数 据 结 构

第六、七、十章 搜索篇 [2] 二叉搜索树

任课老师：郭艳

数据结构课程组

计算机学院

中国地质大学（武汉）2020年秋

上堂课要点回顾

- 字典
  - 字典概念 <名字-属性>对的集合
  - 抽象数据类型
    - class **dataList**: Insert/Remove/SeqSearch方法
    - class **searchList**
  - 字典实现：有序链表、（有序顺序表）、跳表
- 静态搜索
  - **顺序搜索SeqSearch**
    - **$O(n)$；$ASL_{suc}=(n+1)/2$, $ASL_{unsuc}=n+1$**
    - 优缺点
  - 有序顺序表的折半搜索**BiSearch**
    - **$O(\log_2 n)$**
    - 优缺点
  - 跳表（自学）
  - 索引顺序表的分块查找 $O(\sqrt{n})$

第十三次课

阅读：

殷人昆，第279-293页

习题：

作业13

7.2 二叉搜索树

- 二叉搜索树的概念
- 二叉搜索树的搜索
- 二叉搜索树的插入
- 二叉搜索树的删除
- 二叉搜索树性能分析

搜索树的引入

- 分析可用于描述字典的数据结构
  - 线性表：搜索/插入/删除的平均时间为O(n)

基本结构	搜索	插入/删除
基于数组的无序搜索表	0(n)	插入0(1)，删除0(n)
基于数组的有序搜索表	0(logn)	0(n)
无序链表	o(n)	插入0(1)，删除0(n)
有序链表	0(n)	0(n)

  - 跳表：搜索/插入/删除的平均时间为$O(\log_2 n)$，而最坏情况下的时间为O(n)
  - 散列表：平均和最坏时间分别为O(1)和O(n)，可适用于根据元素关键码进行的操作
  - 后两者比较：使用跳表很容易对字典元素进行高效的顺序访问（如按照升序搜索元素），而散列表却做不到这一点。

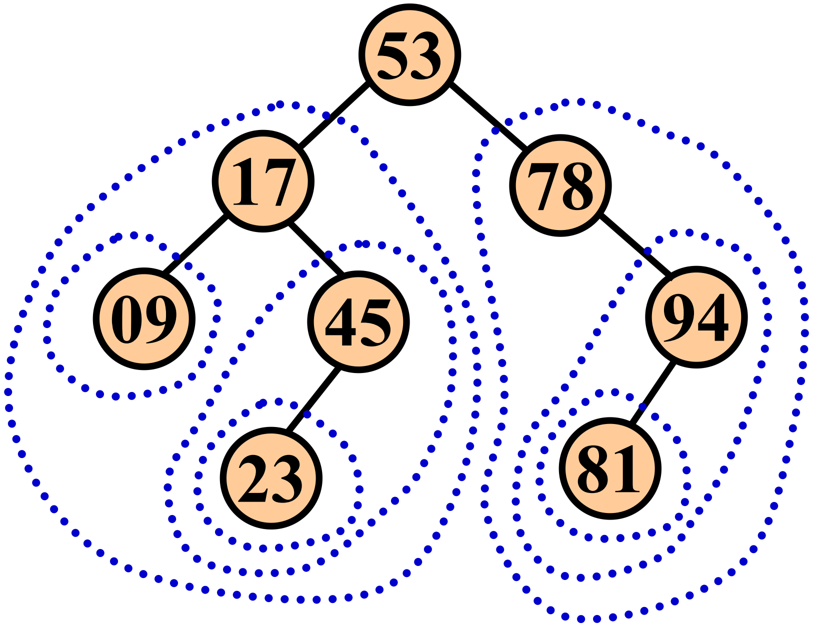
搜索树的引入

- 用平衡搜索树表示字典结构
  - 对n个元素的字典进行搜索、插入或者删除所需要的平均和最坏时间均为$O(\log_2 n)$
  - 所有字典元素能够在线性时间内按升序输出
  - 搜索树既适用于根据元素关键码的操作，也适用于不按精确的关键码匹配进行字典操作的应用（比如寻找关键码大于k 的最小元素）

7.2.1 二叉搜索树的概念

- **二叉搜索树（Binary Sort Tree/Binary Search Tree，BST）**或者是一棵空树，或者是具有下列性质的二叉树：
  - 每个结点都有一个作为搜索依据的关键码(key)，所有结点的关键码互不相同；
  - 左子树（如果存在）上所有结点的关键码都小于根结点的关键码；
  - 右子树（如果存在）上所有结点的关键码都大于根结点的关键码；
  - 左子树和右子树也是二叉搜索树。

二叉搜索树示例



- 对所有非终端结点满足：
  - 结点左子树上所有结点关键码小于该结点关键码
  - 结点右子树上所有结点关键码大于该结点关键码

- 二叉搜索树的性质：中序遍历该树可以按从小到大的顺序将各个结点的关键码排列起来。所以二叉搜索树也称为二叉排序树。（判断一棵二叉树是否为二叉搜索树的方法！）
- 注意：若从根结点到某个叶结点有一条路径，则路径上经过的结点的关键码不一定构成一个有序序列。

