第 15 讲

二分搜索树

# 简介

在本讲中，我们将继续考虑实现集合（或关联数组）接口的方法。这一次，我们将使用二分搜索树来实现这个接口。我们最终将能够实现插入和查找的 O(logn) 最坏情况渐近复杂度。这也扩展到删除，尽管我们不会在本课中讨论该操作。我们的学习目标如下：

**计算思维：**我们发现二叉树是一种组织信息的方式。我们将我们过去研究过的排序概念叠加到它们上，作为获得指数级加速的一种方式。

**算法和数据结构：**我们将二分搜索树作为一种空间高效且可扩展的数据结构，对于许多感兴趣的操作具有潜在的对数复杂性——我们将在下一节课中看到如何保证这个复杂度上界。

**编程：**我们为二叉树定义了一种类型，并使用递归作为实现规范函数及其操作的便捷方法。

# 有序关联数组

哈希表是关联数组，数组中数据组织在一个索引处，该索引由使用哈希函数的键确定。 如果散列函数很好，这意味着元素将被放置在一个合理的随机位置，分布在整个数组中。 如果不好，需要线性搜索来定位元素。

有许多实现关联数组的替代方法。 例如，我们可以将元素存储在一个按键排序的数组中。 这样使用二分搜索的查找将是 O(logn)，但插入将是 O(n)，因为它需要 O(logn) 步骤才能找到正确的位置，但随后需要 O(n) 步才能通过移动所有更大的元素为该新元素腾出空间。 （我们还需要像在无界数组中一样增长数组，以确保它不会耗尽容量。）数组不够灵活，无法快速插入，但我们将在本讲中设计的数据结构将是可以快速插入的。

# 抽象二分搜索

我们需要哪些操作才能执行二分搜索？我们需要一种方法来比较我们正在寻找的键和数据结构中给定元素的键。根据该比较的结果，如果相同则二分搜索完成返回该元素的位置，如果我们要查找的内容较小，则向左前进，或者如果我们要查找的内容较大，则向右前进。对于复杂度 O(logn) 的二分搜索，重要的是二分搜索一次向左或向右前进许多步，而不仅仅是一个元素。事实上，如果我们遵循抽象的二分搜索原则，从数组的中间开始，但只在数组中前进一个索引，我们将获得线性搜索，其复杂度为 O(n)，而不是 O(logn)。

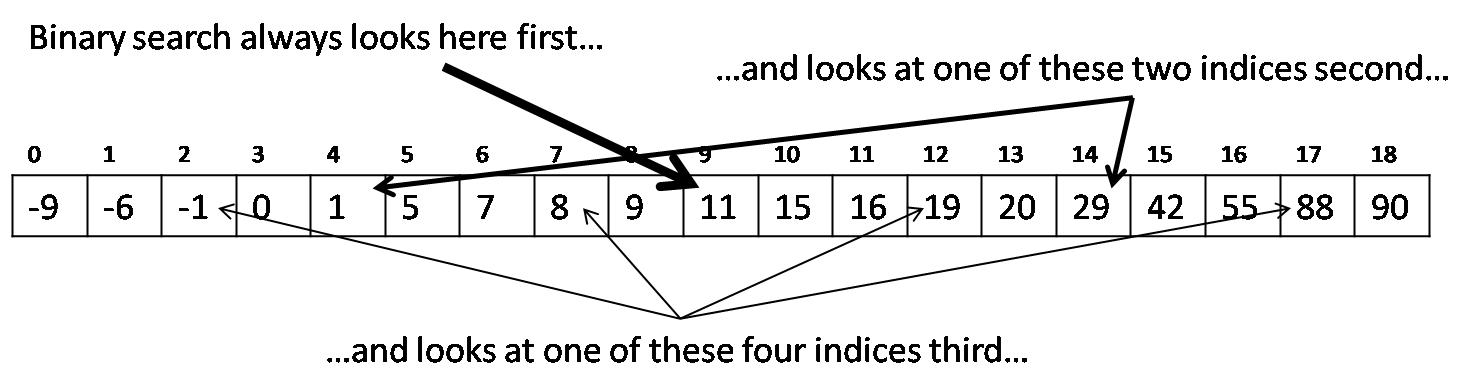
因此，二分搜索需要一种比较键的方法和一种非常快速地跨过数据结构元素的方法，或者向左（朝向具有较小键的元素）或向右（朝向较大的元素）。在我们研究过的基于数组的二分搜索中，每次迭代都会计算一个中点

