数据结构与算法

第 5 章 二叉树

主讲：赵海燕

北京大学信息科学技术学院

“数据结构与算法”教学组

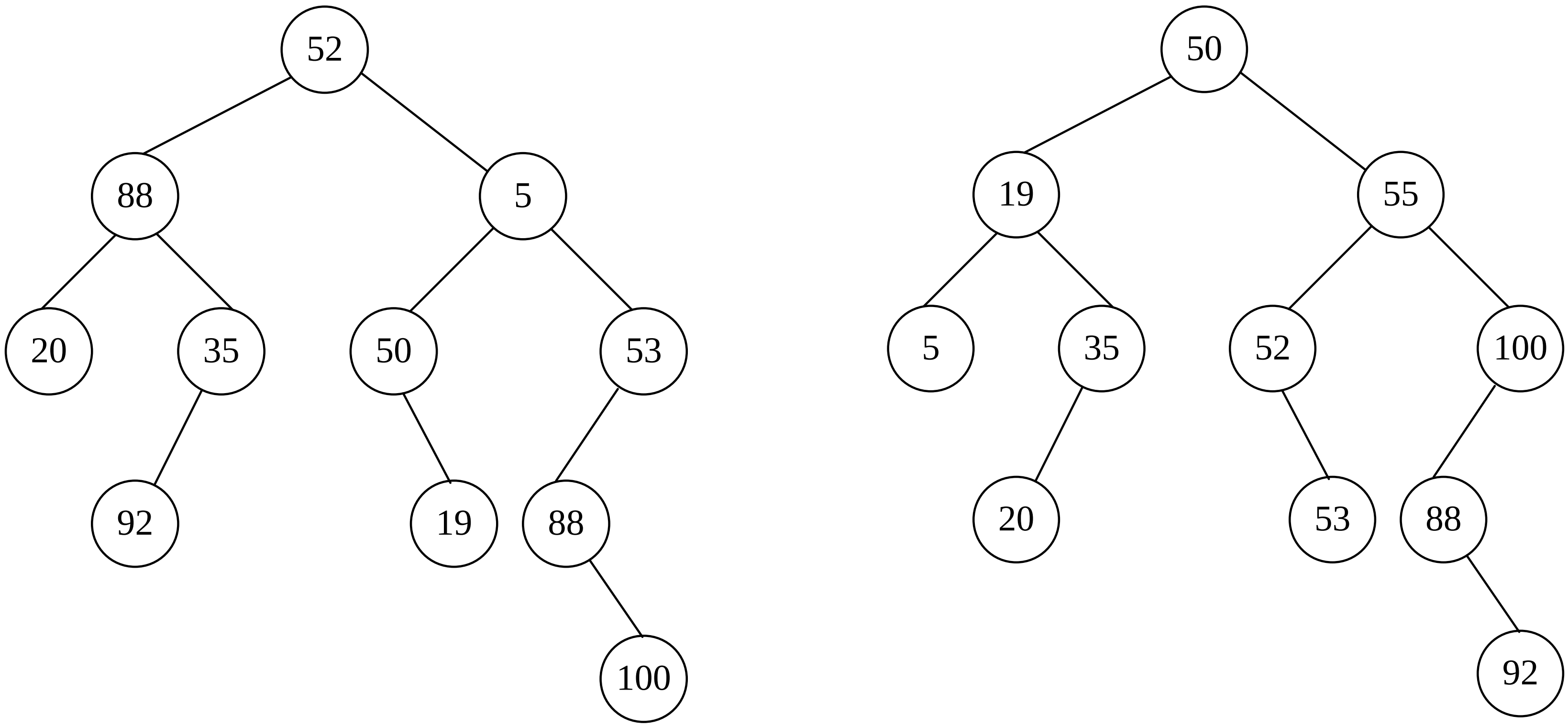
国家精品课“数据结构与算法”

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6, “十一五”国家级规划教材

二叉树回顾



左右两棵二叉树的异同、特点?

