目录

湖南工商大学 算法导论 课程教案

授课题目（教学章、节或主题） **课时安排: 2学时**

第九讲：二叉搜索树 **授课时间 :第九周周一第1、2节**

教学内容（包括基本内容、重点、难点）：

基本内容： （1）二叉搜索树的定义：根结点；左子树；右子树。

（2）查询操作

（3）插入和删除操作

（4）随机构建二叉搜索树

教学重点、难点：重点为二叉搜索树的构建

教学媒体的选择：本章使用大数据分析软件Jupyter教学，Jupyter集课件、Python程序运行、HTML网页制作、Pdf文档生成、Latex文档编译于一身，是算法导论课程教学的最佳选择。

板书设计：黑板分为上下两块，第一块基本定义，推导证明以及例子放在第二块。第一块 整个课堂不擦洗，以便学生随时看到算法流程图以及基本算法理论等内容。

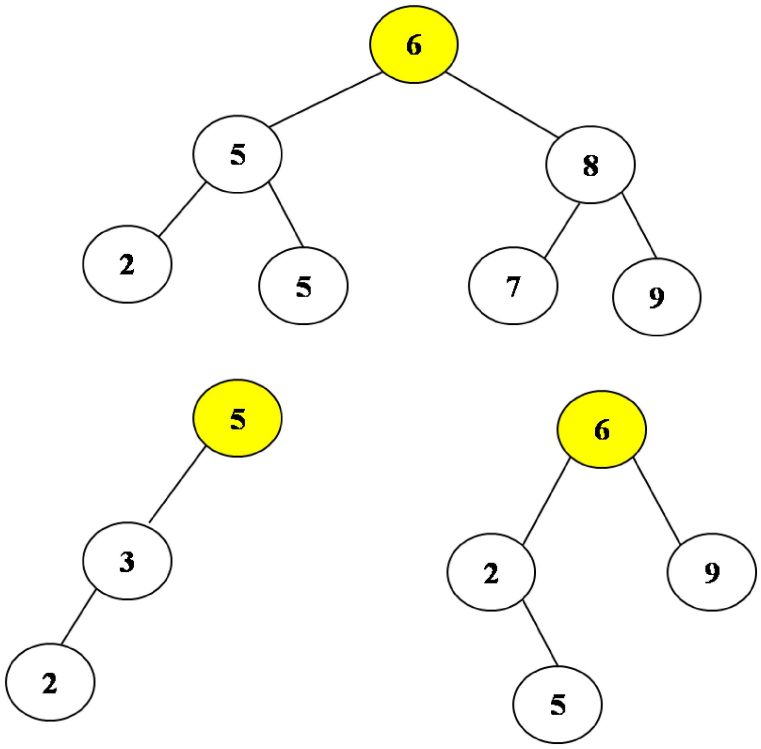
课程过程设计： （1）讲解基本算法理论； （2）举例说明； （3）程序设计与编译； （4）对本课堂进行总 结、讨论； （5）布置作业与实验报告

第九讲 二叉搜索树

1 二叉搜索树的定义

二叉查找树（Binary Search Tree），也称为二叉搜索树、有序二叉树或排序二叉树，是指一棵空树或者具有下列性质的二叉树：

（1）若任意节点的左子树不空，则左子树上所有节点的值均小于或等于它的根节点的值；
（2）若任意节点的右子树不空，则右子树上所有节点的值均大于或等于它的根节点的值；
（3）左、右子树也分别为二叉搜索树。



树中节点的定义如下：

In [1]:

