上节复习

- 静态查找表
  - 仅作查询和检索操作的查找表
  - 顺序查找、折半查找、索引顺序查找
  - 掌握三种方法的查找过程和计算查找次数
- 顺序查找
  - 从最后开始查找，带哨兵
  - 查找成功时的平均查找次数ASL是(n+1)/2，失败查找次数是n+1
- 折半查找：
  - 要求表有序
  - 每次找中间位置元素比较，查找范围缩小一半
  - 平均查找次数ASL≈Log_2n
- 索引顺序查找：
  - 要求有索引表，对主表进行分块和建立索引
  - 索引表有序，块内无序
  - 先用折半查找或顺序查找搜索索引表，再用顺序查找进行块内搜索

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

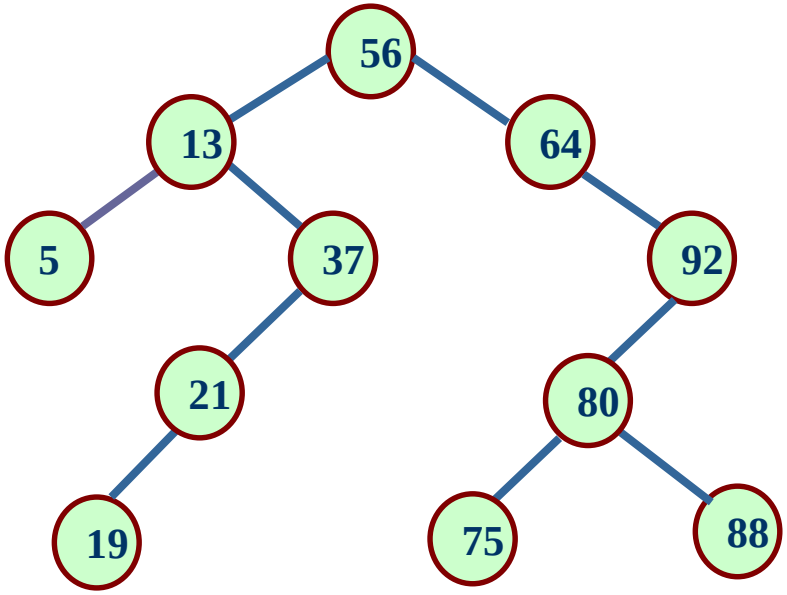
- 二叉排序树BST(Binary Sort Tree或Binary Search Tree)是空树或者是具有如下特性的二叉树：
  - 若它的左子树不空，则左子树上所有结点的值均小于根结点的值
  - 若它的右子树不空，则右子树上所有结点的值均大于根结点的值
  - 它的左、右子树也都分别是二叉排序树。

