The **binarytree** package[*]

Aleksandrina Nikolova
`aayla.secura.1138@gmail.com`

July 26, 2016

Abstract

The binarytree package provides an easy but flexible interface to draw binary trees using TikZ. It uses a path specification of the form ⟨*l/r moves*⟩ :⟨*label*⟩:⟨*color*⟩:⟨*anchor*⟩!⟨*l/r moves*⟩:⟨*label*⟩:⟨*color*⟩:⟨*anchor*⟩!. . . to determine the style for each edge of the tree. It supports the TikZ `external` library and can automatically name the files based on content. The default appearance and behaviour can be customized by a range of options.

Contents

---

[*] This document corresponds to binarytree v1.01, dated 2016/07/25.

1 Introduction

The binarytree package provides a macro, `\BinaryTree`, which accepts a path specification and a number signifying the maximum tree depth. The path specification determines the style for each edge of the tree. The behaviour can be customized by calling `\btreeset` or `\btreesetexternal` with a list of ⟨*key=value*⟩ pairs. These will have effect until the end of the current group. Additionally, one can pass these keys as an optional argument to `\BinaryTree` to have them affect only that single tree.

Each node is named, so you can refer to it later. The root is named `btree-root` while each of its children is named according to its ancestors, for example `btree-l-r` is the name of the right child of the root's left child.

The package supports the TikZ `external` library. To use this feature however one needs to load this library and execute `\tikzexternalize` in the preamble. All of the externalization options passed to, for example, `\btreesetexternal` are only executed once inside the local group of each tree so there is no conflict between them and any global configuration of the library. One may therefore disable externalization globally by calling `\tikzexternaldisable`; externalization will still be enabled for all the trees of binarytree for which the `external` option is `true`.

2 Usage

`\BinaryTree` [⟨*local options*⟩]{⟨*path specification*⟩}{⟨*depth*⟩}

⟨*local options*⟩ is a coma separated list of ⟨*key=value*⟩ pairs which set options locally inside a group for this tree alone. The `external/file name` option, in case externalization is enabled, is allowed only in this context.

⟨*path specification*⟩ is of the form ⟨*l/r moves*⟩:⟨*label*⟩:⟨*color*⟩:⟨*anchor*⟩!⟨*l/r moves*⟩ :⟨*label*⟩:⟨*color*⟩:⟨*anchor*⟩!. . . . Multiple such paths can be given, separated by a coma. If an edge is visited more than once only the last *explicitly set* style will be used (i. e. only when one of ⟨*label*⟩ or ⟨*color*⟩ is given). ⟨*l/r moves*⟩ is a sequence of `l` or `r` characters which visit the corresponding (left or right) child of the current node, and draw an edge in the color ⟨*color*⟩. Upon a change in direction, the previous move may be continued until the end. ⟨*label*⟩ is placed on either the first or every child of the subpath, at the given anchor. The list of moves can be empty, in which case the label and color is applied to the previous (parent) edge. The path after an exclamation mark continues from the last child, while the path after a coma starts at the root again.

⟨*depth*⟩ is the maximum depth to which the tree will be drawn (when continuing we stop at this depth; any moves extending beyond this depth are ignored without error).

2.1 Configuring the defaults

`\btreeset` `{⟨options⟩}`
`\btreesetexternal`

Setting defaults which apply to all trees (until the end of the current group) is done by calling `\btreeset` or `\btreesetexternal`. They accept a single argument—a coma separated list of ⟨*key=value*⟩ pairs. The default key path for `\btreeset` is `/BT/` and that of `\btreesetexternal` is `/BT/external`. The `external/file name` option is *not* accepted here—it only makes sense to give it as a local option to a particular invocation of `\BinaryTree`.

2.2 Options for appearance

The following keys are defined and determine the style for all subsequent trees (until the end of the current group). For any unknown key passed to `\btreeset` (`\btreesetexternal`) or as an optional argument to `\BinaryTree`, it is checked if this key is defined for `/tikz` or `/pgf` (`/tikz/external`).

