc;  }
    ~Node() {}
    // Functions to set and return the key
    virtual int& getKey() { return k;}
    virtual void setKey(const int& k) { this->k = k;}

    // Functions to set and return the children
    virtual Node* left() const { return lc;}
    virtual void setLeft(Node* lc) { this->lc = lc;}
    virtual Node* right() const { return rc;}
    virtual void setRight(Node* rc) { this->rc = rc;} };
```

2. 输入和输出的格式

输入时有提示语句，说明输入的方法与结束条件，输入存储树中的元素序列，以及要查找的元素，最后格式化查找该元素是否成功，若成功同时也要输出查找次数。

3. 算法的具体步骤

1. 实现 BST

采用递归算法，如果根节点为空，则新建一个节点，如果根节点不为空，则进行元素值的大小比较，如果要插入的元素值小于当前根节点，则将该节点的左孩子作为根节点再调用插入函数，如果要插入的元素值大于或等于当前根节点，则将该节点的右孩子作为根节点再调用插入函数。两个路径最终都直到遍历到一个叶子节点，递归结束，将新节点插入到叶子节点的左子节点处或者右子节点处。

为了起到一个良好的封装效果，所以将主要的功能函数写作私有函数，公有成员函数中再对私有函数进行调用，并将根 Root 作为参数传入到函数中。

伪代码如下：

BST 的 ADT 中的私有成员函数：

```
Node* bst::insertHelp(Node* subroot,const int& k)
{
    if(subroot==NULL) return new Node(k,NULL,NULL);
    if(k < subroot->getKey())
    {
        subroot->setLeft(insertHelp(subroot->left(),k));
    }
    else
    {
        subroot->setRight(insertHelp(subroot->right(),k));
    }
    return subroot;
}
```

BST 的 ADT 中的公有成员函数：

```
void bst::insert(const int& k)
{
    root = insertHelp(root,k);
```

```
    count++;//count 值为树中的节点数
}
```

2. 实现清空树的算法思想

同样采用递归删除，从树的叶子节点开始删除，直到将根节点删除。

伪代码如下：

```
private：
void bst::clearHelp(Node* node)
{
    if(node == NULL) return ;
    clearHelp(node->left());
    clearHelp(node->right());
    delete node;
}
public:
void bst::clear()
{
    clearHelp(root);
    root = NULL;
    count = 0;
}
```

3. 查找并输出的算法思想：

私有成员函数中实现：每次定义一个计算变量，用来计数查找次数，每每执行依次查找函数，计数变量都要加 1。传递的参数为当前的根节点，如果根节点为 NULL 值，则说明没有找到，函数返回值为-1，如果根节点不为空，则将该节点的值与查找元素进行比较，相等时是递归终止条件，此时函数返回计数变量的值，得到的是查找次数，如果不相等，则进行比较，如果当前节点的元素值小于 k 值，则将右子节点作为根节点继续查找，如果当前节点的元素值大于 k 值，则将左子节点作为根节点继续查找。

在共有成员函数中，将该树的根节点作为参数传入到函数中并调用函数，得到一个返回值，如果该返回值小于 0（即为-1），则说明，查找失败，可以直接进行输出说明查找失败，如果该返回值大于 0，则说明查找函数返回的是计数变量，即为查找成功，可以直接输出查找成功，并且查找次数即为计数变量的值。

伪代码如下：

```
private:
int bst::findHelp(Node* subroot, const int& k) const
{
    findsum++;
    if(subroot==NULL) return -1;
    else if(k < subroot->getKey())
    {
        return findHelp(subroot->left(),k);
    }
    else if(k > subroot->getKey())
    {
        return findHelp(subroot->right(),k);
```

