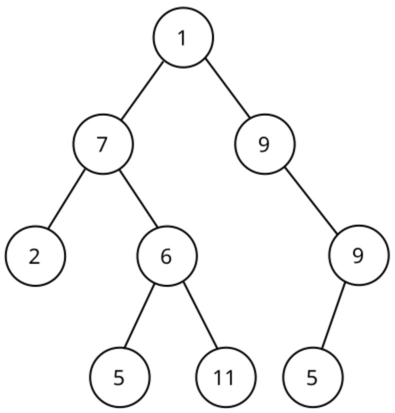
Binary tree

In computer science, a **binary tree** is a tree data structure in which each node has at most two children, referred to as the *left child* and the *right child*. That is, it is a k-ary tree with $k = 2$. A recursive definition using set theory is that a binary tree is a tuple (L, S, R), where L and R are binary trees or the empty set and S is a singleton set containing the root.[1][2]



A labeled binary tree of size 9 (the number of nodes in the tree) and height 3 (the height of a tree defined as the number of edges or links from the top-most or root node to the farthest leaf node), with a root node whose value is 1. The above tree is unbalanced and not sorted.

From a graph theory perspective, binary trees as defined here are arborescences.[3] A binary tree may thus be also called a **bifurcating arborescence**,[3] a term which appears in some early programming books[4] before the modern computer science terminology prevailed. It is also possible to interpret a binary tree as an undirected, rather than 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]

In mathematics, what is termed *binary tree* can vary significantly from author to author. Some use the definition commonly used in computer science,[7] but others define it as every non-leaf having exactly two children and don't necessarily label the children as left and right either.[8]

In computing, binary trees can be used in two very different ways:

- First, as a means of accessing nodes based on some value or label associated with each node.[9] 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.[10] However, the arrangement of particular nodes into the tree is not part of the conceptual information. For example, in a normal binary search tree the placement of nodes depends almost entirely on the order in which they were added, and can be re-arranged (for example by balancing) without changing the meaning.
- Second, as a representation of data with a relevant bifurcating structure. In such cases, the particular arrangement of nodes under and/or to the left or right of other nodes is part of the information (that is, changing it would change the meaning). Common examples occur with Huffman coding and cladograms. The everyday division of documents into chapters, sections, paragraphs, and so on is an analogous example with *n*-ary rather than binary trees.

Definitions

Recursive definition

To define a binary tree, the possibility that only one of the children may be empty must be acknowledged. 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 $T_1 \cdot T_2$ 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

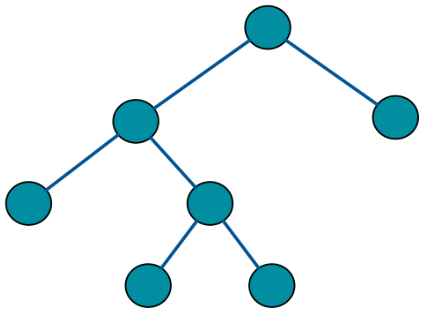
A binary tree is a rooted tree that is also an ordered tree (a.k.a. plane tree) in which every node has at most two children. A rooted tree naturally imparts a notion of levels (distance from the root); thus, for every node, a notion of children may be defined as the nodes connected to it a level below.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

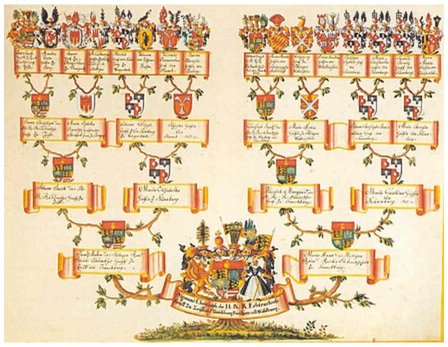
The necessary distinction can be made by first partitioning the edges; i.e., defining the binary tree as triplet (V, E_1, E_2), where $(V, E_1 \cup E_2)$ is a rooted tree (equivalently arborescence) and $E_1 \cap E_2$ is empty, and also requiring that for all $j \in \{ 1, 2 \}$, every node has at most 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.[7]

Types of binary trees

Tree terminology is not well-standardized and therefore may vary among examples in the available literature.

