Interval tree

From Wikipedia, the free encyclopedia

In computer science, an **interval tree** is a tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. It is often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene. A similar data structure is the segment tree.

The trivial solution is to visit each interval and test whether it intersects the given point or interval, which requires $\Theta(n)$ time, where n is the number of intervals in the collection. Since a query may return all intervals, for example if the query is a large interval intersecting all intervals in the collection, this is asymptotically optimal; however, we can do better by considering output-sensitive algorithms, where the runtime is expressed in terms of m, the number of intervals produced by the query. Interval trees are dynamic, i.e., they allow insertion and deletion of intervals. They obtain a query time of $O(\log n)$ while the preprocessing time to construct the data structure is $O(n \log n)$ (but the space consumption is $O(n)$). If the endpoints of intervals are within a small integer range (e.g., in the range $[1,...,O(n)]$), faster data structures exist[1] with preprocessing time $O(n)$ and query time $O(1+m)$ for reporting m intervals containing a given query point.[*citation needed*]

Contents [hide]

1 Naive approach
2 Centered interval tree
  2.1 Construction
  2.2 Intersecting
    2.2.1 With a Point
    2.2.2 With an Interval
  2.3 Higher Dimensions
  2.4 Deletion
  2.5 Balancing
3 Augmented tree
  3.1 Java Example: Adding a new interval to the tree
  3.2 Java Example: Searching a point or an interval in the tree
  3.3 Higher dimension
4 Medial- or length-oriented tree
  4.1 Overlap test
  4.2 Adding interval
  4.3 Searching for all overlapping intervals
5 References
6 External links

Naive approach [edit]

In a simple case, the intervals do not overlap and they can be inserted into a simple binary search tree and queried in $O(\log n)$ time. However, with arbitrarily overlapping intervals, there is no way to compare two intervals for insertion into the tree since orderings sorted by the beginning points or the ending points may be different. A naive approach might be to build two parallel trees, one ordered by the beginning point, and one ordered by the ending point of each interval. This allows discarding half of each tree in $O(\log n)$ time, but the results must be merged, requiring $O(n)$ time. This gives us queries in $O(n + \log n) = O(n)$, which is no better than brute-force.

Interval trees solve this problem. This article describes two alternative designs for an interval tree, dubbed the *centered interval tree* and the *augmented tree*.

Centered interval tree [edit]

Queries require $O(\log n + m)$ time, with n being the total number of intervals and m being the number of reported results. Construction requires $O(n \log n)$ time, and storage requires $O(n)$ space.

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Español
فارسی
日本語
Polski
Српски / srpski
Edit links

Construction [edit]

Given a set of *n* intervals on the number line, we want to construct a data structure so that we can efficiently retrieve all intervals overlapping another interval or point.

We start by taking the entire range of all the intervals and dividing it in half at *x_center* (in practice, *x_center* should be picked to keep the tree relatively balanced). This gives three sets of intervals, those completely to the left of *x_center* which we'll call *S_left*, those completely to the right of *x_center* which we'll call *S_right*, and those overlapping *x_center* which we'll call *S_center*.

The intervals in *S_left* and *S_right* are recursively divided in the same manner until there are no intervals left.

The intervals in S_center that overlap the center point are stored in a separate data structure linked to the node in the interval tree. This data structure consists of two lists, one containing all the intervals sorted by their beginning points, and another containing all the intervals sorted by their ending points.

The result is a ternary tree with each node storing:

- A center point
- A pointer to another node containing all intervals completely to the left of the center point
- A pointer to another node containing all intervals completely to the right of the center point
- All intervals overlapping the center point sorted by their beginning point
- All intervals overlapping the center point sorted by their ending point

Intersecting [edit]

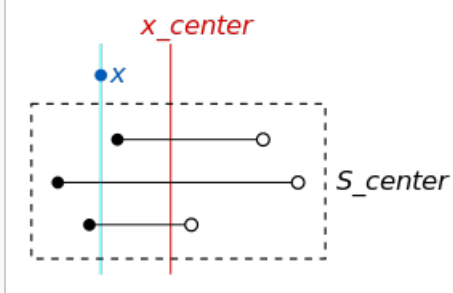
Given the data structure constructed above, we receive queries consisting of ranges or points, and return all the ranges in the original set overlapping this input.

With a Point [edit]

The task is to find all intervals in the tree that overlap a given point *x*. The tree is walked with a similar recursive algorithm as would be used to traverse a traditional binary tree, but with extra affordance for the intervals overlapping the "center" point at each node.

For each tree node, *x* is compared to *x_center*, the midpoint used in node construction above. If *x* is less than *x_center*, the leftmost set of intervals, *S_left*, is considered. If *x* is greater than *x_center*, the rightmost set of intervals, *S_right*, is considered.

