**Interval trees** και **Segment trees** για interval και stabbing Queries αντίστοιχα. Μελέτη απόδοσης των βασικών πράξεων στις δύο δομές.

Ένα *interval tree* είναι μια δομή δεδομένων που χρησιμοποιείται για την αποτελεσματική αναζήτηση και αναζήτηση διαστημάτων ή τμημάτων ενός μονοδιάστατου χώρου. Συνήθως χρησιμοποιείται για την επίλυση προβλημάτων που αφορούν αλληλοεπικαλυπτόμενα διαστήματα, όπως ο προγραμματισμός εργασιών ή η αναζήτηση ελεύθερου χρόνου σε ένα ημερολόγιο.

Από την άλλη ένα *segment tree* είναι ένας τύπος δυαδικού δέντρου που χρησιμοποιείται για την αποθήκευση και αναζήτηση πληροφοριών σχετικά με διαστήματα ή τμήματα ενός μονοδιάστατου χώρου. Χρησιμοποιείται για την εκτέλεση ερωτημάτων εύρους σε έναν πίνακα ή λίστα δεδομένων, όπως η εύρεση του αθροίσματος, της μέγιστης ή της ελάχιστης τιμής μιας περιοχής στοιχείων. Οι κόμβοι φύλλων του δέντρου αντιπροσωπεύουν μεμονωμένα στοιχεία του πίνακα εισόδου και οι εσωτερικοί κόμβοι αντιπροσωπεύουν εύρη στοιχείων. Το δέντρο μπορεί να κατασκευαστεί σε χρόνο O(n log n) και υποστηρίζει ερωτήματα σε χρόνο O(log n), καθιστώντας το μια αποτελεσματική δομή δεδομένων για την επεξεργασία μεγάλων ποσοτήτων δεδομένων.

Όσων αφορά την πολυπλοκότητα των δομών:

* Σε ένα interval tree, η χρονική πολυπλοκότητα της λειτουργίας ενός query είναι O(log n + k), όπου n είναι ο αριθμός των διαστημάτων στο δέντρο και k είναι ο αριθμός των διαστημάτων που τέμνονται με το διάστημα ερωτήματος. Η πολυπλοκότητα του χώρου είναι O(n), όπου n είναι ο αριθμός των διαστημάτων στο δέντρο.
* Σε ένα segment tree, η χρονική πολυπλοκότητα της λειτουργίας ενός query είναι O(log n), όπου n είναι ο αριθμός των στοιχείων στον πίνακα εισόδου. Η πολυπλοκότητα του χώρου είναι O(n), όπου n είναι ο αριθμός των στοιχείων στον πίνακα εισόδου.

Στον κώδικά μου έχω δύο αρχεία. Αρχικά το myIntervalTree.py, στο οποίο έχω υλοποιήσει ένα interval tree:

class IntervalTreeNode:

    def \_\_init\_\_(self, low, high):

        self.low = low

        self.high = high

        self.max = high

        self.left = None

        self.right = None

class IntervalTree:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, low, high):

        self.root = self.\_insert(self.root, low, high)

    def \_insert(self, node, low, high):

        if not node:

            return IntervalTreeNode(low, high)

        if low < node.low:

            node.left = self.\_insert(node.left, low, high)

        else:

            node.right = self.\_insert(node.right, low, high)

        node.max = max(node.max, high)

        return node

    def delete(self, low, high):

        self.root = self.\_delete(self.root, low, high)

    def \_delete(self, node, low, high):

        if not node:

            return None

        if low == node.low and high == node.high:

            if not node.left:

                return node.right

            if not node.right:

                return node.left

            min\_node = self.\_find\_min(node.right)

            node.low = min\_node.low

            node.high = min\_node.high

            node.right = self.\_delete(node.right, min\_node.low, min\_node.high)

        elif low < node.low:

            node.left = self.\_delete(node.left, low, high)

        else:

            node.right = self.\_delete(node.right, low, high)

        node.max = max(node.high, self.\_max(node.left), self.\_max(node.right))

        return node

    def search(self, low, high):

        return self.\_search(self.root, low, high)

    def \_search(self, node, low, high):

        if not node:

            return None

        if node.low <= high and low <= node.high:

            return (node.low, node.high)

        if node.left and node.left.max >= low:

            return self.\_search(node.left, low, high)

        return self.\_search(node.right, low, high)

    def \_find\_min(self, node):

        while node.left:

            node = node.left

        return node

    def \_max(self, node):

        return node.max if node else -float('inf')

    def update(self, low, high, new\_low, new\_high):

        self.delete(low, high)

        self.insert(new\_low, new\_high)

    def print\_nodes(self):

        self.\_print\_nodes(self.root)

    def \_print\_nodes(self, node):

        if node:

            self.\_print\_nodes(node.left)

            print(f'[{node.low}, {node.high}]')

            self.\_print\_nodes(node.right)