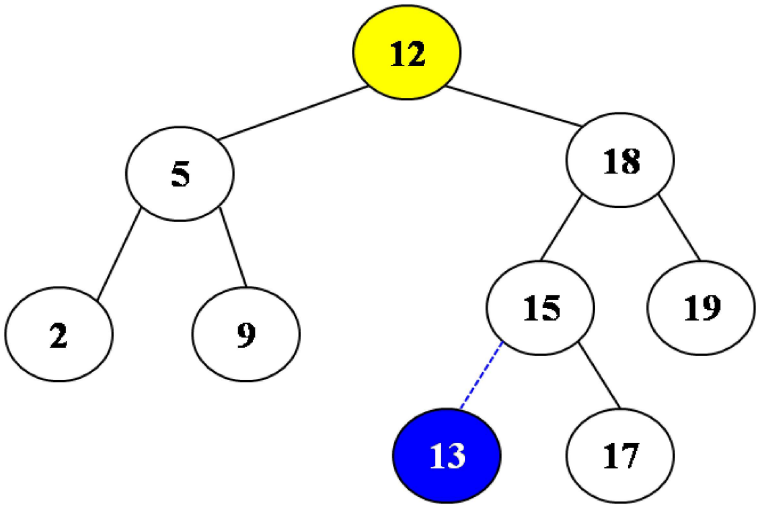
```
class Node:
    def __init__(self, data):
        self.data = data
        self.lchild = None
        self.rchild = None
```

2 二叉搜索树的相关操作

2.1 查找与插入操作

从根节点开始，若插入的值比根节点的值小，则将其插入根节点的左子树；若比根节点的值大，则将其插入根节点的右子树。该操作可使用递归进行实现。

二叉排序树是一种动态树表。其特点是：树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的结点时再进行插入。新插入的结点一定是一个新添加的叶子结点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。如下图所示：



下面的Python代码给出了具体实现：

In [5]:

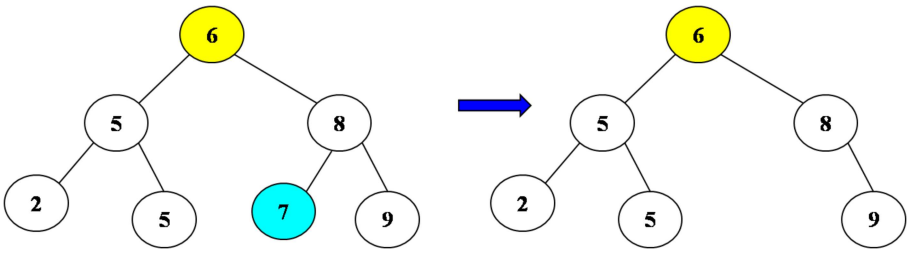
```
 # 搜索
def search(self, node, parent, data):
    if node is None:
        return False, node, parent
    if node.data == data:
        return True, node, parent
    if node.data > data:
        return self.search(node.lchild, node, data)
    else:
        return self.search(node.rchild, node, data)

# 插入
def insert(self, data):
    flag, n, p = self.search(self.root, self.root, data)
    if not flag:
        new_node = Node(data)
        if data > p.data:
            p.rchild = new_node
        else:
            p.lchild = new_node
```