二叉搜索树的类定义

- 二叉搜索树采用二叉链表作为存储表示，但由于结点常用关键码表征，因此在二叉树结点类定义中应增加一些**重载操作**，用以定义元素之间以及元素与关键码之间的比较。
- 二叉搜索树类也可以定义为二叉树类的派生类，但它有自己**特有操作**，如求最小值、最大值，另外搜索、插入和删除操作的含义也不同。
- 二叉搜索树的类定义：完整定义见书P309-310 程序7.9

```
template <class E, class K>     //E是元素类，K是关键字域类
struct BSTNode {                //二叉搜索树结点类
    E data;
    BSTNode<E, K> *left, *right;
    BSTNode (E d, BSTNode<E,K> *l = NULL, BSTNode<E,K> *r = NULL){ data = d;left= l;  right = r; }
    bool operator > (BSTNode<E,K> right){return data.key>right.data.key;}
      //元素根据关键码值判断>、<、==的重载操作符
      //其它函数成员：略
}
```

二叉搜索树的类定义（续）

```
template <class E, class K>
class BST {                          //二叉搜索树类定义
public:  BST(K value);               //构造函数，value是结束标志
         ~BST(){};
         bool Search(const K x) const
            { return (Search(x, root)!=NULL) ? true : false;  }
         //其他函数：略
private: BSTNode<E, K> *root;                 //二叉搜索树的根指针
     BSTNode<E, K> *Search(const K x, BSTNode<E, K> *ptr) ;//搜索
     BSTNode<E, K> *Min(BSTNode<E, K> *ptr) const;//求最小
     BSTNode<E, K> *Max(BSTNode<E, K> *ptr) const;//求最大
     bool Insert(const E& e1, BSTNode<E, K> *&ptr) ;  //递归：插入
     bool Remove(const K x, BSTNode<E, K> *&ptr) ;   //递归：删除
     //其他函数：略
};
```

7.2.2 二叉搜索树的搜索

- 在二叉搜索树上的搜索，是一个从根结点开始，沿某一个分支逐层向下进行比较判等的过程。
- **思想（减而治之策略，仿效有序查找表的折半查找）**

  假设要搜索关键码为x的元素，先从根结点开始，如果根指针为空，则**搜索失败**；否则将给定值x与根结点的关键码进行比较：

  - 如果x**等于**根结点的关键码，则**搜索成功**，返回搜索到的结点地址；
  - 如果x**小于**根结点的关键码，则在左子树中继续搜索；
  - 如果x**大于**根结点的关键码，则在右子树中继续搜索。
- 二叉搜索树的**效率**就在于只需搜索两个子树之一。