- 结论：若按中序遍历一棵二叉排序树，所得到的结点序列是一个递增序列。

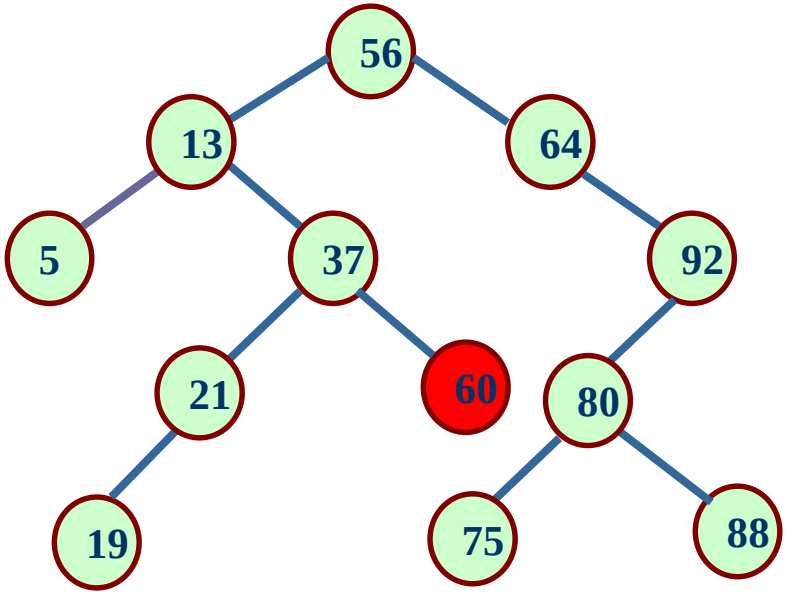
9.2 动态查找表

一. 二叉排序树

- 举例



二叉排序树



非二叉排序树

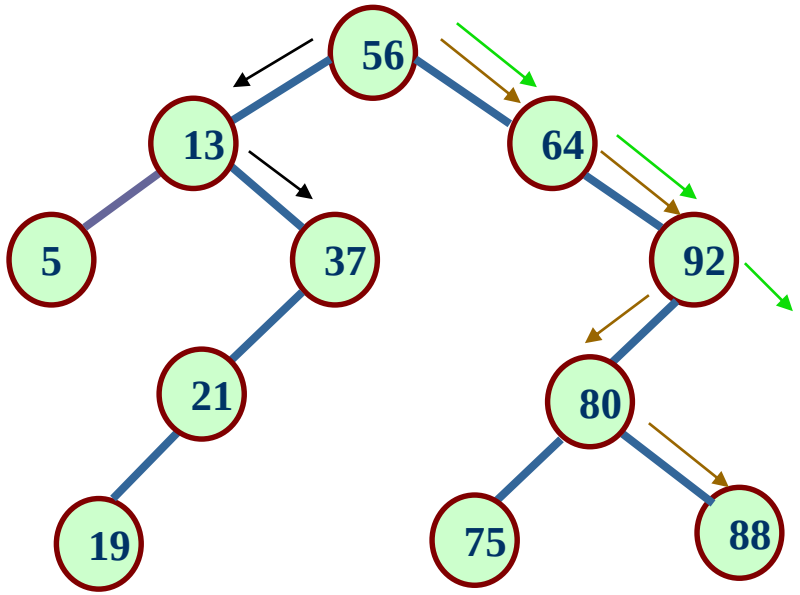
9.2 动态查找表

一. 二叉排序树

- 二叉排序树的查找：
  - 若查找树为空，则查找失败；
  - 若查找树为非空，则给定K值与根结点比较：

1. 若相等，查找成功
2. 若小于，继续在该结点的左子树上进行查找；
3. 若大于，继续在该结点的右子树上进行查找。

- 例如下图，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:

```
typedef  struct  Node
{
    KeyType  key ;    /*  关键字域  */
    …      /*  其它数据域  */
    struct  Node  *Lchild , *Rchild ;
}BSTNode ;
```