As each node is processed as we traverse the tree from the root to a leaf, the ranges in its *S_center* are processed. If *x* is less than *x_center*, we know that all intervals in *S_center* end after *x*, or they could not also overlap *x_center*. Therefore, we need only find those intervals in *S_center* that begin before *x*. We can consult the lists of *S_center* that have already been constructed. Since we only care about the interval beginnings in this scenario, we can consult the list sorted by beginnings. Suppose we find the closest number no greater than *x* in this list. All ranges from the beginning of the list to that found point overlap *x* because they begin before *x* and end after *x* (as we know because they overlap *x_center* which is larger than *x*). Thus, we can simply start enumerating intervals in the list until the startpoint value exceeds *x*.



All intervals in *S_center* that begin before *x* must overlap *x* if *x* is less than *x_center*. Similarly, the same technique also applies in checking a given interval. If a given interval ends at *y* and *y* is less than *x_center*, all intervals in *S_center* that begin before *y* must also overlap the given interval.

Likewise, if *x* is greater than *x_center*, we know that all intervals in *S_center* must begin before *x*, so we find those intervals that end after *x* using the list sorted by interval endings.

If *x* exactly matches *x_center*, all intervals in *S_center* can be added to the results without further processing and tree traversal can be stopped.

With an Interval [edit]

For a result interval r to intersect our query interval q one of the following must hold: the start and/or end point of r is in q; or r completely encloses q.

We first find all intervals with start and/or end points inside q using a separately constructed tree. In the one-dimensional case, we can use a search tree containing all the start and end points in the interval set, each with a pointer to its corresponding interval. A binary search in O(log *n*) time for the start and end of q reveals the

minimum and maximum points to consider. Each point within this range references an interval that overlaps q and is added to the result list. Care must be taken to avoid duplicates, since an interval might both begin and end within q. This can be done using a binary flag on each interval to mark whether or not it has been added to the result set.

Finally, we must find intervals that enclose q. To find these, we pick any point inside q and use the algorithm above to find all intervals intersecting that point (again, being careful to remove duplicates).

Higher Dimensions [edit]

The interval tree data structure can be generalized to a higher dimension *N* with identical query and construction time and O(*n* log *n*) space.

First, a range tree in *N* dimensions is constructed that allows efficient retrieval of all intervals with beginning and end points inside the query region *R*. Once the corresponding ranges are found, the only thing that is left are those ranges that enclose the region in some dimension. To find these overlaps, N interval trees are created, and one axis intersecting *R* is queried for each. For example, in two dimensions, the bottom of the square *R* (or any other horizontal line intersecting R) would be queried against the interval tree constructed for the horizontal axis. Likewise, the left (or any other vertical line intersecting R) would be queried against the interval tree constructed on the vertical axis.

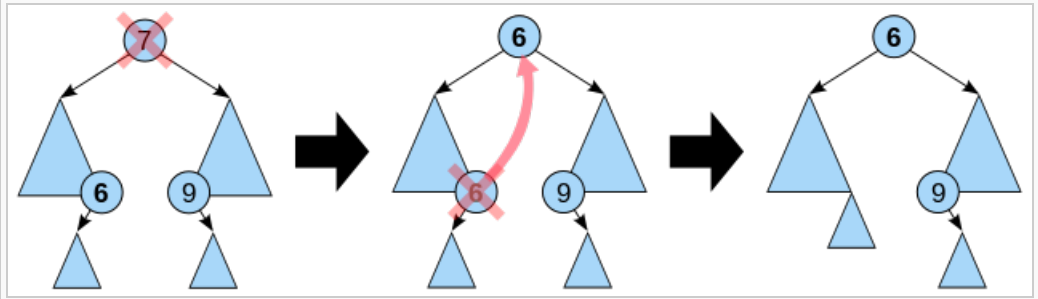
Each interval tree also needs an addition for higher dimensions. At each node we traverse in the tree, *x* is compared with *S_center* to find overlaps. Instead of two sorted lists of points as was used in the one-dimensional case, a range tree is constructed. This allows efficient retrieval of all points in *S_center* that overlap region *R*.

Deletion [edit]

If after deleting an interval from the tree, the node containing that interval contains no more intervals, that node may be deleted from the tree. This is more complex than a normal binary tree deletion operation.

An interval may overlap the center point of several nodes in the tree. Since each node stores the intervals that overlap it, with all intervals completely to the left of its center point in the left subtree, similarly for the right subtree, it follows that each interval is stored in the node closest to the root from the set of nodes whose center point it overlaps.

Normal deletion operations in a binary tree (for the case where the node being deleted has two children) involve promoting a node further from the leaf to the position of the node being deleted (usually the leftmost child of the right subtree, or the rightmost child of the left subtree).



