[**Tree**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-529) structures enable efficient access and efficient update to large collections of data. [**Binary trees**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-binary-tree) in particular are widely used and relatively easy to implement. But binary trees are useful for many things besides searching. Just a few examples of applications that trees can speed up include [**prioritizing jobs**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Heaps.html#heaps), [**describing mathematical expressions**](https://traky.cs.hut.fi/Books/CSE-A1141/html/BinaryTreeImpl.html#binarytreeimpl) and the syntactic elements of computer programs, or organizing the information needed to drive [**data compression algorithms**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-huffman-codes).

# Binary Trees

A [**binary tree**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-binary-tree) is made up of a finite set of elements called [**nodes**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-node). This set either is empty or consists of a node called the [**root**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-root) together with two binary trees, called the left and right [**subtrees**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-subtree), which are disjoint from each other and from the root. (Disjoint means that they have no nodes in common.) The roots of these subtrees are [**children**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-child) of the root. There is an [**edge**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-edge) from a node to each of its children, and a node is said to be the [**parent**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-parent) of its children.

If n1,n2,...,nkn1,n2,...,nk is a sequence of nodes in the tree such that nini is the parent of ni+1 for 1≤i<k, then this sequence is called a [**path**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-path) from n1n1 to nknk. The [**length**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-length) of the path is k−1k−1. If there is a path from node RR to node MM, then RR is an [**ancestor**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-ancestor) of MM, and MM is a [**descendant**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-descendant) of RR.

Thus, all nodes in the tree are descendants of the root of the tree, while the root is the ancestor of all nodes. The [**depth**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-depth) of a node M in the tree is the length of the path from the root of the tree to M. The [**height**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-height) of a tree is one more than the depth of the deepest node in the tree.

All nodes of depth dd are at [**level**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-level) ddin the tree. The root is the only node at level 0, and its depth is 0. A [**leaf node**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-leaf-node) is any node that has two empty children. An [**internal node**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-internal-node) is any node that has at least one non-empty child.

all binary tree nodes have two children (one or both of which might be empty),

A [**binary tree**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-binary-tree) is full if every [**node**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-node) is either a [**leaf node**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-leaf-node) or else it is an [**internal node**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-internal-node) with two non-empty [**children**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-child).

**complete binary tree**

A binary tree where the nodes are filled in row by row, with the bottom row filled in left to right. Due to this requirement, there is only one tree of nn nodes for any value of nn. Since storing the records in an array in row order leads to a simple mapping from a node's position in the array to its [***parent***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-parent), [***siblings***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-sibling), and [***children***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-child), the array representation is most commonly used to implement the complete binary tree. The [***heap***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-heap) data structure is a complete binary tree with partial ordering constraints on the node values.

In a [**tree**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-529), a sibling of [**node**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-node) AA is any other node with the same [**parent**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-parent) as AA.

**max heap**

A [***heap***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-heap) where every [***node***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-node) has a [***key***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-key) value greater than its [***children***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-child). As a consequence, the node with maximum key value is at the [***root***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-root).

The [**heap**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Heaps.html#heaps) data structure is an example of a complete binary tree. The [**Huffman coding tree**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-huffman-coding-tree) is an example of a full binary tree.  "Complete" is a wider word than "full", and complete binary trees tend to be wider than full binary trees because each level of a complete binary tree is as wide as possible.

**Full Binary Tree Theorem:** The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

When analyzing the space requirements for a binary tree implementation, it is useful to know how many empty subtrees a tree contains. A simple extension of the Full Binary Tree Theorem tells us exactly how many empty subtrees there are in any binary tree, whether full or not. Here are two approaches to proving the following theorem, and each suggests a useful way of thinking about binary trees.

That is:

The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.

By definition, every node in binary tree TT has two children, for a total of 2n2n children in a tree of nn nodes. Every node except the root node has one parent, for a total of n−1n−1 nodes with parents. In other words, there are n−1n−1 non-empty children. Because the total number of children is 2n2n, the remaining n+1n+1 children must be empty.

# Binary Tree Traversals

 Any process for visiting all of the nodes in some order is called a [**traversal**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-traversal). Any traversal that lists every node in the tree exactly once is called an [**enumeration**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-enumeration) of the tree's nodes.

we might wish to make sure that we visit any given node before we visit its children. This is called a [**preorder traversal**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-preorder-traversal). The first node printed is the root. Then all nodes of the left subtree are printed (in preorder) before any node of the right subtree.

Alternatively, we might wish to visit each node only after we visit its children (and their subtrees). For example, this would be necessary if we wish to return all nodes in the tree to free store. We would like to delete the children of a node before deleting the node itself. But to do that requires that the children's children be deleted first, and so on. This is called a [**postorder traversal**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-postorder-traversal).