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

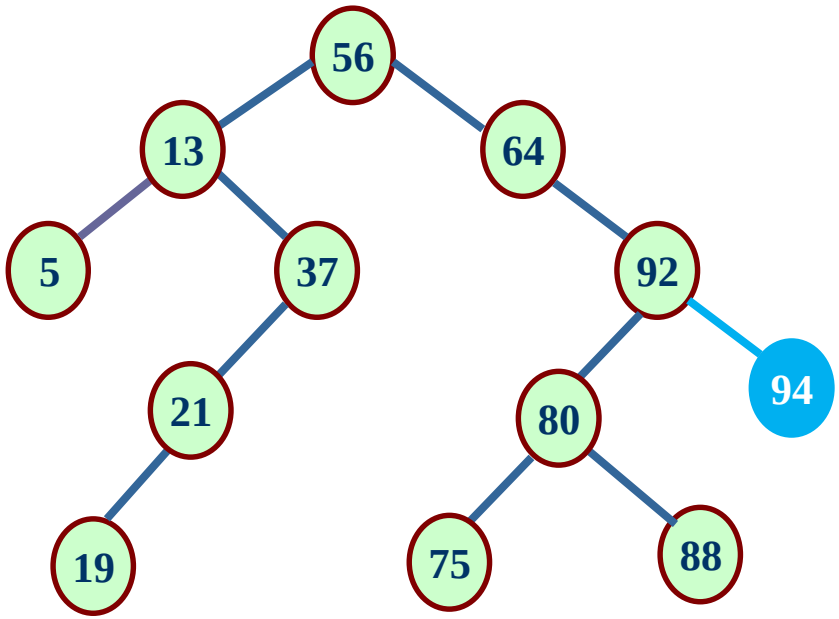
```
BSTNode *BST_Serach(BSTNode *T , KeyType key)
{
    if (T==NULL)  return(NULL) ;
    else
      {  if  (EQ(T->key, key) )
                    return(T) ;
        else if ( LT(key, T->key) )
                    return(BST_Serach(T->Lchild, key)) ;
        else
                    return(BST_Serach(T->Rchild, 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的左孩子或右孩子(新结点值小于或大于该结点值)

- 如下图，查找94不成功则插入

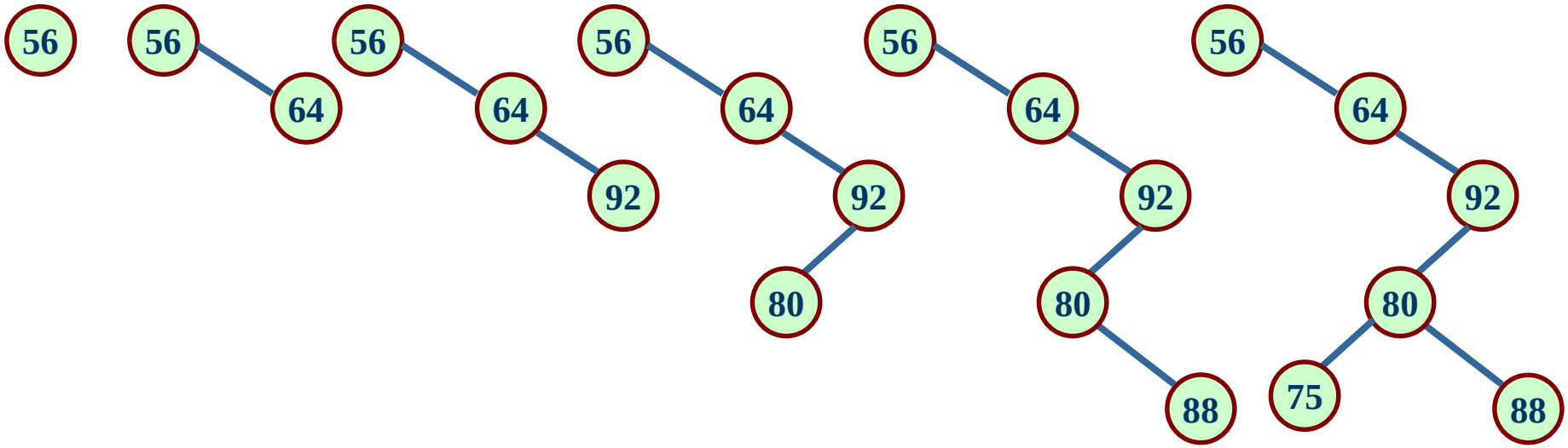


9.2 动态查找表

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)
{
        BSTNode *x ;
        x=(BSTNode *)malloc(sizeof(BSTNode)) ;
        x->key = key;
        x->Lchild=x->Rchild = NULL ;