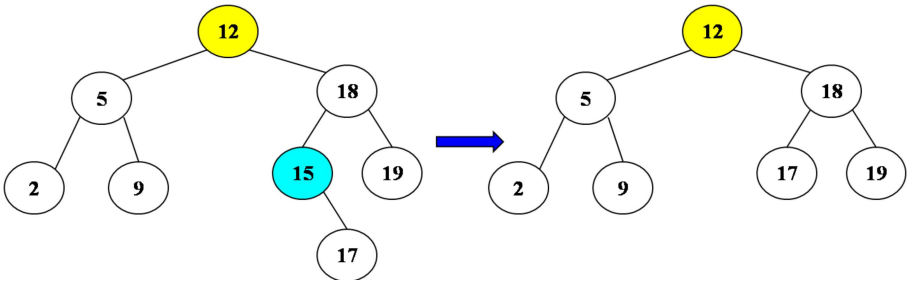
2.2 删除操作

二叉查找树的删除操作分为三种情况：

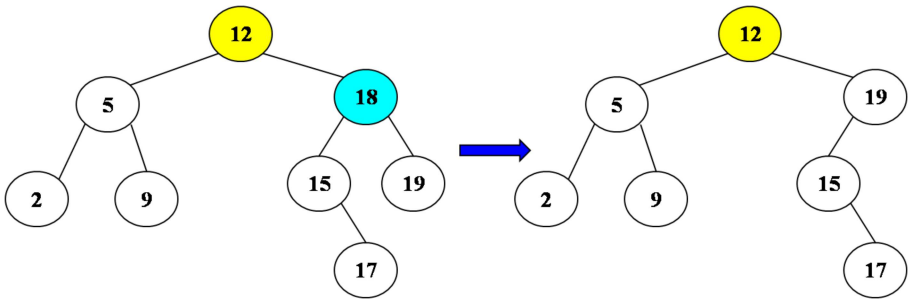
1.如果待删除的节点是叶子节点，那么可以立即被删除，如下图所示：



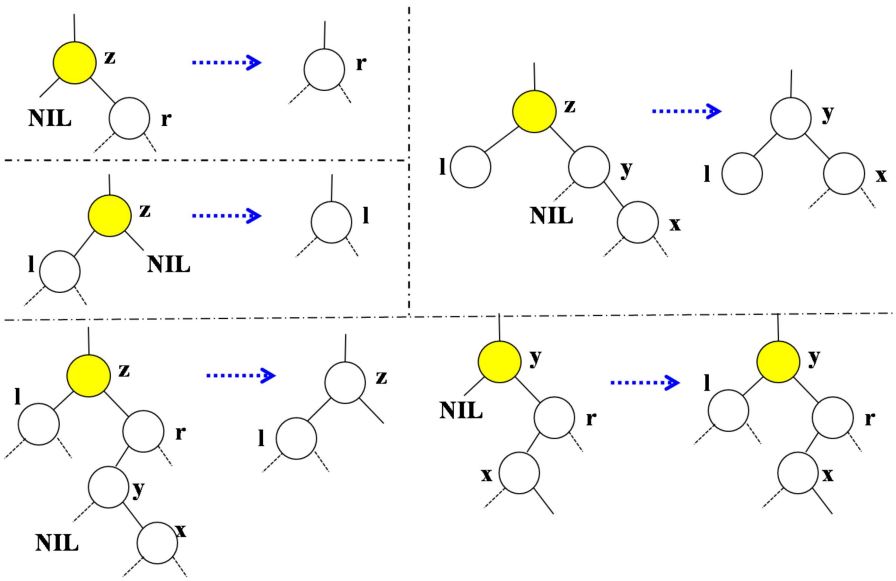
2.如果节点只有一个儿子，则将此节点parent的指针指向此节点的儿子，然后删除节点，如下图所示：



3.如果节点有两个儿子，则将其右子树的最小数据代替此节点的数据，并将其右子树的最小数据删除，如下图所示：



二叉搜索树删除操作全貌：



下面的Python代码给出了具体实现：

In [8]:

