二叉查找树的删除

1、算法实现

Remove方法

```
template <class Record>
Error_code Search_tree<Record> :: remove(const Record
    &target)
/* Post: If a Record with a key matching(符合） that of target
    belongs to the Search_tree a code of success is returned and
    the corresponding node is removed from the tree. Otherwise,
    a code of not_present is returned.
    Uses: Function search_and_destroy */
{
      return search_and_destroy(root, target);
}
```

Search_and_destroy辅助递归函数

```
template <class Record>
Error_code Search_tree<Record> ::
   search_and_destroy(
Binary_node<Record>* &sub_root, const Record
   &target) {
if (sub_root == NULL )
        return not_present;
if ( sub_root->data == target)
        return remove_root(sub_root);
else if (target < sub_root->data)
        return search_and_destroy(sub_root->left, target);
else
        return search_and_destroy(sub_root->right, target);
}
```

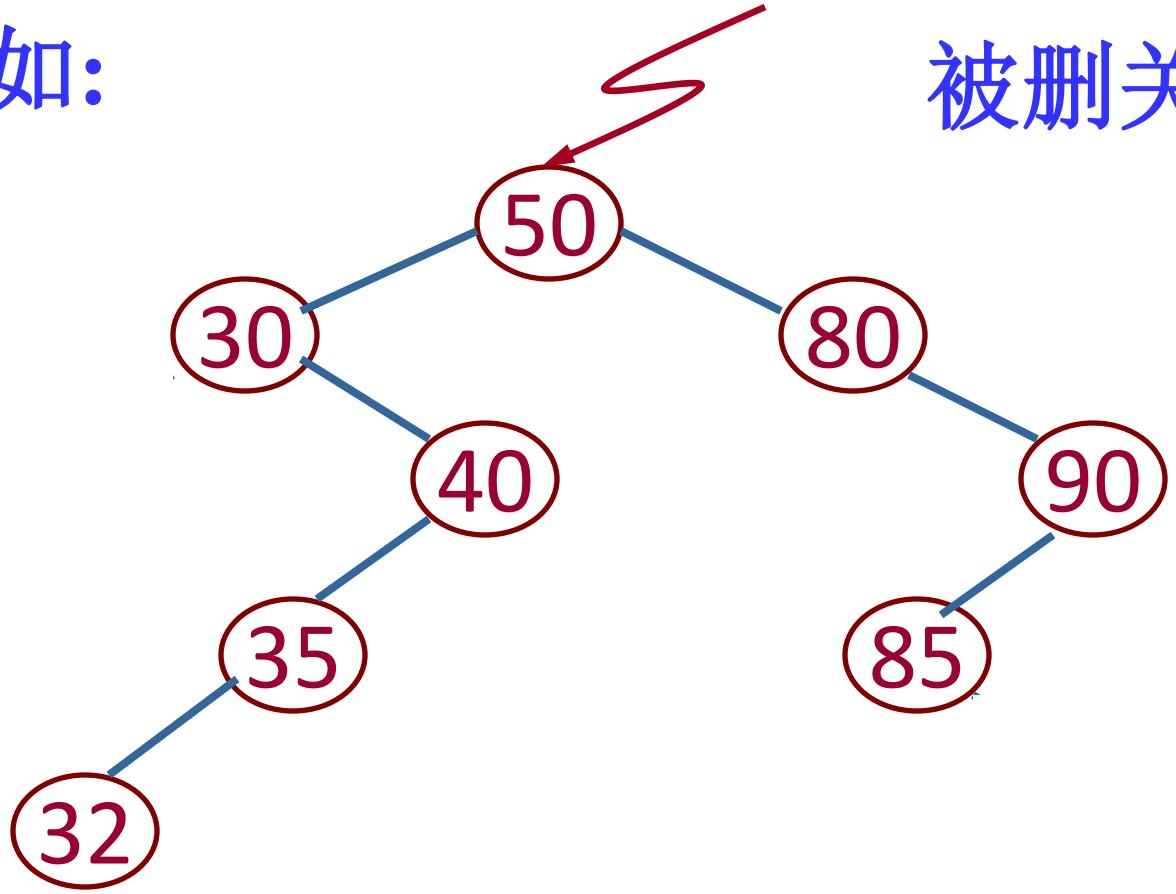
删除指定结点

- 方法分析

（a）被删除的结点是叶子结点

例如：

被删关键字 = 88

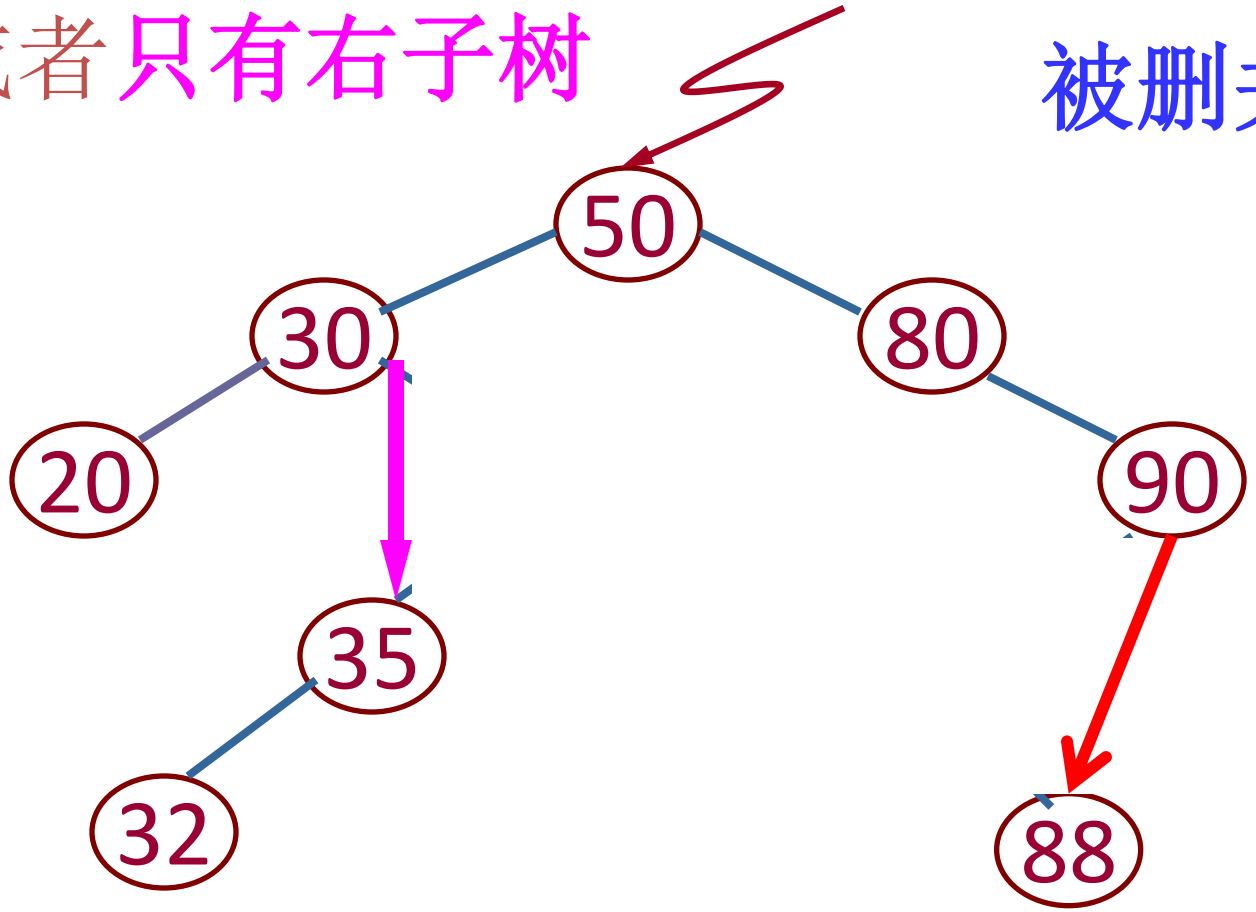


其双亲结点中相应指针域的值改为“空”

