Package ‘data.tree’

March 6, 2018

Type Package
Title General Purpose Hierarchical Data Structure
Version 0.7.5
Date 2018-03-06

VignetteBuilder knitr, rmarkdown
Imports R6, stringr, methods, DiagrammeR
Suggests Formula, graphics, testthat, knitr, rmarkdown, ape, yaml,
       networkD3, jsonlite, igraph, treemap, party, partykit,
       doParallel, foreach, htmlwidgets

Enhances

Description Create tree structures from hierarchical data, and traverse the
tree in various orders. Aggregate, cumulate, print, plot, convert to and from
data.frame and more. Useful for decision trees, machine learning, finance,
conversion from and to JSON, and many other applications.

License GPL (>= 2)

URL http://github.com/gluc/data.tree

BugReports http://github.com/gluc/data.tree/issues

Depends R (>= 3.0)

RoxygenNote 6.0.1

NeedsCompilation no

Author Facundo Munoz [ctb] (improve list conversion),
       Markus Wamser [ctb] (fixed some typos),
       Pierre Formont [ctb] (additional features),
       Kent Russel [ctb] (documentation),
       Noam Ross [ctb] (fixes),
       Duncan Garmonsway [ctb] (fixes),
       Christoph Glur [aut, cre] (R interface)

Maintainer Christoph Glur <christoph.glur@ipub.com>

Repository CRAN

Date/Publication 2018-03-06 22:42:34 UTC
R topics documented:

- acme .......................................................... 3
- Aggregate .................................................. 3
- AreNamesUnique ........................................... 5
- as.data.frame.Node ....................................... 5
- as.dendrogram.Node ....................................... 8
- as.igraph.Node ............................................ 9
- as.list.Node ................................................ 10
- as.Node ...................................................... 11
- as.Node.BinaryTree ...................................... 11
- as.Node.data.frame ....................................... 12
- as.Node.dendrogram ...................................... 14
- as.Node.list ............................................... 15
- as.Node.party ............................................. 17
- as.Node.phylo ............................................. 19
- as.phylo.Node ............................................. 20
- averageBranchingFactor ................................. 20
- CheckNameReservedWord .................................. 21
- Climb ....................................................... 21
- Clone ....................................................... 22
- CreateRandomTree ........................................ 23
- CreateRegularTree ....................................... 24
- Cumulate .................................................... 24
- data.tree ................................................... 25
- DefaultPlotHeight ....................................... 27
- Distance .................................................... 28
- Do .......................................................... 28
- FindNode ................................................... 29
- FormatFixedDecimal ...................................... 30
- FormatPercent ............................................. 31
- Get .......................................................... 31
- GetAttribute ............................................. 33
- GetPhyloNr ................................................ 34
- isLeaf ....................................................... 35
- isNotLeaf .................................................. 35
- isNotRoot .................................................. 36
- isRoot ....................................................... 36
- mushroom ................................................... 37
- Navigate ..................................................... 37
- Node ......................................................... 38
- NODE_RESERVED_NAMES_CONST ........................... 40
- plot.Node ................................................... 40
- print.Node ................................................... 43
- Prune ....................................................... 44
- Revert ...................................................... 45
- Set .......................................................... 45
- SetFormat ................................................... 46
**acme**

Sample Data: A Simple Company with Departments

---

**Description**

Acme’s tree representation is accessed through its root, acme.

**Usage**

data(acme)

**Format**

A data.tree root Node

**Details**

- cost, only available for leaf nodes. Cost of the project.
- p probability that a project will be undertaken.

---

**Aggregate**

Aggregate child values of a Node, recursively.

---

**Description**

The Aggregate method lets you fetch an attribute from a Node’s children, and then aggregate them using aggFun. For example, you can aggregate cost by summing costs of child Nodes. This is especially useful in the context of tree traversal, when using post-order traversal mode.

**Usage**

Aggregate(node, attribute, aggFun, ...)

Arguments

node the Node on which to aggregate
attribute determines what is collected. The attribute can be
  • a.) the name of a field or a property/active of each Node in the tree, e.g. acme$Get("p") or acme$Get("position")
  • b.) the name of a method of each Node in the tree, e.g. acme$Get("levelZeroBased"), where e.g. acme$levelZeroBased <- function() acme$level - 1
  • c.) a function, whose first argument must be a Node e.g. acme$Get(function(node) node$cost

aggFun the aggregation function to be applied to the children’s attributes
... any arguments to be passed on to attribute (in case it’s a function)

Details

As with Get, the attribute can be a field, a method or a function. If the attribute on a child is NULL, Aggregate is called recursively on its children.

See Also

Node

Examples

data(acme)

#Aggregate on a field
Aggregate(acme, "cost", sum)

#This is the same as:
HomeRolledAggregate <- function(node) {
  sum(sapply(node$children, function(child) {
    if (!is.null(child$cost)) child$cost
    else HomeRolledAggregate(child)
  }))
}
HomeRolledAggregate(acme)

#Aggregate using Get
print(acme, "cost", minCost = acme$Get(Aggregate, "cost", min))

#use Aggregate with a function:
Aggregate(acme, function(x) x$cost * x$p, sum)

#cache values along the way
acme$Do(function(x) x$cost <- Aggregate(x, "cost", sum), traversal = "post-order")
acme$IT$cost
AreNamesUnique

Test whether all node names are unique.

Description

This can be useful for some conversions.

Usage

AreNamesUnique(node)

Arguments

node The root Node of the data.tree structure to test

Value

TRUE if all Node$name == TRUE for all nodes in the tree

See Also

as.igraph.Node

Examples

data(acme)
AreNamesUnique(acme)
acme$name <- "IT"
AreNamesUnique(acme)

as.data.frame.Node

Convert a data.tree structure to a data.frame

Description

If a node field contains data of length > 1, then that is converted into a string in the data.frame.
Usage

```r
## S3 method for class 'Node'
as.data.frame(x, row.names = NULL, optional = FALSE, ...,
  traversal = c("pre-order", "post-order", "in-order", "level", "ancestor"),
  pruneFun = NULL, filterFun = NULL, format = FALSE,
  inheritFromAncestors = FALSE)

ToDataFrameTree(x, ..., pruneFun = NULL)

ToDataFrameTable(x, ..., pruneFun = NULL)

ToDataFrameNetwork(x, ..., direction = c("climb", "descend"),
  pruneFun = NULL, format = FALSE, inheritFromAncestors = FALSE)

ToDataFrameTypeCol(x, ..., type = "level", prefix = type, pruneFun = NULL)
```

Arguments

- `x` The root Node of the tree or sub-tree to be convert to a data.frame
- `row.names` NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
- `optional` logical. If TRUE, setting row names and converting column names (to syntactic names: see make.names) is optional.
- `...` the attributes to be added as columns of the data.frame. See `Get` for details. If a specific Node does not contain the attribute, `NA` is added to the data.frame.
- `traversal` any of 'pre-order' (the default), 'post-order', 'in-order', 'level', or 'ancestor'. See `Traverse` for details.
- `pruneFun` allows providing a a prune criteria, i.e. a function taking a Node as an input, and returning TRUE or FALSE. If the pruneFun returns FALSE for a Node, then the Node and its entire sub-tree will not be considered.
- `filterFun` a function taking a Node as an argument. See `Traverse` for details.
- `format` if FALSE (the default), then no formatting will be applied. If TRUE, then the first formatter (if any) along the ancestor path is used for formatting.
- `inheritFromAncestors` if FALSE, and if the attribute is a field or a method, then only a Node itself is searched for the field/method. If TRUE, and if the Node does not contain the attribute, then ancestors are also searched.
- `direction` when converting to a network, should the edges point from root to children ("climb") or from child to parent ("descend")?
- `type` when converting type columns, the type is the discriminator, i.e. an attribute (e.g. field name) of each node
- `prefix` when converting type columns, the prefix used for the column names. Can be NULL to omit prefixes.
as.data.frame.Node

Value

ToDataFrameTree: a data.frame, where each row represents a Node in the tree or sub-tree spanned by x, possibly pruned according to pruneFun.