        if (T==NULL)
          T = x ;
        else
        {  if (EQ(T->key, x->key) )
              retur  n  ; /*  已有结点  */
           else if (LT(x->key, T->key) )
              Insert_BST(T->Lchild, key) ;
           else
              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 ;
    scanf(“%d”, &key) ;
    while (  key != ENDKEY )
    {
        Insert_BST(T, key) ;
        scanf(“%d”, &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

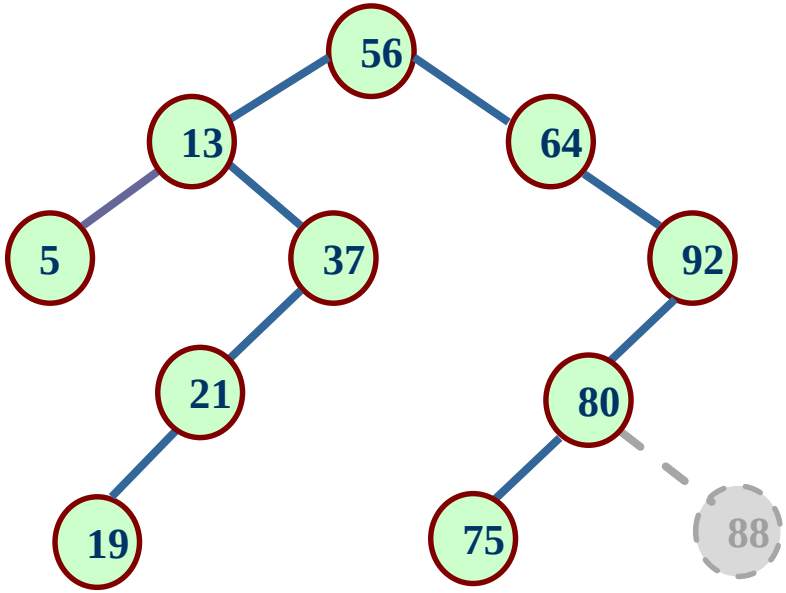
9.2 动态查找表

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

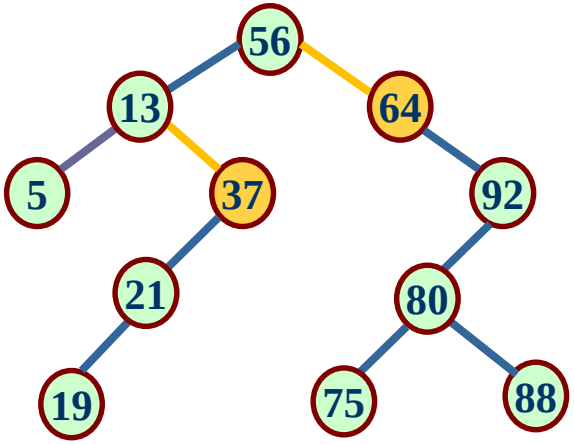
删除结点**88**



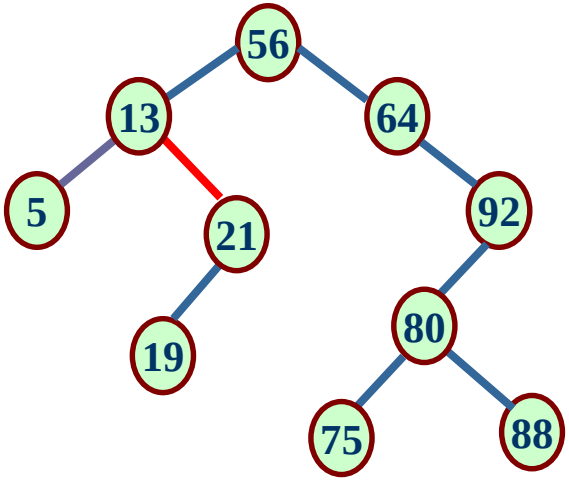
9.2 动态查找表

一. 二叉排序树

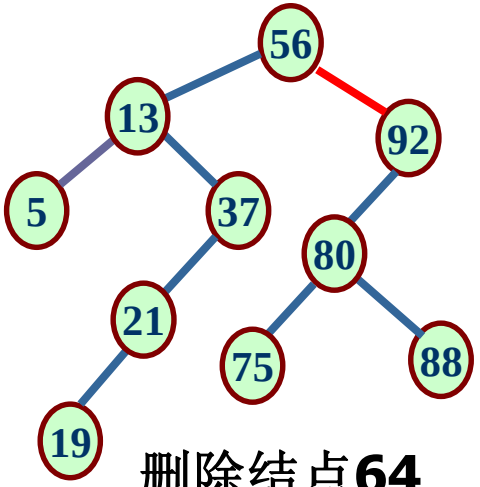
- 二叉树排序树的删除
  - 被删除结点只有左子树或右子树