Και στη συνέχεια στο mySegmentTree.py, ένα segment tree αντίστοιχα.

class SegmentTree:

    def \_\_init\_\_(self, intervals):

        # Initialize the tree with the given intervals

        self.intervals = intervals

        self.tree = [None] \* (4 \* len(intervals))

        self.build(1, 0, len(intervals) - 1)

    def build(self, node, start, end):

    # Recursively build the segment tree

        if start == end:

            self.tree[node] = [self.intervals[start]]

        else:

            mid = (start + end) // 2

            left\_child = 2 \* node

            right\_child = 2 \* node + 1

            self.build(left\_child, start, mid)

            self.build(right\_child, mid + 1, end)

            self.tree[node] = []

            if self.tree[left\_child] is not None:

                self.tree[node] += self.tree[left\_child]

            if self.tree[right\_child] is not None:

                self.tree[node] += self.tree[right\_child]

            self.tree[node] = sorted(self.tree[node])

    def insert(self, interval):

        # Insert a new interval into the tree

        self.intervals.append(interval)

        self.\_insert(1, 0, len(self.intervals) - 1, len(self.intervals) - 1)

    def \_insert(self, node, start, end, idx):

        # Recursively insert an interval into the tree

        if start == end:

            self.tree[node] = [self.intervals[idx]]

        else:

            mid = (start + end) // 2

            left\_child = 2 \* node

            right\_child = 2 \* node + 1

            if len(self.tree) <= right\_child:

                # Increase the size of the tree list to hold the new node

                self.tree += [None] \* (right\_child - len(self.tree) + 1)

            if idx <= mid:

                # Ensure that left child is initialized to an empty list if it is not yet set

                if self.tree[left\_child] is None:

                    self.tree[left\_child] = []

                self.\_insert(left\_child, start, mid, idx)

            else:

                # Ensure that right child is initialized to an empty list if it is not yet set and within range

                if right\_child < len(self.tree) and self.tree[right\_child] is None:

                    self.tree[right\_child] = []

                self.\_insert(right\_child, mid + 1, end, idx)

            # Concatenate the sorted lists of the left and right children, checking for None values

            left\_list = self.tree[left\_child] if self.tree[left\_child] is not None else []

            right\_list = self.tree[right\_child] if right\_child < len(self.tree) and self.tree[right\_child] is not None else []

            self.tree[node] = sorted(left\_list + right\_list)

    def delete(self, idx):

        # Delete an interval from the tree by index

        self.intervals.pop(idx)

        self.\_delete(1, 0, len(self.intervals) - 1, idx)

    def \_delete(self, node, start, end, idx):

        # Recursively delete an interval from the tree

        if start == end:

            self.tree[node] = [self.intervals[start]]

        else:

            mid = (start + end) // 2

            left\_child = 2 \* node

            right\_child = 2 \* node + 1

            if idx <= mid:

                self.\_delete(left\_child, start, mid, idx)

            else:

                self.\_delete(right\_child, mid + 1, end, idx)

            self.tree[node] = sorted(self.tree[left\_child] + self.tree[right\_child])

    def update(self, idx, new\_interval):

        # Update an interval in the tree by index

        self.intervals[idx] = new\_interval

        self.\_update(1, 0, len(self.intervals) - 1, idx)

    def \_update(self, node, start, end, idx):

        # Recursively update an interval in the tree

        if start == end:

            self.tree[node] = [self.intervals[idx]]

        else:

            mid = (start + end) // 2

            left\_child = 2 \* node

            right\_child = 2 \* node + 1

            if idx <= mid:

                self.\_update(left\_child, start, mid, idx)

            else:

                self.\_update(right\_child, mid + 1, end, idx)

            self.tree[node] = sorted(self.tree[left\_child] + self.tree[right\_child])

    def query(self, node, start, end, x):

