Data Structures

Ch7

查找
Searching

2023 年 11 月 23 日

学而不厌 诲人不倦

Chapter 7 查找

- 7.1 概述
- 7.2 线性表查找技术
- 7.3 树表的查找技术
- 7.4 散列表查找技术
- 7.5 各种查找方法的比较
- 7.6 扩展与提高

本章的重点就是研究**查找表的存储方法**以及在此基础上的**查找方法**。

7.3 树表的查找技术

7-3-1　二叉排序树

1. 二叉排序树定义

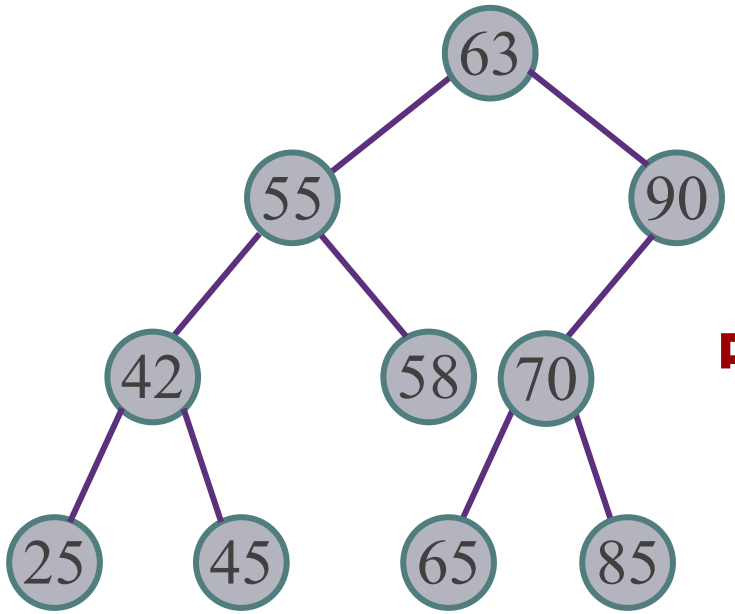
二叉搜索树 Binary Search Tree

二叉排序树或者是一棵空的二叉树，或者是具有下列性质的二叉树：

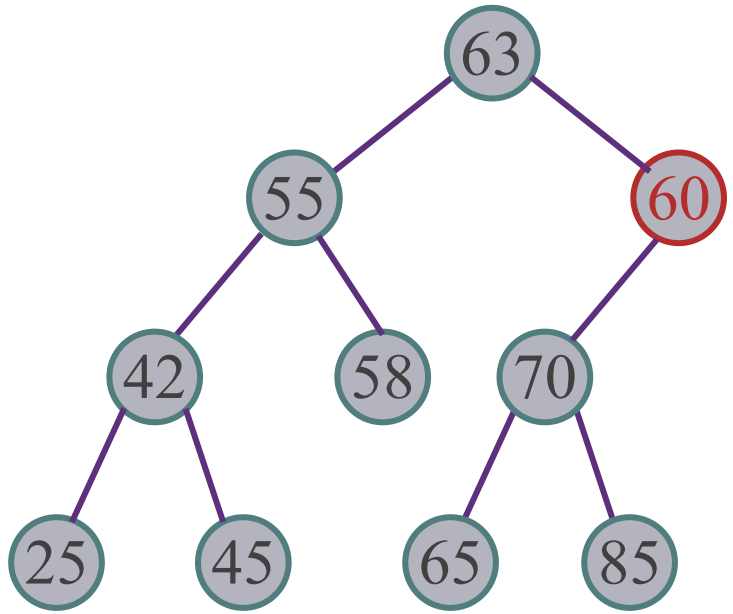
(1) 若它的左子树不空，则左子树上所有结点的值均小于根结点的值

(2) 若它的右子树不空，则右子树上所有结点的值均大于根结点的值

(3) 它的左右子树也都是二叉排序树



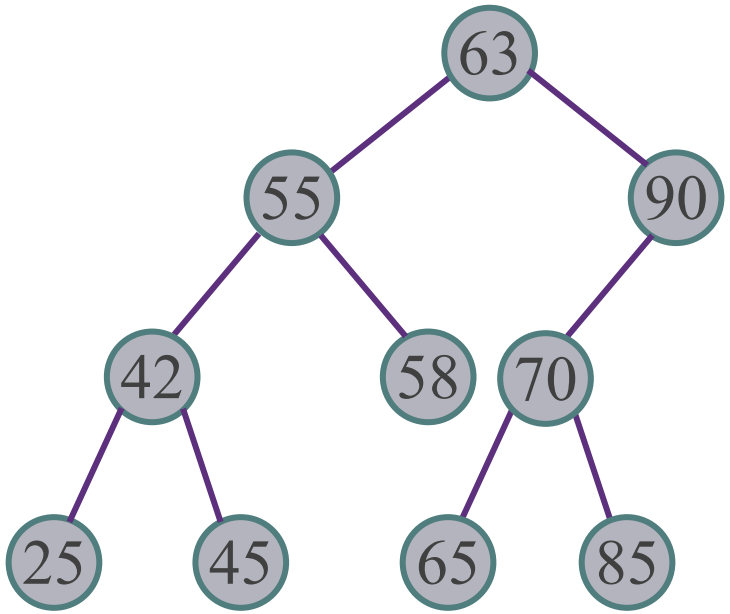
中序序列?