ToDataFrameTable: a data.frame, where each row represents a leaf Node in the tree or sub-tree spanned by x, possibly pruned according to pruneFun.

ToDataFrameNetwork: a data.frame, where each row represents a Node in the tree or sub-tree spanned by x, possibly pruned according to pruneFun. The first column is called 'from', while the second is called 'to', describing the parent to child edge (for direction "climb") or the child to parent edge (for direction "descend"). If AreNamesUnique is TRUE, then the Network is based on the Node$name, otherwise on the Node$pathString.

ToDataFrameTypeCol: a data.frame in table format (i.e. where each row represents a leaf in the tree or sub-tree spanned by x), possibly pruned according to pruneFun. In addition to ..., each distinct type is output to a column.

Examples

```r
data(acme)
acme$fieldsAll
as.data.frame(acme, row.names = NULL, optional = FALSE, "cost", "p")

ToDataFrameTree(acme, "cost", "p")
ToDataFrameNetwork(acme, "cost", "p", direction = "climb")
ToDataFrameTable(acme, "cost", "p")
ToDataFrameTypeCol(acme)

# use the pruneFun:
acme$Do(function(x) x$totalCost <- Aggregate(x, "cost", sum), traversal = "post-order")
ToDataFrameTree(acme, "totalCost", pruneFun = function(x) x$totalCost > 300000)

# inherit
acme$Set(floor = c(1, 2, 3), filterFun = function(x) x$level == 2)
as.data.frame(acme, row.names = NULL, optional = FALSE, "floor", inheritFromAncestors = FALSE)
as.data.frame(acme, row.names = NULL, optional = FALSE, "floor", inheritFromAncestors = TRUE)

# using a function as an attribute:
acme$Accounting$Head <- "Mrs. Numright"
acme$Research$Head <- "Mr. Stein"
acme$IT$Head <- "Mr. Squarehead"
ToDataFrameTable(acme, department = function(x) x$parent$name, "name", "Head", "cost")

# complex TypeCol
acme$IT$Outsource$AddChild("India")
acme$IT$Outsource$AddChild("Poland")
acme$Set(type = c('company', 'department', 'project', 'project', 'department', 'project', 'project', 'department', 'program', 'project', 'project', 'project', 'project'))

print(acme, 'type')
ToDataFrameTypeCol(acme, type = 'type')
```
as.dendrogram.Node  

Convert a Node to a dendrogram

Description

Convert a data.tree structure to a dendrogram

Usage

```r
## S3 method for class 'Node'
as.dendrogram(object, heightAttribute = DefaultPlotHeight,
edgetext = FALSE, ...)
```

Arguments

- `object` The Node to convert
- `heightAttribute` The attribute (field name or function) storing the height
- `edgetext` If TRUE, then the for non-leaf nodes the node name is stored as the dendrogram’s edge text.

Value

An object of class dendrogram

See Also

Other Conversions from Node: ToNewick

Examples

```r
data(acme)
amed <- as.dendrogram(acme)
plot(acmed, center = TRUE)

# you can take an attribute for the height:
acme$Do( function(x) x$myPlotHeight <- (10 - x$level))
amed <- as.dendrogram(acme, heightAttribute = "myPlotHeight")
plot(acmed, center = TRUE)

# or directly a function
amed <- as.dendrogram(acme, heightAttribute = function(x) 10 - x$level)
plot(acmed)
```
as.igraph.Node

Convert a data.tree structure to an igraph network

Description
This requires the igraph package to be installed. Also, this requires the names of the Nodes to be unique within the data.tree structure.

Usage
as.igraph.Node(node, vertexAttributes = character(),
  edgeAttributes = character(), directed = FALSE, direction = c("climb", "descend"))

Arguments

node The root Node to convert

vertexAttributes A vector of strings, representing the attributes in the data.tree structure to add as attributes to the vertices of the igraph

edgeAttributes A vector of strings, representing the attributes in the data.tree structure to add as edge attributes of the igraph

directed Logical scalar, whether or not to create a directed graph.

direction when converting to a network, should the edges point from root to children ("climb") or from child to parent ("descend")?

Value
an igraph object

See Also
AreNamesUnique

Examples

data(acme)
library(igraph)
ig <- as.igraph(acme, "p", c("level", "isLeaf"))
plot(ig)
as.list.Node

Convert a data.tree structure to a list-of-list structure

Description

Convert a data.tree structure to a list-of-list structure

Usage

```r
## S3 method for class 'Node'
as.list(x, mode = c("simple", "explicit"), unname = FALSE,
  nodeName = ifelse(unname, "name", ""), childrenName = "children",
  rootName = "", keepOnly = NULL, ...)
```

```r
ToListSimple(x, nodeName = "name", ...)
```

```r
ToListExplicit(x, unname = FALSE, nodeName = ifelse(unname, "name", ""),
  childrenName = "children", ...)
```

Arguments

- `x`: The Node to convert
- `mode`: How the list is structured. "simple" (the default) will add children directly as nested lists. "explicit" puts children in a separate nested list called `childrenName`
- `unname`: If TRUE, and if `mode` is "explicit", then the nested children list will not have named arguments. This can be useful e.g. in the context of conversion to JSON, if you prefer the children to be an array rather than named objects.
- `nodeName`: The name that should be given to the name element
- `childrenName`: The name that should be given to the children nested list
- `rootName`: The name of the node. If provided, this overrides `Node$name`
- `keepOnly`: A character vector of fields to include in the result. If `NULL` (the default), all fields are kept.
- `...`: Additional parameters passed to `as.list.Node`

Examples

```r
data(acme)

str(ListSimple(acme))
str(ListSimple(acme, keepOnly = "cost"))

str(ListExplicit(acme))
str(ListExplicit(acme, unname = TRUE))
str(ListExplicit(acme, unname = TRUE, nodeName = "id", childrenName = "descendants"))
```
as.Node

Convert an object to a data.tree data structure

Description
Convert an object to a data.tree data structure

Usage
as.Node(x, ...)

Arguments
x  The object to be converted
... Additional arguments

See Also
Other as.Node: as.Node.data.frame, as.Node.dendrogram, as.Node.list, as.Node.phylo

as.Node.BinaryTree

Convert a SplitNode from the party package to a data.tree structure.

Description
Convert a SplitNode from the party package to a data.tree structure.

Usage
## S3 method for class 'BinaryTree'
as.Node(x, ...)

Arguments
x  The BinaryTree
... additional arguments (unused)
Examples

library(party)
airq <- subset(airquality, !is.na(Ozone))
airct <- ctree(Ozone ~ ., data = airq,
               controls = ctree_control(maxsurrogate = 3))

tree <- as.Node(airct)
tree

print(tree,
  "label",
  criterion = function(x) round(x$criterion$maxcriterion, 3),
  statistic = function(x) round(max(x$criterion$statistic), 3)
)

FindNode(tree, 6)$path

as.Node.data.frame  Convert a data.frame to a data.tree structure

Description

Convert a data.frame to a data.tree structure

Usage

## S3 method for class 'data.frame'
as.Node(x, ..., mode = c("table", "network"),
         pathName = "pathString", pathDelimiter = "/", colLevels = NULL,
         na.rm = TRUE)

FromDataFrameTable(table, pathName = "pathString", pathDelimiter = "/",
                    colLevels = NULL, na.rm = TRUE, check = c("check", "no-warn",
                    "no-check"))

FromDataFrameNetwork(network, check = c("check", "no-warn", "no-check"))

Arguments

x  The data.frame in the required format.
...
Any other argument implementations of this might need
mode  Either "table" (if x is a data.frame in tree or table format) or "network"
pathName  The name of the column in x containing the path of the row
pathDelimiter  The delimiter used to separate nodes in pathName
collevels  Nested list of column names, determining on what node levels the attributes are written to.

na.rm  If TRUE, then NA’s are treated as NULL and values will not be set on nodes

table  a data.frame in table or tree format, i.e. having a row for each leaf (and optionally for additional nodes). There should be a column called pathName, separated by pathDelimiter, describing the path of each row.

