第14章

搜索树

本章内容：

- **14.1 定义**
- **14.2 抽象数据类型**
- **14.3 二叉搜索树的操作和实现**
- 14.4 带有相同关键字元素的二叉搜索树
- 14.5 索引二叉搜索树
- 14.6 应用

搜索树

- 搜索树(Search Trees) 是一种适合于描述字典的树形结构 。
  - 比跳表和散列表有更好或类似的性能
  - 特别是对于顺序访问或按排名访问，散列表实现时间性能差，使用搜索树实现则会有更好的时间性能。

14.1 定义

- 14.1.1 二叉搜索树定义
- 有重复值的二叉搜索树
- 14.1.2 索引二叉搜索树

14.1.1 二叉搜索树定义

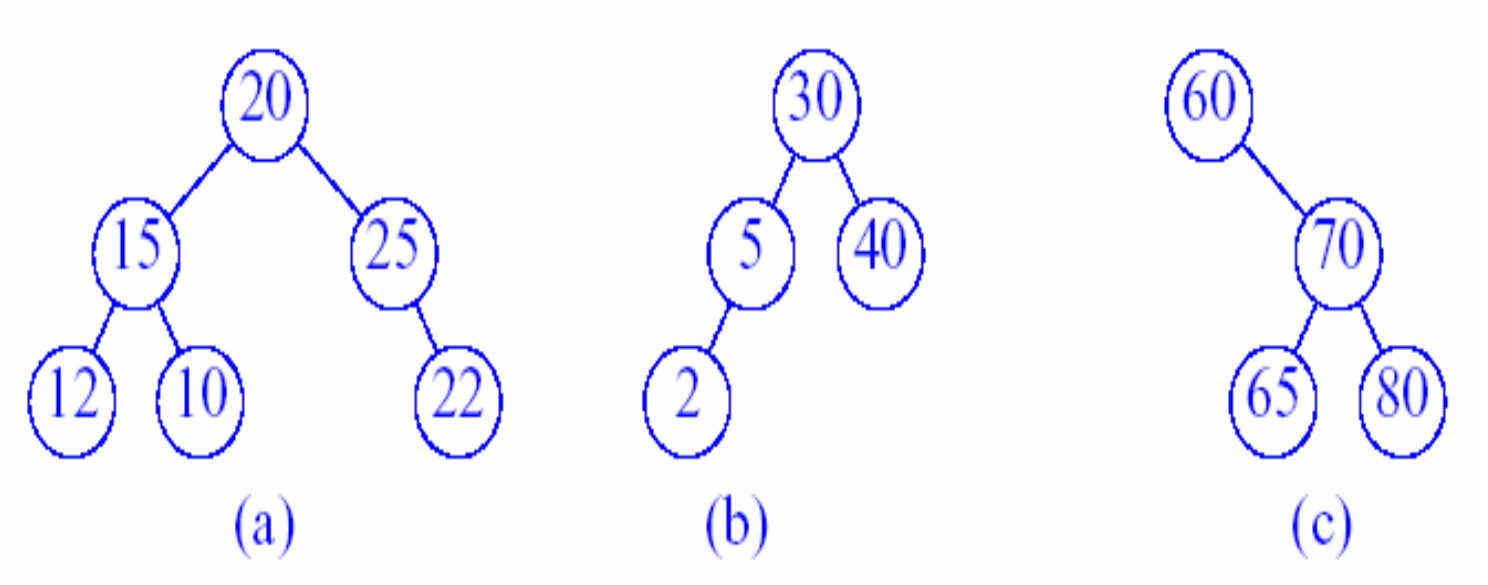
- 定义14-1[二叉搜索树(binary search tree)]：
- 二叉搜索树是一棵可能为空的二叉树，一棵非空的二叉搜索树满足以下特征：

1. 每个元素有一个关键字，并且没有任意两个元素有相同的关键字；因此，所有的关键字都是唯一的。
2. 根节点左子树的关键字(如果有的话)小于根节点的关键字。
3. 根节点右子树的关键字(如果有的话)大于根节点的关键字。
4. 根节点的左右子树也都是二叉搜索树。

[2,3,4可以定义为:任何节点左子树的关键字小于该节点的关键字；任何节点右子树的关键字大于该节点的关键字。]

二叉搜索树：二叉排序树

例：二叉树



- 哪一个是 二叉搜索树?

  → (b)和 (c)

有重复值的二叉搜索树

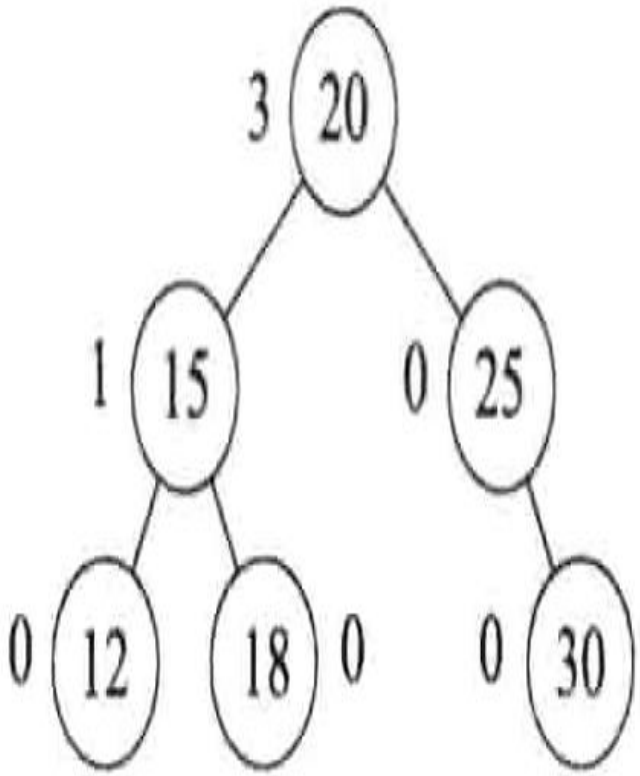
- 放弃二叉搜索树中所有元素拥有不同关键字的要求
  - 用“小于等于”代替特征2 中的“小于”
  - 用“大于等于”代替特征3中的“大于”
  - 这样的二叉搜索树，称为有重复值的二叉搜索树 (binary search tree with duplicates).

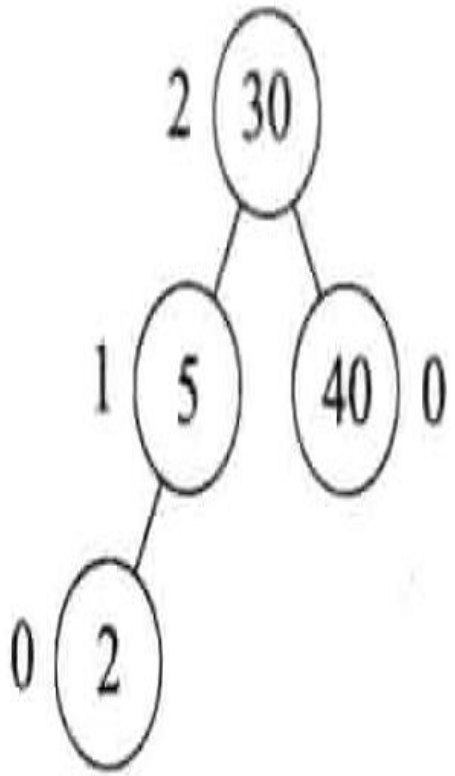
14.1.2 索引二叉搜索树定义

索引二叉搜索树定义：

- 二叉搜索树.
- 在每个节点中添加一个 ‘LeftSize’—该节点左子树的元素个数.
- LeftSize（x)
  - 给出了一个元素在x为根的子树中排名(0起始)。

例：带索引的二叉搜索树





LeftSize和名次

- 一个元素的名次(Rank)是它在排序后的队列中的位置(在中序序列中的位置)。

  [2,6,7,8,10,15,18,20,25,30,35,40]

- rank(2)=0
- rank(15)=5
- rank(20)=7
- LeftSize(x) = rank(x)(在以x为根的子树中的名次)

14.2 抽象数据类型

抽象数据类型bsTree
{
实例

二叉树，每一个节点中有一个数对，数对的一个成员是关键字，另一个成员是数值；所有元素的关键字各不相同；任何节点左子树的关键字小于该节点的关键字；任何节点右子树的关键字大于该节点的关键字。