二叉搜索树

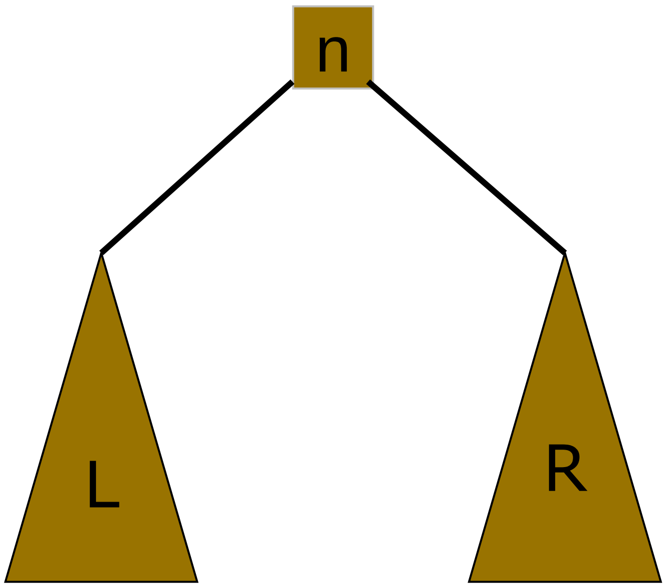
- 二叉树的一个**主要用途是提供对数据（包括索引）的快速检索**，而一般的二叉树对此并不具有性能优势

- **常用名称（同义词）**
  - 二叉搜索树（BinarySearchTree, 简称BST）
  - 二叉查找树
  - 二叉检索树
  - 二叉排序树

二叉搜索树：定义

- 若二叉树中的数据元素包含若干个域（field），其中一个称为**检索码**（key）*K* 的域作为**检索的依据**，则二叉搜索树定义为：
  - 或为**一棵空树**；
  - 或任何一个其码值为 *K* 的结点**均**满足如下条件
    1. 其左子树（若非空）的任一结点的码值均**小于** *K*；
    2. 其右子树（若非空）的任一结点的码值均**大于或等于** *K*；
    3. 其左右子树分别均为二叉搜索树

