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

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，比如，按关键字的升序输出字典元素、按升序找到第k个元素、删除第k个元素，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

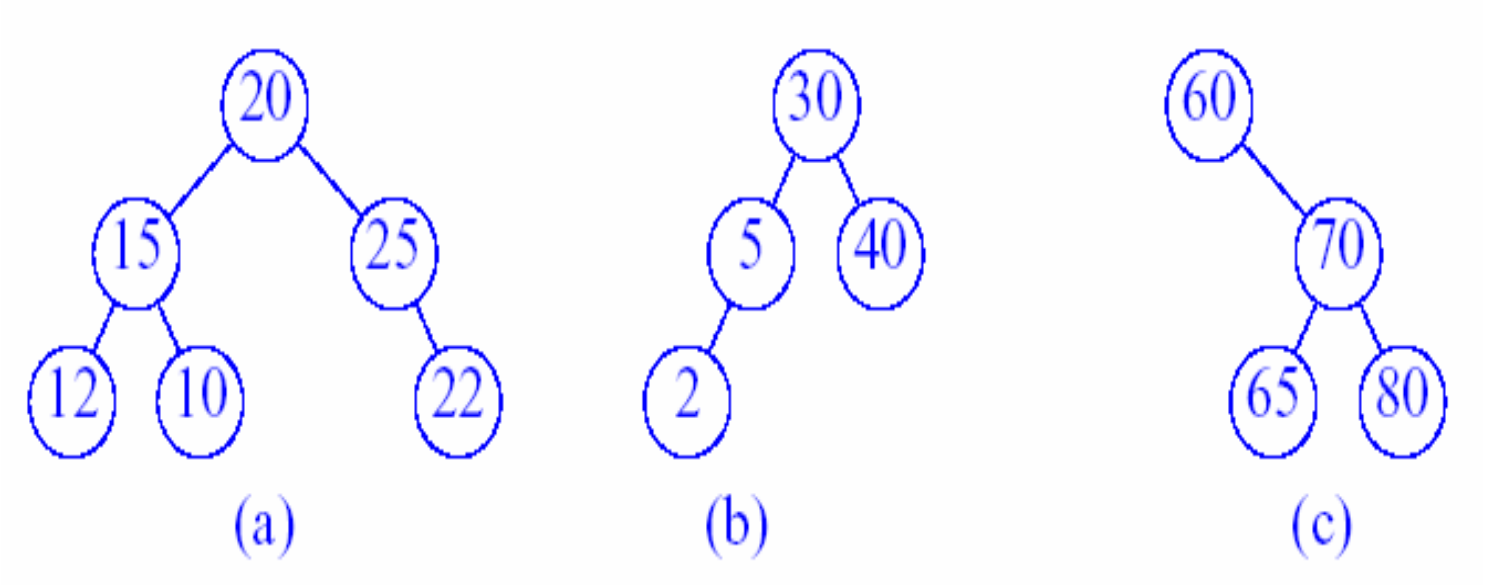
- 定义14-1[二叉搜索树(binary search tree)]：
- 二叉搜索树是一棵可能为空的二叉树，一棵非空的二叉搜索树满足以下特征：

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。]