```
        }
        else return findsum;
    }
    public:
    void bst::find(const int& k) const
    {
        findsum=0;//每次查找之前的计数变量初始化
        int sum = findHelp(root,k);
        if(sum<0)
        {
            cout<<"查找失败,该树中没有这个元素"<<endl;
        }
        else
        {
            cout<<"查找成功，查找次数为："<<sum<<endl;
        }
    }
```

4. 算法的时空分析

1) 向静态查找表中插入元素，时间复杂度 O（logn）。
2) 清空静态查找表中的元素，时间复杂度 O（n）。
3）查找静态查找表中的元素，时间复杂度 O（logn）。

四、调试分析

1. 调试方案设计

调试目的：

测试程序是否可运行，发现代码的语法错误、链接错误、逻辑错误、和运算错误，是否有不严谨之处。

测试样例：（要查找的元素为叶子节点）

下面请输入您要构建的 BST 中的元素序列：

27 32 3 2 4 -1

请输入要查找的元素值（输入-1 时停止查找）：2

调试计划：

设定好断点，在输入之前，单步执行，观察在 if 语句判断时的跳转是否正确。并 add watch 合适的变量值，观察程序执行的步骤。

设置断点：

在第一次输入命令前设置断点，之后单步执行，调用函数，跳转到对应函数中执行命令，观察对命令的执行情况（删除，插入）是否正确，及时进行修改，对代码的不完善之处进行修改。

2. 调试过程和结果，及分析

设置断点位置：

```
15      bst tree;
16      int c;
17      while(cin>>c)
18      {
```

单步进入，执行插入操作，首先输入 27

判断是否满足终止输入条件，如果不满足则执行插入操作。

```
17      while(cin>>c)
18      {
19          if(c==-1) break;
20          else
21          {
22              tree.insert(c);
23          }
```

需要调用 insert 函数，输入的 c 值作为函数的参数传入函数

```
61  void bst::insert(const int& k)
62  {
63      root = insertHelp(root,k);
64      count++;
65  }
```

进而调用函数的私有成员函数 insertHelp

```
14  Node* bst::insertHelp(Node* subroot,const int& k)
15  {
16      if(subroot==NULL) return new Node(k,NULL,NULL);//
17      if(k < subroot->getKey())
18      {
```

第一个元素插入的时候，树还是一棵空树，所以需要新建节点作为根节点，即 27 为根节点。在新建节点的时候，调用抽象基类的构造函数。

```
14      explicit Node(int k, Node* lc, Node* rc)
15      {
16          this->k = k;
17          this->lc = lc;
18          this->rc = rc;
19      }
```

插入第一个元素操作完成，然后输入第二个元素：

```
15      bst tree;
16      int c;
17      while(cin>>c)
18      {
```

输入第二个元素为 32，大于根节点，所以在插入函数中执行递归函数，并且将当前根节点的右子节点作为根节点继续进行插入：

```
14  Node* bst::insertHelp(Node* subroot,const int& k)
15  {
16      if(subroot==NULL) return new Node(k,NULL,NULL);//
17      if(k < subroot->getKey())
18      {
19          subroot->setLeft(insertHelp(subroot->left(),k));
20      }
21      else
22      {
23          subroot->setRight(insertHelp(subroot->right(),k));
24      }
25      return subroot;
```

执行递归函数，再次调用 insertHelp 函数，由于当前根节点为空，所以再次新建节点：

```
14  Node* bst::insertHelp(Node* subroot,const int& k)
15  {
16      if(subroot==NULL) return new Node(k,NULL,NULL);//
17      if(k < subroot->getKey())
18      {
```

而后将该新建节点设置为根节点的右子节点：

```
30      virtual Node* right() const { return rc;}
31      virtual void setRight(Node* rc) { this->rc = rc;}
32  };
```

元素 32 插入成功，插入到了 27 的右子节点处。

继续输入元素 3 2 4，插入操作同理，这里不再赘述，直接进入到查找阶段。

```
26      while(1)
27      {
28          cout<<"请输入要查找的元素值(输入-1时停止查找): ";
29          cin>>k;
30          if(k==-1) break;
31          else tree.find(k);
32      }
```

输入查找元素为 2，经判断不是终止查找条件-1，因此执行 find 函数，将元素值 2 作为函数参数传入进去。