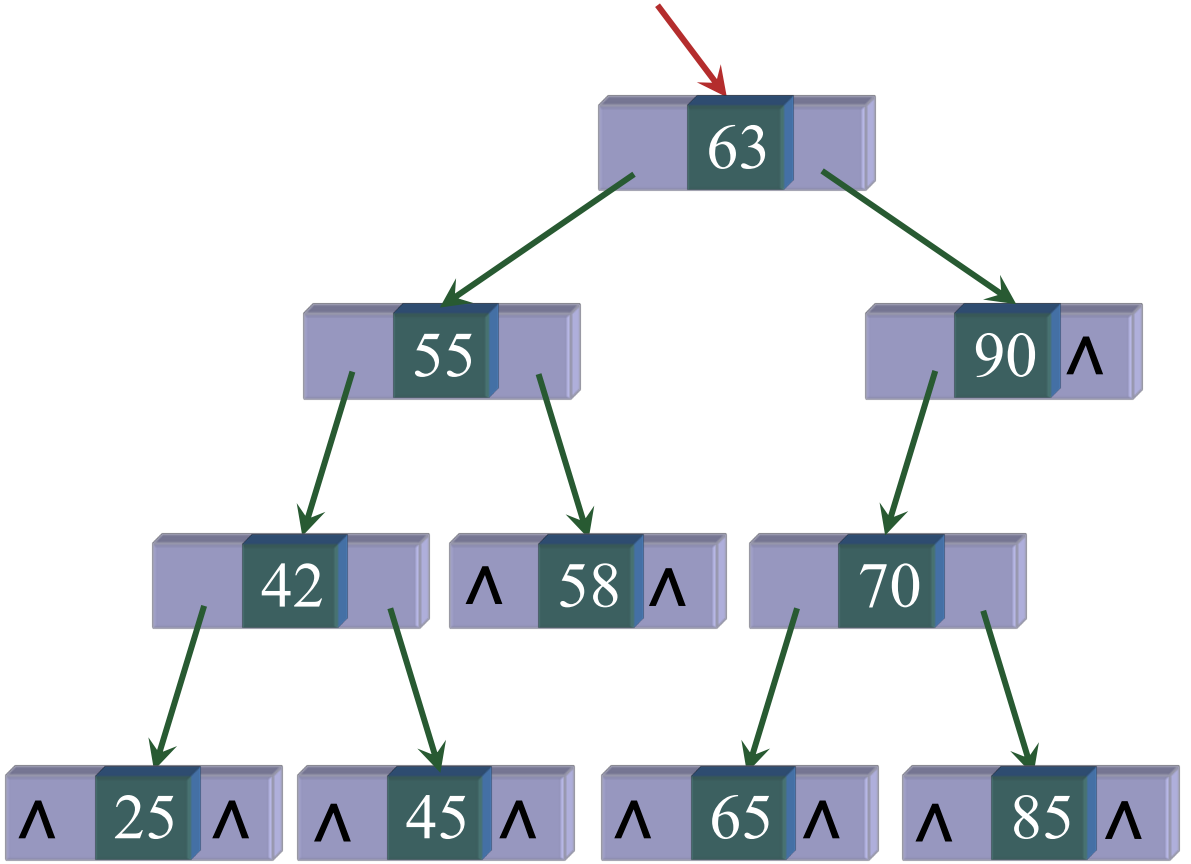


2. 二叉排序树的存储

如何存储二叉排序树？ ⇨ 二叉链表

lchild	data	rchild





2. 二叉排序树的存储

```
class BiSortTree
{
public:
    BiSortTree(int a[ ], int n);
    ~ BiSortTree( ) {Release(root);}
    BiNode<int> *InsertBST(int x) {return InsertBST(root, x);}
    void DeleteBST(BiNode<int> *p, BiNode<int> *f );
    BiNode<int> *SearchBST(int k) {return SearchBST(root, k);}
private:
    BiNode<int> *InsertBST(BiNode<int> *bt , int x);
    BiNode<int> *SearchBST(BiNode<int> *bt, int k);
    void Release(BiNode<DataType> *bt);
    BiNode<int> *root;
};
```

```
template <typename DataType>
struct BiNode
{
    DataType data;
    BiNode< DataType > *lchild, *rchild;
};
```

3. 二叉排序树的查找

二叉排序树的查找效率在于只需查找两个子树之一

在二叉排序树中查找给定值 k 的过程是

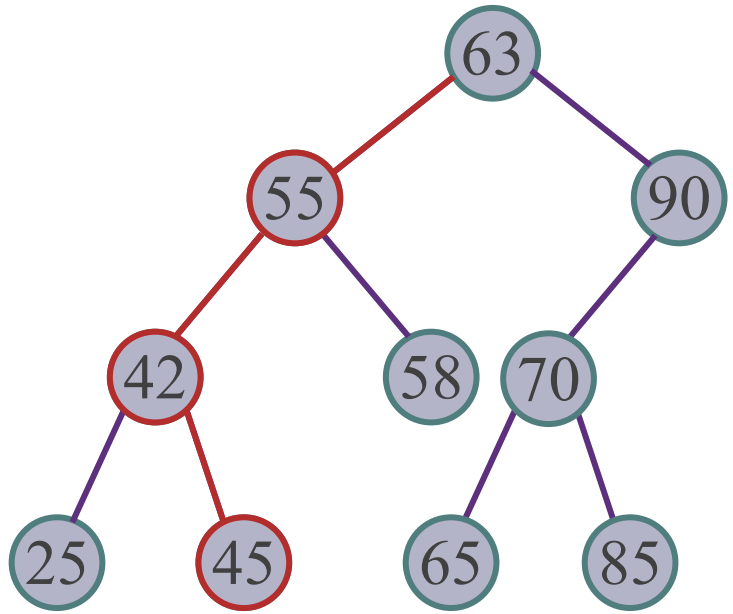
(1) 若 bt 是空树，则查找失败；

(2) 若k = bt->data，则查找成功；

(3) 若k < bt->data，则在 bt 的左子树上查找；

(4) 若k > bt->data，在 bt 的右子树上查找。

```
BiNode<int> * BiSortTree :: SearchBST(BiNode<int> *bt, int k)
{
    if (bt == nullptr) return nullptr;
    if (bt->data == k) return bt;
    else if (bt->data > k) return SearchBST(bt->lchild, k);
    else return SearchBST(bt->rchild, k);
}
```