BST的搜索算法

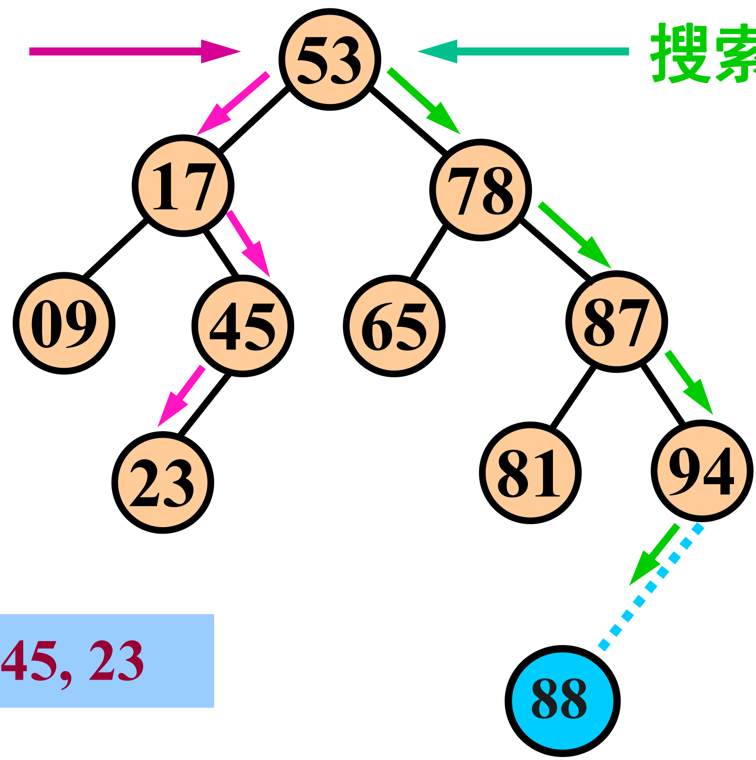
- 与折半搜索类似，也可以写迭代或递归算法。

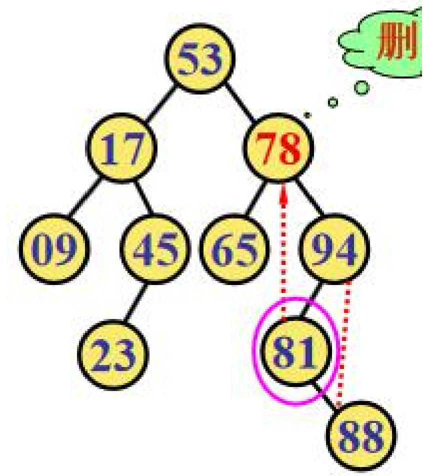
```
template <class E, class K>
BSTNode<E, K> *BST<E, K> :: Search( const K x,
BSTNode<E, K> *ptr) {
 //私有函数：在以ptr为根的二叉搜索树中递归搜索结点x
    if ( ptr == NULL ) return NULL;      //搜索失败
    else if (x == ptr->data)
            return ptr;                  //相等，搜索成功
        else if (x < ptr->data )         //在左子树搜索（重载）
            return Search( x, ptr->left );
        else          //在右子树搜索
            return Search( x, ptr->right );
}
```

BST的搜索过程示例

搜索23

搜索88





搜索成功！

搜索路径：53, 17, 45, 23

搜索失败！

如果是插入操作，此时88作为94的左孩子插入！

7.2.3 二叉搜索树的插入

- 思想
  - 若要在二叉搜索树中插入一个新元素，首先要使用搜索算法检查该元素在树中是否存在；
  - 如果搜索成功，树中已有这个元素，不再插入；

- 如果搜索不成功，则生成新元素结点，（保持二叉搜索树的性质不丢失）把新结点作为叶结点插入到搜索操作停止的地方。

BST的插入算法（递归）

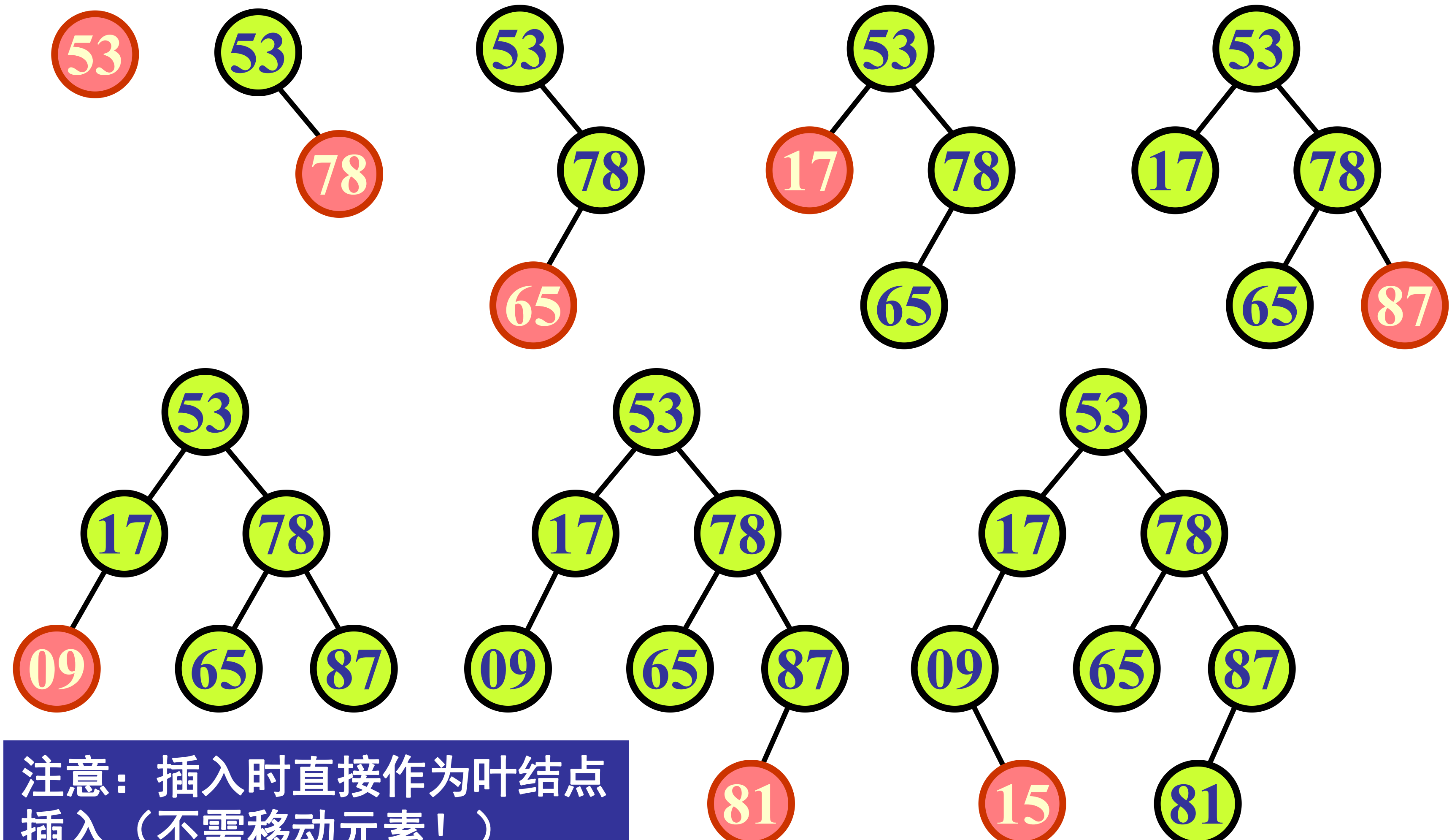
```
template <class E, class K>
bool BST<E, K>::Insert( const E& e1, BSTNode<E, K> *&ptr)
{ //私有函数：在以ptr为根的二叉搜索树中插入元素e1,
  //若树中已存在该元素结点，则不插入
    if ( ptr == NULL ) {                      //新结点作为叶结点插入
        ptr = new BSTNode<E, K> (e1);    //创建新结点
                                   //如果BST为空，则新结点作为根结点
        if ( ptr == NULL ) { cout << “Out of space !” << endl;  exit (1); }
        return true;
    }
    else  if(e1==ptr->data)  return false;    //查找相等返回false
          else if (e1<ptr->data) Insert(e1, ptr->left); //在左子树插入
          else Insert(e1, ptr->right); //在右子树插入
}//思考：非递归算法?
```