操作:

find(k)：返回关键字为k的数对

insert(p)：将数对p插入到搜索树中

erase(k)：删除关键字为k的数对

ascend()：按照关键字的升序排列输出所有数对

}

索引二叉搜索树的抽象数据类型

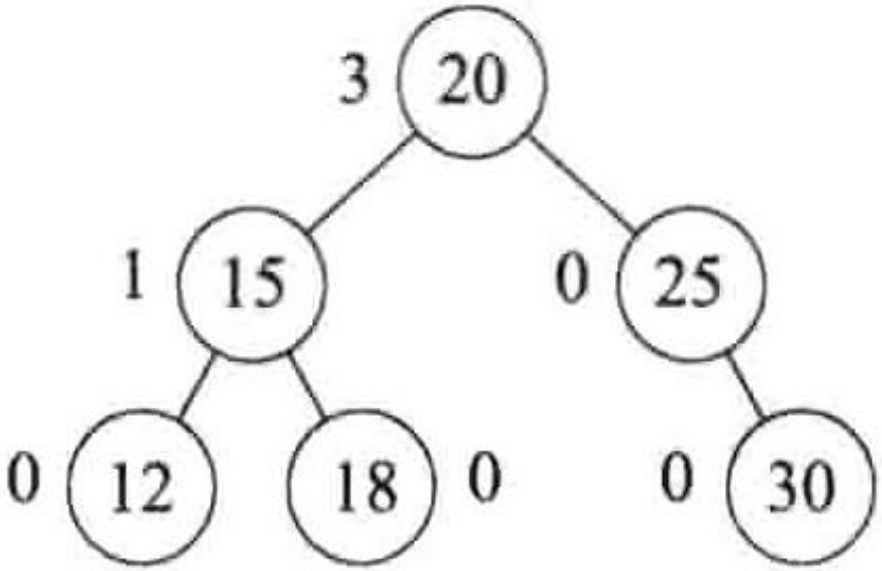
抽象数据类型IndexedBSTree
{
实例
　　与bsTree 的实例相同,只是每一个节点有一个LeftSize 域
操作:
　　find(k)：返回关键字为k的数对
　　get(index)：返回第index个数对
　　insert(p)：将数对p插入到搜索树中
　　erase(k)：删除关键字为k的数对
　　delete(index)：删除第index个数对
　　ascend( )：按照关键字的升序排列输出所有数对
}

14.3二叉搜索树的操作实现

类binarySearchTree 是类linkedBinaryTree的派生类

```
template  <class  K,class  E>
class  binarySearchTree: public linkedBinaryTree <K,E>
{
 public:
   pair<const K,E> * find(const K& theKey) const;
   //返回关键字theKey匹配的数对的指针，若不存在匹配的数
    对，则返回NULL
   void insert(const pair<const K,E>& thePair);
    //插入一个数对thePair,如果存在关键字相同的数对,则覆盖
   void erase(const K& theKey);
   //删除关键字theKey匹配的数对
   void  ascend( )
   {//按照关键字的升序排列输出所有数对
     inOrderOutput( );}
};
```

操作ascend

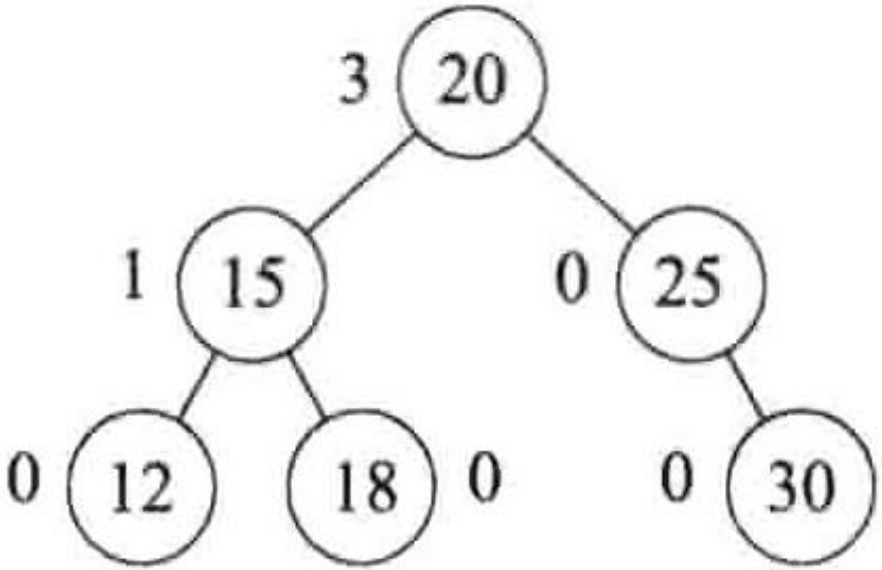


- 我们如何以升序输出所有的元素?
  - 中序遍历.

操作find

- 搜索先从根开始。
- 如果根为空，那么搜索树不包含任何元素，查找失败.
- 如果k 小于根节点的关键值，在左子树中搜索k 。
- 如果k 大于根节点的关键值，在右子树中搜索k 。
- 如果k 等于根节点的关键值，则查找成功。

搜索find示例



- 搜索关键字　18

方法find实现

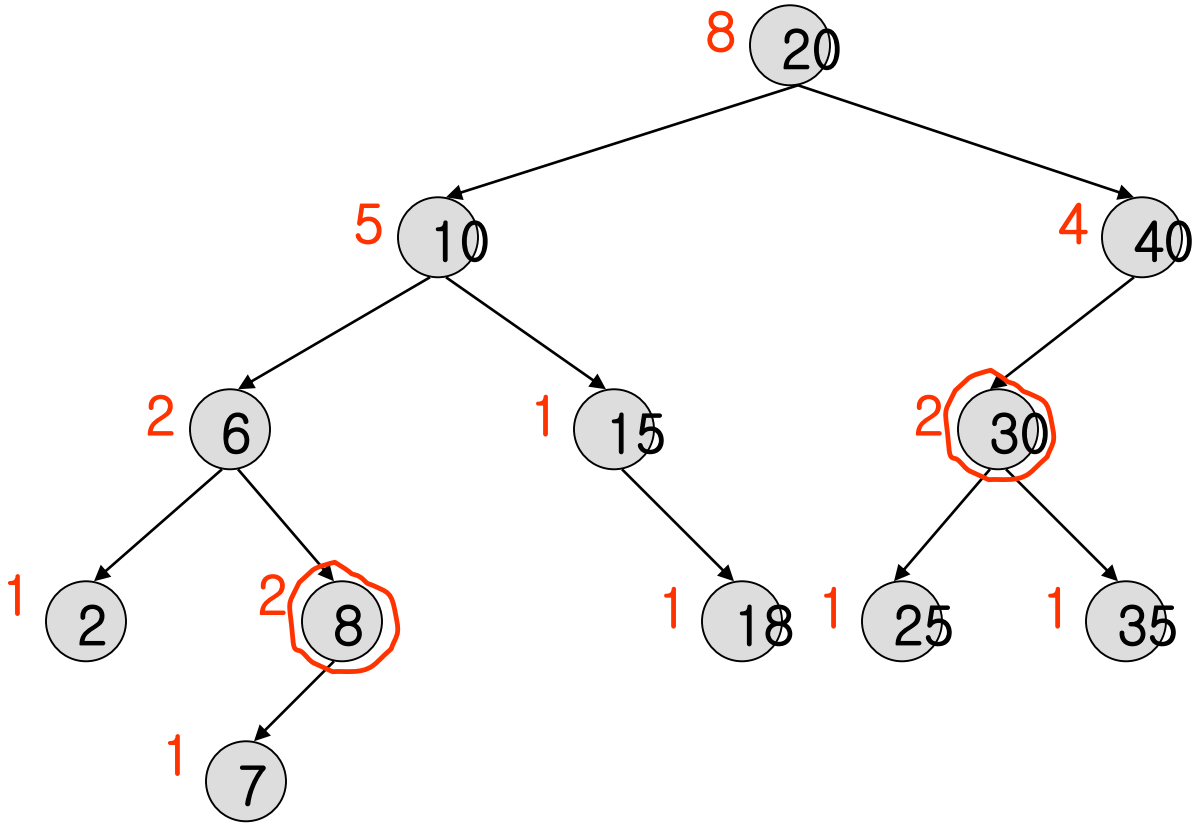
```
template<class K, class E>
pair<const K,E> * find(const K& theKey) const;
{//返回关键字theKey匹配的数对的指针，若不存在匹配的数
   对，则返回NULL
 // 指针p 从根开始搜索，寻找关键字等于theKey的元素(数对)
    BinaryTreeNode<pair<const K,E>> *p = root;
    while (p!=NULL)     // 检查p->element
        If  (theKey < p->element.first)
                p = p->leftChild;
        else  if  (theKey > p->element.first)
                        p = p->rightChild;
                else {// 找到匹配的元素
                        return  &p->element;
    //无匹配的元素
    return NULL;
}
```

•搜索操作的时间复杂性：O(height)

操作 IndexSearch(k)

- IndexSearch(k,e)返回第k个元素。
- 如果k=x.LeftSize,第k个元素是x.element.
- 如果k<x.LeftSize,第k个元素是x的左子树的第k个元素.
- 如果k>x.LeftSize,第k个元素是x的右子树的第(k-x.LeftSize)个元素.

