Binary Search Tree Interview Questions

by

Brent Orlina

Computer Programming III

Nicholas Seward

April 9, 2024

Delete Function

I implemented the delete function by letting the user pass in a value to the function. Then, it recursively checks whether it has found the corresponding value.

The way it does this is by looking at the current node's left and right children and seeing if they match the given value. If not, it goes left or right depending on if the value is higher or lower than the current value

If it matches with the left child, it looks for the node with largest value in the left subtree and sets its value to that.

If the left node with the largest value is the root of the left subtree, then the node simply sets its left child to None. If not, then the parent of the largest node sets its right child to the left child of the largest node.

The mirror applies for if the value matches the right child.

```
def delete(self, v):
    if self.l_child and v == self.l_child.value:
        child = self.l_child
        if child.l_child:
            child.l_child._replace_with_max(child, child)
        elif child.r_child:
            child.r_child._replace_with_min(child, child)
        else:
            self.l_child = None

    elif self.r_child and v == self.r_child.value:
        child = self.r_child
        if child.l_child:
            child.l_child._replace_with_max(child, child)
        elif child.r_child:
            child.r_child._replace_with_min(child, child)
        else:
            self.r_child = None

    elif self.l_child and v < self.value:
        self.l_child.delete(v)

    elif self.r_child and v > self.value:
        self.r_child.delete(v)
```

Guy puts code in image. Asked to leave Seward's classroom.

Delete Function: _replace_with_

These functions are supposed to replace the value of the Node to be deleted with the minimum value in the right subtree or the maximum value in the left subtree.

_replace_with_max is default, however if there isn't a left subtree it uses _replace_with_min, going to the right subtree.

P is the parent of the current node. This is done so that in case the node with the maximum value has a left child, P's right child can be set to that. The same applies for the right subtree with the node with the minimum value.

N is the node whose value is being replaced, so once it reaches the node with the maximum / minimum value, it just sets N's value to its value.

This is implemented recursively because I also needed to check if any of the nodes I visited finding the node with the maximum value is now out of balance after the deletion.

```
    def _replace_with_min(self, P, N):
        if not self.l_child:
            N.value = self.value
            if P == N: P.r_child = self.r_child
            else: P.l_child = self.r_child
        else:
            self.l_child._replace_with_max(self, N)

    def _replace_with_max(self, P, N):
        if not self.r_child:
            N.value = self.value
            if P == N: P.l_child = self.l_child
            else: P.r_child = self.l_child
        else:
            self.r_child._replace_with_max(self, N)
```

In-order Successor of a Value

I implemented this using DFS.

It searches through the tree, left subtrees first, then if it finds a number bigger than itself, it keeps track of that number. If it finds a number smaller than the current record, it replaces the record.

If it's found a number bigger than itself already and it's encountered a number smaller than itself, then we know we're done and we can early return.

This is because we're searching through the left subtrees first and everything to the right of the smaller number will always be bigger than the current record.

```python
def inorder_successor(T: BinaryTree, v):
    Q = deque([T.root])
    V = set()
    found_big = False
    found_num = None
    f = lambda x: x is not None and x not in V
    while Q:
        N = Q[-1]
        children = list(filter(f, [N.r_child, N.l_child]))
        if children:
            for child in children: Q.append(child); V.add(child)

        else:
            P = Q.pop()
            if P.value > v:
                found_big = True
                found_num = (
                    P.value
                    if found_num is None
                    else min(P.value, found_num)
                )

            if found_big and P.value < v:
                return found_num
    return None
```

Sorted Array into BST

I solved this using a recursive divide and conquer algorithm thing.

It starts with calculating the middle nodes and its two children. The middle node is the middle of the array.

The left child is the first quartile of the array and the right child is the third quartile of the array. Then, recursive_insert sets the left child's left and right children and the right child's left and right children with the same logic.

```python
def sorted_to_BST(V):
    assert list(sorted(V)) == V
    def recursive_insert(N: BasicNode, lo, hi):
        if lo == hi:
            return

        mid = (lo+hi) // 2
        lo_mid = (lo + mid) // 2
        hi_mid = (mid+1+lo) // 2

        N.l_child = BasicNode(V[lo_mid])
        N.r_child = BasicNode(V[hi_mid])

        N._update_height()

        recursive_insert(N.l_child, lo, mid)
        recursive_insert(N.r_child, mid+1, hi)

    # Kickstart since we need an initial middle node
    L = 0
    H = len(V)
    M = (L+H) // 2

    N = BasicNode(V[M])

    L_M = (L + M) // 2
    H_M = (M+1+H) // 2

    N.l_child = BasicNode(V[L_M])
    N.r_child = BasicNode(V[H_M])

    N._update_height()

    recursive_insert(N.l_child, L, M)
    recursive_insert(N.r_child, M+1, H)

    return N
```

Left and Right Rotation

The right rotation is implemented by switching the values of setting “switching” A and B, in the cool ASCII diagram.

A becomes the right child of B.

C then becomes the right child of A

E becomes the left child of A.

D becomes the left child of B.

The mirror is done for the left rotation.

```
    # Left Left Rotation
    # ------------------
    # This is actually rotating the node to the right.
    # This is for when A is LEFT-heavy and the LEFT-subtree
    # of B is taller than its right-subtree.
    # e.g.
    #                   (4) A
    #                     / \
    #               -> (3) B   C (1) <- Unbalanced! Rotate
    #                   / \            node A to the Right!
    #                (2) D   E (1)
    #                  /
    #              (1) F
    # We only have to do a simple right rotation on A.

    def _ll_rotate(self):
        a, b = self.value, self.l_child.value
        A = self.r_child
        B, C = self.l_child.r_child, self.l_child.l_child

        self.value = b
        self.r_child = Node(a)
        self.r_child.r_child = A
        self.r_child.l_child = B
        self.l_child = C

        self.r_child._update_height()
        self._update_height()
```

Right rotation