在 Python 系列中构建森林：单线程二叉搜索树

顺煌

2021 年 7 月 11 日 警察

使用 Python 构建线程化二叉搜索树

这是构建森林系列文章中的第三篇文章。它将构建二叉搜索树的一种变体——线程二叉搜索树。

下载源 - 249.4 KB

下载最新源 - 249.4 KB

介绍

本文是在 Python 中构建森林系列的第三篇。在上一篇二叉树遍历中，我们讨论了使用递归方法和辅助堆栈的二叉树遍历。本文将构建二叉搜索树的一种变体——线程二叉搜索树。线程化二叉搜索树利用空的左或右节点的属性来实现某些遍历，而无需使用堆栈或递归方法。

项目设置

与构建二叉搜索树中的其他文章一样，实现假定 Python 3.9 或更新版本。本文增加了两个模块，以我们的项目：*single_threaded_binary_trees.py*为线索二叉树查找树实现和*test_single_threaded_binary_trees.py*它的单元测试。添加这两个文件后，我们的项目布局就变成了这样：

复制代码

```
forest-python
├── forest
│   ├── __init__.py
│   ├── binary_trees
│   │   ├── __init__.py
│   │   ├── binary_search_tree.py
│   │   ├── single_threaded_binary_trees.py
│   │   └── traversal.py
│   └── tree_exceptions.py
└── tests
    ├── __init__.py
    ├── conftest.py
    ├── test_binary_search_tree.py
    ├── test_single_threaded_binary_trees.py
    └── test_traversal.py
```

（完整的代码可在Forest-python 中找到。）

什么是线程二叉树?

线程化二叉树是一种二叉树变体，它通过利用节点的空左或右属性以特定顺序优化遍历。有两种类型的线程二叉树：

- **单线程二叉树**：对于节点的任何空的左属性或右属性，空属性都指向有序的前驱或后继。换句话说，不是让 left 或 right 属性为空，而是 left 或 right 属性指向节点的前驱或后继。单线程二叉树有两种实现方式：左单线程二叉树和右单线程二叉树。
  - **右单线程二叉树**：节点的空右属性指向该节点的**后继**，如果一个节点的左属性为空，则该属性保持为空。
  - **左单线程二叉树**：一个节点的空左属性指向该节点的**前驱**，如果一个节点的右属性为空，则该属性保持为空。
- **双线程二叉树**：对于任何空的左属性或右属性，空属性都指向有序的前驱或后继：如果左属性为空，则左属性指向节点的前驱；如果右属性为空，则右属性指向节点的后继。

尽管可以将线程添加到任何二叉树，但将线程添加到二叉搜索树或其变体，即满足二叉搜索树属性的树是最有益的。因此，在本项目中，我们将实现的线程二叉树是带有线程的二叉搜索树（threaded binary search tree）。在文章的其余部分，我们所指的所有线程二叉树也是二叉搜索树，以避免冗长的阶段。

此外，线程不需要指向其有序的前驱或后继。它也可以是预购或后购。然而， in-order 是二叉搜索树中的排序顺序，因此指向其有序前驱或后继的线程是最常见的。

为什么我们需要线程?

向二叉树添加线程会增加复杂性，那么为什么我们需要线程呢？有几个原因：

- 快速后继或前驱访问
- 某些遍历没有辅助堆栈或递归方法
- 由于不需要辅助堆栈或递归，因此在执行遍历时减少了内存消耗
- 利用浪费的空间。由于节点的空 left 或 right 属性不存储任何内容，我们可以将它们用作线程

对于每种类型的线程二叉树，我们可以总结如下添加线程的好处：

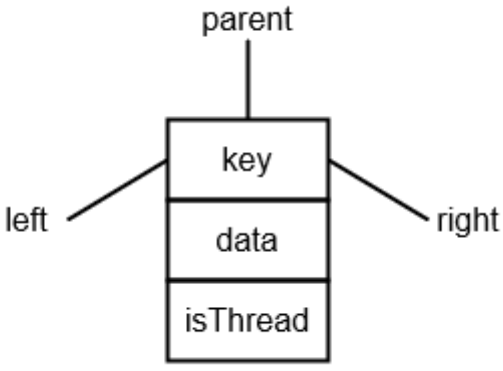
- 正确的线程二叉树可以在没有堆栈或递归方法的情况下执行中序和预序遍历。它还具有快速的后继访问。
- 左线程二叉树可以在没有堆栈或递归方法的情况下执行逆序遍历，并且具有快速的前驱访问。
- 双线程二叉树兼有两种单线程二叉树的优点。

构建单线程二叉搜索树

正如我们在构建二叉搜索树部分所做的那样，本节将介绍实现并讨论实现选择背后的一些想法。

节点

节点结构类似于二叉搜索树节点，除了它有一个额外的字段，`is_thread`。`is_thread`属性的目的是知道左属性还是右属性是线程。



该`is_thread`属性是一个布尔变量：`True`如果该属性（左或右取决于单线程二叉树的类型）是一个线程；`False`除此以外。

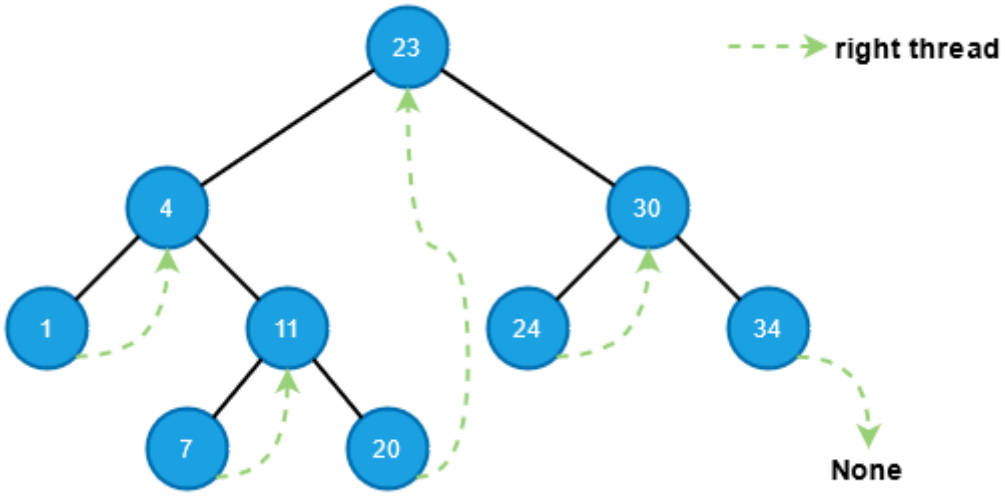
Python 复制代码

```python
@dataclasses.dataclass
class Node:
    key: Any
    data: Any
    left: Optional["Node"] = None
    right: Optional["Node"] = None
    parent: Optional["Node"] = None
    is_thread: bool = False
```

作为二叉搜索树节点，我们将线程二叉树的节点类定义为dataclass。

右单线程二叉搜索树

顾名思义，我们可以将正确的单线程二叉树形象化如下图：



每个节点的空右属性指向它的有序后继，并且它的`is_thread`变量设置为`True`除了最右边的节点。最右边节点的 right 属性保持为空，它`is_thread`是`False`，所以我们知道它是最右边的节点（即没有更多的后继节点）。

与二叉搜索树一样，右线程二叉树有核心功能（`insert`，`delete`和`search`）来构建和修改以及其他不绑定特定树的辅助功能，例如获取最左边的节点和获取树的高度。`__repr__()`我们在二叉搜索树中实现的相同功能也可用于调试目的。

这里的主要区别在于我们实现了不使用堆栈或使用递归方法的中序和预序遍历。

Python 缩小▲ 复制代码

```python
class RightThreadedBinaryTree:

    def __init__(self) -> None:
        self.root: Optional[Node] = None

    def __repr__(self) -> str:
        """Provie the tree representation to visualize its layout."""
        if self.root:
            return (
                f"{type(self)}, root={self.root}, "
                f"tree_height={str(self.get_height(self.root))}"
            )
        return "empty tree"

    def search(self, key: Any) -> Optional[Node]:
        …

    def insert(self, key: Any, data: Any) -> None:
        …
```

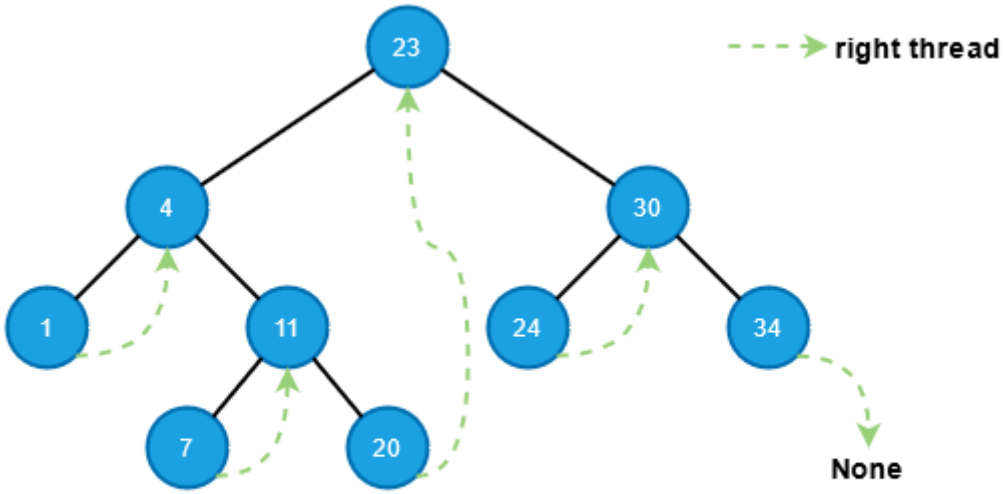
```
    def delete(self, key: Any) -> None:
        …

    @staticmethod
    def get_leftmost(node: Node) -> Node:
        …

    @staticmethod
    def get_rightmost(node: Node) -> Node:
        …

    @staticmethod
    def get_successor(node: Node) -> Optional[Node]:
        …

    @staticmethod
    def get_predecessor(node: Node) -> Optional[Node]:
        …

    @staticmethod
    def get_height(node: Optional[Node]) -> int:
        …

    def inorder_traverse(self) -> traversal.Pairs:
        …

    def preorder_traverse(self) -> traversal.Pairs:
        …
```

插入

该`insert`操作类似于二叉搜索树的插入。不同的是线程化二叉树需要考虑线程更新。因此，插入步骤变为如下：

1. 通过从根遍历树并将新节点的关键字与沿途每个节点的关键字进行比较，找到合适的位置（即新节点的父节点）以插入新节点。当走到右子树时，也要检查`is_thread`变量。如果变量是`True`，我们就到达了叶子层级，叶子节点就是新节点的父节点。
2. 更新新节点的父属性以指向父节点。
3. 如果新节点是父节点的左子节点，则将新节点的右属性设置为父节点，并将`is_thread`变量设置为`True`。更新父节点的左属性以指向新节点。
4. 如果新节点是其父节点的右子节点，则将父节点的right属性复制到新节点的right属性（父节点的right属性必须是插入前的线程），并将`is_thread`变量设置为`True`。更新父节点的 right 属性以指向新节点并将父节点的属性设置`is_thread`为`False`。

