



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  
Deutsch  
فارسی  
Français  
Italiano  
日本語  
Русский  
Српски / srpski  
ไทย  
Türkçe  
Українська  
Tiếng Việt  
中文

Edit links

Create account Log in

Article [Talk](#)

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

# Treap

From Wikipedia, the free encyclopedia  
(Redirected from [Randomized binary search tree](#))

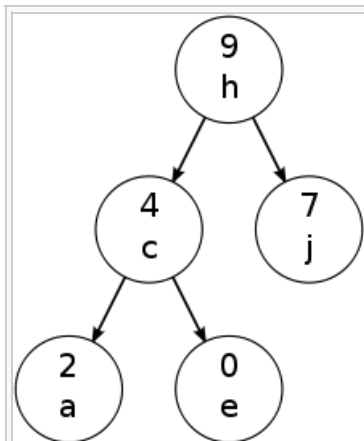
In [computer science](#), the **treap** and the **randomized binary search tree** are two closely related forms of [binary search tree data structures](#) that maintain a dynamic set of ordered keys and allow [binary searches](#) among the keys. After any sequence of insertions and deletions of keys, the shape of the tree is a [random variable](#) with the same probability distribution as a [random binary tree](#); in particular, [with high probability](#) its height is proportional to the [logarithm](#) of the number of keys, so that each search, insertion, or deletion operation takes logarithmic time to perform.

## Contents

[\[hide\]](#)

- Description
- Operations
  - Bulk operations
- Randomized binary search tree
- Comparison
- See also
- References
- External links

## Description [\[edit\]](#)



A treap with alphabetic key and numeric max heap order

The treap was first described by [Cecilia R. Aragon](#) and [Raimund Seidel](#) in 1989;<sup>[1][2]</sup> its name is a [portmanteau](#) of [tree](#) and [heap](#). It is a [Cartesian tree](#) in which each key is given a (randomly chosen) numeric priority. As with any binary search tree, the [inorder traversal](#) order of the nodes is the same as the sorted order of the keys. The structure of the tree is determined by the requirement that it be heap-ordered: that is, the priority number for any non-leaf node must be greater than or equal to the priority of its children. Thus, as with Cartesian trees more generally, the root node is the maximum-priority node, and its left and right subtrees are formed in the same manner from the subsequences of the sorted order to the left and right of that node.

An equivalent way of describing the treap is that it could be formed by inserting the nodes highest-priority-first into a binary search tree without doing any rebalancing. Therefore, if the priorities are independent random numbers (from a distribution over a large enough space of

possible priorities to ensure that two nodes are very unlikely to have the same priority) then the shape of a treap has the same probability distribution as the shape of a [random binary search tree](#), a search tree formed by inserting the nodes without rebalancing in a randomly chosen insertion order. Because random binary search trees are known to have logarithmic height with high probability, the same is true for treaps.

Aragon and Seidel also suggest assigning higher priorities to frequently accessed nodes, for instance by a process that, on each access, chooses a random number and replaces the priority of the node with that number if it is higher than the previous priority. This modification would cause the tree to lose its random shape; instead, frequently accessed nodes would be more likely to be near the root of the tree, causing searches for them to be faster.

Naor and Nissim<sup>[3]</sup> describe an application in maintaining [authorization certificates](#) in [public-key cryptosystems](#).

## Operations [\[edit\]](#)

### Treap

Type Randomized [binary search tree](#)

#### Time complexity in big O notation

	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Part of a series on

### Probabilistic data structures

[Bloom filter](#) · [Count–min sketch](#) · [Quotient filter](#) · [Skip list](#)

#### Random trees

[Random binary tree](#) · [Treap](#) · [Rapidly exploring random tree](#)

#### Related

[Randomized algorithm](#)

[Computer science portal](#)

v · t · e

Treaps support the following basic operations:

- To search for a given key value, apply a standard [binary search algorithm](#) in a binary search tree, ignoring the priorities.
- To insert a new key  $x$  into the treap, generate a random priority  $y$  for  $x$ . Binary search for  $x$  in the tree, and create a new node at the leaf position where the binary search determines a node for  $x$  should exist. Then, as long as  $x$  is not the root of the tree and has a larger priority number than its parent  $z$ , perform a [tree rotation](#) that reverses the parent-child relation between  $x$  and  $z$ .
- To delete a node  $x$  from the treap, if  $x$  is a leaf of the tree, simply remove it. If  $x$  has a single child  $z$ , remove  $x$  from the tree and make  $z$  be the child of the parent of  $x$  (or make  $z$  the root of the tree if  $x$  had no parent). Finally, if  $x$  has two children, swap its position in the tree with the position of its immediate successor  $z$  in the sorted order, resulting in one of the previous cases. In this final case, the swap may violate the heap-ordering property for  $z$ , so additional rotations may need to be performed to restore this property.

