# Analysis of a Javascript Tagged Interval Tree Implementation and Comparison with Existing Solutions

## 1. Introduction to Interval Trees and Tagging

An interval tree represents a specialized tree data structure designed to efficiently manage and query intervals, which are contiguous segments on a number line. The fundamental purpose of this structure is to enable the rapid retrieval of all intervals that exhibit overlap with a given query interval or a specific point.[1] This capability makes interval trees highly valuable in a diverse range of applications where managing and querying temporal or spatial intervals is critical. Examples of such applications include windowing systems, where it's necessary to determine which elements are visible within a particular viewing area, scheduling algorithms that need to identify overlapping events, and bioinformatics, where interval trees are used to analyze genomic data by finding overlaps between gene regions or other genomic features.[1] The efficiency of interval trees, particularly when dealing with a large number of intervals and frequent queries, offers a significant advantage over more straightforward linear search approaches.[4]

The concept of tagging intervals extends the basic functionality of an interval tree by allowing the association of metadata or labels with specific intervals. This process of tagging serves to categorize intervals, enabling more refined filtering and providing additional context or semantic meaning to the data [Provided Code]. For instance, in a time-based application, different time intervals could be tagged with event types such as "meeting," "vacation," or "deadline." In the realm of genomics, specific regions of a chromosome could be tagged with functional annotations like "gene," "regulatory element," or "non-coding RNA." Similarly, in a mapping application, road segments could be tagged with attributes such as "highway," "residential street," or "toll road." The user-provided code includes a getFormattedText method, which suggests a potential application in text formatting or annotation, where tags might correspond to styling or semantic information applied to specific ranges of text [Provided Code]. By introducing this layer of semantic information, tagged interval trees empower more sophisticated queries and data management strategies beyond simple overlap detection.

This report aims to provide a comprehensive analysis of a Javascript implementation of a tagged interval tree, as presented in the user's code snippet. The analysis will involve a detailed examination of the code's structure and algorithms, followed by a comparative study against other existing Javascript interval tree implementations identified in the provided research material. The key aspects of this comparison will

encompass the underlying data structure, the algorithms employed for core operations such as adding and removing intervals and maintaining the tree's properties, the design of the application programming interface (API), and the specific mechanisms for handling tags. The ultimate goal is to provide insights into the characteristics of the user's implementation within the broader context of available solutions.

**2. Detailed Examination of the Provided Code**

The provided Javascript code defines two primary classes: IntervalNode and TaggedIntervalTree. The IntervalNode class serves as the fundamental unit within the interval tree structure. Its constructor initializes an instance with a start and end point, which together define the interval. It also includes an optional tag parameter, allowing metadata to be directly associated with this specific interval. Additionally, each IntervalNode possesses a children property, which is initialized as an empty array, indicating that nodes in this tree can have multiple child nodes, potentially representing nested or segmented intervals within the parent's range. The toString method of the IntervalNode class is designed to recursively generate a string representation of the node and its entire subtree. This method facilitates easy visualization of the tree structure, using indentation to denote the depth of each node and including the tag associated with the interval, if one exists. This method is particularly useful for debugging and gaining a clear understanding of the tree's current state and organization.

The TaggedIntervalTree class encapsulates the overall structure and functionality of the tagged interval tree. Its constructor initializes the tree with a root node, which is an IntervalNode spanning a default interval from 0 to 0, although this can be customized by providing start and end values during the TaggedIntervalTree instantiation. The class provides several methods for interacting with the tree, the first of which is addTag(tag, interval). This method takes a tag (a label to be associated with the interval) and an interval (defined as an array containing the start and end points) as input. It begins by performing a validation check to ensure the provided interval is valid (i.e., the start point is not greater than or equal to the end point). If the interval is valid, the method proceeds by logging an informational message indicating the tag and interval being processed. It then delegates the core logic of adding the tag to the interval to a recursive helper method named _addTagDFS (likely standing for Depth-First Search), starting from the root of the tree. Following the recursive tagging process, the addTag method calls another internal method, _ensureTreeProperties, which is responsible for maintaining certain invariants or structural properties of the interval tree. This separation of concerns – the recursive

tagging and the subsequent tree maintenance – promotes a more modular and potentially easier-to-understand codebase.