`/BT/defaults`

A shortcut for setting all appearance related keys to their default values. `separate` and any keys related to externalization are not affected.

`/BT/grow` `{⟨direction⟩}` (initial: `up`, default: `none`)

A choice of `up`, `down`, `left`, or `right` sets the growth direction of the tree and the default label anchors.

`/BT/root label anchor` `{⟨anchor⟩}` (initial: `none`, default: `none`)
`/BT/left label anchor`
`/BT/right label anchor`
`/BT/final label anchor`

These keys can override the default anchors set by `grow`. An anchor is any valid TikZ anchor, such as `above` or `north east`.

`/BT/root edge` `{⟨true | false⟩}` (initial: `false`, default: `true`)

If set to `true` a single child of the root will be drawn before the rest of the tree. The root label is still placed on the root node, not on this edge, so any anchor is relative to the bottom of this edge.

`/BT/draw missing` `{⟨true | false⟩}` (initial: `false`, default: `true`)

If set to `true` children which have not been visited in the course of constructing the path will be drawn anyway (no label, default color).

`/BT/label on every edge` `{⟨true | false⟩}` (initial: `false`, default: `true`)

If set to `true` ⟨*label*⟩ will be placed on every edge of the subpath segment (delimited by `!`) and ⟨*anchor*⟩ will apply to all of them.

`/BT/math labels` `{⟨true | false⟩}` (initial: `false`, default: `true`)

If set to `true` all labels will be wrapped in math mode.

`/BT/continue at path end` `{⟨true | false⟩}` (initial: `true`, default: `true`)

If set to `true` when the end of a subpath is reached and the current depth is less than the tree depth, the last move will be continued. Then, to explicitly make the path terminate, use an `s` move operation. When set to `false`, one can force a continuation with the `c` move operation.

`/BT/continue after turn` {⟨`true` | `false`⟩} (initial: `true`, default: `true`)

If set to `true` after a change in direction is encountered (as in `lr` or `r!l`) the previous move will be continued until the top of the tree before moving on.

`/BT/default color` {⟨*color*⟩} (initial: `black`, default: `none`)

Set the default color for edges and labels. Any color specification accepted by `\colorlet` is valid.

`/BT/default color after turn` {⟨`true` | `false`⟩} (initial: `true`, default: `true`)

If set to `true` after a change in direction is encountered the default color is used (a previously set style will not however be overridden); otherwise the previously given path color is used, which, if not, empty will cause any previously set style for the edges to be overridden.

`/BT/xscale`
`/BT/yscale` {⟨*scale factor*⟩} (initial: `1`, default: `none`)

Append an x or y scale to the current style. The effect is *accumulative*.

`/BT/scale` {⟨*scale factor*⟩} (initial: `1`, default: `none`)

This is equivalent to setting `xscale=`⟨*scale factor*⟩, `yscale=`⟨*scale factor*⟩

`/BT/label distance` {⟨*length*⟩} (initial: `10pt`, default: `none`)

Sets the distance between the label and the edge.

`/BT/sibling distance` {⟨*length*⟩} (initial: `40mm`, default: `none`)

Sets the distance between siblings *on level 1* (root's immediate children). See below.

`/BT/level distance` {⟨*length*⟩} (initial: `20mm`, default: `none`)

Sets the distance between levels *on level 1* (root's immediate children). See below.

`/BT/sibling distance scales`
`/BT/level distance scales` {⟨`true` | `false`⟩} (initial: `true`, default: `true`)

If set to `true` the sibling/level distance will be scaled by the level number on each level.

`/BT/top padding`
`/BT/bottom padding`
`/BT/left padding`
`/BT/right padding` {⟨*length*⟩} (initial: `3pt`, default: `none`)

Add an additional padding on the corresponding side. This is useful if one of the labels of intermediate outer children extends beyond the bounding box, or if the label anchor for the root (final children) is changed (in which case one would need to set a negative padding to compensate for assuming the labels are below (above) the node).

