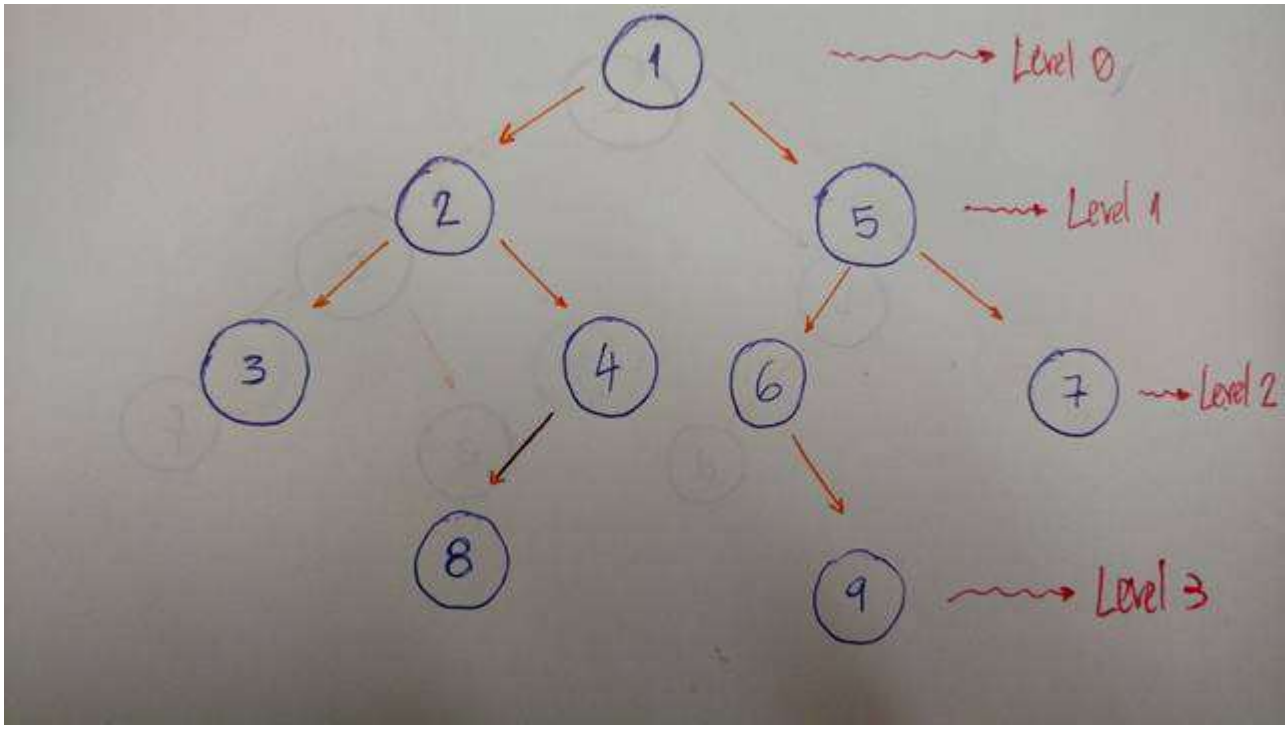
树

广度优先搜索（BFS）

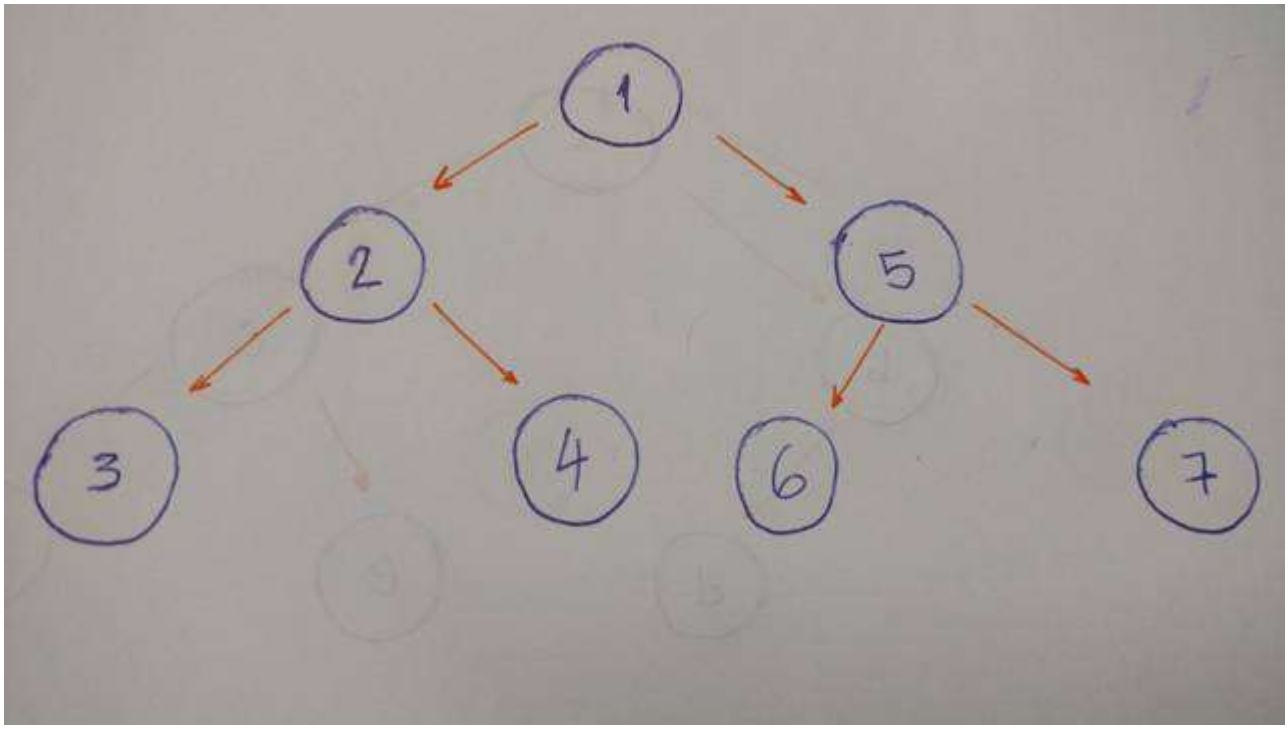
二叉搜索树

广度优先搜索（BFS）

BFS是一层层逐渐深入的遍历算法。



下面这个例子是用来帮我们更好的解释该算法。



我们来一层一层的遍历这棵树。本例中，就是1-2-5-3-4-6-7.

- 0层/深度0：只有值为1的节点
- 1层/深度1：有值为2和5的节点
- 2层/深度2：有值为3、4、6、7的节点

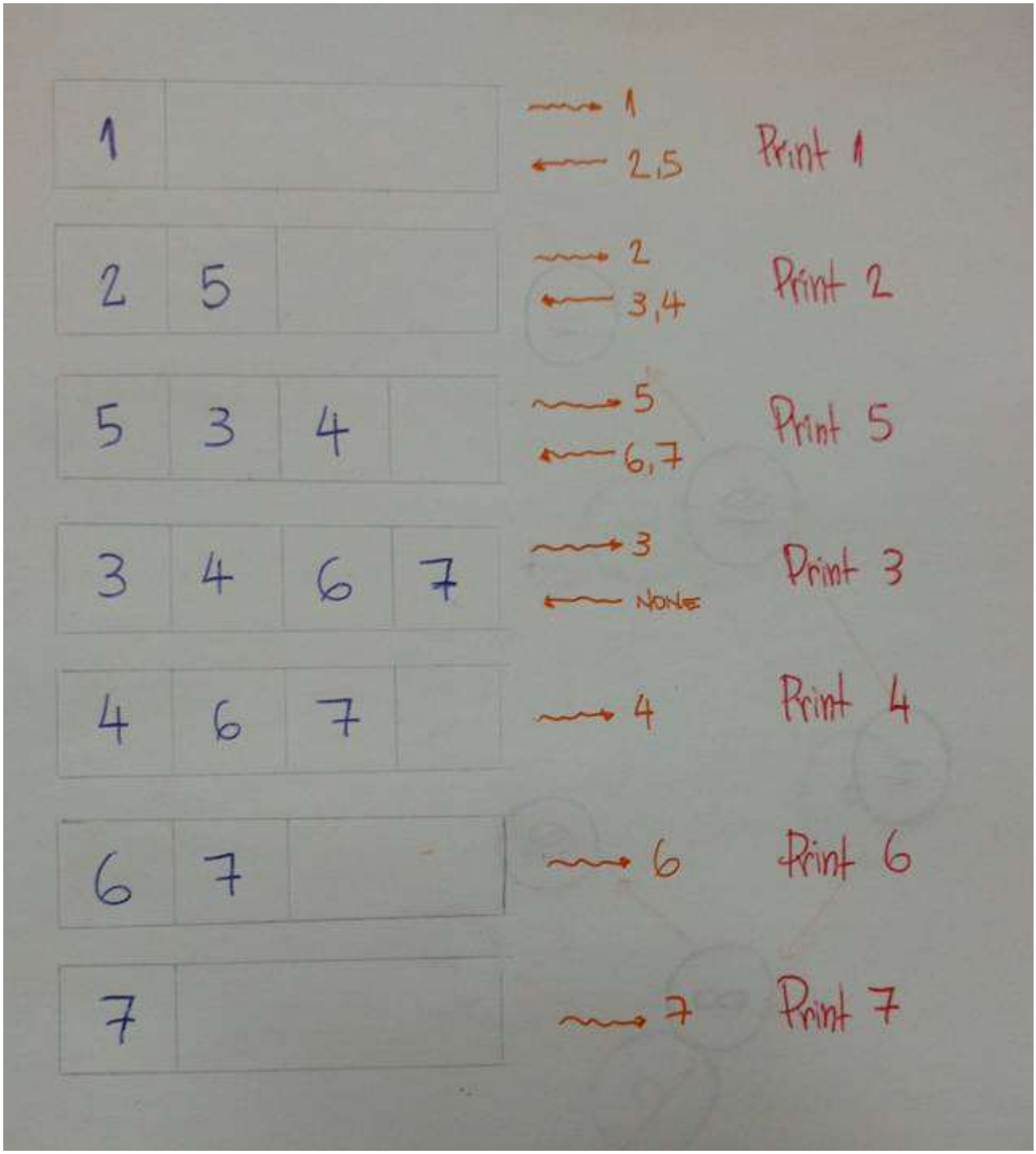
好，下面我们来编写实现代码。

```
def bfs(self):    queue = Queue()
    queue.put(self)

    while not queue.empty():
        current_node = queue.get()
        print(current_node.value)

        if current_node.left_child:
            queue.put(current_node.left_child)

        if current_node.right_child:
            queue.put(current_node.right_child)
```