**int** mid = lower + (upper - lower) / 2;

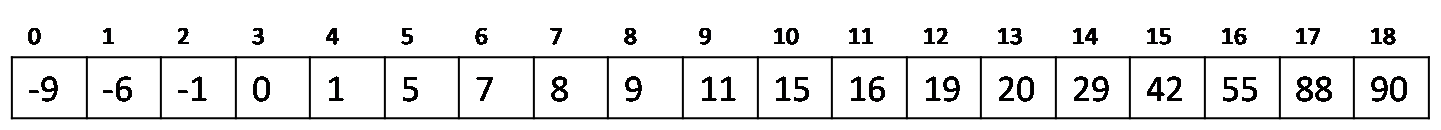
下一次迭代的新界限是（如果我们要搜索的键小于中间的元素）upper = mid; 或（如果键更大）lower = mid + 1;

所以我们知道下一个值 mid 将是 (lower + mid) / 2 或 ((mid + 1) + upper) / 2 （忽略溢出的可能性）。

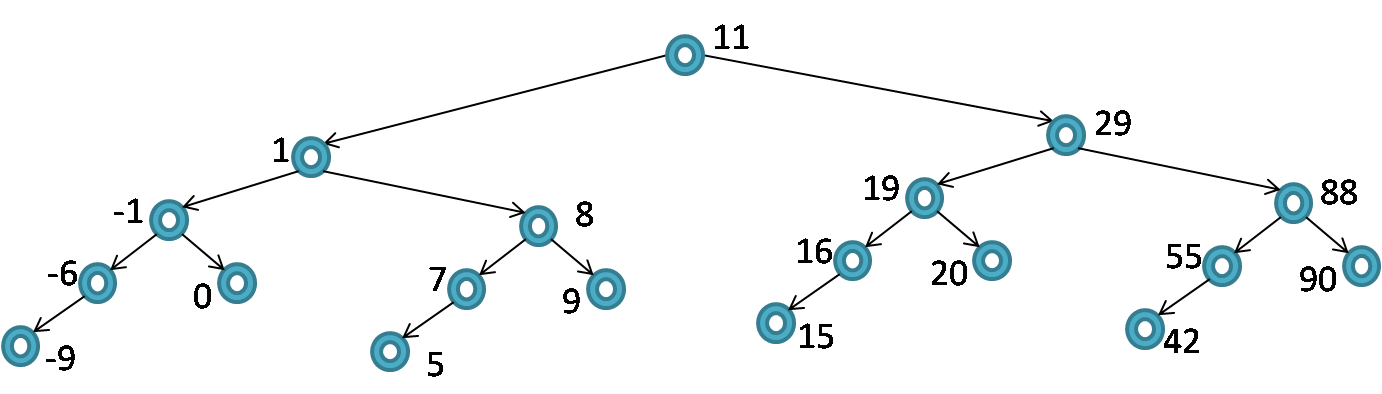
这种模式继续下去，给定任何排序数组，我们可以枚举所有可能的二分搜索：



这种模式意味着对任意索引处的数组元素的常数时间访问对于进行二分搜索不是必需的！ 要对上面的数组进行二分查找，我们只需要从数组索引 9（包含 11）到数组索引 4 和 14（分别包含 1 和 29）的常数时间访问，从数组索引 4 到数组索引 2 的常数时间访问 和 7，依此类推。 在二分搜索的每一点，我们知道我们的搜索最多会以两种方式中的一种进行，因此我们将使用指针结构明确表示这些选择，从而得到二叉树的结构。 我们通过在这个数组上运行二分搜索得到的树结构......



...对应于这个二叉树：



# 用指针表示二叉树

为了使用指针表示二叉树，我们使用具有两个指针的结构：一个指向左孩子，一个指向右孩子。 如果没有孩子，则指针为 NULL。 树的叶子是具有两个 NULL 指针的节点。

**typedef struct** tree\_node tree;

**struct** tree\_node {

elem data; tree\* left; tree\* right;

};

与我们用于哈希表的完全通用的数据实现不同，为了简单起见，我们假设客户向我们提供了一种已知为指针的 elem 类型和单一函数 elem\_compare。

/\* Client-side interface \*/

// typedef \_\_\_\_\_\_\* elem;