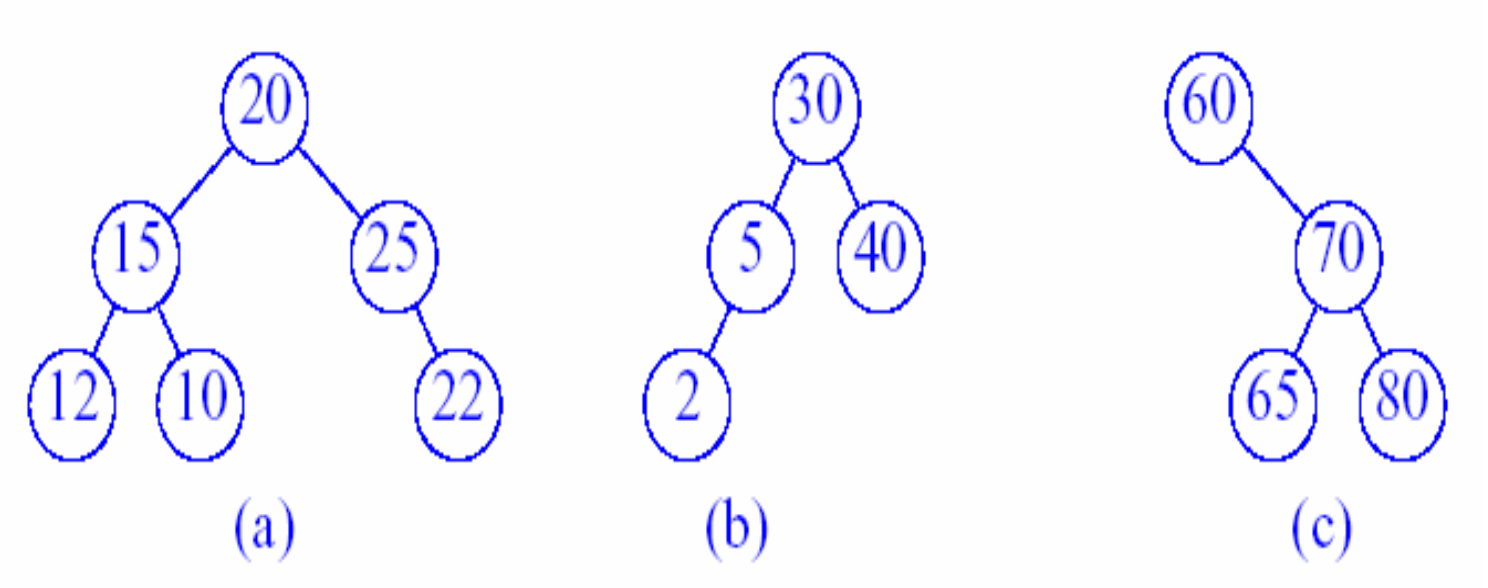
二叉搜索树：二叉排序树

例：二叉树



- 哪一个是 二叉搜索树?

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

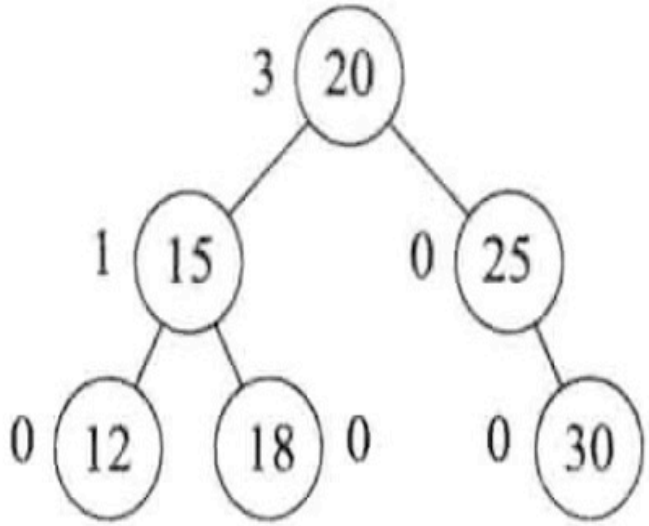
- 放弃二叉搜索树中所有元素拥有不同关键字的要求
  - 用“小于等于”代替特征2 中的“小于”
  - 用“大于等于”代替特征3中的“大于”
  - 这样的二叉搜索树，称为**有重复值的二叉搜索树**(binary search tree with duplicates).