4. 二叉排序树的插入

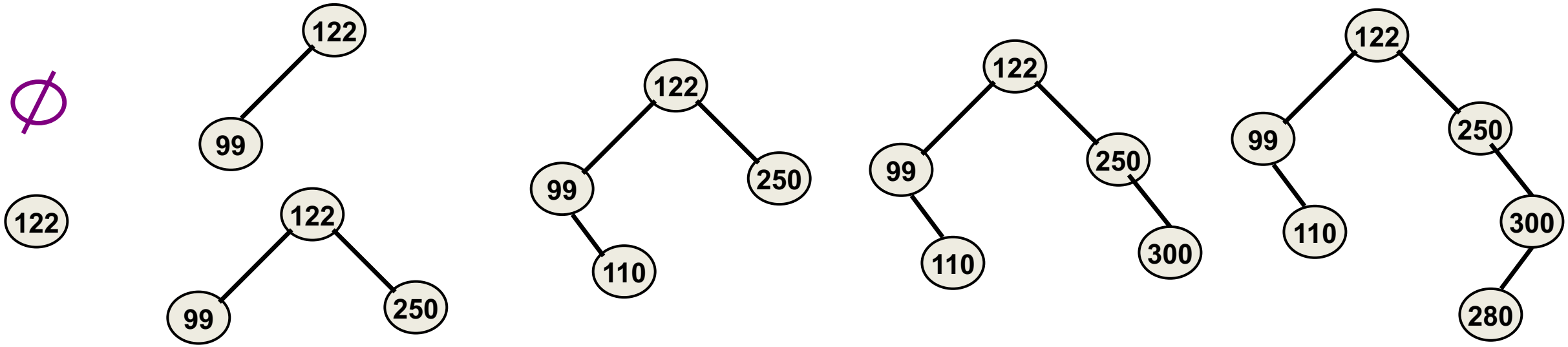
二叉排序树的生成方法

首先执行查找算法，找出被插入结点的父亲结点。

- 判断被插结点是其父亲结点的左、右儿子。将被插结点作为叶子结点插入。
- 若二叉树为空。则首先单独生成根结点。

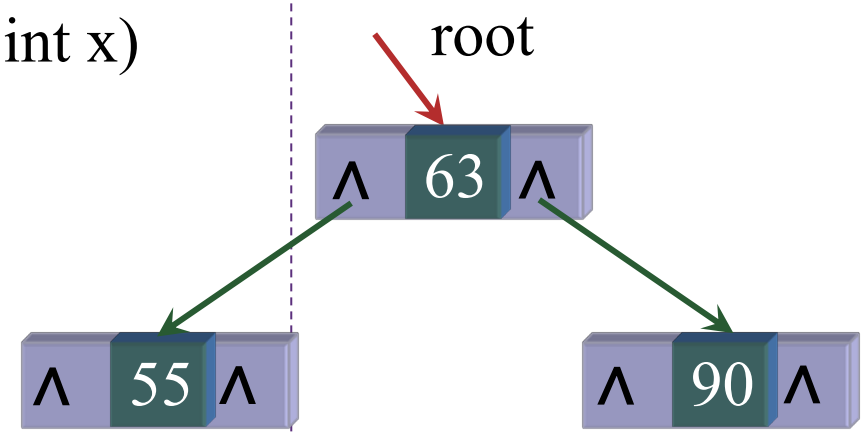
注意：新插入的结点总是叶子结点。

例题：将数的序列：122、99、250、110、300、280 作为二叉分类树的结点的关键字值，生成二叉排序树。



4. 二叉排序树的插入

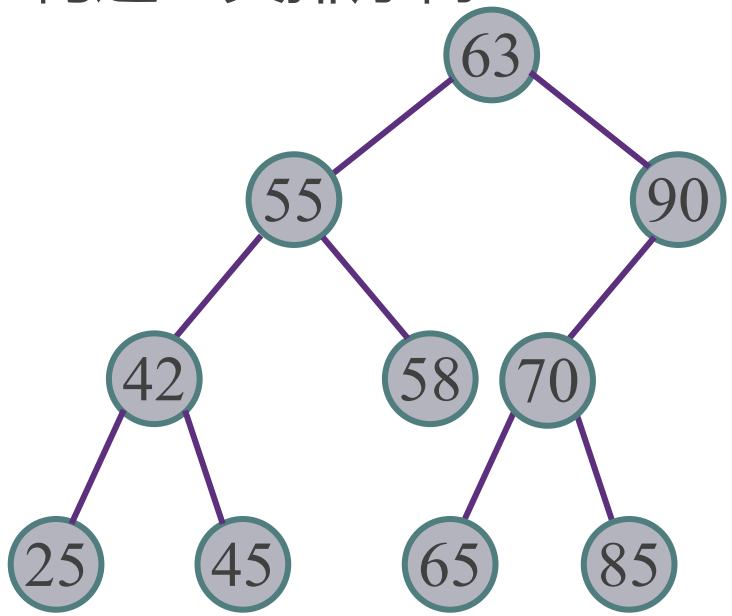
```
BiNode<int> * BiSortTree::InsertBST(BiNode<int> *bt, int x)
{
    if (bt == nullptr) {
        BiNode<int> *s = new BiNode<int>; s->data = x;
        s->lchild = s->rchild = nullptr;
        bt = s;
        return bt;
    }
    else if (bt->data > x) bt->lchild = InsertBST(bt->lchild, x);
    else bt->rchild = InsertBST(bt->rchild, x);
}
```