（b）被删除的结点只有左子树

或者只有右子树

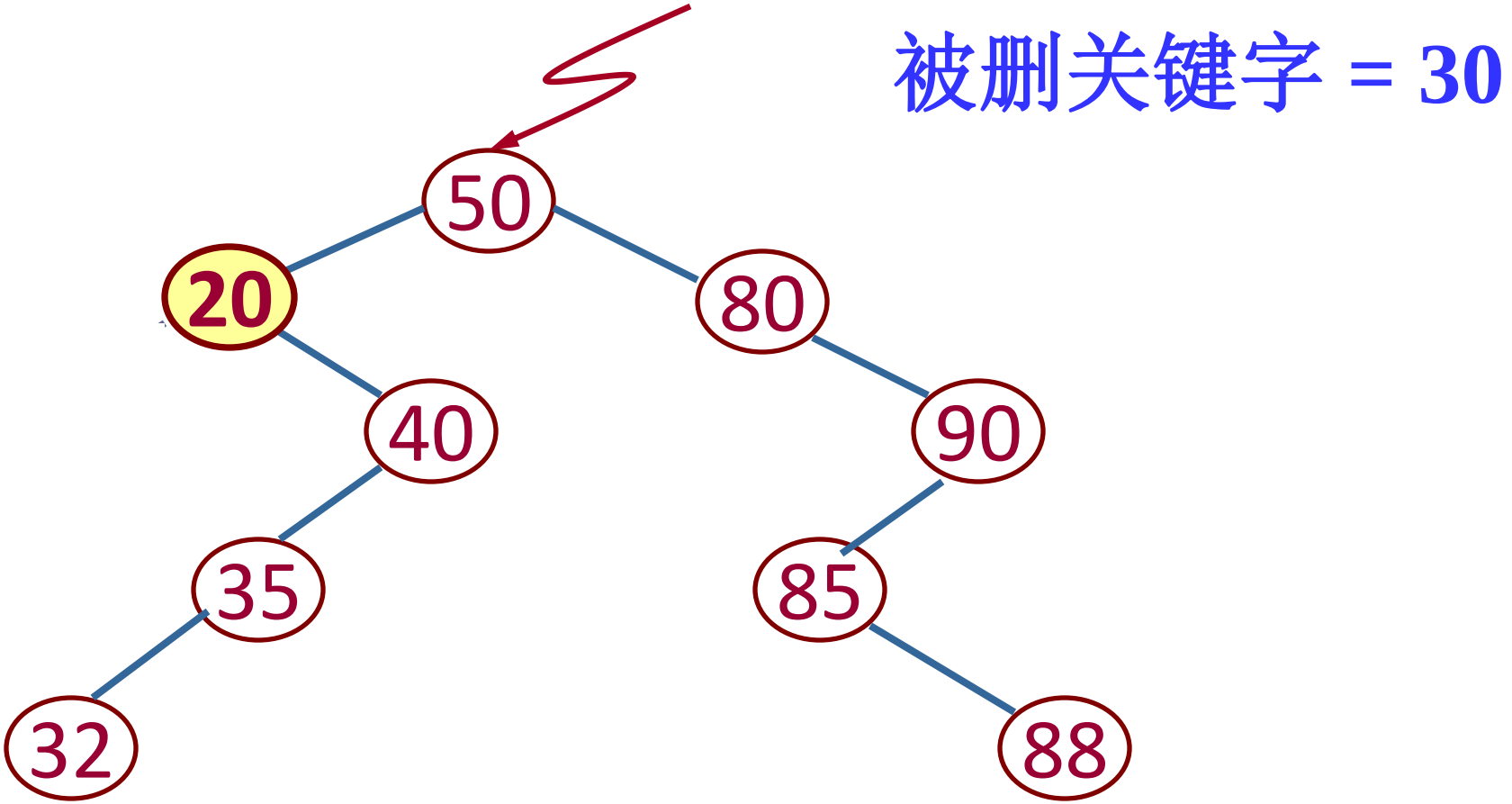
被删关键字 = 85

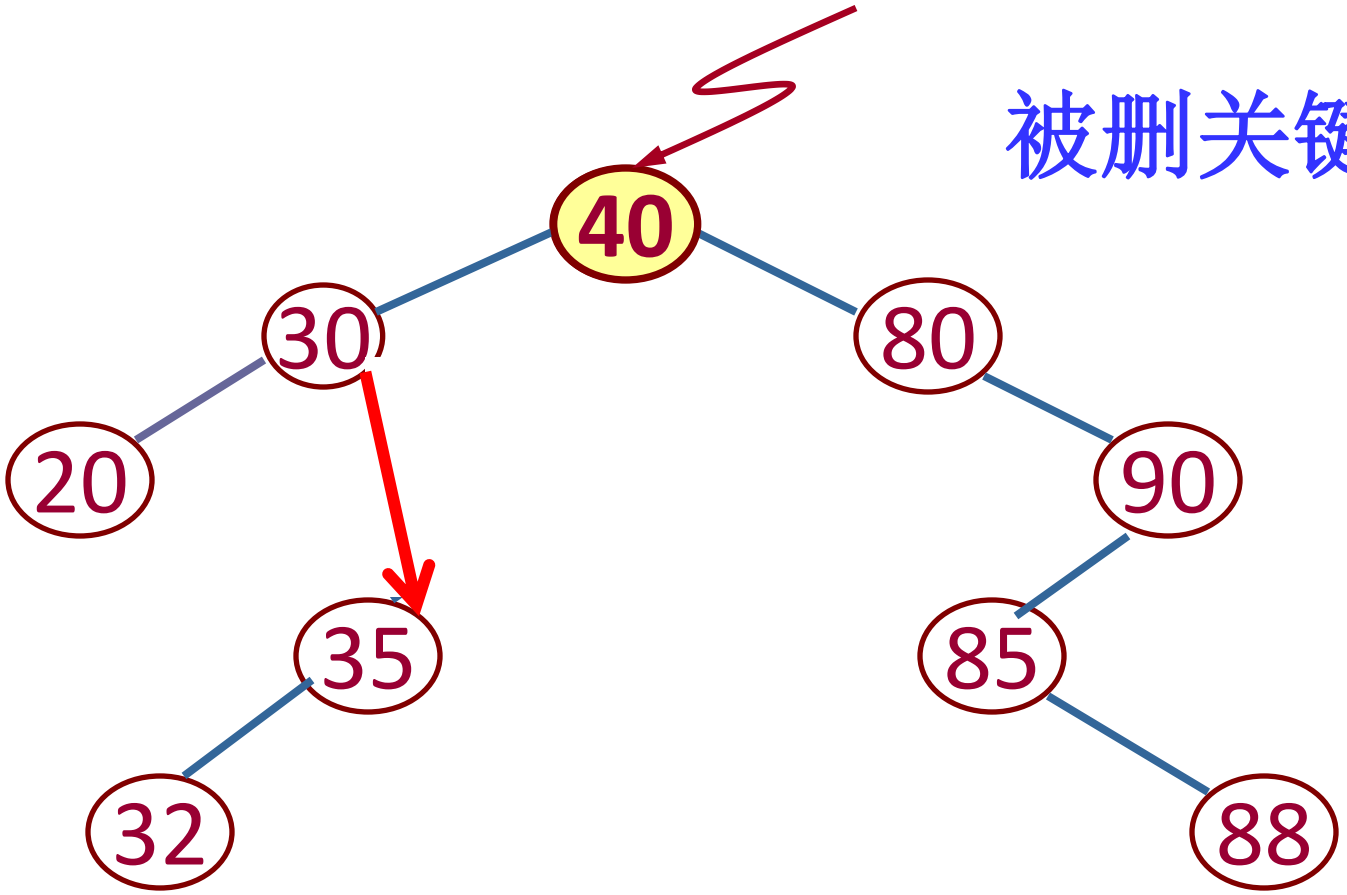


其双亲结点的相应指针域的值改为“指向被删除结点的左子树或右子树”。

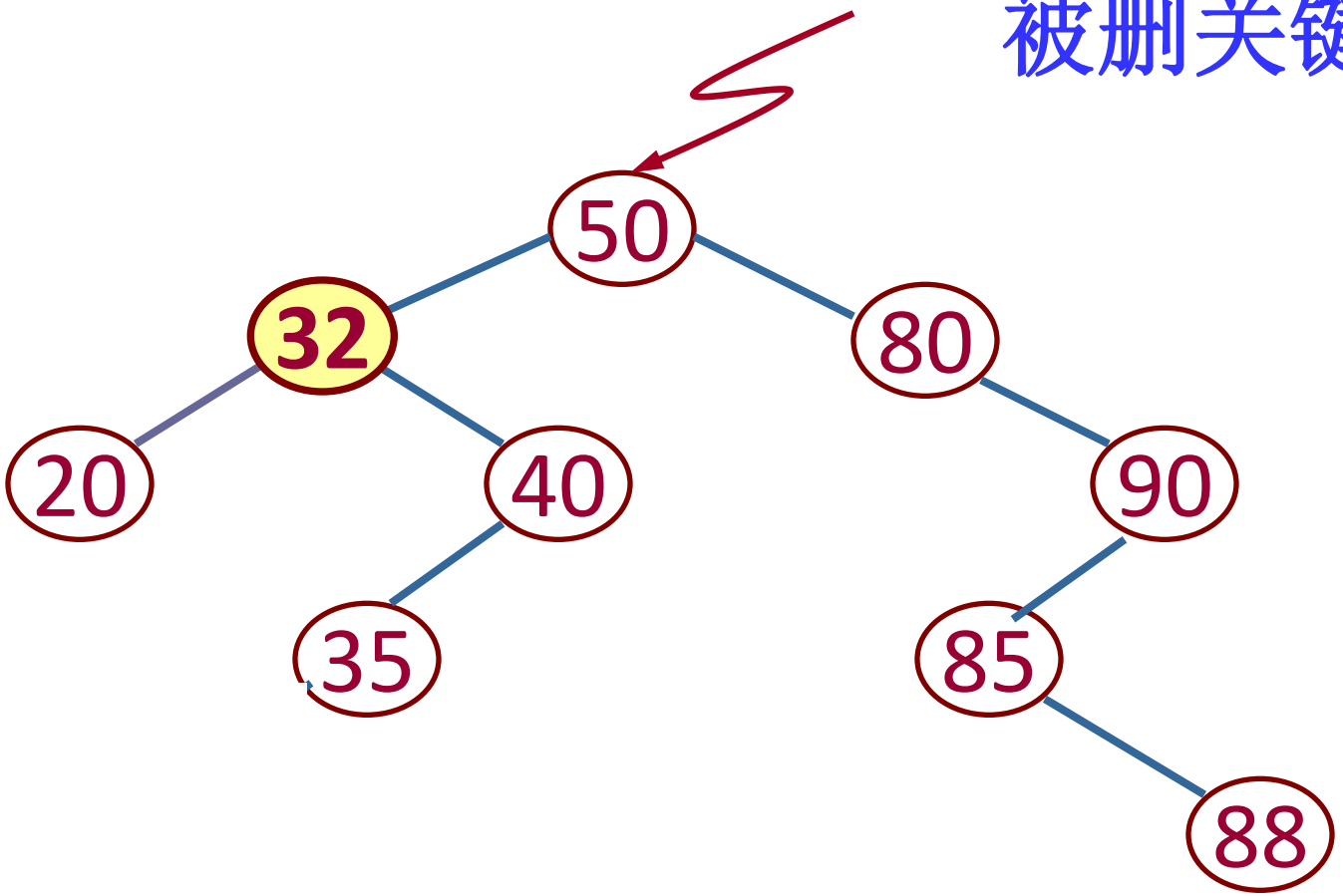
（c）被删除的结点既有左子树，也有右子树

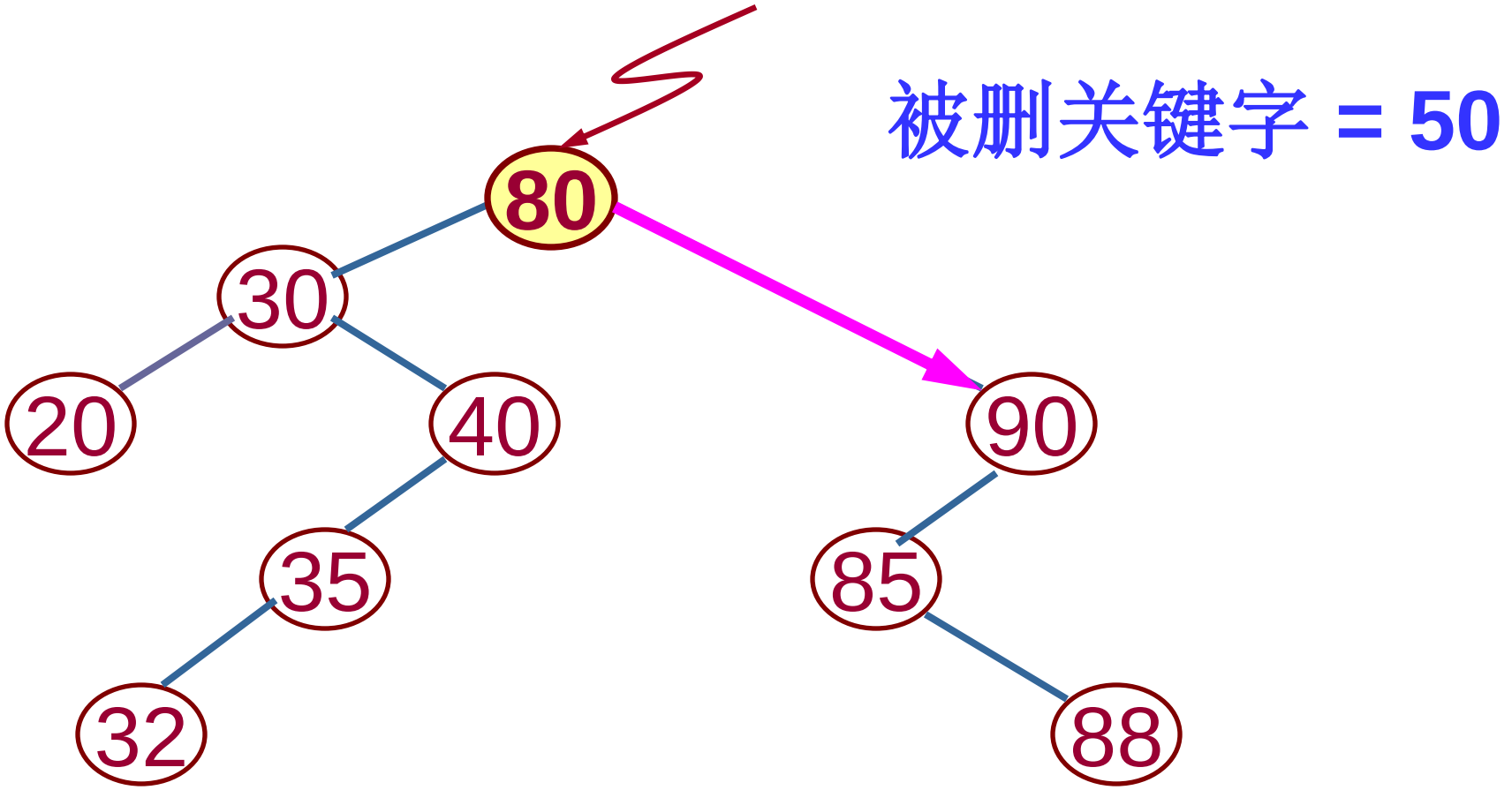
- 方法一：以被删结点的中序前驱结点的值替代之，然后再删除该前驱结点




被删关键字 = 50
40
30
80
20
90
35
85
32
88

- 方法二：以其中序后继替代之，然后再删除该后继结点




被删关键字 = 50
80
30
20
40
35
32
90
85
88

- **remove_root算法**

Binary Search Trees

```
template <class Record>
Error_code Search_tree<Record> ::
   remove_root(Binary_node<Record> *
   &sub_root)
/* Pre: sub_root is either NULL or points to a subtree of the
      Search_tree .
   Post: If sub_root is NULL , a code of not_present is returned.
      Otherwise, the root of the subtree is removed in such a way
      that the  properties of a binary search tree are preserved. The
      parameter sub_root is reset as the root of the modified subtree,
      and success is returned. */
```

