Binary tree

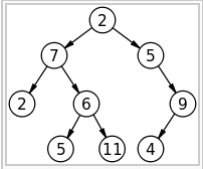
From Wikipedia, the free encyclopedia

Not to be confused with B-tree.

In computer science, a **binary tree** is a tree data structure in which each node has at most two children, which are referred to as the *left* child and the *right* child. A recursive definition using just set theory notions is that a (non-empty) binary tree is a triple (*L*, *S*, *R*), where *L* and *R* are binary trees or the empty set and *S* is a singleton set.[1] Some authors allow the binary tree to be the empty set as well.[2]

From a graph theory perspective, binary (and K-ary) trees as defined here are actually arborescences.[3] A binary tree may thus be also called a **bifurcating arborescence**[3]—a term which actually appears in some very old programming books,[4] before the modern computer science terminology prevailed. It is also possible to interpret a binary tree as an undirected, rather than a directed graph, in which case a binary tree is an ordered, rooted tree.[5] Some authors use **rooted binary tree** instead of *binary tree* to emphasize the fact that the tree is rooted, but as defined above, a binary tree is always rooted.[6] A binary tree is a special case of an ordered K-ary tree, where *k* is 2.

In computing, binary trees are seldom used solely for their structure. Much more typical is to define a labeling function on the nodes, which associates some value to each node.[7] Binary trees labelled this way are used to implement binary search trees and binary heaps, and are used for efficient searching and sorting. The designation of non-root nodes as left or right child even when there is only one child present matters in some of these applications, in particular it is significant in binary search trees.[8] In mathematics, what is termed *binary tree* can vary significantly from author to author. Some use the definition commonly used in computer science,[9] but others define it as every non-leaf having exactly two children and don't necessarily order (as left/right) the children either.[10]



A labeled binary tree of size 9 and height 3, with a root node whose value is 2. The above tree is unbalanced and not sorted.

Contents [hide]

Definitions [edit]

Recursive definition [edit]

This article **may need to be rewritten entirely to comply with Wikipedia's quality standards**, as section. You can help. The discussion page may contain suggestions. *(July 2014)*

Another way of defining a *full* binary tree is a recursive definition. A full binary tree is either:[11]

- A single vertex.
- A graph formed by taking two (full) binary trees, adding a vertex, and adding an edge directed from the new vertex to the root of each binary tree.

This also does not establish the order of children, but does fix a specific root node.

To actually define a binary tree in general, we must allow for the possibility that only one of children may be empty. An artifact, which in some textbooks is called an *extended binary tree* is needed for that purpose. An extended binary tree is thus recursively defined as:[11]

- the empty set is an extended binary tree
- if T_1 and T_2 are extended binary trees, then denote by $T_1 \bullet T_2$ the extended binary tree obtained by adding a root *r* connected to the left to T_1 and to the right to T_2 by adding edges when these sub-trees are non-empty.

Another way of imagining this construction (and understanding the terminology) is to consider instead of the empty set a different type of node—for instance square nodes if the regular ones are circles.[12]

Using graph theory concepts [edit]

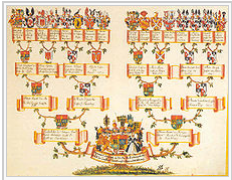
A binary tree is a rooted tree that is also an ordered tree (a.k.a. plane tree) in which every node has at most two children. A rooted tree naturally imparts a notion of levels (distance from the root), thus for every node a notion of children may be defined as the nodes connected to it a level below. Ordering of these children (e.g., by drawing them on a plane) makes possible to distinguish left child from right child.[13] But this still doesn't distinguish between a node with left but not a right child from a one with right but no left child.

The necessary distinction can be made by first partitioning the edges, i.e., defining the binary tree as triplet (V, E_1, E_2), where $(V, E_1 \cup E_2)$ is a rooted tree (equivalently arborescence) and $E_1 \cap E_2$ is empty, and also requiring that for all $j \in \{1, 2\}$ every node has at most one E_j child.[14] A more informal way of making the distinction is to say, quoting the Encyclopedia of Mathematics, that "every node has a left child, a right child, neither, or both" and to specify that these "are all different" binary trees.[9]

Types of binary trees [edit]

Tree terminology is not well-standardized and so varies in the literature.