IndexSearch 示例

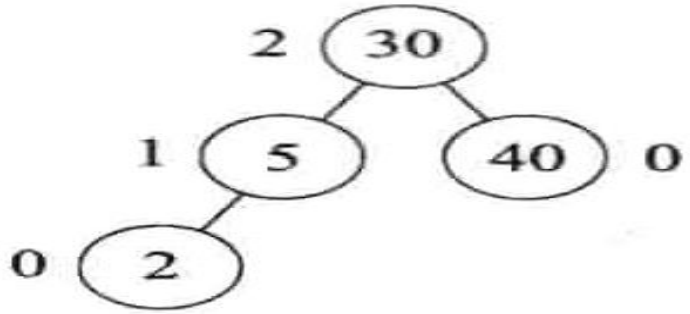


- 哪个是第4个元素？
- 哪个是第10个元素?

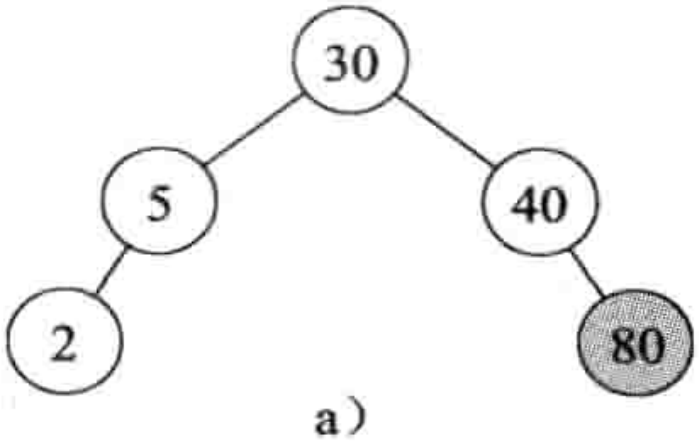
操作Insert

- 若在二叉搜索树中插入一个新元素e，
- 首先搜索，验证e 的关键值是否存在
- 如果搜索成功，那么新元素将不被插入
- 如果搜索不成功，那么新元素将被插入到搜索的中断点。

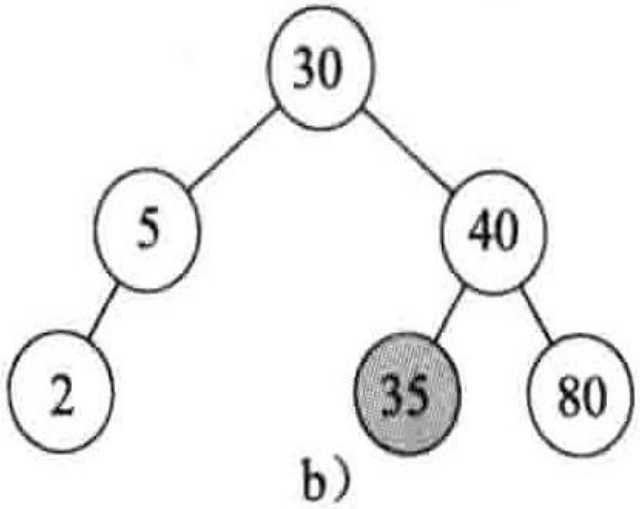
操作insert 示例



插入80



插入35



方法insert实现—1/2

```
template<class K, class E>
void binarySearchTree<E,K>::
                    insert(const pair<const K,E>& thePair);
{//插入一个数对thePair,如果存在关键字相同的数对,则覆盖
 BinaryTreeNode<pair<const K,E>>*p = root, // 搜索指针
                                   *pp = NULL; // p的父节点指针
 // 寻找插入点
 while (p!=NULL) {// 检查p->element
   pp = p;
   // 将p移向孩子节点
   if  (thePair.first < p->element.first)  p = p->leftChild;
   else  if  (thePair.first > p->element.first)  p = p->rightChild;
         else p->element.second =thePair.second; // 覆盖旧值
}
```

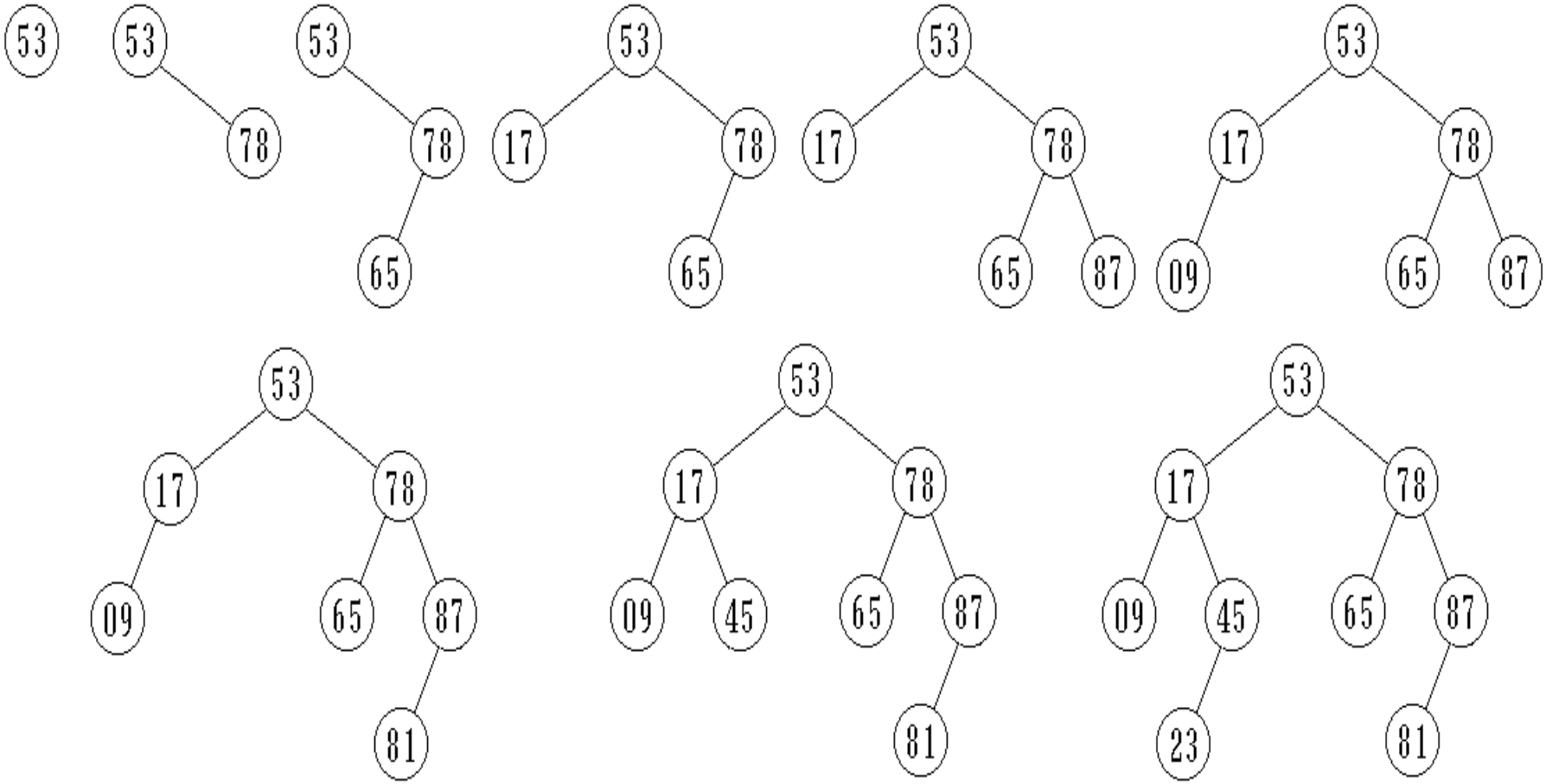
方法insert实现—1/2

```
 // 为thePair 建立一个新节点，并将该节点连接至 pp
BinaryTreeNode<pair<const K,E>>*newNode =
      new BinaryTreeNode<pair<const K,E>> (thePair);
if (root!=NULL) {// 树非空
     if (thePair.first < pp->element.first)
              pp->leftChild = newNode;
     else   pp->rightChild = newNode;
     }
else // 插入到空树中
       root = newNode;
}
```

