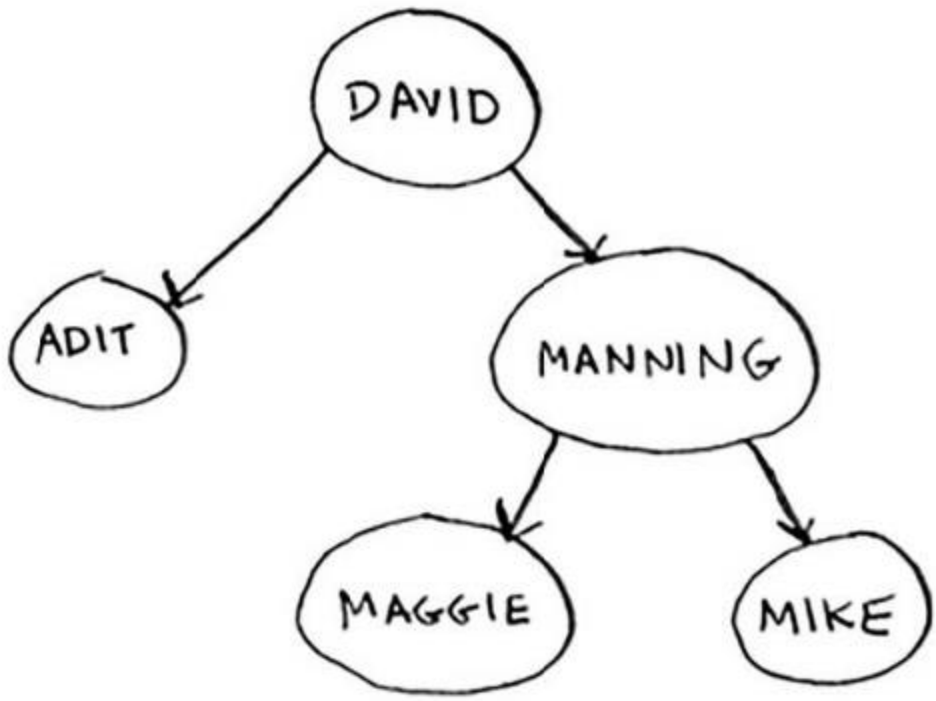
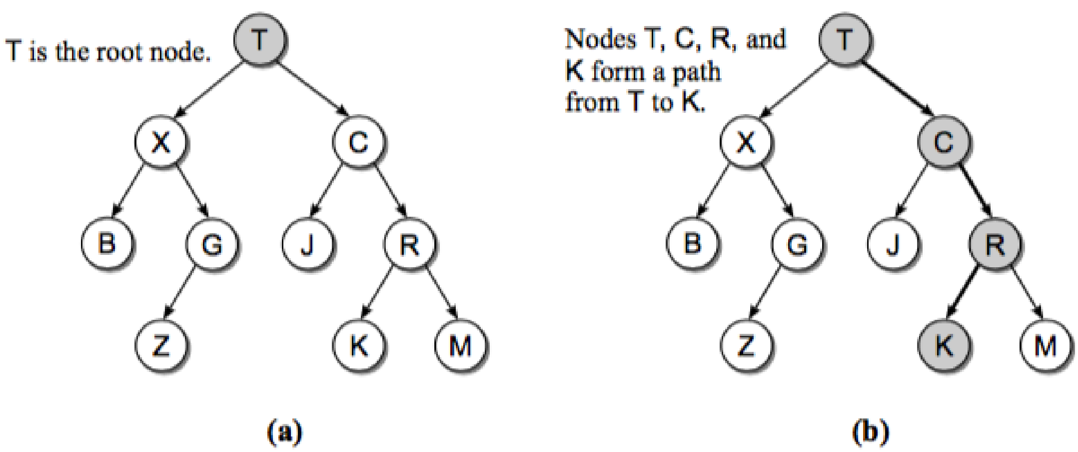
树(Tree)