**int** elem\_compare(elem k1, elem k2)

/\*@**requires** k1 != NULL && k2 != NULL; @\*/

/\*@**ensures** -1 <= \result && \result <= 1; @\*/ ;

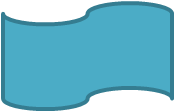
我们要求 elem 类型的有效值为非 NULL。 — 实际上，基于树的关联数组的实现将使用 NULL 来表示元素不存在。

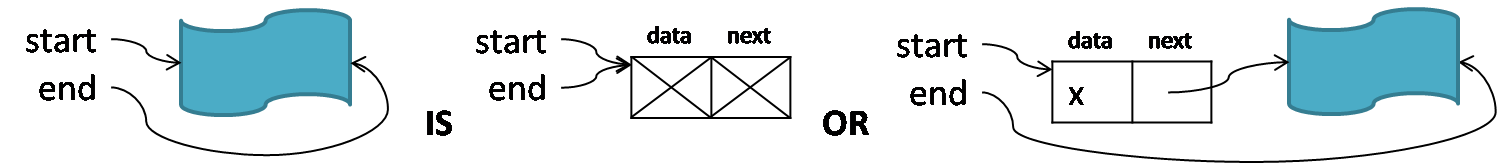
客户提供的elem\_compare函数与我们用于哈希表的等价函数不同。对于二分搜索树，我们需要比较键 k1 和 k2 并确定 k1 < k2、k1 = k2 还是 k1 > k2。一个常见的方法是比较函数返回一个整数 r，其中 r < 0 表示 k1 < k2，r = 0 表示 k1 = k2，r > 0 表示 k1 > k2。我们的约定规定我们期望 elem\_compare 只返回 -1、0 或 1。

树是我们见过的第二种递归数据结构：一个树节点有两个包含指向树节点的指针的字段。到目前为止，我们只将递归数据结构视为链表，可以是哈希表中的链，也可以是堆栈或队列中的链表段。

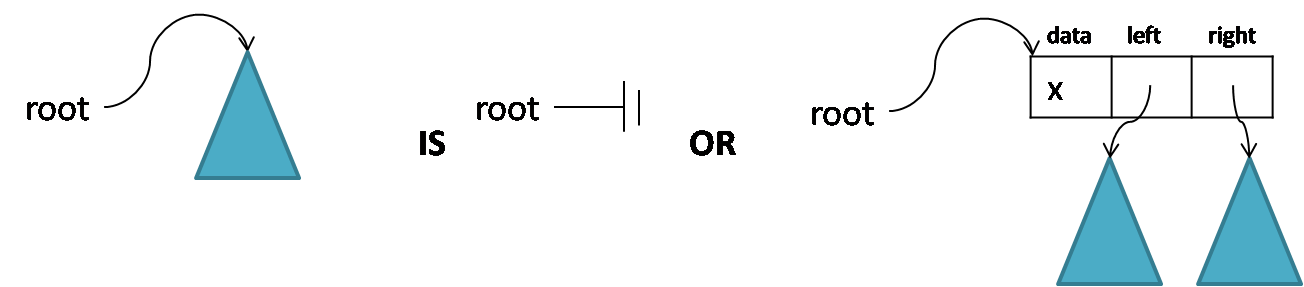
让我们记住我们如何描绘列表段。任何列表段都由两个指针引用：start 和 end，并且该列表的构造方式有两种可能性，这两种可能性都需要 start 为非空（并且 start->data 也满足我们对 elem 值的约束） .

1. **bool** is\_segment(list\* start, list\* end) {
2. **if** (start == NULL) **return** false;
3. **if** (start->data == NULL) **return** false;
4. **if** (start == end) **return** true;
5. **return** is\_segment(start->next, end);
6. }

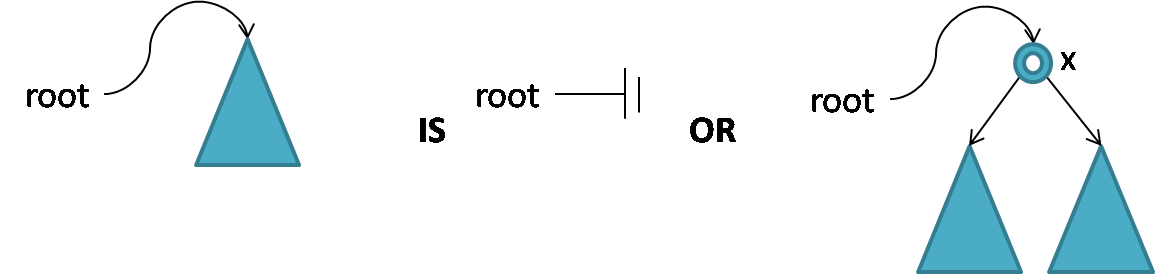
我们可以通过使用样的表示任意段的图片来以图形方式表示这些选择。 然后我们知道每个段都有一两种形式：