5. 二叉排序树的构造

例如，给定查找集合{63, 55, 42, 45, 58, 90, 70, 25, 85, 65}，构造二叉排序树

```
BiSortTree::BiSortTree(int a[ ], int n)
{
    root = nullptr;
    for (int i = 0; i < n; i++)
        root = InsertBST(root, a[i]);
}
```



(1) 每次插入的新结点都是二叉排序树上新的叶子结点;

(2) 找到插入位置后，不必移动其它结点，仅需修改某个结点的指针;

(3) 在左子树/右子树的查找过程与在整棵树上查找过程相同;

(4) 新插入的结点没有破坏原有结点之间的关系。

6. 二叉排序树的删除

和插入相反，删除在查找成功之后进行，并且要求在删除二叉排序树上某个结点之后，仍然保持二叉排序树的特性。

可分三种情况讨论：

1)、被删除的结点是叶子结点

2)、被删除的结点只有左子树或者只有右子树。

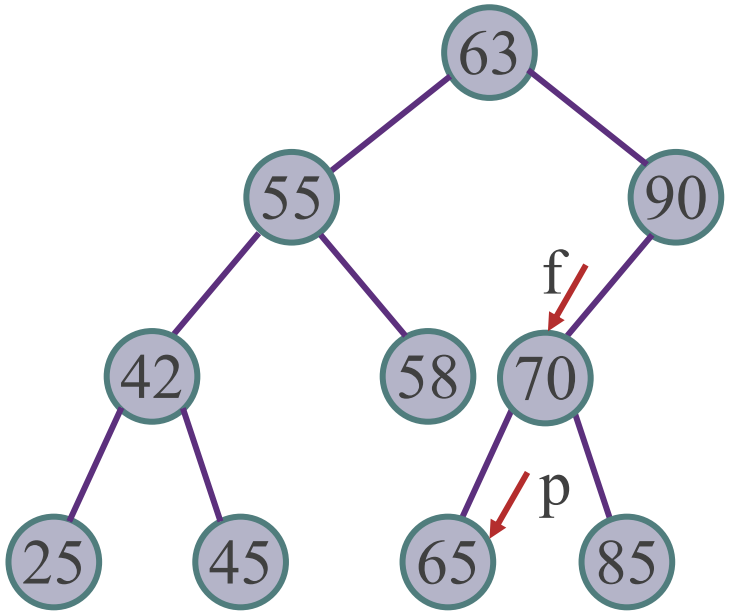
3)、被删除的结点既有左子树，也有右子树。

6. 二叉排序树的删除

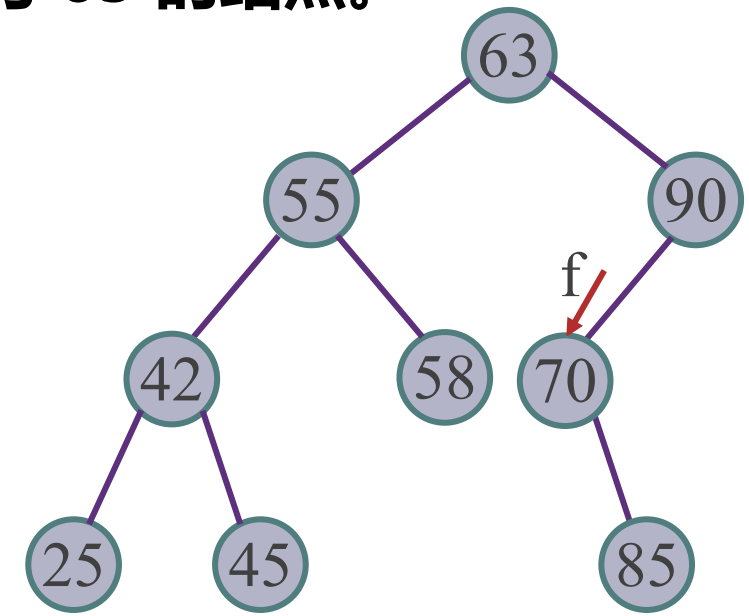
1) 被删除的结点是叶子

•直接删除，更改它的父亲结点的相应指针为空。

如：删除数据为 65 的结点。



f->lchild = nullptr;