树结构是一种包括节点(nodes)和边(edges)的拥有层级关系的一种结构，它的形式和家谱树非常类似：



如果你了解 Linux 文件结构（tree 命令），它的结构也是一棵树。我们快速看下树涉及到的一些概念：

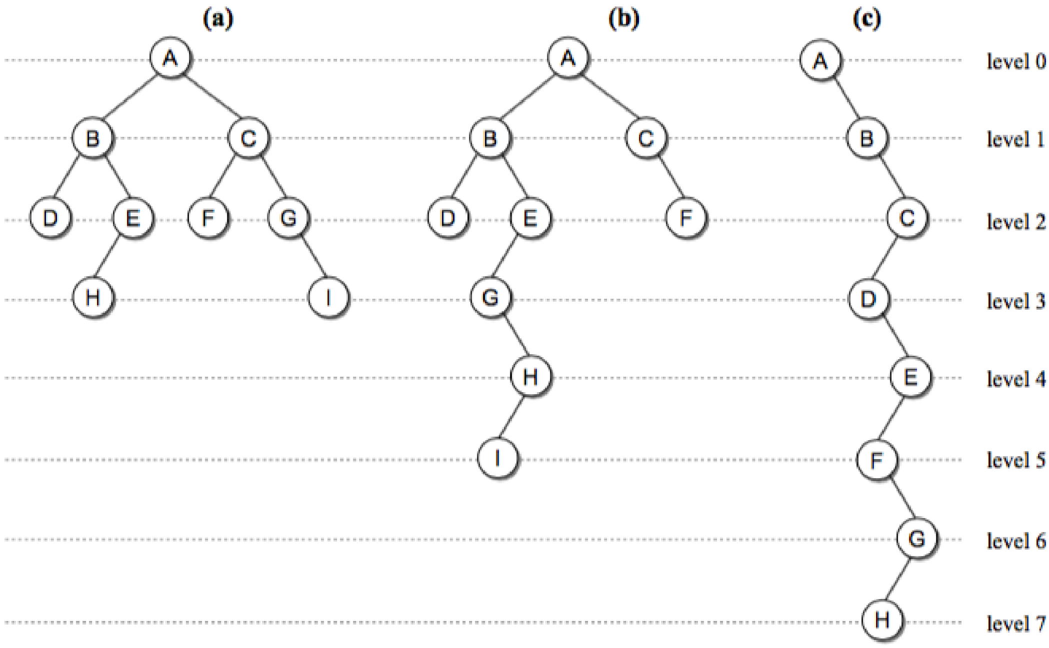


- 根节点(root)：树的最上层的节点，任何非空的树都有一个节点
- 路径(path)：从起始节点到终止节点经历过的边
- 父亲(parent)：除了根节点，每个节点的上一层边连接的节点就是它的父亲(节点)
- 孩子(children)：每个节点由边指向的下一层节点
- 兄弟(siblings)：同一个父亲并且处在同一层的节点

- 子树(subtree)：每个节点包含它所有的后代组成的子树
- 叶子节点(leaf node)：没有孩子的节点成为叶子节点

二叉树

了解完树的结构以后，我们来看树结构里一种简单但是却比较常用的树-二叉树。 二叉树是一种简单的树，它的每个节点最多只能包含两个孩子，以下都是一些合法的二叉树：



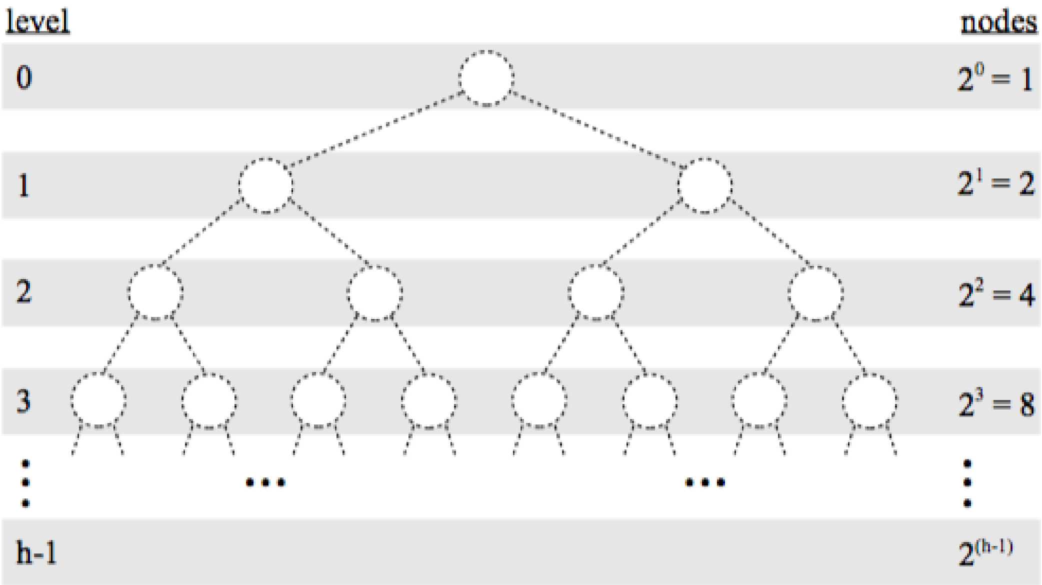


Figure 13.6: Possible slots for the placement of nodes in a binary tree.

通过上边这幅图再来看几个二叉树相关的概念:

- 节点深度(depth)：节点对应的 level 数字
- 树的高度(height)：二叉树的高度就是 level 数 + 1，因为 level 从 0 开始计算的
- 树的宽度(width)：二叉树的宽度指的是包含最多节点的层级的节点数
- 树的 size：二叉树的节点总个数。

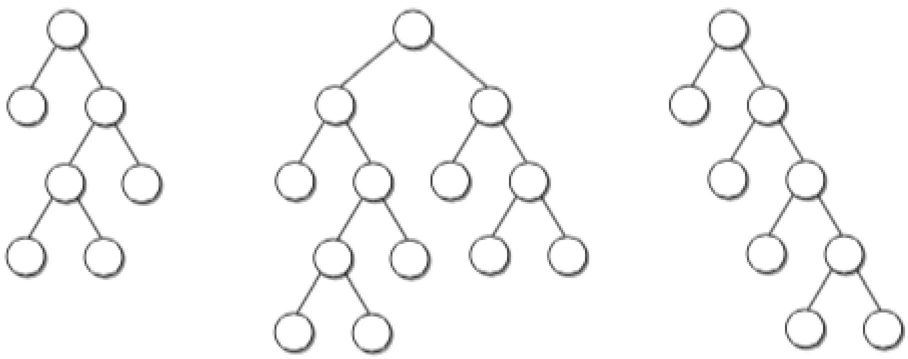
一棵 size 为 n 的二叉树高度最多可以是 n，最小的高度是 $ \lfloor lgn \rfloor + 1 $，这里 log 以 2 为底简写为 lgn，和算法导论保持一致。这个结果你只需要用高中的累加公式就可以得到。

