数据结构

深圳技术大学
大数据与互联网学院

第九章 查找

9.1 静态查找表
9.2 动态查找表
9.3 哈希表

9.2 动态查找表

一. 查找表

- 数据的组织和查找是大多数应用程序的核心，而查找是所有数据处理中最基本、最常用的操作。特别当查找的对象是一个庞大数量的数据集合中的元素时，查找的方法和效率就显得格外重要。
- 静态查找表
  - 仅作查询和检索操作的查找表。
- 动态查找表
  - 在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已存在的某个数据元素

9.2 动态查找表

- 动态查找的引入：当查找表以线性表的形式组织时，若对查找表进行插入、删除或排序操作，就必须移动大量的记录，当记录数很多时，这种移动的代价很大。
- 表结构本身是在查找过程中动态生成的
  - 若表中存在其关键字等于给定值key的记录,表明查找成功；否则插入关键字等于key的记录。
  - 利用树的形式组织查找表，可以对查找表进行动态高效的查找。

9.2 动态查找表

一. 二叉排序树

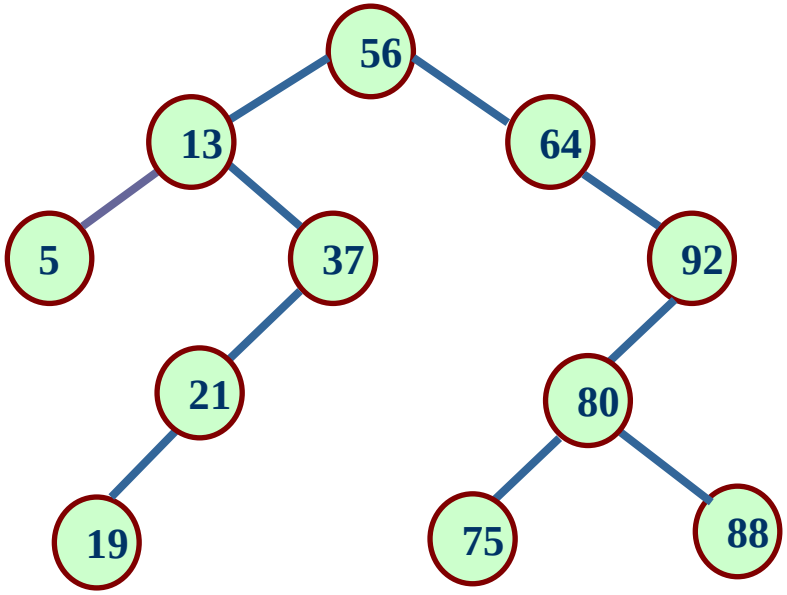
- 二叉排序树是空树或者是具有如下特性的二叉树：
  - 若它的左子树不空，则左子树上所有结点的值均小于根结点的值
  - 若它的右子树不空，则右子树上所有结点的值均大于根结点的值
  - 它的左、右子树也都分别是二叉排序树。

