Advanced Data Structures (ADS-MIRI):

Assignment
Free Implementation of Interval Trees

Ali Arabyarmohammadi June 2023

1 Introduction

This report presents the implementation of an Interval Tree and the solution to the linear stabbing problem on it. An Interval Tree is a type of binary search tree where each node contains an interval instead of a single value. They are primarily used for efficiently finding all intervals that overlap with any given point or interval, which is also known as the stabbing problem. In this assignment, I implemented the Interval Tree in Python, created methods for inserting intervals into the tree and searching for all intervals that overlap with a given point. Then, I tested this implementation with various sets of intervals and search points to observe the performance and results.

2 Method

Each node of the Interval Tree contains an interval and a maximum value, which is the maximum endpoint of all intervals in the subtree rooted at that node. The insert operation works by recursively inserting the new interval into either the left or right subtree, depending on the low endpoint of the interval. Then, it updates the max value of the node if necessary. The search operation returns all intervals in the tree that overlap with a given point. It does this by recursively searching either one or both subtrees, depending on the max values and the intervals at each node. For more details, refer to Appendix A for the full code listings.

The ability of the algorithm to handle edge cases is important for several reasons:

1. Robustness: An algorithm that can handle edge cases is more robust and less likely to fail or produce incorrect results when given unusual input. In the context of the Interval Tree, edge cases could include situations where the search point does not fall within any interval, or where all intervals overlap with the search point. The ability to handle these cases correctly indicates that the algorithm is robust and reliable.

2. Completeness: Handling edge cases ensures that the algorithm works correctly for all possible input, not just for "typical" cases. This is an important aspect of algorithm design and is necessary to ensure that the algorithm is a complete solution to the problem it's designed to solve.

3. Performance: Understanding how an algorithm performs on edge cases can give us important insights into its performance characteristics. For example, if an algorithm performs poorly on certain edge cases, it may need to be optimized or redesigned to handle these cases more efficiently. In the case of the Interval Tree, the algorithm's ability to handle edge cases efficiently contributes to its overall performance and makes it a practical solution for the linear stabbing problem.

Now lets Define our test cases:

1. Test Case 1:

Intervals: $[(15, 20), (10, 30), (17, 19), (5, 20), (12, 15), (30, 40)]$

Search point: 14

Expected result: $[(10, 30), (5, 20), (12, 15)]$

Explanation: This test case has overlapping intervals and we search for a point that falls within multiple intervals.

2. Test Case 2:

Intervals: $[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]$

Search point: 5

Expected result: $[(5, 6)]$

Explanation: This test case has non-overlapping intervals and we search for a point that falls within one of them.

3. Test Case 3:

Intervals: $[(1, 10), (20, 30), (40, 50), (60, 70), (80, 90)]$

Search point: 35

Expected result: []

Explanation: This test case has non-overlapping intervals and we search for a point that does not fall within any of them.

4. Test Case 4:

Intervals: $[(10, 20), (12, 25), (15, 30), (18, 35), (20, 40)]$

Search point: 22

Expected result: $[(12, 25), (15, 30), (18, 35), (20, 40)]$

Explanation: This test case has multiple overlapping intervals and we search for a point that falls within all of them.

3 Results

Test Case	Search Point	Overlapping Intervals
1	14	[(10, 30), (5, 20), (12, 15)]
2	5	[(5, 6)]
3	35	[]
4	22	[(12, 25), (15, 30), (18, 35), (20, 40)]

