



WIKIPEDIA
The Free Encyclopedia

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
Български
Español
فارسی
Русский
Српски / srpski
Українська
中文

Edit links

[Create account](#) [Log in](#)

Article [Talk](#)

[Read](#) [Edit](#) [View history](#)

Search

Segment tree

From Wikipedia, the free encyclopedia



This article **relies largely or entirely upon a single source**. Relevant discussion may be found on the [talk page](#). Please help [improve this article](#) by introducing [citations](#) to additional sources. *(November 2007)*

In [computer science](#), a **segment tree** is a [tree data structure](#) for storing [intervals](#), or segments. It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, its structure cannot be modified once it is built. A similar data structure is the [interval tree](#).

A segment tree for a set *I* of *n* intervals uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time. Segment trees support searching for all the intervals that contain a query point in $O(\log n + k)$, *k* being the number of retrieved intervals or segments.^[1]

Applications of the segment tree are in the areas of [computational geometry](#), and [geographic information systems](#).

The segment tree can be generalized to higher [dimension](#) spaces as well.

Contents [\[hide\]](#)

- [1 Structure description](#)
- [2 Storage requirements](#)
- [3 Construction](#)
- [4 Query](#)
- [5 Generalization for higher dimensions](#)
- [6 Notes](#)
- [7 History](#)
- [8 References](#)
- [9 Sources cited](#)

Structure description [\[edit\]](#)

This section describes the structure of a segment tree in a one-dimensional space.

Let *S* be a set of intervals, or segments. Let *p*₁, *p*₂, ..., *p*_{*m*} be the list of distinct interval endpoints, sorted from left to right. Consider the partitioning of the real line induced by those points. The regions of this partitioning are called *elementary intervals*. Thus, the elementary intervals are, from left to right:

$$(-\infty, p_1], [p_1, p_1], (p_1, p_2], [p_2, p_2], \dots, (p_{m-1}, p_m), [p_m, p_m], (p_m, +\infty)$$

That is, the list of elementary intervals consists of open intervals between two consecutive endpoints *p*_{*i*} and *p*_{*i*+1}, alternated with closed intervals consisting of a single endpoint. Single points are treated themselves as intervals because the answer to a query is not necessarily the same at the interior of an elementary interval and its endpoints.^[2]

Given a set *I* of intervals, or segments, a segment tree *T* for *I* is structured as follows:

- T* is a [binary tree](#).
- Its [leaves](#) correspond to the elementary intervals induced by the endpoints in *I*, in an ordered way: the leftmost leaf corresponds to the leftmost interval, and so on. The elementary interval corresponding to a leaf *v* is denoted *Int*(*v*).
- The [internal nodes](#) of *T* correspond to intervals that are the [union](#) of elementary intervals: the interval *Int*(*N*) corresponding to node *N* is the union of the intervals corresponding to the leaves of the tree rooted at *N*. That implies that *Int*(*N*) is the union of the intervals of its two children.
- Each node or leaf *v* in *T* stores the interval *Int*(*v*) and a set of intervals, in some data structure. This canonical subset of node *v* contains the intervals [*x*, *x*] from *I* such that [*x*, *x*] contains *Int*(*v*) and does not contain *Int*(parent(*v*)). That is, each node in *T* stores the segments that span through its interval, but do not span through the interval of its parent.^[3]

Storage requirements [\[edit\]](#)

This section analyzes the storage cost of a segment tree in a one-dimensional space.

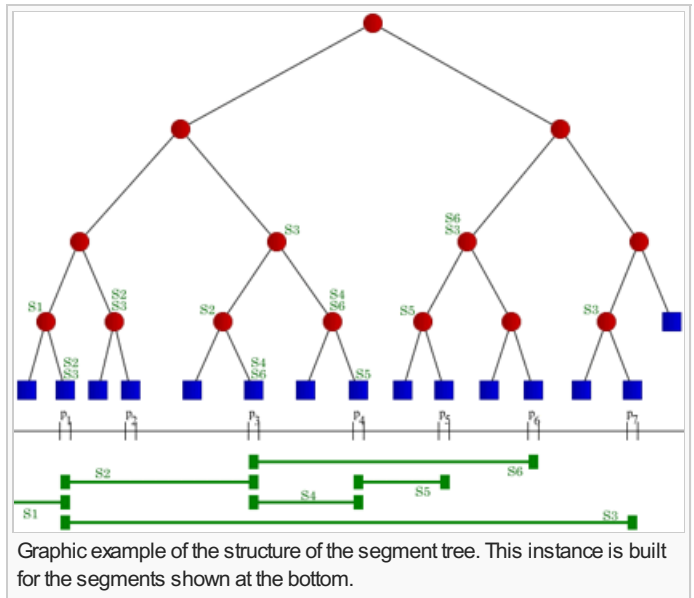
A segment tree T on a set I of n intervals uses $O(n \log n)$ storage.