### Bulk operations [\[edit\]](#)

In addition to the single-element insert, delete and lookup operations, several fast "bulk" operations have been defined on treaps: [union](#), [intersection](#) and [set difference](#). These rely on two helper operations, *split* and *merge*.

- To split a treap into two smaller treaps, those smaller than key  $x$ , and those larger than key  $x$ , insert  $x$  into the treap with maximum priority—larger than the priority of any node in the treap. After this insertion,  $x$  will be the root node of the treap, all values less than  $x$  will be found in the left subtree, and all values greater than  $x$  will be found in the right subtree. This costs as much as a single insertion into the treap.
- Merging two treaps that are the product of a former split, one can safely assume that the greatest value in the first treap is less than the smallest value in the second treap. Create a new node with value  $x$ , such that  $x$  is larger than this max-value in the first treap, and smaller than the min-value in the second treap, assign it the minimum priority, then set its left child to the first heap and its right child to the second heap. Rotate as necessary to fix the heap order. After that it will be a leaf node, and can easily be deleted. The result is one treap merged from the two original treaps. This is effectively "undoing" a split, and costs the same.

The union of two treaps  $t_1$  and  $t_2$ , representing sets  $A$  and  $B$  is a treap  $t$  that represents  $A \cup B$ . The following recursive algorithm computes the union:

```
function union( $t_1$ ,  $t_2$ ):
    if  $t_1 = \text{nil}$ :
        return  $t_2$ 
    if  $t_2 = \text{nil}$ :
        return  $t_1$ 
    if priority( $t_1$ ) < priority( $t_2$ ):
        swap  $t_1$  and  $t_2$ 
     $t_<$ ,  $t_>$  ← split  $t_2$  on key( $t_1$ )
    return new node(key( $t_1$ ),
                    union(left( $t_1$ ),  $t_<$ ),
                    union(right( $t_1$ ),  $t_>$ ))
```

Here, *split* is presumed to return two trees: one holding the keys less its input key, one holding the greater keys. (The algorithm is [non-destructive](#), but an in-place destructive version exists as well.)

The algorithm for intersection is similar, but requires the *join* helper routine. The complexity of each of union, intersection and difference is  $O(m \log \frac{n}{m})$  for treaps of sizes  $m$  and  $n$ , with  $m \leq n$ . Moreover, since the recursive calls to union are independent of each other, they can be executed [in parallel](#).<sup>[4]</sup>

### Randomized binary search tree [\[edit\]](#)

The randomized binary search tree, introduced by Martínez and Roura subsequently to the work of Aragon and Seidel on treaps,<sup>[5]</sup> stores the same nodes with the same random distribution of tree shape, but maintains different information within the nodes of the tree in order to maintain its randomized structure.

Rather than storing random priorities on each node, the randomized binary search tree stores a small integer at each node, the number of its descendants (counting itself as one); these numbers may be maintained during tree rotation operations at only a constant additional amount of time per rotation. When a key  $x$  is to be inserted into a tree that already has  $n$  nodes, the insertion algorithm chooses with probability  $1/(n + 1)$  to place  $x$  as the new root of the tree, and otherwise it calls the insertion procedure recursively to insert  $x$  within the left or right subtree (depending on whether its key is less than or greater than the root). The numbers of descendants are used by the algorithm to calculate the necessary probabilities for the random choices at each step. Placing  $x$  at

the root of a subtree may be performed either as in the treap by inserting it at a leaf and then rotating it upwards, or by an alternative algorithm described by Martínez and Roura that splits the subtree into two pieces to be used as the left and right children of the new node.

The deletion procedure for a randomized binary search tree uses the same information per node as the insertion procedure, and like the insertion procedure it makes a sequence of  $O(\log n)$  random decisions in order to join the two subtrees descending from the left and right children of the deleted node into a single tree. If the left or right subtree of the node to be deleted is empty, the join operation is trivial; otherwise, the left or right child of the deleted node is selected as the new subtree root with probability proportional to its number of descendants, and the join proceeds recursively.