设待删除结点为p，其双亲结点为f，且 p 是 f 的左/右孩子

6. 二叉排序树的删除

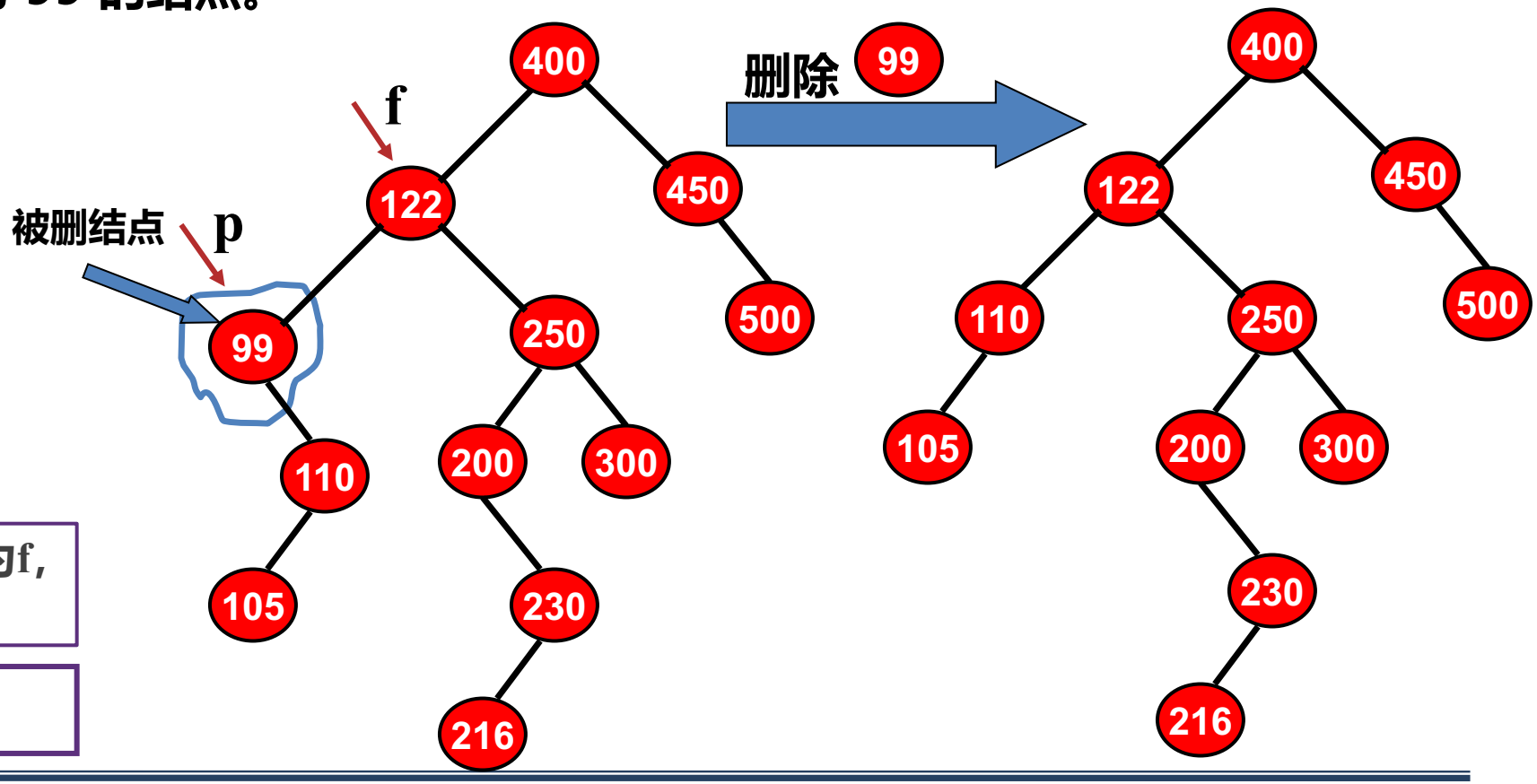
2) 被删除的结点只有左子树或者只有右子树。

•如下图所示，删除数据为 99 的结点。

结论：

•将双亲结点中相应指针域指向被删除结点的左子树（或右子树）

• 释放被删结点的空间。

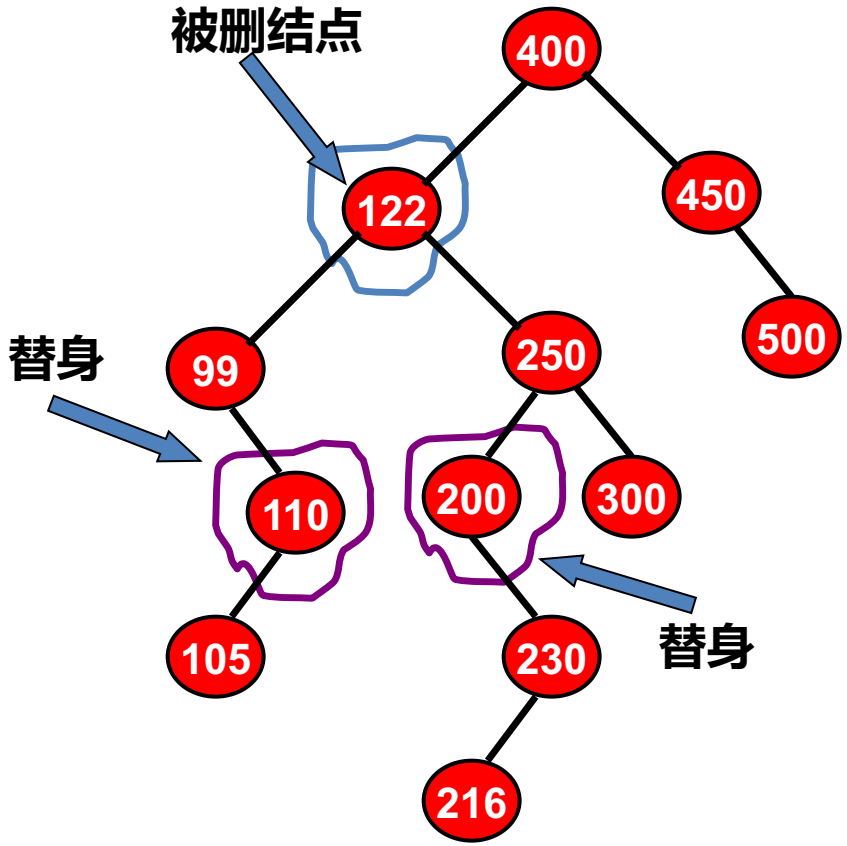


设待删除结点为p，其双亲结点为f，且 p 是 f 的左/右孩子

```
f->lchild = p->rchild;
```

6. 二叉排序树的删除

3) 被删除的结点既有左子树，也有右子树。



结论：1. 先将替身的数据复制到被删结点

2. **删除替身结点**，并释放替身结点的空间。

谁是替身?