我们将为树创建类似的图：不包含元素的树为 NULL，非空树是具有三个字段的结构：数据和左右指针，它们本身就是树。



我们通常会使用一种更像图的方式来呈现树，而不是显式地绘制出带有三个字段的 tree\_node 结构：



这个递归定义可以直接编码成一个非常简单的数据结构不变性is\_tree。 它检查的很少：只是所有数据字段都是非空的，正如客户端接口所要求的那样。 如果它终止，它也确保没有循环； 一个循环会导致不终止，就像 is\_segment 一样。

1. **bool** is\_tree(tree\* root) {
2. **if** (root == NULL) **return** true;
3. **return** root->data != NULL
4. && is\_tree(root->left) && is\_tree(root->right);
5. }

## 排序不变性

只有对数组进行了排序，二分搜索才对数组正确。 这样我们才知道，如果我们要查找的元素小于中间元素，则不看数组的上半部分是可以的，因为在排序后的数组中， 如果有的话，它只能出现在下半部分。 为了使二分搜索在二叉搜索树上正常工作，我们需要保持相应的数据结构不变：节点右侧的所有元素的键都大于该节点的键。 并且该节点左侧的所有节点的键都小于该节点的键。 这种排序不变性是二叉搜索树的核心思想； 这就是使二叉树成为二叉搜索树的原因。

**排序不变性**。 在二叉搜索树中任何键为 k 的节点上，左子树中元素的所有键都严格小于 k，而右子树中元素的所有键都严格大于 k。

这个不变性意味着在树中没有键出现多次，我们必须确保我们的插入函数保持这个不变性。

我们暂时不会编写用于检查排序不变性的代码，因为这非常困难。 我们将首先讨论二叉搜索树的查找和插入函数。

# 查找某个键

排序不变性让我们可以在二叉搜索树中找到元素 e，就像我们在更抽象的树数据结构上使用二分搜索找到元素一样。 这是一个搜索的递归算法，从树的根开始：

1. 如果树是空的，停止。

2.将当前节点的key k与e进行比较。 如果相等就停止。

3. 如果 e 小于 k，则继续搜索左子树。

4. 如果 e 大于 k，则继续搜索右子树。

以下这个搜索的实现体现了上面的非正式描述。 回想一下，如果 x < y，elem\_compare(x,y) 返回 -1，如果 x = y，则返回 0，如果 x > y，则返回 1。

1. **bool** tree\_lookup(tree\* T, elem x)
2. //@**requires** is\_tree(T);
3. {
4. **if** (T == NULL) **return** false;
5. **int** cmp = elem\_compare(x, T->data);
6. **if** (cmp == 0) {
7. **return** true;
8. } **else if** (cmp < 0) {
9. **return** tree\_lookup(T->left, x);
10. } **else** {
11. //@assert cmp > 0;
12. **return** tree\_lookup(T->right, x);
13. }
14. }

我们在这里选择了一个递归实现，遵循树的结构，但实际上循环版本也可能是一个合理的选择（见练习 1）。

# 复杂性

如果我们的二叉搜索树是完美平衡的，即每个子树的左侧节点数与右侧节点数相同，那么排序不变性将确保搜索具有给定键的元素具有渐近复杂度 O(logn )，其中 n 是树中元素的数量。 每次我们将元素 x 与完美平衡树的根进行比较时，我们要么停止，要么丢弃树中的一半元素。

一般来说，我们可以说查找的成本是 O(h)，其中 h 是树的高度。 我们将高度定义为从根开始的任何指针序列可以达到的最大节点数。 一棵空树的高度为 0，而有两个孩子的树的高度为任一孩子的最大高度加 1。

# 接口

在我们讨论二叉搜索树的插入之前，我们应该指定接口并讨论我们将如何实现它。 请记住，我们假设 elem 的单客户定义和 elem\_compare 的单客户定义，而不是使用 void 指针和函数指针的完全通用版本。