- A **rooted** binary tree has a root node and every node has at most two children.
- A **full** binary tree (sometimes referred to as a **proper**[*citation needed*] or **plane** binary tree)[15][16] is a tree in which every node in the tree has either 0 or 2 children.
- A **perfect** binary tree is a binary tree in which all interior nodes have two children *and* all leaves have the same *depth* or same *level*.[17] (This is ambiguously also called a **complete** or **full** binary tree.[*citation needed*]) An example of a perfect binary tree is the ancestry chart of a person to a given depth, as each person has exactly two biological parents (one mother and one father).
- In a **complete** binary tree every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes at the last level *h*.[18] An alternative definition is a perfect tree whose rightmost leaves (perhaps all) have been removed. A binary tree is called an **almost complete** binary tree or **nearly complete** binary tree if the last level is not completely filled.[*clarification needed*] A complete binary tree is used as a specialized data structure called a binary heap.[18]
- In the **infinite complete** binary tree, every node has two children (and so the set of levels is countably infinite). The set of all nodes is countably infinite, but the set of all infinite paths from the root is uncountable, having the cardinality of the continuum. These paths correspond by an order-preserving bijection to the points of the Cantor set, or (using the example of a Stern–Brocot tree) to the set of positive irrational numbers.


An ancestry chart which maps to a perfect depth-4 binary tree.

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
العربية
Български
Català
Čeština
Deutsch
Español
Esperanto
فارسی
Français
한국어
हिन्दी
Bahasa Indonesia
Íslenska
Italiano
עברית
Монгол
日本語
Polski
Português
Română
Русский
Shqip
Slovenčina
Slovenščina
Српски / srpski
Suomi
Svenska
ไทย
Українська
中文
Edit links

- A **balanced** binary tree has the minimum possible maximum height (a.k.a. depth) for the leaf nodes, because for any given number of leaf nodes the leaf nodes are placed at the greatest height possible.[*clarification needed*]

```
h      Balanced      Unbalanced, h = (n + 1)/2 - 1
0:      ABCDE               ABCDE
       /     \             /     \
1:    ABCD    E           ABCD    E
     /    \              /    \
2:  AB     CD           ABC     D
   /  \   /  \         /   \
3: A   B  C   D       AB    C
                     /  \
4:                  A    B
```

  One common balanced tree structure is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1.[19] One may also consider binary trees where no leaf is much farther away from the root than any other leaf. (Different balancing schemes allow different definitions of "much farther".[20])

- A **degenerate** (or **pathological**) tree is where each parent node has only one associated child node.[*citation needed*] This means that performance-wise[*clarification needed*], the tree will behave like a linked list data structure.

Properties of binary trees [edit]

- The number of nodes n in a full binary tree, is at least $n = 2(h+1) - 1$ and at most $n = 2^{h+1} - 1$, where h is the height of the tree. A tree consisting of only a root node has a height of 0.
- The number of leaf nodes l in a perfect binary tree, is $l = (n+1)/2$ because the number of non-leaf (a.k.a. internal) nodes $n - l = \sum_{k=0}^{\log_2(l)-1} 2^k = 2^{\log_2(l)} - 1 = l - 1$.
- This means that a perfect binary tree with l leaves has $n = 2l - 1$ nodes.
- In a **balanced** full binary tree, $h = \lceil \log_2(l) \rceil + 1 = \lceil \log_2((n+1)/2) \rceil + 1 = \lceil \log_2(n+1) \rceil$ (see ceiling function).
- In a **perfect** full binary tree, $l = 2^h$ thus $n = 2^{h+1} - 1$.
- The maximum possible number of null links (i.e., absent children of the nodes) in a **complete** binary tree of *n* nodes is (*n*+1), where only 1 node exists in bottom-most level to the far left.
- The number of internal nodes in a **complete** binary tree of *n* nodes is $\lfloor n/2 \rfloor$.
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$.[21]