Proof:

Lemma: Any interval $[x, x']$ of I is stored in the canonical set for at most two nodes at the same depth.

Proof: Let v_1, v_2, v_3 be the three nodes at the same depth, numbered from left to right; and let $p(v)$ be the parent node of any given node v . Suppose $[x, x']$ is stored at v_1 and v_3 . This means that $[x, x']$ spans the whole interval from the left endpoint of $\text{Int}(v_1)$ to the right endpoint of $\text{Int}(v_3)$. Note that all segments at a particular level are non-overlapping and ordered from left to right: this is true by construction for the level containing the leaves, and the property is not lost when moving from any level to the one above it by combining pairs of adjacent segments. Now either $p(v_2) = p(v_1)$, or the former is to the right of the latter (edges in the tree do not cross). In the first case, $\text{Int}(p(v_2))$'s leftmost point is the same as $\text{Int}(v_1)$'s leftmost point; in the second case, $\text{Int}(p(v_2))$'s leftmost point is to the right of $\text{Int}(p(v_1))$'s rightmost point, and therefore also to the right of $\text{Int}(v_1)$'s rightmost point. In both cases, $\text{Int}(p(v_2))$ begins at or to the right of $\text{Int}(v_1)$'s leftmost point. Similar reasoning shows that $\text{Int}(p(v_2))$ ends at or to the left of $\text{Int}(v_3)$'s rightmost point. $\text{Int}(p(v_2))$ must therefore be contained in $[x, x']$; hence, $[x, x']$ will not be stored at v_2 .

The set I has at most $4n + 1$ elementary intervals. Because T is a binary balanced tree with at most $4n + 1$ leaves, its height is $O(\log n)$. Since any interval is stored at most twice at a given depth of the tree, that the total amount of storage is $O(n \log n)$.^[4]



Construction [\[edit\]](#)

This section describes the construction of a segment tree in a one-dimensional space.

A segment tree from the set of segments I , can be built as follows. First, the endpoints of the intervals in I are sorted. The elementary intervals are obtained from that. Then, a balanced binary tree is built on the elementary intervals, and for each node v it is determined the interval $\text{Int}(v)$ it represents. It remains to compute the canonical subsets for the nodes. To achieve this, the intervals in I are inserted one by one into the segment tree. An interval $X = [x, x']$ can be inserted in a subtree rooted at T , using the following procedure:^[5]

- If $\text{Int}(T)$ is contained in X then store X at T , and finish.
- Else:
 - If X intersects the interval of the left child of T , then insert X in that child, recursively.
 - If X intersects the interval of the right child of T , then insert X in that child, recursively.

The complete construction operation takes $O(n \log n)$ time, n being the number of segments in I .

Proof

Sorting the endpoints takes $O(n \log n)$. Building a balanced binary tree from the sorted endpoints, takes linear time on n .

The insertion of an interval $X = [x, x']$ into the tree, costs $O(\log n)$.

Proof: Visiting every node takes constant time (assuming that canonical subsets are stored in a simple data structure like a [linked list](#)). When we visit node v , we either store X at v , or $\text{Int}(v)$ contains an endpoint of X . As proved above, an interval is stored at most twice at each level of the tree. There is also at most one node at every level whose corresponding interval contains x , and one node whose interval contains x' . So, at most four nodes per level are visited. Since there are $O(\log n)$ levels, the total cost of the insertion is $O(\log n)$.^[1]

Query [\[edit\]](#)

This section describes the query operation of a segment tree in a one-dimensional space.