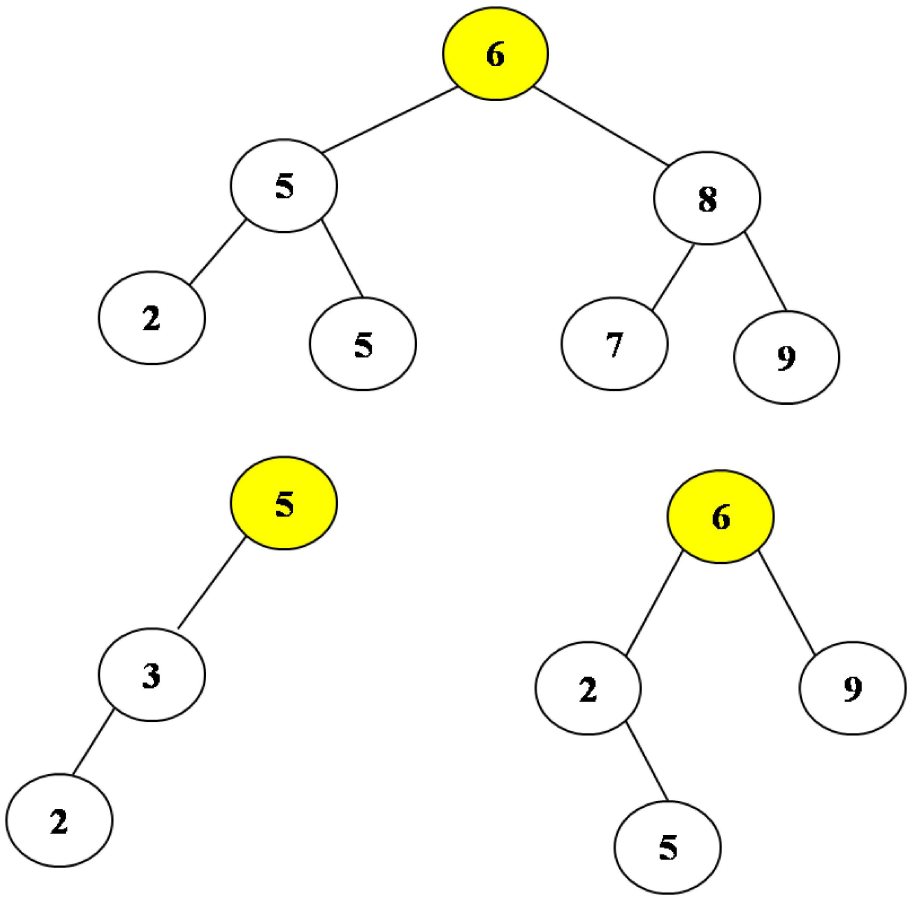
```
# 删除
def delete(self, root, data):
    flag, n, p = self.search(root, root, data)
    if flag is False:
        print("无该关键字，删除失败")
    else:
        if n.lchild is None:
            if n == p.lchild:
                p.lchild = n.rchild
            else:
                p.rchild = n.rchild
            del n
        elif n.rchild is None:
            if n == p.lchild:
                p.lchild = n.lchild
            else:
                p.rchild = n.lchild
            del n
        else:  # 左右子树均不为空
            pre = n.rchild
            if pre.lchild is None:
                n.data = pre.data
                n.rchild = pre.rchild
                del pre
            else:
                next = pre.lchild
                while next.lchild is not None:
                    pre = next
                    next = next.lchild
                n.data = next.data
                pre.lchild = next.rchild
                del next
```

2.3 遍历操作

1.先序遍历

先序遍历的访问顺序如下:

- 访问根节点
- 先序遍历左子树
- 先序遍历右子树



以上图为例，访问顺序分别为为：（一）6，5，2，5，8，7，9；（二）5，3，2；（三）6，2，5，9

下面的Python代码给出了具体实现：

In [11]:

```
# 先序遍历
def preOrderTraverse(self, node):
    if node is not None:
        print(node.data)
        self.preOrderTraverse(node.lchild)
        self.preOrderTraverse(node.rchild)
```

2.中序遍历

中序遍历的访问顺序如下:

- 中序遍历左子树
- 访问根节点
- 中序遍历右子树

以上图为例，访问顺序分别为为：（一）2，5，5，7，9，8，6；（二）2，3，5；（三）5，2，9，6

下面的Python代码给出了具体实现：

In [18]:

```
# 中序遍历
def inOrderTraverse(self, node):
    if node is not None:
        self.inOrderTraverse(node.lchild)
        print(node.data)
        self.inOrderTraverse(node.rchild)
```

3.后序遍历

后序遍历的访问顺序如下:

- 后序遍历左子树
- 后序遍历右子树
- 访问根节点

以上图为例，访问顺序分别为为：（一）2，5，5，6，7，8，9；（二）2，3，5；（三）2，5，6，9

下面的Python代码给出了具体实现：

In [20]:

```
# 后序遍历
def postOrderTraverse(self, node):
    if node is not None:
        self.postOrderTraverse(node.lchild)
        self.postOrderTraverse(node.rchild)
        print(node.data)
```