- 插入操作的时间复杂性： O(height)

输入数据，建立二叉搜索树的过程

输入数据序列 { 53, 78, 65, 17, 87, 09, 81, 45, 23 }

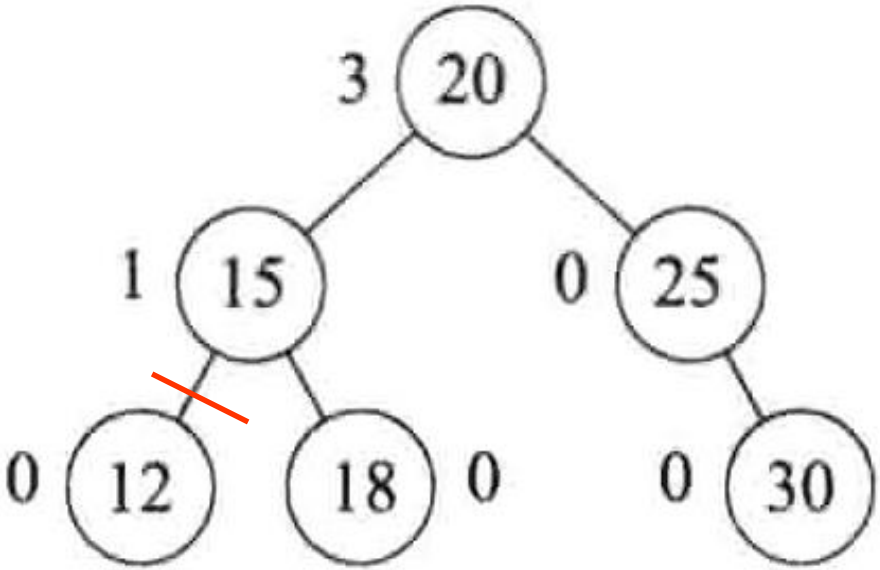


方法erase(theKey)

- 对删除来说，我们考虑包含被删除元素的节点p 的三种情况：
  1. p 是树叶；
  2. p 只有一个非空子树；
  3. p 有两个非空子树。

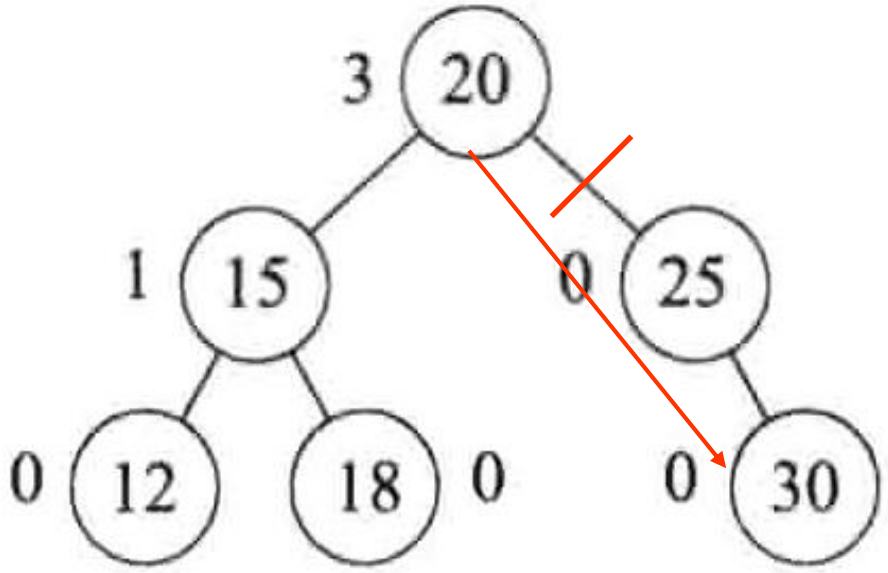
情况1:删除叶子节点

- 对于情况1, 丢弃树叶节点 。
- 例, 删除元素的关键字为12



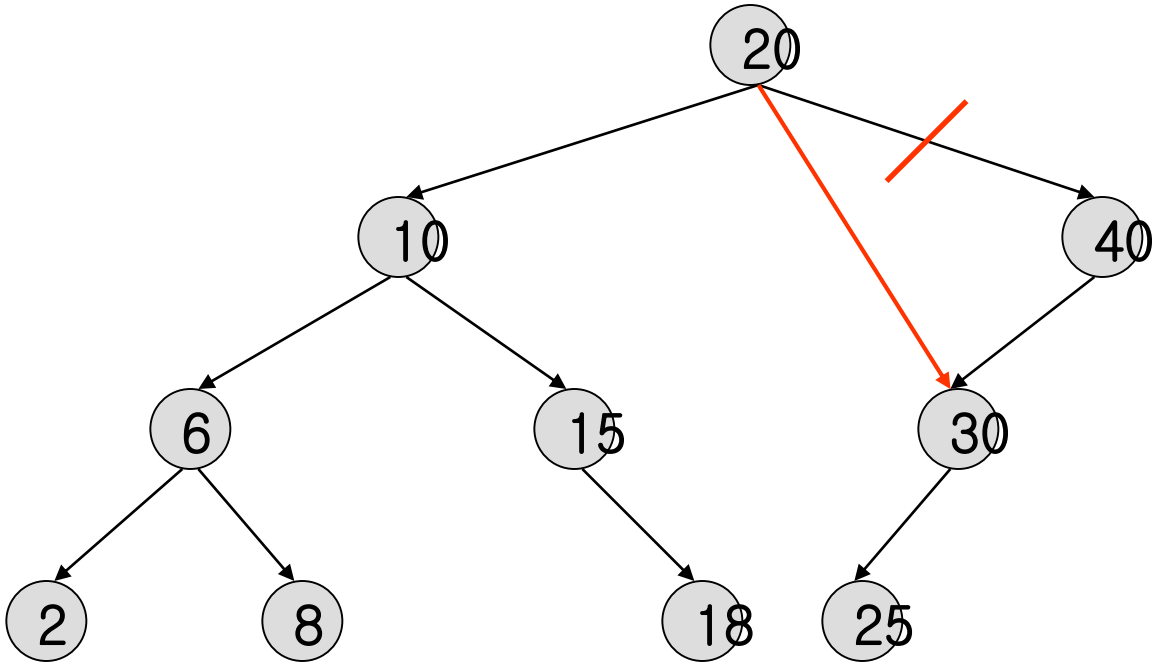
情况2：删除一个度为1的节点

•例: 删除元素的关键字为25



1. 如果p没有父节点(即p是根节点)，则将p丢弃，p的唯一子树的根节点成为新的搜索树的根节点。
2. 如果p有父节点pp，则修改pp的指针，使得pp指向p的唯一孩子，然后删除节点p。