check  Either
  • "check": if the name conformance should be checked and warnings should be printed in case of non-conformance (the default)
  • "no-warn": if the name conformance should be checked, but no warnings should be printed in case of non-conformance (if you expect non-conformance)
  • "no-check" or FALSE: if the name conformance should not be checked; use this if performance is critical. However, in case of non-conformance, expect cryptic follow-up errors

network  A data.frame in network format, i.e. it must adhere to the following requirements:
  • It must contain as many rows as there are nodes (excluding the root, there is no row for the root)
  • Its first and second columns contain the network relationships. This can be either climbing (from parent to children) or descending (from child to parent)
  • Its subsequent columns contain the attributes to be set as fields on the nodes
  • It must contain a single root
  • There are no cycles in the network

Value

The root Node of the data.tree structure

See Also

as.data.frame.Node

Other as.Node: as.Node.dendrogram, as.Node.list, as.Node.phylo, as.Node

Examples

data(acme)

#Tree
x <- ToDataFrameTree(acme, "pathString", "p", "cost")
x
xN <- as.Node(x)
print(xN, "p", "cost")

#Table
x <- ToDataFrameTable(acme, "pathString", "p", "cost")
as.Node.dendrogram

Convert a dendrogram to a data.tree Node

Description

Convert a dendrogram to a data.tree Node

Usage

## S3 method for class 'dendrogram'
as.Node(x, name = "Root", heightName = "plotHeight",
check = c("check", "no-warn", "no-check"), ...)

Arguments

- `x` The dendrogram
- `name` The name of the root Node
- `heightName` The name under which the dendrogram’s height is stored
- `check` Either
  - "check": if the name conformance should be checked and warnings should be printed in case of non-conformance (the default)
  - "no-warn": if the name conformance should be checked, but no warnings should be printed in case of non-conformance (if you expect non-conformance)
  - "no-check" or FALSE: if the name conformance should not be checked; use this if performance is critical. However, in case of non-conformance, expect cryptic follow-up errors
- ... Additional parameters

```r
x
xN <- FromDataFrameTable(x)
print(xN, "p", "cost")

# More complex Table structure, using collevels
acme$Set(floor = c(1,2,3), filterFun = function(x) x$level == 2)
x <- ToDataFrameTable(acme, "pathString", "floor", "p", "cost")
x
xN <- FromDataFrameTable(x, colLevels = list(NULL, "floor", c("p", "cost")), na.rm = TRUE)
print(xN, "floor", "p", "cost")

# Network
x <- ToDataFrameNetwork(acme, "p", "cost", direction = "climb")
x
xN <- FromDataFrameNetwork(x)
print(xN, "p", "cost")
```
Value

The root Node of a data.tree

See Also

Other as.Node: as.Node.data.frame, as.Node.list, as.Node.phylo, as.Node

Examples

hc <- hclust(dist(USArrests), "ave")
dend1 <- as.dendrogram(hc)
tree1 <- as.Node(dend1)
tree1$fieldsAll
tree1$totalCount
tree1$leafCount
tree1$height

as.Node.list

Convert a nested list structure to a data.tree structure

Description

Convert a nested list structure to a data.tree structure

Usage

## S3 method for class 'list'
as.Node(x, mode = c("simple", "explicit"), nameName = "name", childrenName = "children", nodeName = NULL, check = c("check", "no-warn", "no-check"), ...)

FromListExplicit(explicitList, nameName = "name", childrenName = "children", nodeName = NULL, check = c("check", "no-warn", "no-check"))

FromListSimple(simpleList, nameName = "name", nodeName = NULL, check = c("check", "no-warn", "no-check"))

Arguments

x

The list to be converted.

mode

How the list is structured. "simple" (the default) will interpret any list to be a child. "explicit" assumes that children are in a nested list called childrenName

nameName

The name of the element in the list that should be used as the name, can be NULL if mode = explicit and the children lists are named, or if an automatic name (running number) should be assigned
childrenName      The name of the element that contains the child list (applies to mode 'explicit' only).

nodeName         A name suggestion for x, if the name cannot be deferred otherwise. This is for example the case for the root with mode explicit and named lists.

check             Either
                    • "check": if the name conformance should be checked and warnings should be printed in case of non-conformance (the default)
                    • "no-warn": if the name conformance should be checked, but no warnings should be printed in case of non-conformance (if you expect non-conformance)
                    • "no-check" or FALSE: if the name conformance should not be checked; use this if performance is critical. However, in case of non-conformance, expect cryptic follow-up errors

...                Any other argument to be passed to generic sub implementations

explicitList      A list in which children are in a separate nested list called childrenName.

simpleList        A list in which children are stored as nested list alongside other fields. Any list is interpreted as a child Node

See Also
        Other as.Node: as.Node.data.frame, as.Node.dendrogram, as.Node.phylo, as.Node

Examples

kingJosephs <- list(name = "Joseph I",
                     spouse = "Mary",
                     born = "1818-02-23",
                     died = "1839-08-29",
                     children = list(
                        list(name = "Joseph II",
                             spouse = "Kathryn",
                             born = "1839-03-28",
                             died = "1865-12-19"),
                        list(name = "Helen",
                             born = "1840-17-08",
                             died = "1845-01-01")
                     )
)

FromListExplicit(kingJosephs)

kingJosephs <- list(head = "Joseph I",
                     spouse = "Mary",
                     born = "1818-02-23",
                     died = "1839-08-29",
                     list(head = "Joseph II",
                          spouse = "Kathryn",
                          born = "1839-03-28",
                          died = "1865-12-19"),
                     list(head = "Helen",
                          spouse = "Kathryn",
                          born = "1840-17-08",
                          died = "1845-01-01")
)
```
born = "1840-17-08",
died = "1845-01-01"
)
FromListSimple(kingJosephs, nameName = "head")

kingJosephs <- list(spouse = "Mary",
  born = "1818-02-23",
  died = "1839-08-29",
  `Joseph II` = list(spouse = "Kathryn",
    born = "1839-03-28",
    died = "1865-12-19"),
  Helen = list(born = "1840-17-08",
    died = "1845-01-01")
)
FromListSimple(kingJosephs, nodeName = "Joseph I")
```

**as.Node.party**

Convert a party from the partykit package to a data.tree structure.

**Description**

Convert a party from the partykit package to a data.tree structure.

**Usage**

```r
## S3 method for class 'party'
as.Node(x, ...)
```

**Arguments**

- `x` The party object
- `...` other arguments (unused)

**Examples**

```r
library(partykit)
data("WeatherPlay", package = "partykit")
### splits ###
# split in overcast, humidity, and windy
sp_o <- partysplit(1L, index = 1:3)
sp_h <- partysplit(3L, breaks = 75)
sp_w <- partysplit(4L, index = 1:2)

### query labels
character_split(sp_o)

### nodes ###
```
## set up partynode structure
pn <- partynode(l, split = sp_o, kids = list(
  partynode(2L, split = sp_h, kids = list(
    partynode(3L, info = "yes"),
    partynode(4L, info = "no"))),
  partynode(5L, info = "yes"),
  partynode(6L, split = sp_w, kids = list(
    partynode(7L, info = "yes"),
    partynode(8L, info = "no"))))))

## tree ###
## party: associate recursive partynode structure with data
py <- party(pn, WeatherPlay)
tree <- as.Node(py)

print(tree,
  "splitname",
  count = function(node) nrow(node$data),
  "splitLevel")

SetNodeStyle(tree,
  label = function(node) paste0(node$name, ": ", node$splitname),
  tooltip = function(node) paste0(nrow(node$data), " observations"),
  fontname = "helvetica")

SetEdgeStyle(tree,
  arrowhead = "none",
  label = function(node) node$splitLevel,
  fontname = "helvetica",
  penwidth = function(node) 12 * nrow(node$data)/nrow(node$root$data),
  color = function(node) {
    paste0("grey",
    100 - as.integer( 100 * nrow(node$data)/nrow(node$root$data))
  }
)