A query for a segment tree, receives a point q_x , and retrieves a list of all the segments stored which contain the point q_x .

Formally stated; given a node (subtree) v and a query point q_x , the query can be done using the following algorithm:

- Report all the intervals in $I(v)$.
- If v is not a leaf:
 - If q_x is in $\text{Int}(\text{left child of } v)$ then
 - Perform a query in the left child of v .
 - If q_x is in $\text{Int}(\text{right child of } v)$ then
 - Perform a query in the right child of v .

In a segment tree that contains n intervals, those containing a given query point can be reported in $O(\log n + k)$ time, where k is the number of reported intervals.

Proof: The query algorithm visits one node per level of the tree, so $O(\log n)$ nodes in total. In the other hand, at a node v , the segments in I are reported in $O(1 + k_v)$ time, where k_v is the number of intervals at node v , reported. The sum of all the k_v for all nodes v visited, is k , the number of reported segments.^[4]

Generalization for higher dimensions ^[edit]

The segment tree can be generalized to higher dimension spaces, in the form of multi-level segment trees. In higher dimension versions, the segment tree stores a collection of axis-parallel (hyper-)rectangles, and can retrieve the rectangles that contain a given query point. The structure uses $O(n \log^d n)$ storage, and answers queries in $O(\log^d n)$.

The use of [fractional cascading](#) lowers the query time bound by a logarithmic factor. The use of the [interval tree](#) on the deepest level of associated structures lowers the storage bound with a logarithmic factor.^[6]

Notes ^[edit]

The query that asks for all the intervals containing a given point, is often referred as *stabbing query*.^[7]

The segment tree is less efficient than the [interval tree](#) for range queries in one dimension, due to its higher storage requirement: $O(n \log n)$ against the $O(n)$ of the interval tree. The importance of the segment tree is that the segments within each node's canonical subset can be stored in any arbitrary manner.^[7]

For n intervals whose endpoints are in a small integer range (e.g., in the range $[1, \dots, O(n)]$), optimal data structures^[which?] exist with a linear preprocessing time and query time $O(1+k)$ for reporting all k intervals containing a given query point.

Another advantage of the segment tree is that it can easily be adapted to counting queries; that is, to report the number of segments containing a given point, instead of reporting the segments themselves. Instead of storing the intervals in the canonical subsets, it can simply store the number of them. Such a segment tree uses linear storage, and requires an $O(\log n)$ query time, so it is optimal.^[8]

A version for higher dimensions of the interval tree and the [priority search tree](#) does not exist, that is, there is no clear extension of these structures that solves the analogous problem in higher dimensions. But the structures can be used as associated structure of segment trees.^[6]

History ^[edit]




This section requires [expansion](#).
(November 2007)

The segment tree was discovered by J. L. Bentley in 1977; in "Solutions to Klee's rectangle problems".^[7]

References ^[edit]

- [^] ^a ^b (de Berg et al. 2000, p. 227)
- [^] (de Berg et al. 2000, p. 224)
- [^] (de Berg et al. 2000, pp. 225–226)
- [^] ^a ^b (de Berg et al. 2000, p. 226)
- [^] (de Berg et al. 2000, pp. 226–227)
- [^] ^a ^b (de Berg et al. 2000, p. 230)
- [^] ^a ^b ^c (de Berg et al. 2000, p. 229)

Sources cited [\[edit\]](#)

- de Berg, Mark; van Kreveld, Marc; Overmars, Mark; Schwarzkopf, Otfried (2000). "More Geometric Data Structures". *Computational Geometry: algorithms and applications* (2nd ed.). Springer-Verlag Berlin Heidelberg New York. doi:10.1007/978-3-540-77974-2 [↗](#). ISBN 3-540-65620-0.
- <http://www.cs.nthu.edu.tw/~wkhon/ds/ds10/tutorial/tutorial6.pdf> 

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 · Ternarysearch · X-fast · Y-fast	
Spatial data partitioning trees	BK · BSP · Cartesian · Hilbert R · <i>k</i> -d (implicit <i>k</i> -d) · M · Metric · MMP · 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 · Mørkle · PQ · Range · SPQR · Top	

Categories: [Trees \(data structures\)](#) | [Binary trees](#) | [Computer graphics data structures](#)