        # Query the tree to find intervals that contain x

        if self.tree[node] is None:

            return []

        if start == end:

            return [interval for interval in self.tree[node] if interval[0] <= x <= interval[1]]

        else:

            mid = (start + end) // 2

            left\_child = 2 \* node

            right\_child = 2 \* node + 1

            if x <= mid:

                return self.query(left\_child, start, mid, x)

            else:

                return self.query(right\_child, mid + 1, end, x) + self.query(left\_child, start, mid, x)

    def query\_point(self, x):

        # Query the tree for a point

        result = self.query(1, 0, len(self.intervals) - 1, x)

        return list(set(result))

Αφού έχω υλοποιήσιε τις δύο δομές δεδομένων, κάνω performance testing χρονομετρώτας τον χρόνο υλοποίησης των πράξεών στις δύο δομες.   
  
Για το interval tree:

import time

from myIntervalTree import IntervalTree

tree = IntervalTree()

# Insert intervals

tree.insert(1, 5)

tree.insert(2, 9)

tree.insert(3, 8)

tree.insert(6, 10)

start\_time = time.time()

tree.insert(8, 9)

end\_time = time.time()

print("Time for Insert:", end\_time - start\_time, "seconds")

print('Nodes after Insert:')

tree.print\_nodes()

# Delete an interval

start\_time = time.time()

tree.delete(2, 9)

end\_time = time.time()

print("Time for Delete:", end\_time - start\_time, "seconds")

print('Nodes after Delete:')

tree.print\_nodes()

# Update an interval

start\_time = time.time()

tree.update(3, 8, 10, 15)

end\_time = time.time()

print("Time for Update:", end\_time - start\_time, "seconds")

print('Nodes after Update:')

tree.print\_nodes()

# Search for an interval

start\_time = time.time()

result = tree.search(8, 9)

end\_time = time.time()

print("Time for Search:", end\_time - start\_time, "seconds")

print(f'First intersection: {result}')

Και αντίστοιχα για το segment tree:

import time

from mySegmentTree import SegmentTree

intervals = [(1, 5), (2, 9), (3, 8), (6, 10), (8, 9)]

tree = SegmentTree(intervals)

# Query the tree for a point

start\_time = time.time()

result = tree.query\_point(2)

end\_time = time.time()

print(result)

print("Time for query:", end\_time - start\_time, "seconds")

print(result)  # Output: [(2, 9), (1, 5)]

# Insert a new interval into the tree

start\_time = time.time()

tree.insert((7, 11))

end\_time = time.time()

print(tree.intervals)  # Output: [(1, 5), (2, 9), (3, 8), (6, 10), (8, 9), (7, 11)]

print("Time for Insert:", end\_time - start\_time, "seconds")

# Query the tree for a point after inserting a new interval

result = tree.query\_point(5)

print(result)  # Output: [(2, 9), (3, 8), (1, 5)]

# Update an interval in the tree

start\_time = time.time()

tree.update(0, (2, 4))

end\_time = time.time()

print("Time for Update:", end\_time - start\_time, "seconds")

print(tree.intervals)  # Output: [(2, 4), (2, 9), (3, 8), (6, 10), (8, 9), (7, 11)]

# Query the tree for a point after updating an interval

result = tree.query\_point(3)

print(result)  # Output: [(2, 9), (3, 8), (2, 4)]

# Delete an interval from the tree

start\_time = time.time()

tree.delete(2)

end\_time = time.time()

print(tree.intervals)  # Output: [(2, 4), (2, 9), (6, 10), (8, 9), (7, 11)]

print("Time for Delete:", end\_time - start\_time, "seconds")

# Query the tree for a point after deleting an interval

result = tree.query\_point(9)

print(result)  # Output: [(2, 9), (7, 11), (6, 10)]

Επειδή μονο με αυτες τις εγγραφές ο χρόνος πηγαίνει προσεγγιστικά στο 0, επαναλαμβάνω τη διαδικασία στα αρχεία advPerfTestingIntervalTree.py και advPerfTestingSegmentTree.py με 1000 εγγραφές όπου τα αποτελέσματα είναι σαφέστερα.