`/BT/framed` {⟨`true` | `false`⟩} (initial: `false`, default: `true`)

If set to `true` the bounding box will be drawn.

`/BT/separate` {⟨`true` | `false`⟩} (initial: `false`, default: `true`)

If set to `true` the whole tree will be wrapped in a `tikzpicture` environment, otherwise—in a `scope` environment.

2.3 Options for externalization

`/BT/external` {⟨`true` | `false`⟩} (initial: `false`, default: `true`)

If set to `true` externalization will be enabled and the following keys will have effect on how the file names. This requires one to load the `external` library of TikZ and initialize it (by calling `\tikzexternalize`). One may then disable it globally by calling `\tikzexternaldisable`, it will be enabled locally inside the group of each tree.

`/BT/external/use automatic file name` {⟨`true` | `false`⟩} (initial: `false`, default: `true`)

Generate a (not so short or friendly looking) file name from the global style and the path specification. If it is set to `false`, a figure name (by default `binary-tree_`) is used with each filename being the figure name plus a number appended at the end, indicating the order in which the trees are encountered in the document. This option is useful if one regularly changes the order of the trees and wishes to avoid having to recompile them. Under some circumstances different looking trees may end up with the same name (for example if math labels are used, which cannot be safely expanded into text and are thus ignored). In these cases one may either locally disable this option or specify a file name explicitly by either calling `\tikzsetnextfilename` or by using the following key.

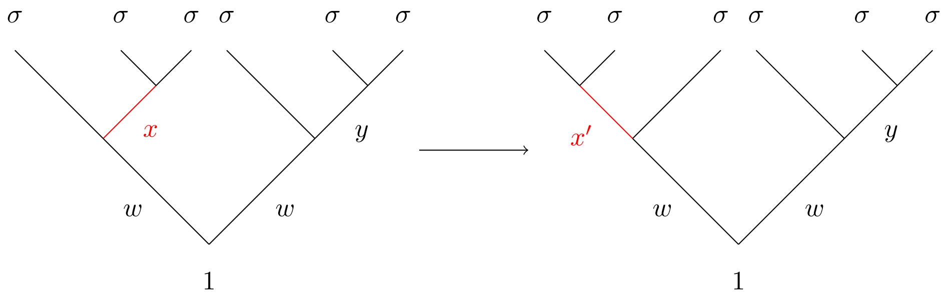
`/BT/external/file name` {⟨*file name*⟩} (initial: `none`, default: `none`)

Explicitly set the file name for the currently drawn tree regardless of whether automatic file naming is enabled or not. This key is only accepted as a local option to each `\BinaryTree`.

Additionally, any unknown keys defined for `/tikz/external` are accepted, collected and will be executed once we're inside the local group for each tree. So if one wishes to, one can, without a conflict, set global options for externalization (by calling for example `\tikzset`⟨*key=value*⟩), which will be used by all other TikZ figures and by binarytree by default, and different settings (via `\btreesetexternal` or local options to `\BinaryTree`) for the binarytree figures.