左子树中最大的结点(被删结点的左子树中的最右的结点，其右儿子指针为空)

或

右子树中最小的结点(被删结点的右子树中的最左的结点，其左儿子指针为空)

注意：

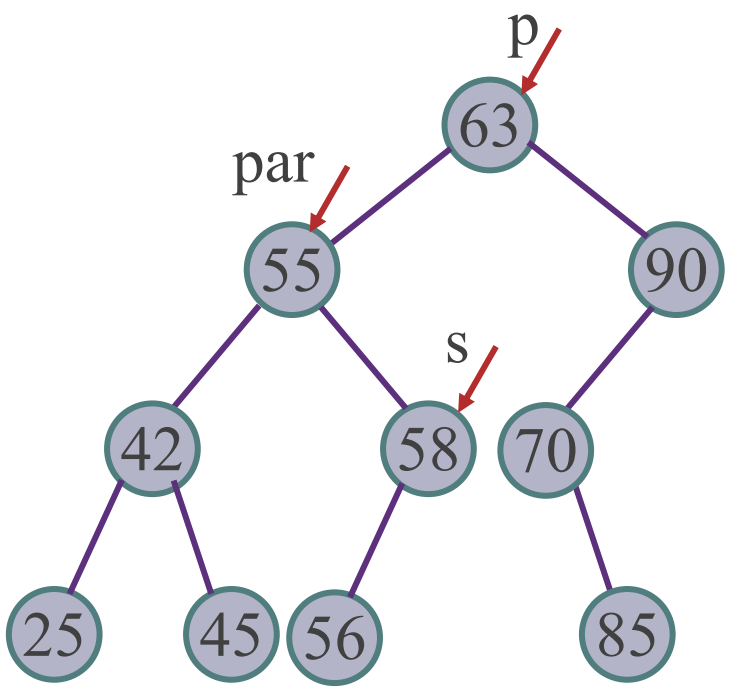
结点 110 右儿子必为空

结点 200 左儿子必为空

要点：维持二叉分类树的特性不变。**在中序遍历中紧靠着被删结点的结点才有资格作为“替身”**。

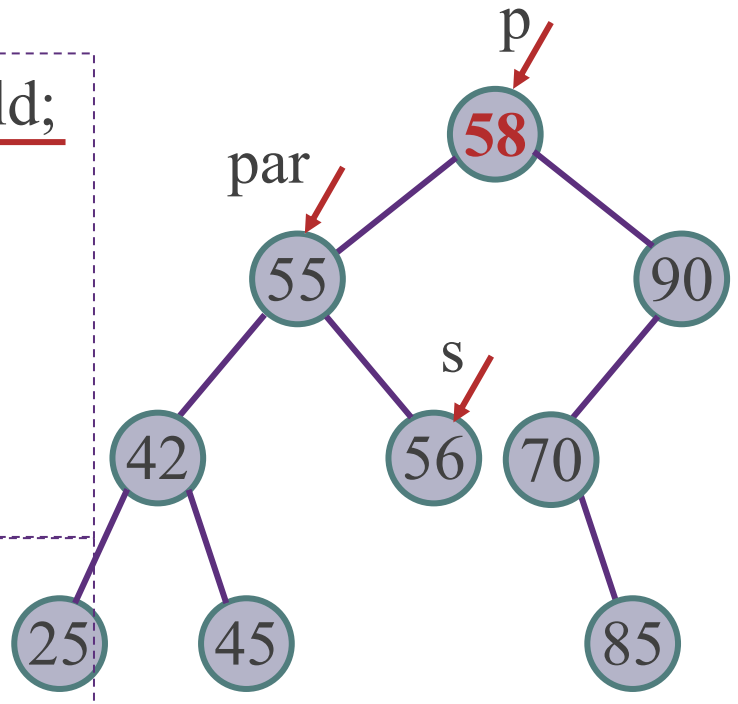
6. 二叉排序树的删除

3) 被删除的结点既有左子树，也有右子树。




```
BiNode *par = p, *s = p->lchild;
while (s->rchild != nullptr)
{
    par = s;
    s = s->rchild;
}
```