  - 删除结点，让其父结点指向该结点的指针指向其左子树(或右子树)，即用孩子结点替代被删除结点即可



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

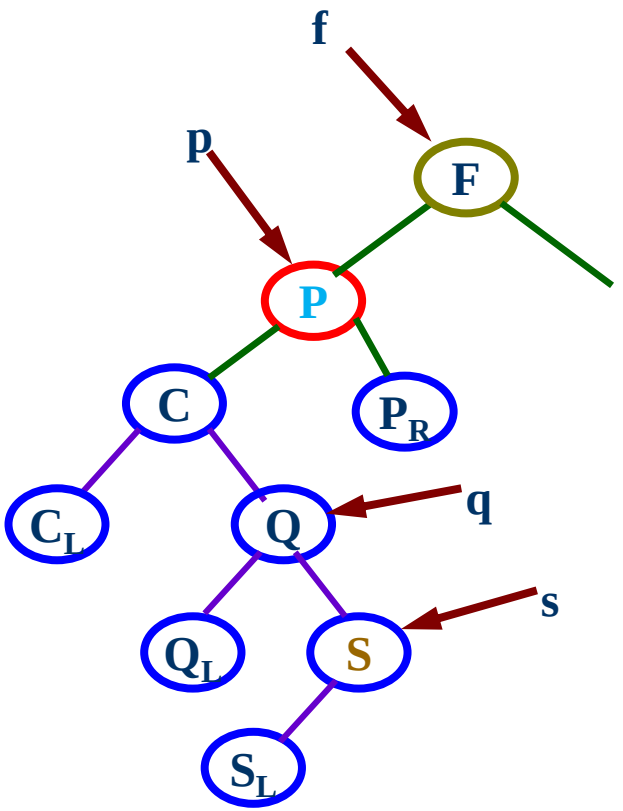
■ 二叉树排序树的删除

- 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（或直接后继）结点S替换被删除结点P，然后再删除该直接前驱（或直接后继）结点S
  - 用直接前驱替换的实质就是用当前结点P的左子树的最大右子孙S（ S是P的左子树的最右结点）替换P，然后再删除S
  - 用直接后继替换的实质就是用当前结点P的右子树的最小左子孙S（ S是P的右子树的最左结点）替换P，然后再删除S

9.2 动态查找表

一. 二叉排序树

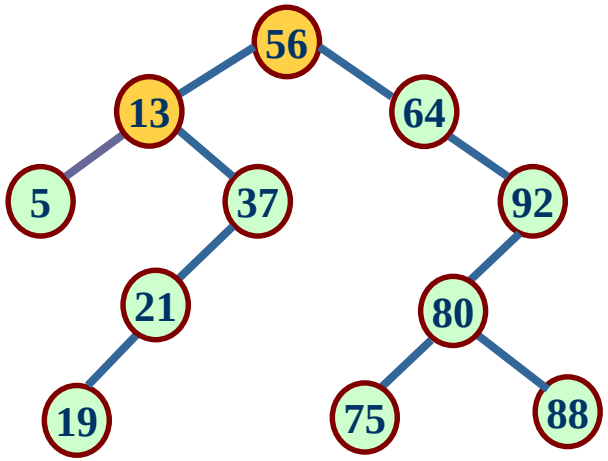
- 二叉树排序树的删除
  - 被删除结点P既有左子树，又有右子树
  - 以中序遍历时的直接前驱S替代被删除结点P，然后再删除该直接前驱S （S若有孩子，只可能有左孩子）
  - 实质就是用当前结点P的左子树的最大右子孙S（ S是P的左子树的最右结点）替换P，然后再删除S



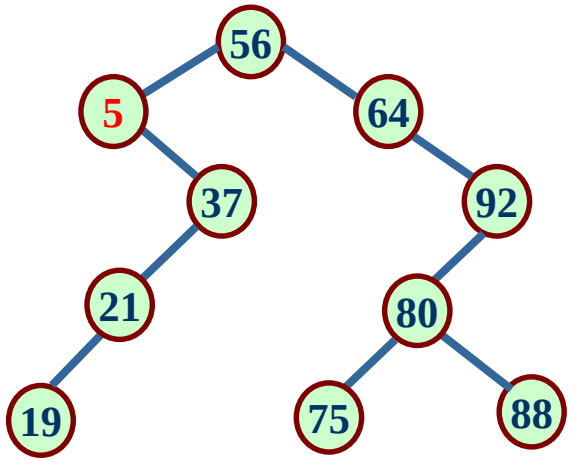
9.2 动态查找表

一. 二叉排序树

- 二叉树排序树的删除
  - 举例：

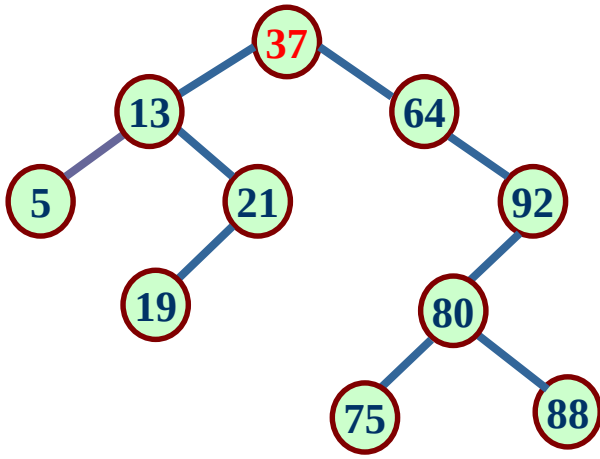


原图



删除结点**13**

用其左子树的
最右子孙替换