Deleting a node with two children from a binary search tree using the in-order predecessor (rightmost node in the left subtree, labelled **6**).

As a result of this promotion, some nodes that were above the promoted node will become descendents of it; it is necessary to search these nodes for intervals that also overlap the promoted node, and move those intervals into the promoted node. As a consequence, this may result in new empty nodes, which must be deleted, following the same algorithm again.

Balancing [edit]

The same issues that affect deletion also affect rotation operations; rotation must preserve the invariant that intervals are stored as close to the root as possible.

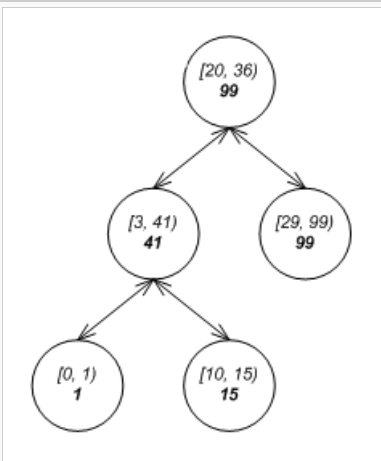
Augmented tree [edit]

Another way to represent intervals is described in Cormen et al. (2009, Section 14.3: Interval trees, pp. 348–354).

Both insertion and deletion require O(log *n*) time, with *n* being the total number of intervals in the tree prior to

the insertion or deletion operation.

Use a simple ordered tree, for example a binary search tree or self-balancing binary search tree, where the tree is ordered by the 'low' values of the intervals, and an extra annotation is added to every node recording the maximum high value among the tree: the node's high value and the high values of both its subtrees. It is simple to maintain this attribute in only O(*h*) steps during each addition or removal of a node, where *h* is the height of the node added or removed in the tree, by updating all ancestors of the node from the bottom up. Additionally, the tree rotations used during insertion and deletion may require updating the high value of the affected nodes.



An augmented tree with low value as the key and maximum high as the extra annotation.

For example, when testing if the given interval *I1=[-5 ,2)* overlaps the intervals in the tree shown above, we can immediately skip the right subtrees of nodes holding intervals *I2=[20, 36)* and *I3=[3, 41)* while traversing the tree. Since *I2*'s low value *20* and *I3*'s low value *3* are past the given intterval *I1*'s end point *2*, all the low values in the right subtrees in question must also be past *I1*'s end point and therefore can be skipped.

Now, it is known that two intervals *A* and *B* overlap only when both *A*.low ≤ *B*.high and *A*.high ≥ *B*.low. When searching the trees for nodes overlapping with a given interval, you can immediately skip:

- all nodes to the right of nodes whose low value is past the end of the given interval.
- all nodes that have their maximum 'high' value below the start of the given interval.

A total order can be defined on the intervals by ordering them first by their 'low' value and finally by their 'high' value. This ordering can be used to prevent duplicate intervals from being inserted into the tree in O(log *n*) time, versus the O(*k* + log *n*) time required to find duplicates if *k* intervals overlap a new interval.

Java Example: Adding a new interval to the tree [edit]

The key of each node is the interval itself, hence nodes are ordered first by low value and finally by high value, and the value of each node is the end point of the interval:

```
public void add(Interval i) {
    put(i, i.getEnd());
}
```

Java Example: Searching a point or an interval in the tree [edit]

To search for an interval, you walk the tree, omitting those branches which can't contain what you're looking for. The simple case is looking for a point:

```
// Search for all intervals which contain "p", starting with the
// node "n" and adding matching intervals to the list "result"
public void search(IntervalNode n, Point p, List<Interval> result) {
    // Don't search nodes that don't exist
    if (n == null)
```

```
            return;

        // If p is to the right of the rightmost point of any interval
        // in this node and all children, there won't be any matches.
        if (p.compareTo(n.getValue()) > 0)
            return;

        // Search left children
        if (n.getLeft() != null)
            search(IntervalNode (n.getLeft()), p, result);

        // Check this node
        if (n.getKey().contains(p))
            result.add(n.getKey());

        // If p is to the left of the start of this interval,
        // then it can't be in any child to the right.
        if (p.compareTo(n.getKey().getStart()) < 0)
            return;

        // Otherwise, search right children
        if (n.getRight() != null)
            search(IntervalNode (n.getRight()), p, result);
    }
```

where

- `a.compareTo(b)` returns a negative value if a < b
- `a.compareTo(b)` returns zero if a = b
- `a.compareTo(b)` returns a positive value if a > b

The code to search for an interval is similar, except for the check in the middle:

```
    // Check this node
    if (n.getKey().overlapsWith(i))
        result.add (n.getKey());
```

`overlapsWith()` is defined as:

```
    public boolean overlapsWith(Interval other) {
        return start.compareTo(other.getEnd()) <= 0 &&
               end.compareTo(other.getStart()) >= 0;
    }
```

Higher dimension [edit]

This can be extended to higher dimensions by cycling through the dimensions at each level of the tree. For example, for two dimensions, the odd levels of the tree might contain ranges for the *x* coordinate, while the even levels contain ranges for the *y* coordinate. However, it is not quite obvious how the rotation logic will have to be extended for such cases to keep the tree balanced.