- A **rooted** binary tree has a root node and every node has at most two children.


A full binary tree

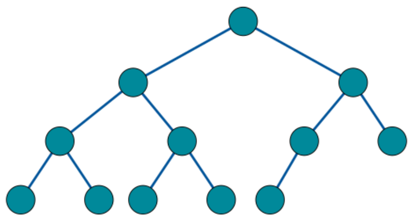

An ancestry chart which can be mapped to a perfect 4-level binary tree.

- A **full** binary tree (sometimes referred to as a **proper**,[15] **plane**, or **strict** binary tree)[16][17] is a tree in which every node has either 0 or 2 children. Another way of defining a full binary tree is a recursive definition. A full binary tree is either:[11]
    - A single vertex (a single node as the root node).

- A tree whose root node has two subtrees, both of which are full binary trees.

- 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* (the level of a node defined as the number of edges or links from the root node to a node).[18] A perfect binary tree is a full binary tree.

- A **complete** binary tree is a binary tree in which 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.[19] A perfect tree is therefore always complete but a complete tree is not always perfect. Some authors use the term **complete** to refer instead to a **perfect** binary tree as defined above, in which case they call this type of tree (with a possibly not filled last level) an **almost complete** binary tree or **nearly complete** binary tree.[20][21] A complete binary tree can be efficiently represented using an array.[19]



A complete binary tree (that is not full)

- The **infinite complete** binary tree is a tree with $\aleph_0$ levels, where for each level d the number of existing nodes at level d is equal to 2^d. The cardinal number of the set of all levels is $\aleph_0$ (countably infinite). The cardinal number of the set of all paths (the "leaves", so to speak) is uncountable, having the cardinality of the continuum.

- A **balanced** binary tree is a binary tree structure in which the left and right subtrees of every node differ in height (the number of edges from the top-most node to the farthest node in a subtree) by no more than 1 (or the skew is no greater than 1).[22] 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".[23])

- A **degenerate** (or **pathological**) tree is where each parent node has only one associated child node.[24] This means that the tree will behave like a linked list data structure. In this case, an advantage of using a binary tree is significantly reduced because it is essentially a linked list which time complexity is O(n) (n as the number of nodes) and it has more data space than the linked list due to two pointers per node, while the complexity of O($\log_2 n$) for data search in a balanced binary tree is normally expected.

Properties of binary trees

- The number of nodes n in a **full** binary tree is at least $2h+1$ and at most $2^{h+1}-1$ (i.e., the number of nodes in a **perfect** binary tree), where h is the height of the tree. A tree consisting of only a root node has a height of 0. The least number of nodes is obtained by adding only two children nodes per adding height so $2h+1$ (1 for counting the root node). The maximum number of nodes is obtained by fully filling nodes at each level, i.e., it is a perfect tree. For a perfect tree, the number of nodes is $1+2+4+\ldots+2^h=2^{h+1}-1$, where the last equality is from the geometric series sum.

- The number of leaf nodes l in a **perfect** binary tree is $l=(n+1)/2$ (where n is the number of nodes in the tree) because $n=2^{h+1}-1$ (by using the above property) and the number of leaves is 2^h so $n=2\cdot 2^h-1=2l-1\rightarrow l=(n+1)/2$. It also means that $n=2l-1$. In terms of the tree height h, $l=(2^{h+1}-1+1)/2=2^h$.