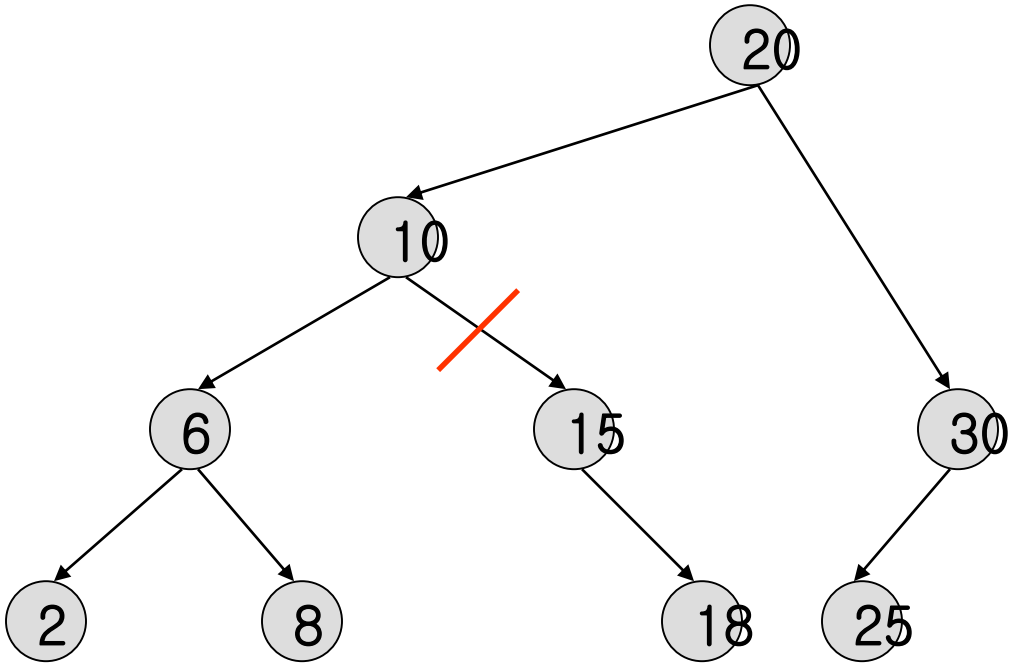
情况2：删除一个度为1的节点



•例: 删除元素的key=40

情况2：删除一个度为1的节点

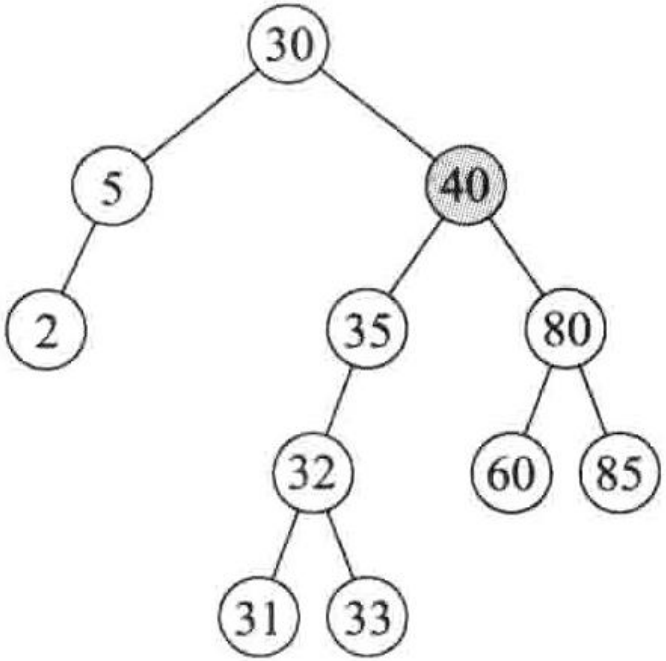
例：删除元素的key=15

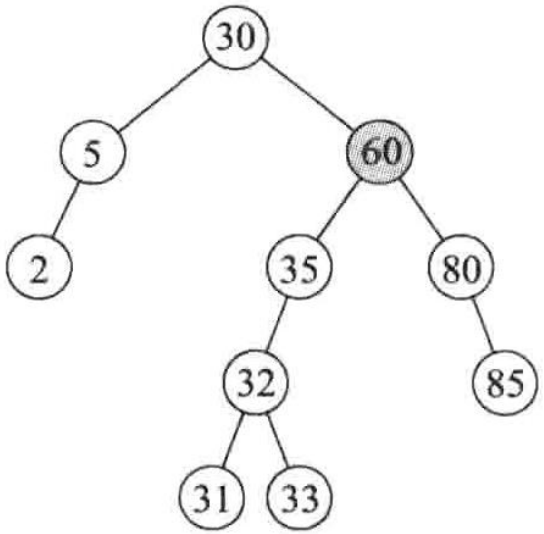


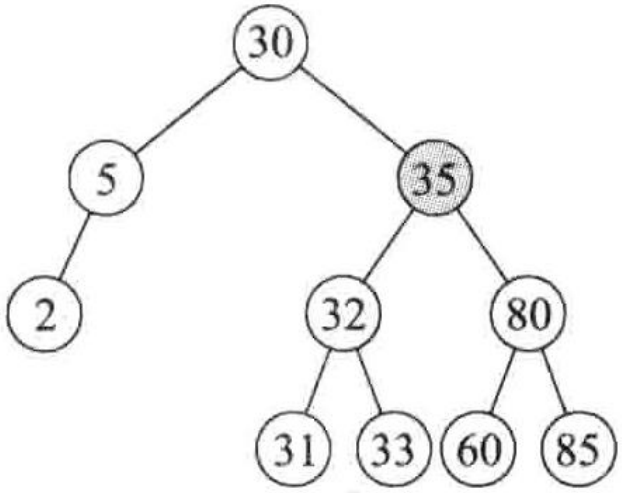
二叉搜索树的删除

- 如果p有两个非空子树。
- 只需将该元素替换为它的左子树中的最大元素或右子树中的最小元素。
  - 可以按下述方法来查找到左子树中的最大元素：首先移动到子树的根，然后沿着各节点的右孩子指针移动，直到右孩子指针为0为止。
  - 类似地，也可以找到右子树中的最小元素：首先移动到子树的根，然后沿着各节点的左孩子指针移动，直到左孩子指针为0为止。
- 问题转化为情况1或2。