- 二叉排序树又称二叉查找树

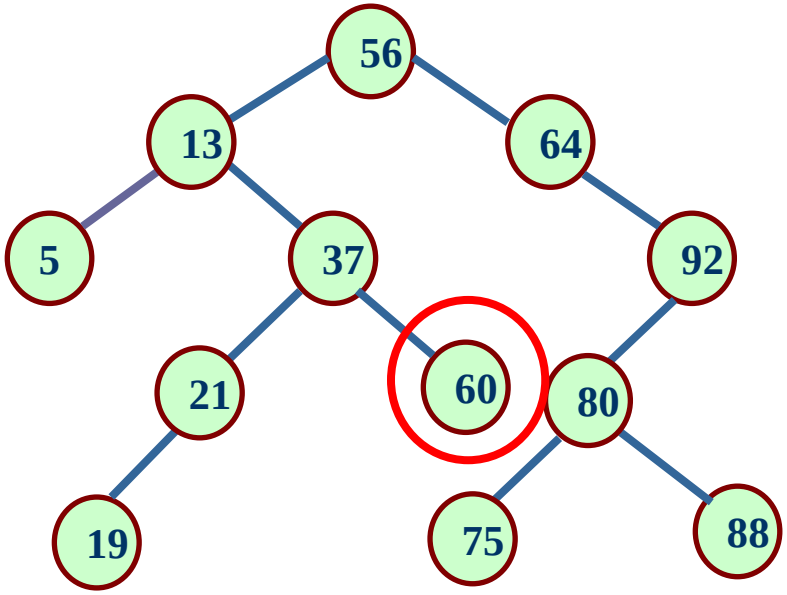
9.2 动态查找表

一. 二叉排序树

- 举例



二叉排序树



非二叉排序树

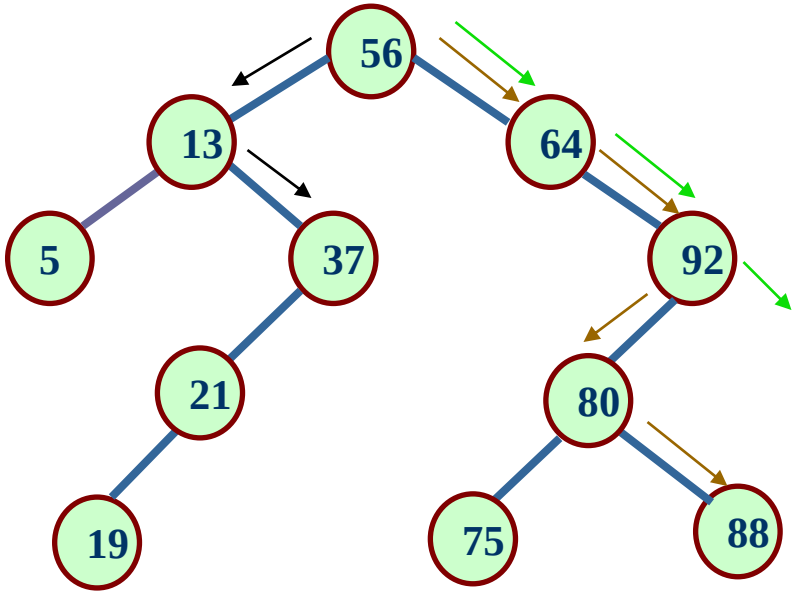
9.2 动态查找表

一. 二叉排序树

- 二叉排序树的查找：
  - 给定值与根结点比较：

1. 若相等，查找成功
2. 若小于，查找左子树
3. 若大于，查找右子树

- 如右图在二叉排序树中查找**37、88、94**



9.2 动态查找表

一. 二叉排序树

- 二叉排序树的树结点定义

```
class BSTNode {
    KeyType  key ;   //关键字域
    BSTNode  *Lchild, *Rchild; //  指针部分
};
```

9.2 动态查找表

一. 二叉排序树

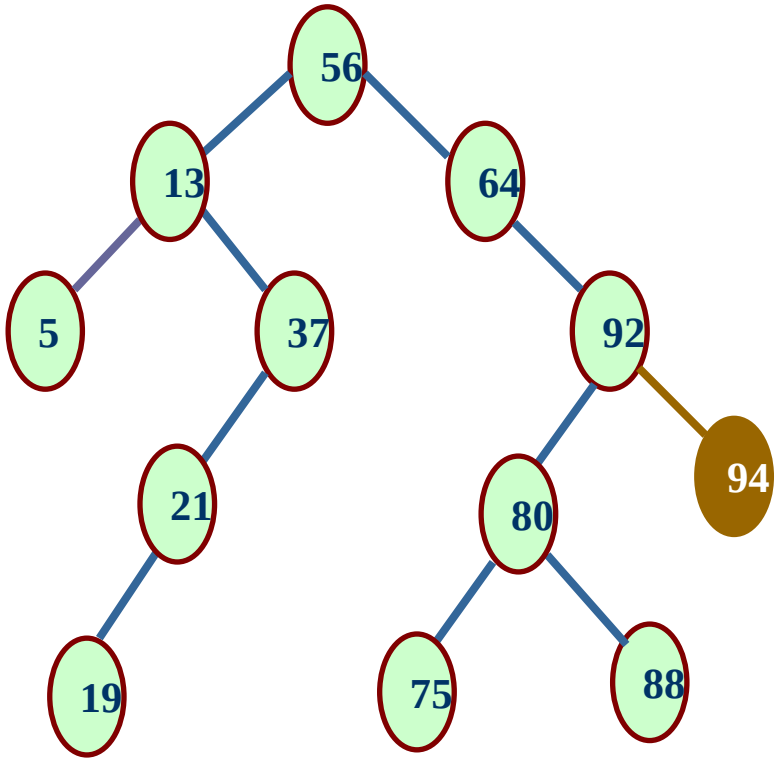
- 二叉排序树查找的递归程序

```
BSTNode* BST_Serach(BSTNode *T , KeyType key){
    if (T==NULL)  return(NULL) ;
    else {
        if ( Equal(T->key, key) )
              return(T) ;
        else if ( LessThan(key, T->key) )
            return(BST_Serach(T->Lchild, key)) ;
        else
            return(BST_Serach(T->Rchild, key)) ;
    }
}
```