The _addTagDFS(node, tag, start, end) method is central to the process of adding a tag to a specific interval within the tree. This method takes the current node being examined, the tag to be added, and the start and end points of the target interval as arguments. It first adjusts the start and end values to ensure they fall within the interval of the current node, using Math.max to ensure the start is not before the node's start and Math.min to ensure the end is not after the node's end. An early return condition is implemented to handle cases where, after adjustment, the start is no longer less than the end, indicating an invalid interval within the current node's context. The method then checks if the current node already has the same tag. If it does, no further action is needed for this branch of the recursion, and the method returns. If the current node has no children, a new IntervalNode is created with the given tag and the adjusted start and end points, and this new node is added as a child to the current node. However, if the current node does have children, the method proceeds to determine the correct insertion points for the new tagged interval among the existing children. This involves iterating through the node.children and identifying potential gaps or overlaps relative to the start and end of the interval being tagged. The logic checks if a new node needs to be inserted before the first child, between any two consecutive children, or after the last child. For any child whose interval overlaps with the interval being tagged (currentPos < child.interval[1]), the _addTagDFS method is recursively called on that child to propagate the tagging process down the tree. Finally, after identifying all the necessary insertion points, the method creates new IntervalNodes for each of these points, assigning them the provided tag and the corresponding adjusted start and end values. These new nodes are then inserted into the node.children array at the calculated indices using the splice method. The insertion of these new nodes is performed in reverse order to prevent issues with index shifting during the insertion process. This detailed logic suggests that the tree aims to maintain a structure where tagged intervals are appropriately segmented and placed within the hierarchy of nodes.

The TaggedIntervalTree class also includes a removeTag(tag, interval) method, which is present twice in the provided code. This method is designed to remove a specific tag from a given interval. Similar to the addTag method, it first validates the input interval. If the interval is valid, it logs a message indicating the tag and interval being processed. It then calls a recursive helper method, _removeTagDFS, starting from the root of the tree, to handle the actual removal process. After the recursive removal is attempted, the method again calls _ensureTreeProperties to ensure the tree's

structural integrity is maintained. The method returns a boolean value indicating whether a tag was successfully removed. The duplication of this method in the code might indicate an oversight or an area for code refinement.

The _removeTagDFS(node, tag, start, end) method is responsible for the recursive logic of removing a tag from a specified interval within the tree. It begins by adjusting the start and end of the target interval to the boundaries of the current node's interval, resulting in effectiveStart and effectiveEnd. If these effective boundaries indicate no overlap (effectiveStart >= effectiveEnd), the method returns an object containing information about the lack of removal, the remaining interval, and an empty list of nodes to be rehooked. If the current node's tag matches the tag to be removed, the method proceeds to handle four distinct cases based on the relationship between the removal interval and the current node's interval. The first case addresses scenarios where the removal interval is entirely contained within the current node's interval (not touching the start and end). This involves categorizing the current node's children into those before, inside, and after the removal interval. Children overlapping with the removal interval are further processed recursively. New nodes might be created to represent the portions of the original interval before and after the removed segment, and these, along with any remaining or rehooked nodes from the children, are added to a list of nodes to be rehooked. The method then returns an object indicating a successful removal and providing the list of nodes to be rehooked. The second case handles removal intervals that start at or before the current node's start but end within it. Here, the current node's start interval is adjusted to the end of the removal interval, and any affected children are processed recursively. The third case deals with removal intervals that start within the current node's interval but extend to or beyond its end. In this situation, the current node's end interval is adjusted to the start of the removal interval, and affected children are processed recursively. The fourth case occurs when the removal interval completely covers the current node's interval. In this scenario, the method processes the children recursively to remove the tag from them and returns an object indicating a complete node removal along with a list of processed children (or their rehooked nodes). If the current node does not have the tag to be removed, the method iterates through its children. For each child that overlaps with the target interval, the _removeTagDFS method is called recursively. If a child's tag is removed, the parent node's children array is updated by either removing the child entirely and adding any nodes from its rehookNodeList, or by keeping the adjusted child. The method also handles cases where a remaining portion of the removal interval needs to be processed further by making recursive calls with the remainingInterval. Finally, the method ensures that all child intervals are properly nested within the parent's interval. It returns an object indicating whether any tag was

removed and providing the remaining part of the original removal interval that might need further processing.