下图展示了节点插入的步骤。



我们可以如下实现插入：

Python 缩小▲ 复制代码

```python
def insert(self, key: Any, data: Any) -> None:
    new_node = Node(key=key, data=data)
    parent: Optional[Node] = None
    current: Optional[Node] = self.root

    while current:
        parent = current
        if new_node.key < current.key:
            current = current.left
        elif new_node.key > current.key:
            # If the node is thread, meaning it's a leaf node.
            if current.is_thread:
                current = None
            else:
                current = current.right
        else:
            raise tree_exceptions.DuplicateKeyError(key=new_node.key)
    new_node.parent = parent
    # If the tree is empty
    if parent is None:
        self.root = new_node
    elif new_node.key < parent.key:
        parent.left = new_node

        # Update thread
        new_node.right = parent
        new_node.is_thread = True

    else:
        # Update thread
        new_node.is_thread = parent.is_thread
        new_node.right = parent.right
        parent.is_thread = False
        # Parent's right must be set after thread update
        parent.right = new_node
```

搜索

搜索操作也类似于二叉搜索树的搜索，但需要检查`is_thread`变量以确定是否到达叶级。

1. 从根开始遍历树并沿着树遍历将键与每个节点的键进行比较
2. 如果一个键匹配，我们就找到了节点。
3. 如果到达叶子后没有key匹配（如果`is_thread`是`True`，则表示该节点是叶子节点），则它不存在于树中。

该实现类似于我们在二叉搜索树中制作的二叉搜索树，只是做了一个简单的修改——检查`is_thread`。

Python 复制代码

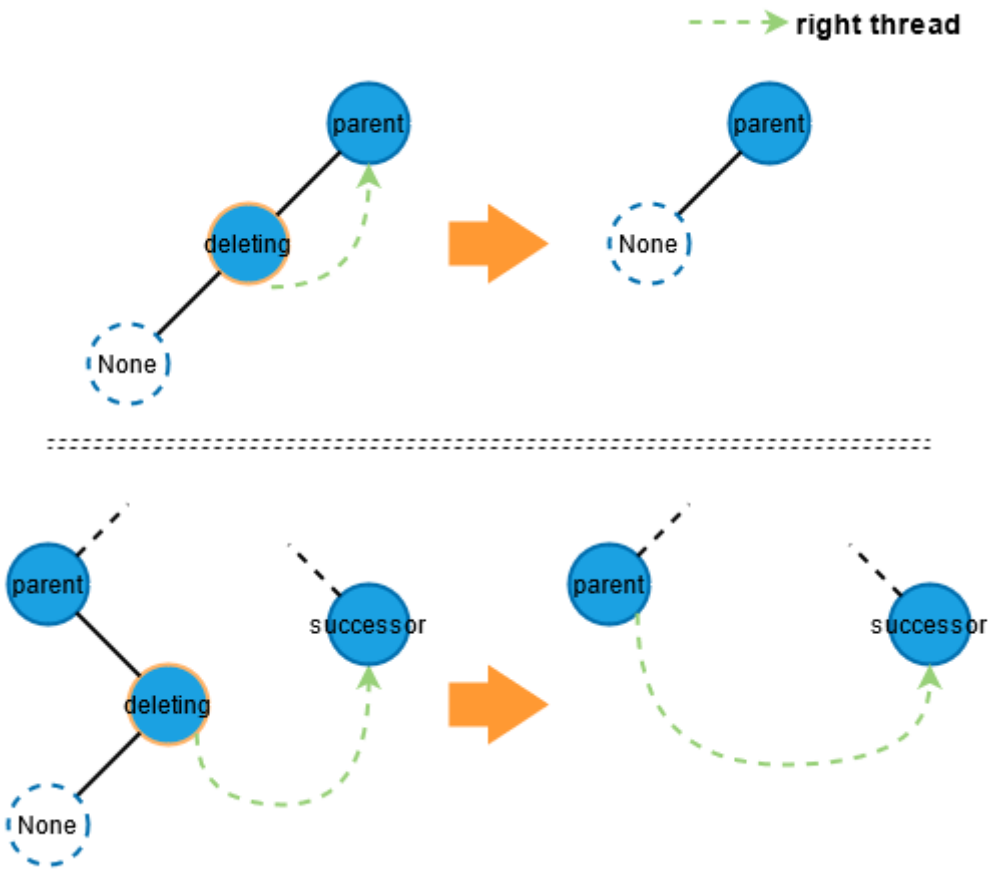
```
def search(self, key: Any) -> Optional[Node]:
    current = self.root
    while current:
        if key == current.key:
            return current
        elif key < current.key:
            current = current.left
        else:  # key > current.key
            if current.is_thread:
                break
            current = current.right
    return None
```

删除

和二叉搜索树一样，要删除的右线程树节点有三种情况：没有子节点、只有一个子节点、两个子节点。我们还使用我们在二叉搜索树中所做的移植技术：删除将子树替换为要删除的节点。虽然基本思想是一样的，但是`transplant`函数和`delete`函数都需要把正确的线程带入一个计数中。我们需要记住的最重要的一点是，当我们删除一个节点时，如果有另一个节点的右属性指向要删除的节点，我们需要更新该节点的线程（即右属性）。