Binary Search Trees

- implementations:
  - **Auxiliary Function to Remove One Node**

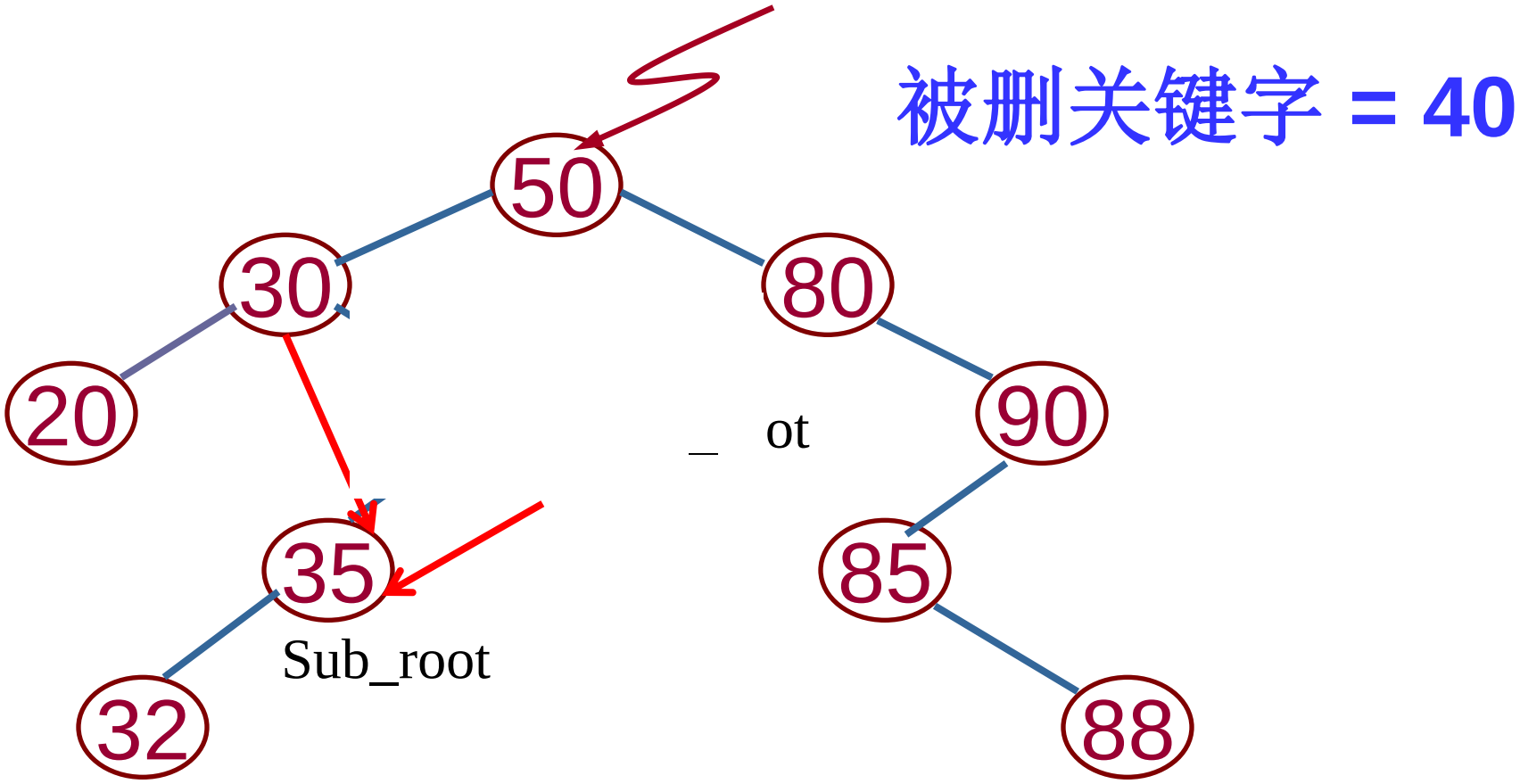
```
{
    if (sub_root == NULL)
            return not_present;
    Binary_node<Record> *to_delete = sub_root;
    if (sub_root->right == NULL)
       sub_root = sub_root->left;
    else if (sub_root->left == NULL)
      sub_root = sub_root->right;
```