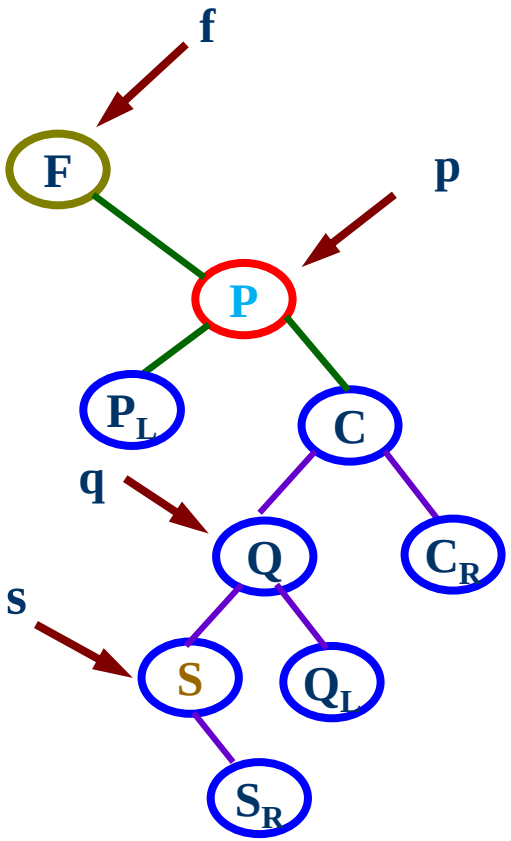
删除结点**56**

用其左子树的
最右子孙替换

9.2 动态查找表

一. 二叉排序树

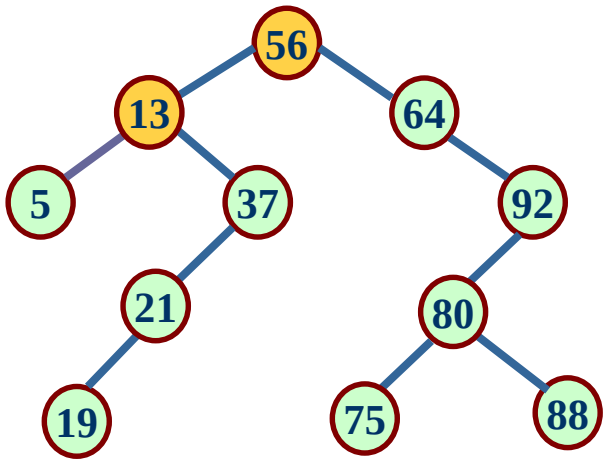
- 二叉树排序树的删除
  - 被删除结点P既有左子树，又有右子树
  - 以中序遍历时的直接后继S替代被删除结点P，然后再删除该直接后继S（S 若有孩子，只可能有右孩子）
  - 实质就是用当前结点P的右子树的最小左子孙S（S是P的右子树的最左结点）替换P，然后再删除S



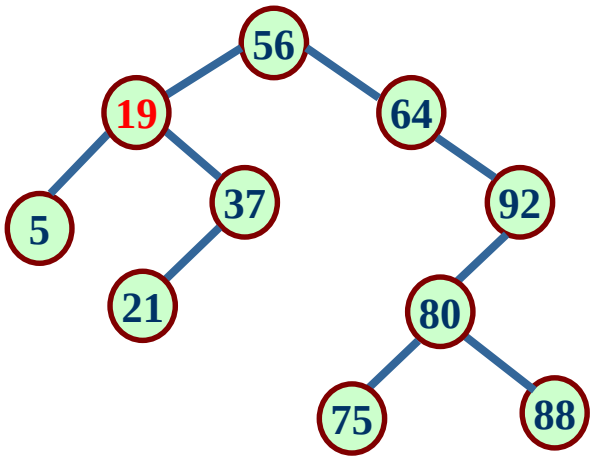
9.2 动态查找表

一. 二叉排序树

- 二叉树排序树的删除
  - 举例：

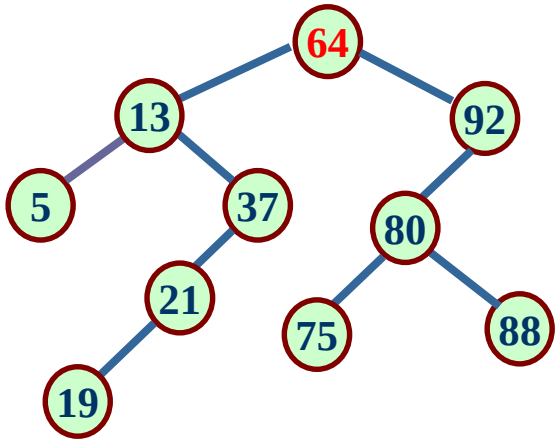


原图



删除结点13

用其右子树的
最左子孙替换



删除结点56

用其右子树的
最左子孙替换

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(BiTree &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 (EQ(key, T->data.key))  // 找到关键字等于key的数据元素
        return Delete(T);
    else if (LT(key, T->data.key)) return DeleteBST(T->lchild, key);
    else return DeleteBST(T->rchild, key);
  }
} // DeleteBST
```