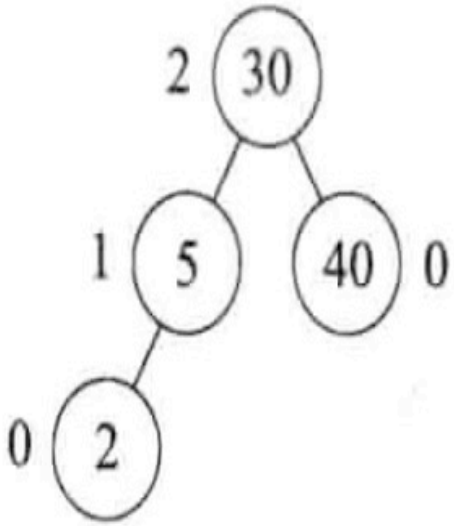
14.1.2 索引二叉搜索树定义

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

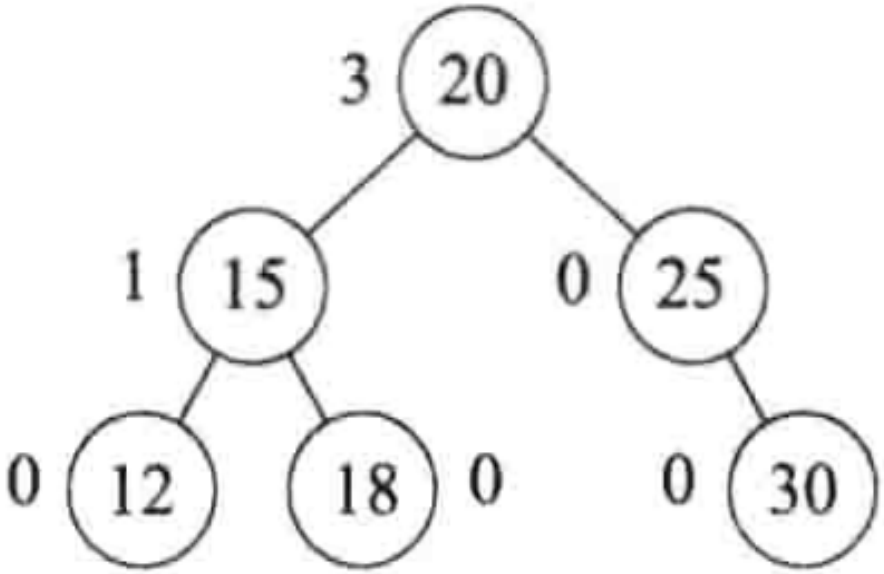
二叉搜索树数量和形状随着操作而改变，所以二叉搜索树要用链表来描述

14.3二叉搜索树的操作实现

类binarySearchTree 是类linkedBinaryTree的派生类

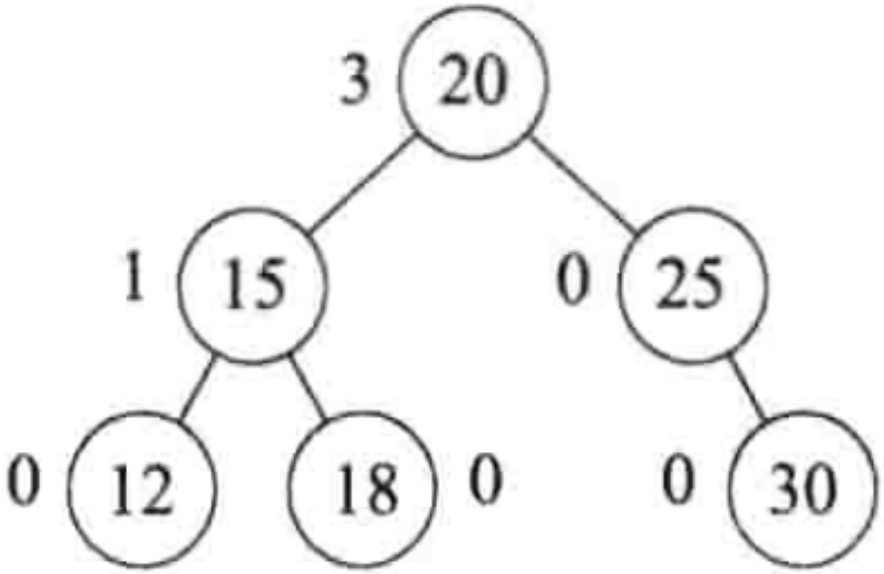
```
template  <class  K,class  E>
class  binarySearchTree: public linkedBinaryTree <K,E>
{
 public:
   pair<const K,E> * find(const K& theKey) const;
   //返回关键字theKey匹配的数对的指针，若不存在匹配的数对，则返回NULL
   void insert(const pair<const K,E>& thePair);
    //插入一个数对thePair,如果存在关键字相同的数对,则覆盖
   void erase(const K& 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



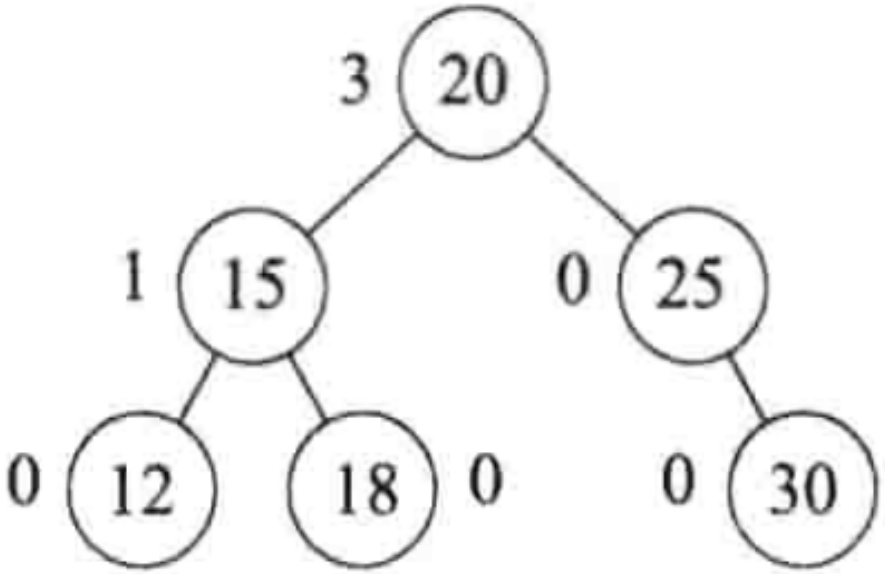
- 我们如何以升序输出所有的元素?

操作ascend



- 我们如何以升序输出所有的元素?
  - 中序遍历.

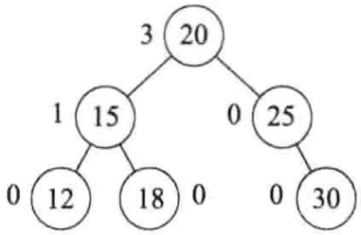
搜索find示例



- 搜索关键字　18

方法find实现

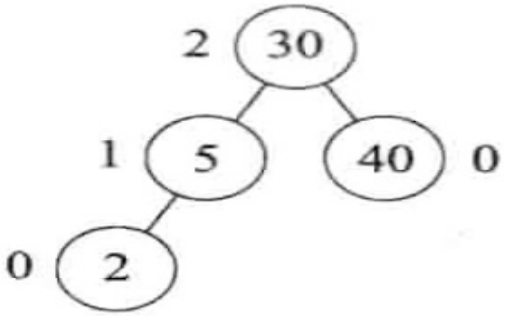
```
template<class K, class E>
pair<const K,E> * find(const K& theKey) const;
{//返回关键字theKey匹配的数对的指针，若不存在匹配的数
   对，则返回NULL
 // 指针p 从根开始搜索，寻找关键字等于theKey的元素(数对)
    BinaryTreeNode<pair<const K,E>> *p = root;
    while (p!=NULL)     // 检查p->element
        If  (theKey < p->element.first)
              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