BST的构造

- 已知一个元素关键码序列，利用二叉搜索树的插入算法，可以很方便地建立一棵二叉搜索树。
- **思想**：从空的二叉搜索树开始，每输入一个数据，就调用二叉搜索树的插入算法，将新结点插入到树中，直至输入结束标志。

BST的构造-示例

输入元素的关键码序列 { 53, 78, 65, 17, 87, 09, 81, 15 }，建立二叉搜索树的过程：



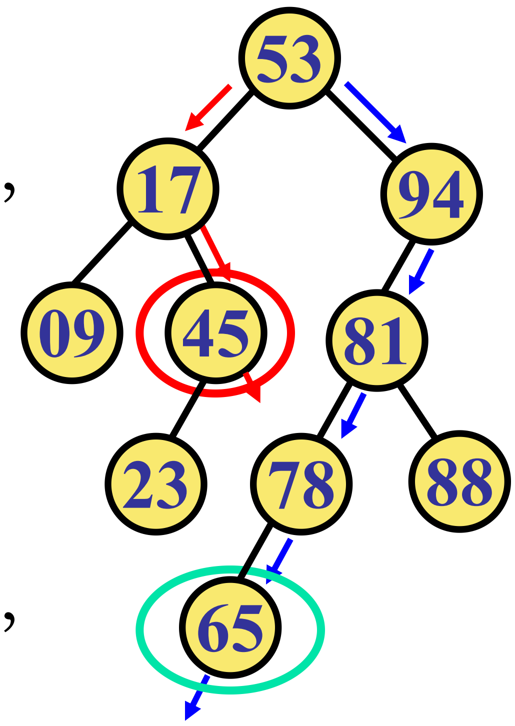
注意：插入时直接作为叶结点插入（不需移动元素！）

BST的构造算法

```
template <class E, class K>
BST<E, K> :: BST ( ) {
//输入元素序列，建立一棵二叉搜索树。RefValue 是输入
//结束标志。这个值应取不可能在输入序列中出现的值，
//例如输入序列的值都是正整数时，取RefValue为0或负数。
    E x;
    root = NULL;  RefValue = value;          //置空树
    cin >> x;                                //输入数据
    while ( x != RefValue )                  //不是结束标志
       { Insert ( x, root );  cin >> x; }    //插入，继续输入
}
```

在BST中搜索子树的最值元素

- 查找左子树中的最大元素:
  - 首先移动到左子树的根,
  - 然后沿着各结点的右孩子指针移动,
  - 直到右孩子指针为NULL为止;
- 查找右子树中的最小元素:
  - 首先移动到右子树的根,
  - 然后沿着各结点的左孩子指针移动,
  - 直到左孩子指针为NULL为止。