一些特殊的二叉树

在了解了二叉树的术语和概念之后，我们来看看一些特殊的二叉树，后续章节我们会用到：

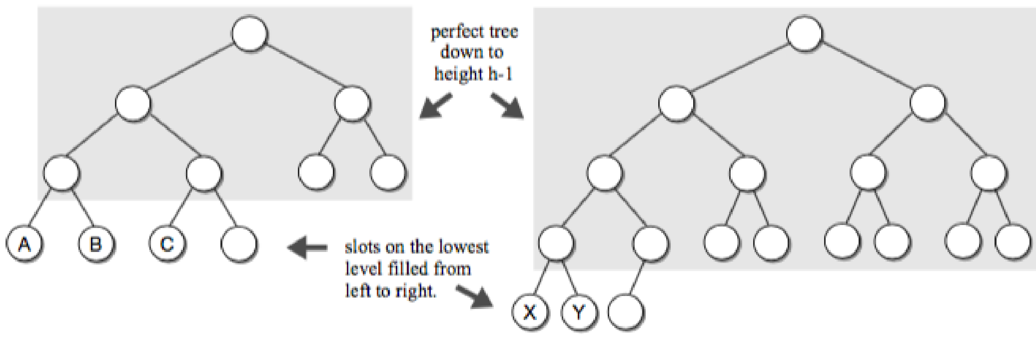
满二叉树(full binary tree)

如果每个内部节点（非叶节点）都包含两个孩子，就成为满二叉树。下边是一些例子，它可以有多种形状：



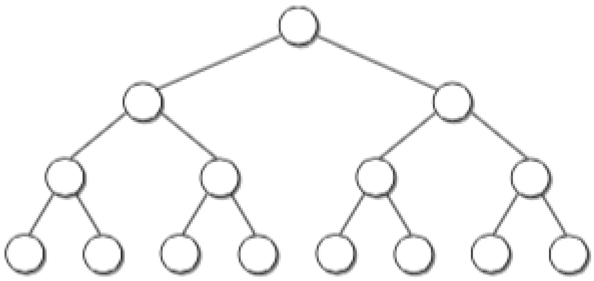
完全二叉树(complete binary tree)

当一个高度为 h 的完美二叉树减少到 h-1，并且最底层的槽被毫无间隙地从左到右填充，我们就叫它完全二叉树。 下图就是完全二叉树的例子：



完美二叉树(perfect binary tree)

当所有的叶子节点都在同一层就是完美二叉树，毫无间隙填充了 h 层。



二叉树的表示

那么怎么表示一棵二叉树呢？其实你发现会和链表有一些相似之处，一个节点，然后节点需要保存孩子的指针，我以构造下边这个二叉树为例子： 我们先定义一个类表示节点：

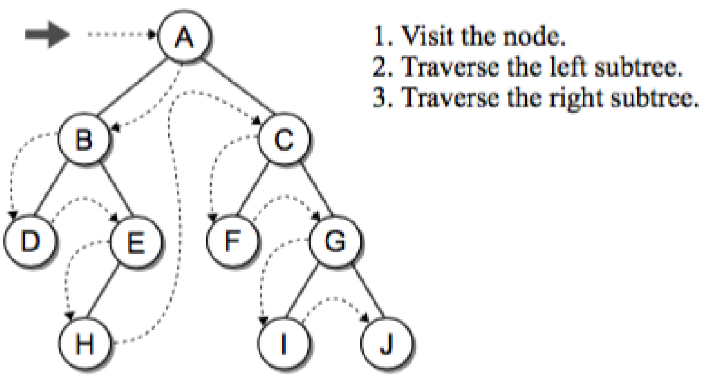


Figure 13.12: The logical ordering of the nodes with a preorder traversal.

```
class Node(object):
    def __init__(self, data, left=None, right=None):
        self.data, self.left, self.right = data, left, right
```

当然和链表类似，root 节点是我们的入口，于是乎定义一个二叉树：

```
class Tree(object):
    def __init__(self, root=None):
        self.root = root
```

怎么构造上图中的二叉树呢，似乎其他课本没找到啥例子(有些例子是写了一堆嵌套节点来定义，很难搞清楚层次关系)，我自己定义了一种方法，首先我们输入节点信息，仔细看下边代码，叶子节点的 left 和 right 都是 None，并且只有一个根节点 A：

```
node_list = [
    {'data': 'A', 'left': 'B', 'right': 'C', 'is_root': True},
    {'data': 'B', 'left': 'D', 'right': 'E', 'is_root': False},
    {'data': 'D', 'left': None, 'right': None, 'is_root': False},
    {'data': 'E', 'left': 'H', 'right': None, 'is_root': False},
    {'data': 'H', 'left': None, 'right': None, 'is_root': False},
    {'data': 'C', 'left': 'F', 'right': 'G', 'is_root': False},
    {'data': 'F', 'left': None, 'right': None, 'is_root': False},
    {'data': 'G', 'left': 'I', 'right': 'J', 'is_root': False},
    {'data': 'I', 'left': None, 'right': None, 'is_root': False},
    {'data': 'J', 'left': None, 'right': None, 'is_root': False},
]
```