9.2 动态查找表

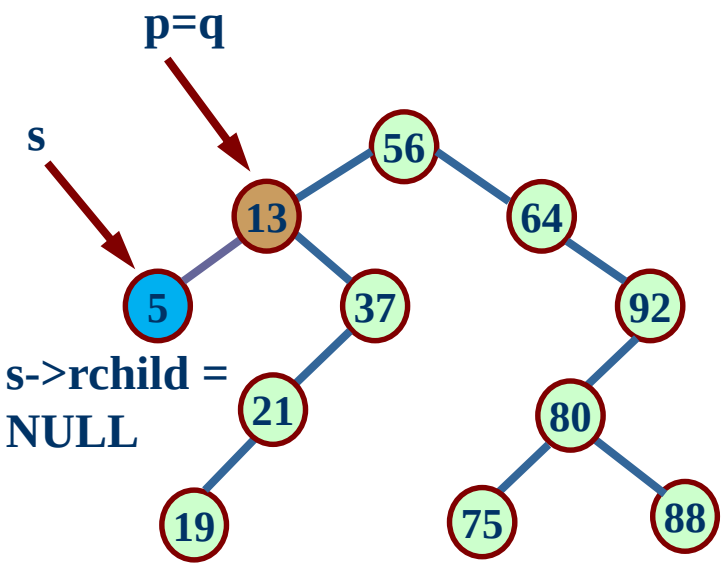
一. 二叉排序树

二叉树排序树删除的递归程序

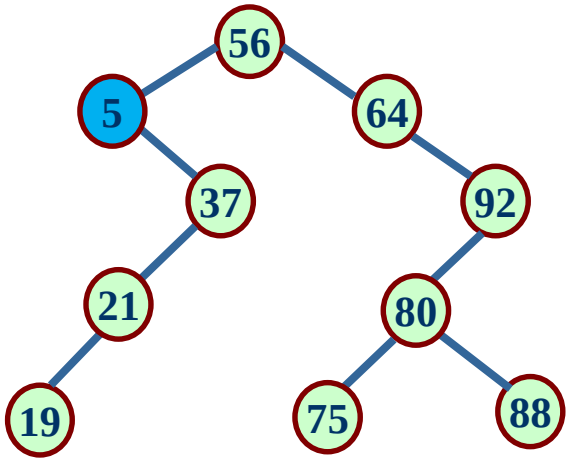
```
Status Delete(BiTree &p) // 删除结点p，并重接它的左或右子树
 {      BiTree q, s;
       if (!p->rchild)   // 右子树空,则只需重接它的左子树
       {  q = p;  p = p->lchild;  free(q);   }
        else if (!p->lchild)   // 左子树空,只需重接它的右子树
       {  q = p;  p = p->rchild;  free(q);  }
        else {  // 左右子树均不空
             q = p;  s = p->lchild;
             while ( s->rchild )   // 转左，然后向右到尽头
                       { q = s;  s = s->rchild; } //此时，q是s的父结点
             p->data = s->data;                  // s指向被删结点的替换结点
             if (q != p)  q->rchild = s->lchild;  // 重接*q的右子树
             else  q->lchild = s->lchild;        // 重接*q的左子树
             free(s);
       }
       return TRUE;
} // Delete
```

9.2 动态查找表

```
// 左右子树均不空
    q = p;  s = p->lchild;
    while ( s->rchild )
      { q = s;  s = s->rchild; }
    p->data = s->data;                  // s指向被删结点的替换结点
    if (q != p)  q->rchild = s->lchild;  // 重接*q的右子树
    else  q->lchild = s->lchild;         // 重接*q的左子树
    free(s);
```



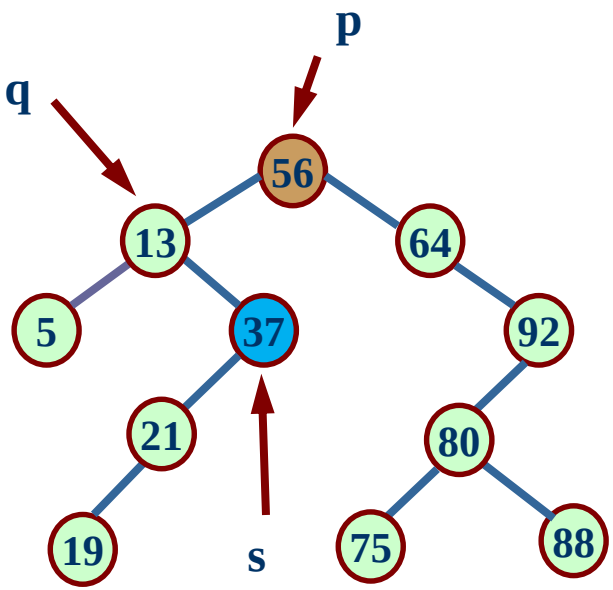
原图

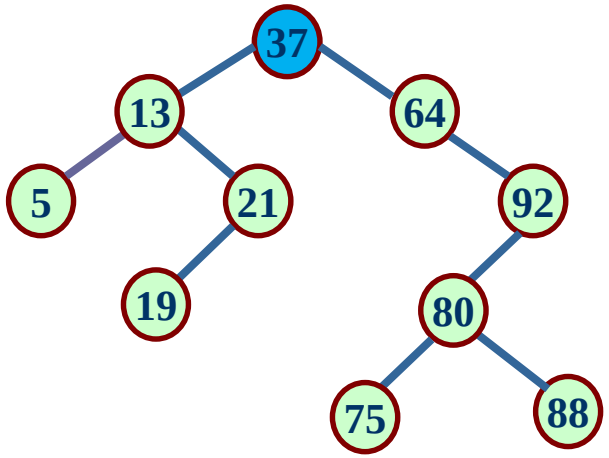


删除结点13

9.2 动态查找表

```
// 左右子树均不空
    q = p;  s = p->lchild;
    while ( s->rchild )
     { q = s;  s = s->rchild; } //此时，q是s的父结点
    p->data = s->data;                  // s指向被删结点的替换结点
    if (q != p)  q->rchild = s->lchild;  // 重接*q的右子树
    else  q->lchild = s->lchild;         // 重接*q的左子树
    free(s);
```





删除结点56

9.2 动态查找表

一. 二叉排序树

- 算法性能
  - 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，保持了链式存储结构在执行插入或删除操作时不用移动数据元素的优点，只要找到合适的插入和删除的位置后，仅需要修改链接指针即可，插入和删除的时间性能比较好。