7.2.4 二叉搜索树的删除

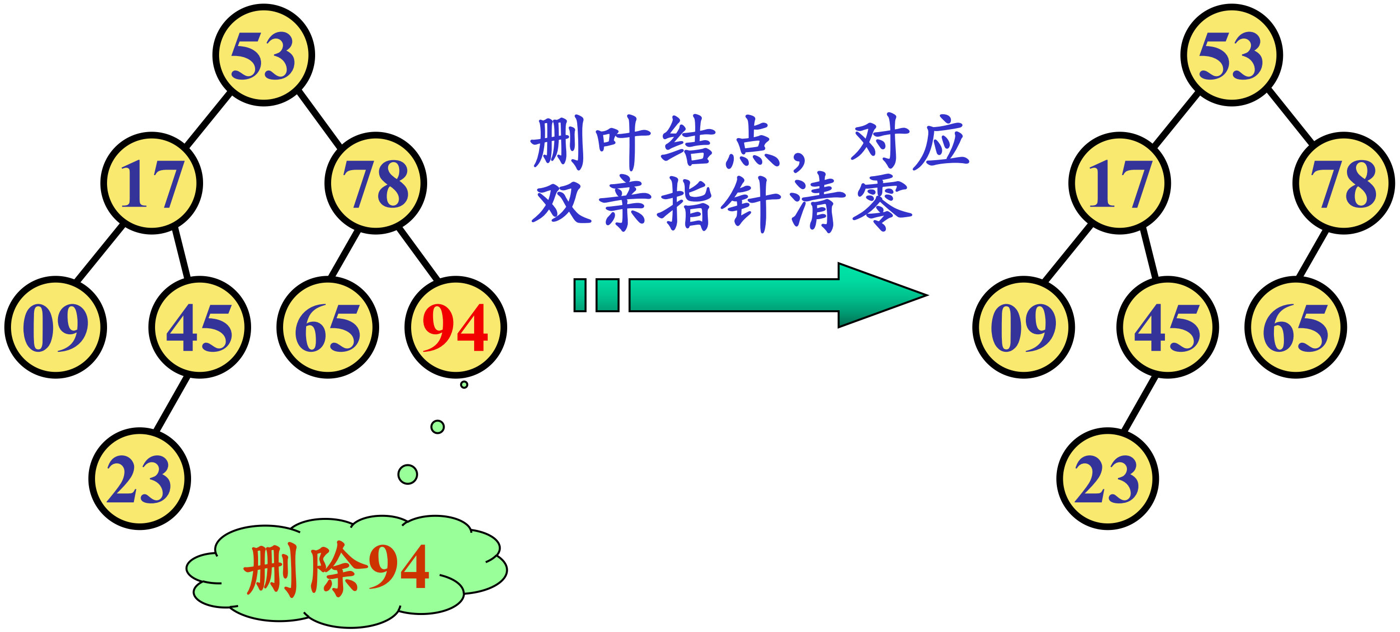
- 在二叉搜索树中删除一个元素结点时，首先也要检查该元素在树中是否存在；当搜索成功时，才能进行删除。
- 在删除过程中，必须将因删除结点而断开的二叉链表重新链接起来，同时**确保二叉搜索树的性质不会失去**。
- 为保证在删除后树的搜索性能不至于降低，还需要防止重新链接后树的高度增加。

BST结点的删除

- 被删结点具有以下四种情况：
  - 1）被删结点是叶结点
  - 2）被删结点只有左子树（右子树空）
  - 3）被删结点只有右子树（左子树空）
  - 4）被删结点左、右子树都不空