然后我们给 BinTreeNode 定义一个 build_from 方法，当然你也可以定义一种自己的构造方法。

二叉树的遍历

不知道你有没有发现，二叉树其实是一种递归结构，因为单独拿出来一个 subtree 子树出来，其实它还是一棵树。那遍历它就很方便啦，我们可以直接用递归的方式来遍历它。但是当处理顺序不同的时候，树又分为三种遍历方式：

- 先(根)序遍历：先处理根，之后是左子树，然后是右子树
- 中(根)序遍历：先处理左子树，之后是根，最后是右子树
- 后(根)序遍历：先处理左子树，之后是右子树，最后是根

我们来看下实现，其实算是比较直白的递归函数：

```
def iter_node1(self, node):
        if node is not None:
            print(node.data)
            self.iter_node1(node.left)
            self.iter_node1(node.right)
```

二叉树层序遍历

除了递归的方式遍历之外，我们还可以使用层序遍历的方式。层序遍历比较直白，就是从根节点开始按照一层一层的方式遍历节点。 我们可以从根节点开始，之后把所有当前层的孩子都按照从左到右的顺序放到一个列表里，下一次遍历所有这些孩子就可以了。

```
    def iter_node2(self, node):
        node_list = [node]

        for node in node_list:
            print(node.data)
            if node.left:
                node_list.append(node.left)
            if node.right:
                node_list.append(node.right)
```

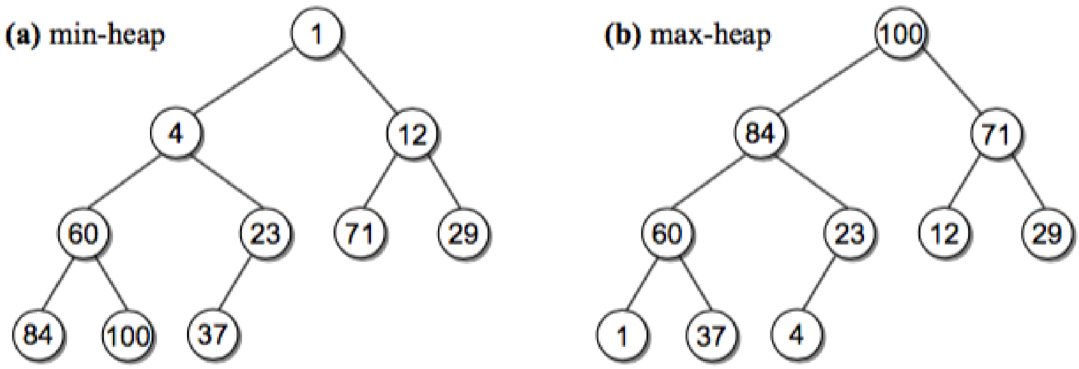
还有一种方式就是使用一个队列，之前我们知道队列是一个先进先出结构，如果我们按照一层一层的顺序从左往右把节点放到一个队列里， 也可以实现层序遍历：

堆(heap)

什么是堆？

堆是一种完全二叉树（请你回顾下上一章的概念），有最大堆和最小堆两种。

- 最大堆：对于每个非叶子节点 V，V 的值都比它的两个孩子大，称为 最大堆特性(heap order property) 最大堆里的根总是存储最大值，最小的值存储在叶节点。
- 最小堆：和最大堆相反，每个非叶子节点 V，V 的两个孩子的值都比它大。



堆的操作

堆提供了很有限的几个操作：

- 插入新的值。插入比较麻烦的就是需要维持堆的特性。需要 sift-up 操作，具体会在视频和代码里解释，文字描述起来比较麻烦。
- 获取并移除根节点的值。每次我们都可以获取最大值或者最小值。这个时候需要把底层最右边的节点值替换到 root 节点之后 执行 sift-down 操作。