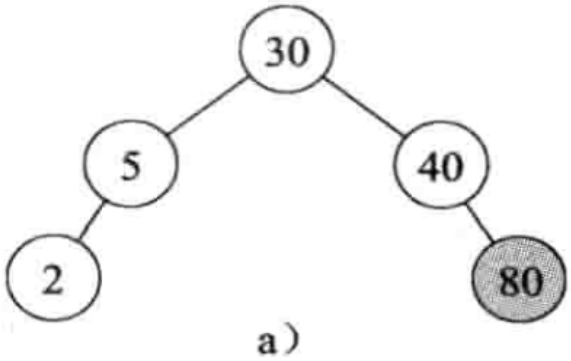
- 搜索关键字　18

•搜索操作的时间复杂性：O(height)

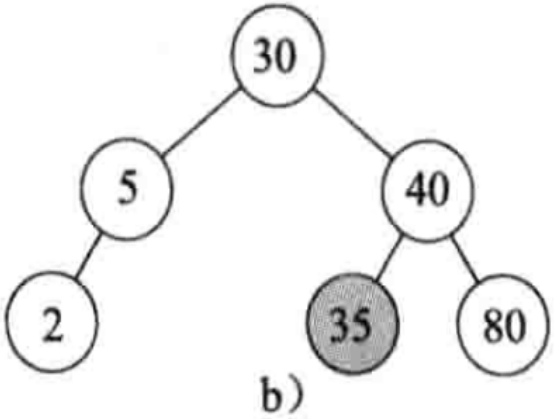
操作insert 示例



插入80



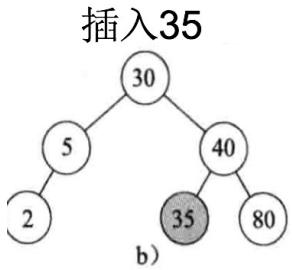
插入35



- 首先通过查找来确定是否存在某个元素，如果不存在，那就将新元素作为搜索中断节点的孩子插入二叉搜索树

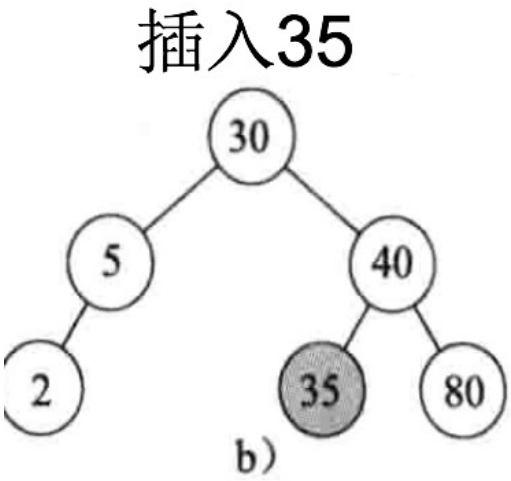
方法insert实现—1/2

```
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;
}
```