情况3：删除一个度为2的节点

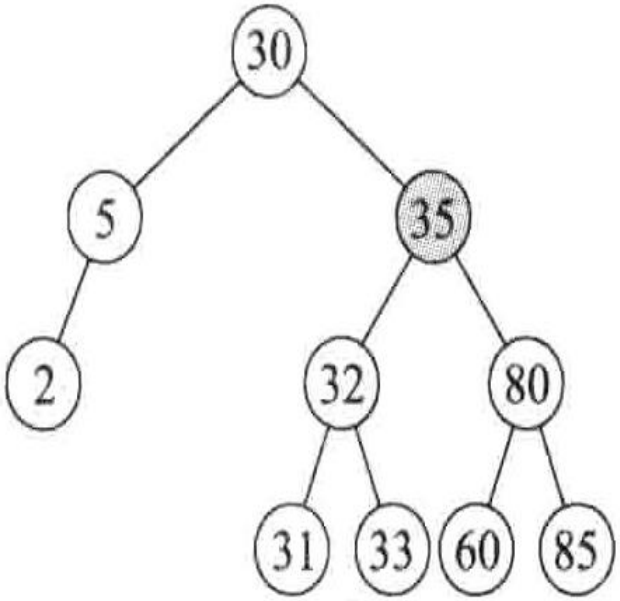


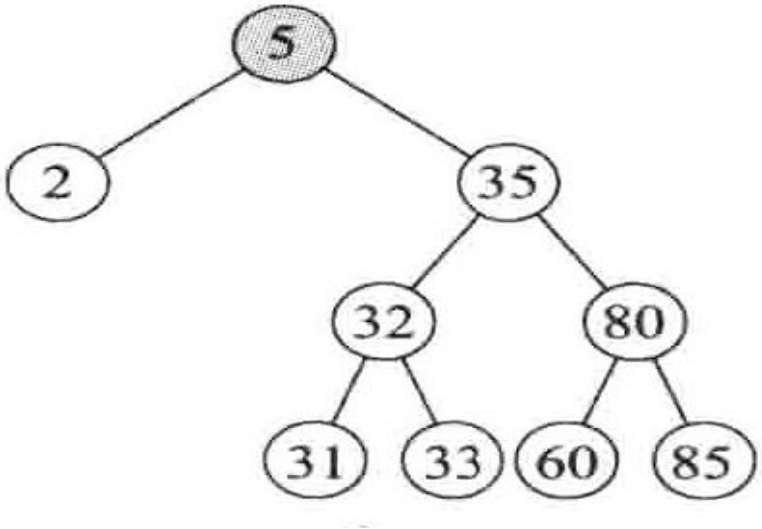




- 例：删除40

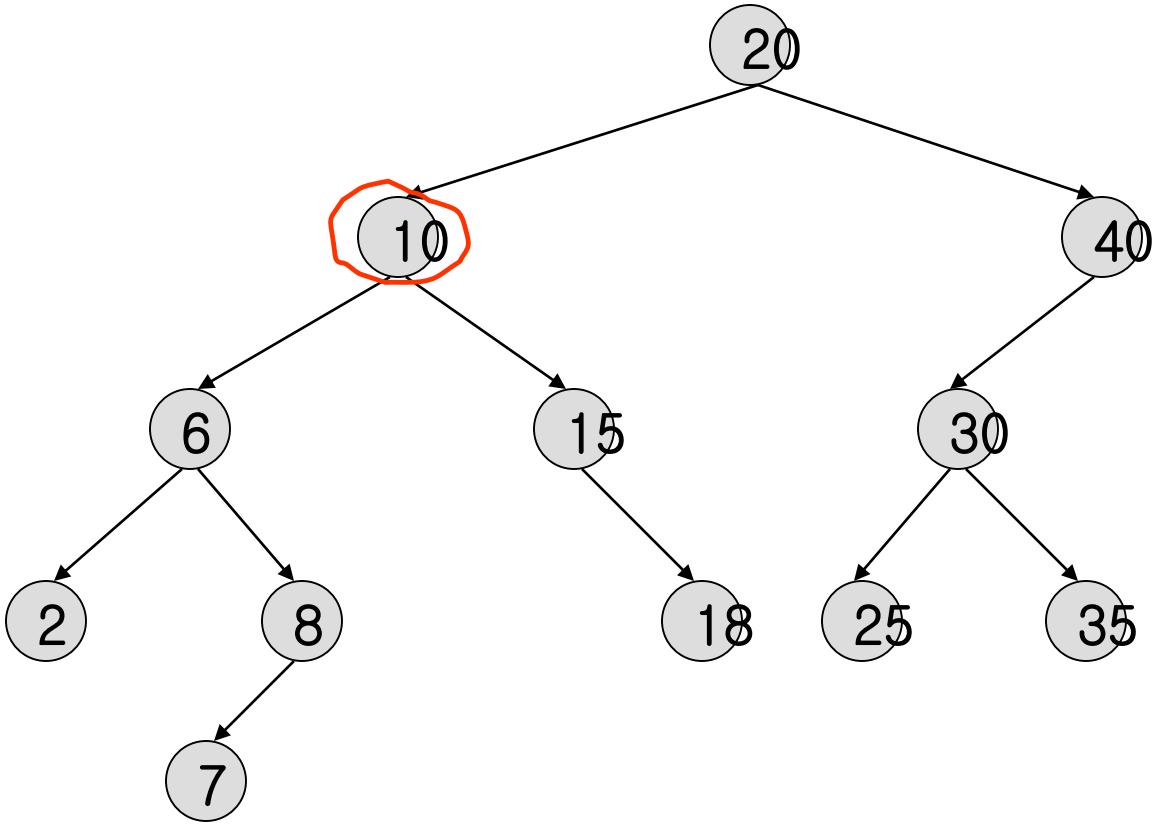
情况3：删除一个度为2的节点





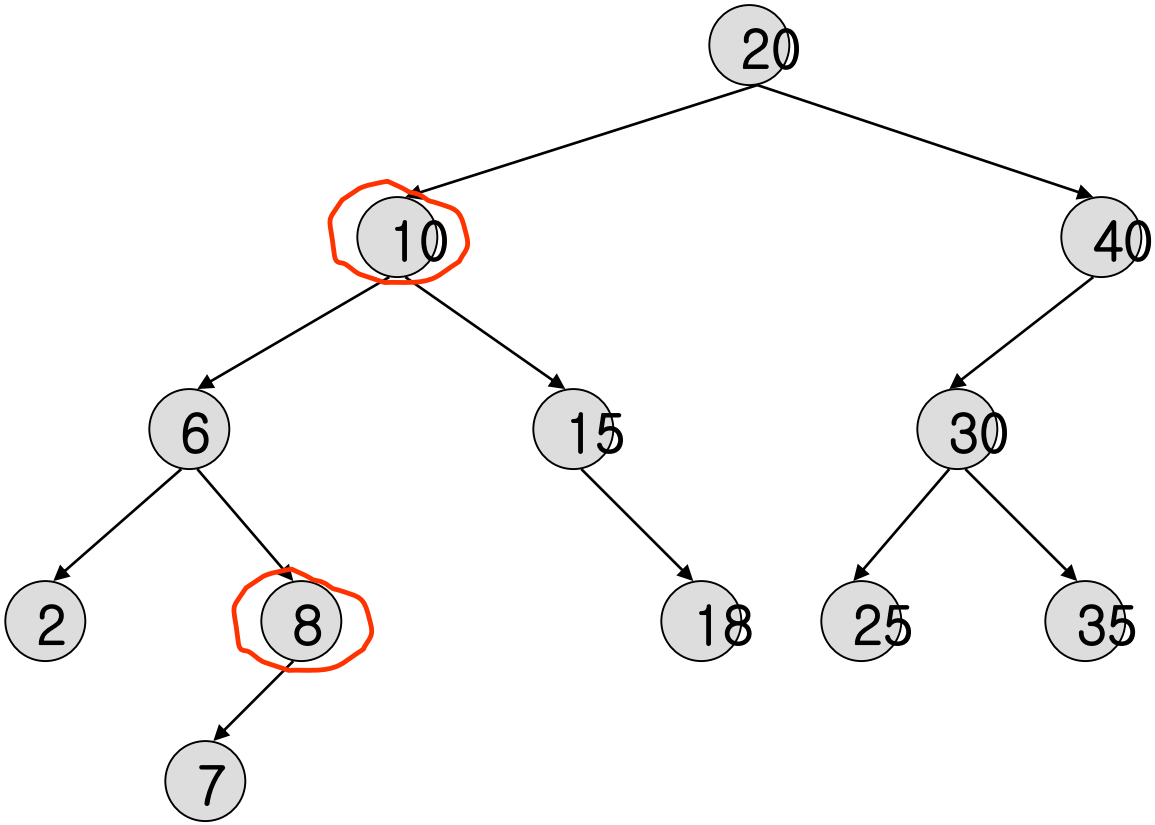
- 例：删除30

情况3：删除一个度为2的节点



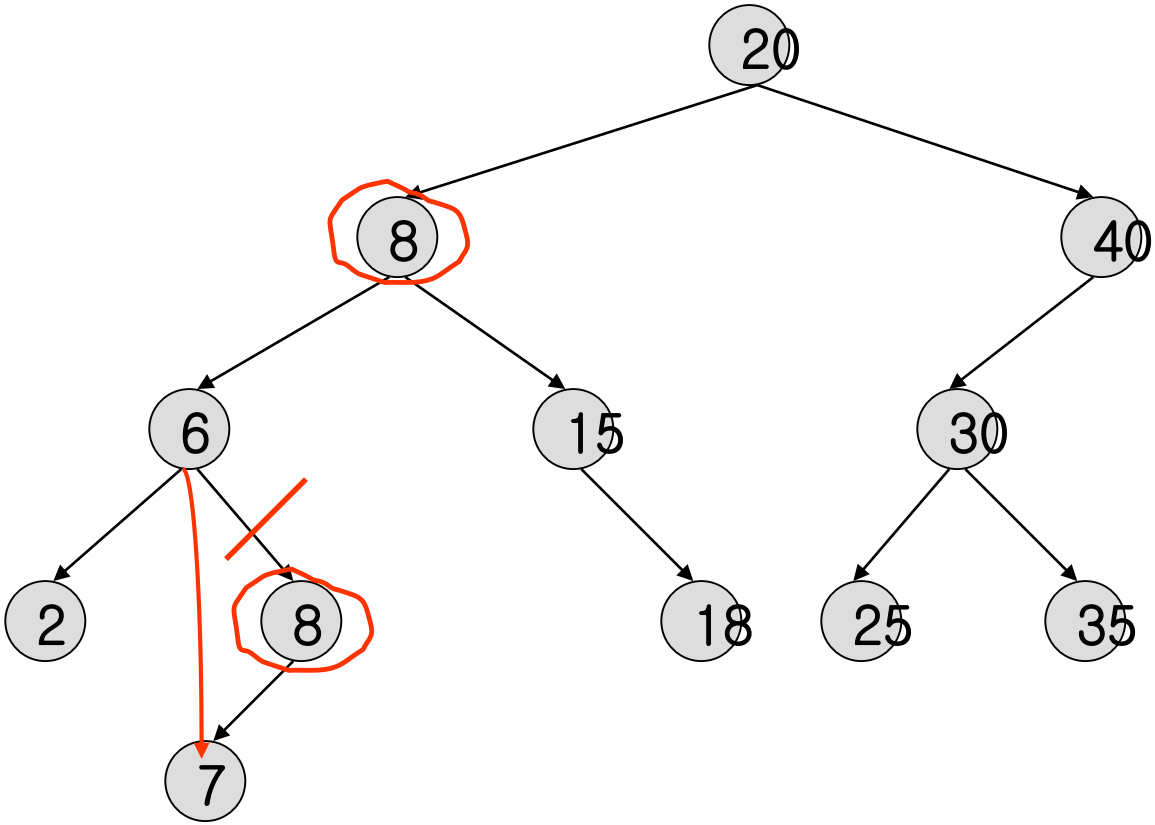
- 例：删除元素的key=10

情况3：删除一个度为2的节点



- 例：删除元素的key=10
- 可以用它左子树中的最大元素来替换它
- 也可以用它右子树中的最小元素来替换它
- 哪一个是左子树中的最大元素?