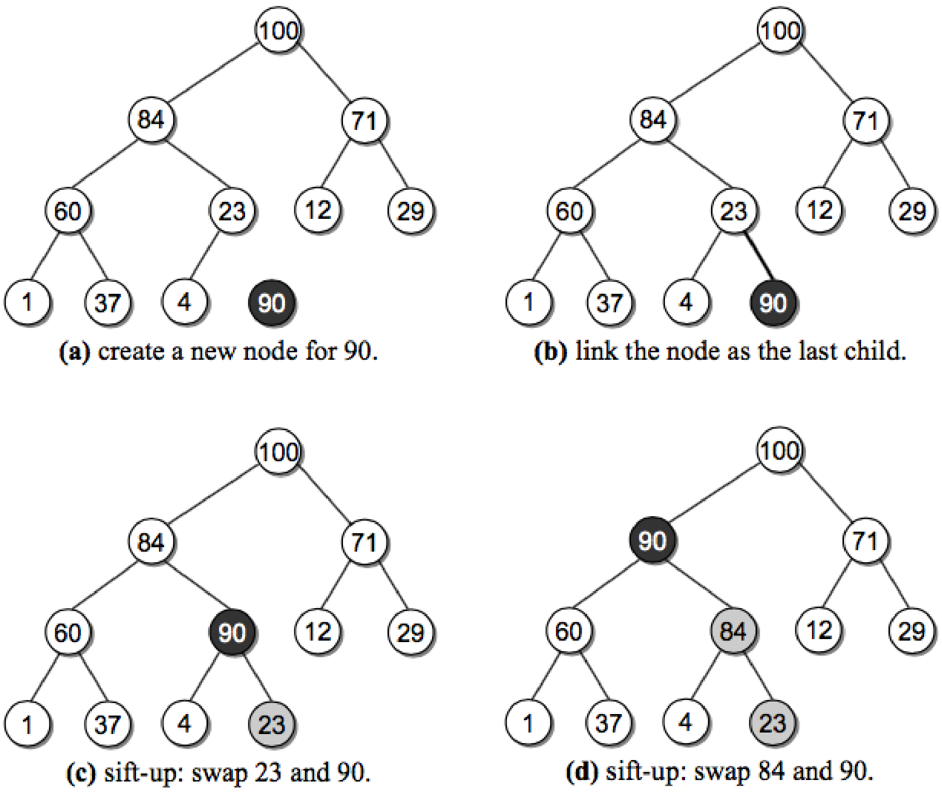


Figure 13.23: The steps to insert value 90 into the heap.

堆的表示

之前我们用一个节点类和二叉树类表示树，这里其实用数组就能实现堆。

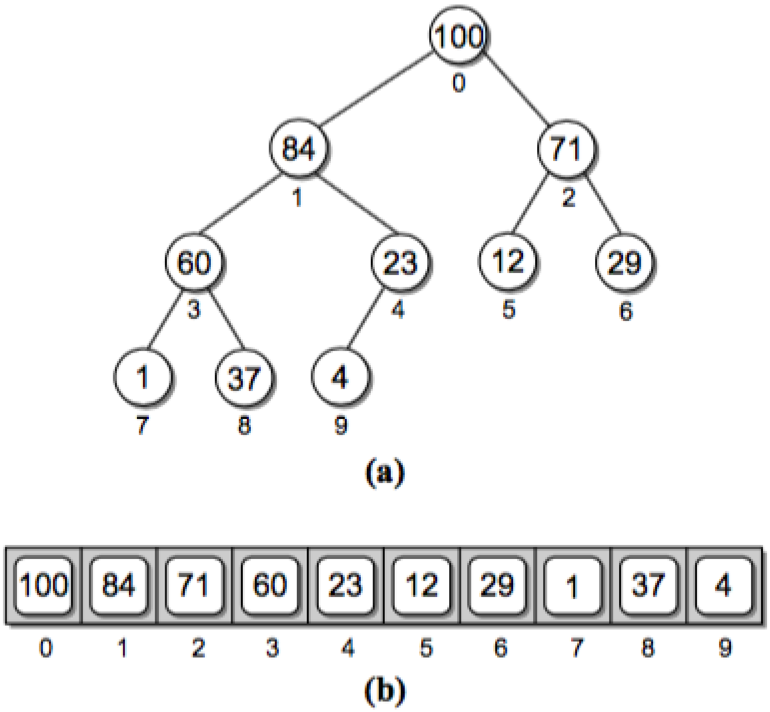


Figure 13.27: A heap can be implemented using an array or vector.

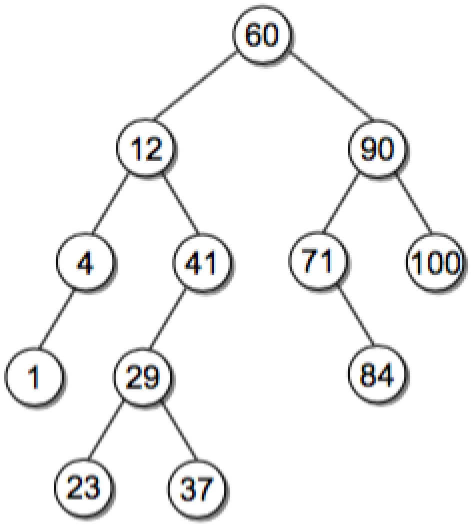
仔细观察下，因为完全二叉树的特性，树不会有间隙。对于数组里的一个下标 i，我们可以得到它的父亲和孩子的节点对应的下标：

```
parent = int((i-1) / 2)    # 取整
left = 2 * i + 1
right = 2 * i + 2
```

超出下标表示没有对应的孩子节点。

二叉查找树(BST)

二叉树的一种应用就是来实现堆，我们再看看用二叉查找树(Binary Search Tree，BST)。 前面有章节说到了查找操作，包括线性查找、二分查找、哈希查找等，线性查找效率比较低，二分又要求必须是有序的序列， 为了维持有序插入的代价比较高、哈希查找效率很高但是浪费空间。能不能有一种插入和查找都比较快的数据结构呢？二叉查找树就是这样一种结构，可以高效地插入和查询节点。



BST 定义

二叉查找树是这样一种二叉树结构，它的每个节点的左子节点小于该节点，每个节点的右子节点小于该节点。

构造一个 BST

我们还像之前构造二叉树一样，按照上图构造一个 BST 用来演示：

```
class BST(object):
    def __init__(self, root=None):
        self.root = root

    @classmethod
    def build_from(cls, node_list):
```