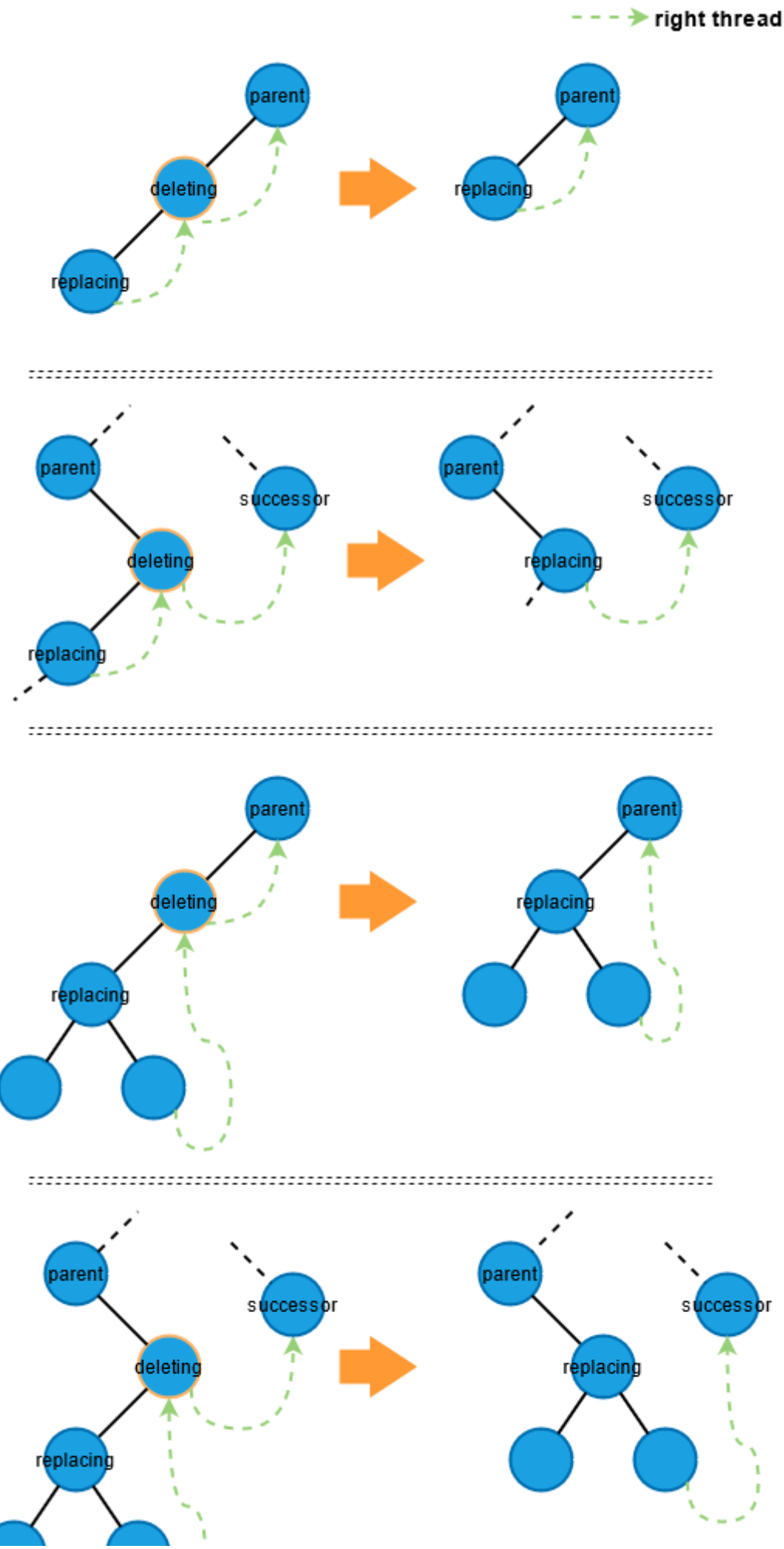
案例 1：没有孩子

如果要删除的节点没有子节点，则其左属性为空，`is_thread`为`True`。关于线程，我们需要考虑两种情况。见下图：



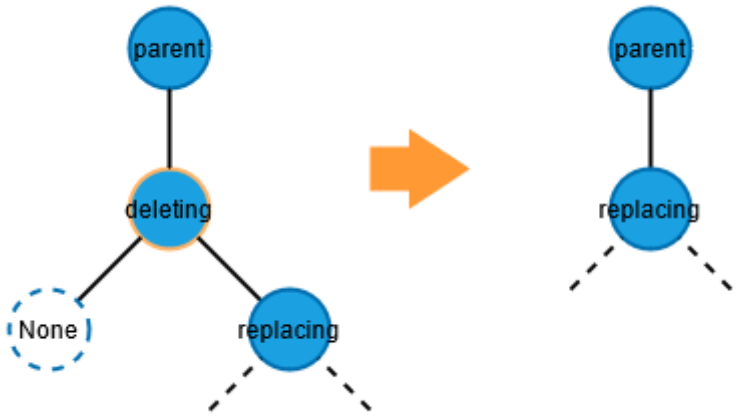
案例 2a：只有一个左孩子

如果要删除的节点只有一个左孩子，则表示该节点的`is_thread`是`True`。我们需要更新线程的情况如下图：



情况 2b：只有一个合适的孩子

如果要删除的节点只有一个右孩子，则表示该节点的左属性为空，`is_thread`为`False`。由于要删除的节点没有左子节点，也就意味着没有人指向该节点。

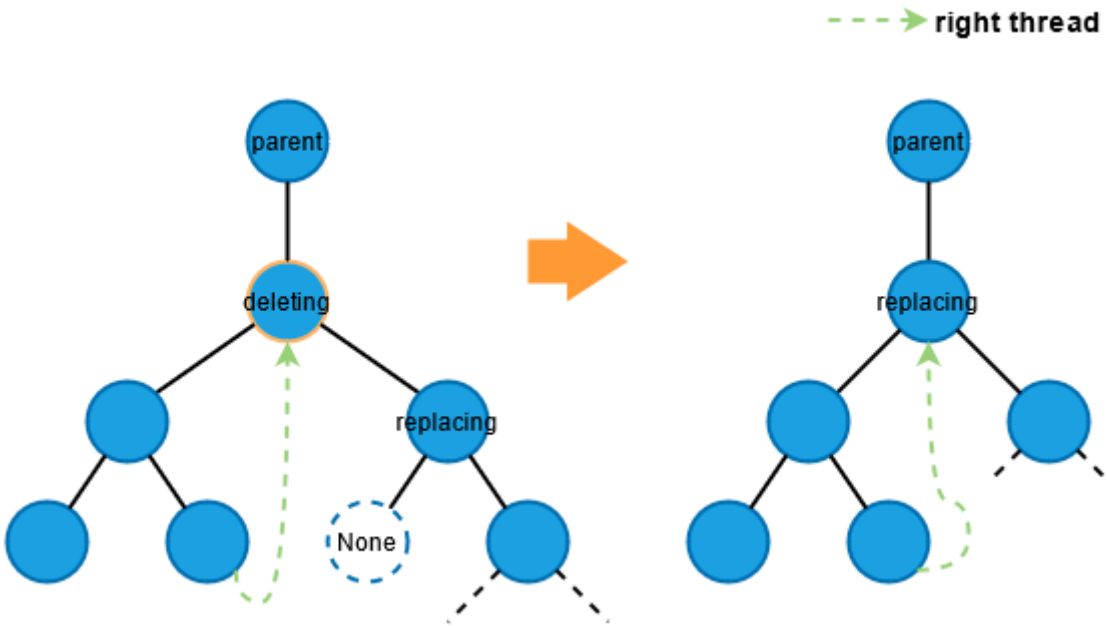


案例 3：两个孩子

与二叉搜索树删除类似，待删除节点有两个子节点的情况可以分解为两个子情况：

3.a 删除节点的右子节点也是右子树中最左边的节点。

在这种情况下，右孩子必须只有一个右孩子。因此，我们可以用它的右孩子替换删除节点，如下图所示：

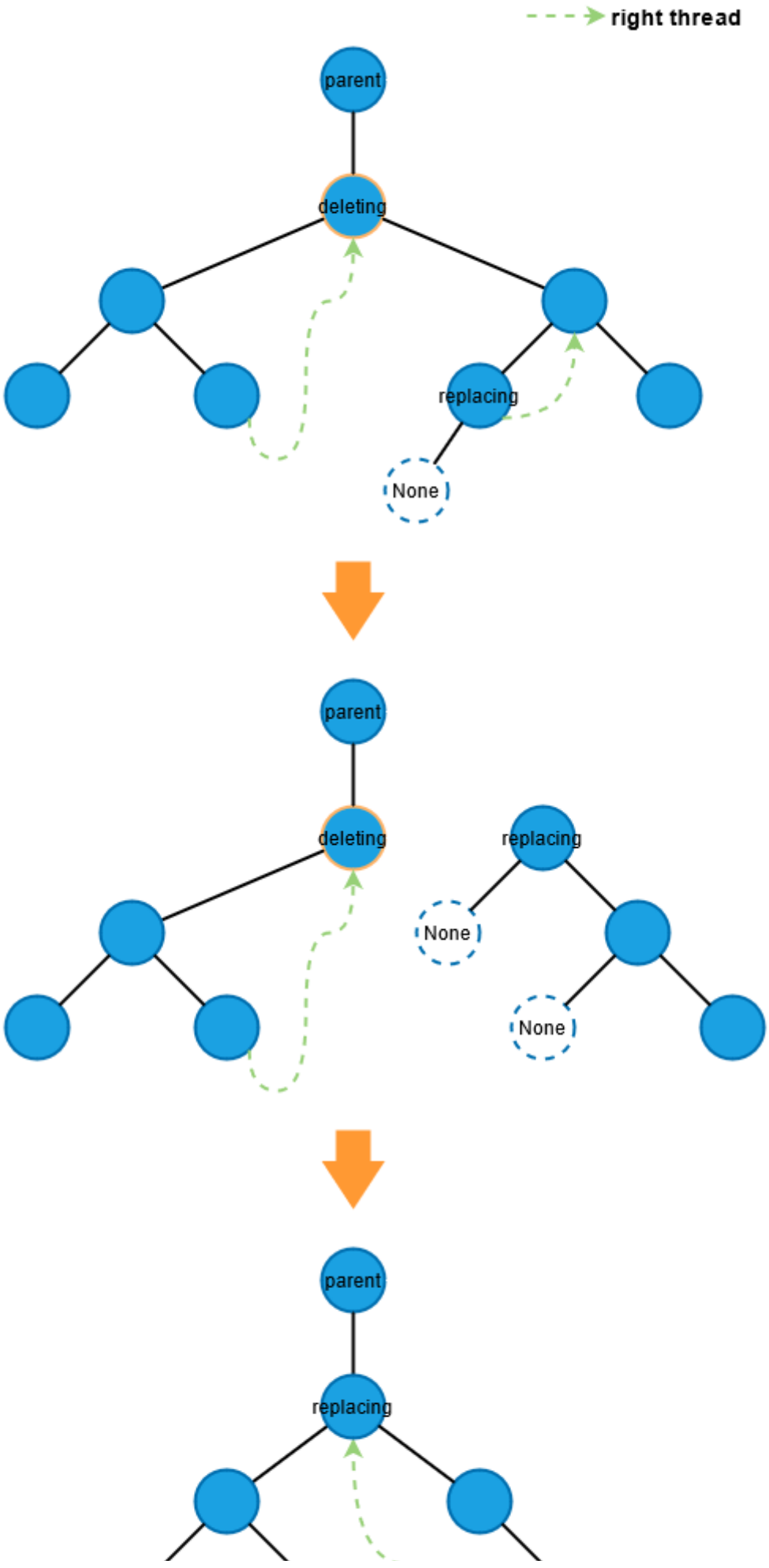


3.b. 删除节点的右孩子也有两个孩子。

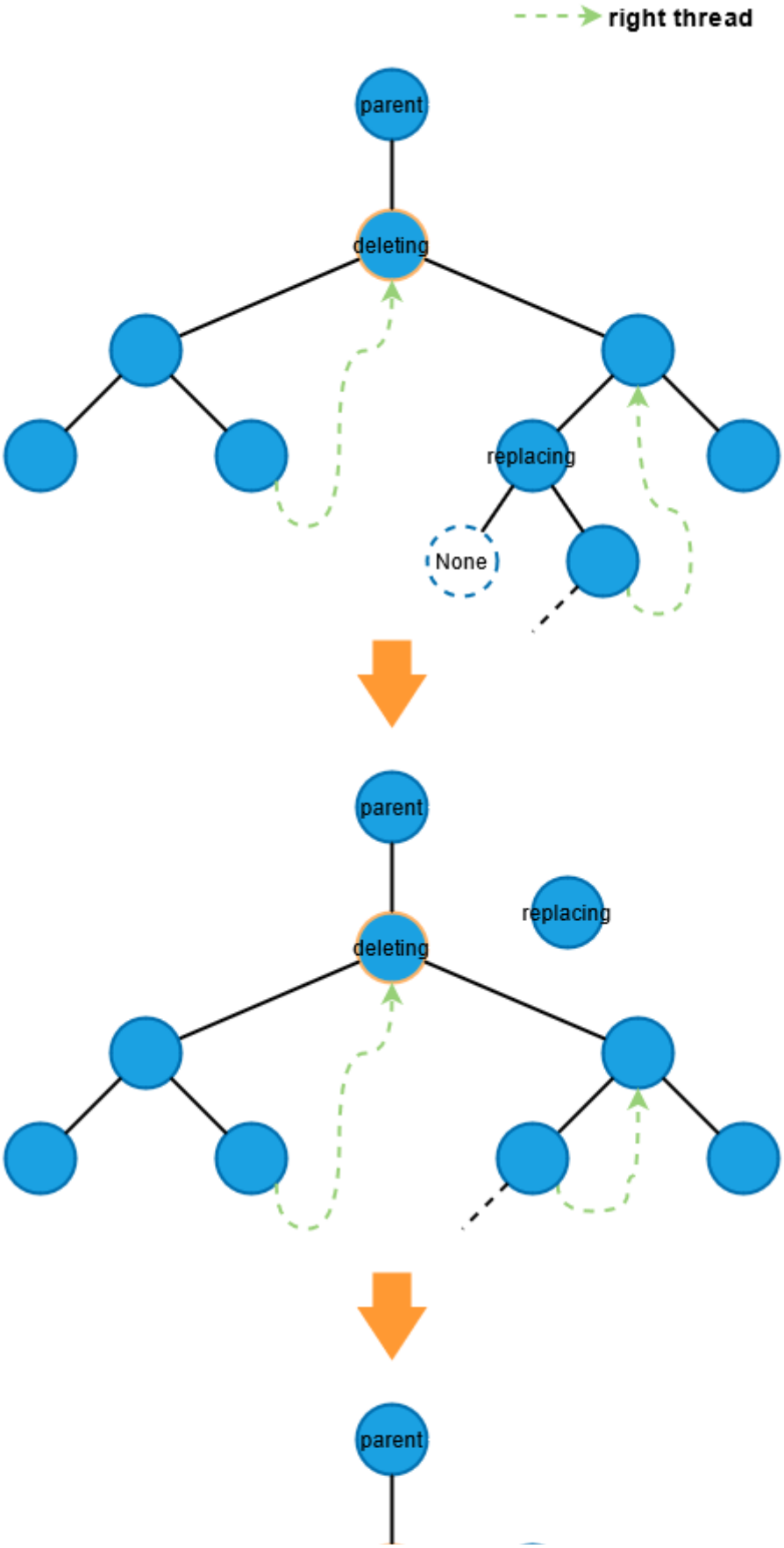
在这种情况下，我们从右子树中找到最左边的节点来替换要删除的节点。请注意，当我们从右子树中取出最左边的节点时，它也属于删除情况：情况 1：没有孩子或情况 2：只有一个右孩子。否则，它不能是最左边的节点。