二叉搜索树：不变量

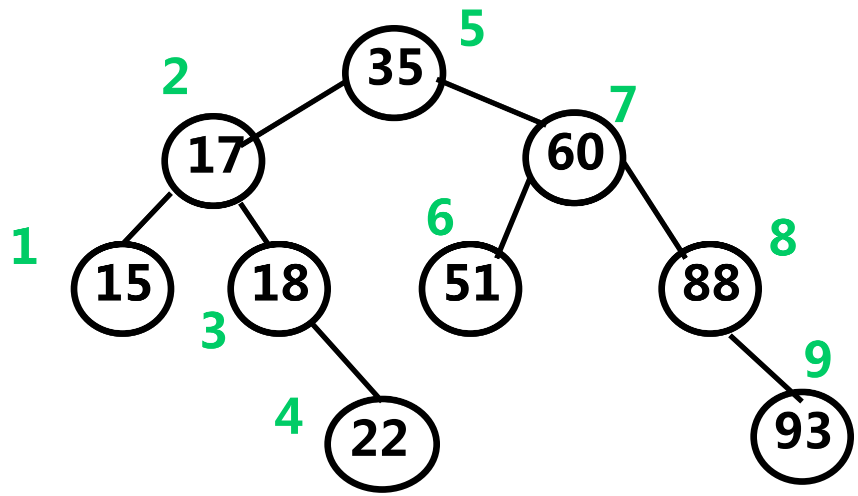


Splitting invariant

L.k < n.k

R.k ≥ n.k

二叉搜索树的性质



- 一棵二叉搜索树中序序列是按**码值由小到大**的排列

二叉搜索树上的操作

- 检索
- 插入（生成）
- 删除

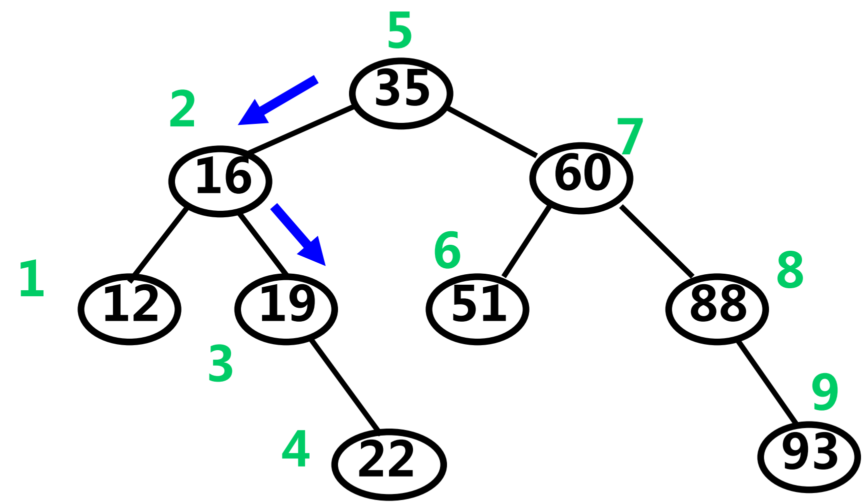
BST的检索

- **步骤**
  - 从根结点开始，在二叉搜索树中检索值 *K*。若根结点储存的值为 *K*，则检索结束
  - 若*K* 小于根结点的值，则只需检索左子树
  - 若*K* 大于根结点的值，只检索右子树

  如此， 一直持续到 *K* 被找到 或 遇上了一个树叶结点
  - 若遇上叶结点仍没发现 *K*， 那么*K* 不在该二叉搜索树中
- **代价如何?**
  - **二叉搜索树的效率体现在只需检索两个子树之一**

BST检索示例

查找码为 35 的结点
查找码为 19 的结点
查找码为 93 的结点
查找码为 40 的结点



比较次数?

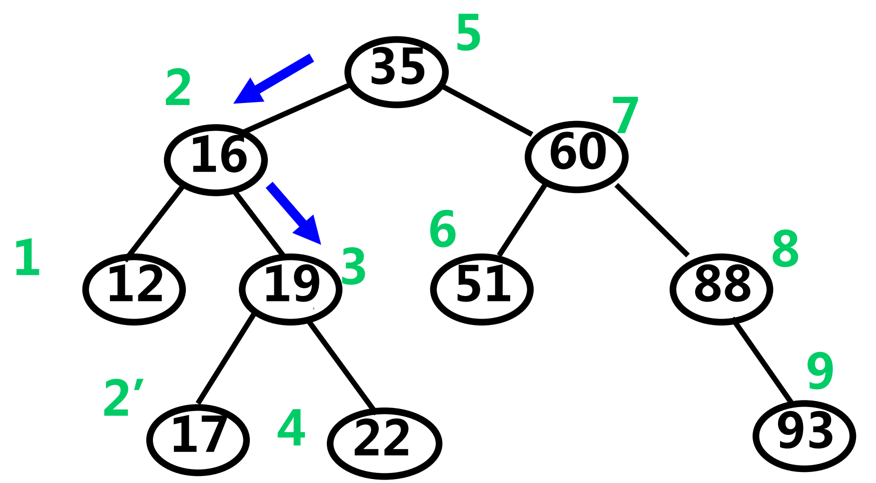
时间复杂度?

每次只需与结点的一棵子树相比较

O(h)

BST插入示例

- 插入17



BST的插入

- **待插入的结点码为 K**
  - 从根结点开始，若根结点为空，则将K 结点作为根插入，操作结束
  - 若K小于根结点的值，将其插入左子树
  - 若K大于根结点的值，将其插入右子树
- **保证结点插入后仍符合BST的定义，即，满足BST的不变量**

BST的插入算法

```
template<class T>
void BinarySearchTree<T>::InsertNode(BinaryTreeNode<T> *root , BinaryTreeNode<T>
      *newpointer)  {
      BinaryTreeNode<T> *pointer = NULL;
      if (root == NULL) {                              // 如果是空树
            Initialize(newpointer);                    // 则用指针newpointer作为树根
            return;
      }
      else pointer = root;
      while (pointer != NULL) {
         if (newpointer->value() == pointer->value())  // 如果存在相等的元素则不用插入
            return ;
        else if (newpointer->value() < pointer->value()) { // 如果待插入结点小于pointer的关键码值
            if (pointer->leftchild() == NULL) {         // 如果pointer没有左孩子
               pointer->left = newpointer;              // newpointer作为pointer的左子树
               return;
            }
            else pointer = pointer->leftchild();        // 向左下降
        }
        else {       // 若待插入结点大于pointer的关键码值
            if (pointer->rightchild() == NULL) {        // 如果pointer没有右孩子
               pointer->right = newpointer;             // newpointer作为pointer的右子树
               return;
            }
            else   pointer = pointer->rightchild();     // 向右下降
       }
      }
}
```