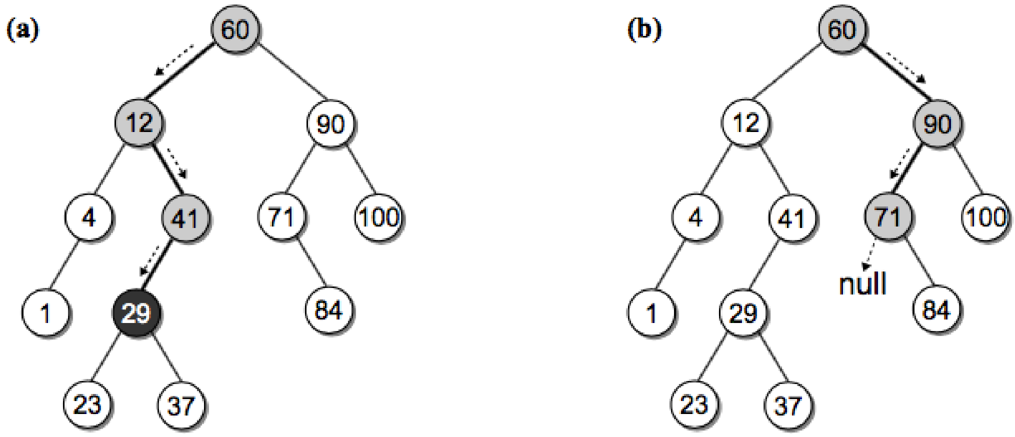
```
        cls.size = 0
        key_to_node_dict = {}
        for node_dict in node_list:
            key = node_dict['key']
            key_to_node_dict[key] = BSTNode(key)

        for node_dict in node_list:
            key = node_dict['key']
            node = key_to_node_dict[key]
            if node_dict['is_root']:
                root = node
            node.left = key_to_node_dict.get(node_dict['left'])
            node.right =
key_to_node_dict.get(node_dict['right'])
            cls.size += 1
        return cls(root)


NODE_LIST = [
    {'key': 60, 'left': 12, 'right': 90, 'is_root': True},
    {'key': 12, 'left': 4, 'right': 41, 'is_root': False},
    {'key': 4, 'left': 1, 'right': None, 'is_root': False},
    {'key': 1, 'left': None, 'right': None, 'is_root': False},
    {'key': 41, 'left': 29, 'right': None, 'is_root': False},
    {'key': 29, 'left': 23, 'right': 37, 'is_root': False},
    {'key': 23, 'left': None, 'right': None, 'is_root': False},
    {'key': 37, 'left': None, 'right': None, 'is_root': False},
    {'key': 90, 'left': 71, 'right': 100, 'is_root': False},
    {'key': 71, 'left': None, 'right': 84, 'is_root': False},
    {'key': 100, 'left': None, 'right': None, 'is_root':
False},
    {'key': 84, 'left': None, 'right': None, 'is_root': False},
]
bst = BST.build_from(NODE_LIST)
```

BST 操作

查找

如何查找一个指定的节点呢，根据定义我们知道每个内部节点左子树的 key 都比它小，右子树的 key 都比它大，所以 对于带查找的节点

search_key，从根节点开始，如果 search_key 大于当前 key，就去右子树查找，否则去左子树查找。 一直到当前节点是 None 了说明没找到对应 key。



好，撸代码：

```
def _bst_search(self, subtree, key):
    if subtree is None:   # 没找到
        return None
    elif key < subtree.data:
        return self._bst_search(subtree.left, key)
    elif key > subtree.data:
        return self._bst_search(subtree.right, key)
    else:
        return subtree

def get(self, key, default=None):
    node = self._bst_search(self.root, key)
    if node is None:
        return default
    else:
        return node.value
```

获取最大和最小 key 的节点

其实还按照其定义，最小值就一直向着左子树找，最大值一直向右子树找，递归查找就行。

```
def _bst_min_node(self, subtree):
    if subtree is None:
        return None
    elif subtree.left is None:   # 找到左子树的头
```

```
            return subtree
        else:
            return self._bst_min_node(subtree.left)

    def bst_min(self):
        node = self._bst_min_node(self.root)
        return node.value if node else None
```

插入

插入节点的时候我们需要一直保持 BST 的性质，每次插入一个节点，我们都通过递归比较把它放到正确的位置。 你会发现新节点总是被作为叶子结点插入。（请你思考这是为什么）