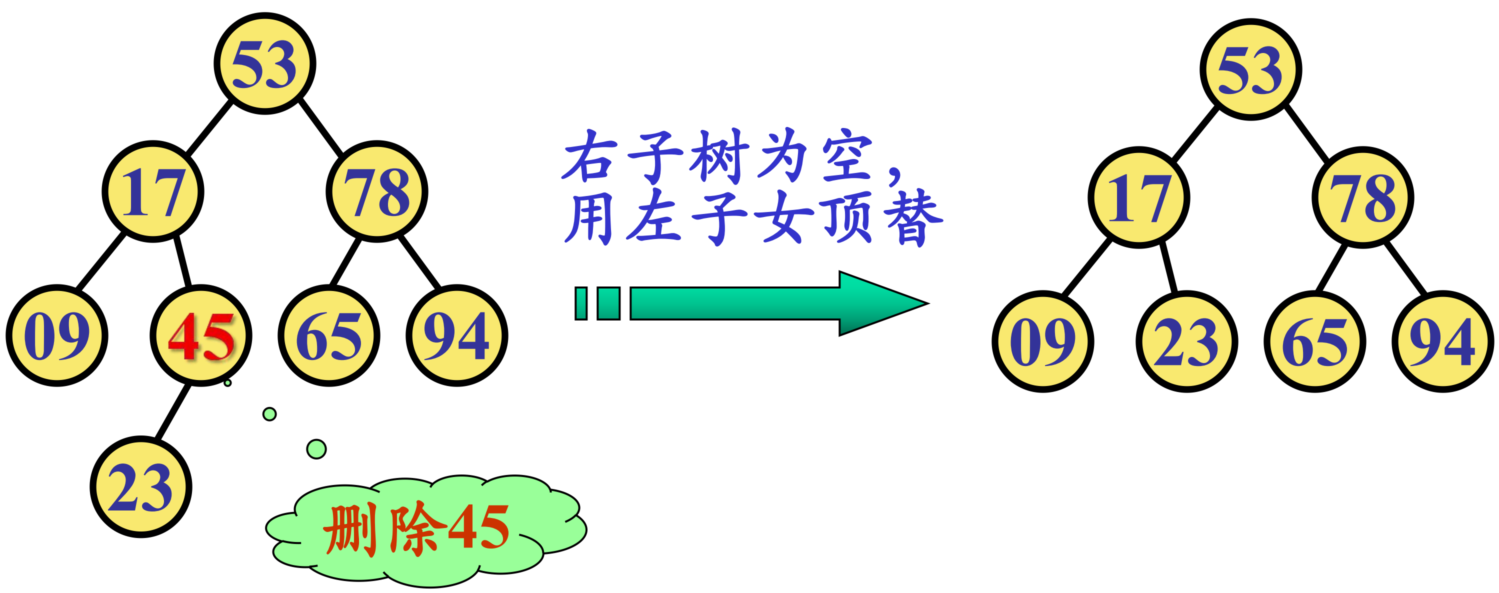
BST结点的删除-情况1

- 1）被删结点是叶结点。只需将其双亲结点指向它的指针清零，再释放它即可。



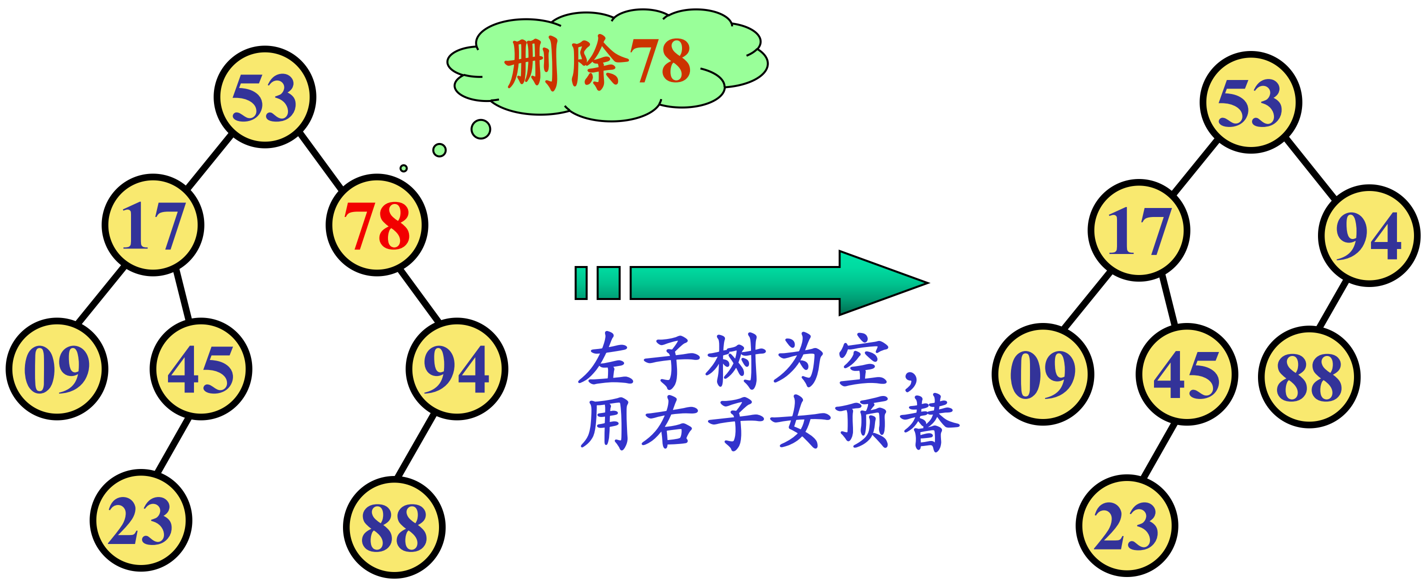
BST结点的删除-情况2

- 2）被删结点只有左子树。将其双亲结点指向它的指针改为指向它的左子女结点，再释放它。



BST结点的删除-情况3

- 3）被删结点只有右子树。将其双亲结点指向它的指针改为指向它的右子女结点，再释放它。



BST结点的删除-情况4

- 4）被删结点左、右子树都不空。在它的右子树中寻找中序下的第一个结点（关键码最小），或者在它的左子树中寻找中序下的最后一个结点（关键码最大），用它的值填补到被删结点中，再来处理这个结点的删除问题（递归处理）。