9.2 动态查找表

一. 二叉排序树

- 二叉树排序树的插入
  - 二叉排序树是一种动态树表
  - 当树中不存在查找的结点时，作插入操作
  - 新插入的结点一定是叶子结点，只需改动一个结点的指针
  - 该叶子结点是查找不成功时路径上访问的最后一个结点左孩子或右孩子(新结点值小于或大于该结点值)

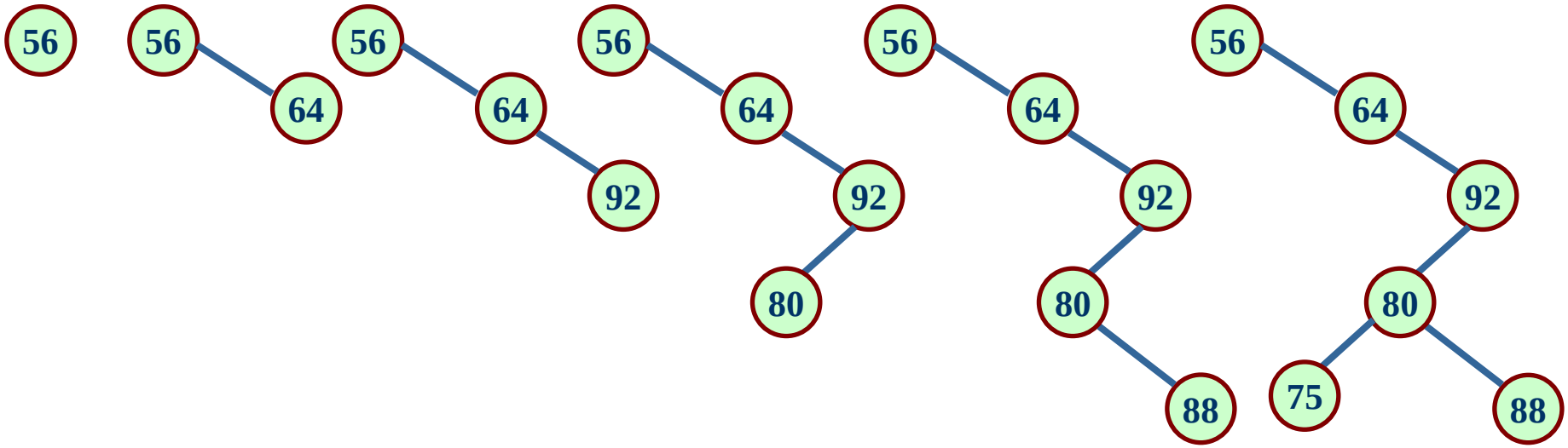


- 如右图，查找94不成功则插入

9.2 动态查找表

一. 二叉排序树

- 二叉排序树的生成
  - 例如，在初始为空的二叉排序树中依次插入56, 64, 92, 80, 88, 75，以下是生成过程



9.2 动态查找表

一. 二叉排序树

- 二叉排序树插入的递归程序

```
void Insert_BST (BSTNode* &T , KeyType key){
    if (T==NULL) { // 插入该结点
        T = new BSTNode;
        T->key = key;
        T->Lchild = T->Rchild = NULL ;
    } else {
        if (Equal(T->key, key) )
            return; //  已有结点
        else if (LessThan(key, T->key) )// 插入到左子树
            Insert_BST(T->Lchild, key);
        else   // 插入到右子树
            Insert_BST(T->Rchild, key) ;
    }
}
```