- For any non-empty binary tree with l leaf nodes and i_2 nodes of degree 2 (internal nodes with two child nodes), $l=i_2+1$.[25] The proof is the following. For a perfect binary tree, the total number of nodes is $n=2^{h+1}-1$ (A perfect binary tree is a full binary tree.) and $l=2^h$, so $i=n-l=(2^{h+1}-1)-2^h=2^h-1=l-1\rightarrow l=i+1$. To make a full binary tree from a perfect binary tree, a pair of two sibling nodes are removed one by one. This results in "two leaf nodes removed" and "one internal node removed" and "the removed internal node becoming a leaf node", so one leaf node and one internal node is removed per removing two sibling nodes. As a result, $l=i+1$ also holds for a full binary tree. To make a binary tree with a leaf node without its sibling, a single leaf node is removed from a full binary tree, then "one leaf node removed" and "one internal nodes with two children removed" so $l=i+1$ also holds. This relation now covers all non-empty binary trees.

- With given n nodes, the minimum possible tree height is $h_{\min}=\log_2(n+1)-1$ with which the tree is a balanced full tree or perfect tree. With a given height h, the number of nodes can't exceed the $2^{h+1}-1$ as the number of nodes in a perfect tree. Thus $n\leq 2^{h+1}-1\rightarrow h\geq\log_2(n+1)-1$.

- A binary Tree with l leaves has at least the height $h_m=\log_2(l)$. With a given height h, the number of leaves at that height can't exceed 2^h as the number of leaves at the height in a perfect tree. Thus $l\leq 2^h\rightarrow h\geq\log_2(l)$.

- In a non-empty binary tree, if n is the total number of nodes and e is the total number of edges, then $e=n-1$. This is obvious because each node requires one edge except for the root node.

- The number of null links (i.e., absent children of the nodes) in a binary tree of n nodes is $(n + 1)$.
- The number of internal nodes in a **complete** binary tree of n nodes is $\lfloor n/2 \rfloor$.

Combinatorics

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), 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:

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

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 6:

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

These Dyck words do not correspond to binary trees in the same way. Instead, they are related by the following recursively defined bijection: the Dyck word equal to the empty string corresponds to the binary tree of size 0 with only one leaf. Any other Dyck word can be written as $(w_1)w_2$, where w_1,w_2 are themselves (possibly empty) Dyck words and where the two written parentheses are matched. The bijection is then defined by letting the words w_1 and w_2 correspond to the binary trees that are the left and right children of the root.

A bijective correspondence can also 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

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

Nodes and references

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.[26]

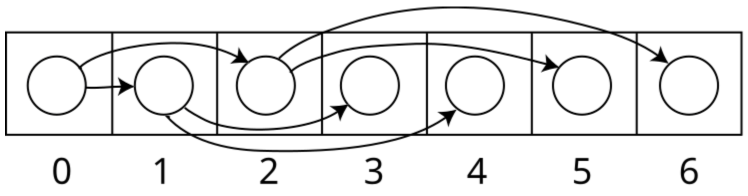
In languages with tagged unions such as ML, a tree node is often a tagged union of two types of nodes, one of which is a 3-tuple of data, left child, and right child, and the other of which is a "leaf" node, which contains no data and functions much like the null value in a language with pointers. For example, the following line of code in OCaml (an ML dialect) defines a binary tree that stores a character in each node.[27]

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

Arrays

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). Alternatively, with a 1-indexed array, the implementation is simplified with children found at $2i$ and $2i+1$, and parent found at $\lfloor i/2 \rfloor$.[28]

This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. It is often used for binary heaps.[29]