BST的插入分析

- 执行插入操作时，不必像有序线性表中插入元素那样涉及大量元素的移动，只需改动特定结点的空指针插入一个叶结点即可
- 与结点的检索操作一样，插入一个新结点的时间复杂度与根到插入位置的**路径长度相关**，因此在树形平衡时二叉搜索树的效率相当高

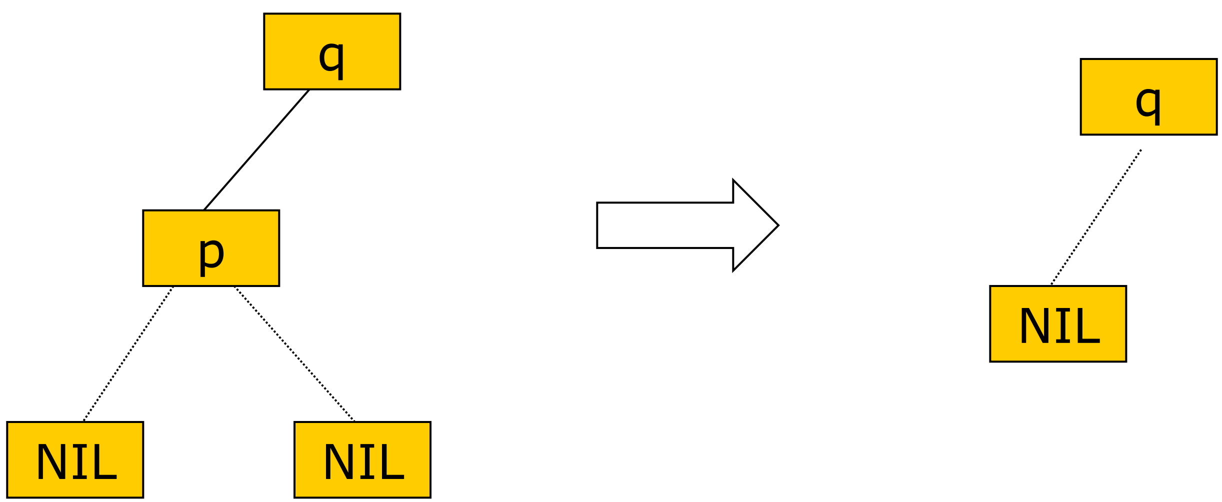
BST上的删除

- 删除BST上的一个结点，相当于删除有序序列中的一个结点，要求删除后仍能保持BST的排序特性，且树高变化较小

BST上的删除 #1

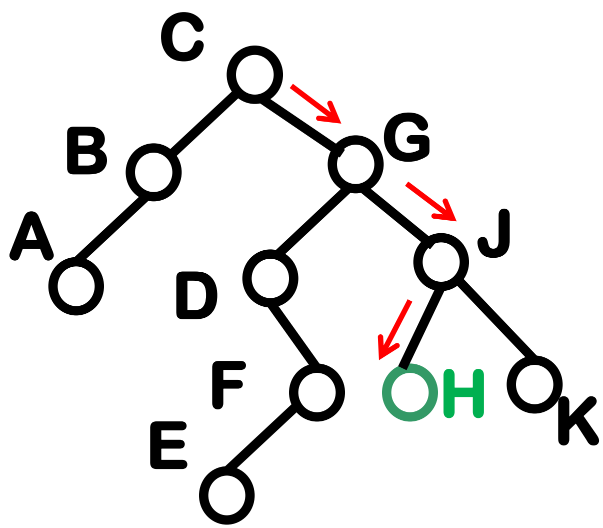
情况1. 叶结点

可以直接删除，其父结点的相应指针置为空



删除示例 #1

删除 H



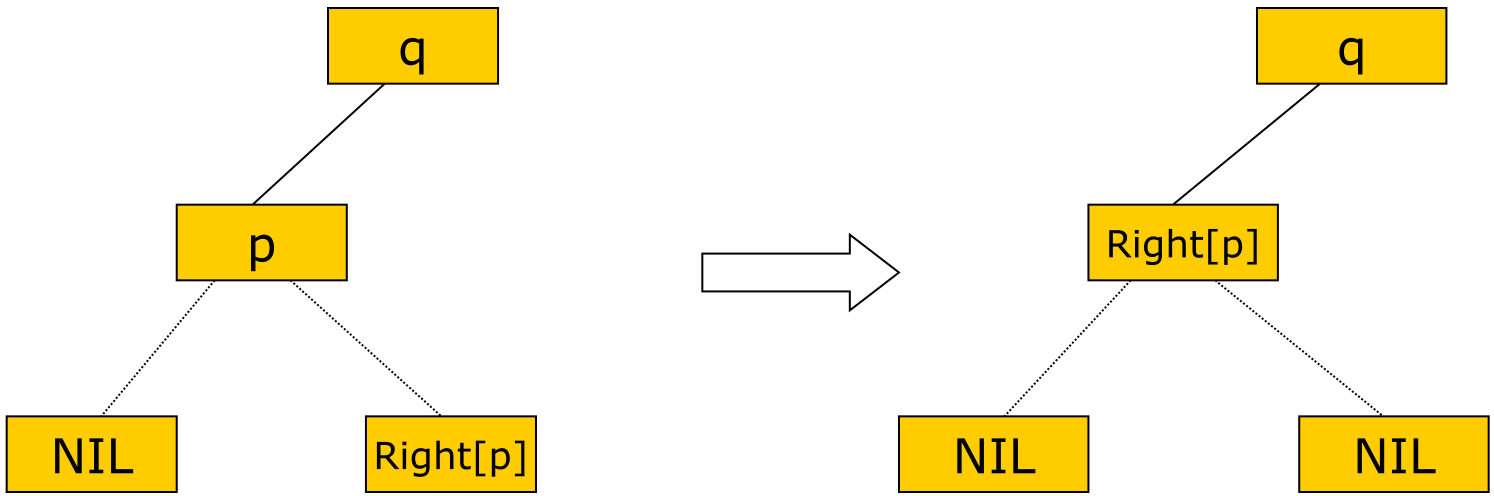
BST上的删除 #2

情况2. 只有一个子结点的结点p，分以下四种情况：

1. p是 q 的左子结点， p只有左子结点
2. p是 q 的左子结点， p只有右子结点
3. p是 q 的右子结点， p只有左子结点
4. p是 q 的右子结点， p只有右子结点