3 Examples

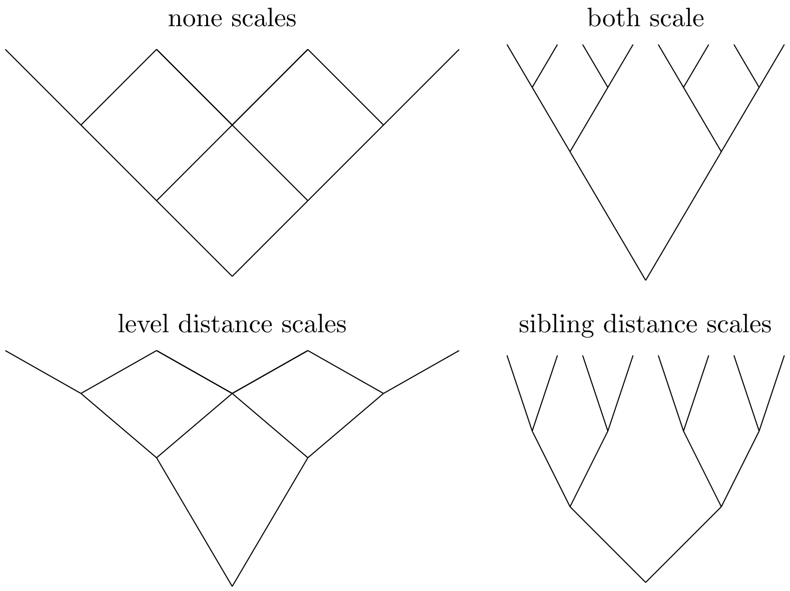
- Each node has a unique name, but you can name the entire tree using TikZ 's `local bounding box key` and refer to it for example in a graph.



```
\documentclass{standalone}
\usepackage{binarytree}
\usetikzlibrary{graphs}

\begin{document}
\btreeset{math labels,scale=0.7}
\begin{tikzpicture}
  \BinaryTree[local bounding box=INIT]{%
    :1!l:w!r:x:red!l:\sigma,ll!l:\sigma,lr!r:\sigma,
    r:w!r:y!r:\sigma,rr!l:\sigma,rl!l:\sigma}{3}
  \BinaryTree[local bounding box=FINAL,xshift=10cm]{
    :1!l:w!l:x’:red!l:\sigma,lr!r:\sigma,ll!r:\sigma,
    r:w!r:y!r:\sigma,rr!l:\sigma,rl!l:\sigma}{3}
  \graph[use existing nodes]{ INIT -> FINAL};
\end{tikzpicture}
\end{document}
```

- Here is a simple example which shows how the scaling of the sibling or level distance affects the appearance of the tree. The trees are otherwise identical.



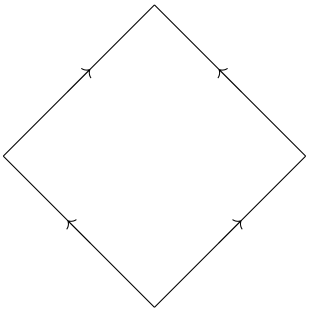
```
\documentclass{standalone}
\usepackage{binarytree}

\begin{document}
```

```
\btreeset{%
  draw missing,%
  separate,%
  level distance scales=false,%
  sibling distance scales=false,%
  scale=0.5}

\def\arraystretch{1.5}
\begin{tabular}{cc}
  none scales & both scale \\
  \BinaryTree{}{3} &
  \BinaryTree[yscale=1.7,level distance scales,%
                        sibling distance scales]{}{3} \\
  level distance scales & sibling distance scales \\
  \BinaryTree[yscale=1.7,level distance scales]{r}{3} &
  \BinaryTree[sibling distance scales]{}{3}
\end{tabular}
\end{document}
```

- You have control over which edges are drawn implicitly. Surely there is another way to do this in Ti*k*Z!



```
\documentclass{standalone}
\usepackage{binarytree}

\begin{document}
\btreeset{%
  math labels,%
  separate,%
  level distance scales=false,%
  sibling distance scales=false,%
  continue after turn=false,%
  continue at path end=false,%
  left label anchor=center,%
  right label anchor=center}

\BinaryTree{l:\nwarrow!r:\nearrow,r:\nearrow!l:\nwarrow}{3}
\end{document}
```

- The next example simply aims to demonstrate that placing two identical trees, and using the externalization and automatic file naming features, one would only compile the tree once. You need to manually compile the `makefile` and the next time the example is compiled, the figures will be included.