The _ensureTreeProperties(node) method plays a critical role in maintaining the structural integrity and organizational principles of the tagged interval tree. This method takes a node as input and recursively ensures several key properties throughout the subtree rooted at that node. First, if the node is null or has no children, the method returns without any action. Otherwise, it begins by sorting the children of the current node based on their start positions using a standard array sort. This ensures that the children are ordered along the interval axis. Next, it filters the node.children array to remove any children that represent empty intervals (where the start is not less than the end). If, after filtering, the node has no children remaining, the method returns. The method then proceeds to merge any adjacent or overlapping child nodes that have the same tag. It iterates through the sorted children, comparing each child with the next. If two consecutive children have the same tag and their intervals are adjacent or overlapping (the next child's start is less than or equal to the current child's end), their intervals are merged by updating the end of the current child to be the maximum of their ends, and the children of the next child are appended to the current child's children. The next child is then effectively skipped in the iteration. If the children do not meet these merging criteria, the current child is added to a new mergedChildren array. After iterating through all the children, the node.children array is replaced with this mergedChildren array. Following the merging process, the method recursively calls _ensureTreeProperties on each of the children to ensure these properties are maintained down the tree. It also enforces the "Inside" property, ensuring that the interval of each child is completely contained within the interval of its parent node by adjusting the child's start to be no earlier than the parent's start and the child's end to be no later than the parent's end. Furthermore, the method checks for a specific "Tag" property: a child node cannot have the same tag as its parent. If such a case is found, the grandchildren of the child (those with a different tag than the parent) are moved up to become direct children of the current node, and the child itself is marked for removal. Finally, after the recursive calls and the tag property checks, the node.children array is filtered one last time to remove any children that have the same tag as the parent, and the remaining children are sorted again to ensure the correct order. This comprehensive method ensures that the tree remains well-organized, with non-overlapping (at the same level), sorted, and merged intervals, and that the tagging hierarchy adheres to the defined rules.

The TaggedIntervalTree class also provides a hasTag(tag, interval) method, which allows checking if a given interval has a specific tag associated with it. This method

takes the tag and the interval (as a start and end array) as input and delegates the actual checking process to another recursive helper method, _checkTagDFS, starting from the root of the tree.

The _checkTagDFS(node, tag, start, end) method performs the recursive traversal to determine if a node with the specified tag fully contains the given interval. It first checks if the current node's tag matches the target tag and if the node's interval encompasses the query interval (i.e., the node's start is less than or equal to the query interval's start, and the node's end is greater than or equal to the query interval's end). If both conditions are met, the method returns true, indicating that the tag is present for the given interval. If the current node does not satisfy these conditions, the method iterates through its children. For each child, it checks if there is any overlap between the child's interval and the query interval. If an overlap exists, the _checkTagDFS method is recursively called on that child with the same tag and interval. If any of these recursive calls return true, the current call also returns true. If the loop finishes without finding a match in any of the children, the method returns false. This recursive approach ensures that the entire relevant subtree is searched for an interval that both has the specified tag and fully contains the query interval.

The getFormattedText(text) method demonstrates a specific application of the tagged interval tree. It takes a string of text as input and returns a new string where HTML-like tags have been inserted based on the tagged intervals stored in the tree. This method utilizes an inner function, collectMarkers, which recursively traverses the interval tree. For each node that has a tag, this function records two markers: one for the opening tag at the start of the node's interval and one for the closing tag at the end of the interval. Each marker stores the position, the tag name, and whether it's an opening or closing tag. After the collectMarkers function has traversed the entire tree and collected all the markers, these markers are sorted based on their position. In cases where multiple markers occur at the same position, closing tags are prioritized to appear before opening tags. This sorting is crucial for ensuring correct nesting of the generated HTML-like tags. Finally, the method iterates through the sorted markers. It builds the result string by appending the portion of the original text between the last processed marker and the current marker's position. Then, it appends the appropriate HTML-like tag (e.g., <tag> for an opening marker and </tag> for a closing marker). After processing all the markers, any remaining portion of the original text after the last marker is appended to the result. This method effectively uses the tagged intervals in the tree to add formatting or semantic markup to the original text.

Lastly, the toString() method of the TaggedIntervalTree class simply calls the toString() method of its root node. This provides a straightforward way to obtain a

string representation of the entire tree structure, starting from the root.