Combinatorics [edit]

> This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(July 2014)*

In combinatorics one considers the problem of counting the number of full binary trees of a given size. Here the trees have no values attached to their nodes (this would just multiply the number of possible trees by an easily determined factor), and trees are distinguished only by their structure; however the left and right child of any node are distinguished (if they are different trees, then interchanging them will produce a tree distinct from the original one). The size of the tree is taken to be the number *n* of internal nodes (those with two children); the other nodes are leaf nodes and there are *n* + 1 of them. The number of such binary trees of size *n* is equal to the number of ways of fully parenthesizing a string of *n* + 1 symbols (representing leaves) separated by *n* binary operators (representing internal nodes), so as to determine the argument subexpressions of each operator. For instance for *n* = 3 one has to parenthesize a string like $X * X * X * X$, which is possible in five ways:

$$((X*X)*X)*X, \quad (X*(X*X))*X, \quad (X*X)*(X*X), \quad X*((X*X)*X), \quad X*(X*(X*X)).$$

The correspondence to binary trees should be obvious, and the addition of redundant parentheses (around an already parenthesized expression or around the full expression) is disallowed (or at least not counted as producing a new possibility).

There is a unique binary tree of size 0 (consisting of a single leaf), and any other binary tree is characterized by the pair of its left and right children; if these have sizes *i* and *j* respectively, the full tree has size *i* + *j* + 1. Therefore the number C_n of binary trees of size *n* has the following recursive description $C_0 = 1$, and $C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$ for any positive integer *n*. It follows that C_n is the Catalan number of index *n*.

The above parenthesized strings should not be confused with the set of words of length 2*n* in the Dyck language, which consist only of parentheses in such a way that they are properly balanced. The number of such strings satisfies the same recursive description (each Dyck word of length 2*n* is determined by the Dyck subword enclosed by the initial '(' and its matching ')' together with the Dyck subword remaining after that closing parenthesis, whose lengths 2*i* and 2*j* satisfy *i* + *j* + 1 = *n*); this number is therefore also the Catalan number C_n. So there are also five Dyck words of length 10:

$$()()(), \quad ()(()), \quad (())(), \quad (()()), \quad ((()))$$

These Dyck words do not correspond in an obvious way to binary trees. A bijective correspondence can nevertheless be defined as follows: enclose the Dyck word in an extra pair of parentheses, so that the result can be interpreted as a Lisp list expression (with the empty list () as only occurring atom); then the dotted-pair expression for that proper list is a fully parenthesized expression (with NIL as symbol and '.' as operator) describing the corresponding binary tree (which is in fact the internal representation of the proper list).

The ability to represent binary trees as strings of symbols and parentheses implies that binary trees can represent the elements of a free magma on a singleton set.

Methods for storing binary trees [edit]

Binary trees can be constructed from programming language primitives in several ways.

Nodes and references [edit]

In a language with records and references, binary trees are typically constructed by having a tree node structure which contains some data and references to its left child and its right child. Sometimes it also contains a reference to its unique parent. If a node has fewer than two children, some of the child pointers may be set to a special null value, or to a special sentinel node.

This method of storing binary trees wastes a fair bit of memory, as the pointers will be null (or point to the sentinel) more than half the time; a more conservative representation alternative is threaded binary tree.[22]

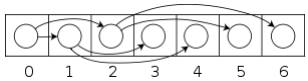
In languages with tagged unions such as ML, a tree node is often a tagged union of two types of nodes, one of which is a 3-tuple of data, left child, and right child, and the other of which is a "leaf" node, which contains no data and functions much like the null value in a language with pointers. For example, the following line of code in OCaml (an ML dialect) defines a binary tree that stores a character in each node.[23]

```
type chr_tree = Empty | Node of char * chr_tree * chr_tree
```

Arrays [edit]

Binary trees can also be stored in breadth-first order as an implicit data structure in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index *i*, its children are found at indices $2i + 1$ (for the left child) and $2i + 2$ (for the right), while its parent (if any) is found at index $\left\lfloor \frac{i-1}{2} \right\rfloor$ (assuming the root has index zero). This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it is expensive to grow and wastes space proportional to $2^h - n$ for a tree of depth *h* with *n* nodes.