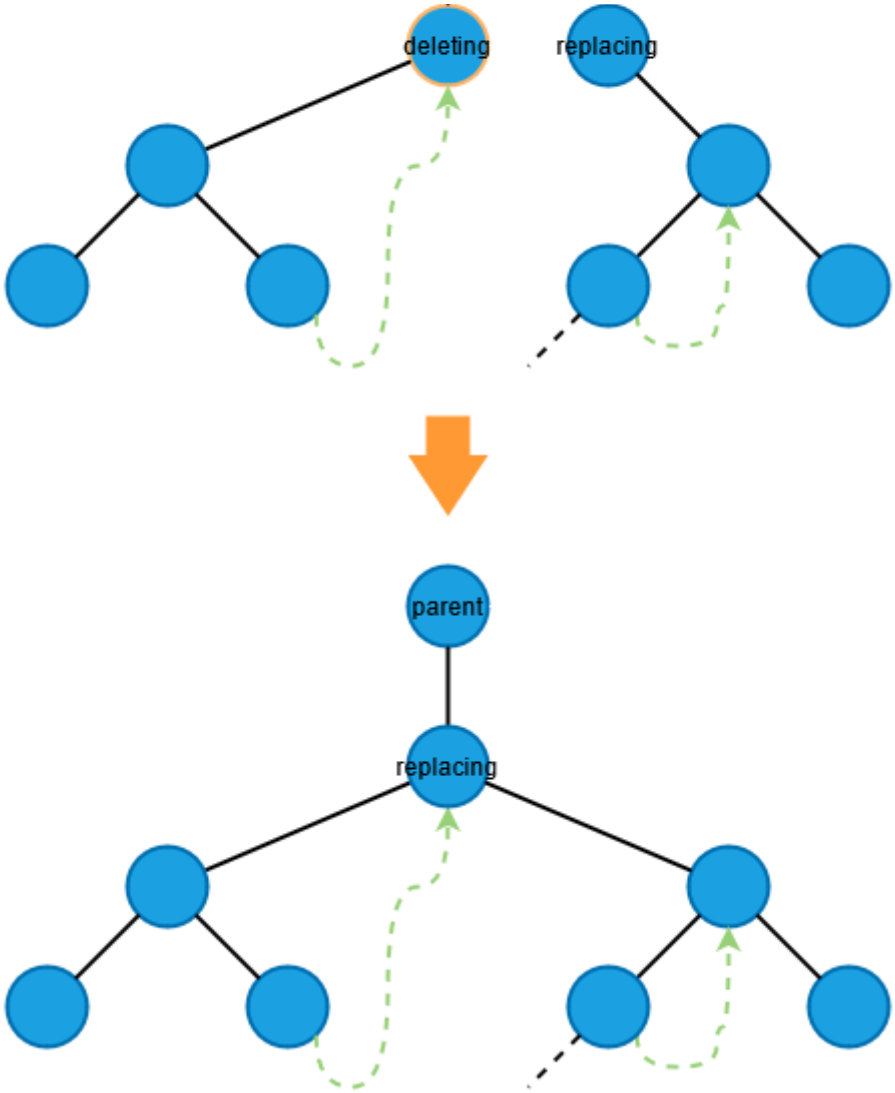
因此，我们使用该`transplant`函数两次：一次取出最左边的节点，另一次将删除的节点替换为原来的最左边的节点。下图展示了我们执行删除时线程的考虑。

替换节点没有子节点：


right thread
parent
deleting
replacing
None
parent
deleting
replacing
None
None
parent
replacing

替换节点只有一个右孩子:





基于上面的图片，我们可以实现delete和transplant功能如下：

Python 缩小▲ 复制代码

```
def delete(self, key: Any) -> None:
    if self.root and (deleting_node := self.search(key=key)):
        # Case 1: no child
        if deleting_node.left is None and (
            deleting_node.right is None or deleting_node.is_thread
        ):
            self._transplant(deleting_node=deleting_node, replacing_node=None)

        # Case 2a: only one right child
        elif deleting_node.left is None and deleting_node.is_thread is False:
            self._transplant(
                deleting_node=deleting_node, replacing_node=deleting_node.right
            )

        # Case 2b: only one left left child
        elif deleting_node.left and deleting_node.is_thread:
            predecessor = self.get_predecessor(node=deleting_node)
            if predecessor:
                predecessor.right = deleting_node.right
            self._transplant(
                deleting_node=deleting_node, replacing_node=deleting_node.left
            )

        # Case 3: two children
        elif (
            deleting_node.left
            and deleting_node.right
```

```
            and deleting_node.is_thread is False
        ):
            predecessor = self.get_predecessor(node=deleting_node)
            replacing_node: Node = self.get_leftmost(node=deleting_node.right)
            # the leftmost node is not the direct child of the deleting node
            if replacing_node.parent != deleting_node:
                if replacing_node.is_thread:
                    self._transplant(
                        deleting_node=replacing_node, replacing_node=None
                    )
                else:
                    self._transplant(
                        deleting_node=replacing_node,
                        replacing_node=replacing_node.right,
                    )
                replacing_node.right = deleting_node.right
                replacing_node.right.parent = replacing_node
                replacing_node.is_thread = False

            self._transplant(
                deleting_node=deleting_node, replacing_node=replacing_node
            )
            replacing_node.left = deleting_node.left
            replacing_node.left.parent = replacing_node
            if predecessor and predecessor.is_thread:
                predecessor.right = replacing_node
        else:
            raise RuntimeError("Invalid case. Should never happened")

def _transplant(self, deleting_node: Node, replacing_node: Optional[Node]) -> None:
    if deleting_node.parent is None:
        self.root = replacing_node
        if self.root:
            self.root.is_thread = False
    elif deleting_node == deleting_node.parent.left:
        deleting_node.parent.left = replacing_node
        if replacing_node:
            if deleting_node.is_thread:
                if replacing_node.is_thread:
                    replacing_node.right = replacing_node.right
    else:  # deleting_node == deleting_node.parent.right
        deleting_node.parent.right = replacing_node
        if replacing_node:
            if deleting_node.is_thread:
                if replacing_node.is_thread:
                    replacing_node.right = replacing_node.right
        else:
            deleting_node.parent.right = deleting_node.right
            deleting_node.parent.is_thread = True

    if replacing_node:
        replacing_node.parent = deleting_node.parent
```

获取高度

为了计算线程化二叉树的树高，我们可以像在二叉搜索树中所做的那样，为每个孩子的高度递归地将高度增加 1。如果一个节点有两个孩子，我们使用max函数从孩子那里得到更大的高度，并将最高的加一。主要区别在于我们`is_thread`用来检查节点是否有右孩子。

复制代码

```
@staticmethod
def get_height(node: Optional[Node]) -> int:
    if node:
        if node.left and node.is_thread is False:
            return (
                max(
```

```
                    RightThreadedBinaryTree.get_height(node.left),
                    RightThreadedBinaryTree.get_height(node.right),
                )
                + 1
            )

        if node.left:
            return RightThreadedBinaryTree.get_height(node=node.left) + 1

        if node.is_thread is False:
            return RightThreadedBinaryTree.get_height(node=node.right) + 1
    return 0
```

获取最左边和最右边的节点

获取最左边和最右边节点的实现类似于二叉搜索树。要得到最右边的节点，除了检查右边的属性是否为空，我们还需要检查是否`is_thread`为`True`。因此，我们可以`get_rightmost`像下面这样修改函数：

Python 复制代码

```
@staticmethod
def get_rightmost(node: Node) -> Node:
    current_node = node
     while current_node.is_thread is False and current_node.right:
        current_node = current_node.right
    return current_node
```

该`get_leftmost`实施是相同的`get_leftmost`二进制搜索树。

Python 复制代码

```
@staticmethod
def get_leftmost(node: Node) -> Node:
    current_node = node
    while current_node.left:
        current_node = current_node.left
    return current_node
```

前任和继任者

由于右线程树提供了快速的有序后继访问（如果一个节点的右属性是一个线程，则右属性指向节点的有序后继），如果正确的属性，我们可以通过跟随其右属性来获取节点的后继是一个线程。否则，该节点的后继是该节点右子树的最左边的节点。

Python 复制代码

```
@staticmethod
def get_successor(node: Node) -> Optional[Node]:
    if node.is_thread:
        return node.right
    else:
        if node.right:
            return RightThreadedBinaryTree.get_leftmost(node=node.right)
        # if node.right is None, it means no successor of the given node.
        return None
```

的实现`get_predecessor`与二分查找前置函数相同。

Python 复制代码

```
@staticmethod
def get_predecessor(node: Node) -> Optional[Node]:
    if node.left:
        return RightThreadedBinaryTree.get_rightmost(node=node.left)
    parent = node.parent
```

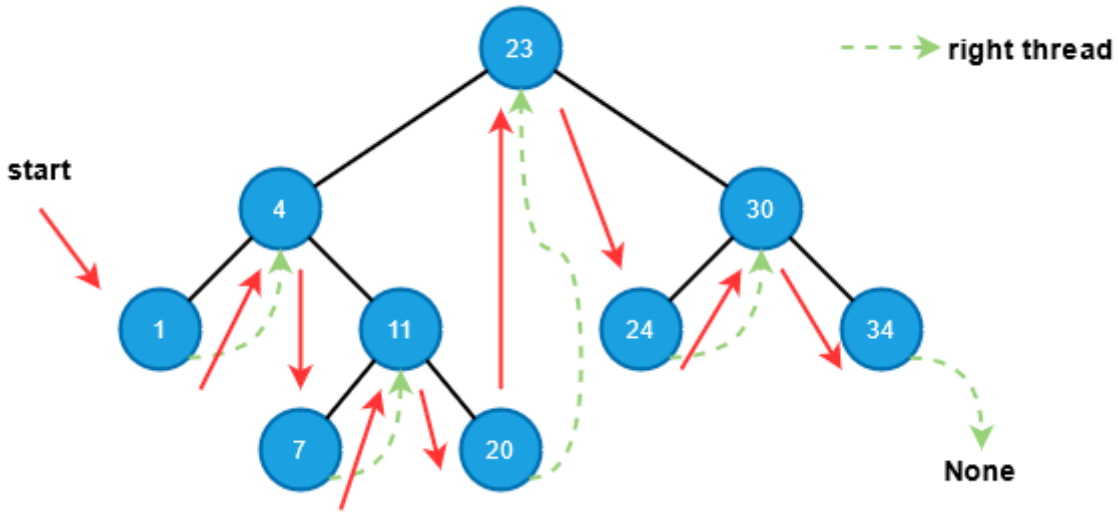
```
    while parent and node == parent.left:
        node = parent
        parent = parent.parent
    return parent
```

有序遍历

正确的线程树提供的一个好处是我们可以在不使用辅助或递归方法的情况下进行有序遍历。算法如下：

1. 从整个树的最左边的节点开始。
2. 如果正确的属性是线程，则遵循正确的属性；如果右侧属性不是线程，则转到子树的最左侧节点。
3. 重复步骤 2，直到正确的属性为None。