An [**inorder traversal**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-inorder-traversal) first visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree). The [**binary search tree**](https://traky.cs.hut.fi/Books/CSE-A1141/html/BST.html#bst) makes use of this traversal to print all nodes in ascending order of value.

a tree is composed of a collection of node objects.

**interface** BinNode { *// Binary tree node ADT*

*// Get and set the element value*

**public** Object element();

**public** **void** setElement(Object v);

*// return the children*

**public** BinNode left();

**public** BinNode right();

*// return TRUE if a leaf node, FALSE otherwise*

**public** **boolean** isLeaf();

}

A traversal routine is naturally written as a recursive function. Its input parameter is a pointer to a node which we will call rt because each node can be viewed as the root of a some subtree. The initial call to the traversal function passes in a pointer to the root node of the tree. The traversal function visits rt and its children (if any) in the desired order. For example, a preorder traversal specifies that rt be visited before its children. This can easily be implemented as follows.

**static** **void** preorder(BinNode rt) {

**if** (rt == **null**) **return**; *// Empty subtree - do nothing*

visit(rt); *// Process root node*

preorder(rt.left()); *// Process all nodes in left*

preorder(rt.right()); *// Process all nodes in right*

}

preorder

The tree is going through the order: root, left underbody, right underbody.

inner order (inorder)

The tree is going through the order: the left underbody, the root, the right underbody.

postorder

The tree is going through the order: the left underbody, the right underbody, root.

levelorder

The wood is passed through each level from top to bottom from left to right.

**void** **traversePreOrder**(link t) {

**if** (t != NULL) {

visit(t);

traversePreOrder(t.getLeft());

traversePreOrder(t.getRight());

}

}

**void** **traverseInOrder**(link t) {

**if** (t != NULL) {

traverseInOrder(t.getLeft());

visit(t);

traverseInOrder(t.getRight());

}

}

**void** **traversePostOrder**(link t) {

**if** (t != NULL) {

traversePostOrder(t.getLeft());

traversePostOrder(t.getRight());

visit(t);

}

}

The preorder can also be implemented using a stack. Example in Java language for binary tree.

**void** **traversePreOrder**(link t) {

stack.push(t);

**do** {

t = stack.pop();

**if** (t != NULL) {

visit(t);

push(t.getRight());

push(t.getLeft());

}

} **while** ( !stack.empty() );

}

The order can be implemented by a queue. Example in C language for binary tree.

**void** **traverse\_levelorder**(lint t) {

queue.put(t);

**do** {

t = queue.get();

**if** (t != **null**) {

visit(t);

put(t.getLeft());

put(t.getRight());

}

} **while** ( !queue.empty() );

}

Dynamic data structure

A common way to implement a tree is to use a dynamic data structure where each node has its own record or class manifestation. This implementation sets a maximum limit on the number of children and is therefore not available in the case of general wood but is particularly suitable for situations where the number of children is constant. The following are examples of a binary tree node that is implemented in this way using the C and Java languages.

**struct** node {

**char** key;

**struct** node \*left;

**struct** node \*right;

};

**typedef** **struct** node \* link;

**class** **TreeNode**

{

**private** Object data;

**private** TreeNode left, right;

TreeNode (Object element)

{

data = element;

left = **null**;

right = **null**;

}

}

TreeNode root, p, q;

n order to make the structure as easy as possible, wood is often a separate 'Head' node, referred to as root and z-node, to which the child's links to the leaves or the empty links of the inner nodes refer to (see Figure 3). With these additions, entire wood can be treated with the same operations without the need for special roots and leaves.

If the tree should also be able to move upwards, the links should also be added to the nodes. If the tree needs to move upwards (including the general tree), it can easily be implemented as a simple table. The nodes are numbered and each node corresponds to a table index. That index stores the node number of that node.

A complete binary tree table presentation

A complete binary tree can be represented in a simple table so that the tree can be run both up and down. It is exactly the first element in the table. The node in k at index k is index k / 2 and children are in indices 2k and 2k + 1. For example, a binoculars are implemented in this way by a table, even if it is logically binary. More about this next week.

Bare bark of general wood

The common tree can be converted to "binary form" as follows:

Each node has a link to the first left to the left of the node.

Each node is linked to its closest sister to the right.

In this case, two links from each node of the new representation will be generated (see Figure 4). With the left hand you get to the deeper level in the original tree and the right lets you access the other children of the nun's nose.

# 6.1. General Trees[¶](https://traky.cs.hut.fi/Books/CSE-A1141/html/GenTreeIntro.html#general-trees)