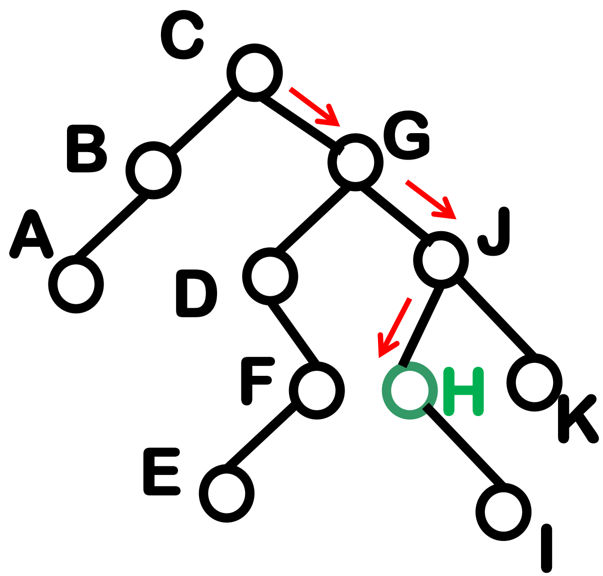
可让此子结点直接代替即可

BST上的删除 #2



删除示例 #2

删除 H



BST上的删除 #3

情况3. 被删结点 p 的左右子结点皆不空

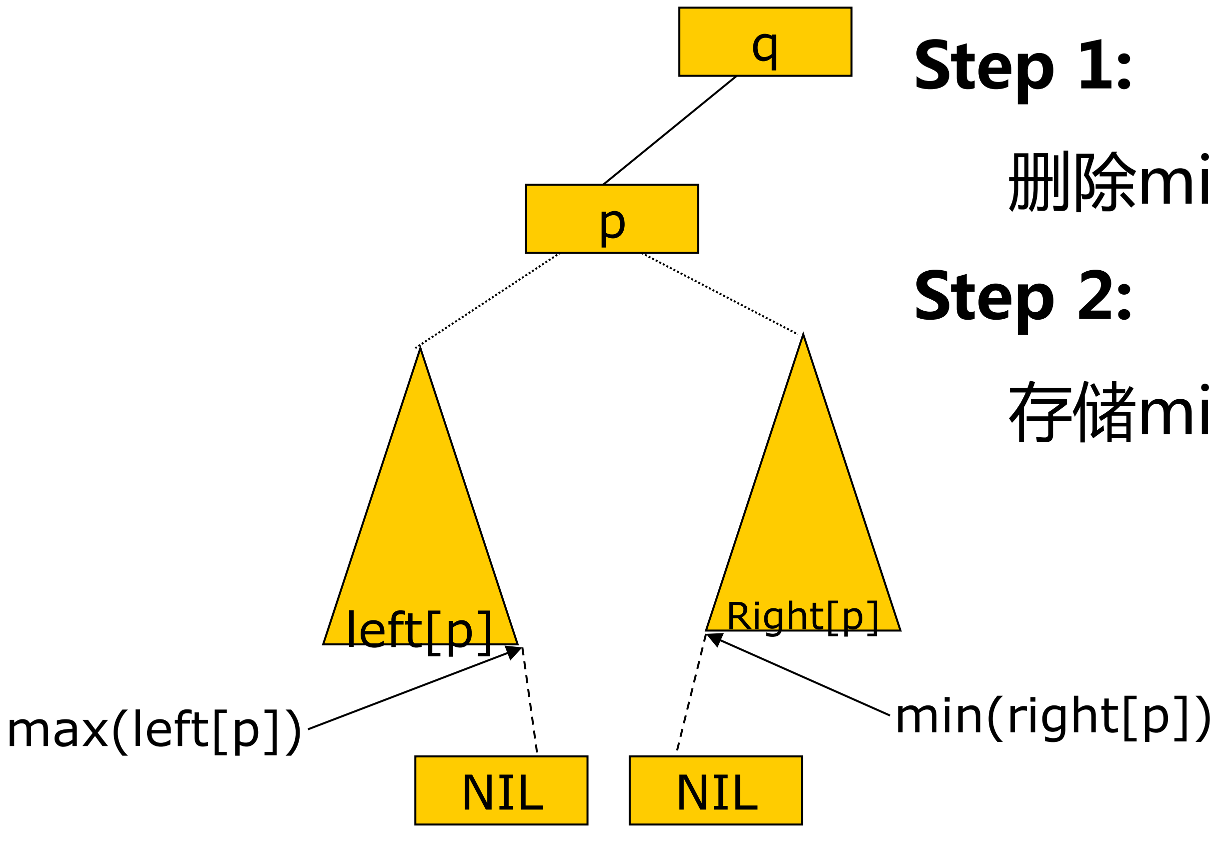
根据BST性质，此时要寻找能替换 **p** 的结点：比 p 的左子树中所有结点大，比p的右子树中所有结点小（或不大于）。两个选择：

- 左子树中**最大者**
- 右子树中**最小者**

二者都至多只有一个子结点（归结为情况#1 和情况#2）

- Why?