9.2 动态查找表

一. 二叉排序树

- 构建整棵二叉排序树的递归程序

```
#define ENDKEY  65535
BSTNode *create_BST(){
   KeyType key ;
   BSTNode *T=NULL ;
   cin >> key ;
   while (key!=ENDKEY){
        Insert_BST(T, key) ;
        cin >> key ;
   }
   return(T) ;
}
```

9.2 动态查找表

一. 二叉排序树

- 二叉树排序树的删除
  - 删除二叉排序树中的一个结点后，必须保持二叉排序树的特性（左子树的所有结点值小于根结点，右子树的所有结点值大于根结点）
  - 也即保持中序遍历后，输出为有序序列

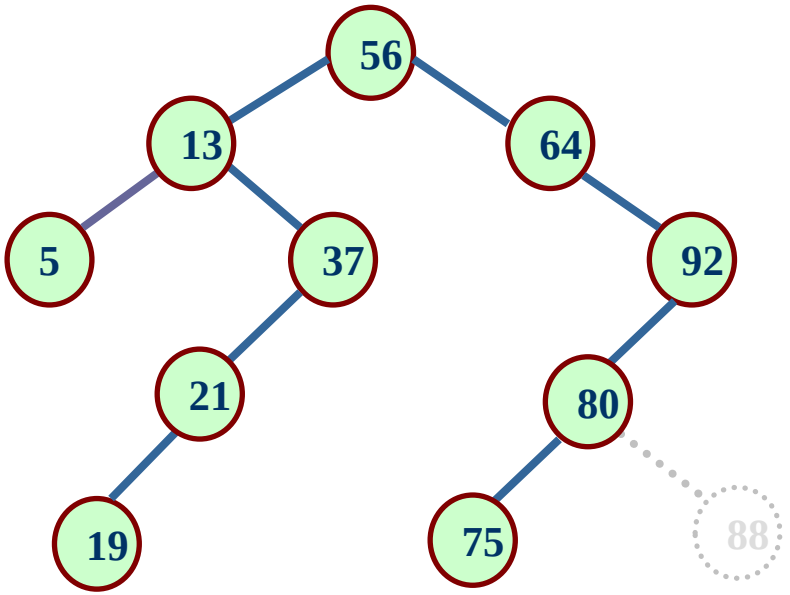
- 被删除结点具有以下三种情况：
  - 叶子结点
  - 只有左子树或右子树
  - 同时有左、右子树