### 3. Comparative Analysis with Existing Javascript Interval Tree Implementations

Several Javascript libraries provide interval tree implementations, each with its own set of features and design choices. Comparing the user's tagged interval tree with these existing solutions can highlight its unique aspects and potential strengths or weaknesses.

The static-interval-tree library [4] focuses on efficiently finding overlapping intervals but is designed for a static set of integer intervals that are bulk-loaded and cannot be modified afterwards. It does not support tagging. In contrast, the user's implementation allows for dynamic addition and removal of tagged intervals, making it suitable for scenarios where the set of intervals may change over time and where associating metadata (tags) with these intervals is required.

The rb-interval-tree [5] provides a balanced red-black interval tree that supports dynamic insertion, search for overlaps, and removal of intervals, with the ability to associate a generic value with each interval. While it offers similar dynamic capabilities to the user's code, it uses a standard red-black tree for balancing, whereas the user's implementation employs a custom _ensureTreeProperties method. Furthermore, the user's code has an explicit tag property with specific semantics demonstrated in the getFormattedText method, which is not directly present in rb-interval-tree. Red-black trees offer well-established performance guarantees for core operations.

The intervaltree library by hhamalai [6] is another Javascript implementation that supports querying intervals based on points or ranges and allows associating data with intervals. Similar to rb-interval-tree, it lacks the explicit tag semantics and the specific tree property maintenance rules found in the user's code, such as the merging of same-tagged adjacent intervals and the constraint against parent-child nodes having the same tag.

shinout/interval-tree [7] is an older, unmaintained library that also provides functionality for adding intervals with associated data and searching for overlaps. Its successor, interval-tree2 [8], continues this trend of associating data with intervals identified by an ID. Neither of these libraries appears to have the explicit tagging mechanism or the custom tree property maintenance of the user's implementation.

The interval-tree-type-js library [9] implements a stable interval tree using a red-black augmented tree for self-balancing and allows associating a value with intervals. While

it uses a standard self-balancing mechanism, similar in goal to the user's _ensureTreeProperties, it does not explicitly support tagging as defined in the user's code.

@flatten-js/interval-tree [10] implements an interval binary search tree that stores pairs of <key:interval, value: any> and supports insertion, deletion, and range queries. It provides a general mechanism for associating values with intervals but does not have the specific tagging features of the user's code.

davidisaaclee/interval-tree [11] offers an immutable interval tree with a functional interface, using symmetric augmentation and storing state as plain objects. While it associates data with intervals via an id, it does not have an explicit tag property, and its augmentation likely serves different purposes than the user's custom tree property maintenance.

The interval tree implementation in mgechev/javascript-algorithms [13] focuses on core interval tree operations like adding intervals and checking for point containment or interval intersection, using a max endpoint for augmentation. It does not include explicit tagging functionality.

node-interval-tree [14] is implemented as an augmented AVL tree, where each node maintains a list of records (interval and data). It supports dynamic operations and allows inserting the same interval multiple times with different data. While it provides a flexible way to associate data, it does not have the specific tag property or the custom tree property maintenance rules of the user's code.

| **Feature** | **User's Code** | **static-interval-tree** | **rb-interval-tree** | **intervaltree (hhamalai)** | **shinout/interval-tree** | **interval-tree-type-js** | **@flatten-js/interval-tree** | **davidisaaclee/interval-tree** | **mgechev/javascript-algorithms** | **node-interval-tree** |
|---|---|---|---|---|---|---|---|---|---|---|
| Tagging Support (tag property) | Yes | No | No | No | No | No | No | No | No | No |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Immutability | No | No | No | No | No | No | No | Yes | No | No |
| Self-Balancing Mechanism | Custom | Augmented BST | Red-Black | Implicit | Implicit | Red-Black Augmented | Binary Search Tree | Symmetrically Augmented | Implicit | AVL |
| Point Queries | No | Yes | Yes | Yes | Yes | Yes | No | No | Yes | Yes |
| Range Overlap Queries | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Supports Associated Data/Value | Yes (tag) | Yes (id) | Yes (value) | Yes (data) | Yes (data) | Yes (value) | Yes (value) | Yes (id) | No | Yes (data) |
| Custom Tree Property Maintenance | Yes | No | No | No | No | No | No | No | No | No |
| Supports Merging | Yes | No | No | No | No | No | No | No | No | No |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prevents Same-Tagged Parent-Child | Yes | N/A | No | No | No | No | No | No | No | No |
| Text Formatting Capability | Yes | No | No | No | No | No | No | No | No | No |

The table above summarizes the key features of the user's implementation compared to the other libraries. It highlights that the explicit tagging mechanism and the custom tree property maintenance, including the prevention of same-tagged parent-child nodes and the text formatting capability, appear to be unique or less common features among the reviewed solutions.