下图中的红色箭头展示了线程的有序遍历。



并在不使用辅助堆栈或递归的情况下实现该功能。

Python 复制代码

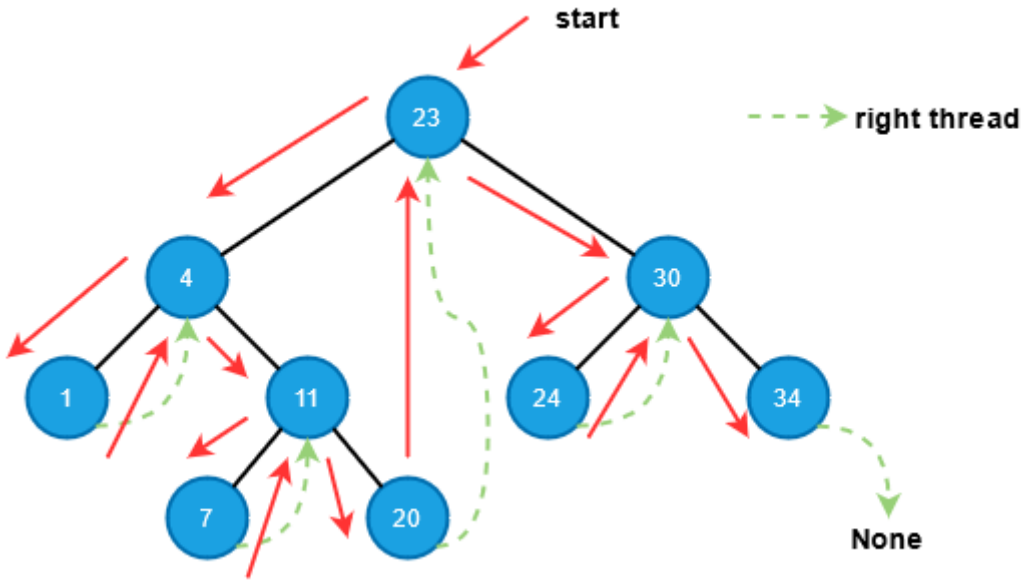
```
def inorder_traverse(self) -> traversal.Pairs:
    if self.root:
        current: Optional[Node] = self.get_leftmost(node=self.root)
        while current:
            yield (current.key, current.data)

            if current.is_thread:
                current = current.right
            else:
                if current.right is None:
                    break
                current = self.get_leftmost(current.right)
```

预购遍历

右线程树还提供了一种更简单的方法来进行预序遍历，并且比中序遍历更简单。

1. 从根开始。
2. 如果 left 属性不为空，则转到左孩子。
3. 如果 left 属性为空，则跟随线程向右。
4. 重复步骤 2 和 3，直到正确的属性为空。

下图中下面的红色箭头表示线程的方式有序遍历。



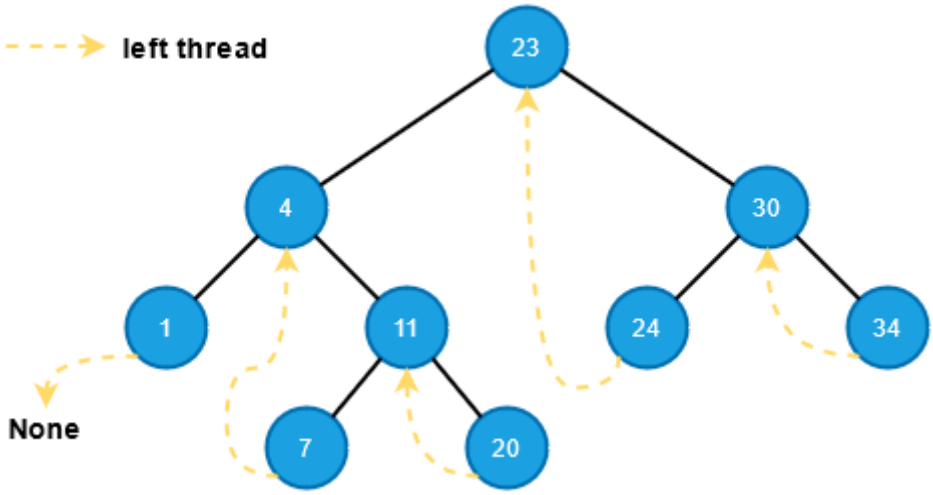
前序遍历可以实现如下：

Python 复制代码

```
def preorder_traverse(self) -> traversal.Pairs:
    current = self.root
    while current:
        yield (current.key, current.data)

        if current.is_thread:
            # If a node is thread, it must have a right child.
            current = current.right.right  # type: ignore
        else:
            current = current.left
```

左单线程二叉搜索树

左线程树与右线程树对称。如果左线程树中任何节点的左属性为空，则左属性是一个线程并指向节点的有序前驱，`is_thread`变量设置为`True`。左线程树可以在下图中可视化。



左线程树的类布局与右线程树几乎相同。唯一的区别是左线程树提供了简单的反向中序遍历，而不是中序和预序遍历。

Python 缩小▲ 复制代码

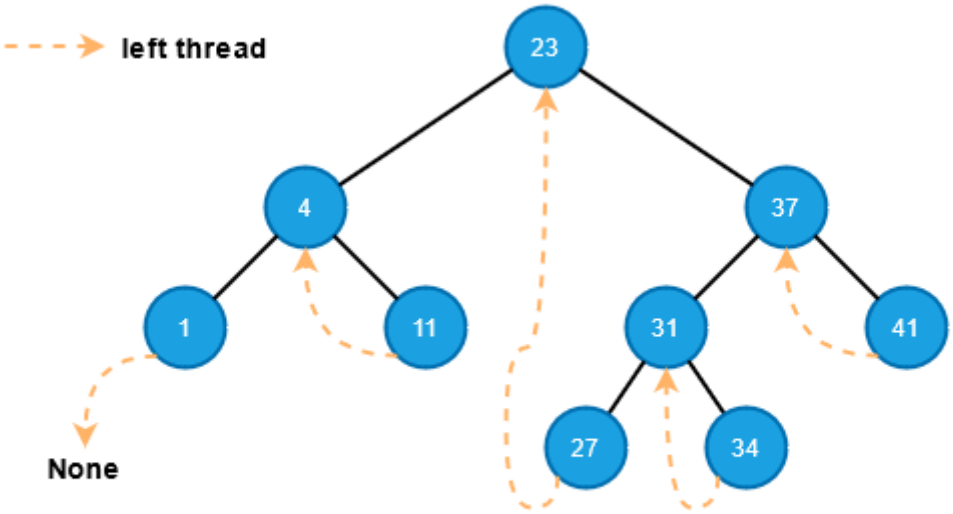
```
class LeftThreadedBinaryTree:

    def __init__(self) -> None:
        self.root: Optional[Node] = None

    def __repr__(self) -> str:
        """Provie the tree representation to visualize its layout."""
        if self.root:
            return (
                f"{type(self)}, root={self.root}, "
                f"tree_height={str(self.get_height(self.root))}"
            )
        return "empty tree"

    def search(self, key: Any) -> Optional[Node]:
        …

    def insert(self, key: Any, data: Any) -> None:
        …

    def delete(self, key: Any) -> None:
        …

    @staticmethod
    def get_leftmost(node: Node) -> Node:
        …

    @staticmethod
    def get_rightmost(node: Node) -> Node:
        …

    @staticmethod
    def get_successor(node: Node) -> Optional[Node]:
        …

    @staticmethod
    def get_predecessor(node: Node) -> Optional[Node]:
        …

    @staticmethod
    def get_height(node: Optional[Node]) -> int:
        …

    def reverse_inorder_traverse(self) -> traversal.Pairs:
        …
```

插入

该`insert`操作类似于右螺纹树。不同之处在于螺纹在左侧。

1. 通过从根遍历树并将新节点的关键字与沿途每个节点的关键字进行比较，找到合适的位置（即新节点的父节点）以插入新节点。当走到左子树时，也要检查`is_thread`变量。如果变量是`True`，我们就到达了叶子层级，叶子节点就是父节点。
2. 找到父节点后，更新父节点的左边（或右边取决于key）指向新节点。
3. 更新新节点的父属性以指向父节点。
4. 如果新节点是其父节点的右子节点，则将新节点的左属性指向父节点，并将`is_thread`变量设置为`True`。
5. 如果新节点是父节点的左子节点，则将父节点的左属性复制到新节点的左属性（父节点的左属性必须是插入前的线程）并设置`is_thread True`。更新父节点的左属性以指向新节点。

下图展示了节点插入的步骤。



实现如下：

Python 缩小▲ 复制代码

```
def insert(self, key: Any, data: Any) -> None:
    new_node = Node(key=key, data=data)
    parent: Optional[Node] = None
    current: Optional[Node] = self.root

    while current:
        parent = current
        if new_node.key < current.key:
            # If the node is thread, meaning it's a leaf node.
            if current.is_thread:
                current = None
            else:
                current = current.left
        elif new_node.key > current.key:
            current = current.right
        else:
            raise tree_exceptions.DuplicateKeyError(key=new_node.key)
    new_node.parent = parent
    # If the tree is empty
    if parent is None:
        self.root = new_node
    elif new_node.key > parent.key:
        parent.right = new_node
        # Update thread
        new_node.left = parent
        new_node.is_thread = True

    else:
        # Update thread
        new_node.is_thread = parent.is_thread
        new_node.left = parent.left
        parent.is_thread = False
        # Parent's left must be set after thread update
        parent.left = new_node
```

搜索

该`search`操作类似于右螺纹二叉树，所以我们需要检查`is_thread`变量来确定，如果我们到达叶。

1. 从根开始遍历树并沿着树遍历将键与每个节点的键进行比较
2. 如果一个键匹配，我们就找到了节点。
3. 如果到达叶子后没有key匹配（如果`is_thread`是`True`，也表示该节点是叶子节点），则它不存在于树中。

实现与右线程树中的搜索非常相似。

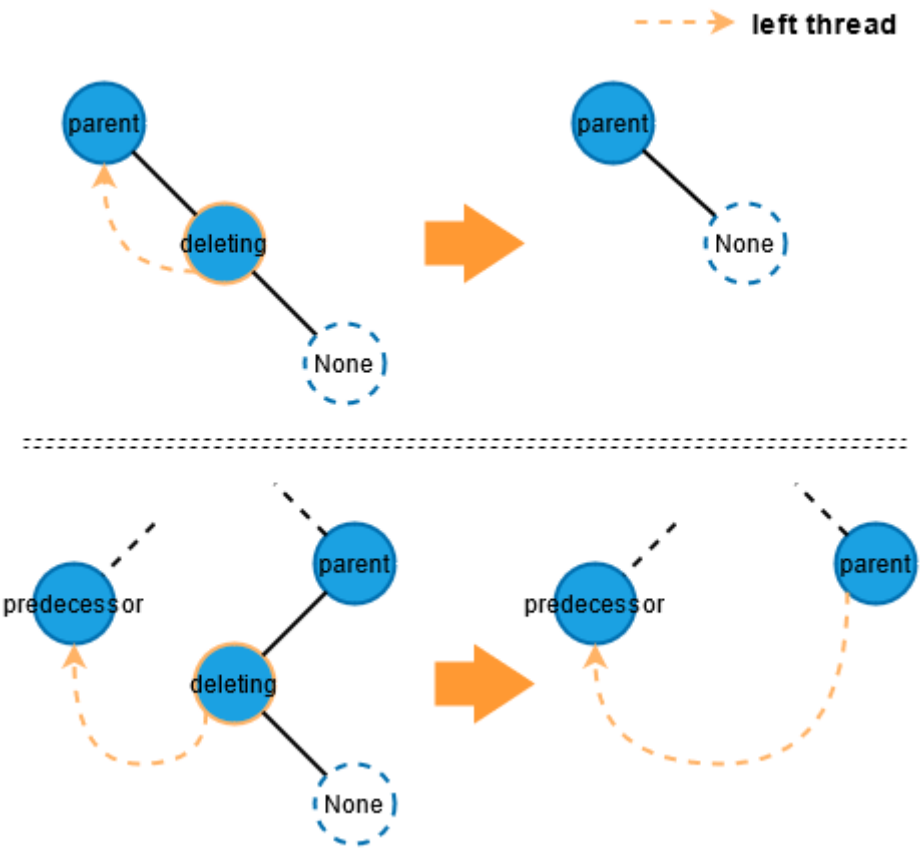
Python 复制代码

```python
def search(self, key: Any) -> Optional[Node]:
    current = self.root

    while current:
        if key == current.key:
            return current
        elif key < current.key:
            if current.is_thread is False:
                current = current.left
            else:
                break
        else:  # key > current.key:
            current = current.right
    return None
```