情况3：删除一个度为2的节点



- 例： 删除元素的key=10
- 可以用它左子树中的最大元素来替换它
- 也可以用它右子树中的最小元素来替换它
- 哪一个是左子树中的最大元素?

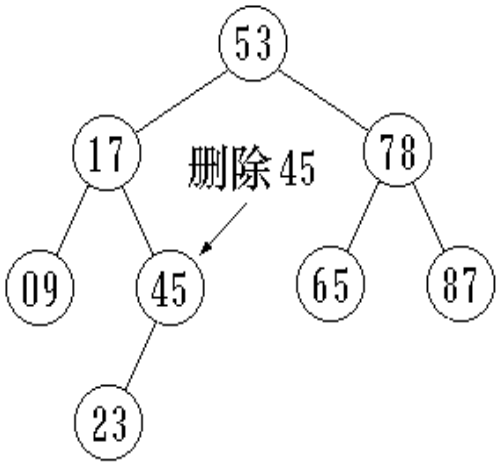
```
template<class K, class E>
void binarySearchTree<E,K>::erase(const K& theKey)
{// 删除关键字为theKey 的元素(数对)
// 将p 指向关键字为theKey的节点
BinaryTreeNode<pair<const K,E>>*p = root, // 搜索指针
                                *pp = NULL; // p的父节点指针
while (p !=NULL && p->element.first != theKey)
  {// 移动到p的孩子
    pp = p;
    if  (theKey < p->element.first)   p = p->leftChild;
    else   p = p->rightChild;
    }
If   (p==NULL)   return; // 没有关键字为theKey的元素
```

// 对树进行重构

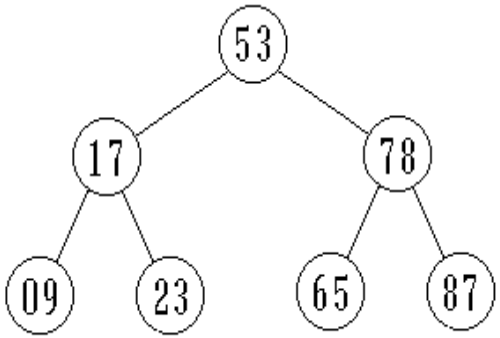
```
// 处理p有两个孩子的情形
if (p->leftChild && p->rightChild)
{//两个孩子，转换成有0或1个孩子的情形

  // 在p 的左子树中寻找最大元素
  BinaryTreeNode<pair<const K,E>>*s = p->leftChild,
                                      *ps = p; // s的父节点
  while (s->rightChild!=NULL) {// 移动到最大的元素
        ps = s;
        s = s->rightChild;}
  p->element = s->element; // 将最大元素从s移动到p
  BinaryTreeNode<pair<const K,E>>*q =
       new BinaryTreeNode<pair<const K,E>>
             (s->element, p->leftChild, p->rightChild);
  ......//与P关联的指针修改为与q关联
          //p指向新的删除节点s，pp为p的父节点
}
```

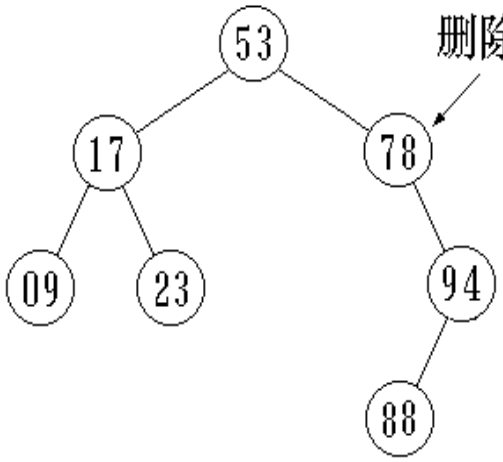
```
// p 最多有一个孩子
// 在c 中保存孩子指针
BinaryTreeNode<pair<const K,E>>*c;
if  (p->leftChild != NULL)  c = p->leftChild;
                            else   c = p->rightChild;

// 删除p
if (p = = root)   root = c;
else
   {// p 是pp的左孩子还是pp的右孩子?
     if (p = = pp->leftChild)
              pp->leftChild = c;
     else   pp->rightChild = c;
    }
treesize--;
delete   p;
}
```

删除情况汇总

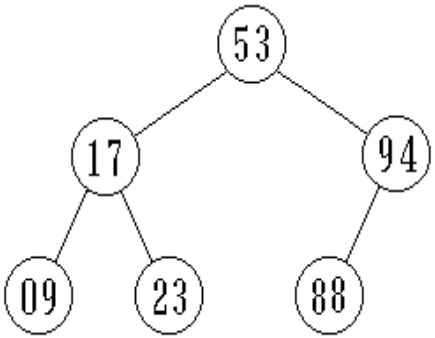


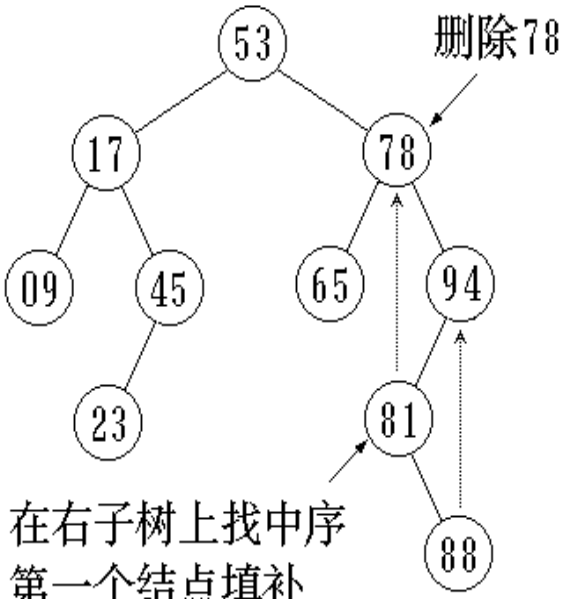
缺右子树用左子女填补



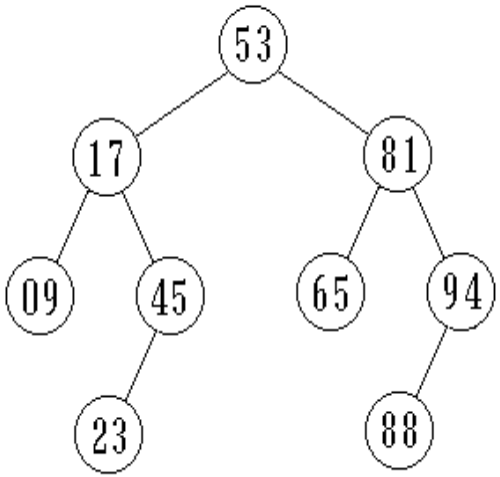


缺左子树用右子女填补





在右子树上找中序第一个结点填补



类dBinarySearchTree

- 有重复值的二叉搜索树(The binary search tree with duplicates —dBinarySearchTree)
- 在实现dBinarySearchTree类时，只需把binarySearchTree::insert的while循环(见程序14-5)改为：

```
while (p) {
    pp = p;
    if (thePair.first <= p->element. first)
        p = p->leftChild;
    else  p = p->rightChild;
}   //程序14-7
```