A much simpler solution is to use nested interval trees. First, create a tree using the ranges for the *y* coordinate. Now, for each node in the tree, add another interval tree on the *x* ranges, for all elements whose *y* range intersect that node's *y* range.

The advantage of this solution is that it can be extended to an arbitrary amount of dimensions using the same code base.

At first, the cost for the additional trees might seem prohibitive but that is usually not the case. As with the solution above, you need one node per *x* coordinate, so this cost is the same in both solutions. The only difference is that you need an additional tree structure per vertical interval. This structure is typically very small (a pointer to the root node plus maybe the number of nodes and the height of the tree).

Medial- or length-oriented tree [edit]

A medial- or length-oriented tree is similar to an augmented tree, but symmetrical, with the binary search tree ordered by the medial points of the intervals. There is a maximum-oriented binary heap in every node, ordered

by the length of the interval (or half of the length). Also we store the minimum and maximum possible value of the subtree in each node (thus the symmetry).

Overlap test [edit]

Using only start and end values of two intervals (a_i, b_i), for $i = 0, 1$, the overlap test can be performed as follows:

$$a_0 \leqslant a_1 < b_0 \quad \text{OR} \quad a_0 < b_1 \leqslant b_0 \quad \text{OR} \quad a_1 \leqslant a_0 < b_1 \quad \text{OR} \quad a_1 < b_0 \leqslant b_1$$

But with defining:

$$m_i = \frac{a_i + b_i}{2}$$

$$d_i = \frac{b_i - a_i}{2}$$

The overlap test is simpler:

$$|m_1 - m_0| < d_0 + d_1$$

Adding interval [edit]

Adding new intervals to the tree is the same as for a binary search tree using the medial value as the key. We push d_i onto the binary heap associated with the node, and update the minimum and maximum possible values associated with all higher nodes.

Searching for all overlapping intervals [edit]

Let's use a_q, b_q, m_q, d_q for the query interval, and M_n for the key of a node (compared to m_i of intervals)

Starting with root node, in each node, first we check if it is possible that our query interval overlaps with the node subtree using minimum and maximum values of node (if it is not, we don't continue for this node).

Then we calculate $\min\{d_i\}$ for intervals inside this node (not its children) to overlap with query interval (knowing $m_i = M_n$):

$$\min\{d_i\} = |m_q - M_n| - d_q$$

and perform a query on its binary heap for the d_i's bigger than $\min\{d_i\}$

Then we pass through both left and right children of the node, doing the same thing. In the worst-case, we have to scan all nodes of the binary search tree, but since binary heap query is optimum, this is acceptable (a 2-dimensional problem can not be optimum in both dimensions)

This algorithm is expected to be faster than a traditional interval tree (augmented tree) for search operations, adding elements is a little slower (the order of growth is the same).

References [edit]

1. ^ Range Queries#Semigroup operators

- Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*, Second Revised Edition. Springer-Verlag 2000. Section 10.1: Interval Trees, pp. 212–217.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009), *Introduction to Algorithms* (3rd ed.), MIT Press and McGraw-Hill, ISBN 978-0-262-03384-8
- Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985
- Jens M. Schmidt. *Interval Stabbing Problems in Small Integer Ranges*. DOI. ISAAC'09, 2009

External links [edit]

- CGAL : Computational Geometry Algorithms Library in C++ contains a robust implementation of Range Trees
- Interval Tree (an augmented self balancing avl tree implementation)
- Interval Tree (a ruby implementation)

v · t · e **Tree data structures** [hide]

Search trees (dynamic sets/associative arrays) 2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B^x · (Optimal) Binary search · Dancing · HTree · **Interval** · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T ·

	Treap · UB · Weight-balanced
Heaps	Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · Van Emde Boas
Tries	Hash · Radix · Suffix · Ternary search · X-fast · Y-fast
Spatial data partitioning trees	BK · BSP · Cartesian · Hilbert R · *k*-d (implicit *k*-d) · M · Metric · MVP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X
Other trees	Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary · Left-child right-sibling · Link/cut · Log-structured merge · Merkle · PQ · Range · SPQR · Top

Categories: Search trees