3. 完整代码

In [33]:

```
# encoding: utf-8
class Node:
    def __init__(self, data):
        self.data = data
        self.lchild = None
        self.rchild = None

class BST:
    def __init__(self, node_list):
        self.root = Node(node_list[0])
        for data in node_list[1:]:
            self.insert(data)

    # 搜索
    def search(self, node, parent, data):
        if node is None:
            return False, node, parent
        if node.data == data:
            return True, node, parent
        if node.data > data:
            return self.search(node.lchild, node, data)
        else:
            return self.search(node.rchild, node, data)

    # 插入
    def insert(self, data):
        flag, n, p = self.search(self.root, self.root, data)
        if not flag:
            new_node = Node(data)
            if data > p.data:
                p.rchild = new_node
            else:
                p.lchild = new_node

    # 删除
    def delete(self, root, data):
        flag, n, p = self.search(root, root, data)
        if flag is False:
            print("无该关键字，删除失败")
        else:
            if n.lchild is None:
                if n == p.lchild:
                    p.lchild = n.rchild
                else:
                    p.rchild = n.rchild
                del p
```

```
            elif n.rchild is None:
                if n == p.lchild:
                    p.lchild = n.lchild
                else:
                    p.rchild = n.lchild
                del p
            else:  # 左右子树均不为空
                pre = n.rchild
                if pre.lchild is None:
                    n.data = pre.data
                    n.rchild = pre.rchild
                    del pre
                else:
                    next = pre.lchild
                    while next.lchild is not None:
                        pre = next
                        next = next.lchild
                    n.data = next.data
                    pre.lchild = next.rchild
                    del p


    # 先序遍历
    def preOrderTraverse(self, node):
        if node is not None:
            print(node.data,end=" ")
            self.preOrderTraverse(node.lchild)
            self.preOrderTraverse(node.rchild)

    # 中序遍历
    def inOrderTraverse(self, node):
        if node is not None:
            self.inOrderTraverse(node.lchild)
            print(node.data,end=" ")
            self.inOrderTraverse(node.rchild)

    # 后序遍历
    def postOrderTraverse(self, node):
        if node is not None:
            self.postOrderTraverse(node.lchild)
            self.postOrderTraverse(node.rchild)
            print(node.data,end=" ")

a = [49, 38, 65, 97, 60, 76, 13, 27, 5, 1]
print("原数列为：",a)

bst = BST(a)  # 创建二叉查找树
print("中序遍历结果为：")
```

```
95  bst.inOrderTraverse(bst.root)  # 中序遍历
96
97  bst.delete(bst.root, 49)
98  print()
99  print("删除节点'49'后中序遍历结果为：")
100 bst.inOrderTraverse(bst.root)
101
```

```
原数列为： [49, 38, 65, 97, 60, 76, 13, 27, 5, 1]
中序遍历结果为：
1 5 13 27 38 49 60 65 76 97
删除节点'49'后中序遍历结果为：
1 5 13 27 38 60 65 76 97
```

引用及参考：

[1]《数据结构》李春葆著

[2] https://blog.csdn.net/u010089444/article/details/70854510?utm_source=itdadao&utm_medium=referral (https://blog.csdn.net/u010089444/article/details/70854510?utm_source=itdadao&utm_medium=referral)

课后练习

1. 写出归并排序的Python代码或 C语言代码。

2. 用归并排序代码实现对数组 $A = 3, 41, 52, 26, 38, 57, 9, 49$ 进行排序。

3. 使用图示，说明上面数组A归并排序的操作过程。

讨论、思考题、作业： 第22页 练习 2.3.1, 2.3.3

参考资料（含参考书、文献等）：算法笔记. 胡凡、曾磊，机械工业出版社，2016.

授课类型（请打√）：理论课□ 讨论课□ 实验课□ 练习课□ 其他□

教学过程设计（请打√）：复习□ 授新课□ 安排讨论□ 布置作业□

教学方式（请打√）：讲授□ 讨论□ 示教□ 指导□ 其他□

教学资源（请打√）：多媒体□ 模型□ 实物□ 挂图□ 音像□ 其他□

填表说明：1、每项页面大小可自行添减；2、教学内容与讨论、思考题、作业部分可合二为一。

In [ ]:

```
1
```