Do(tree$leaves,
  function(node) {
    SetNodeStyle(node,
      shape = "box",
      color = ifelse(node$splitname == "yes", "darkolivegreen4", "lightsalmon4"),
      fillcolor = ifelse(node$splitname == "yes", "darkolivegreen1", "lightsalmon"),
      style = "filled,rounded",
      penwidth = 2
    )
  })

plot(tree)
Description

Convert a phylo object from the ape package to a Node

Usage

```r
## S3 method for class 'phylo'
as.Node(x, heightName = "plotHeight",
       replaceUnderscores = TRUE, namesNotUnique = FALSE, ...)
```

Arguments

- `x` The phylo object to be converted
- `heightName` If the phylo contains edge lengths, then they will be converted to a height and stored in a field named according to this parameter (the default is "height")
- `replaceUnderscores` if TRUE (the default), then underscores in names are replaced with spaces
- `namesNotUnique` if TRUE, then the name of the Nodes will be prefixed with a unique id. This is useful if the children of a parent have non-unique names.
- `...` any other parameter to be passed to sub-implementations

See Also

Other ape phylo conversions: `getPhyloNr, as.phylo.Node`

Other as.Node: `as.Node.data.frame, as.Node.dendrogram, as.Node.list, as.Node`

Examples

```r
# Which bird families have the max height?
library(ape)
data(bird.families)
bf <- as.Node(bird.families)
height <- bf$height
t <- Traverse(bf, filterFun = function(x) x$level == 25)
Get(t, "name")
```
### as.phylo.Node

*Convert a Node to a phylo object from the ape package.*

**Description**

This method requires the ape package to be installed and loaded.

**Usage**

```r
as.phylo.Node(x, heightAttribute = DefaultPlotHeight, ...)
```

**Arguments**

- `x`: The root node of the tree or sub-tree to be converted
- `heightAttribute`: The attribute (field name or function) storing the height
- `...`: Any other argument

**See Also**

Other ape phylo conversions: `getPhyloNr`, `as.Node.phylo`

**Examples**

```r
library(ape)
data(acme)
acmephylo <- as.phylo(acme)
# plot(acmephylo)
```

### averageBranchingFactor

*Calculate the average number of branches each non-leaf has*

**Description**

Calculate the average number of branches each non-leaf has

**Usage**

```r
averageBranchingFactor(node)
```

**Arguments**

- `node`: The node
CheckNameReservedWord  Checks whether name is a reserved word, as defined in NODE_RESERVED_NAMES_CONST.

Description
Checks whether name is a reserved word, as defined in NODE_RESERVED_NAMES_CONST.

Usage
CheckNameReservedWord(name, check = c("check", "no-warn", "no-check"))

Arguments
name  the name to check
check Either
  • "check": if the name conformance should be checked and warnings should be printed in case of non-conformance (the default)
  • "no-warn": if the name conformance should be checked, but no warnings should be printed in case of non-conformance (if you expect non-conformance)
  • "no-check" or FALSE: if the name conformance should not be checked; use this if performance is critical. However, in case of non-conformance, expect cryptic follow-up errors

Climb  Climb a tree from parent to children, by provided criteria.

Description
This method lets you climb the tree, from crutch to crutch. On each Node, the Climb finds the first child having attribute value equal to the provided argument.

Usage
#node$Climb(...)  Climb(node, ...)

Arguments
node  The root Node of the tree or subtree to climb
...  an attribute name to searched value pairlist. For brevity, you can also provide a character vector.
Clone

Value
the Node having path .... or NULL if such a path does not exist

See Also
Node
Navigate

Examples

```r
data(acme)

# the following are all equivalent
Climb(acme, 'IT', 'Outsource')
Climb(acme, name = 'IT', name = 'Outsource')
Climb(acme, 'IT')$Climb('Outsource')
Navigate(acme, path = "IT/Outsource")

Climb(acme, name = 'IT')

Climb(acme, position = c(2, 1))
# or, equivalent:
Climb(acme, position = 2, position = 1)
Climb(acme, name = "IT", cost = 250000)

tree <- CreateRegularTree(5, 2)
tree$Climb(c("1", "1"), position = c(2, 2))$path
```

Clone
 Clone a tree (creates a deep copy)

Description
The method also clones object attributes (such as the formatters), if desired. If the method is called on a non-root, then the parent relationship is not cloned, and the resulting Node will be a root.

Usage
Clon(node, pruneFun = NULL, attributes = FALSE)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>the root node of the tree or sub-tree to clone</td>
</tr>
<tr>
<td>pruneFun</td>
<td>allows providing a a prune criteria, i.e. a function taking a Node as an input, and returning TRUE or FALSE. If the pruneFun returns FALSE for a Node, then the Node and its entire sub-tree will not be considered.</td>
</tr>
<tr>
<td>attributes</td>
<td>if FALSE, then R class attributes (e.g. formatters and grViz styles) are not cloned. This makes the method faster.</td>
</tr>
</tbody>
</table>
CreateRandomTree

Value

the clone of the tree or sub-tree

See Also

SetFormat

Examples

data(acme)
acmeClone <- Clone(acme)
acmeClone$name <- "New Acme"
# acmeClone does not point to the same reference object anymore:
acme$name

# cloning a subtree
data(acme)
itClone <- Clone(acme$IT)
itClone$isRoot

CreateRandomTree Create a tree for demo and testing

Description

Create a tree for demo and testing

Usage

CreateRandomTree(nodes = 100, root = Node$new("1"), id = 1)

Arguments

nodes The number of nodes to create
root the previous node (for recursion, typically use default value)
id The id (for recursion)
CreateRegularTree

Create a tree for demo and testing

Description

Create a tree for demo and testing

Usage

create_regular_tree(height = 5, branching_factor = 3, parent = Node$new("1"))

Arguments

- **height**: the number of levels
- **branching_factor**: the number of children per node
- **parent**: the parent node (for recursion)

Cumulate

Cumulate values among siblings

Description

For example, you can sum up values of siblings before this Node.

Usage

cumulate(node, attribute, aggFun, ...)

Arguments

- **node**: The node on which we want to cumulate
- **attribute**: determines what is collected. The attribute can be
  - a.) the name of a **field** or a **property/active** of each Node in the tree, e.g. `acme$Get("p")` or `acme$Get("position")`
  - b.) the name of a **method** of each Node in the tree, e.g. `acme$Get("levelZeroBased")`, where e.g. `acme$levelZeroBased <- function() acme$level - 1`
  - c.) a **function**, whose first argument must be a Node e.g. `acme$Get(function(node) node$cost * ...)
- **aggFun**: the aggregation function to be applied to the children’s attributes
- **...**: any arguments to be passed on to `attribute` (in case it’s a function)
Examples

```r
data(acme)
acme$Do(function(x) x$cost <- Aggregate(x, "cost", sum), traversal = "post-order")
acme$Do(function(x) x$cumCost <- Cumulate(x, "cost", sum))
print(acme, "cost", "cumCost")
```

Description

`data.tree` is to hierarchical data what `data.frame` is to tabular data: An extensible, general purpose structure to store, manipulate, and display hierarchical data.

Introduction

Hierarchical data is ubiquitous in statistics and programming (XML, search trees, family trees, classification, file system, etc.). However, no general-use tree data structure is available in R. Where tabular data has `data.frame`, hierarchical data is often modeled in lists of lists or similar makeshifts. These structures are often difficult to manage. This is where the `data.tree` package steps in. It lets you build trees of hierarchical data for various uses: to print, to rapid prototype search algorithms, to test out new classification algorithms, and much more.

Tree Traversal

`data.tree` allows to Traverse trees in various orders (pre-order, post-order, level, etc.), and it lets you run operations on Nodes via Do. Similarly, you can collect and store data while traversing a tree using the Get and the Set methods.

Methods

The package also contains utility functions to Sort, to Prune, to Aggregate and Cumulate and to print in custom formats.

Construction and Conversion

The package also contains many conversions from and to `data.tree` structures. Check out the see also section of as.Node.