9.2 动态查找表

一. 二叉排序树

■ 算法性能

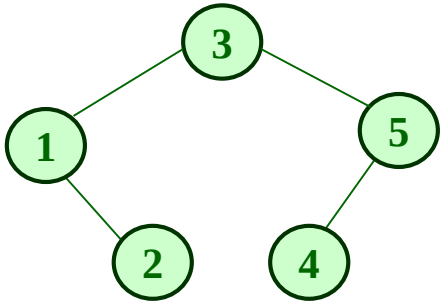
对于每一棵特定的二叉排序树，均可按照平均查找长度的定义来求它的 ASL 值，显然，由值相同的 n 个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长度的值不同，甚至可能差别很大。

例如：

由关键字序列 3，1，2，5，4 构造而得的二叉排序树，

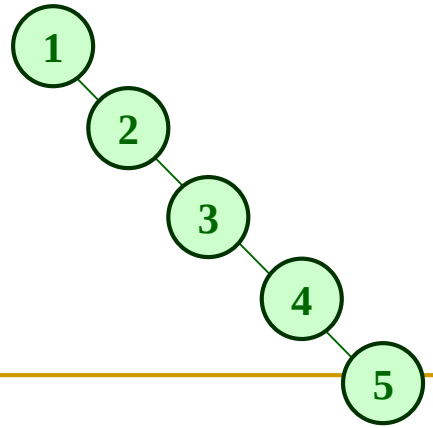
ASL =（1+2+3+2+3）/ 5

= 2.2



由关键字序列 1，2，3，4，5 构造而得的二叉排序树，

ASL =（1+2+3+4+5）/ 5 = 3

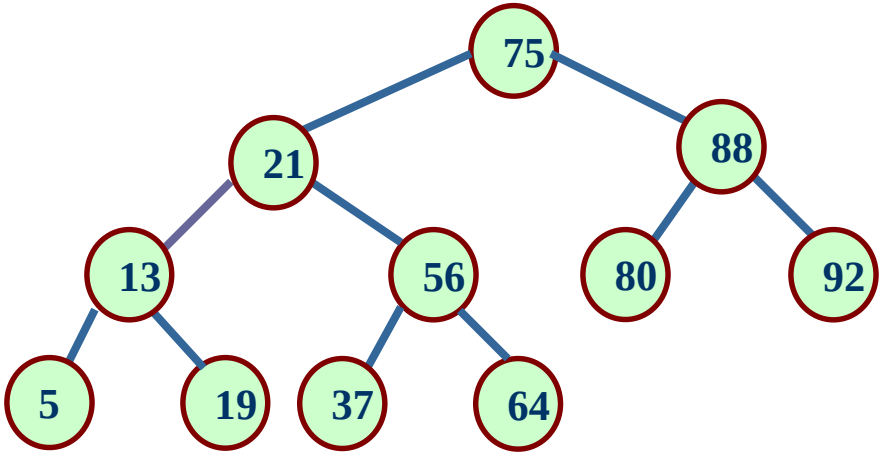


9.2 动态查找表

一. 二叉排序树

- 算法性能
  - 二叉排序树的查找就是根结点到要查找的结点的路径，其比较次数等于给定值的结点在二叉排序树的层次

在最好的情况下，二叉排序树为一棵近似完全二叉树时，其查找深度为$\log_2 n$量级，即其时间复杂度为$O(\log_2 n)$

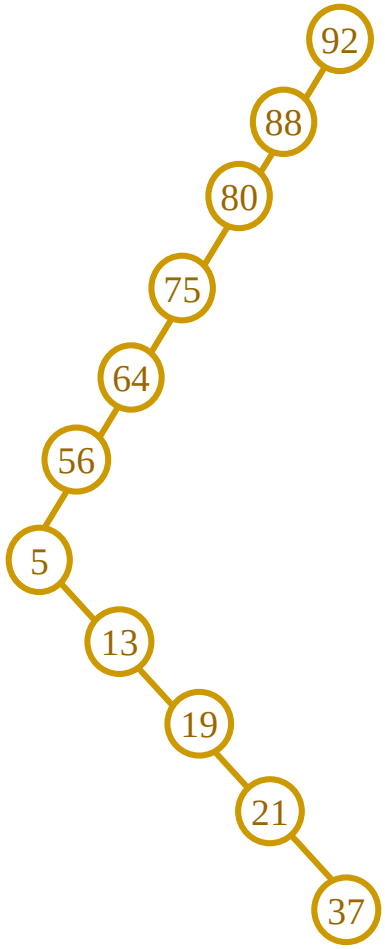


9.2 动态查找表

一. 二叉排序树

- 算法性能
  - 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，增加了查找的时间

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数量级，即其时间复杂度为0(n)



练习

一. 已知数据序列为33、66、22、88、11、27、44、55，把该数列依次插入到初始为空的二叉排序树中

- 请画出最终生成的二叉排序树
- 若查找27和50，请写出每个数据的查找次数，并画出查找后的二叉排序树
- 在执行完前面查找后，逐个删除结点11、27、66，请画出删除后的二叉排序树