为了实现BFS算法，我们需要用到一个数据结构，那就是队列。

队列具体是用来干什么的呢?

请看下面解释。

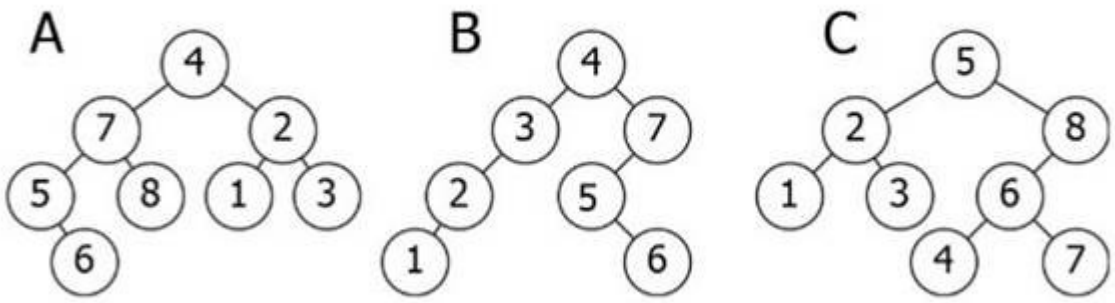


1. 首先用put方法将根节点添加到队列中。
2. 当队列不为空时迭代。
3. 获取队列中的第一个节点，然后输出其值。
4. 将其左孩子和右孩子添加到队列（如果它有孩子的话）。
5. 在队列的帮助下我们将每一个节点的值一层层的输出。