寻找结点78在中序下的直接前驱；用65顶替78。
寻找结点78在中序下的直接后继；用81顶替78。

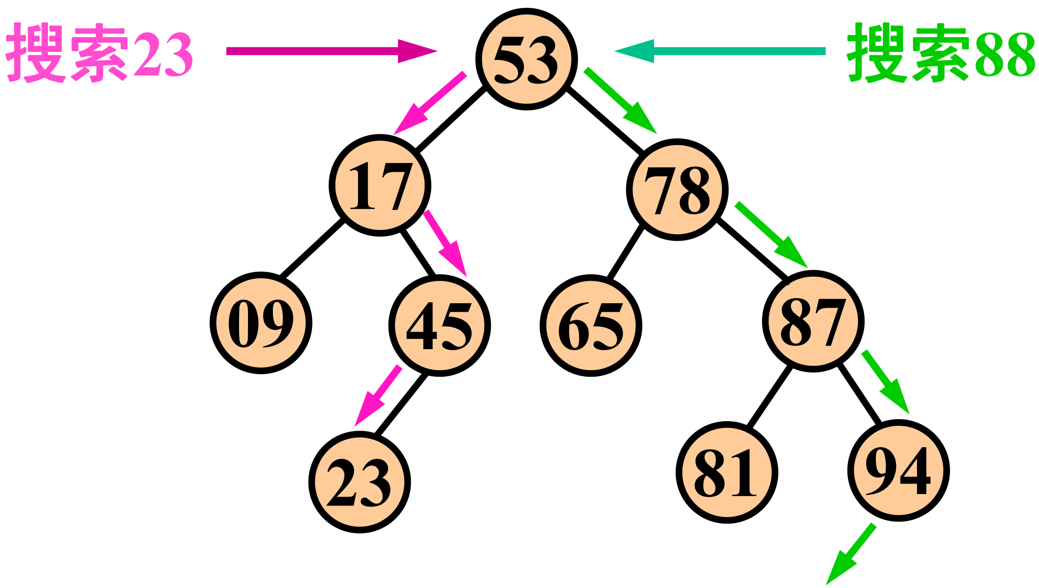
BST的删除算法

```
template <class E, class K>
bool BST <E, K>::Remove (const K x, BSTNode<E, K> *&ptr)
{ //私有递归函数：在以ptr为根的二叉搜索树中删除含x的结点，
 //若删除成功，则新根通过ptr返回。
    BSTNode<E, K> *temp;
    if ( ptr == NULL )
            return false;
    else
        if ( x < ptr->data )
            Remove ( x, ptr->left );      //在左子树中继续搜索、删除
        else if ( x > ptr->data )
            Remove ( x, ptr->right );  //在右子树中继续搜索、删除
        else if ( ptr->left != NULL && ptr->right != NULL ) {
            //找到要删除的关键码为x的结点，并且该结点有两个子女
```

BST的删除算法（续）

```
            temp =  ptr->right;    //找ptr右子树中序下的第一个结点
            while ( temp->left != NULL )  temp = temp->left;
            ptr->data = temp->data;   //用该结点数据替换ptr的数据
            Remove ( ptr->data, ptr->right );/*在ptr的右子树中删除具有一
样值的结点*/
        }
      else { // ptr指示的要删除的结点只有一个或零个子女
             temp = ptr;
             if ( ptr->left == NULL )
                      ptr = ptr->right;     //只有右子女或零个子女
             else ptr = ptr->left;         //只有左子女
             delete temp;
             return true;
        }
}
```

7.2.5 二叉搜索树性能分析



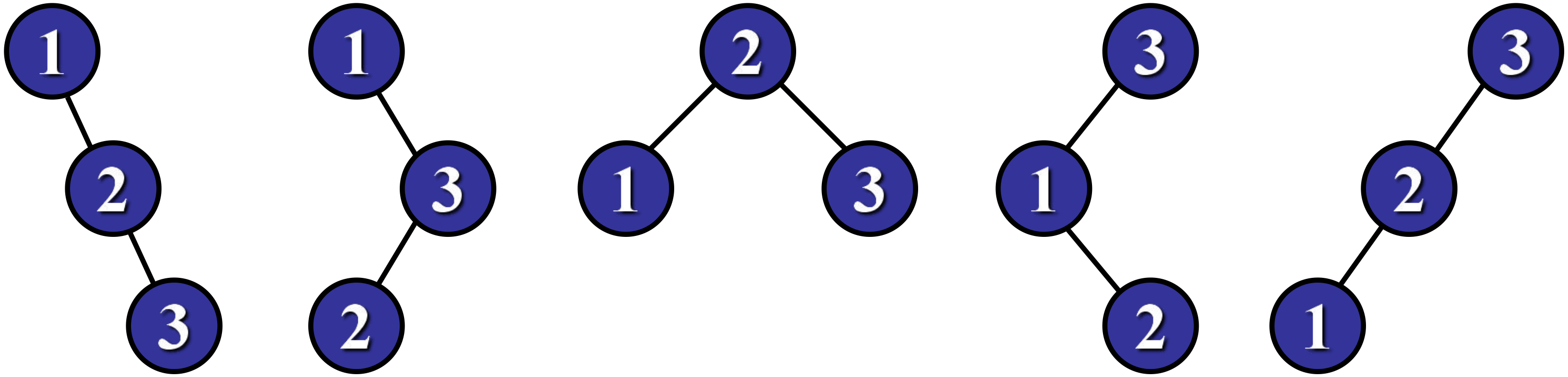
- 显然，二叉搜索树搜索时的最大比较次数取决于树的高度，为O(h)。但由于二叉搜索树是根据输入序列动态生成的，如果输入序列选得不好，会建立起一棵单支树，使得二叉搜索树的高度达到最大。