BST上的删除 #3



删除过程：

Step 1:

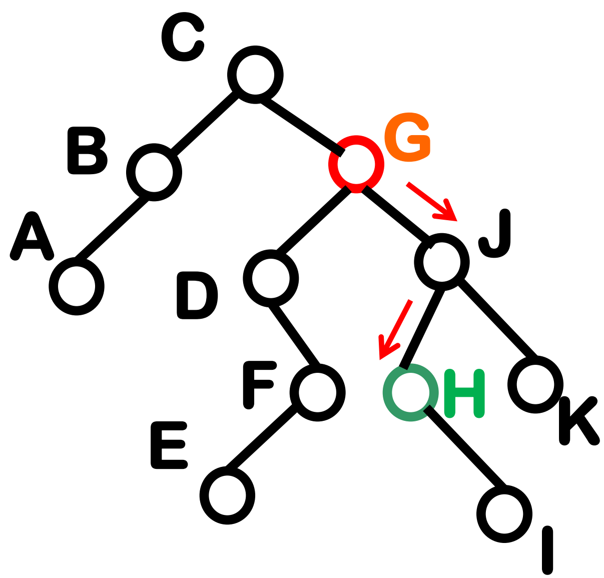
删除min(right[p])

Step 2:

存储min(right[p])到p中

删除示例 #3

删除 G



BST的删除算法

时间代价?

如何计算?

有关BST的讨论

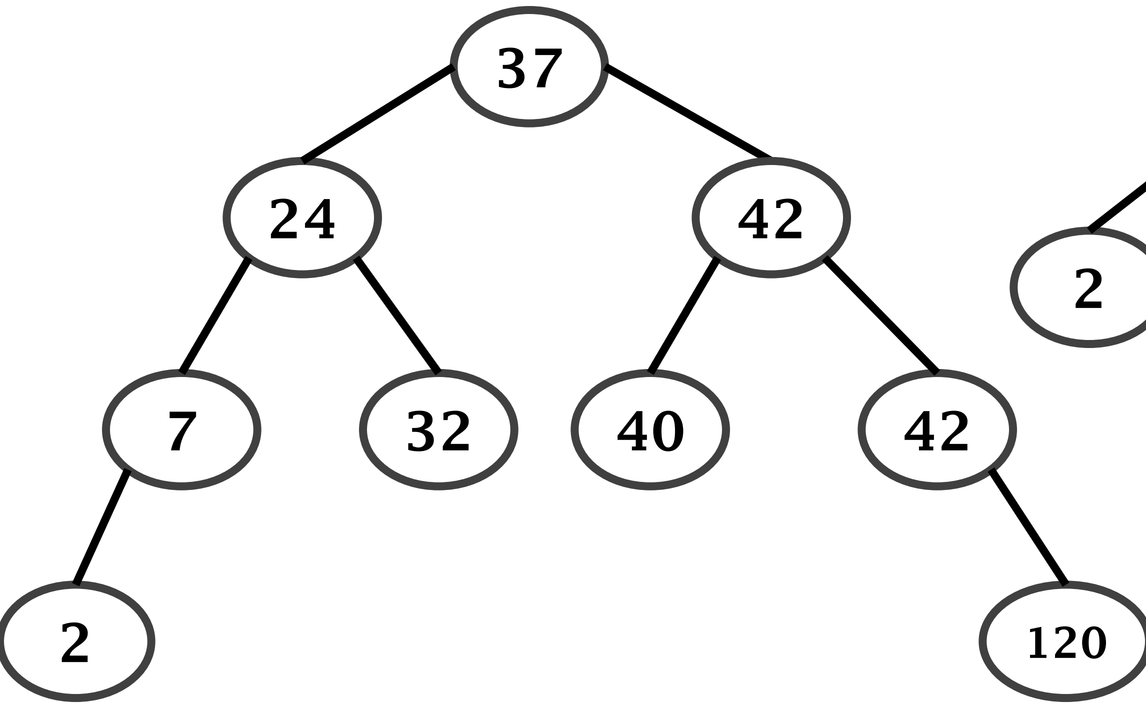
- **所有操作都围绕BST的性质，并一定要保持这个性质**
- **有待考虑：**
  - 是否允许有重复关键码
  - 多次插入和删除之后，BST的形状变化，平衡的问题
    - AVL 树，红黑树等