/\* Library interface \*/

// typedef \_\_\_\_\_\_\* bst\_t;

bst\_t bst\_new()

/\*@**ensures** \result != NULL; @\*/ ;

**void** bst\_insert(bst\_t B, elem x)

/\*@**requires** B != NULL && x != NULL; @\*/ ;

**bool** bst\_lookup(bst\_t B, elem x)

/\*@**requires** B != NULL && x != NULL; @\*/ ;

我们不能将 bst\_t 定义为 tree\*，原因有两个。 一个原因是新树应该是空的，但空树由指针 NULL 表示，这将违反 bst\_new 后置条件。 更根本的原因是，如果 NULL 是一棵空树的表示，那么就没有办法编写一个函数来强制在树中插入额外的元素。 这是因为函数调用会复制作为参数传递的值。

这里通常的解决方案是我们已经用于堆栈、队列和哈希表的解决方案：我们有一个标头，在这种情况下，它只包含一个指向树根的指针。 我们经常在这些标头中保留与数据结构相关的其他信息，例如大小。

1. **typedef struct** bst\_header bst;
2. **struct** bst\_header {
3. tree\* root; 4 };

5

1. **bool** is\_bst(bst\* B) {
2. **return** B != NULL && is\_tree(B->root);
3. }

在二叉搜索树中查找则只调用我们已经定义的递归函数：

1. **bool** bst\_lookup(bst\* B, elem x)
2. //@**requires** is\_bst(B) && x != NULL;
3. {
4. **return** tree\_lookup(B->root, x);
5. }

is\_bst 和 is\_tree 之间以及 bst\_lookup 和 tree\_lookup 之间的关系是共同的。 非递归函数is\_bst被赋予非递归结构bst\_header，然后在树节点的递归结构上调用递归辅助函数is\_tree。

# 插入某个元素

使用标头结构，可以直接实现 bst\_insert。 我们只是像我们正在寻找给定的元素一样进行。 如果我们找到一个具有等价元素的节点，我们只需覆盖它的数据字段。 否则，我们将新键插入它应该存在的位置，如果它最初就在那里。 然而，这最后一句造成了一个小困难。 当我们到达一个空指针（这表明键不在树中）时，我们无法替换它指向的内容（它没有指向任何东西！）。 相反，我们返回新树，以便父节点可以修改自身。

1. tree\* tree\_insert(tree\* T, elem x)
2. //@**requires** is\_tree(T) && x != NULL;
3. //@**ensures** is\_tree(\result);
4. {
5. **if** (T == NULL) {
6. /\* create new node and return it \*/
7. T = **alloc**(**struct** tree\_node);
8. T->data = x;
9. T->left = NULL; // Not required (initialized to NULL)
10. T->right = NULL; // Not required (initialized to NULL)
11. **return** T;
12. } **else** {
13. **int** cmp = elem\_compare(x, T->data);
14. **if** (cmp == 0) {
15. T->data = x;
16. } **else if** (cmp < 0) {
17. T->left = tree\_insert(T->left, x);
18. } **else** {
19. //@assert cmp > 0;
20. T->right = tree\_insert(T->right, x);
21. }
22. }

38

1. **return** T;
2. }

返回的子树也将存储为新的根：

1. **void** bst\_insert(bst\* B, elem x)
2. //@**requires** is\_bst(B)
3. //@**requires** x != NULL;
4. //@**ensures** is\_bst(B);
5. {
6. B->root = tree\_insert(B->root, x);
7. }

# 检查排序不变性

当我们分析实现搜索和插入的递归函数的结构时，我们很想尝试定义一个简单（但错误的！） 二叉树的排序不变性如下：树 T 是排序的，只要

1. T 为空，或

2. T在根节点有key k，TL为左子树，TR为右子树，

• TL 为空，或者TL 的key 小于k 且TL 是有序的； 和

• TR 为空，或者TR 的key 大于k 并且TR 是有序的。

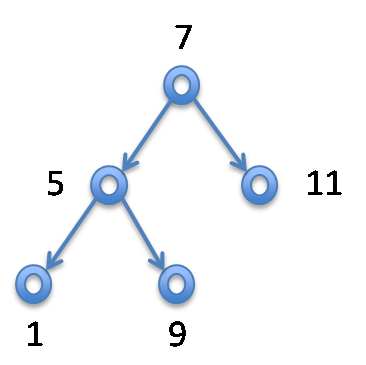
这将产生以下代码：

1. /\* THIS CODE IS BUGGY \*/
2. **bool** is\_ordered(tree\* T) {
3. **if** (T == NULL) **return** true; /\* an empty tree is a BST \*/
4. elem k = T->data;
5. **return** (T->left == NULL
6. || (elem\_compare(T->left->data), k) < 0
7. && is\_ordered(T->left)))
8. && (T->right == NULL
9. || (elem\_compare(k, T->right->data)) < 0
10. && is\_ordered(T->right)));