```
30          if(k==-1) break;
31          else tree.find(k);
32      }
```

执行 find 函数，首先进行计数变量的初始化：

```
67  void bst::find(const int& k) const
68  {
69      findsum=0;
```

然后执行私有成员函数 findHelp，root 根节点作为参数传入进去：

```
28  int bst::findHelp(Node* subroot, const int& k) const
29  {
30      findsum++;
```

每执行一次查找函数，计数变量加一。由于要查找的变量 2 小于根节点，则进入如下分支，并且再次调用查找函数，此时将左子节点作为根节点进行传参。

```
32      else if(k < subroot->getKey())
33      {
34          return findHelp(subroot->left(),k);
35      }
```

经过有限次数的比较和调用，终于找到了与 k 相等的元素值，此时返回计数变量的值：

```
39      }
40      else return findsum;
```

判断函数返回值是否小于 0. 如果不是，则说明找到了这个元素值：

```
70      int sum = findHelp(root,k);
71      if(sum<0)
72      {
73          cout<<"查找失败,该树中没有这个元素"<<endl;
74      }
75      else
76      {
77          cout<<"查找成功，查找次数为："<<sum<<endl;
78      }
```

则有输出，查找成功，并且输出了查找次数。本次查找完成，继续执行下一次查找造作，同理本次操作。

输出结果如图：

```
请输入要查找的元素值(输入-1时停止查找)：2
查找成功，查找次数为：3
```

五、测试结果

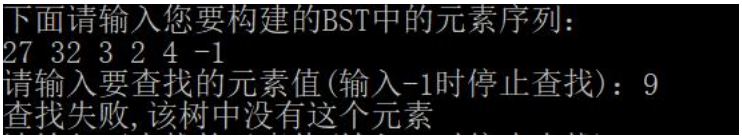
样例 1：构建一棵普通 BST，要查找的元素为树中根节点：

```
下面请输入您要构建的BST中的元素序列:
27 32 3 2 4 -1
请输入要查找的元素值(输入-1时停止查找)：27
查找成功，查找次数为：1
```

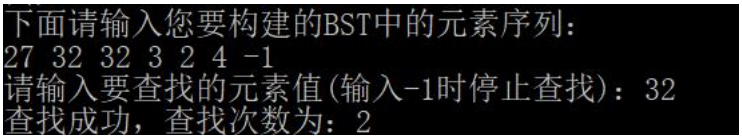
样例 2：构建一棵普通 BST，要查找的元素为树中叶子结点：

```
下面请输入您要构建的BST中的元素序列:
27 32 3 2 4 -1
请输入要查找的元素值(输入-1时停止查找)：2
查找成功，查找次数为：3
```

样例 3：构建一棵普通的 BST 树，要查找的元素不是树中的元素：



样例 4：树中有两个相等的元素的情况：



样例 5：创建一棵空树：

六、实验日志

2019 年 5 月 15 号，期中考完了，开始写实验四，在期中考试之前对 BST 着重复习过，所以看这个实验要基于 BST 实现静态查找表就显得相对简单了一些。主要是 BST 树的建立和查找算法的实现，两者都要用到递归的算法。并且重要的是 BST 的性质，左子节点的值小于根节点，右子节点的值大于或等于根节点。一个下午就将代码写好了，主要框架节点的抽象基类，BST 的 ADT，然后具体实现各个功能函数，最后写主函数验证功能。

2019 年 5 月 18 日，转眼间，周六了，写完了代码我就搁置了几天，惯性思维还以为是周日截止呢，登陆课程网站猛然发现居然今晚就截止了，我这个拖延症晚期患者再一次得到了教训，我就从中午开始写报告，一直到现在，天已经黑了，我还没写完呢。感觉这次实验不是很难，但是一到阐述算法上，用寥寥数语又难以表达清晰，所以不免多多啰嗦几句，尽我所能让实验报告变得美观一点，看着也舒服。

本次实验内容为课堂所学，这告诉我们只要将平时知识掌握牢固，实验就不会一筹莫展。继续努力吧！