**4. Discussion and Key Differences**

The user's tagged interval tree implementation shares fundamental similarities with other interval tree libraries in that it provides a way to store and manage intervals, supports dynamic operations like adding and removing intervals, and offers some form of querying. However, several key differences distinguish it from the existing solutions found in the research material.

One of the most notable differences is the explicit tag property and its semantic use, particularly within the getFormattedText method. This method clearly demonstrates how the tags associated with intervals can drive a specific application, in this case, the formatting of text. While other libraries often allow associating generic data or values with intervals, the user's code provides a dedicated tag property and illustrates a concrete use case where these tags have a defined meaning that influences further processing. This level of explicit tagging and its integration with a formatting function appears to be a unique characteristic or at least a less common focus in the reviewed libraries.

Another significant difference lies in the approach to maintaining the tree's structural

properties. The user's implementation utilizes a custom _ensureTreeProperties method, which enforces a specific set of rules beyond standard self-balancing. These rules include sorting children, merging adjacent or overlapping children with the same tag, ensuring children's intervals are within their parent's, and preventing a node from having the same tag as its parent. This custom maintenance logic contrasts with the use of standard self-balancing tree algorithms like Red-Black trees (in rb-interval-tree and interval-tree-type-js) or AVL trees (in node-interval-tree) found in other implementations. The user's custom rules, especially those related to tagging, suggest a design tailored to specific requirements regarding the organization and hierarchy of tagged intervals.

The logic within the addTag and removeTag methods is also specifically designed to handle the tagging aspect and to maintain the custom tree properties defined in _ensureTreeProperties. This includes the process of finding insertion points based on existing children and the handling of node splitting and rehooking during tag removal, which are likely influenced by the need to preserve the tagging structure and adhere to the custom tree invariants.

Furthermore, the hasTag method in the user's code provides a specific type of query – checking for full interval containment with a specific tag – which might not be directly offered as a primary query type in other libraries that often focus on overlap or point containment queries.

Finally, compared to some more general-purpose interval tree libraries, the user's implementation might potentially lack certain standard interval tree features, such as highly optimized retrieval of all intervals overlapping a given interval or specialized queries for different types of interval relationships like strict envelopment. The focus seems to be more on the tagging and the specific structural rules enforced.

In essence, the user's implementation appears to be a specialized tagged interval tree designed for applications where intervals need to be categorized and manipulated based on tags, with a particular emphasis on maintaining a specific structure and enabling tag-driven processing like text formatting.

**5. Conclusion and Potential Improvements**

The provided Javascript implementation of a tagged interval tree demonstrates several strengths, particularly its explicit tagging feature and its clear application in the getFormattedText method, indicating a focused design for a specific domain, potentially text annotation or processing. The custom _ensureTreeProperties method

allows for the enforcement of specific structural rules related to tagging, such as merging same-tagged intervals and preventing same-tagged parent-child relationships, which are not commonly found in standard interval tree libraries.

Based on the comparative analysis, several potential areas for optimization or further development could be considered.

First, the performance characteristics of the custom _ensureTreeProperties method, especially in scenarios with large trees and frequent updates, could be evaluated. Comparing its efficiency against standard self-balancing tree algorithms like AVL or Red-Black trees might reveal opportunities for improvement in terms of time complexity guarantees for core operations. Leveraging well-established and thoroughly analyzed balancing algorithms could potentially enhance the overall performance and stability of the tree structure.[5]

Second, addressing the code duplication in the removeTag method would improve the code's clarity and maintainability, reducing the risk of errors. Consolidating the logic into a single method would be a beneficial refactoring.

Third, expanding the query capabilities of the tree by adding support for other common interval tree queries, such as efficiently finding all intervals that overlap a given query interval or finding intervals that contain a specific point, could broaden the applicability of the implementation to a wider range of use cases.[4]

Fourth, the addition of comprehensive unit tests would be highly beneficial to ensure the correctness and robustness of all methods, particularly the more complex recursive functions like _addTagDFS and _removeTagDFS, as well as the tree property maintenance logic. Thorough testing helps in identifying and fixing potential bugs and ensures the reliability of the implementation.

