# The `Tree` object



You can use the `Tree` object to build trees of other objects. Each object represents one node of a tree—it could be an isolated node, or part of a larger tree.

This version is somewhat limited in that children cannot be deleted, but parents can have any number of children, including zero.

In classic object-oriented design, the usual practice is to create a base tree class, and customize it to add the other fields you want. However, I prefer a different approach, *delegation*, as championed in Erich Gamma's book *Design Patterns*, among other sources.

With delegation, if you have an object that may be structured in trees, each of your objects has a pointer to a `Tree` object, and that `Tree` object has a pointer back at your object.

For example, suppose you are building a tree of topics in an outline. Your `Topic` object will have methods to obtain its parent topic, and it will also have methods to produce its child topics (if any). Each `Topic` object delegates those methods to its related `Tree` object, and each `Topic` object should have a field (perhaps named `.tree`) to point to the corresponding `Tree` node.

From the `Tree` object's point of view, the `Topic` object is considered the `Tree` object's *value*.

## *Methods*

`Tree_New`

When you create an object that has "treeness," you should create a corresponding `Tree` object pointing back at your object, using:

```
Tree_New ( parent, value )
```

where the arguments have these definitions:

| | |
|---|---|
| `parent` | The `Tree` object that is the parent of the one you are creating. If you are creating a tree node that has no parent (for example, because it is the root of the tree), pass a value of `&null` to this argument. |
| `value` | The object that is related to the new `Tree` object. |

To continue the example of the `Topic` object above, suppose the state for a Topic object is declared like this:

```
record topicTag ( tree, ...  )
```

Then the code to create a new `Topic` object might look like this:

```
local topic

topic       :=  topicTag ( );
topic.tree  :=  Tree_New ( Tree_Value ( parentTopic ), topic );
  ...
```

We assume that `parentTopic` is the parent of the new topic in the tree, or `&null` if it is the root.

After this code is executed, the `topic` object will point to its `Tree` object in its `topic.tree`, and the `Tree` object will point back to `topic` as its value.

`Tree_Value`

To obtain the value of a `Tree` object, use:

```
Tree_Value ( tree )
```

This method will return the value passed in to the `Tree_New()` method upon its creation.

`Tree_Set_Value`

If you want to change the value of a `Tree` object, use:

```
Tree_Set_Value ( tree, value )
```

where `value` is the new value you want stored in `tree`.

`Tree_Parent`

To find the parent `Tree` object of any `Tree` object, use:

```
Tree_Parent ( tree )
```

This method returns `&null` if `tree` is isolated or the root node; otherwise it returns `tree`'s parent as a `Tree` object.

`Tree_N_Children`

To find out how many children a `Tree` node has, use:

```
Tree_N_Children ( tree )
```

The number of children is returned as an integer. If no children have ever been added, it returns zero.

`Tree_Nth_Child`

To retrieve the $n$th child of a `Tree` node, use:

`Tree_Nth_Child ( tree,` $n$ `)`

If the given `tree` object has fewer than $n$ children, it fails; otherwise it returns the `Tree` object for the $n$th child of `tree`.

`Tree_Birth_Order`

If `Tree` object $P$ has a child $C$, you can find out the "birth order" of $C$ by:

```
Tree_Birth_Order ( tree )
```

If `tree` is the $n$th child to be added to its parent, this method returns $n$ as an integer. If `tree` has no parent, the method returns 1.

## Intended functions

More information about my particular variant of the Cleanroom software development methodology can be found at:

    http://www.nmt.edu/~john/soft/clean

Here are the Cleanroom intended functions for the above methods.

```
#--
# Tree_New ( value )
#   [ return a tree node with no parent, no children, and
#     the given value (which may be & null)
#   ]
#--
# Tree_Value ( self )
#   [ returns self's value
#   ]
#--
# Tree_Set_Value ( self, value )
#   [ self  :=  self with the new value attached
#   ]
#--
# Tree_Add_Child ( self, child )
#   [ if child is a Tree object ->
#       self   :=  self with child added as its new last child
#       child  :=  child with self added as its parent
#   ]
#--
# Tree_Parent ( self )
#   [ if self has a parent ->
#       return that parent
#     else ->  fail
#   ]
#--
# Tree_N_Children ( self )
#   [ returns the number of self's children as an integer
#   ]
#--
# Tree_Gen_Children ( self )
#   [ generates the child Tree objects of self in ascending birth order
#   ]
#--
# Tree_Nth_Child ( self, n )
#   [ if self has at least n children ->
#       return the nth child Tree object
#     else ->  fail
#   ]
#--
```

```
#--
# Tree_Birth_Order ( self )
#   [ if self has a parent ->
#       return self's child number relative to that parent
#     else ->  return 1
#   ]
#--
```

---

Written by John W. Shipman (john@nmt.edu). This version printed 1996-11-05.