## Comparison [\[edit\]](#)

The information stored per node in the randomized binary tree is simpler than in a treap (a small integer rather than a high-precision random number), but it makes a greater number of calls to the random number generator ( $O(\log n)$  calls per insertion or deletion rather than one call per insertion) and the insertion procedure is slightly more complicated due to the need to update the numbers of descendants per node. A minor technical difference is that, in a treap, there is a small probability of a collision (two keys getting the same priority), and in both cases there will be statistical differences between a true random number generator and the [pseudo-random number generator](#) typically used on digital computers. However, in any case the differences between the theoretical model of perfect random choices used to design the algorithm and the capabilities of actual random number generators are vanishingly small.

Although the treap and the randomized binary search tree both have the same random distribution of tree shapes after each update, the history of modifications to the trees performed by these two data structures over a sequence of insertion and deletion operations may be different. For instance, in a treap, if the three numbers 1, 2, and 3 are inserted in the order 1, 3, 2, and then the number 2 is deleted, the remaining two nodes will have the same parent-child relationship that they did prior to the insertion of the middle number. In a randomized binary search tree, the tree after the deletion is equally likely to be either of the two possible trees on its two nodes, independently of what the tree looked like prior to the insertion of the middle number.


## See also [\[edit\]](#)

- [Finger search](#)

## References [\[edit\]](#)

- ↑ Aragon, Cecilia R.; Seidel, Raimund (1989), "Randomized Search Trees"  (PDF), *Proc. 30th Symp. Foundations of Computer Science (FOCS 1989)*, Washington, D.C.: IEEE Computer Society Press, pp. 540–545, doi:10.1109/SFCS.1989.63531 [↗](#), ISBN 0-8186-1982-1
- ↑ Seidel, Raimund; Aragon, Cecilia R. (1996), "Randomized Search Trees" [↗](#), *Algorithmica* **16** (4/5): 464–497, doi:10.1007/s004539900061 [↗](#)
- ↑ Nao, M.; Nissim, K. (April 2000), "Certificate revocation and certificate update"  (PDF), *IEEE Journal on Selected Areas in Communications* **18** (4): 561–570, doi:10.1109/49.839932 [↗](#).
- ↑ Blelloch, Guy E.; Reid-Miller, Margaret, (1998), "Fast set operations using treaps", *Proc. 10th ACM Symp. Parallel Algorithms and Architectures (SPAA 1998)*, New York, NY, USA: ACM, pp. 16–26, doi:10.1145/277651.277660 [↗](#), ISBN 0-89791-989-0.
- ↑ Martínez, Conrado; Roura, Salvador (1997), "Randomized binary search trees" [↗](#), *Journal of the ACM* **45** (2): 288–323, doi:10.1145/274787.274812 [↗](#)

## External links [\[edit\]](#)

- [Collection of treap references and info](#) [↗](#) by Cecilia Aragon
- [Open Data Structures - Section 7.2 - Treap: A Randomized Binary Search Tree](#) [↗](#)
- [Treap Applet](#) [↗](#) by Kubo Kovac
- [Animated treap](#) [↗](#)
- [Randomized binary search trees](#) . Lecture notes from a course by Jeff Erickson at UIUC. Despite the title, this is primarily about treaps and [skip lists](#); randomized binary search trees are mentioned only briefly.
- [A high performance key-value store based on treap](#) [↗](#) by Junyi Sun
- [VB6 implementation of treaps](#) [↗](#). Visual basic 6 implementation of treaps as a COM object.
- [ActionScript3 implementation of a treap](#) [↗](#)
- [Pure Python and Cython in-memory treap and duptreap](#) [↗](#)
- [Treaps in C#](#) [↗](#). By Roy Clemmons

- [Pure Go in-memory, immutable treaps](#)
- [Pure Go persistent treap key-value storage library](#)

<span>v · t · e</span>	<b>Tree data structures</b> <span>[hide]</span>
<b>Search trees</b> (dynamic sets/associative arrays)	<span>2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B<sup>X</sup> · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T · Treap · UB · Weight-balanced</span>
<b>Heaps</b>	<span>Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · Van Emde Boas</span>
<b>Tries</b>	<span>Hash · Radix · Suffix · Ternary search · X-fast · Y-fast</span>
<b>Spatial data partitioning trees</b>	<span>BK · BSP · Cartesian · Hilbert R · <i>k</i>-d (implicit <i>k</i>-d) · M · Metric · MVP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X</span>
<b>Other trees</b>	<span>Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary · Left-child right-sibling · Link/cut · Log-structured merge · Merkle · PQ · Range · SPQR · Top</span>

Categories: [Heaps \(data structures\)](#) | [Binary trees](#) | [Probabilistic data structures](#) | [Search trees](#)

This page was last modified on 10 July 2015, at 01:57.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