二叉搜索树

二叉搜索树有时候被称为二叉有序树或二叉排序树，二叉搜索树的值存储在有序的顺序中，因此，查找表和其他的操作可以使用折半查找原理。——Wikipedia

二叉搜索树中的一个重要性质是，二叉搜索树中一个节点的值大于其左子树任一节点的值，但是小于其右子树任一节点的值。



以下是上述插图的详解：

A 是反的二叉搜索树。子树 7-5-8-6应该在右边，而子树2-1-3 应该在左边。

B 是唯一正确的选项。它满足二叉搜索树的性质。

C 有一个问题：值为4的那个节点应该在根节点的左边，因为这个节点的值比根节点的值5小。

让我们用代码实现一个二叉搜索树！

现在是时候开始写代码了！

我们要干点什么？我们会插入一个新的节点，搜索一个值，删除一些节点以及二叉搜索树的平衡。

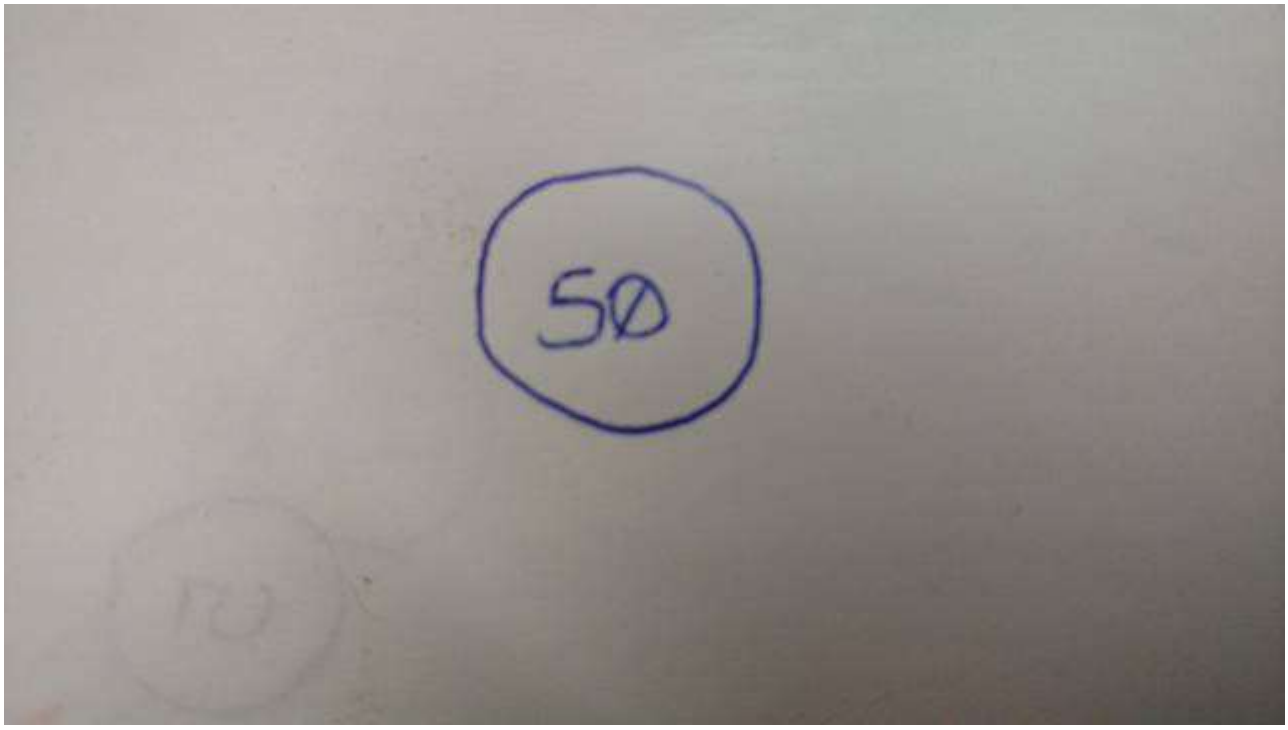
让我们开始吧！

插入：向我们的树添加新的节点

现在想像一下我们有一棵空树，我们想将几个节点添加到这棵空树中，这几个节点的值为：

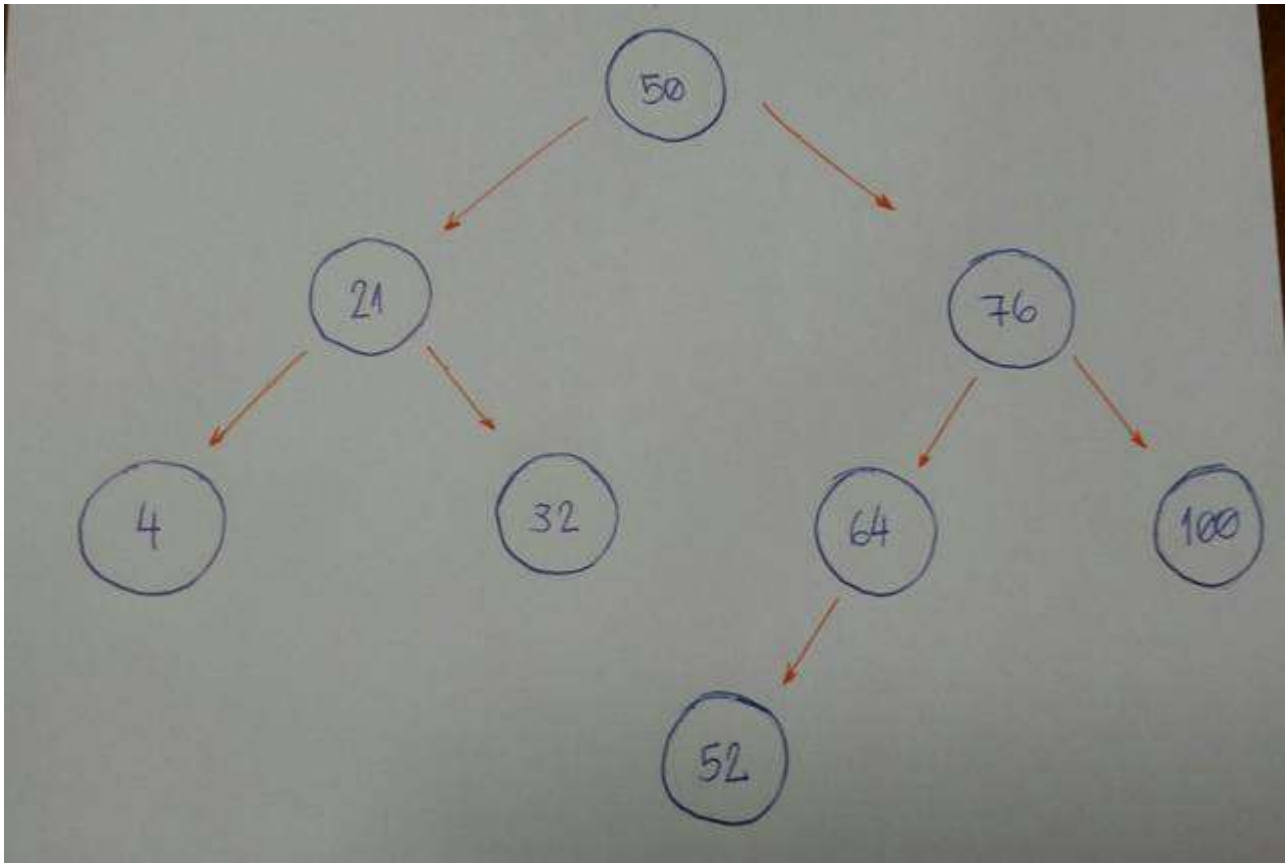
50、76、21、4、32、100、64、52。

首先我们需要知道的是，50是不是这棵树的根节点。



现在我们开始一个一个的插入节点。

- 76比50大，所以76插入在右边。
- 21比50小，所以21插入在左边。
- 4比50小。但是50已经有了值为21的左孩子。然后，4又比21小，所以将其插入在21的左边。
- 32比50小。但是50已经有了值为21的左孩子。然后，32又比21大，所以将其插入在21的右边。
- 100比50大。但是50已经有了一个值为76的右孩子。然后，100又比76大，所以将其插入在76的右边。
- 64比50大。但是50已经有了一个值为76的右孩子。然后，64又比76小，所以将其插入在76的左边。
- 52比50大。但是50已经有了一个值为76的右孩子。52又比76小，但是76也有一个值为64左孩子。52又比64小，所以将52插入在64的左边。



你注意到这里的模式了吗?

让我们把它分解。

1. 新节点值大于当前节点还是小于当前节点?
2. 如果新节点的值大于当前节点，则转到右子树。如果当前节点没有右孩子，则在那里插入新节点，否则返回步骤1。
3. 如果新节点的值小于当前节点，则转到左子树。如果当前节点没有左孩子，则在那里插入新节点，否则返回步骤1。
4. 这里我们没有处理特殊情况。当新节点的值等于节点的当前值时，使用规则3。考虑在子树的左侧插入相等的值。

现在我们来编码。

```
class BinarySearchTree:
    def __init__(self, value):
        self.value = value
        self.left_child = None
        self.right_child = None

    def insert_node(self, value):
        if value <= self.value and self.left_child:
            self.left_child.insert_node(value)
        elif value <= self.value:
```

```
            self.left_child = BinarySearchTree(value)
        elif value > self.value and self.right_child:
            self.right_child.insert_node(value)
        else:
            self.right_child = BinarySearchTree(value)
```

看起来很简单。

该算法的强大之处是其递归部分，即第9行和第13行。这两行代码均调用 insert_node 方法，并分别为其左孩子和右孩子使用它。第11行和第15行则在子节点处插入新节点。