You can construct a tree from a data.frame using as.Node.data.frame, and convert it back using as.data.frame.Node. Similar options exist for list of lists. For more specialized conversions, see as.dendrogram.Node, as.Node.dendrogram, as.phylo.Node and as.Node.phylo

Finally, easy conversion options from and to JSON, YAML, igraph, and more exist.
Node and Reference Semantics

The entry point to the package is `Node`. Each tree is composed of a number of Nodes, referencing each other.

One of most important things to note about `data.tree` is that it exhibits reference semantics. In a nutshell, this means that you can modify your tree along the way, without having to reassign it to a variable after each modification. By and large, this is a rather exceptional behavior in R, where value-semantics is king most of the time.

Applications

`data.tree` is not optimised for computational speed, but for implementation speed. Namely, its memory footprint is relatively large compared to traditional R data structures. However, it can easily handle trees with several thousand nodes, and once a tree is constructed, operations on it are relatively fast. `data.tree` is always useful when

- you want to develop and test a new algorithm
- you want to import and convert tree structures (it imports and exports to list-of-list, data.frame, yaml, json, igraph, dendrogram, phylo and more)
- you want to play around with data, display it and get an understanding
- you want to test another package, to compare it with your own results
- you need to do homework

For a quick overview of the features, read the `data.tree` vignette by running `vignette("data.tree")`. For stylized applications, see `vignette("applications", package='data.tree'")

See Also

`Node`

For more details, see the `data.tree` vignette by running: `vignette("data.tree")`

Examples

```r
data(acme)
print(acme)
acme$fieldsAll
acme$count
acme$totalCount
acme$isRoot
acme$height
print(acme, "p", "cost")

outsource <- acme$IT$Outsource
class(outsource)
print(outsource)
outsource$Fields
outsource$IsLeaf
outsource$Level
outsource$Path
outsource$p
```
DefaultPlotHeight

Calculates the height of a Node given the height of the root.

Description

This function puts leaves at the bottom (not hanging), and makes edges equally long. Useful for easy plotting with third-party packages, e.g. if you have no specific height attribute, e.g. with as.dendrogram.Node, ToNewick, and as.phylo.Node

Usage

DefaultPlotHeight(node, rootHeight = 100)

Arguments

node The node

rootHeight The height of the root

Examples

data(acme)
dacme <- as.dendrogram(acme, heightAttribute = function(x) DefaultPlotHeight(x, 200))
plot(dacme, center = TRUE)
## Distance

**Find the distance between two nodes of the same tree**

### Description

The distance is measured as the number of edges that need to be traversed to reach node2 when starting from node1.

### Usage

```
Distance(node1, node2)
```

### Arguments

- **node1**: the first node in the tree
- **node2**: the second node in the same tree

### Examples

```javascript
data(acme)
Distance(FindNode(acme, "Outsource"), FindNode(acme, "Research"))
```

---

## Do

**Executes a function on a set of nodes**

### Description

Executes a function on a set of nodes

### Usage

```less
# OO-style:
# node$Do(fun,
# ...
# traversal = c("pre-order", "post-order", "in-order", "level", "ancestor"),
# pruneFun = NULL,
# filterFun = NULL)

# traditional:
Do(nodes, fun, ...)
```
FindNode

Arguments

- **nodes**
  The nodes on which to perform the Get (typically obtained via Traverse)
- **fun**
  the function to execute. The function is expected to be either a Method, or to take a Node as its first argument
- ... any additional parameters to be passed on to fun

See Also

- Node
- Get
- Set
- Traverse

Examples

```r
data(acme)
acme$Do(function(node) node$expectedCost <- node$p * node$cost)
print(acme, "expectedCost")
```
See Also

AreNamesUnique, Traverse

Examples

data(acme)
FindNode(acme, "Outsource")

# re-usable hashed index for multiple searches:
if(!AreNamesUnique(acme)) stop("Hashed index works for unique names only!")
trav <- Traverse(acme, "level")
names(trav) <- Get(trav, "name")
nameIndex <- as.environment(trav)
# you could also use hash from package hash instead!
# nameIndex <- hash(trav)
nameIndex$Outsource
nameIndex$IT

FormatFixedDecimal  Format a Number as a Decimal

Description

Simple function that can be used as a format function when converting trees to a data.frame

Usage

FormatFixedDecimal(x, digits = 3)

Arguments

x  a numeric scalar or vector
digits  the number of digits to print after the decimal point

Value

A string corresponding to x, suitable for printing

Examples

data(acme)
print(acme, prob = acme$Get("p", format = function(x) FormatFixedDecimal(x, 4)))
**FormatPercent**

*Format a Number as a Percentage*

**Description**

This utility method can be used as a format function when converting trees to a `data.frame`.

**Usage**

```r
FormatPercent(x, digits = 2, format = "f", ...)
```

**Arguments**

- `x`: A number
- `digits`: The number of digits to print
- `format`: The format to use
- `...`: Any other argument passed to `formatC`

**Value**

A string corresponding to `x`, suitable for printing

**See Also**

- `formatC`

**Examples**

```r
data(acme)
print(acme, prob = acme$Get("p", format = FormatPercent))
```

---

**Get**

*Traverse a Tree and Collect Values*

**Description**

The `Get` method is one of the most important ones of the `data.tree` package. It lets you traverse a tree and collect values along the way. Alternatively, you can call a method or a function on each `Node`. 
Usage

# OO-style:
# node$Get(attribute,
# ...,
# traversal = c("pre-order", "post-order", "in-order", "level", "ancestor"),
# pruneFun = NULL,
# filterFun = NULL,
# format = FALSE,
# inheritFromAncestors = FALSE)

# traditional:
Get(nodes, attribute,
...,
format = FALSE,
inheritFromAncestors = FALSE,
simplify = c(TRUE, FALSE, "array", "regular"))

Arguments

nodes The nodes on which to perform the Get (typically obtained via Traverse)

attribute determines what is collected. The attribute can be

• a.) the name of a field or a property/active of each Node in the tree, e.g.
   acme$Get("p") or acme$Get("position")

• b.) the name of a method of each Node in the tree, e.g. acme$Get("levelZeroBased"),
   where e.g. acme$levelZeroBased <- function() acme$level - 1

• c.) a function, whose first argument must be a Node e.g. acme$Get(function(node) node$cost * node$weight)

... in case the attribute is a function or a method, the ellipsis is passed to it as additional arguments.

format if FALSE (the default), no formatting is being used. If TRUE, then the first formatter (if any) found along the ancestor path is being used for formatting (see SetFormat). If format is a function, then the collected value is passed to that function, and the result is returned.

inheritFromAncestors if TRUE, then the path above a Node is searched to get the attribute in case it is NULL.

simplify same as sapply, i.e. TRUE, FALSE or "array". Additionally, you can specify "regular" if each returned value is of length > 1, and equally named. See below for an example.

Value

a vector containing the attributes collected during traversal, in traversal order. NULL is converted to NA, such that length(Node$Get) == Node$totalCount
GetAttribute

See Also
   Node
   Set
   Do
   Traverse

Examples

  data(acme)
  acme$Get("level")
  acme$Get("totalCount")

  acme$Get(function(node) node$cost * node$p,
                 filterFun = isLeaf)

  #This is equivalent:
  nodes <- Traverse(acme, filterFun = isLeaf)
  Get(nodes, function(node) node$cost * node$p)

  #simplify = "regular" will preserve names
  acme$Get(function(x) c(position = x$position, level = x$level), simplify = "regular")

GetAttribute  Get an attribute from a Node.

Description

Get an attribute from a Node.

Usage

GetAttribute(node, attribute, ..., format = FALSE, 
             inheritFromAncestors = FALSE, nullAsNa = TRUE)

Arguments

  node                The Node from which the attribute should be fetched.
  attribute           determines what is collected. The attribute can be
                      • a.) the name of a field or a property/active of each Node in the tree, e.g.
                          acme$Get("p") or acme$Get("position")
                      • b.) the name of a method of each Node in the tree, e.g. acme$Get("levelZeroBased"),
                          where e.g. acme$levelZeroBased <- function() acme$level - 1
                      • c.) a function, whose first argument must be a Node e.g. acme$Get(function(node) node$cost * node$p)
in case the attribute is a function or a method, the ellipsis is passed to it as additional arguments.