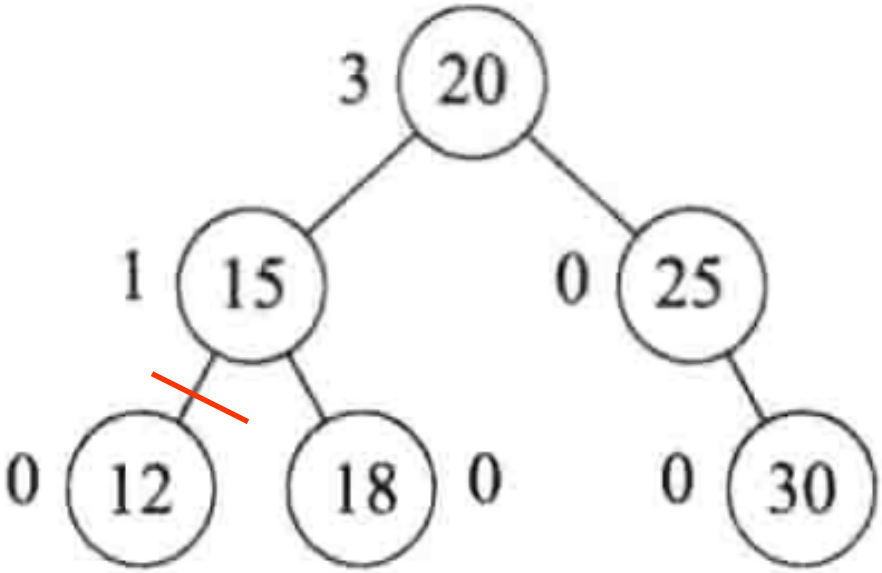
- 插入操作的时间复杂性： O(height)

方法erase(theKey)

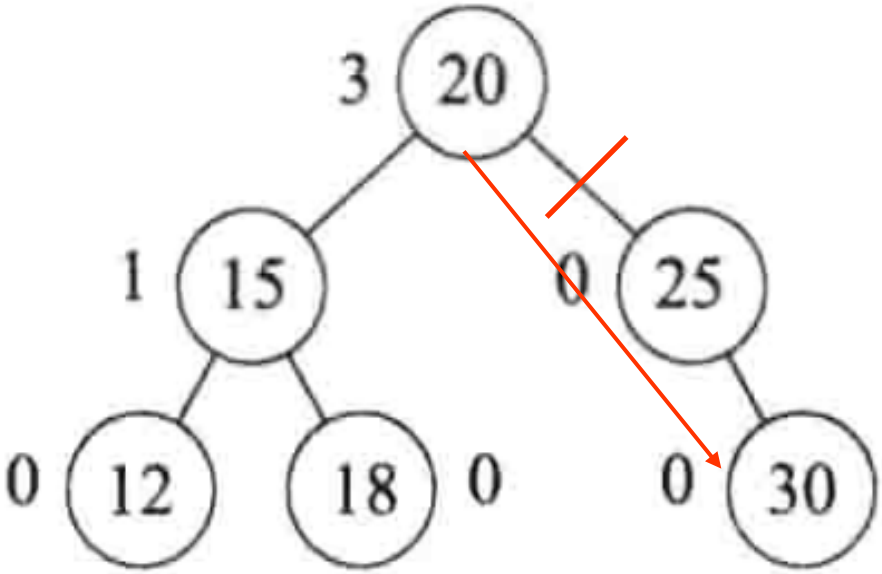
- 对删除来说，我们考虑包含被删除元素的节点p 的三种情况：
  1. p 是树叶；
  2. p 只有一个非空子树；
  3. p 有两个非空子树。

