Binary Search Tree Interview Question Practice

Jayden Patel

The Assignment

Create a Binary Search Tree, then add the following features:

- Delete a node from a BST:
  - Write a function to delete a given node from a BST while maintaining its BST properties.
- Find the in-order successor of a given node in a BST:
  - Write a function to find the in-order successor of a given node in a BST. The in-order successor is the node with the next higher key value in the in-order traversal of the BST.
- Convert a sorted array into a balanced BST
  - Given a sorted array, write a function to convert it into a height-balanced BST.
- Rotate left and rotate right:
  - Give a node, write a function that can either rotate it right or left.

Helper Functions

- The first function simply returns true or false if the given value is in the tree (or sub tree if not starting at the root node).
- The second function returns the minimum value node in a tree (or sub tree) starting at the given node.
- The third function updates the current node's height using its children's height.

```
def contains(self, value):
    if self.value == value: return True
    if value < self.value and self.left:
        return self.left.contains(value)
    if value > self.value and self.right:
        return self.right.contains(value)
    return False
```

```
def minValueNode(self, node):
    current = node
    while current.left is not None:
        current = current.left
    return current
```

```
def update_height(self):
    self.height = 0
    left_height = self.left.height if self.left else 0
    right_height = self.right.height if self.right else 0
    self.height = 1 + max(left_height, right_height)
```

Deleting a node

- When a value is being deleted, the tree calls a recursive delete function on the root node (on the next slide).
- This recursive function slides down the tree until the requested value is found, then the node is deleted.

```
def delete(self, key):
    if type(key) not in [int, float]: raise
       Exception("invalid input data type")
    if self.root and self.contains(key):
        self.root = self.root.delete(self.root,key)
        self.count -= 1
```

Deleting a node cont.

- If the node being deleted has only one child, its child is simply remapped to the node's parent.
- If the node has two children, then the inorder successor of node's sub tree takes its place.
- The rest of the tree after this successor is shifted upwards following the same principles as the delete function.

```
def delete(self, next, key):
    if next is None:
        return next

    if key < next.value:
        next.left = self.delete(next.left, key)
    elif key > next.value:
        next.right = self.delete(next.right, key)
    else:
        if next.left is None:
            temp = next.right
            next = None
            return temp
        elif next.right is None:
            temp = next.left
            next = None
            return temp

        temp = self.minValueNode(next.right)
        next.value = temp.value
        next.right = self.delete(next.right, temp.value)
    next.balance_tree()
    return next
```

Inorder Successor

- The function slides down the tree until the given value is found.
  - If the value is not in the tree, then the successor is None.
- Once the node is found, the BST calls the inOrderSuccessor function on that node.
  - If the node does not have children, then it starts at the tree's root.

```
def inorderSuccessor(self, key):
    if type(key) not in [int, float]: raise
        Exception("invalid input data type")
    node = self.root
    while node is not None and node.value != key:
        if key < node.value:
            node = node.left
        else:
            node = node.right
    successor = None
    if node is None:
        return None
    if node.right is None and node.left is None:
        successor = node.inorderSuccessor(key, self.root)
    else: successor = node.inorderSuccessor(key)
    return successor.value if successor is not None else
None
```

Inorder Successor cont.

- When inOrderSuccesor is called on a node, it checks if there are any nodes to the right.
  - If there are, the successor is the minimum value node in that sub tree.
- Otherwise, it searches for a successor on the left sub tree.

```
def inorderSuccessor(self, key, root=None):
        if self.right is not None:
            return self.minValueNode(self.right)
        successor = None
        start = self
        if root: start = root
        while start is not None:
            if key < start.value:
                successor = start
                start = start.left
            elif key > start.value:
                start = start.right
            else:
                break
        return successor
```

Importing Data

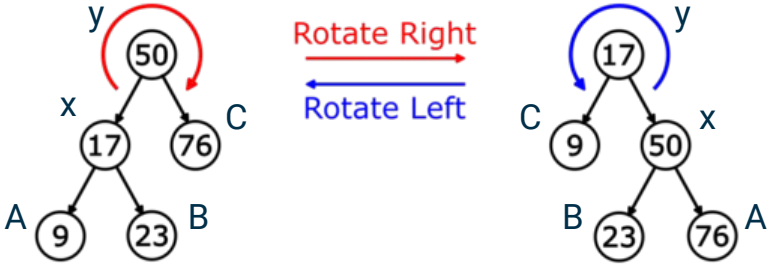
- In the constructor of the BST, there is an argument to include a list or tuple of items to add to the tree.
  - It iterates through the given data and adds each item to the tree.
- Similarly, in the tree's add function, it checks if the given value is a list or tuple. If so, it iterates through and adds each to the tree.
- Through this implementation, it does not matter if the input data is sorted.

```
def __init__(self, data=None):
    self.root = None
    self.count = 0
    self.min = 0
    self.max = 0
    if data:
        if type(data) in [list, tuple]:
            for value in data:
                self.add(value)
        else: raise Exception("invalid input data type")
```

```
def add(self, value):
    if type(value) in [list, tuple]:
        for item in value:
            self.add(item)
    ...
```

Rotating Nodes

- Capital letters represent subtrees, while lowercase letters represent values.
- In a right rotation, the current node's value is replaced with the value to the left. Then, a new node for the old value is added to the right.
- The sub trees are placed back in order:
  - A: right of the new node
  - B: left of the new node
  - C: left of the new root node

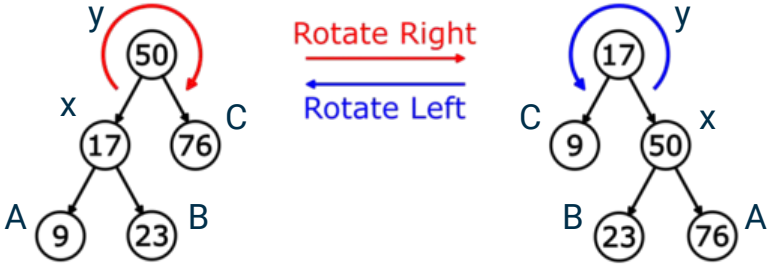



```
def rotate_right(self):
    if self.left:
        x = self.left.value
        y = self.value
        A = self.left.left
        B = self.left.right
        C = self.right

        self.value = x
        self.right = Node(y)
        self.left = A
        self.right.left = B
        self.right.right = C
        self.right.update_height()
        self.update_height()
    else: raise Exception("invalid rotation")
```

Rotating Nodes

- Rotations in BSTs are symmetric.
- This means that to rotate left, the code is the same except every direction is swapped.




```
def rotate_left(self):
    if self.right:
        x = self.right.value
        y = self.value
        A = self.right.right
        B = self.right.left
        C = self.left

        self.value = x
        self.left = Node(y)
        self.right = A
        self.left.right = B
        self.left.left = C
        self.left.update_height()
        self.update_height()
    else: raise Exception("invalid rotation")
```

Binary Search Tree Interview Question Practice

Jayden Patel