`format` if FALSE (the default), no formatting is being used. If TRUE, then the first formatter (if any) found along the ancestor path is being used for formatting (see `SetFormat`). If format is a function, then the collected value is passed to that function, and the result is returned.

`inheritFromAncestors` if TRUE, then the path above a Node is searched to get the attribute in case it is NULL.

`nullAsNa` If TRUE (the default), then NULL is returned as NA. Otherwise it is returned as NULL.

**Examples**

```r
data(acme)
getattribute(acme$IT$Outsource, "cost")
```

---

**GetPhyloNr**

* Determine the number a Node has after conversion to a phylo object *

**Description**

Use this function when plotting a Node as a phylo, e.g. to set custom labels to plot.

**Usage**

```r
getPhyloNr(x, type = c("node", "edge"))
```

**Arguments**

- `x` The Node
- `type` Either "node" (the default) or "edge" (to get the number of the edge from x to its parent)

**Value**

an integer representing the node

**See Also**

Other ape phylo conversions: `as.Node.phylo`, `as.phylo.Node`
isLeaf

Examples

library(ape)
library(data.tree)
data(acme)
ap <- as.phylo(acme)
#plot(ap)
#nodelabels("IT Dep.", GetPhyloNr(Climb(acme, "IT")))
#edgelabels("Good!", GetPhyloNr(Climb(acme, "IT", "Switch to R"), "edge"))

isLeaf  Check if a Node is a leaf

Description
Check if a Node is a leaf

Usage

isLeaf(node)

Arguments
node The Node to test.

Value
TRUE if the Node is a leaf, FALSE otherwise

isNotLeaf  Check if a Node is not a leaf

Description
Check if a Node is not a leaf

Usage

isNotLeaf(node)

Arguments
node The Node to test.

Value
FALSE if the Node is a leaf, TRUE otherwise
isNotRoot  

**Description**  
Check if a Node is not a root

**Usage**  
isNotRoot(node)

**Arguments**

- node  
The Node to test.

**Value**  
FALSE if the Node is the root, TRUE otherwise

---

isRoot  

**Description**  
Check if a Node is the root

**Usage**  
isRoot(node)

**Arguments**

- node  
The Node to test.

**Value**  
TRUE if the Node is the root, FALSE otherwise
mushroom

Sample Data: Data Used by the ID3 Vignette

Description
mushroom contains attributes of mushrooms. We can use this data to predict a mushroom’s toxicity based on its attributes. The attributes available in the data set are:

Usage
data(mushroom)

Format
data.frame

Details
- color the color of a mushroom
- size whether a mushroom is small or large
- points whether a mushroom has points
- edibility whether a mushroom is edible or toxic

Navigate

Navigate to another node by relative path.

Description
Navigate to another node by relative path.

Usage
Navigate(node, path)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>The starting Node to navigate</td>
</tr>
<tr>
<td>path</td>
<td>A string or a character vector describing the path to navigate</td>
</tr>
</tbody>
</table>

Details
The path is always relative to the node. Navigation to the parent is defined by . . ., whereas navigation to a child is defined via the child’s name. If path is provided as a string, then the navigation steps are separated by '/'.

Create a data.tree Structure With Nodes

Description

Node is at the very heart of the data.tree package. All trees are constructed by tying together Node objects.

Usage

```r
# n1 <- Node$new("Node 1")
```

Format

An `R6Class` generator object

Details

Assemble Node objects into a data.tree structure and use the traversal methods to set, get, and perform operations on it. Typically, you construct larger tree structures by converting from `data.frame`, `list`, or other formats.

Fields

- `children` A list of child Nodes
- `parent` The node's parent Node

Methods

- `Node$new(name)` Creates a new Node called name. Often used to construct the root when creating trees programmatically.
- `AddChild(name)` Creates a new Node called name and adds it to this Node as a child.
- `AddChildNode(node)` Adds a Node as a child.
- `AddSibling(name)` Creates a new Node called name and adds it after this Node as a sibling.
- `AddSiblingNode(sibling)` Adds a new Node after this Node, as a sibling.
- `RemoveChild(name)` Remove the child Node called name from a Node and returns it.
RemoveAttribute(name, stopIfNotAvailable) Removes attribute called name from this Node. Gives an error if stopIfNotAvailable and the attribute does not exist.

Climb(...) Find a node with path ..., where the ... arguments are the names of the Nodes, or other field values.

Navigate(path) Find a node by relative path

FindNode(name) Find a node with name name. Especially useful if AreNamesUnique is TRUE

Get(attribute, ..., traversal = c("pre-order", "post-order", "in-order", "level", "ancestor"), pruneFun = NULL, filterFun = NULL)
Traverses the tree and collects values along the way.

Do(fun, ..., traversal = c("pre-order", "post-order", "in-order", "level", "ancestor"), pruneFun = NULL, filterFun = NULL)
Traverses the tree and calls fun on each node.

Set(..., traversal = c("pre-order", "post-order", "in-order", "level", "ancestor"), pruneFun = NULL, filterFun = NULL)
Traverses the tree and assigns the args along the way, recycling the args.

Sort(attribute, ..., decreasing = FALSE, recursive = TRUE) Sort children of a node with respect to an attribute (field, method, active, function)

Revert(recursive = TRUE) Revert the sort order of a node

Prune(pruneFun) Prune a tree. The pruneFun takes a node as its first argument, and returns TRUE if the node should be kept, FALSE otherwise

Actives (aka Properties)

name Gets or sets the name of a Node. For example Node$name <- "Acme"

parent Gets or sets the parent Node of a Node. Only set this if you know what you are doing, as you might mess up the tree structure!

children Gets or sets the children list of a Node. Only set this if you know what you are doing, as you might mess up the tree structure!

siblings Returns a list of the siblings of this Node

fields Gets the names of the set properties of a Node

fieldsAll Gets the names of the set properties of a Node or any of its sub-Nodes

isLeaf Returns TRUE if the Node is a leaf, FALSE otherwise

isRoot Returns TRUE if the Node is the root, FALSE otherwise

count Returns the number of children of a Node

totalCount Returns the total number of Nodes in the tree

path Returns a vector of mode character containing the names of the Nodes in the path from the root to this Node

pathString Returns a string representing the path to this Node, separated by backslash

levelName Returns the name of the Node, preceded by level times '*'. Useful for printing.

leafCount Returns the number of leaves are below a Node

leaves Returns a list containing all the leaf Nodes

level Returns an integer representing the level of a Node. For example, the root has level 1.

height Returns max(level) of any of the Nodes of the tree

averageBranchingFactor Returns the average number of crotches below this Node

root Returns the root Node of a Node’s tree
See Also

For more details see the data.tree documentations, or the data.tree vignette: vignette("data.tree")

Examples

```r
library(data.tree)
acme <- Node$new("Acme Inc.")
accounting <- acme$AddChild("Accounting")$AddSibling("Research")$AddChild("New Labs")$parent$AddSibling("IT")$AddChild("Outsource")

print(acme)
```

---

```r
NODE_RESERVED_NAMES_CONST

Names that are reserved by the Node class.
```

Description

These are reserved by the Node class, you cannot use these as attribute names. Note also that all fields starting with a . are reserved.

Usage

```r
NODE_RESERVED_NAMES_CONST
```

Format

An object of class character of length 40.

---

```r
plot.Node

Plot a graph, or get a graphviz dot representation of the tree
```

Description

Use these methods to style your graph, and to plot it. The functionality is built around the DiagrammeR package, so for anything that goes beyond simple plotting, it is recommended to read its documentation at http://rich-iannone.github.io/DiagrammeR/docs.html
plot.Node

Usage

```r
## S3 method for class 'Node'
plot(x, ..., direction = c("climb", "descend"),
     pruneFun = NULL, output = "graph")
```

ToDiagrammeRGraph(root, direction = c("climb", "descend"), pruneFun = NULL)

SetNodeStyle(node, inherit = TRUE, keepExisting = FALSE, ...)