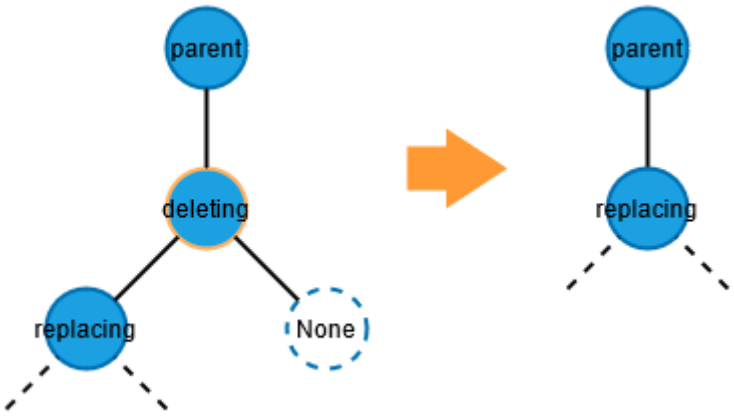
删除

同理，左线程二叉树的删除与右线程二叉树对称，有三种情况：无子树、只有一个子树、两个子树。

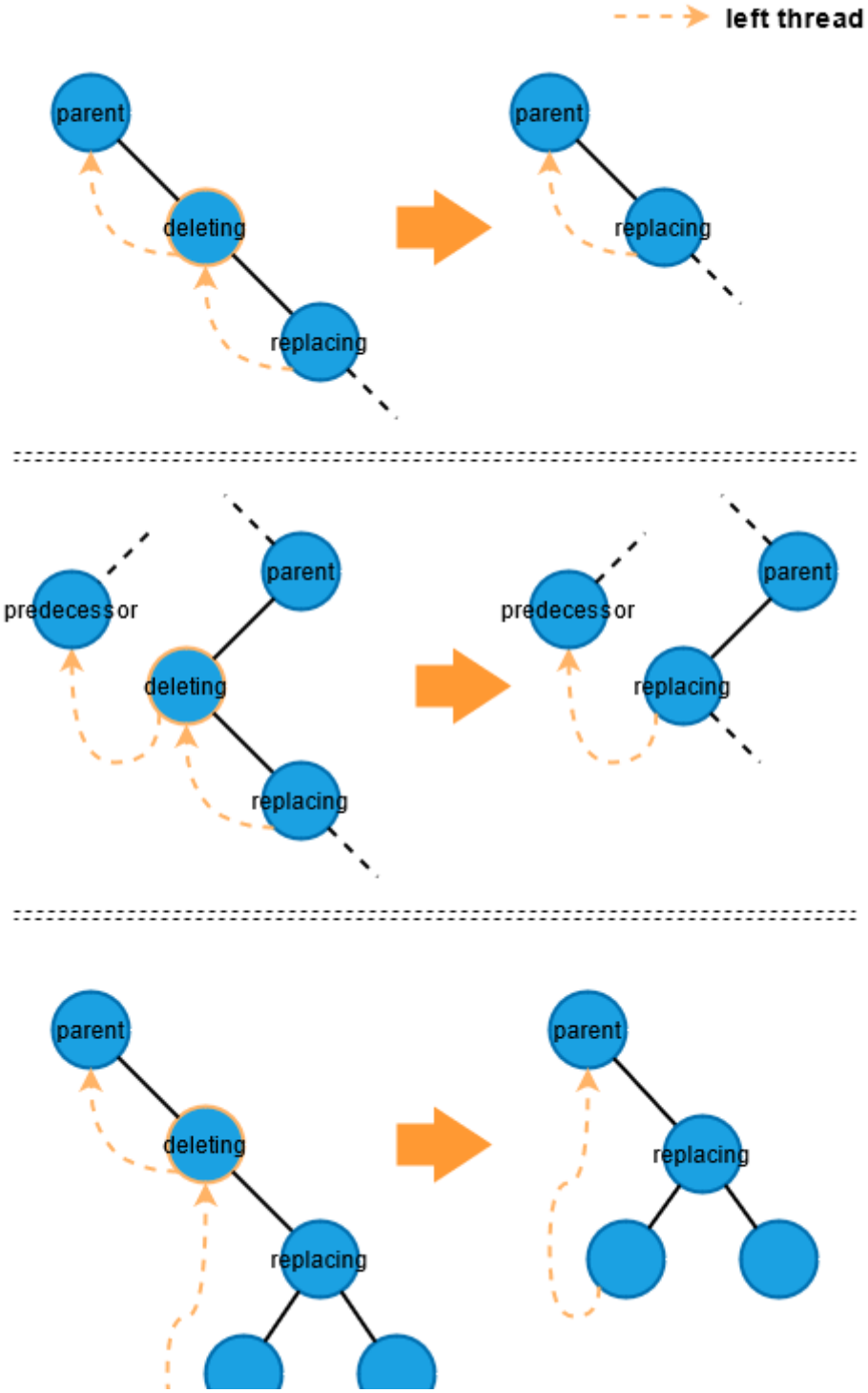
案例 1：没有孩子

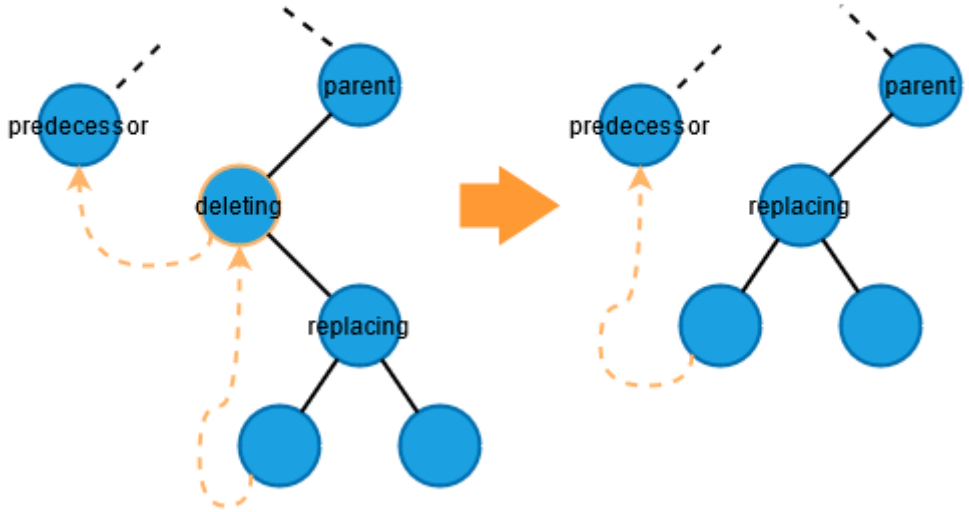


案例 2a：只有一个左孩子



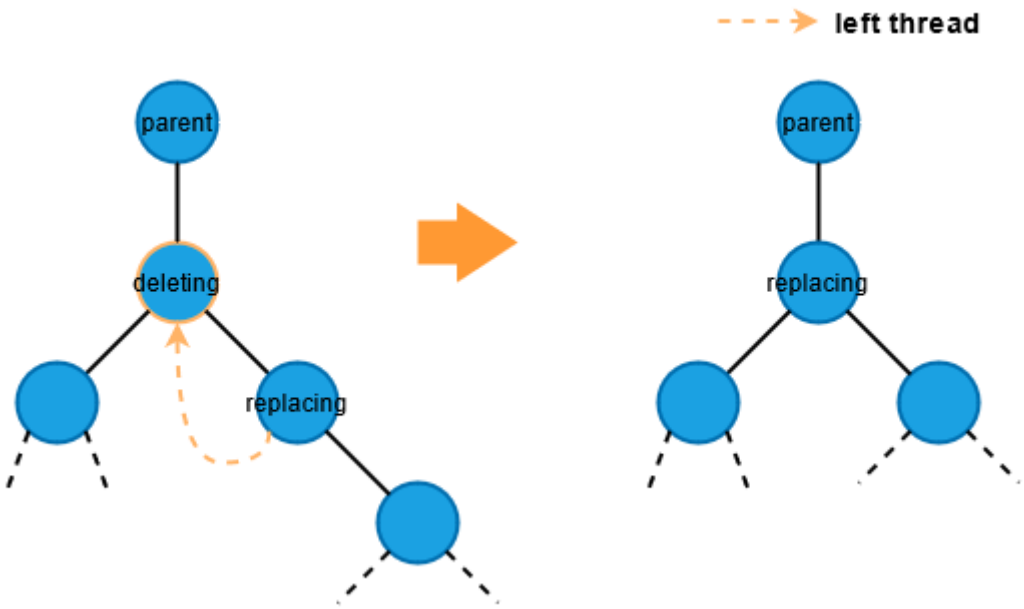
情况 2b：只有一个合适的孩子





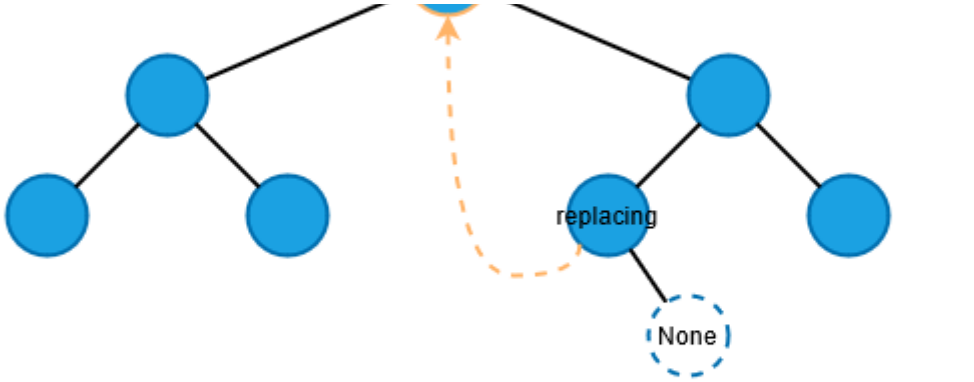
案例 3：两个孩子

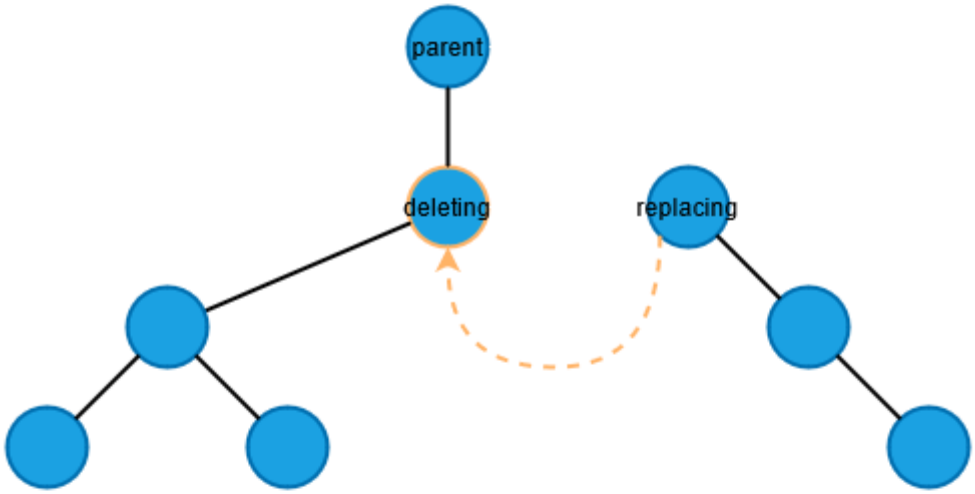
3.a 删除节点的右子节点也是右子树中最左边的节点。

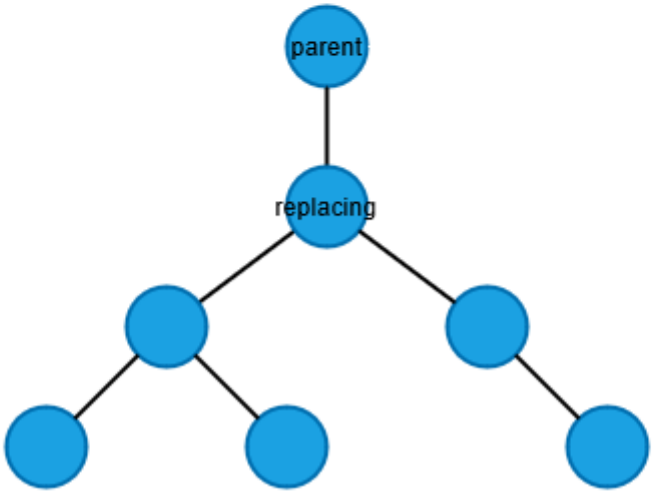


3.b. 删除节点的右孩子也有两个孩子。

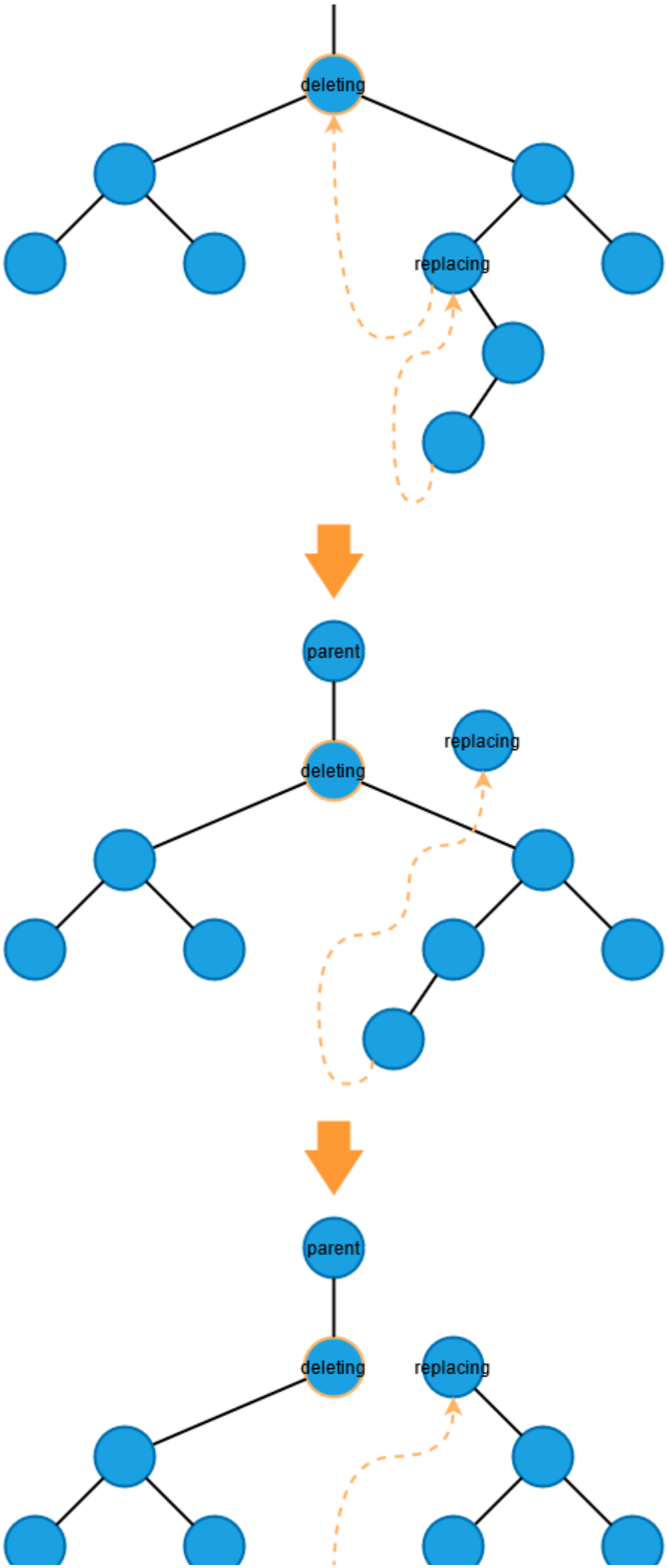
替换节点没有子节点：

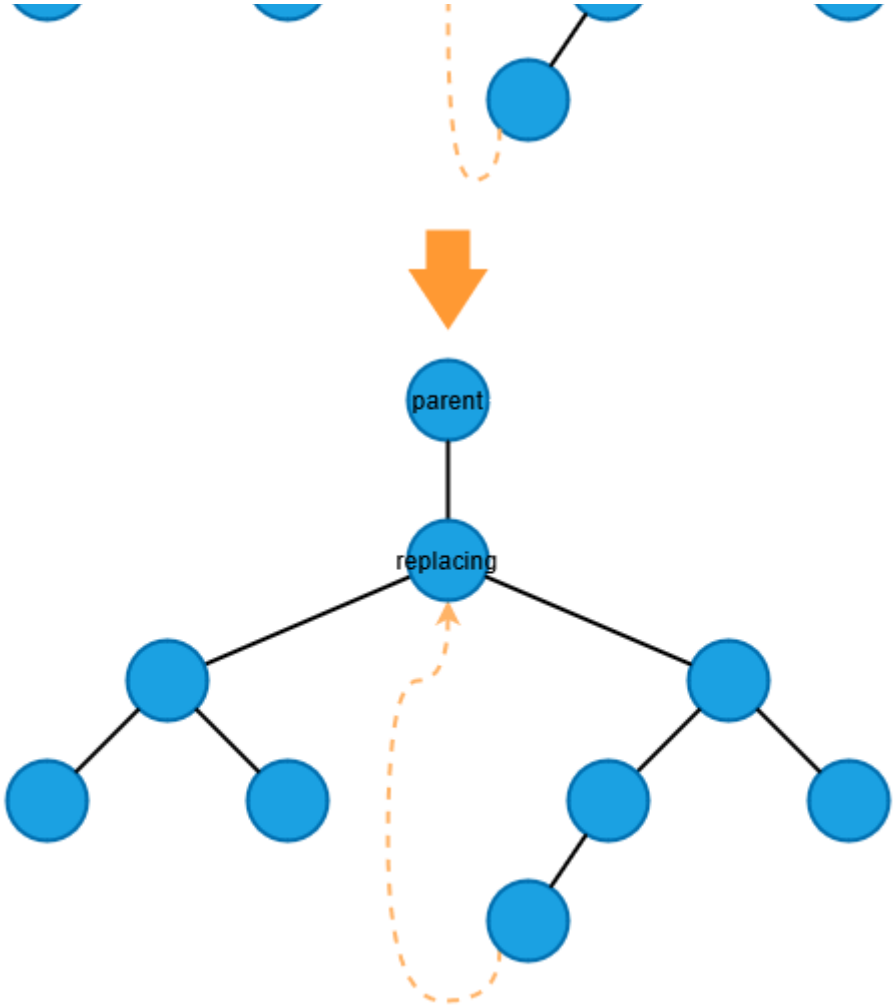






替换节点只有一个右孩子:


deleting
replacing
parent
deleting
replacing
parent
deleting
replacing



和右线程二叉树一样，我们也使用我们在二叉搜索树删除中做的移植技术，用要删除的节点替换子树。

Python 缩小▲ 复制代码

```python
def delete(self, key: Any) -> None:
    if self.root and (deleting_node := self.search(key=key)):
        # Case 1: no child
        if deleting_node.right is None and (
            deleting_node.left is None or deleting_node.is_thread
        ):
            self._transplant(deleting_node=deleting_node, replacing_node=None)

        # Case 2a: only one right child
        elif deleting_node.right and deleting_node.is_thread:
            successor = self.get_successor(node=deleting_node)
            if successor:
                successor.left = deleting_node.left
            self._transplant(
                deleting_node=deleting_node, replacing_node=deleting_node.right
            )

        # Case 2b: only one left left child
        elif (deleting_node.right is None) and (deleting_node.is_thread is False):
            self._transplant(
                deleting_node=deleting_node, replacing_node=deleting_node.left
            )

        # Case 3: two children
        elif deleting_node.right and deleting_node.left:
            replacing_node: Node = self.get_leftmost(node=deleting_node.right)
            successor = self.get_successor(node=replacing_node)