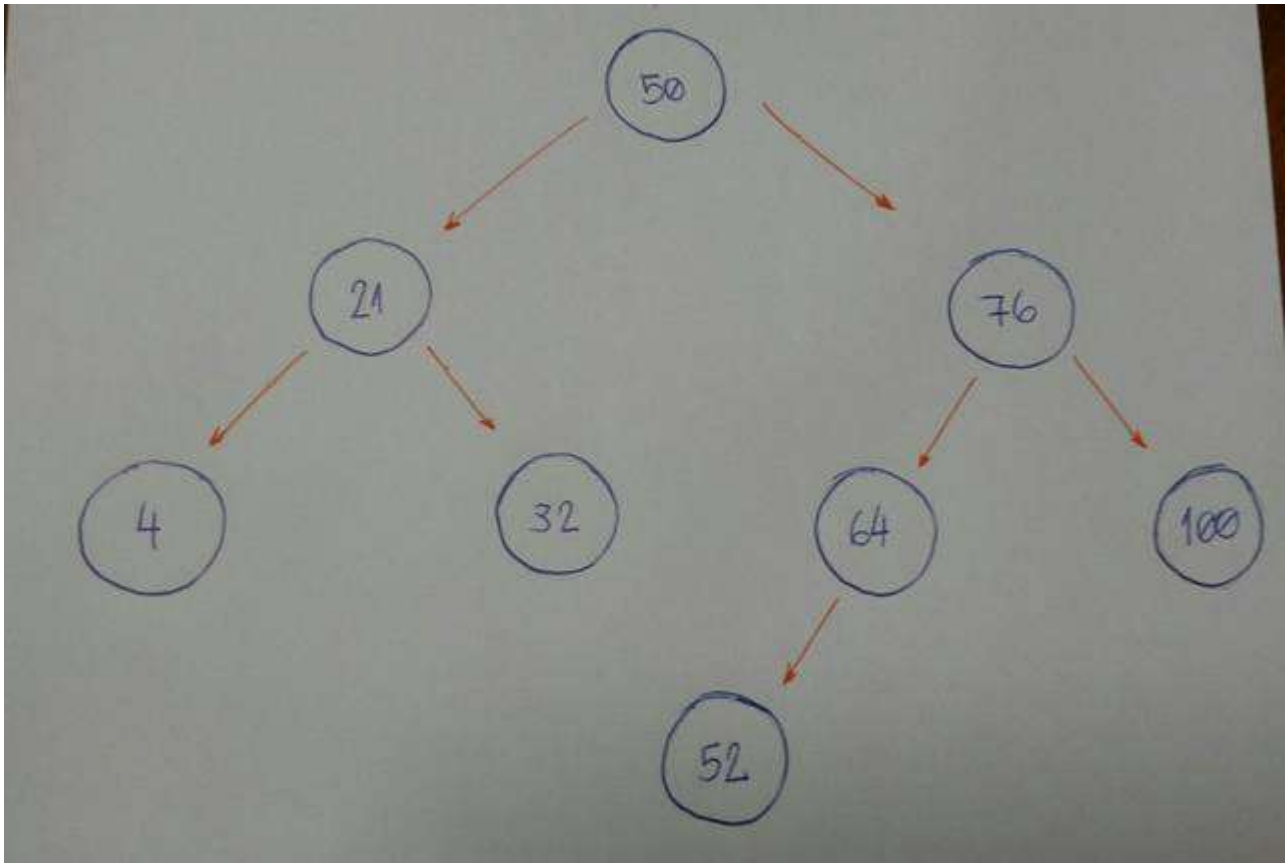
我们来搜索节点值

我们现在要构建的算法是关于搜索的。对于给定的值（整数），我们会搜索出我们的二叉查找树有或者没有这个值。

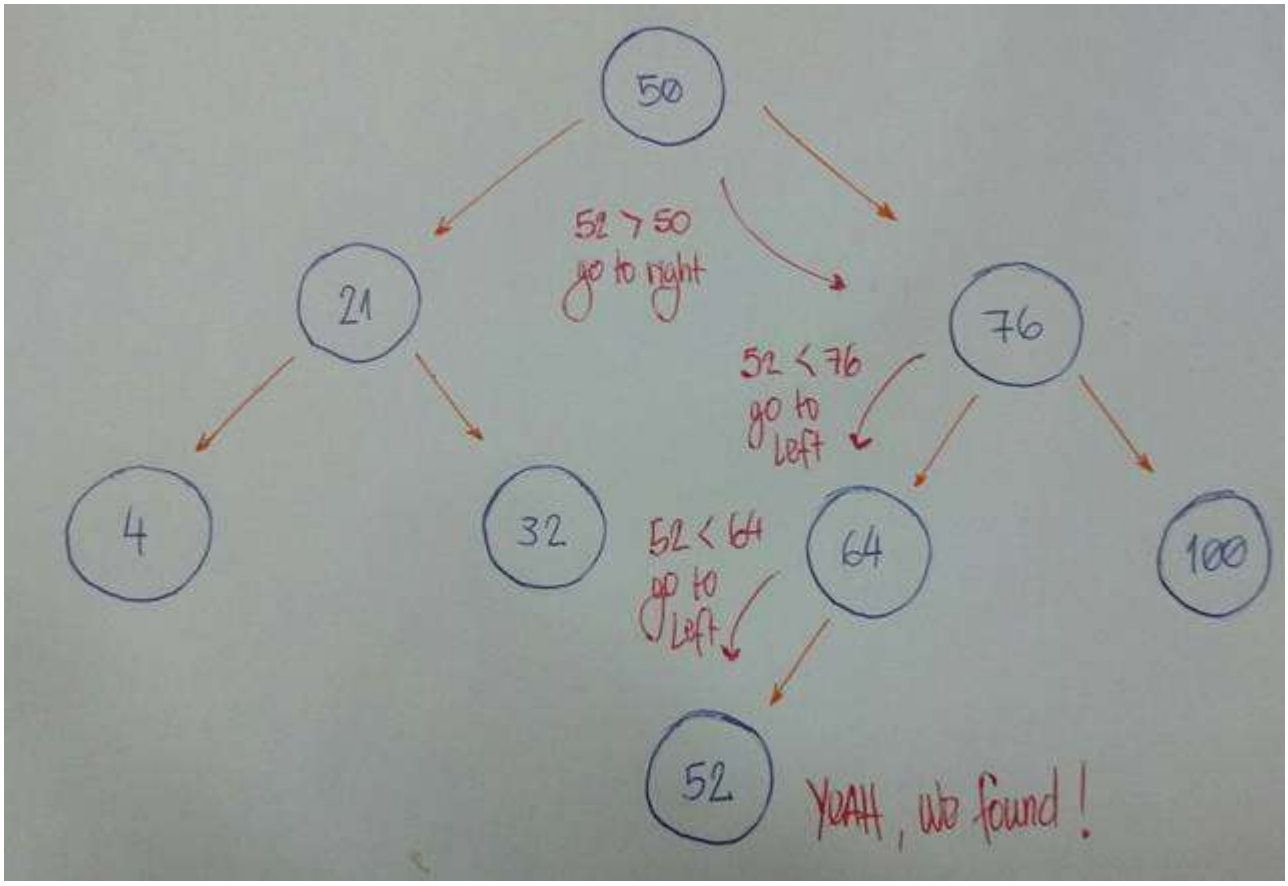
需要注意的一个重要事项是我们如何定义树的插入算法。 首先我们有根节点。所有左子树的节点值都比根节点小。所有右子树的节点值都比根节点大。