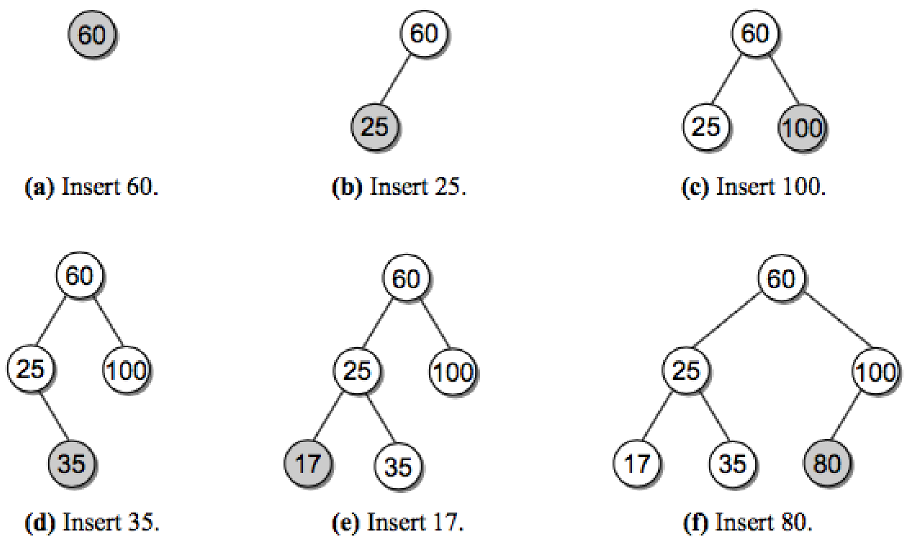


Figure 14.5: Building a binary tree by inserting the keys [60, 25, 100, 35, 17, 80]

```
def _bst_insert(self, subtree, data):
        """ 插入并且返回根节点

        :param subtree:
        :param key:
        :param value:
        """
        if subtree is None:   # 插入的节点一定是根节点，包括 root 为空的情况
            subtree = BSTNode(data)
        elif data < subtree.data:
            subtree.left = self._bst_insert(subtree.left,data)
        elif data > subtree.data:
            subtree.right = self._bst_insert(subtree.right,data)
```

```python
        return subtree

    def add(self, data):
        node = self._bst_search(self.root, data)
        if node is not None:   # 更新已经存在的 key
            return False
        else:
            self.root = self._bst_insert(self.root, data)
            self.size += 1
            return True
```

删除节点

删除操作相比上边的操作要麻烦很多，首先需要定位一个节点，删除节点后，我们需要始终保持 BST 的性质。 删除一个节点涉及到三种情况：

- 节点是叶节点
- 节点有一个孩子
- 节点有两个孩子

我们分别来看看三种情况下如何删除一个节点：

删除叶节点

这是最简单的一种情况，只需要把它的父亲指向它的指针设置为 None 就好。

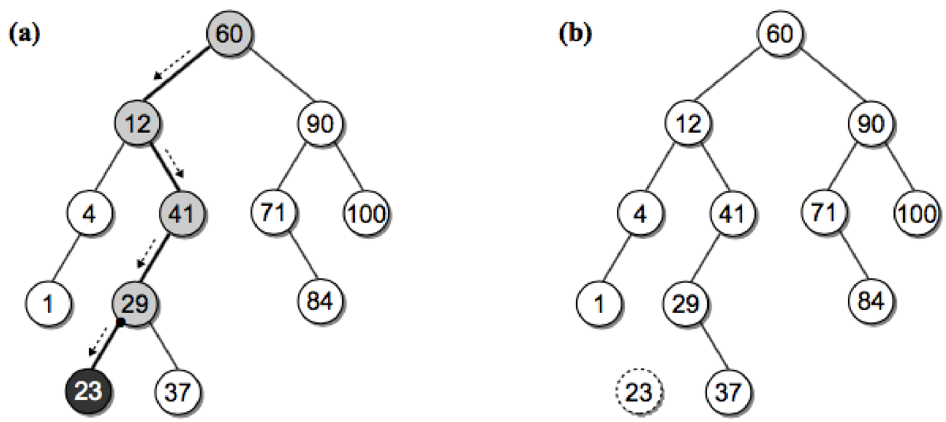


Figure 14.8: Removing a leaf node from a binary search tree: (a) finding the node and unlinking it from its parent; and (b) the tree after removing 23.

删除只有一个孩子的节点

删除有一个孩子的节点时，我们拿掉需要删除的节点，之后把它的父亲指向它的孩子就行，因为根据 BST 左子树都小于节点，右子树都大于节点的特性，删除它之后这个条件依旧满足。

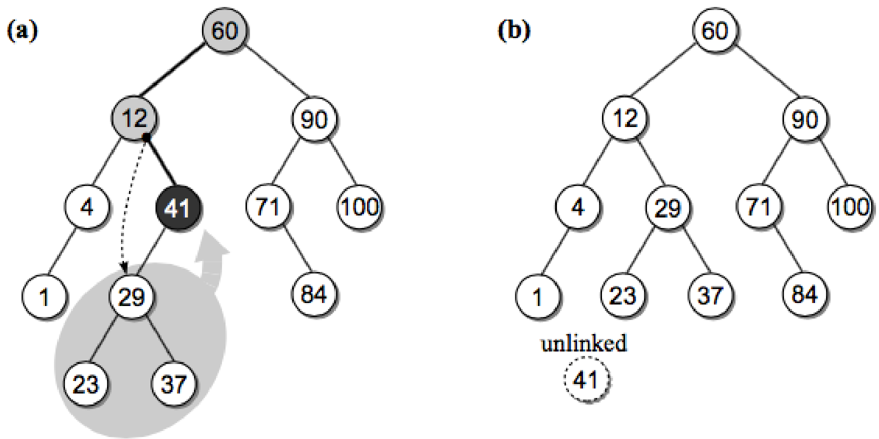


Figure 14.10: Removing an interior node (41) with one child: (a) redirecting the link from the node's parent to its child subtree; and (b) the tree after removing 41.

删除有两个孩子的内部节点

假如我们想删除 12 这个节点改怎么做呢？你的第一反应可能是按照下图的方式：

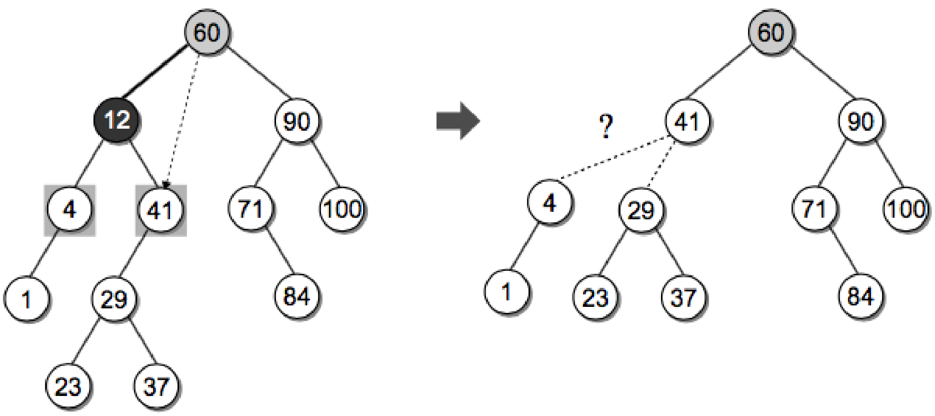


Figure 14.11: Attempting to remove an interior node with two children by replacing the node with one of its children.