            # the leftmost node is not the direct child of the deleting node
            if replacing_node.parent != deleting_node:
                self._transplant(
```

```
                    deleting_node=replacing_node,
                    replacing_node=replacing_node.right,
                )
                replacing_node.right = deleting_node.right
                replacing_node.right.parent = replacing_node

            self._transplant(
                deleting_node=deleting_node, replacing_node=replacing_node
            )
            replacing_node.left = deleting_node.left
            replacing_node.left.parent = replacing_node
            replacing_node.is_thread = False
            if successor and successor.is_thread:
                successor.left = replacing_node
        else:
            raise RuntimeError("Invalid case. Should never happened")

def _transplant(self, deleting_node: Node, replacing_node: Optional[Node]) -> None:
    if deleting_node.parent is None:
        self.root = replacing_node
        if self.root:
            self.root.is_thread = False
    elif deleting_node == deleting_node.parent.left:
        deleting_node.parent.left = replacing_node
        if replacing_node:
            if deleting_node.is_thread:
                if replacing_node.is_thread:
                    replacing_node.left = deleting_node.left
        else:
            deleting_node.parent.left = deleting_node.left
            deleting_node.parent.is_thread = True
    else:  # deleting_node == deleting_node.parent.right
        deleting_node.parent.right = replacing_node
        if replacing_node:
            if deleting_node.is_thread:
                if replacing_node.is_thread:
                    replacing_node.left = deleting_node.left

    if replacing_node:
        replacing_node.parent = deleting_node.parent
```