让我们看一个例子。

假设我们有这棵树。



现在我们想知道是否有一个节点的值为52。



让我们把它分解。

1. 我们以根节点作为当前节点开始。给定值小于当前节点值吗？如果是，那么我将在左子

树上查找它。

2. 给定值大于当前节点值吗？如果是，那么我们将在右子树上查找它。
3. 如果规则 #1 和 #2 均为假，我们可以比较当前节点值和给定值是否相等。如果返回真，那么我们可以说：“是的，我们的树拥有给定的值。” 否则，我们说：“不，我们的树没有给定的值。”

代码如下：

```
class BinarySearchTree:
    def __init__(self, value):
        self.value = value
        self.left_child = None
        self.right_child = None

    def find_node(self, value):
        if value < self.value and self.left_child:
            return self.left_child.find_node(value)
        if value > self.value and self.right_child:
            return self.right_child.find_node(value)

        return value == self.value
```

代码分析：

- 第8行和第9行归于规则#1。
- 第10行和第11行归于规则#2。
- 第13行归于规则#3。

我们如何测试它?

让我们通过初始化值为15的根节点创建二叉查找树。

```
bst = BinarySearchTree(15)
```

现在我们将插入许多新节点。

```
bst.insert_node(10)
bst.insert_node(8)
bst.insert_node(12)
bst.insert_node(20)
bst.insert_node(17)
bst.insert_node(25)
bst.insert_node(19)
```

对于每个插入的节点，我们测试是否 find_node 方法真地管用。

```
print(bst.find_node(15)) # True
print(bst.find_node(10)) # True
print(bst.find_node(8)) # True
print(bst.find_node(12)) # True
print(bst.find_node(20)) # True
print(bst.find_node(17)) # True
print(bst.find_node(25)) # True
print(bst.find_node(19)) # True
```

是的，它对这些给定值管用！让我们测试一个二叉查找树中不存在的值。

```
print(bst.find_node(0)) # False
```

查找完毕

删除：移除和组织

删除是一个更复杂的算法，因为我们需要处理不同的情况。对于给定值，我们需要删除具有此值的节点。想象一下这个节点的以下场景：它没有孩子，有一个孩子，或者有两个孩子。

- 场景 #1： 一个没有孩子的节点（叶节点）。

```
#         |50|                              |50|
#       /     \                           /    \
#     |30|     |70|   (DELETE 20) --->   |30|   |70|
#    /    \                                \
# |20|    |40|                             |40|
```

如果要删除的节点没有子节点，我们简单地删除它。该算法不需要重组树。

- 场景 #2： 仅有一个孩子（左或右孩子）的节点。

```
#         |50|                              |50|
#       /     \                           /    \
#     |30|     |70|   (DELETE 30) --->   |20|   |70|
#    /
# |20|
```

在这种情况下，我们的算法需要使节点的父节点指向子节点。如果节点是左孩子，则使其父节点指向其子节点。如果节点是右孩子，则使其父节点指向其子节点。

- 场景 #3： 有两个孩子的节点。

```
#         |50|                              |50|
```

```
#       /       \                           /    \
#     |30|      |70|   (DELETE 30) --->    |40|   |70|
#    /    \                                /
# |20|    |40|                          |20|
```