This method of storage is often used for binary heaps. No space is wasted because nodes are added in breadth-first order.



Encodings [edit]

Succinct encodings [edit]

A succinct data structure is one which occupies close to minimum possible space, as established by information theoretical lower bounds. The number of different binary trees on n nodes is C_n, the

nth Catalan number (assuming we view trees with identical *structure* as identical). For large n, this is about 4^n; thus we need at least about $\log_2 4^n = 2n$ bits to encode it. A succinct binary tree therefore would occupy $2n + o(n)$ bits.

One simple representation which meets this bound is to visit the nodes of the tree in preorder, outputting "1" for an internal node and "0" for a leaf. [1] If the tree contains data, we can simply simultaneously store it in a consecutive array in preorder. This function accomplishes this:

```
function EncodeSuccinct(node n, bitstring structure, array data) {
    if n = nil then
        append 0 to structure;
    else
        append 1 to structure;
        append n.data to data;
        EncodeSuccinct(n.left, structure, data);
        EncodeSuccinct(n.right, structure, data);
}
```

The string *structure* has only $2n + 1$ bits in the end, where n is the number of (internal) nodes; we don't even have to store its length. To show that no information is lost, we can convert the output back to the original tree like this:

```
function DecodeSuccinct(bitstring structure, array data) {
    remove first bit of structure and put it in b
    if b = 1 then
        create a new node n
        remove first element of data and put it in n.data
        n.left = DecodeSuccinct(structure, data)
        n.right = DecodeSuccinct(structure, data)
        return n
    else
        return nil
}
```

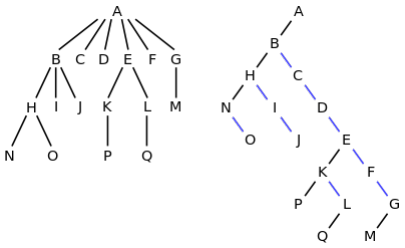
More sophisticated succinct representations allow not only compact storage of trees but even useful operations on those trees directly while they're still in their succinct form.

Encoding general trees as binary trees [edit]

There is a one-to-one mapping between general ordered trees and binary trees, which in particular is used by Lisp to represent general ordered trees as binary trees. To convert a general ordered tree to binary tree, we only need to represent the general tree in left child-right sibling way. The result of this representation will be automatically binary tree, if viewed from a different perspective. Each node *N* in the ordered tree corresponds to a node *N'* in the binary tree; the *left* child of *N'* is the node corresponding to the first child of *N*, and the *right* child of *N'* is the node corresponding to *N* 's next sibling --- that is, the next node in order among the children of the parent of *N*. This binary tree representation of a general order tree is sometimes also referred to as a left child-right sibling binary tree (LCRS tree), or a doubly chained tree, or a Filial-Heir chain.

One way of thinking about this is that each node's children are in a linked list, chained together with their *right* fields, and the node only has a pointer to the beginning or head of this list, through its *left* field.

For example, in the tree on the left, A has the 6 children {B,C,D,E,F,G}. It can be converted into the binary tree on the right.



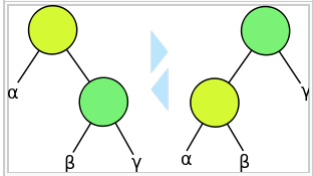
The binary tree can be thought of as the original tree tilted sideways, with the black left edges representing *first child* and the blue right edges representing *next sibling*. The leaves of the tree on the left would be written in Lisp as:

(((N O) I J) C D ((P) (Q)) F (M))

which would be implemented in memory as the binary tree on the right, without any letters on those nodes that have a left child.

Common operations [edit]

There are a variety of different operations that can be performed on binary trees. Some are mutator operations, while others simply return useful information about the tree.