examples/binarytree-ex4.pdf

```
\documentclass{standalone}
\usepackage{binarytree}
\usetikzlibrary{external}
\tikzexternalize
\tikzexternaldisable

\begin{document}
\btreesetexternal{%
  use automatic file name,%
  mode=list and make}
\btreeset{%
  top padding=0pt,%
  left padding=0pt,%
  right padding=0pt,%
  bottom padding=0pt,%
  external=false,separate=true,%
  label on every edge,%
  left label anchor=center,%
  right label anchor=center,%
  continue after turn=false,%
  continue at path end=false,%
  level distance scales=false,%
  sibling distance scales=false,%
  scale=0.7}

\BinaryTree{lrr:x,llrr:x,rll:x,rrll:x}{5}%
\BinaryTree{lrr:x,llrr:x,rll:x,rrll:x}{5}
\end{document}
```

---

4 Known Issues

- Calculation of the bounding box uses the default (for the chosen growth direction) label anchors instead of any explicitly set anchors. It also does not take into account the width of the labels of any intermediate (outer) or final middle children.

- Spaces between the semicolon and the color/anchor cannot be handled.

- The automatically generated file name may not be unique: two or more differently styled trees may end up with the same name, thus only the one that was compiled first will be used. On the other hand, two trees that *are* the same may end up with different names if the subpaths are given in a different order.

5 Planned changes

- Options to control how a file name is generated:
  - use figure name instead of the generated file name for figures where an unknown keys have been used
  - use figure name instead of the generated file name for figures where math mode for labels has been used
  - force writing the expanded label into the file name, even if math mode is set—will only work if the label expands to text without special characters; at user's discretion
  - configure how detailed the generated file name is: whether or not to ignore labels, distances, scales, etc.

- Option to specify label position along the edge.

6 Implementation

The tree is drawn in a simple recursive way, which creates exactly two child nodes for each previous node, up to a maximum depth of `\BT@max@depth`. The entire appearance is controlled by a global style `/tikz/binary tree` (which is determined by the setting of various pgf keys) as well as individual styles for each edge, which are created on the fly as the path specification is processed. Each style lives under the `/tikz/binary tree` directory and is named according to the unique place of the child in the tree, by appending `l@` or `r@` to the name of its parent. The root is called `@` and has a depth of 0, its immediate children are `@r@` and `@l@` on level 1, and so on. Each node is also named similarly, for example `btree-l-l-r` or `btree-root`.

We create template styles for each type of node: root, intermediate labelled, final labelled, unlabelled, or unvisited. When setting a style for a particular edge, the relevant template is used and a macro with the name `\`⟨*child name*⟩ `styled` is defined (let to `\relax`). This is so we don't override a previously set style unless one of ⟨*label*⟩, ⟨*color*⟩ or ⟨*anchor*⟩ is given explicitly. It is also used when drawing the tree to check if such a style exists, otherwise the style is set to `/tikz/binary tree/default`.

Ti*k*Z would put unnecessary white space around the tree (especially if there are missing outer children), so we need to calculate the actual size of the tree and clip the bounding box accordingly: we need the left and right width, height and depth, since these will be coordinates for the clipping rectangle. It is mostly calculated on the fly, starting with 0pt for each 4 coordinates. If `draw missing` is `true`, we set the bounding box size to the maximum for this depth (and don't change it further). Otherwise we proceed in the following way: the height is easy to deal with—we add a level distance to the height each time a new level is created (we keep track by defining a macro `\level` ⟨*depth*⟩ `exists`). The width is trickier—consider for example the case where there are no children on the right, but there are right children to one of the left nodes; they may extend to the right side of the box. What is done is we keep track of a current x coordinate (which is relative to root, being negative on the left) and each time any left (right) edge is created, we compare the (magnitude of the) coordinate to the current left (right) width of the box, and if greater—set the box width to it.

Additionally, we save the height (width) of the root's label since this will (may) add to the depth (width) of the box, but we do not add it to the size of the box just yet (since the root label may change several times). Furthermore, we keep track of the maximum height (width) of any of the final children's labels since these will increase the "height"[1] of the box when the tree is growing vertically (sideways). We also save the width (height) of the left- and rightmost final children, as these will increase the "width" of the box when growing vertically (sideways). At the end all of these are used to increase the box (still assuming the tree is growing upwards). Finally, the four sides are cycled according to the actual growing direction.