9.2 动态查找表

一. 二叉排序树

- 二叉树排序树的删除
  - 被删除结点是叶子结点，则直接删除结点，并让其父结点指向该结点的指针变为空

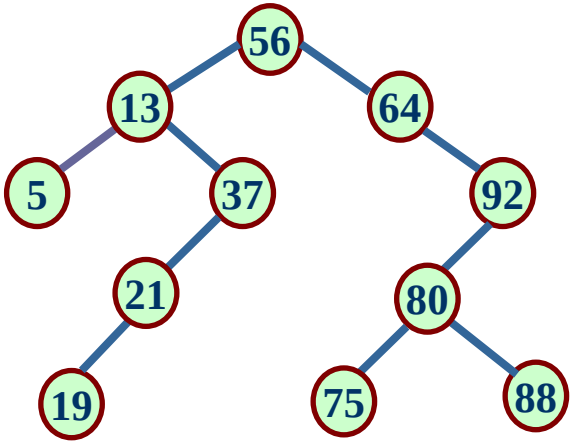
删除结点**88**



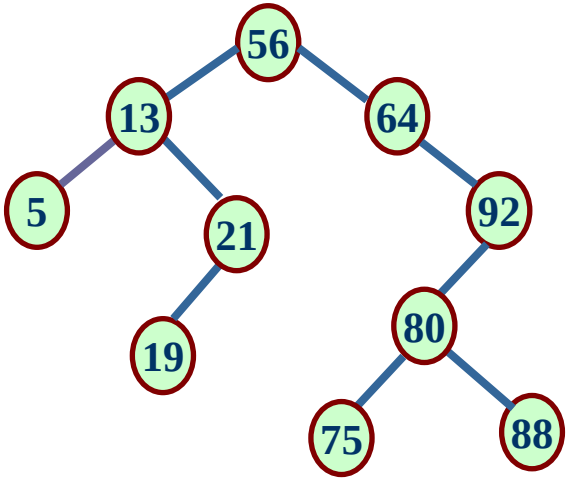
9.2 动态查找表

一. 二叉排序树

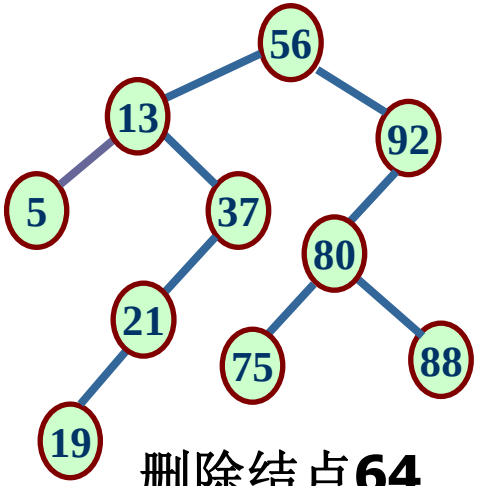
- 二叉树排序树的删除
  - 被删除结点只有左子树或右子树
  - 删除结点，让其父结点指向该结点的指针指向其左子树(或右子树)，即用孩子结点替代被删除结点即可



原图



删除结点37
(只有左子树)

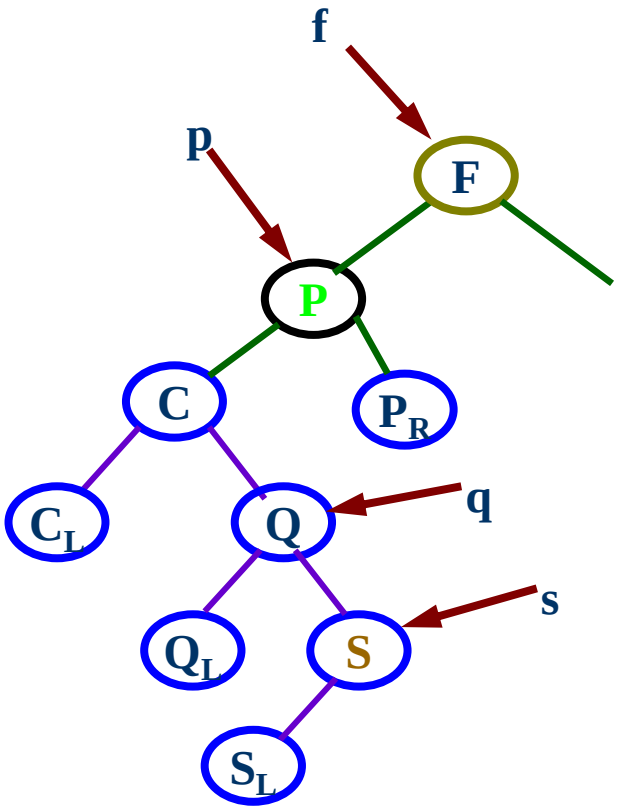


删除结点64
(只有右子树)

9.2 动态查找表

一. 二叉排序树

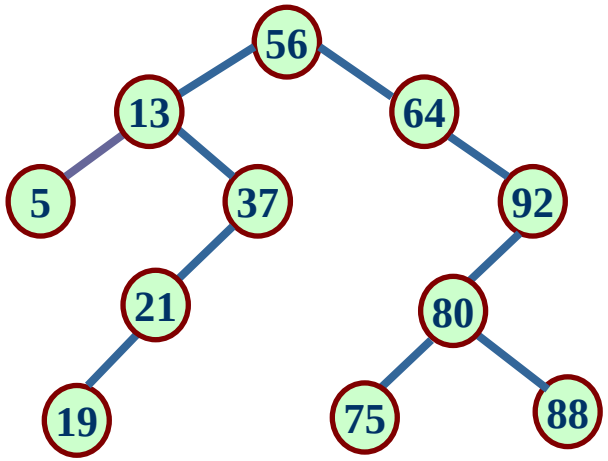
- 二叉树排序树的删除
  - 被删除结点P既有左子树，又有右子树
  - 以中序遍历时的直接前驱S替代被删除结点P，然后再删除该直接前驱（只可能有左孩子）
  - 实质就是用当前结点P的左子树的最大右孩子S替换P，然后再删除S