Tree rotations are very common internal operations on self-balancing binary trees.

Insertion [edit]

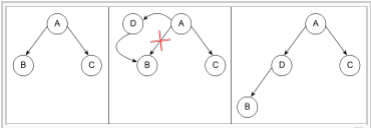
Nodes can be inserted into binary trees in between two other nodes or added after a leaf node. In binary trees, a node that is inserted is specified as to which child it is.

Leaf nodes [edit]

To add a new node after leaf node A, A assigns the new node as one of its children and the new node assigns node A as its parent.

Internal nodes [edit]

Insertion on internal nodes is slightly more complex than on leaf nodes. Say that the internal node is node A and that node B is the child of A. (If the insertion is to insert a right child, then B is the right child of A, and similarly with a left child insertion.) A assigns its child to the new node and the new node assigns its parent to A. Then the new node assigns its child to B and B assigns its parent as the new node.


The process of inserting a node into a binary tree

Deletion [edit]

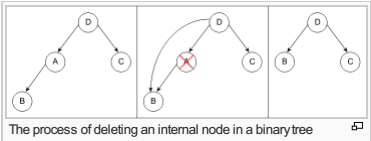
Deletion is the process whereby a node is removed from the tree. Only certain nodes in a binary tree can be removed unambiguously.[24]

Node with zero or one children [edit]

Suppose that the node to delete is node A. If A has no children, deletion is accomplished by setting the child of A's parent to null. If A has one child, set the parent of A's child to A's parent and set the child of A's parent to A's child.

Node with two children [edit]

In a binary tree, a node with two children cannot be deleted unambiguously.[24] However, in certain binary trees (including binary search trees) these nodes *can* be deleted, though with a rearrangement of the tree structure.


The process of deleting an internal node in a binary tree

Traversal [edit]

Main article: Tree traversal

Pre-order, in-order, and post-order traversal visit each node in a tree by recursively visiting each node in the left and right subtrees of the root.

Depth-first order [edit]

In depth-first order, we always attempt to visit the node farthest from the root node that we can, but with the caveat that it must be a child of a node we have already visited. Unlike a depth-first search on graphs, there is no need to remember all the nodes we have visited, because a tree cannot contain cycles. Pre-order is a special case of this. See depth-first search for more information.

Breadth-first order [edit]

Contrasting with depth-first order is breadth-first order, which always attempts to visit the node closest to the root that it has not already visited. See breadth-first search for more information. Also called a *level-order traversal*.

In a complete binary tree, a node's breadth-index ($i - (2^d - 1)$) can be used as traversal instructions from the root. Reading bitwise from left to right, starting at bit $d - 1$, where d is the node's distance from the root ($d = floor(log2(i+1))$) and the node in question is not the root itself ($d > 0$). When the breadth-index is masked at bit $d - 1$, the bit values `0` and `1` mean to step either left or right, respectively. The process continues by successively checking the next bit to the right until there are no more. The rightmost bit indicates the final traversal from the desired node's parent to the node itself. There is a time-space trade-off between iterating a complete binary tree this way versus each node having pointer/s to its sibling/s.

See also [edit]

- 2–3 tree
- 2–3–4 tree
- AA tree
- Ahnentafel
- AVL tree
- B-tree
- Binary space partitioning
- Huffman tree
- K-ary tree
- Kraft's inequality
- Optimal binary search tree
- Random binary tree
- Recursion (computer science)
- Red–black tree
- Rope (computer science)
- Self-balancing binary search tree
- Splay tree
- Strahler number
- Tree of primitive Pythagorean triples#Alternative methods of generating the tree

References [edit]

Citations [edit]