SetEdgeStyle(node, inherit = TRUE, keepExisting = FALSE, ...)

SetGraphStyle(root, keepExisting = FALSE, ...)

GetDefaultTooltip(node)

Arguments

- **x**
  - The root node of the data.tree structure to plot

- **...**
  - For the SetStyle methods, this can be any stlyeName / value pair. See http://graphviz.org/Documentation.php for details. For the plot.Node generic method, this is not used.

- **direction**
  - when converting to a network, should the edges point from root to children ("climb") or from child to parent ("descend")?

- **pruneFun**
  - allows providing a a prune criteria, i.e. a function taking a Node as an input, and returning TRUE or FALSE. If the pruneFun returns FALSE for a Node, then the Node and its entire sub-tree will not be considered.

- **output**
  - a string specifying the output type; graph (the default) renders the graph using the grViz function and visNetwork renders the graph using the visnetwork function.

- **root**
  - The root Node of the data.tree structure to visualize.

- **node**
  - The Node of the data.tree structure on which you would like to set style attributes.

- **inherit**
  - If TRUE, then children will inherit this node's style. Otherwise they inherit from this node's parent. Note that the inherit always applies to the node, i.e. all style attributes of a node and not to a single style attribute.

- **keepExisting**
  - If TRUE, then style attributes are added to possibly existing style attributes on the node.

Details

Use SetNodeStyle and SetEdgeStyle to define the style of your plot. Use plot to display a graphical representation of your tree.

The most common styles that can be set on the nodes are:

- color
- fillcolor
• fixedsize true or false
• fontcolor
• fontname
• fontsize
• height
• penwidth
• shape box, ellipse, polygon, circle, box, etc.
• style
• tooltip
• width

The most common styles that can be set on the edges are:

• arrowhead e.g. normal, dot, vee
• arrowsize
• arrowtail
• color
• dir forward, back, both, none
• fontcolor
• fontname
• fontsize
• headport
• label
• minlen
• penwidth
• tailport
• tooltip

A good source to understand the attributes is http://graphviz.org/Documentation.php. Another good source is the DiagrammeR package documentation, or more specifically: http://rich-iannone.github.io/DiagrammeR/docs.html

In addition to the standard GraphViz functionality, the data.tree plotting infrastructure takes advantage of the fact that data.tree structure are always hierarchic. Thus, style attributes are inherited from parents to children on an individual basis. For example, you can set the fontcolor to red on a parent, and then all children will also have red font, except if you specifically disallow inheritance. Labels and tooltips are never inherited.

Another feature concerns functions: Instead of setting a fixed value (e.g. setNodeStyle(acme, label = "Acme. Inc")), you can set a function (e.g. setNodeStyle(acme, label = function(x) x$name)). The function must take a Node as its single argument. Together with inheritance, this becomes a very powerful tool.

The getDefaultTooltip method is a utility method that can be used to print all fields of a Node.

There are some more examples in the 'applications' vignette, see vignette('applications', package = "data.tree")
Examples

data(acme)
SetGraphStyle(acme, rankdir = "TB")
SetEdgeStyle(acme, arrowhead = "vee", color = "blue", penwidth = 2)
# per default, Node style attributes will be inherited:
SetNodeStyle(acme, style = "filled,rounded", shape = "box", fillcolor = "GreenYellow",
fontname = "helvetica", tooltip = GetDefaultTooltip)
SetNodeStyle(acme$IT, fillcolor = "LightBlue", penwidth = "5px")
# inheritance can be avoided:
SetNodeStyle(acme$Accounting, inherit = FALSE, fillcolor = "Thistle",
fontcolor = "Firebrick", tooltip = "This is the accounting department")
SetEdgeStyle(acme$Research$'New Labs',
color = "red",
label = "Focus!",
penwidth = 3,
fontcolor = "red")
# use Do to set style on specific nodes:
Do(acme$leaves, function(node) SetNodeStyle(node, shape = "egg"))
plot(acme)


Description

Print a Node in a human-readable fashion.

Usage

## S3 method for class 'Node'
print(x, ..., pruneMethod = c("simple", "dist", NULL),
limit = 100, pruneFun = NULL)

Arguments

x

The Node.

...  

Node attributes to be printed. Can be either a character (i.e. the name of a Node field), a Node method, or a function taking a Node as a single argument. See `Get` for details on the meaning of attribute.

pruneMethod

The method can be used to prune for printing in a simple way. If NULL, the entire tree is displayed. If "simple", then only the first limit nodes are displayed. If "dist", then Nodes are removed everywhere in the tree, according to their level. If pruneFun is provided, then pruneMethod is ignored.

limit

The maximum number of nodes to print. Can be NULL if the entire tree should be printed.

pruneFun

allows providing a a prune criteria, i.e. a function taking a Node as an input, and returning TRUE or FALSE. If the pruneFun returns FALSE for a Node, then the Node and its entire sub-tree will not be considered.
**Examples**

data(acme)
print(acme, "cost", "p")
print(acme, "cost", probability = "p")
print(acme, expectedCost = function(x) x$cost * x$p)
do.call(print, c(acme, acme$fieldsAll))

tree <- CreateRegularTree(4, 5)
# print entire tree:
print(tree, pruneMethod = NULL)
# print first 20 nodes:
print(tree, pruneMethod = "simple", limit = 20)
# print 20 nodes, removing leafs first:
print(tree, pruneMethod = "dist", limit = 20)
# provide your own pruning function:
print(tree, pruneFun = function(node) node$position != 2)

---

**Prune**

**Prunes a tree.**

**Description**

Pruning refers to removing entire subtrees. This function has side-effects, it modifies your data.tree structure!

**Usage**

Prune(node, pruneFun)

**Arguments**

- **node**: The root of the sub-tree to be pruned
- **pruneFun**: allows providing a a prune criteria, i.e. a function taking a Node as an input, and returning TRUE or FALSE. If the pruneFun returns FALSE for a Node, then the Node and its entire sub-tree will not be considered.

**Value**

the number of nodes removed

**See Also**

Node
**Revert**

Reverts the sort order of a node’s children.

**Usage**

```r
revert(node, recursive = TRUE)
```

**Arguments**

- `node`: the Node whose children’s sort order is to be reverted
- `recursive`: If TRUE, then revert is called recursively on all children.

**Value**

returns the Node invisibly (for chaining)

**See Also**

- `Node`
- `Sort`

---

**Set**

Traverse a Tree and Assign Values

**Description**

The method takes one or more vectors as an argument. It traverses the tree, whereby the values are picked from the vector. Also available as OO-style method on `Node`.

---

**Examples**

```r
data(acme)
acme$Do(function(x) x$cost <- Aggregate(x, "cost", sum))
Prune(acme, function(x) x$cost > 700000)
print(acme, "cost")
```
SetFormat

Usage

```
# OO-style:
# node$set(...,
#   traversal = c("pre-order", "post-order", "in-order", "level", "ancestor"),
#   pruneFun = NULL,
#   filterFun = NULL)
# traditional:
Set(nodes, ...)
```

Arguments

- **nodes**: The nodes on which to perform the Get (typically obtained via Traverse)
- ... each argument can be a vector of values to be assigned. Recycled.