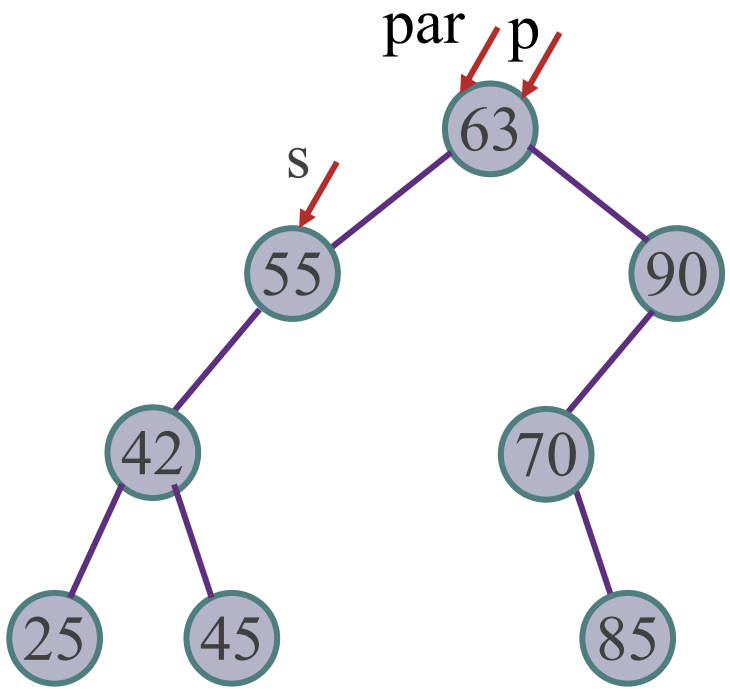
```
p->data = s->data;
par->rchild = s->lchild;
```



操作：以其**左子树中的最大值**结点替换之，然后再删除该结点

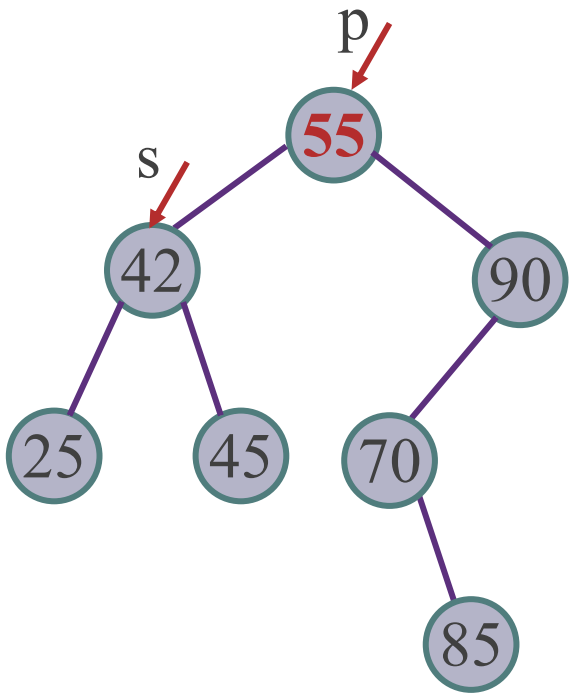
6. 二叉排序树的删除

3) 被删除的结点既有左子树，也有右子树。



特殊情况：左子树中的最大值结点是被删结点的孩子

```
if (p == par)
    par->lchild = s->lchild;
```



操作：以其**左子树中的最大值**结点替换之，然后再删除该结点

6. 二叉排序树的删除

```
template <typename DataType>
void BiSortTree::DeleteBST(BiNode<int> *p, BiNode<int> *f )
{
    if ((p->lchild == nullptr) && (p->rchild == nullptr)) {     //p为叶子
        f->lchild = NULL; delete p; return;
    }
    if (p->rchild == nullptr) {                 //p只有左子树
        f->lchild = p->lchild; delete p; return;
    }
    if (p->lchild == nullptr) {                 //p只有右子树
        f->lchild = p->rchild; delete p; return;
    }
```

6. 二叉排序树的删除

/*p的左右子树均不空*/

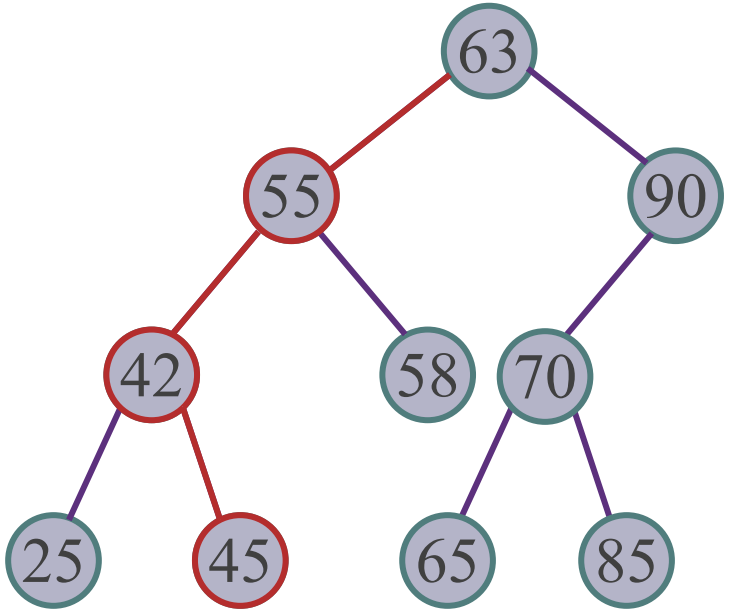
```
/*查找左子树的最右下结点*/
BiNode<int> *par = p, *s = p->lchild;
while (s->rchild != nullptr)
{
    par = s;
    s = s->rchild;
}
p->data = s->data;
if (par == p) par->lchild = s->lchild;
else par->rchild = s->lchild;
delete s;
```

```
/*查找右子树的最左下结点*/
BiNode<int> *par = p, *s = p->rchild;
while (s->lchild != nullptr)
{
    par = s;
    s = s->lchild;
}
p->data = s->data;
if (par == p) par->rchild = s->rchild;
else par->lchild = s->rchild;
delete s;
```

7. 二叉排序树的性能

二叉排序树的查找性能取决于什么?