1. ^ Rowan Garnier; John Taylor (2009). *Discrete Mathematics: Proofs, Structures and Applications, Third Edition*. CRC Press. p. 620. ISBN 978-1-4398-1280-8.
2. ^ Steven S Skiena (2009). *The Algorithm Design Manual*. Springer Science & Business Media. p. 77. ISBN 978-1-84800-070-4.
3. ^ [a] [b] Knuth (1997). *The Art Of Computer Programming, Volume 1, 3/E*. Pearson Education. p. 363. ISBN 0-201-89683-4.
4. ^ Iván Flores (1971). *Computer programming system/360*. Prentice-Hall. p. 39.
5. ^ Kenneth Rosen (2011). *Discrete Mathematics and Its Applications, 7th edition*. McGraw-Hill Science. p. 749. ISBN 978-0-07-338309-5.
6. ^ David R. Mazur (2010). *Combinatorics: A Guided Tour*. Mathematical Association of America. p. 246. ISBN 978-0-88385-762-5.
7. ^ David Makinson (2009). *Sets, Logic and Maths for Computing*. Springer Science & Business Media. p. 199. ISBN 978-1-84628-845-6.
8. ^ Jonathan L. Gross (2007). *Combinatorial Methods with Computer Applications*. CRC Press. p. 248. ISBN 978-1-58488-743-0.
9. ^ [a] [b] Hazewinkel, Michiel, ed. (2001), "Binary tree", *Encyclopedia of Mathematics*, Springer, ISBN 978-1-55608-010-4 also in print as Michiel Hazewinkel (1997). *Encyclopaedia of Mathematics. Supplement I*. Springer Science & Business Media. p. 124. ISBN 978-0-7923-4709-5.
10. ^ L.R. Foulds (1992). *Graph Theory Applications*. Springer Science & Business Media. p. 32. ISBN 978-0-387-97599-3.
11. ^ [a] [b] Kenneth Rosen (2011). *Discrete Mathematics and Its Applications 7th edition*. McGraw-Hill Science. pp. 352–353. ISBN 978-0-07-338309-5.
12. ^ Te Chiang Hu; Man-tak Shing (2002). *Combinatorial Algorithms*. Courier Dover Publications. p. 162. ISBN 978-0-486-41962-6.
13. ^ Lih-Hsing Hsu; Cheng-Kuan Lin (2008). *Graph Theory and Interconnection Networks*. CRC Press. p. 66. ISBN 978-1-4200-4482-9.
14. ^ J. Flum; M. Grohe (2006). *Parameterized Complexity Theory*. Springer. p. 245. ISBN 978-3-540-29953-0.
15. ^ "full binary tree". NIST.
16. ^ Richard Stanley, Enumerative Combinatorics, volume 2, p.36
17. ^ "perfect binary tree". NIST.
18. ^ [a] [b] "complete binary tree". NIST.
19. ^ Aaron M. Tenenbaum, et al. Data Structures Using C, Prentice Hall, 1990 ISBN 0-13-199746-7
20. ^ Paul E. Black (ed.), entry for *data structure* in *Dictionary of Algorithms and Data Structures*. *U.S. National Institute of Standards and Technology*. *15 December 2004*. *Online version* *Accessed 2010-12-19*.
21. ^ Mehta, Dinesh; Sartaj Sahni (2004). *Handbook of Data Structures and Applications*. Chapman and Hall. ISBN 1-58488-435-5.
22. ^ D. Samanta (2004). *Classic Data Structures*. PHI Learning Pvt. Ltd. pp. 264–265. ISBN 978-81-203-1874-8.
23. ^ Michael L. Scott (2009). *Programming Language Pragmatics* (3rd ed.). Morgan Kaufmann. p. 347. ISBN 978-0-08-092299-7.
24. ^ [a] [b] Dung X. Nguyen (2003). "Binary Tree Structure". rice.edu. Retrieved December 28, 2010.

Bibliography [edit]

- Donald Knuth. *The Art of Computer Programming vol 1. Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.3, especially subsections 2.3.1–2.3.2 (pp. 318–348).

External links [edit]

- binary trees entry in the FindStat database
- Gamedev.net introduction on binary trees
- Binary Tree Proof by Induction
- Balanced binary search tree on array How to create bottom-up an Ahnentafel list, or a balanced binary search tree on array

Wikimedia Commons has media related to ***Binary trees***.

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: Binary trees | Data structures