当节点有两个孩子，则需要从该节点的右孩子开始，找到具有最小值的节点。我们将把具有最小值的这个节点置于被删除的节点的位置。

是时候编码了。

```
def remove_node(self, value, parent):
    if value < self.value and self.left_child:
        return self.left_child.remove_node(value, self)
    elif value < self.value:
        return False
    elif value > self.value and self.right_child:
        return self.right_child.remove_node(value, self)
    elif value > self.value:
        return False
    else:
        if self.left_child is None and self.right_child is None and self == parent.left_child:
            parent.left_child = None
            self.clear_node()
        elif self.left_child is None and self.right_child is None and self == parent.right_child:
            parent.right_child = None
            self.clear_node()
        elif self.left_child and self.right_child is None and self == parent.left_child:
            parent.left_child = self.left_child
            self.clear_node()
        elif self.left_child and self.right_child is None and self == parent.right_child:
            parent.right_child = self.left_child
            self.clear_node()
        elif self.right_child and self.left_child is None and self == parent.left_child:
            parent.left_child = self.right_child
            self.clear_node()
        elif self.right_child and self.left_child is None and self == parent.right_child:
            parent.right_child = self.right_child
            self.clear_node()
        else:
            self.value = self.right_child.find_minimum_value()
            self.right_child.remove_node(self.value, self)

        return True
```

1. 首先: 注意下参数 value 和 parent 。我们想找到值等于该 value 的 node ，并且该 node 的父节点对于删除该 node 是至关重要的。
2. 其次: 注意下返回值。我们的算法将会返回一个布尔值。我们的算法在找到并删除该节点时返回 true 。否则返回 false 。
3. 第2行到第9行：我们开始查找等于我们要查找的给定的 value 的 node 。如果该 value 小于 current node 值，我们进入左子树，递归处理（当且仅当，current node 有左孩子）。如果该值大于，则进入右子树。递归处理。
4. 第10行: 我们开始考虑删除算法。
5. 第11行到第13行: 我们处理了没有孩子、并且是父节点的左孩子的节点。我们通过设置父节点的左孩子为空来删除该节点。

6. 第14行和第15行: 我们处理了没有孩子、并且是父节点的右孩子的节点。我们通过设置父节点的右孩子为空来删除该节点。
7. 清除节点的方法：我将会在后续文章中给出 clear_node 的代码。该函数将节点的左孩子、右孩子和值都设置为空。
8. 第16行到第18行: 我们处理了只有一个孩子（左孩子）、并且它是其父节点的左孩子的节点。我们将父节点的左孩子设置为当前节点的左孩子（这是它唯一拥有的孩子）。
9. 第19行到第21行: 我们处理了只有一个孩子（左孩子）、并且它是其父节点的右孩子的节点。我们将父节点的右孩子设置为当前节点的左孩子（这是它唯一拥有的孩子）。
10. 第22行到第24行: 我们处理了只有一个孩子（右孩子），并且是其父节点的左孩子的节点。我们将父节点的左孩子设置为当前节点的右孩子（这是它唯一的孩子）。
11. 第25行到第27行: 我们处理了只有一个孩子（右孩子），并且它是其父节点的右孩子的节点。我们将父节点的右孩子设置为当前节点的右孩子（这是它唯一的孩子）。
12. 第28行到第30行: 我们处理了同时拥有左孩子和右孩子的节点。我们获取拥有最小值的节点（代码随后给出），并将其值设置给当前节点。通过删除最小的节点完成节点移除。
13. 第32行: 如果我们找到了要查找的节点，就需要返回 true 。从第11行到第31行，我们处理了这些情况。所以直接返回 true ， 这就够了。

- clear_node 方法：设置节点的三个属性为空——(value, left_child, and right_child)

```
def clear_node(self):
    self.value = None
    self.left_child = None
    self.right_child = None
```

- find_minimum_value方法：一路向下找最左侧的。如果我们无法找到任何节点，我们找其中最小的。

```
def find_minimum_value(self):
    if self.left_child:
        return self.left_child.find_minimum_value()
    else:
        return self.value
```

让我们测试一下。

我们将使用这个树来测试我们的 remove_node 算法。

```
#         |15|
#       /    \
#     |10|    |20|
#    /   \    /   \
# |8|   |12| |17| |25|
#             \
#             |19|
```

删除值为 8 的节点。它是一个没有孩子的节点。

```
print(bst.remove_node(8, None)) # True
bst.pre_order_traversal()
```

```
#     |15|
#   /      \
# |10|     |20|
#    \    /    \
#   |12| |17| |25|
#          \
#          |19|
```

删除值为 17 的节点。它是一个只有一个孩子的节点。

```
print(bst.remove_node(17, None)) # True
bst.pre_order_traversal()
```

```
#        |15|
#      /      \
#    |10|     |20|
#       \    /    \
#      |12| |19| |25|
```

最后，删除一个拥有两个孩子的节点。它就是我们的树的根节点。

```
print(bst.remove_node(15, None)) # True
bst.pre_order_traversal()
```

```
#        |19|
#      /      \
#    |10|     |20|
#       \          \
#        |12|     |25|
```

测试完成。