Encodings

Succinct encodings

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.[30] 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
}
```

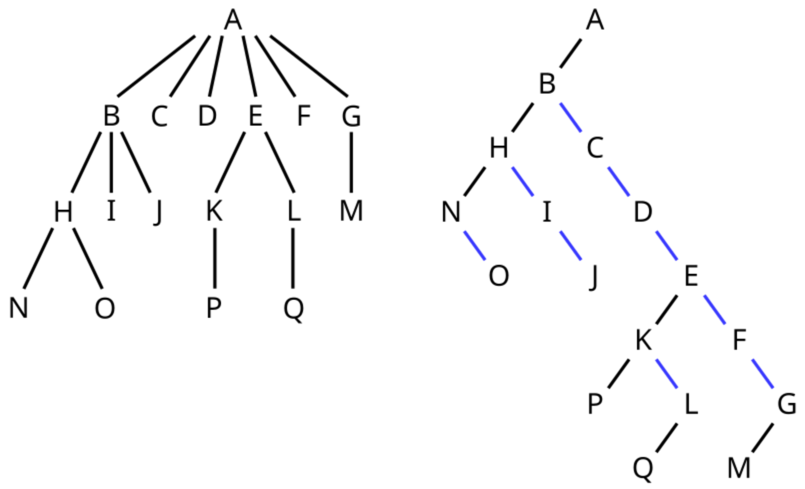
More sophisticated succinct representations allow not only compact storage of trees but even useful operations on those trees directly while they're still in their succinct form.

Encoding ordered trees as binary trees

There is a natural one-to-one correspondence between ordered trees and binary trees. It allows any ordered tree to be uniquely represented as a binary tree, and vice versa:

Let *T* be a node of an ordered tree, and let *B* denote *T's* image in the corresponding binary tree. Then *B's left* child represents *T's* first child, while the *B's right* child represents *T'*s next sibling.

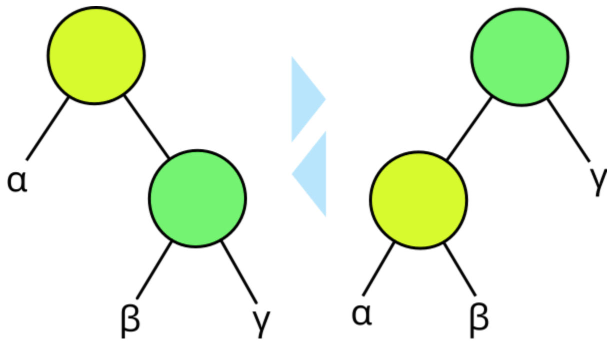
For example, the ordered tree on the left and the binary tree on the right correspond:



In the pictured binary tree, the black, left, edges represent *first child*, while the blue, right, edges represent *next sibling*.

This representation is called a left-child right-sibling binary tree.

Common operations



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

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.

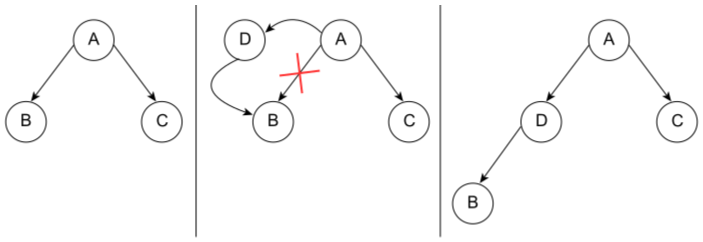
Insertion

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 whose child it will be.

Leaf nodes

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



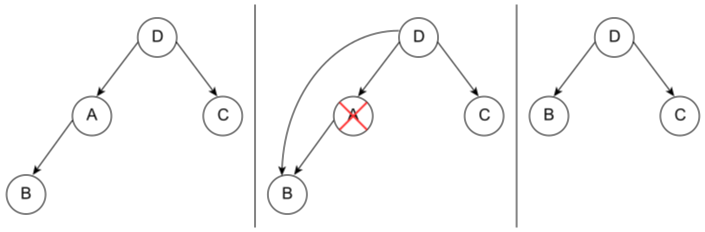
The process of inserting a node into a binary tree

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.

Deletion

Deletion is the process whereby a node is removed from the tree. Only certain nodes in a binary tree can be removed unambiguously.[31]

Node with zero or one children



The process of deleting an internal node in a binary tree

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

In a binary tree, a node with two children cannot be deleted unambiguously.[31] However, in certain binary trees (including binary search trees) these nodes *can* be deleted, though with a rearrangement of the tree structure.

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. Below are the brief descriptions of above mentioned traversals.

Pre-order

In pre-order, we always visit the current node; next, we recursively traverse the current node's left subtree, and then we recursively traverse the current node's right subtree. The pre-order traversal is a topologically sorted one, because a parent node is processed before any of its child nodes is done.

In-order

In in-order, we always recursively traverse the current node's left subtree; next, we visit the current node, and lastly, we recursively traverse the current node's right subtree.

Post-order

In post-order, we always recursively traverse the current node's left subtree; next, we recursively traverse the current node's right subtree and then visit the current node. Post-order traversal can be useful to get postfix expression of a binary expression tree.[32]