Finally, the logic for finding insertion points in the _addTagDFS method could be further examined to ensure optimal efficiency, especially in scenarios with a large number of children per node.

By considering these potential improvements, the user can further refine and enhance their tagged interval tree implementation, potentially leveraging the strengths of standard data structure algorithms while retaining the unique features tailored to their specific application domain.

**Works cited**

1. Interval tree - Wikipedia, accessed April 24, 2025, https://en.wikipedia.org/wiki/Interval_tree
2. Interval Tree using GNU Tree-based container - GeeksforGeeks, accessed April 24, 2025, https://www.geeksforgeeks.org/interval-tree-using-gnu-tree-based-container/
3. Practical Applications of Interval Tree - Stack Overflow, accessed April 24, 2025, https://stackoverflow.com/questions/29653116/practical-applications-of-interval-tree
4. ucscXena/static-interval-tree: Fast overlapping interval ... - GitHub, accessed April 24, 2025, https://github.com/ucscXena/static-interval-tree
5. circlingthesun/rb-interval-tree: Balanced interval tree for ... - GitHub, accessed April 24, 2025, https://github.com/circlingthesun/rb-interval-tree
6. hhamalai/intervaltree: An interval tree implemented in JS - GitHub, accessed April 24, 2025, https://github.com/hhamalai/intervaltree
7. shinout/interval-tree: interval tree in javascript - GitHub, accessed April 24, 2025, https://github.com/shinout/interval-tree
8. shinout/interval-tree2: interval tree in CoffeeScript, available in any JS runtime - GitHub, accessed April 24, 2025, https://github.com/shinout/interval-tree2
9. pineapplemachine/interval-tree-type-js - GitHub, accessed April 24, 2025, https://github.com/pineapplemachine/interval-tree-type-js
10. alexbol99/flatten-interval-tree: Interval binary search tree - GitHub, accessed April 24, 2025, https://github.com/alexbol99/flatten-interval-tree
11. An immutable interval tree in Javascript. - GitHub, accessed April 24, 2025, https://github.com/davidisaaclee/interval-tree
12. Releases · davidisaaclee/interval-tree - GitHub, accessed April 24, 2025, https://github.com/davidisaaclee/interval-tree/releases
13. data-structures/interval-tree.js - Documentation, accessed April 24, 2025, https://mgechev.github.io/javascript-algorithms/data-structures_interval-tree.js.html
14. ShieldBattery/node-interval-tree: An Interval Tree data structure. - GitHub, accessed April 24, 2025, https://github.com/ShieldBattery/node-interval-tree
15. node-interval-tree/index.ts at master - GitHub, accessed April 24, 2025, https://github.com/ShieldBattery/node-interval-tree/blob/master/index.ts
16. Haxe/hxnodejs externs for the node-interval-tree npm library - GitHub, accessed April 24, 2025, https://github.com/proletariatgames/hxnodejs-node-interval-tree
17. node-interval-tree - NPM, accessed April 24, 2025, https://www.npmjs.com/package/node-interval-tree
18. Interval Tree | GeeksforGeeks, accessed April 24, 2025, https://www.geeksforgeeks.org/interval-tree/
19. Introduction to Red-Black Tree | GeeksforGeeks, accessed April 24, 2025, https://www.geeksforgeeks.org/introduction-to-red-black-tree/
20. util/rb_tree: Add augmented trees and interval trees (!22071) · Merge requests · Mesa / mesa, accessed April 24, 2025, https://gitlab.freedesktop.org/mesa/mesa/-/merge_requests/22071
21. chaimleib/intervaltree: A mutable, self-balancing interval tree. Queries may be by

point, by range overlap, or by range containment. - GitHub, accessed April 24, 2025, https://github.com/chaimleib/intervaltree
22. intervaltree.py, accessed April 24, 2025, http://home.cc.umanitoba.ca/~psgendb/doc/local/pkg/ugene/tools/tophat2/intervaltree/intervaltree.py
23. PyIntervalTree · PyPI, accessed April 24, 2025, https://pypi.org/project/PyIntervalTree/