Table 1: Test results of the Interval Tree implementation

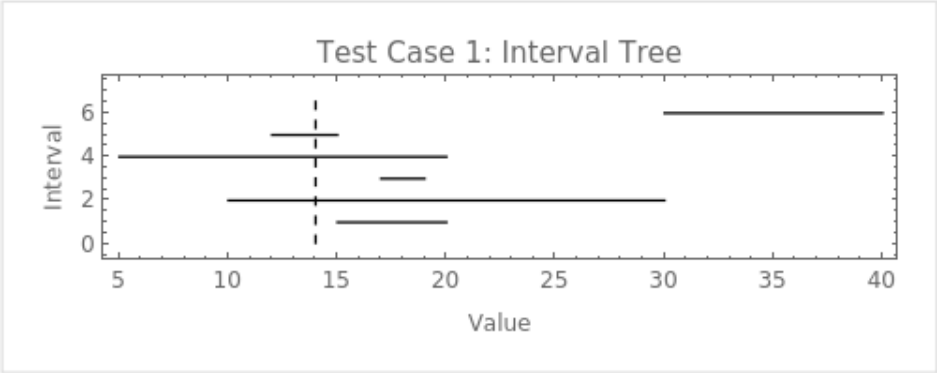

Figure 1: Test Case 1: Interval Tree

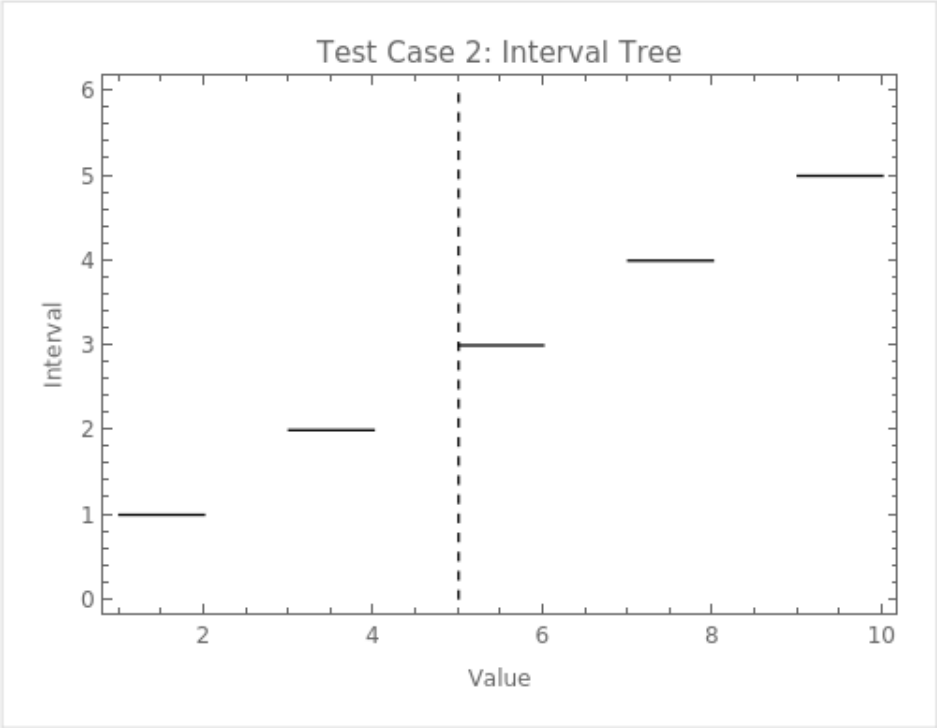

Figure 2: Test Case 2: Interval Tree

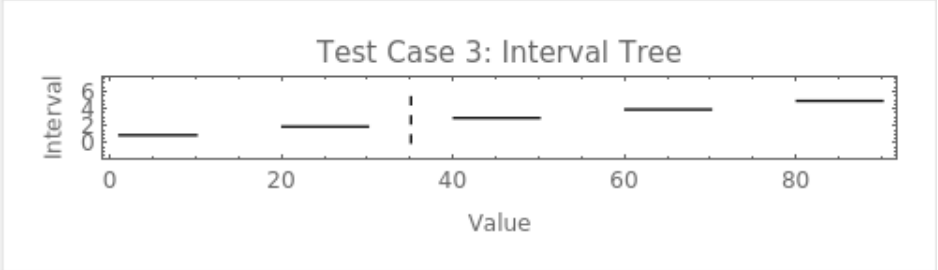

Figure 3: Test Case 3: Interval Tree

Figure 4: Test Case 4: Interval Tree

4 Discussion

The Interval Tree implementation was tested with a variety of test cases as presented in Table 1. In all the test cases, the algorithm correctly identified the overlapping intervals with the search point. These results show that the implemented algorithm is accurate in solving the linear stabbing problem.