二叉搜索树的高度

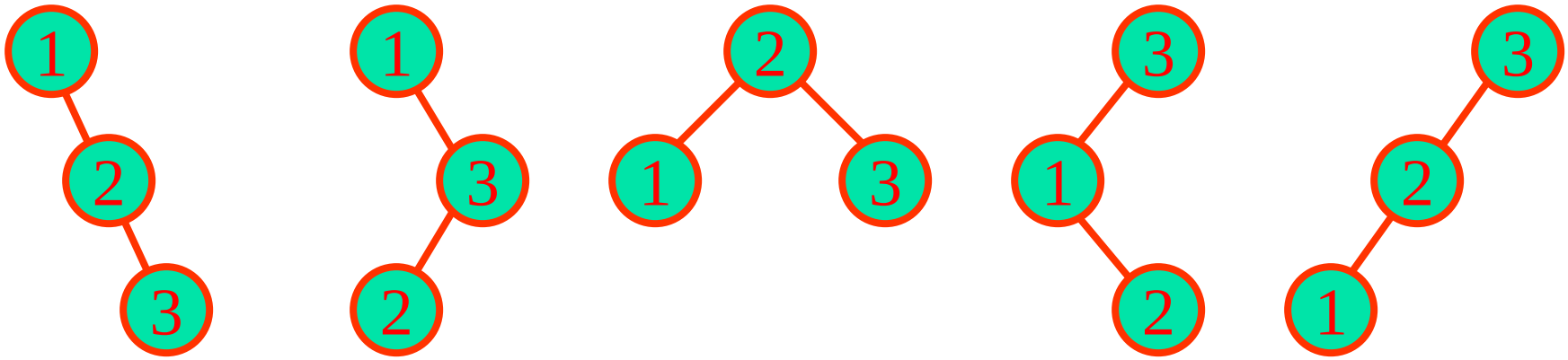
- 最大:
  - 关键字为[1,2,3,…….n]的元素按顺序插入到一棵空的二叉搜索树时.
  - n
  - 搜索、插入和删除操作所需要的时间: O(n)
- 平均:
  - O(logn)
  - 搜索、插入和删除操作所需要的时间:O(logn)

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。这直接影响到二叉搜索树的搜索性能。

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，这样必然会降低搜索性能。

{2, 1, 3}

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



14.6 应用

- 14.6.1 直方图
- 14.6.2 箱子装载

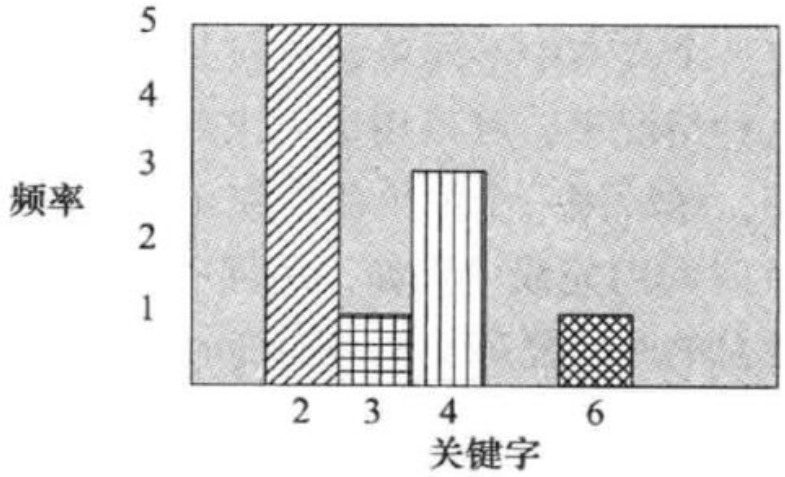
14.6.1 直方图

n=10; 关键字=[2, 4, 2, 2, 3, 4, 2, 6, 4, 2]

a）输入

关键字	频率
2	5
3	1
4	3
6	1

b）输出直方图表格



c）直方图条形图

图 14-5　直方图举例

14.6.1 直方图

程序 14-8　简单的直方图程序

```
void main(void)
{// 非负整型值的直方图
   int n,                                          // 元素个数
       r;                                          // 0 至 r 之间的值
   cout << "Enter number of elements and range"
        << endl;
   cin >> n >> r;

   // 生成直方图数组 h
   int *h = new int[r+1];

   // 将数组 h 初始化为 0
   for (int i = 0; i <= r; i++)
      h[i] = 0;

   // 输入数据，然后计算直方图
   for (i = 1; i <= n; i++)
   {// 假设输入的值在 0 至 r 之间
      int key;                                     // 输入值
      cout << "Enter element " << i << endl;
      cin >> key;
      h[key]++;
   }

   // 输出直方图
   cout << "Distinct elements and frequencies are"
        << endl;
   for (i = 0; i <= r; i++)
      if (h[i] != 0)
         cout << i << "    " << h[i] << endl;
}
```

14.6.1 直方图

程序 14-9　使用搜索树的直方图

```
int main(void)
{//使用搜索树的直方图
   int n;                                                    //元素个数
   cout << "Enter number of elements" << endl;
   cin >> n;

   //输入元素，然后插入树
   binarySearchTreeWithVisit<int, int> theTree;
   for (int i = 1; i <= n; i++)
   {
      pair<int, int> thePair;                                //输入元素
      cout << "Enter element " << i << endl;
      cin >> thePair.first;                                  //关键字
      thePair.second = 1;                                    //频率
      //将 thePair 插入树，除非存在与之匹配的元素
      //在后一种情况下，count 值增 1
      theTree.insert(thePair, add1);
   }

   //输出不同的关键字和它们的频率
   cout << "Distinct elements and frequencies are"
        << endl;
   theTree.ascend();
}
```

14.6.2 箱子装载

在箱子装载问题中，箱子的数量不限，每个箱子的容量为 binCapacity，待装箱的物品有 n 个。物品 i 需要占用的箱子容量为 objectSize[i]，0 ≤ objectSize[i] ≤ binCapacity。所谓**可行装载**（feasible packing），是指所有物品都装入箱子而不溢出。所谓**最优装载**（optimal packing）是指使用箱子最少的可行装载。

假设要装载物品 *i*，已使用的箱子有 9 个（*a*~*i*），它们都有剩余容量。这些剩余容量分别是 1,3,12,6,8,1,20,6 和 5。注意，箱子不同，但剩余容量可能相同。可以用一棵带有重复关键字的二叉搜索树（即 dBinarySearchTree 的实例）来描述这 9 个箱子，每个箱子的剩余容量作为节点的关键字。

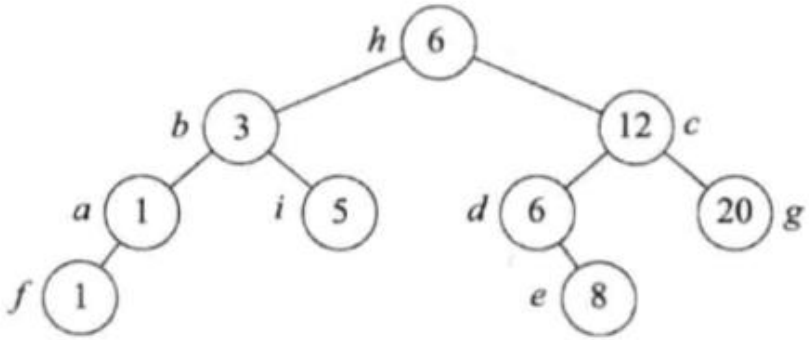


图 14-6　带有重复关键字的二叉搜索树

14.6.2 箱子装载

如果要装载的物品 i 需要 objectSize[i]=4 个单位的空间，再看另一个例子，假设 objectSize[i]=7。

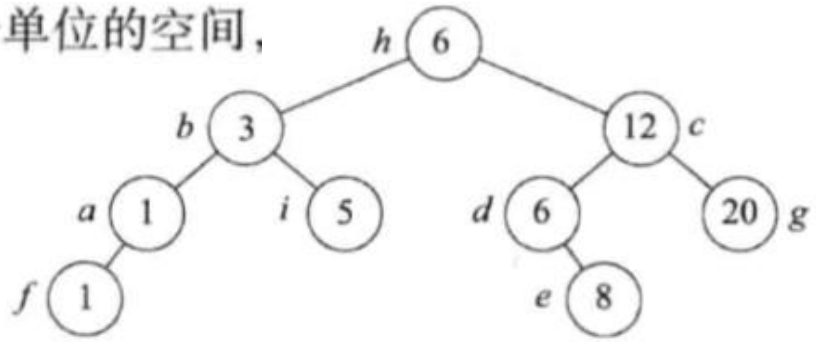


图 14-6　带有重复关键字的二叉搜索树

当我们为物品 i 找到最匹配的箱子后，可以将它从搜索树中删除，将其剩余容量减去 objectSize[i]，再将它重新插入树中（除非它的剩余容量为零）。若没有找到最匹配的箱子，则启用一个新箱子。

作业：

- 10，15（只写算法）
- 输入一个正整数序列{100，50，280，450，66，200，30，260}，建立一棵二叉搜索树，要求：
  - (1) 画出该二叉搜索树；
  - (2) 画出删除结点280后的二叉搜索树。

若T为BSTree，添加函数计算FindMax,Findmin。

若T为BSTree，添加函数计算Split（K & k, t), 该函数寻找关键字为k的节点，并把以该节点为根的子树去掉，该子树指针赋值给t。

若T1，T2为两棵不同的BSTree，如何合并成一棵所有key不相同的BSTree?