BST性能分析（续）

例如，同样 3 个数据{ 1, 2, 3 }，**输入顺序**不同，建立起来的二叉搜索树的形态也不同，从而直接影响到二叉搜索树的性能。

{2, 1, 3}

{1, 2, 3}　{1, 3, 2}　{2, 3, 1}　{3, 1, 2}　{3, 2, 1}

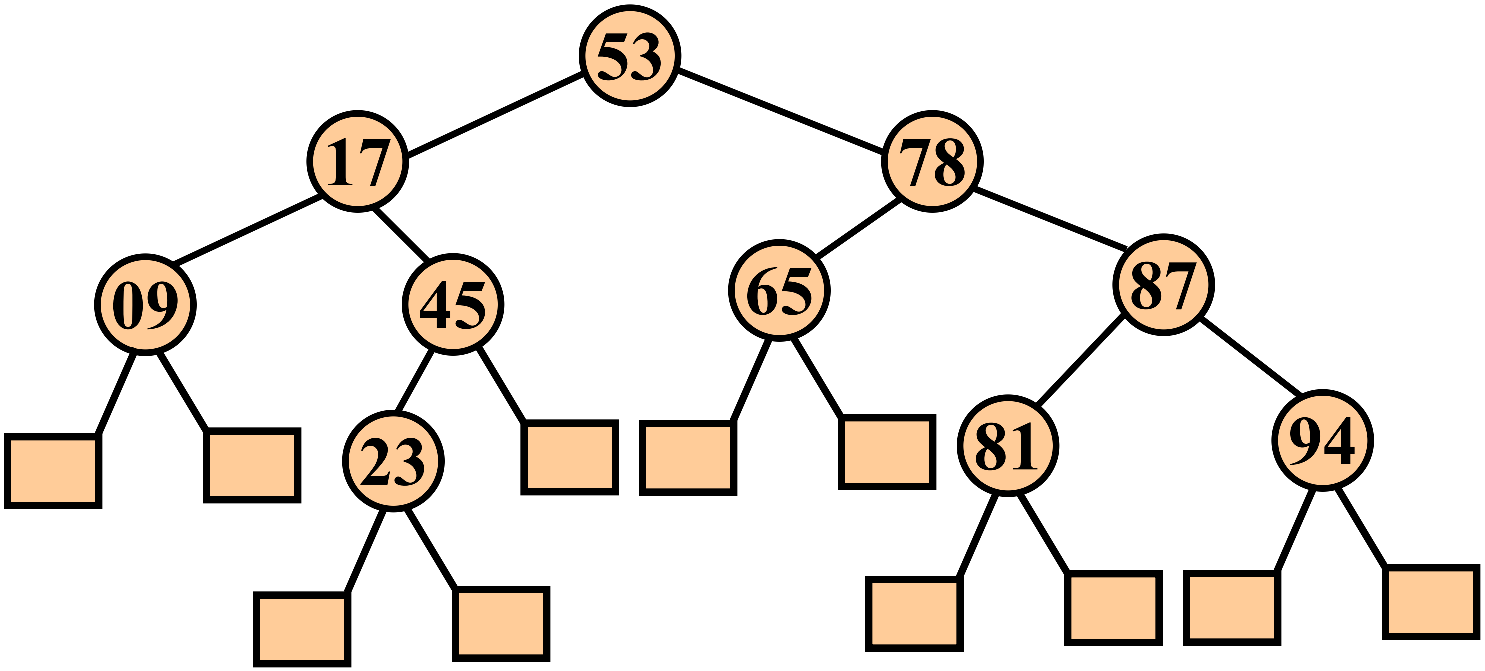


BST性能分析（续）

- 含有n个结点的二叉搜索树的高度最大为n，最小为 $\lceil \log_2(n+1) \rceil$
- 有序插入时形成一棵单支树，此时是最坏情况，对树的搜索、插入和删除操作所需要的时间均为O(n)。
- 在随机情况下，搜索、插入和删除操作的平均时间是 $O(\log_2 n)$。（证明见P317）

BST性能分析（续）

- 利用扩充二叉搜索树计算搜索成功和失败ASL值。



$$ASL_{succ} = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{10}(1\times1+3\times2+5\times4+7\times3) = 4.8$$

$$ASL_{unsucc} = \frac{1}{n+1}\sum_{j=1}^{n+1} C_j = \frac{1}{11}(5\times6+8\times6) = \frac{78}{11} = 7.07$$

二叉搜索树总结

- 组织内存索引
  - 二叉搜索树是适用于内存储器的一种重要的树形索引
  - 外存常用B/B$^+$树
- 保存性质 vs. 保存性能
  - 插入新结点或删除已有结点，要保证操作结束后仍符合二叉搜索树的定义
- 定义不允许出现重复关键码，但在实际应用中可以扩展此定义。允许有重复关键码时：
  - 重复关键码应有规律地出现，例如，右子树
    注意：搜索、插入、删除都要一致
  - 删除时，应删右子树中最小

作业13

概念题：

1、P342 7.8

电子作业：

2、实现二叉搜索树插入的非递归算法