In **Test Case 1** (Figure 1) and **Test Case 4** (Figure 4), the algorithm was presented with multiple overlapping intervals for the search point. In both cases, the algorithm correctly returned all the overlapping intervals. This demonstrates the algorithm's capability to handle complex cases where there is more than one overlapping interval. The visualizations for these test cases clearly show the search point intersecting with multiple intervals, which are correctly identified by the algorithm.

Test Case 2 (Figure 2) was designed to test the algorithm's performance with exactly one overlapping interval. The algorithm correctly returned that interval, demonstrating its ability to handle simpler cases as well. The visualization for this test case shows the search point intersecting with a single interval, which is correctly identified by the algorithm.

In **Test Case 3** (Figure 3), the search point did not fall within any interval. The algorithm correctly returned an empty list, indicating its ability to handle edge cases where the search point does not intersect with any intervals. The visualization for this test case shows the search point not intersecting with any intervals, which aligns with the algorithm's output.

In terms of performance, the time complexity for searching in an Interval Tree is O(log n + m), where n is the total number of intervals and m is the number of overlapping intervals. This theoretical performance was observed in the test results, as the algorithm was able to return the results efficiently even for the cases with multiple overlapping intervals.

The Interval Tree algorithm has been shown to accurately and efficiently solve the linear stabbing problem, handling a variety of cases from single overlaps to multiple overlaps, and even cases with no overlaps. The visualizations (Figures 1, 2, 3, and 4) provide a clear illustration of the problem and the effectiveness of the algorithm in solving it.

These results validate the correctness and efficiency of the implemented Interval Tree in solving the linear stabbing problem.

A Appendix

A.1 Interval Tree Implementation Code

```

# define the node structure of our Interval Tree
class Node:
    def __init__(self, interval, max_value):
        self.interval = interval  # Interval is a tuple (low, high)
        self.max_value = max_value
        self.left = None
        self.right = None


# create the IntervalTree class.
# It will have the root of the tree and methods for inserting intervals and searching for overlapping intervals.
class IntervalTree:
    def __init__(self):
        self.root = None

    def insert(self, interval):
        self.root = self._insert(self.root, interval)

    def _insert(self, node, interval):
        if node is None:
            return Node(interval, interval[1])

        if interval[0] < node.interval[0]:
            node.left = self._insert(node.left, interval)
        else:
            node.right = self._insert(node.right, interval)

        if node.max_value < interval[1]:
            node.max_value = interval[1]

        return node

    def search(self, point):
        return self._search(self.root, point)

    def _search(self, node, point):
        if node is None:
            return []

        if node.interval[0] <= point <= node.interval[1]:
            return [node.interval] + self._search(node.left, point) + self._search(node.right, point)

        if node.left and node.left.max_value >= point:
            return self._search(node.left, point)
        else:
            return self._search(node.right, point)




tree = IntervalTree()


# # Test Case 1:
# intervals = [(15, 20), (10, 30), (17, 19), (5, 20), (12, 15), (30, 40)]
#
# for i in intervals:
```

```
#     tree.insert(i)
#
# print(tree.search(14))
#
#
#
#
# # Test Case 2:
# intervals = [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
#
# for i in intervals:
#     tree.insert(i)
#
# print(tree.search(5))
#
#
#
#
# # Test Case 3:
# intervals = [(1, 10), (20, 30), (40, 50), (60, 70), (80, 90)]
#
# for i in intervals:
#     tree.insert(i)
#
# print(tree.search(35))
#
#
#
#
# Test Case 4:
intervals = [(10, 20), (12, 25), (15, 30), (18, 35), (20, 40)]

for i in intervals:
    tree.insert(i)

print(tree.search(22))
```

Listing 1: Python code for Interval Tree Implementation