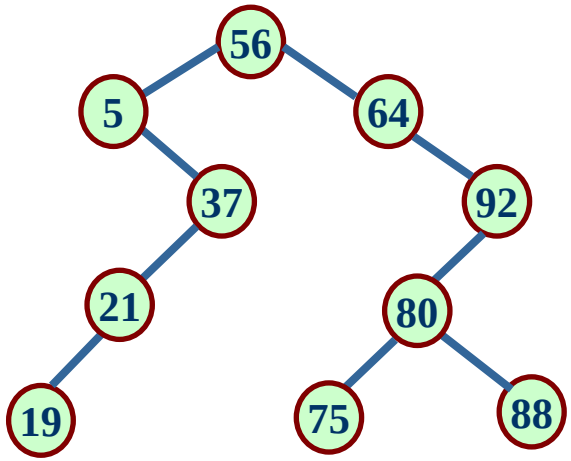
9.2 动态查找表

一. 二叉排序树

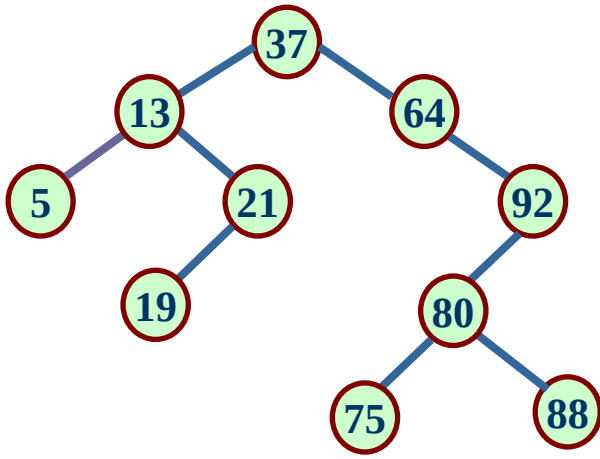
- 二叉树排序树的删除
  - 被删除结点P既有左子树，又有右子树，举例



原图



删除结点13



删除结点56

9.2 动态查找表

一. 二叉排序树

- 二叉树排序树删除的递归程序

```
Status DeleteBST(BSTNode *T,  KeyType key) { // 算法9.7
    // 若二叉排序树T中存在关键字等于key的数据元素时，
    // 则删除该数据元素结点p，并返回TRUE；否则返回FALSE
    if (!T) return FALSE; // 不存在关键字等于key的数据元素
    else {
        if (Equal(key, T->key))  // 找到关键字等于key的数据元素
            return Delete(T);
        else if (LessThan(key, T->key))
            return DeleteBST(T->Lchild, key);
        else return DeleteBST(T->Rchild, key);
  }
} // DeleteBST
```