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

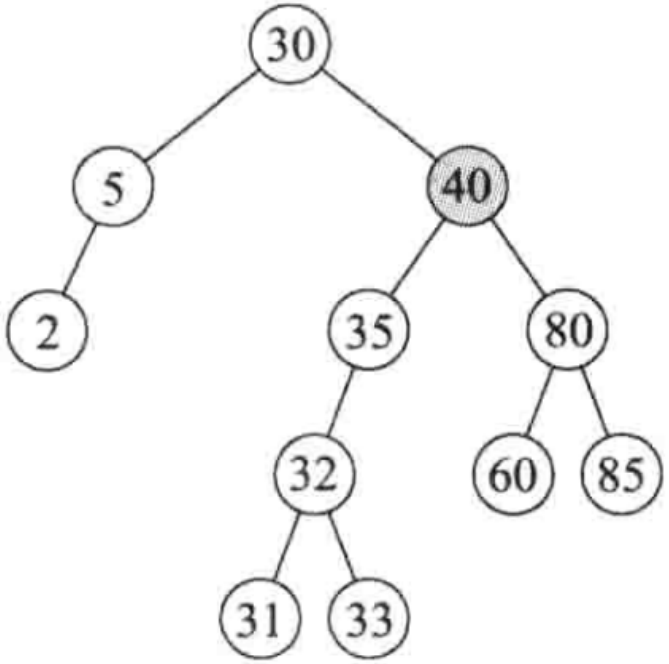


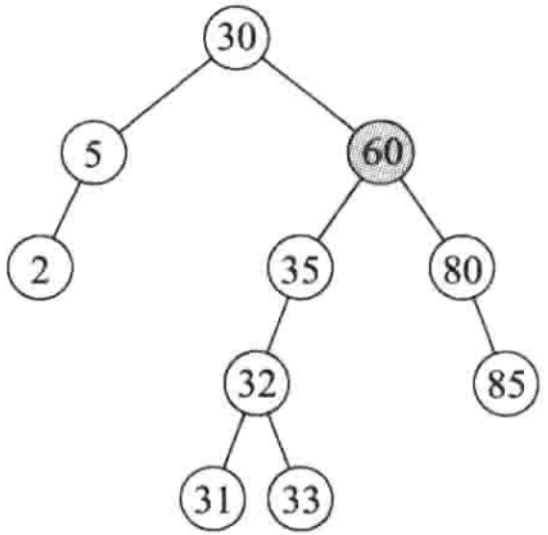
情况2：删除一个度为1的节点

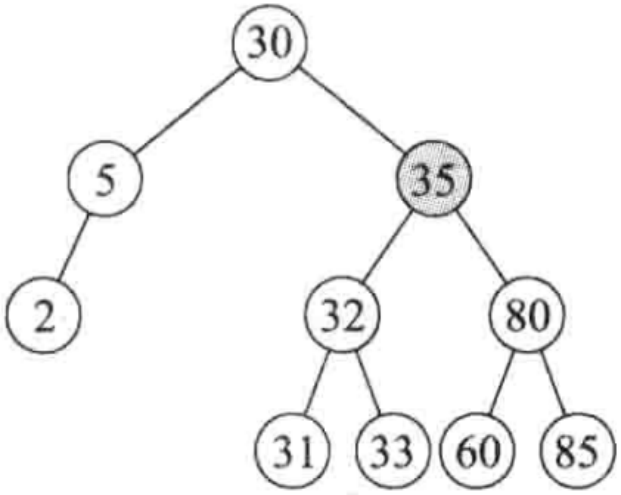


•例: 删除元素的关键字为25

情况3：删除一个度为2的节点

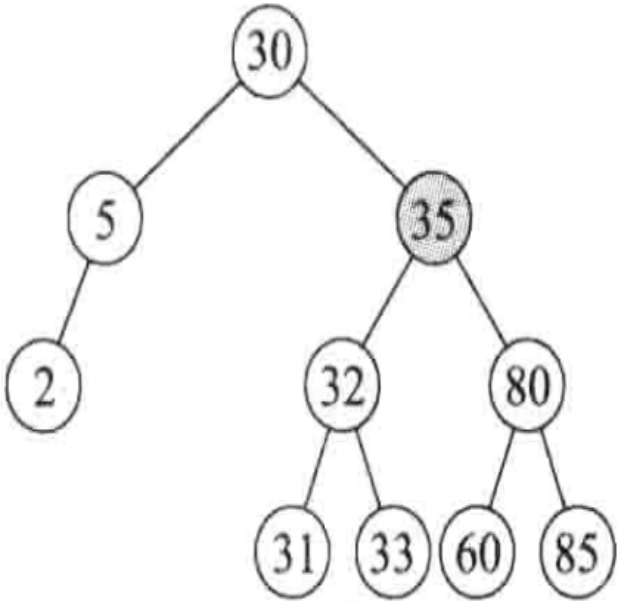


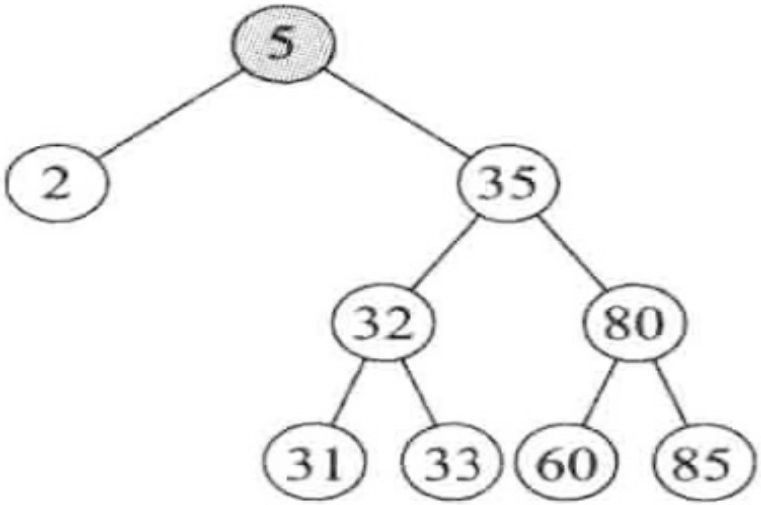




- 例：删除40

情况3：删除一个度为2的节点

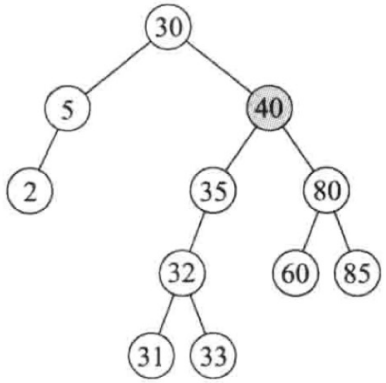




- 例：删除30

```
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



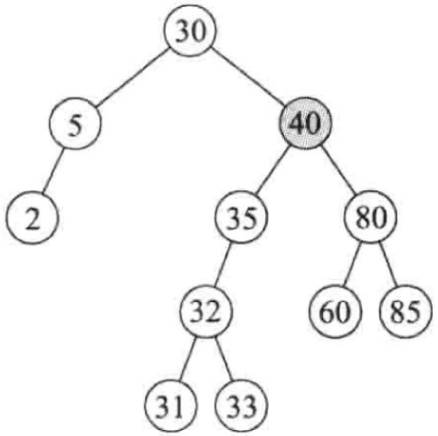

```
// 处理p有两个孩子的情形
if (p->leftChild && p->rightChild)
{//两个孩子，转换成有0或1个孩子的情形

  // 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
```

// 对树进行重构

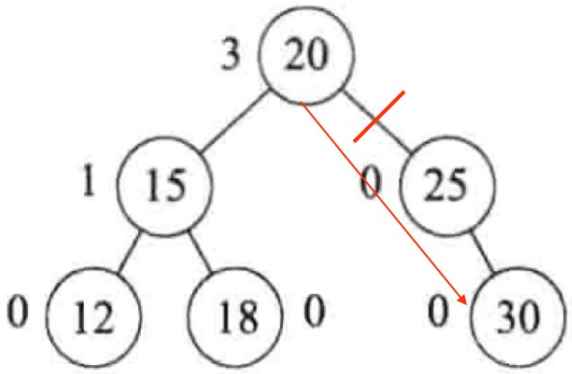
```