60 }

虽然这对于二叉搜索树应该总是正确的，但它远弱于在本次课开始时陈述的排序不变性。 在继续阅读之前，您应该检查您对该不变性的理解，以举出一个满足上述条件但违反排序不变性的树。

这实际上存在不止一个问题。 最明显的是以下树将通过此测试：



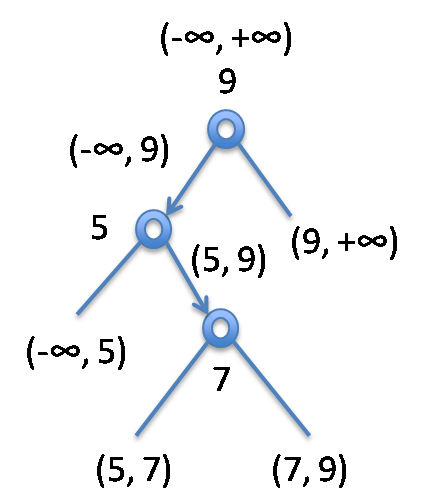
即使在局部看，左侧节点的键总是更小而右侧总是更大，其实键为 9 的节点位于错误的位置，用我们的搜索算法我们不会找到它，因为我们会在根的右子树中查找。

考虑不变性的另一种方法如下。 假设我们在一个带有键 k 的节点。

1. 如果我们去左子树，我们在子树中的键上建立一个上界：它们都必须小于 k。

2. 如果我们去右子树，我们在子树中的键上建立一个下界：它们都必须大于 k。

然后一般的想法是递归地遍历树，并为树中的所有键传递一个具有下界和上界的区间。 下图说明了这个想法。 我们从根以无限制的区间开始，写为 (−∞,+∞)，允许任何键。 像往常一样，我们将区间写为 (x,z) = {y | x < y 和 y < z}。 在叶子上，我们写出子树的区间。 例如，如果节点的左子树的键为 7，则其所有键都必须在区间 (5,7) 内。



实现这个想法的唯一困难是无界区间，上面写为 -∞ 和 +∞。 以下是一种可能的解决方案：我们不仅传递键值，还传递，我们可以从中提取键的界的特定元素。 由于 elem 必须是指针类型，这允许我们在没有下界或上界的情况下传递的是 NULL。

1. **bool** is\_ordered(tree\* T, elem lower, elem upper) {
2. **if** (T == NULL) **return** true;
3. **return** T->data != NULL
4. && (lower == NULL || elem\_compare(lower, T->data) < 0)
5. && (upper == NULL || elem\_compare(T->data, upper) < 0)
6. && is\_ordered(T->left, lower, T->data)
7. && is\_ordered(T->right, T->data, upper);
8. }

