Binary Search Tree

Jacques Mock Schindler

Invalid Date

Ein binärer Suchbaum (Binary Search Tree, BST) ist eine Datenstruktur, die es ermöglicht, Daten in einer hierarchischen Struktur zu speichern. Jeder Knoten im Baum hat maximal zwei Kinder, wobei das linke Kind kleiner und das rechte Kind grösser als der Knoten selbst ist. Dies ermöglicht das effiziente Suchen, Einfügen und Löschen von Elementen.

Sollen beispielsweise die Zahlen 42, 23, 17, 34, 56, 78 und 12 der Reihe nach in einen binären Suchbaum eingefügt werden, geschieht dies wie folgt:

```
        42
       /  \
     23    56
    /  \      \
   17   34    78
  /
12
```

Um einen Knoten zu löschen, gibt es drei Fälle zu beachten: 1. Der Knoten ist ein Blatt (keine Kinder): Der Knoten kann einfach entfernt werden. 2. Der Knoten hat ein Kind: Das Kind ersetzt den Knoten. 3. Der Knoten hat zwei Kinder: Der Knoten wird durch den kleinsten Knoten im rechten Teilbaum ersetzt.

In den folgenden Abschnitten findet sich eine Mögliche Implementierung eines binären Suchbaums in Python.

Klasse BSTNode

```python
class BSTNode:
    def __init__(self, key, value=None):
        self.key = key
        self.value = value
        self.parent = None
        self.left = None
        self.right = None

    def __str__(self):
        key = str(self.key)
        parent = 'None' if self.parent is None else str(self.parent.key)
        left = 'None' if self.left is None else str(self.left.key)
        right = 'None' if self.right is None else str(self.right.key)
        s = (
            f'\tParent = {parent}\n'
            f'\tKey = {key}\n'
            f'Left = {left}\tRight = {right}'
        )
        return s
```

Klasse BST

```python
class BST:
    def __init__(self, key=None, value=None):
        if key is None:
            self.root = None
        else:
            node = BSTNode(key, value)
            self.root = node

    def insert(self, key, value=None, root=None):
        node = BSTNode(key, value)
        if self.root is None:
            self.root = node
            return

        if root is None:
            root = self.root

        if key < root.key and root.left is None:
            root.left = node
            node.parent = root
            return

        if key < root.key:
            root = root.left
            self.insert(key, value, root)
        if key > root.key and root.right is None:
            root.right = node
            node.parent = root
            return

        if key > root.key:
            root = root.right
            self.insert(key, value, root)

    def min(self, bst=None):
        if bst is None:
            minimum = self.root
        else:
            minimum =bst.root

        while minimum.left is not None:
            minimum = minimum.left

        return minimum

    def max(self, bst=None):
        if bst is None:
            maximum = self.root
        else:
            maximum = bst.root

        while maximum.right is not None:
            maximum = maximum.right

        return maximum

    def search(self, key, node=None):
        # If initial call or we've hit None in recursion
        if node is None:
            if self.root is None:  # Empty tree
                return -1
            node = self.root

        # Found the key
        if key == node.key:
            return node

        # Key doesn't exist in this path
        if key < node.key:
            if node.left is None:
                return -1
            return self.search(key, node.left)
        else:  # key > node.key
            if node.right is None:
                return -1
            return self.search(key, node.right)

    def delete(self, key):
        # Find the node to delete
        node = self.search(key)

        # If node not found, return
        if node == -1:
            return

        self._delete_node(node)

    def _delete_node(self, node):
        # Case 1: Node has no children (leaf node)
        if node.left is None and node.right is None:
            if node == self.root:
                self.root = None
            else:
                if node.parent.left == node:
                    node.parent.left = None
                else:
                    node.parent.right = None

        # Case 2: Node has only one child
        elif node.left is None:  # Has only right child
            if node == self.root:
                self.root = node.right
                node.right.parent = None
            else:
                if node.parent.left == node:
                    node.parent.left = node.right
                else:
                    node.parent.right = node.right
                node.right.parent = node.parent

        elif node.right is None:  # Has only left child
```

```python
            if node == self.root:
                self.root = node.left
                node.left.parent = None
            else:
                if node.parent.left == node:
                    node.parent.left = node.left
                else:
                    node.parent.right = node.left
                node.left.parent = node.parent

        # Case 3: Node has two children
        else:
            # Find successor (smallest node in right subtree)
            successor = None
            current = node.right

            while current.left is not None:
                current = current.left

            successor = current

            # Copy successor's key and value to the node
            node.key = successor.key
            node.value = successor.value

            # Delete the successor (which has at most one right child)
            self._delete_node(successor)

    def iterate(self, node=None, result=None):
        # Initialize result list on first call
        if result is None:
            result = []

        # Use root if no starting node provided
        if node is None:
            if self.root is None:  # Empty tree
                return result
            node = self.root

        # In-order traversal: left -> current -> right
        if node.left is not None:
            self.iterate(node.left, result)
        result.append(node)

        if node.right is not None:
            self.iterate(node.right, result)

        return result
```