但是这种方式可能会影响树的高度，降低查找的效率。这里我们用另一种非常巧妙的方式。 还记得上边提到的吗，如果你中序遍历 BST 并且输出每个节点的 key，你会发现就是一个有序的数组。 [1 4 12 23 29 37 41 60 71 84 90 100]。这里我们定义两个概念，逻辑前任(predecessor)和后继(successor)，请看下图:

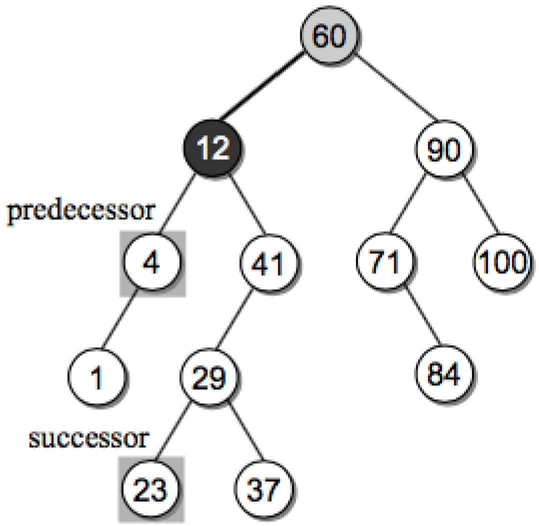


Figure 14.12: The logical successor and predecessor of node 12.

12 在中序遍历中的逻辑前任和后继分别是 4 和 23 节点。于是我们还有一种方法来删除 12 这个节点：

- 找到待删除节点 N(12) 的后继节点 S(23)
- 复制节点 S 到节点 N
- 从 N 的右子树中删除节点 S，并更新其删除后继节点后的右子树

说白了就是找到后继并且替换，这里之所以能保证这种方法是正确的，你会发现替换后依旧是保持了 BST 的性质。 有个问题是如何找到后继节点呢？待删除节点的右子树的最小的节点不就是后继嘛，上边我们已经实现了找到最小 key 的方法了。

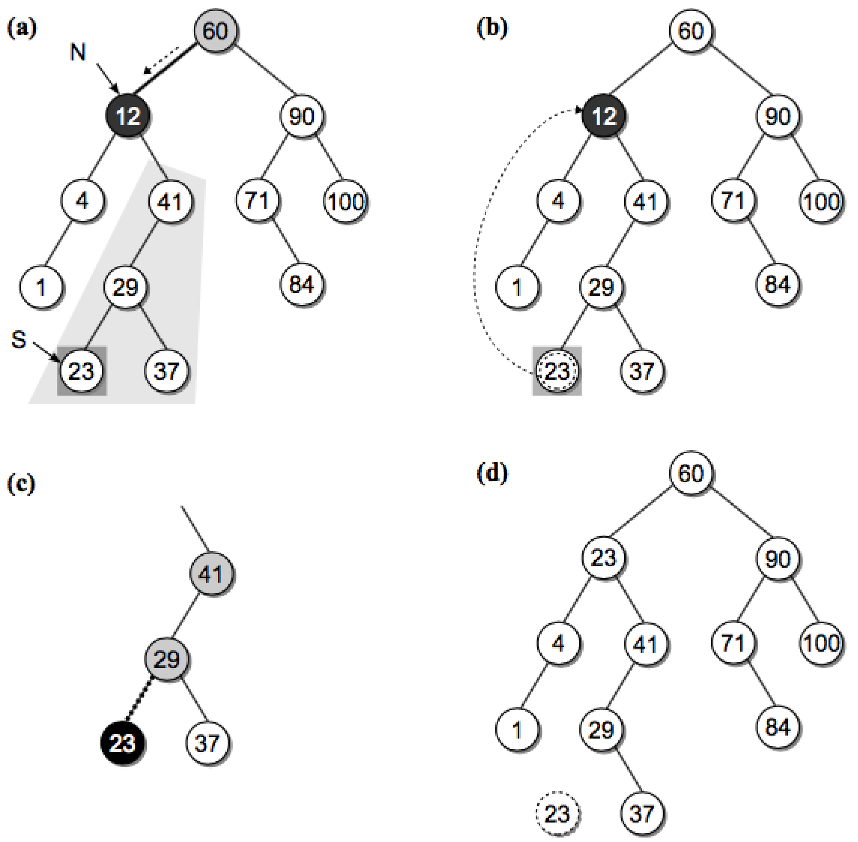


Figure 14.13: The steps in removing a key from a binary search tree: (a) find the node, N, and its successor, S; (b) copy the successor key from node N to S; (c) remove the successor key from the right subtree of N; and (d) the tree after removing 12.