Value

invisibly returns the nodes (useful for chaining)

See Also

- Node
- Get
- Do
- Traverse

Examples

```
data(acme)
acme$set(departmentId = 1:acme$totalCount, openingHours = NULL, traversal = "post-order")
acme$set(head = c("Jack Brown",
                  "Mona Moneyhead",
                  "Dr. Frank N. Stein",
                  "Eric Nerdahl" ),
       filterFun = function(x) !x$isLeaf
      )
print(acme, "departmentId", "head")
```

Description

Formatter functions set on a Node act as a default formatter when printing and using the Get method. The formatter is inherited, meaning that whenever Get fetches an attribute from a Node, it checks on the Node or on any of its ancestors whether a formatter is set.
Sort

Usage

SetFormat(node, name, formatFun)

Arguments

node The node on which to set the formatter
name The attribute name for which to set the formatter
formatFun The formatter, i.e. a function taking a value as an input, and formatting returning the formatted value

See Also

Get
print.Node

Examples

data(acme)
acme$set(id = 1:(acme$totalCount))
SetFormat(acme, "id", function(x) FormatPercent(x, digits = 0))
SetFormat(Climb(acme, "IT"), "id", FormatFixedDecimal)
print(acme, "id")
# Calling Get with an explicit formatter will overwrite the default set on the Node:
print(acme, id = acme$get("id", format = function(x) paste0("id: ", x)))

# Or, to avoid formatters, even though you set them on a Node:
print(acme, id = acme$get("id", format = identity))

Sort

Sort children of a Node or an entire data.tree structure

Description

You can sort with respect to any argument of the tree. But note that sorting has side-effects, meaning that you modify the underlying, original data.tree object structure.

Usage

Sort(node, attribute, ..., decreasing = FALSE, recursive = TRUE)
Arguments

node The node whose children are to be sorted
attribute determines what is collected. The attribute can be
  • a.) the name of a **field** or a **property/active** of each Node in the tree, e.g. acme$Get("p") or acme$Get("position")
  • b.) the name of a **method** of each Node in the tree, e.g. acme$Get("levelZeroBased"), where e.g. acme$levelZeroBased <- function() acme$level - 1
  • c.) a **function**, whose first argument must be a Node e.g. acme$Get(function(node) node$cost * node$weight)

... any parameters to be passed on the the attribute (in case it’s a method or a function)
decreasing sort order
recursive if TRUE, Sort will be called recursively on the Node’s children. This allows sorting an entire tree.

Value

Returns the node on which Sort is called, invisibly. This can be useful to chain Node methods.

See Also

Node
Revert

Examples

data(acme)
acme$Do(function(x) x$totalCost <- Aggregate(x, "cost", sum), traversal = "post-order")
Sort(acme, "totalCost", decreasing = FALSE)
print(acme, "totalCost")

---

ToNewick  *Write a data.tree structure to Newick notation*

Description

To read from Newick, you can use the ape package, and convert the resulting phylo object to a data.tree structure.

Usage

ToNewick(node, heightAttribute = DefaultPlotHeight, ...)
**Arguments**

- **node**: The root Node of a tree or sub-tree to be converted.
- **heightAttribute**: The attribute (field name, method, or function) storing or calculating the height for each Node.
- **...**: parameters that will be passed on the heightAttributeName, in case it is a function.

**See Also**

Other Conversions from Node: as.dendrogram.Node

**Examples**

```r
data(acme)
ToNewick(acme)
ToNewick(acme, heightAttribute = NULL)
ToNewick(acme, heightAttribute = function(x) DefaultPlotHeight(x, 200))
ToNewick(acme, rootHeight = 200)
```

---

**Traverse**

*Traverse a tree or a sub-tree*

**Description**

Traverse takes the root of a tree or a sub-tree, and "walks" the tree in a specific order. It returns a list of Node objects, filtered and pruned by filterFun and pruneFun.

**Usage**

```r
Traverse(node, traversal = c("pre-order", "post-order", "in-order", "level", "ancestor"), pruneFun = NULL, filterFun = NULL)
```

**Arguments**

- **node**: the root of a tree or a sub-tree that should be traversed.
- **traversal**: any of 'pre-order' (the default), 'post-order', 'in-order', 'level', 'ancestor', or a custom function (see details).
- **pruneFun**: allows providing a prune criteria, i.e. a function taking a Node as an input, and returning TRUE or FALSE. If the pruneFun returns FALSE for a Node, then the Node and its entire sub-tree will not be considered.
- **filterFun**: allows providing a filter, i.e. a function taking a Node as an input, and returning TRUE or FALSE. Note that if filter returns FALSE, then the node will be excluded from the result (but not the entire subtree).
Details

The traversal order is as follows. (Note that these descriptions are not precise and complete. They are meant for quick reference only. See the data.tree vignette for a more detailed description).

**pre-order**  Go to first child, then to its first child, etc.

**post-order** Go to the first branch’s leaf, then to its siblings, and work your way back to the root

**in-order**  Go to the first branch’s leaf, then to its parent, and only then to the leaf’s sibling

**level**  Collect root, then level 2, then level 3, etc.

**ancestor** Take a node, then the node’s parent, then that node’s parent in turn, etc. This ignores the pruneFun

**function** You can also provide a function, whose sole parameter is a Node object. The function is expected to return the node’s next node, a list of the node’s next nodes, or NULL.

Value

a list of Nodes

See Also

Node
Get
Set
Do
Index

*Topic **Newick**
  ToNewick, 48

*Topic **datasets**
  acme, 3
  mushroom, 37
  Node, 38
  NODE_RESERVED_NAMES_CONST, 40

  acme, 3
  Aggregate, 3, 25
  AreNamesUnique, 5, 7, 29, 39
  as.data.frame.Node, 5, 13, 25
  as.dendrogram.Node, 8, 25, 27, 49
  as.igraph.Node, 9
  as.list.Node, 10
  as.Node, 11, 13, 15, 16, 19, 25
  as.Node.BinaryTree, 11
  as.Node.data.frame, 11, 12, 15, 16, 19, 25
  as.Node.dendrogram, 11, 13, 14, 16, 19, 25
  as.Node.list, 11, 13, 15, 19
  as.Node.party, 17
  as.Node.phylo, 11, 13, 15, 16, 19, 20, 25, 34
  as.phylo.Node, 19, 20, 25, 27, 34
  averageBranchingFactor, 20

  CheckNameReservedWord, 21
  Climb, 21, 38, 39
  Clone, 22
  CreateRandomTree, 23
  CreateRegularTree, 24
  Cumulate, 24, 25

  data.tree, 25, 26, 40
  data.tree-package (data.tree), 25
  DefaultPlotHeight, 27
  dendrogram, 8, 14
  Distance, 28
  Do, 25, 28, 33, 39, 46, 50

  FindNode, 29, 39

  FormatFixedDecimal, 30
  FormatPercent, 31
  FromDataFrameNetwork
    (as.Node.data.frame), 12
  FromDataFrameTable
    (as.Node.data.frame), 12
  FromListExplicit (as.Node.list), 15
  FromListSimple (as.Node.list), 15
  Get, 4, 6, 25, 29, 31, 39, 46, 50
  GetAttribute, 33
  GetDefaultTooltip (plot.Node), 40
  GetPhyloNr, 19, 20, 34
  isLeaf, 35
  isNotLeaf, 35
  isNotRoot, 36
  isRoot, 36

  mushroom, 37

  Navigate, 22, 37, 39
  Node, 4, 21, 22, 25, 26, 29, 31, 33, 37, 38, 41, 42, 44–46, 48–50
  NODE_RESERVED_NAMES_CONST, 40

  plot.Node, 40
  print, 25
  print.Node, 43
  Prune, 25, 39, 44

  R6Class, 38
  Revert, 39, 45, 48

  sapply, 32
  Set, 25, 29, 33, 39, 45, 50
  SetEdgeStyle (plot.Node), 40
  SetFormat, 32, 34, 46
  SetGraphStyle (plot.Node), 40
  SetNodeStyle (plot.Node), 40
  Sort, 25, 39, 45, 47
ToDataFrameNetwork
   (as.data.frame.Node), 5
ToDataFrameTable (as.data.frame.Node), 5
ToDataFrameTree (as.data.frame.Node), 5
ToDataFrameTypeCol
   (as.data.frame.Node), 5
ToDiagrammeRGraph (plot.Node), 40
ToListExplicit (as.list.Node), 10
ToListSimple (as.list.Node), 10
ToNewick, 8, 27, 48
Traverse, 6, 25, 29, 32, 33, 46, 49