//与P关联的指针修改为与q关联
          //p指向新的删除节点s，pp为p的父节点
    if (pp == NULL)
       root = q;
    else if (p = = pp->leftChild)
            pp->leftChild = q;
     else   pp->rightChild = q;

   if  (ps == p) pp =q;
   else pp =ps;
   delete p;
   p = s;
}
```



```
// 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;
}
```



•例: 删除元素的关键字为25

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.
  - 高度：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)

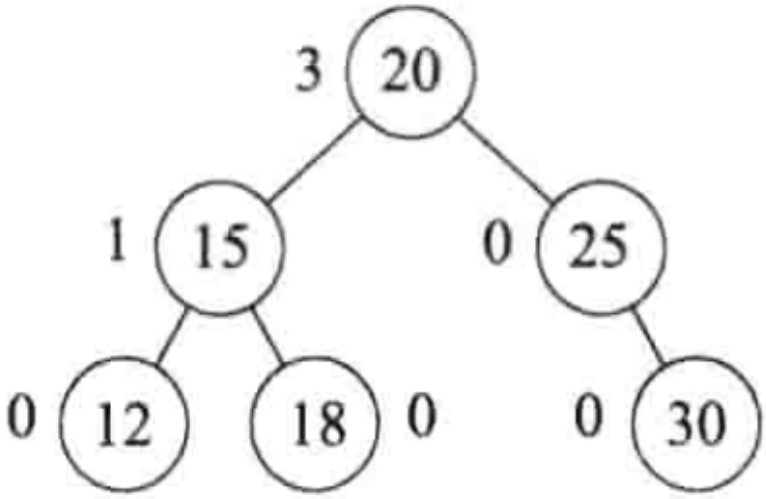
类dBinarySearchTree

- 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
```

索引搜索二叉树

- 查找索引为2的元素
  - 根的leftSize值为3，所以在根的左子树
  - 在左子树索引为2
  - 左子树的根15的leftSize值为1，所以在15的右子树索引为：2-（ leftSize +1）计算为0