Binary Search Trees

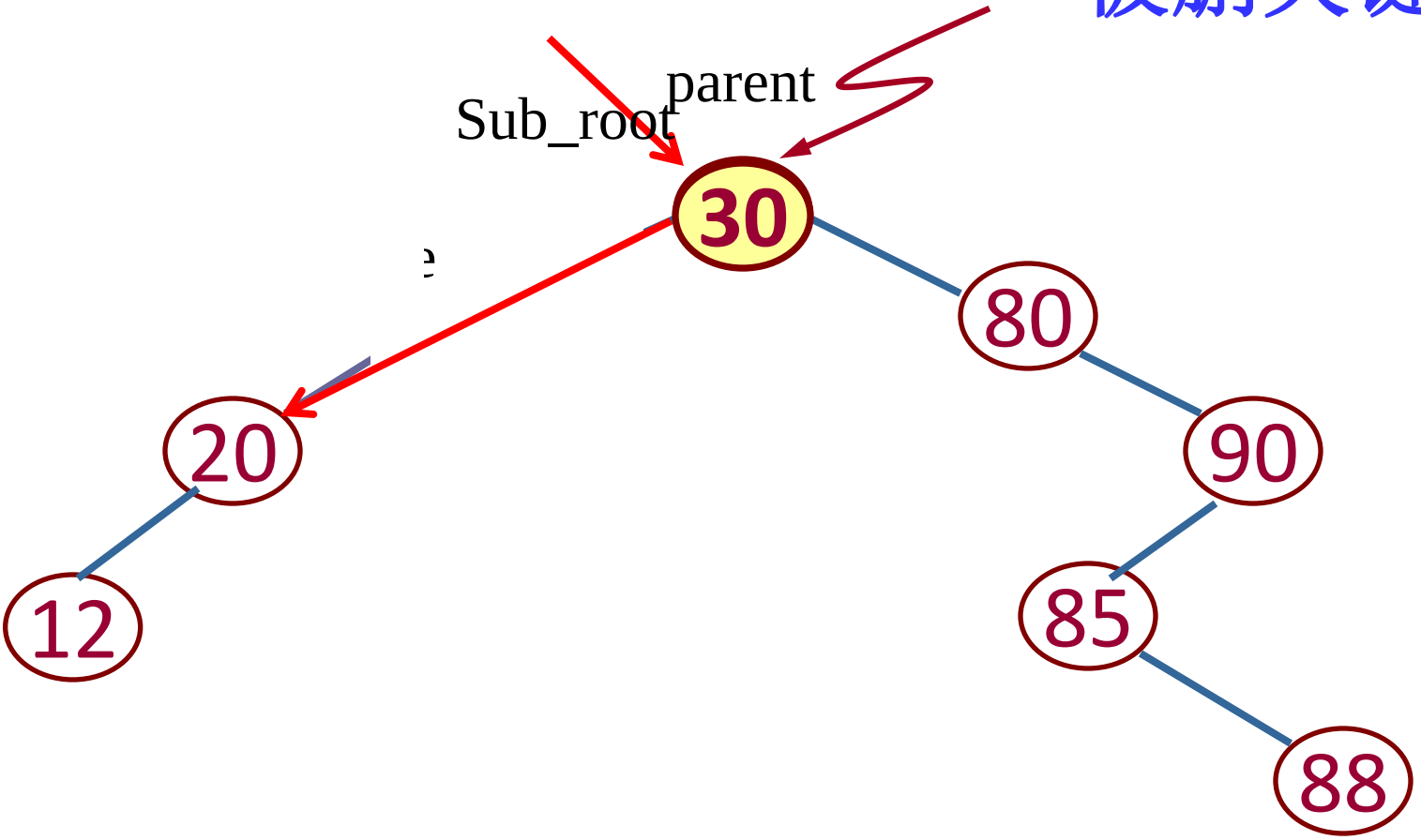
```
else { // Neither subtree is empty.
   to_delete = sub_root->left;
   // Move left to find predecessor(前驱）.
Binary_node<Record> *parent = sub_root;
// parent of to_delete
while (to_delete->right != NULL){
      parent = to_delete;
      to_delete = to_delete->right;
      }
}
```

```
sub_root->data = to_delete->data; // Move from to_delete to root.
if (parent == sub_root)
        sub_root->left = to_delete->left;
else
        parent->right = to_delete->left;
}
delete to_delete; //Remove to_delete from tree.
return success;
```