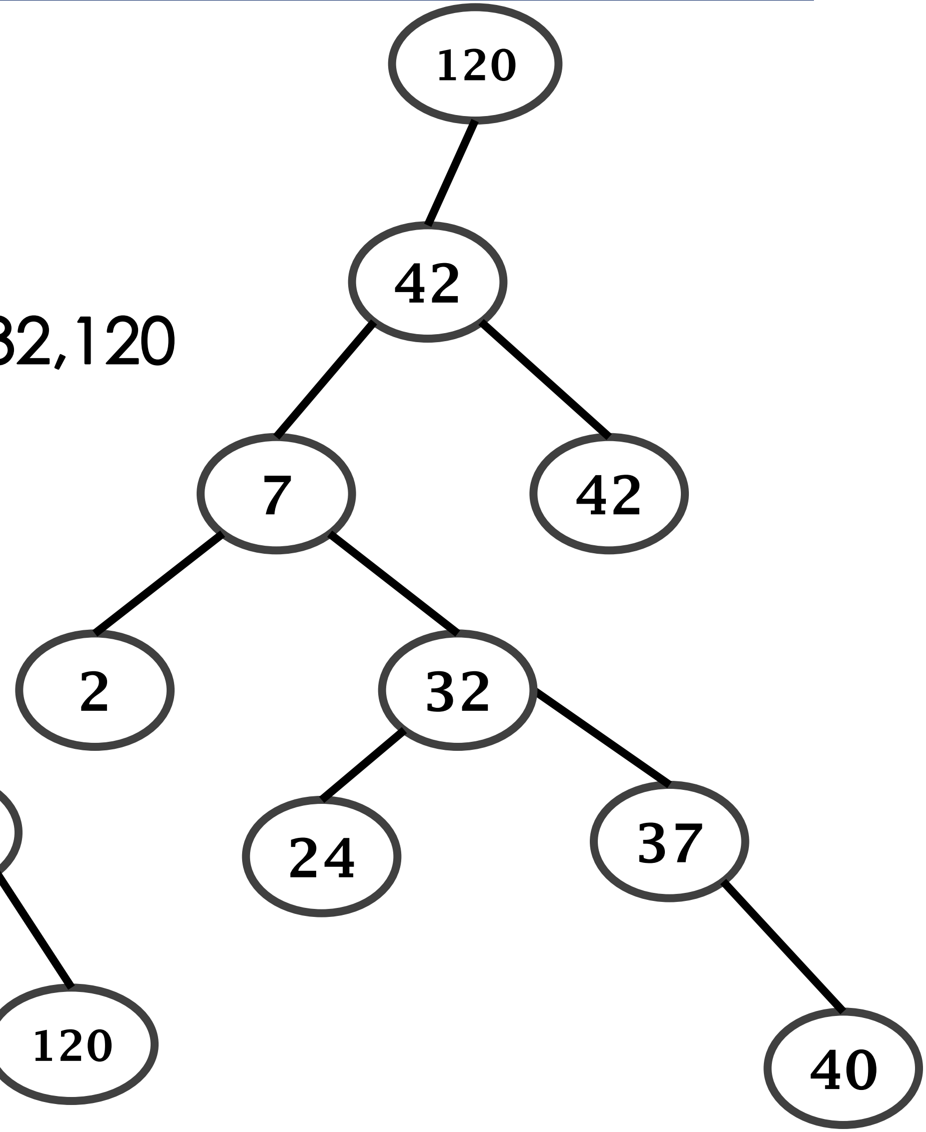
BST的建立

- 建立一个给定的关键码集合的BST，可从一个空的BST开始，将关键码逐个插进去
- 将关键码集合组织成BST，实际上起了对集合里元素按关键码排序的作用，中序周游BST得到已排序的关键码序列

BST的建立：示例

插入顺序：37,24,42,7,2,40,42,32,120





插入顺序：120,42,42,7,2,32,37,24,40

BST总结

- 树结构的一个重要应用是用来组织索引，BST是适用于内存储器的一种重要的树索引
- BST的插入和删除运算非常简单，代价与树高成正比
  - 往BST插入新结点或删除已有结点时，须保证操作结束后仍符合BST的定义

思考

- 如何防止BST退化成线性结构?