---

[1] Here "height" means height if the tree is growing up, depth if it's growing down, left/right width if it's growing left/right; and similarly for "width"

6.1 Allocating registers

`\BT@bbox@r@width`
`\BT@bbox@l@width`
`\BT@bbox@height`
`\BT@bbox@depth`
`\BT@bbox@current@x`
`\BT@root@width`
`\BT@root@height`
`\BT@max@final@width`
`\BT@max@final@height`
`\BT@l@final@width`
`\BT@l@final@height`
`\BT@r@final@width`
`\BT@r@final@height`
`\BT@bbox@padding@top`
`\BT@bbox@padding@bottom`
`\BT@bbox@padding@left`
`\BT@bbox@padding@right`
`\BT@sibling@distance`
`\BT@level@distance`
`\BT@label@distance`

The following lengths are used in calculating the bounding box for clipping the final picture.

```
\newdimen\BT@bbox@r@width
\newdimen\BT@bbox@l@width
\newdimen\BT@bbox@height
\newdimen\BT@bbox@depth
\newdimen\BT@bbox@current@x
\newdimen\BT@root@width
\newdimen\BT@root@height
\newdimen\BT@max@final@width
\newdimen\BT@max@final@height
\newdimen\BT@l@final@width
\newdimen\BT@l@final@height
\newdimen\BT@r@final@width
\newdimen\BT@r@final@height
\newdimen\BT@bbox@padding@top
\newdimen\BT@bbox@padding@bottom
\newdimen\BT@bbox@padding@left
\newdimen\BT@bbox@padding@right
\newdimen\BT@sibling@distance
\newdimen\BT@level@distance
\newdimen\BT@label@distance
```

`\ifBT@root@edge`
`\ifBT@draw@missing`
`\ifBT@label@every@edge`
`\ifBT@math@labels`
`\ifBT@continue@at@end`
`\ifBT@continue@after@turn`
`\ifBT@default@color@after@turn`
`\ifBT@sibling@distance@scales`
`\ifBT@level@distance@scales`
`\ifBT@framed`
`\ifBT@separate`
`\ifBT@external`
`\ifBT@auto@file@name`

The following `\if`'s are set by `pgfkeys` when the corresponding keys are used.

```
\newif\ifBT@root@edge
\newif\ifBT@draw@missing
\newif\ifBT@label@every@edge
\newif\ifBT@math@labels
\newif\ifBT@continue@at@end
\newif\ifBT@continue@after@turn
\newif\ifBT@default@color@after@turn
\newif\ifBT@sibling@distance@scales
\newif\ifBT@level@distance@scales
\newif\ifBT@framed
\newif\ifBT@separate
\newif\ifBT@external
\newif\ifBT@auto@file@name
```

6.2 Keys, styles and defaults

6.2.1 For internal use

`\BT@key@is@tikz@or@error`
`T@key@is@tikz@external@or@error`
`/BT/.unknown`
`/BT/external/.unknown`

`\BT@key@is@tikz@or@error` and `\BT@key@is@tikz@external@or@error` are called from the `.unknown` handlers in `/BT/` and `/BT/external/` pgf directories respectively, and check if the key is defined for `/tikz/` or `/pgf/`, or `/tikz/external`.
`#1`: ⟨*key basename*⟩, `#2`: ⟨*value*⟩

```
\def\BT@key@is@tikz@or@error#1#2{%
```

```
  \tikzset{binary tree/.append code={%
      \pgfkeys{/tikz/#1/.try=#2,%
        /pgf/#1/.lastretry=#2}}}}
\def\BT@key@is@tikz@external@or@error#1#2{%
  \tikzset{binary tree/externalize/.append code={%
      \pgfkeys{/tikz/external/#1=#2}}}}
\pgfkeys{/BT/.cd,%
  .unknown/.code={%
    \expandafter\BT@key@is@tikz@or@error\expandafter{%
      \pgfkeyscurrentname}{#1}},%
  external/.unknown/.code={%
    \expandafter\BT@key@is@tikz@external@or@error\expandafter{%
      \pgfkeyscurrentname}{#1}}}
```