这将检查我们之前的 is\_tree 检查的所有属性，因此我们可以在is\_ordered 基础上实现 is\_tree：

1. **bool** is\_tree(tree\* T) {
2. **return** is\_ordered(T, NULL, NULL);
3. }

62

1. **bool** is\_bst(bst B) {
2. **return** B != NULL && is\_tree(B->root);
3. }

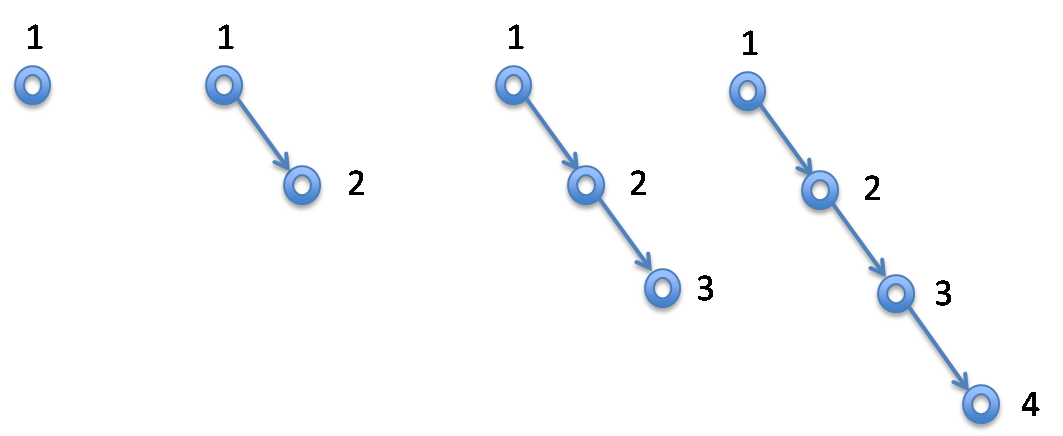
需要注意的是：函数 tree\_insert 的 is\_ordered(T, NULL, NULL) 前置和后置条件实际上不足以证明递归函数的正确性。 类似的注意适用于 tree\_lookup。 这是因为缺少上下界信息。

我们将在课程的稍后部分回到这个问题。

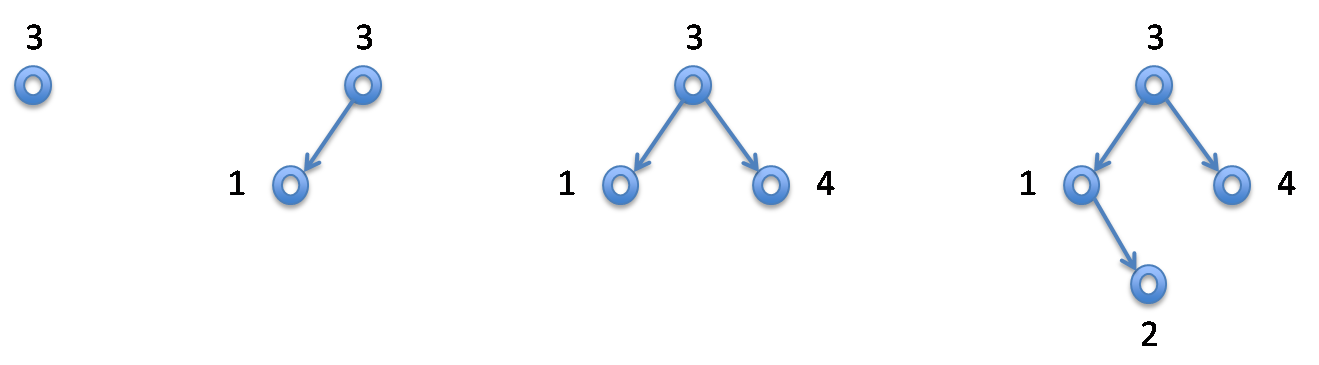
# 二叉搜索树的形状

我们已经提到平衡二叉搜索树具有良好的属性，例如插入和搜索的对数时间复杂度。 问题是二叉搜索树是否会平衡。 这取决于插入的顺序。 考虑插入数字 1、2、3 和 4。

如果我们按递增顺序插入它们，我们将依次获得以下树。



同样，如果我们按降序插入它们，我们会得到一条一直向左的直线。 如果我们按 3、1、4、2 的顺序插入，我们将获得以下二叉搜索树序列：



显然，最后一棵树更加平衡。在极端情况下，如果我们按顺序或逆序插入带有键的元素，则树将是线性的，n 项的搜索时间将为 O(n)。

这些观察意味着注意树的平衡非常重要。我们将在后面的课程中讨论保持二叉搜索树平衡的方法。

# 练习

**练习 1**. 将 tree\_lookup 重写为循环而不是递归。

**练习 2.** 将 tree\_insert 重写为循环而不是递归。 [提示：当我们替换一个为空的节点时，困难将是更新父节点中的指针。为此，我们可以保留一个“尾随”指针，它应该是当前正在考虑的节点的父节点。]

**练习 3**. 二叉搜索树接口只期望客户提供一个用于键比较的函数：

int elem\_compare(elem k1, elem k2);

相反，另一种设计是期望客户端提供一组 elem 比较函数，每个结果一个：

bool elem\_equal(elem k1, elem k2);

bool elem\_greater(elem k1, elem k2);

bool elem\_less(elem k1, elem k2);

这种设计的优点和缺点是什么？