9.2 动态查找表

一. 二叉排序树

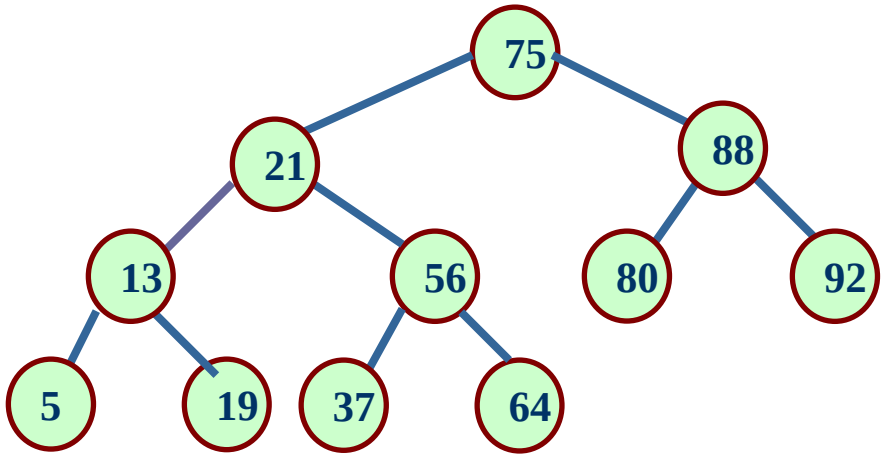
- 二叉树排序树删除的递归程序

```
Status Delete(BSTNode *p) {// 删除结点p，并重接它的左或右子树
      BSTNode *s; //保存重新连接的结点
   if (!p->Lchild && !p->Rchild) { // 左右均为空，是叶子结点
         if (p->parent->Lchild == p) p->parent-Lchild = NULL;
           else p->parent-Rchild = NULL;
           delete p;
      } else if (!p->Rchild){    // 右子树空则只需重接它的左子树
       s = p->Lchild; *p = *s; delete s;
      } else if (!p->Lchild) {    // 左子树空则只需重接它的右子树
            s = p->Rchild; *p = *s; delete s;
      } else {  // 左右子树均不空
       s = p->Lchild;                  // 转左，然后向右到尽头，
       while (s->Rchild)  s = s->Rchild;  //s指向被删结点的替换结点
       if (s->parent != p)
           s->parent->rchild = s->lchild;  // 重接s父结点的右子树
       else p->lchild = s->lchild;          // 重接s父结点的左子树
       *p = *s; delete s;
   }
   return TRUE;
} // Delete
```

9.2 动态查找表

一. 二叉排序树

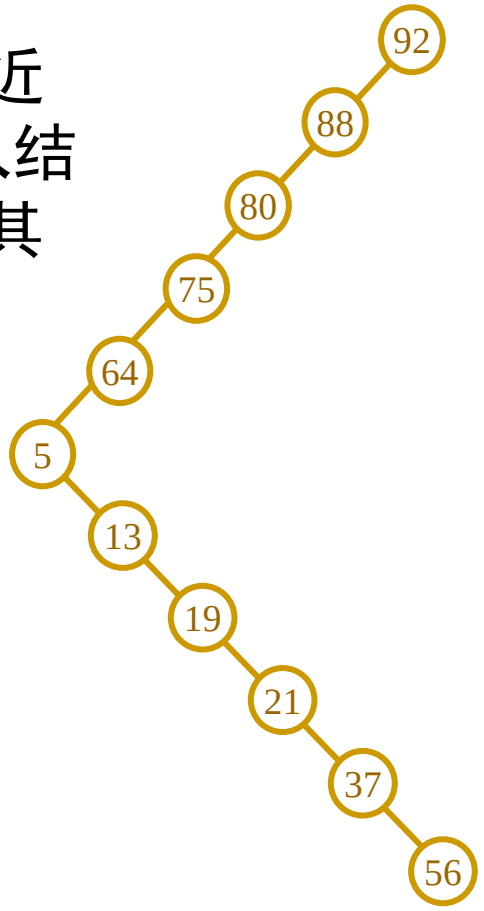
- 算法性能
  - 在最好的情况下，二叉排序树为一近似完全二叉树时，其查找深度为$\log_2 n$量级，即其时间复杂性为$O(\log_2 n)$



9.2 动态查找表

一. 二叉排序树

- 算法性能
  - 在最坏的情况下，二叉排序树为近似线性表时(如以升序或降序输入结点时)，其查找深度为n量级，即其时间复杂性为0(n)



9.2 动态查找表

一. 二叉排序树

- 二叉树排序树的特性
  - 一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列（通过中序遍历）
  - 插入新记录时，只需改变一个结点的指针，相当于在有序序列中插入一个记录而不需要移动其它记录
  - 二叉排序树既拥有类似于折半查找的特性，又采用了链表作存储结构
  - 当插入记录的次序不当时(如升序或降序)，则二叉排序树深度很深(11)，增加了查找的时间