Depth-first order

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

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 ($d = \lfloor \log_2(i+1) \rfloor$) 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

- 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
- Unrooted binary tree

References

Citations

1. Rowan Garnier; John Taylor (2009). *Discrete Mathematics:Proofs, Structures and Applications, Third Edition* (https://books.google.com/books?id=WnkZSSc4IkoC&pg=PA620) . CRC Press. p. 620. ISBN 978-1-4398-1280-8.
2. Steven S Skiena (2009). *The Algorithm Design Manual* (https://books.google.com/books?id=7XUSn0IKQEgC&pg=PA77) . Springer Science & Business Media. p. 77. ISBN 978-1-84800-070-4.
3. 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* (https://books.google.com/books?id=yI4Jx5Obr08C&pg=PA246) . Mathematical Association of America. p. 246. ISBN 978-0-88385-762-5.
7. "Binary tree" (https://www.encyclopediaofmath.org/index.php?title=Binary_tree) , *Encyclopedia of Mathematics*, EMS Press, 2001 [1994] also in print as Michiel Hazewinkel (1997). *Encyclopaedia of Mathematics. Supplement I* (https://books.google.com/books?id=3ndQH4mTzWQC&pg=PA124) . Springer Science & Business Media. p. 124. ISBN 978-0-7923-4709-5.
8. L.R. Foulds (1992). *Graph Theory Applications* (https://books.google.com/books?id=IK7kreGl3vkC&pg=PA32) . Springer Science & Business Media. p. 32. ISBN 978-0-387-97599-3.
9. David Makinson (2009). *Sets, Logic and Maths for Computing*. Springer Science & Business Media. p. 199. ISBN 978-1-84628-845-6.
10. Jonathan L. Gross (2007). *Combinatorial Methods with Computer Applications* (https://books.google.com/books?id=hamtabmh0ZoC&pg=PA248) . CRC Press. p. 248. ISBN 978-1-58488-743-0.
11. 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* (https://books.google.com/books?id=vbxdqhDKOSYC&pg=PA66) . 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. Tamassia, Michael T. Goodrich, Roberto (2011). *Algorithm design : foundations, analysis, and Internet examples* (2 ed.). New Delhi: Wiley-India. p. 76. ISBN 978-81-265-0986-7.
16. "full binary tree" (http://xlinux.nist.gov/dads/HTML/fullBinaryTree.html) . NIST.
17. Richard Stanley, Enumerative Combinatorics, volume 2, p.36
18. "perfect binary tree" (https://xlinux.nist.gov/dads/HTML/perfectBinaryTree.html) . NIST.
19. "complete binary tree" (https://xlinux.nist.gov/dads/HTML/completeBinaryTree.html) . NIST.
20. "almost complete binary tree" (https://web.archive.org/web/20160304081430/http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/bintree.html) . Archived from the original (http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/bintree.html) on 2016-03-04. Retrieved 2015-12-11.
21. "nearly complete binary tree" (http://homepages.math.uic.edu/~leon/cs-mcs401-s08/handouts/nearly_complete.pdf) (PDF). Archived (https://ghostarchive.org/archive/20221009/http://homepages.math.uic.edu/~leon/cs-mcs401-s08/handouts/nearly_complete.pdf) (PDF) from the original on 2022-10-09.