（c）被删除的结点没有右子树

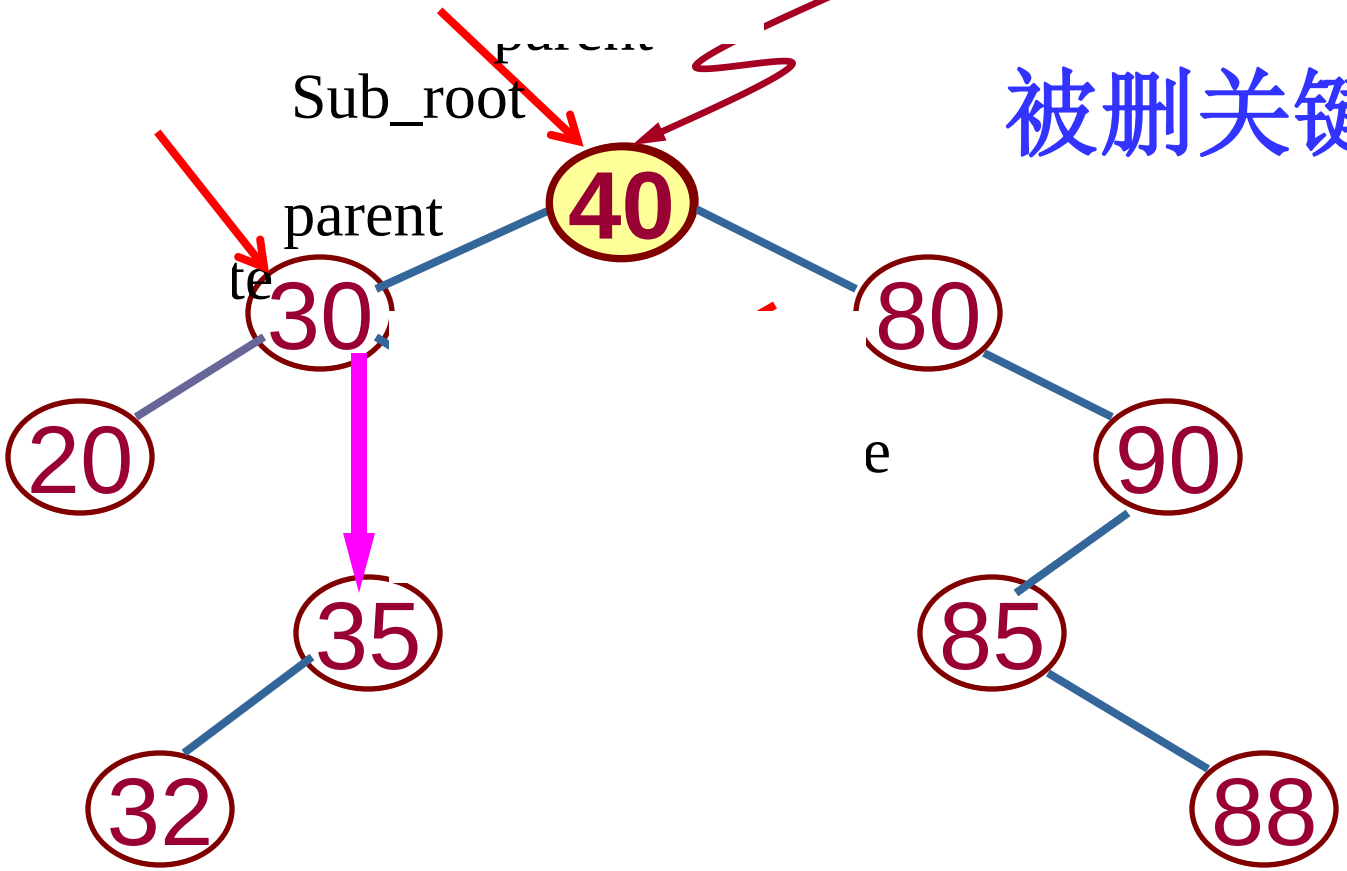


被删关键字 = 50

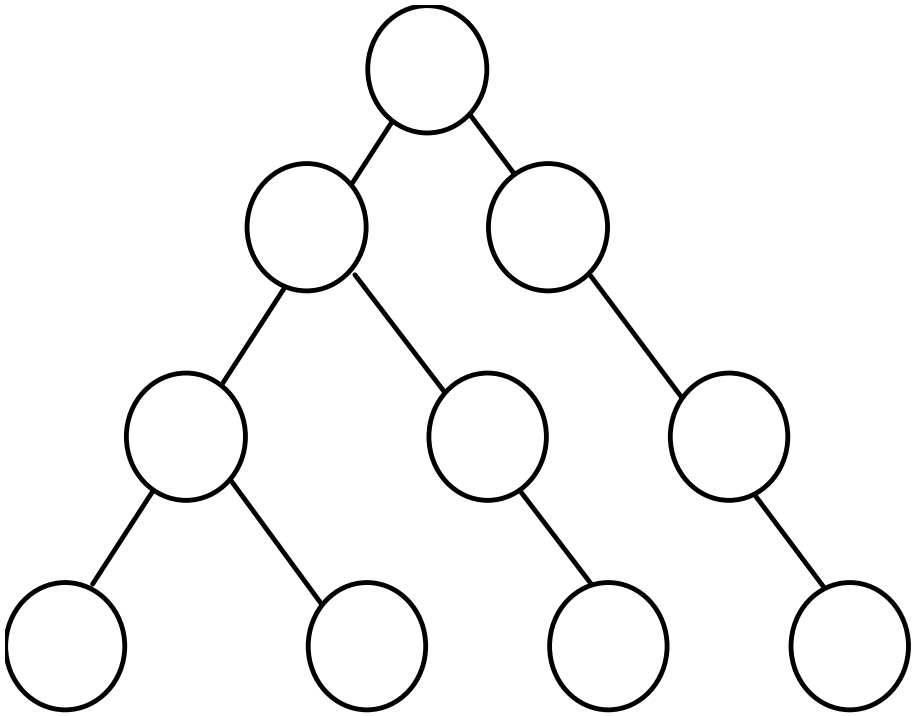


（c）被删除的结点既有左子树，也有右子树

被删关键字 = 50



• 下图所示形态的二叉排序树的各结点的关键字分别是1到10的10个自然数，填上各结点的关键字，并画出删除关键字为4的结点后的二叉排序树。



已知一组元素为（53，17，9，81，45，23，94，88，65）
（1）按元素输入顺序依次插入到初始为空的二叉查找树，画出最后形成的二叉查找树；
（2）画出在（1）所建立的二叉查找树上删除 53之后形成的二叉查找树，要求生成的二叉查找树是平衡的。

Binary Search Trees

- implementations:
  - **Treesort（树排序）**
    - When a binary search tree is traversed in inorder, the keys will come out in sorted order.
    - This is the basis for a sorting method, called *treesort*: Take the entries to be sorted, use the method insert to build them into a binary search tree, and then use inorder traversal to put them out in order.

Binary Search Trees

- implementations:
  - **Treesort（树排序）**
    - Theorem（定理） 10.1 Treesort makes exactly the same comparisons of keys as does quicksort（快速排序） when the pivot（基准） for each sublist（子表） is chosen to be the first key in the sublist.
    - Corollary(推论） 10.2 In the average case, on a randomly ordered list of length n, treesort performs:$2n \ln n + O(n) = 1.39n \lg n + O(n)$ comparisons of keys.

Binary Search Trees

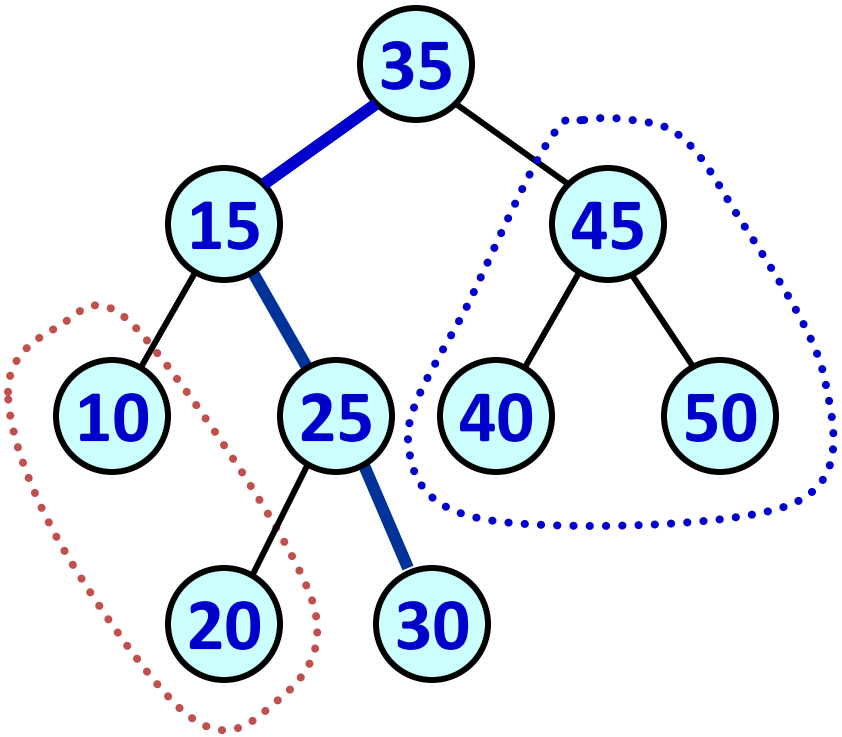
- implementations:
  - **Advantages and Drawback of Tree sort**
    - First advantage(优点） of treesort over quicksort: The nodes need not all be available at the start of the process, but are built into the tree one by one as they become available.
    - Second advantage: The search tree remains available for later insertions and removals（删除）.
    - Drawback（缺点）: If the keys are already sorted, then treesort will be a disaster(灾难）--the search tree it builds will reduce to a chain.
    - Treesort should never be used if the keys are already