比较次数不超过树的深度



在二叉排序树中执行插入和删除操作

⇩

查找插入和删除的位置

⇩

修改相应指针

⇩

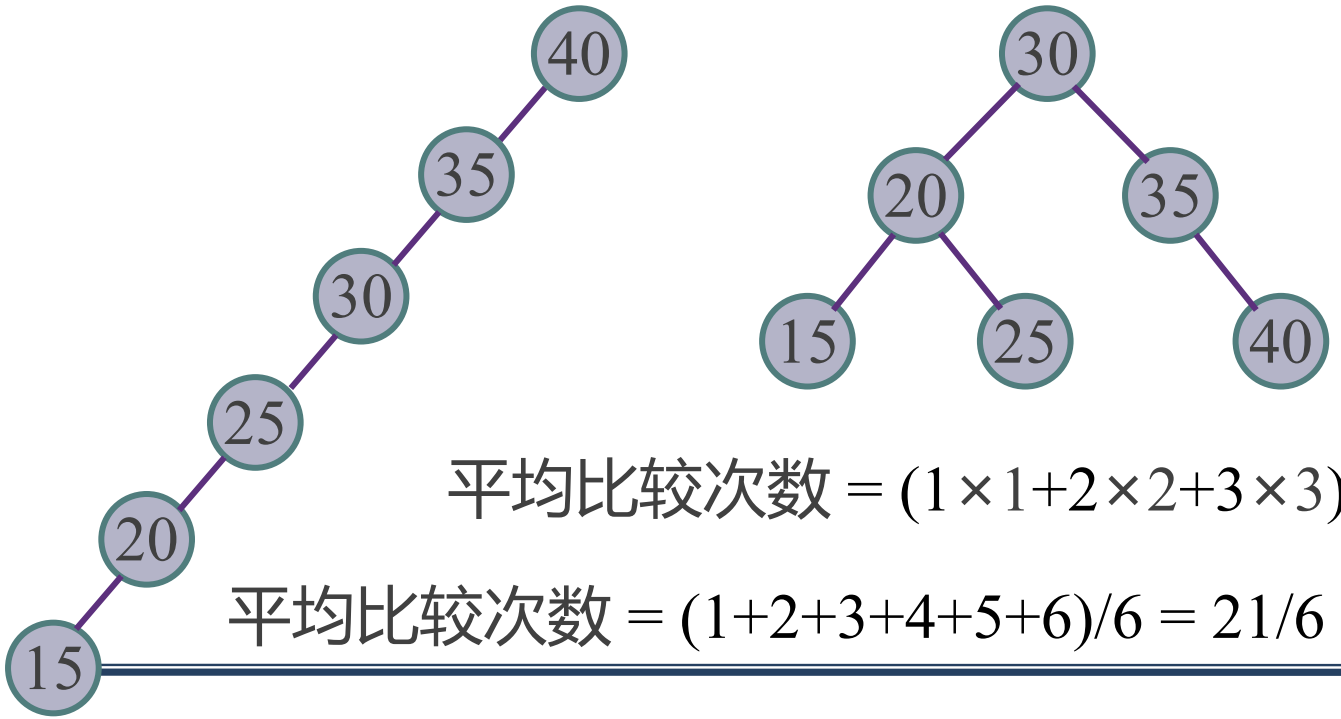
插入、删除、查找的时间复杂度相同

7. 二叉排序树的性能

二叉排序树的深度是多少？取决于什么？ 查找集合的初始排列

例如，给定查找集合{40, 35, 30, 25, 20, 15}，构造的二叉排序树深度为 n

例如，给定查找集合{15, 20, 25, 30, 35, 40}，构造的二叉排序树深度为 $\lfloor \log_2 n \rfloor + 1$



- 最坏情况：退化为线性查找
- 最好情况：相当于折半查找
- 平均情况：$O(n) \sim O(\log_2 n)$

平均比较次数 = (1×1+2×2+3×3)/6 = 14/6

平均比较次数 = (1+2+3+4+5+6)/6 = 21/6

小结

1. 掌握二叉排序树的构建、查找、插入、删除原理
2. 掌握二叉排序树的实现方法和性能分析方法
3. 掌握平衡二叉树的相关概念和不同类型平衡调整方法
4. 了解平衡二叉树的性能分析方法
5. 掌握B树的定义和查找方法
6. 理解B树的插入和删除方法

作业

1．设查找的关键字序列为(35,26,6,96,75,12,46,58,32)，请构造出对应的二叉排序树和平衡二叉树。

实验安排

实验八 查找算法的实现与应用

一、实验目的

1. 掌握二叉排序树的逻辑结构和存储结构
2. 掌握二叉排序树**构建**原理及实现方法
3. 掌握二叉排序树**查找并插入结点**的原理及实现方法
4. 掌握二叉排序树**查找并删除结点**的原理及实现方法
5. 用C++语言实现相关算法，并上机调试。

二、实验内容

1. 建立二叉排序树类。
2. 实现二叉排序树的建立、查找。
3. 实现二叉排序树的插入、删除。
4. 给出测试过程和测试结果。

实验时间：　第16周周四晚
22网安：18:30-20:10　22物联网：20:10-21:50
实验地点：软件基础实验室301（老干部处）

实验报告要求：测试数据不低于10个，每插入一个结点，绘制树的形状。

12月14日前提交预习报告，12月24日前提交正式报告。

Thank You !

Q & A