22. Aaron M. Tenenbaum, et al. Data Structures Using C, Prentice Hall, 1990 ISBN 0-13-199746-7
23. 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 (http://xw2k.nist.gov/dads//HTML/balancedtree.html) Archived (https://web.archive.org/web/20101221085950/http://xw2k.nist.gov/dads/) December 21, 2010, at the Wayback Machine Accessed 2010-12-19.
24. Parmar, Anand K. (2020-01-22). "Different Types of Binary Tree with colourful illustrations" (https://towardsdatascience.com/5-types-of-binary-tree-with-cool-illustrations-9b335c430254) . *Medium*. Retrieved 2020-01-24.
25. Mehta, Dinesh; Sartaj Sahni (2004). *Handbook of Data Structures and Applications*. Chapman and Hall. ISBN 1-58488-435-5.

26. D. Samanta (2004). *Classic Data Structures*. PHI Learning Pvt. Ltd. pp. 264–265. ISBN 978-81-203-1874-8.

27. Michael L. Scott (2009). *Programming Language Pragmatics* (3rd ed.). Morgan Kaufmann. p. 347. ISBN 978-0-08-092299-7.

28. *Introduction to algorithms*. Cormen, Thomas H., Cormen, Thomas H. (2nd ed.). Cambridge, Mass.: MIT Press. 2001. p. 128. ISBN 0-262-03293-7. OCLC 46792720 (https://search.worldcat.org/oclc/46792720) .

29. Laakso, Mikko. "Priority Queue and Binary Heap" (http://www.cse.hut.fi/en/research/SVG/TRAKLA2/tutorials/heap_tutorial/taulukkona.html) . *University of Aalto*. Retrieved 2023-10-11.

30. Demaine, Erik. "6.897: Advanced Data Structures Spring 2003 Lecture 12" (https://web.archive.org/web/20051124175104/http://theory.csail.mit.edu/classes/6.897/spring03/scribe_notes/L12/lecture12.pdf) (PDF). MIT CSAIL. Archived from the original (http://theory.csail.mit.edu/classes/6.897/spring03/scribe_notes/L12/lecture12.pdf) (PDF) on 24 November 2005. Retrieved 14 April 2022.

31. Dung X. Nguyen (2003). "Binary Tree Structure" (http://www.clear.rice.edu/comp212/03-spring/lectures/22/) . rice.edu. Retrieved December 28, 2010.

32. Wittman, Todd (2015-02-13). "Lecture 18: Tree Traversals" (https://web.archive.org/web/20150213195803/http://www.math.ucla.edu/~wittman/10b.1.10w/Lectures/Lec18.pdf) (PDF). Archived from the original (http://www.math.ucla.edu/~wittman/10b.1.10w/Lectures/Lec18.pdf) (PDF) on 2015-02-13. Retrieved 2023-04-29.

Bibliography

- 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

- binary trees (http://www.findstat.org/BinaryTrees) Archived (https://web.archive.org/web/20200923083018/http://www.findstat.org/BinaryTrees) 2020-09-23 at the Wayback Machine entry in the FindStat (http://www.findstat.org/) database
- Binary Tree Proof by Induction (http://www.brpreiss.com/books/opus4/html/page355.html)
- Balanced binary search tree on array How to create bottom-up an Ahnentafel list, or a balanced binary search tree on array (http://piergiu.wordpress.com/2010/02/21/balanced-binary-search-tr

ee-on-array/)

- Binary trees and Implementation of the same with working code examples (https://web.archive.org/web/20190407044321/http://www.cpphub.com/search/label/Binary%20trees)
- Binary Tree JavaScript Implementation with source code (https://lufemas.github.io/binary-tree-jr/)