获取高度

该`get_height`函数与右线程二叉树对称。

复制代码

```
@staticmethod
def get_height(node: Optional[Node]) -> int:
    if node:
        if node.right and node.is_thread is False:
            return (
                max(
                    LeftThreadedBinaryTree.get_height(node.left),
                    LeftThreadedBinaryTree.get_height(node.right),
                )
                + 1
            )

        if node.right:
            return LeftThreadedBinaryTree.get_height(node=node.right) + 1

        if node.is_thread is False:
            return LeftThreadedBinaryTree.get_height(node=node.left) + 1

    return 0
```

获取最左边和最右边的节点

由于左线程树与右线程树对称，所以我们在尝试获取最左边的节点时需要检查是否`is_thread`是，`True`并检查左属性是否为空。

Python 复制代码

```
@staticmethod
def get_leftmost(node: Node) -> Node:
    current_node = node

    while current_node.left and current_node.is_thread is False:
        current_node = current_node.left
    return current_node
```

该`get_rightmost`实施是相同的`get_rightmost`二进制搜索树。

Python 复制代码

```
@staticmethod
def get_rightmost(node: Node) -> Node:
    current_node = node
    while current_node.right:
        current_node = current_node.right
    return current_node
```

前任和继任者

根据左线程树的定义：节点的空左属性指向其有序前驱。如果节点`is_thread`是，我们可以通过跟随线程来简单地获取节点的前任`True`。

Python 复制代码

```
@staticmethod
def get_predecessor(node: Node) -> Optional[Node]:
    if node.is_thread:
        return node.left
    else:
        if node.left:
            return LeftThreadedBinaryTree.get_rightmost(node=node.left)
        # if node.left is None, it means no predecessor of the given node.
        return None
```

的实现`get_successor`与二叉搜索树的后继者相同。

Python 复制代码

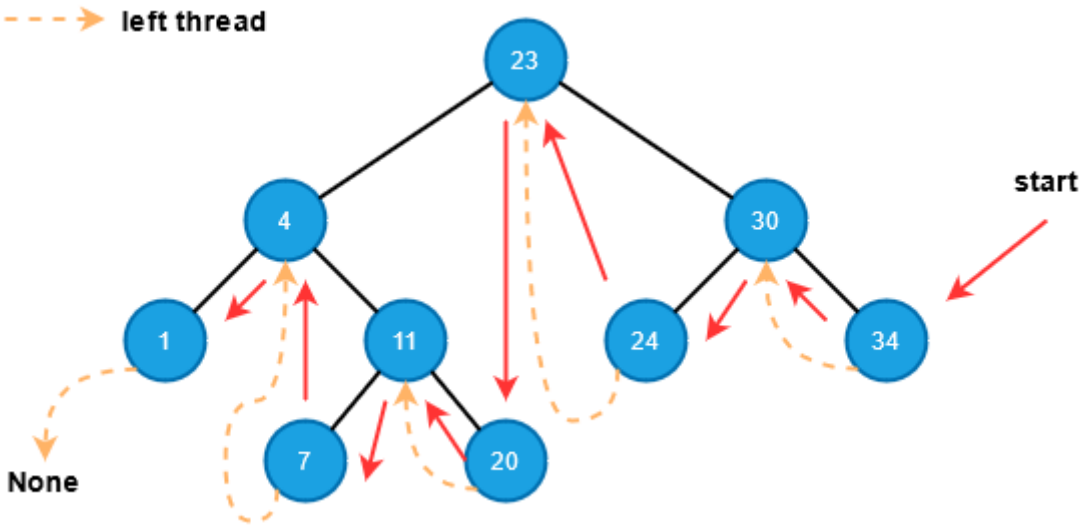
```
@staticmethod
def get_successor(node: Node) -> Optional[Node]:
    if node.right:
        return LeftThreadedBinaryTree.get_leftmost(node=node.right)
    parent = node.parent
    while parent and node == parent.right:
        node = parent
        parent = parent.parent
    return parent
```

逆序遍历

对于左线程，左线程树不需要递归或辅助堆栈来执行逆序遍历。

1. 从整个树的最右边的节点开始。
2. 如果left属性是线程，则跟随线程；如果 left 属性不是线程，则转到子树最右侧的节点。
3. 重复步骤 2 直到左属性为`None`。

下图中的红色箭头展示了逆序遍历的线程化方式。



下面是实现：

Python 复制代码

```
def reverse_inorder_traverse(self) -> traversal.Pairs:
    if self.root:
        current: Optional[Node] = self.get_rightmost(node=self.root)
        while current:
            yield (current.key, current.data)

            if current.is_thread:
                current = current.left
            else:
                if current.left is None:
                    break
                current = self.get_rightmost(current.left)
```

测试

与往常一样，我们应该尽可能多地对我们的代码进行单元测试。在这里，我们使用在构建二叉搜索树中创建的conftest.py中的`basic_tree`函数来测试我们的单线程二叉树。检查test_single_threaded_binary_trees.py以获取完整的单元测试。

分析

线程化二叉树操作的运行时间与普通二叉搜索树相同。

Operations	Average	Worst
Insert	$O(log_2 n)$	$O(n)$
Search	$O(log_2 n)$	$O(n)$
Delete	$O(log_2 n)$	$O(n)$
Leftmost (Min)	$O(log_2 n)$	$O(n)$
Rightmost (Max)	$O(log_2 n)$	$O(n)$
Predecessor	$O(log_2 n)$	$O(n)$
Successor	$O(log_2 n)$	$O(n)$

虽然右线程树提供了一种更简单的方法来检索节点的后继节点，而左线程树提供了一种更直接的方法来获取节点的前任，但线程并没有节省大量的运行时间。主要原因是这些函数仍然需要调用`get_leftmost`和`get_rightmost`函数，它们的平均运行时间为 O(lg n)，最坏情况下的运行时间为 O(n)。

但是，线程确实有助于特定遍历的空间复杂度。

Type of Threaded Trees	Traversal Type	Space Complexity
Right Threaded Tree	In-Order Traversal	$O(1)$
	Pre-Order Traversal	$O(1)$
Left Threaded Tree	Reverse In-Order Traversal	$O(1)$

例子

将线程添加到二叉搜索树使其实现更加复杂，但是当遍历很重要但涉及空间消耗时，它们可能是一种解决方案。例如，我们想建立一个数据库，用户会经常按升序和降序访问数据，但我们的内存有限（即，无法使用辅助堆栈或递归方法，因为涉及空间消耗）。在这种情况下，我们可以使用线程化二叉树来实现升序和降序访问的索引。

Python 缩小▲ 复制代码

```python
from typing import Any

from forest.binary_trees import single_threaded_binary_trees
from forest.binary_trees import traversal

class MyDatabase:
    """Example using threaded binary trees to build an index."""

    def __init__(self) -> None:
        self._left_bst = single_threaded_binary_trees.LeftThreadedBinaryTree()
        self._right_bst = single_threaded_binary_trees.RightThreadedBinaryTree()

    def _persist(self, payload: Any) -> str:
        """Fake function pretent storing data to file system.

        Returns
        -------
        str
            Path to the payload.
```

```
        """
        return f"path_to_{payload}"

    def insert_data(self, key: Any, payload: Any) -> None:
        """Insert data.

        Parameters
        ----------
        key: Any
            Unique key for the payload
        payload: Any
            Any data
        """
        path = self._persist(payload=payload)
        self._left_bst.insert(key=key, data=path)
        self._right_bst.insert(key=key, data=path)

    def dump(self, ascending: bool = True) -> traversal.Pairs:
        """Dump the data.

        Parameters
        ----------
        ascending: bool
            The order of data.

        Yields
        ------
        Pairs
            The next (key, data) pair.
        """
        if ascending:
            return self._right_bst.inorder_traverse()
        else:
            return self._left_bst.reverse_inorder_traverse()

if __name__ == "__main__":

    # Initialize the database.
    my_database = MyDatabase()

    # Add some items.
    my_database.insert_data("Adam", "adam_data")
    my_database.insert_data("Bob", "bob_data")
    my_database.insert_data("Peter", "peter_data")
    my_database.insert_data("David", "david_data")

    # Dump the items in ascending order.
    print("Ascending...")
    for contact in my_database.dump():
        print(contact)

    print("\nDescending...")
    # Dump the data in decending order.
    for contact in my_database.dump(ascending=False):
        print(contact)
```

（完整的示例可在single_tbst_database.py 中找到。）

输出将如下所示：

复制代码

```
Ascending...
('Adam', 'path_to_adam_data')
('Bob', 'path_to_bob_data')
('David', 'path_to_david_data')
('Peter', 'path_to_peter_data')

Descending...
```

```
('Peter', 'path_to_peter_data')
('David', 'path_to_david_data')
('Bob', 'path_to_bob_data')
('Adam', 'path_to_adam_data')
```

概括

尽管添加线程会增加复杂性并且不会提高运行时性能，但线程化二叉树利用浪费的左或右属性并提供一种简单的方法来检索前驱或后继，并提供一种无需使用堆栈或递归方法即可执行某些遍历的解决方案。当涉及空间复杂性，并且特定的遍历（例如，有序遍历）很关键时，线程二叉树可能是一种解决方案。

下面的文章将构建具有单线程二叉树优点但也更复杂的双线程二叉树。

历史

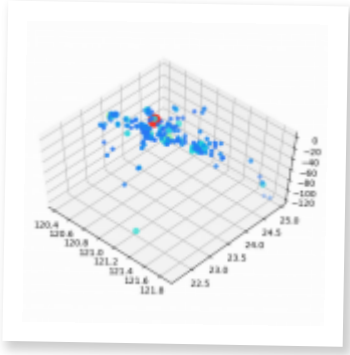
- 路
- 3 2021四月：初始版本

本文最初发布于https://shunsvineyard.info/2021/04/02/build-the-forest-in-python-series-single-threaded-binary-search-trees

执照

分享

关于作者



顺煌

软件开发人员（高级）
美国

手表该会员

我叫舜。我是一名软件工程师，也是一名基督徒。我目前在一家初创公司工作。
我的网站：https
://shunsvineyard.info 邮箱：zsh@shunsvineyard.info

评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

-- 本论坛暂无消息 --