更多算法

- 找最大值（最小值）
- 判别是否为二叉查找树
- 以有序序列创建平衡二叉树.....

二叉查找树例

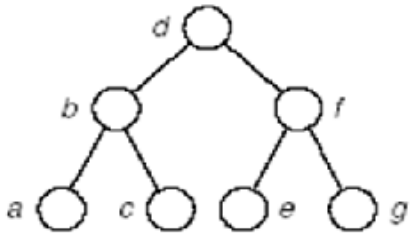
- 结点左子树上所有关键码小于结点关键码；
- 右子树上所有关键码大于结点关键码；

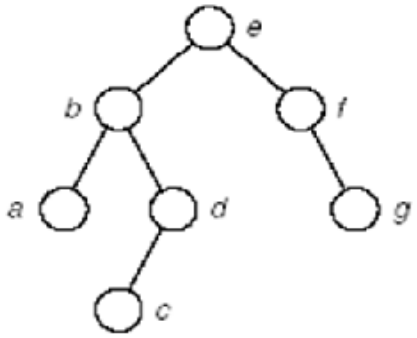


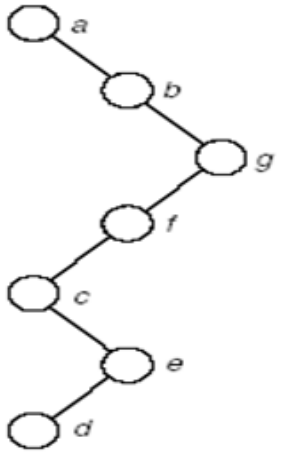
中序遍历得到有序序列，所以也称二叉查找树为二叉排序树；

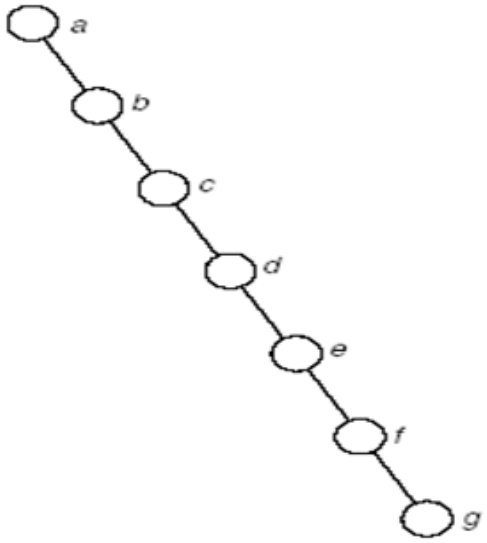
没有相同值结点；

具有同一组关键字的二叉查找树



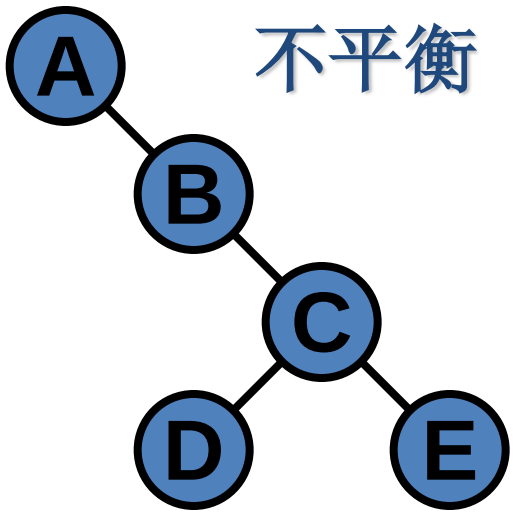




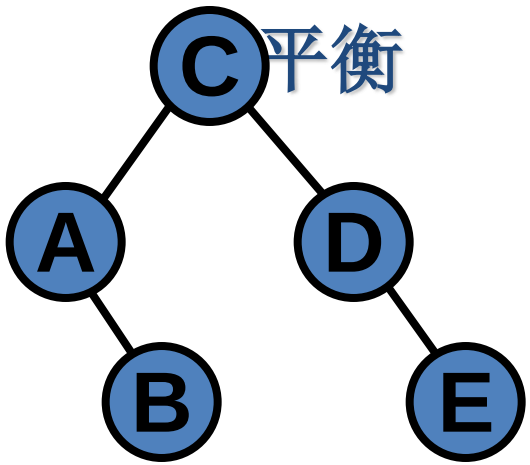


AVL树 高度平衡的二叉搜索树

- **AVL 树的定义** 一棵 AVL 树或者是空树，或者是具有下列性质的二叉查找树：它的根结点的左右子树的高度差的绝对值不超过1，并且根的左子树和右子树都是 AVL 树。



不平衡



平衡

结点的平衡因子
bf (balance factor)

- 每个结点附加一个数字，给出该结点右子树的高度减去左子树的高度所得的高度差，这个数字即为结点的平衡因子**bf**。
- **AVL**树任一结点平衡因子只能取 **–1, 0, 1**。
- 如果一个结点的平衡因子的绝对值大于**1**，则这棵二叉搜索树就失去了平衡，不再是**AVL**树。

如果一棵有 n 个结点的二叉搜索树是高度平衡的，其高度可保持在$O(\log_2 n)$，平均搜索长度也可保持在$O(\log_2 n)$。