应用

- 直方图问题
  - 输入由n个关键字所构成的集合，然后输出一个列表，它包含不同关键字及其每个关键字在集合中出现的次数（频率）。

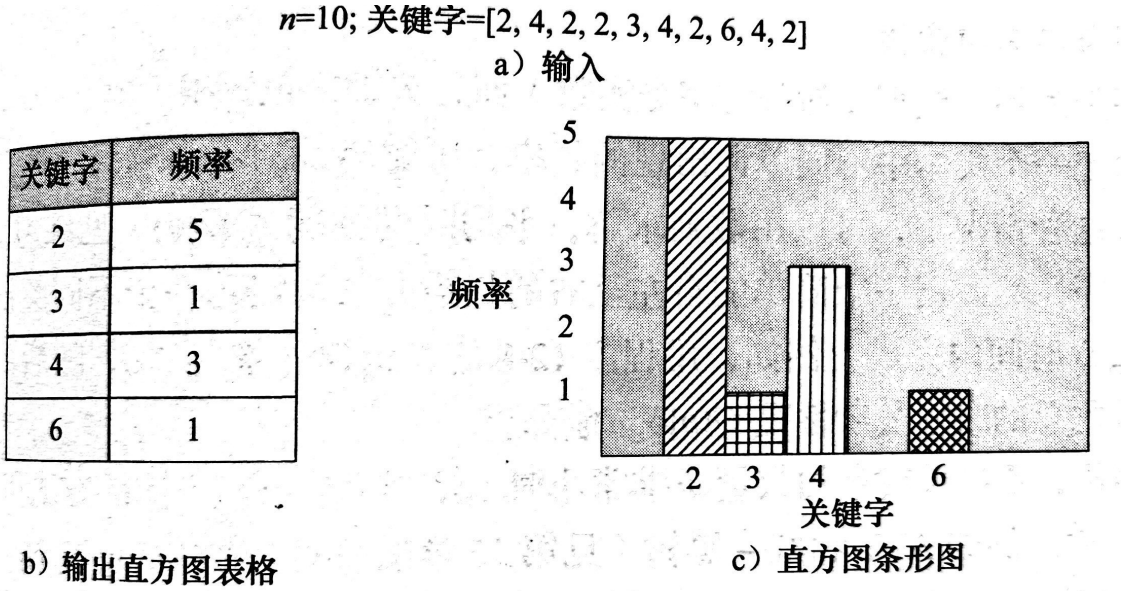


图 14-5　直方图举例

应用

- 简单直方图程序
  - 当关键字的值是0到r范围内的整数，且r的值足够小时。用数组元素h[i]代表关键字i的频率，比如关键字是小写字母，可以用映射
    [a, b, …, z] = [0, 1, …, 25]

```
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;
}
```

应用

- 直方图与二叉搜索树
  - 不是整型而且关键词范围很大时，可以用散列，平均时间性能O(n)
  - 当与n相比，不同关键字数量m非常小时，用排序求解直方图的方法可以进一步改进：
  - 用二叉搜索树平均复杂性为*O(nlogm)*
  - 类binarySearchTreeWithVisit，是类binarySearchTree的扩展，其中增加了以下公有函数：

```
void insert(const pair<const K,E>& thePair, void(*visit)(E&))
```

该函数将元素thePair插入搜索树，若存在对应关键字，就调用visit函数

应用

- 直方图与二叉搜索树

```
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++)
  {                                                         // 输入元素
    pair<int, int> thePair;
    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();
}
```

- 输入数据插入类型为binarySearchTreeWithVisit的对象，然后调用函数ascend输出直方图
- 二叉搜索树的每一个元素都有两个成员，第一个成员是关键字，第二个成员是关键字的频率

作业

- P346： 6，10

P321. 41. 补充说明

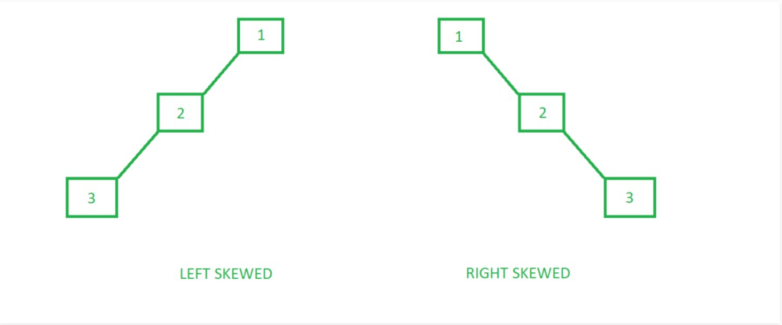
Suppose you are to code text that uses n symbols. A simple way to assign codes that satisfy the prefix property is to start with a right-skewed extended binary tree that has n external nodes, sort the n symbols in decreasing order of frequency $F()$, and assign symbols to external nodes so that an inorder listing of external nodes gives the symbols in decreasing order of their frequency. The sort step takes $O(n \log n)$ time, and the remaining steps take $O(n)$ time. So this method has the same asymptotic complexity as the optimal method described in Section 12.6.3.

Skewed Binary Tree

A skewed binary tree is a type of binary tree in which all the nodes have only either one child or no child.

Types of Skewed Binary trees

There are 2 special types of skewed tree:



1. Left Skewed Binary Tree:

These are those skewed binary trees in which all the nodes are having a left child or no child at all. It is a left side dominated tree. All the right children remain as null.