6.2.2 The user interface

`\@btreeset`
`\btreeset`
`\btreesetexternal`

The user should use `\btreeset` and `\btreesetexternal` to customize the default tree style. Some keys however only make sense for a single invocation of `\BinaryTree`, so we do not allow these to be set via `\btreeset` or `\btreesetexternal`

```
\def\@btreeset#1#2{%
  \pgfkeyssavekeyfilterstateto{\BT@restore@pgf@key@filter}%
```

`local option outside of scope` will handle only keys which are defined and belong to family `local options`, so that `.unknown` and `external/.unknown` will still handle undefined keys.

```
  \pgfkeys{/pgf/key filters/not/.install key filter={%
      /pgf/key filters/and={/pgf/key filters/active families}{%
        /pgf/key filters/defined}},%
    /BT/local option outside of scope/.install key filter handler}%
  \pgfqkeysactivatesinglefamilyandfilteroptions{/BT/local options}{#1}{#2}%
  \BT@restore@pgf@key@filter}
\def\btreeset{\@btreeset{/BT}}
\def\btreesetexternal{\@btreeset{/BT/external}}
```

`/BT/local options`
`T/local option outside of scope`

```
\pgfkeys{/BT/.cd,%
  local options/.is family,%
  local option outside of scope/.code={%
    \PackageError{binarytree}{'\pgfkeyscurrentkey' only allowed %
      locally\MessageBreak for each \string\BinaryTree}{}}}
```

6.2.3 Keys for customizing the tree appearance

The following macros are used in the calculation of the bounding box size.

`\BT@set@max@l@or@r@bbox@size`

Increase the bounding box on the left or right side so it accommodates all children on that side.

```
\def\BT@set@max@l@or@r@bbox@size#1{%
```

```
  \BT@bbox@current@x=\z@\relax%
  \@tempcnta=\BT@root@depth\relax%
  \loop%
    \ifnum\@tempcnta < \numexpr\BT@max@depth\relax%
    \advance\@tempcnta\@ne\relax%
    \BT@check@if@new@level\@tempcnta%
    \csname BT@#1@xshift\endcsname\@tempcnta%
    \csname BT@update@bbox@#1@width\endcsname%
  \repeat}%
```

`\BT@bbox@scale@labels`
`\BT@bbox@scale@padding`

When TikZ applies the scaling set by the user, the labels, label distance and the paddings will not be scaled, so we divide them by the scale here.

```
\def\BT@bbox@scale@labels{%
  \pgfmathparse{\BT@root@height / \BT@yscale}%
  \BT@root@height\pgfmathresult pt\relax%
  \pgfmathparse{\BT@root@width / \BT@xscale}%
  \BT@root@width\pgfmathresult pt\relax%
  \pgfmathparse{\BT@max@final@height / \BT@yscale}%
  \BT@max@final@height\pgfmathresult pt\relax%
  \pgfmathparse{\BT@max@final@width / \BT@xscale}%
  \BT@max@final@width\pgfmathresult pt\relax%
  \pgfmathparse{\BT@l@final@width / \BT@xscale}%
  \BT@l@final@width\pgfmathresult pt\relax%
  \pgfmathparse{\BT@r@final@width / \BT@xscale}%
  \BT@r@final@width\pgfmathresult pt\relax}%
\def\BT@bbox@scale@padding{%
  \pgfmathparse{\BT@bbox@padding@top / \BT@yscale}%
  \BT@bbox@padding@top\pgfmathresult pt\relax%
  \pgfmathparse{\BT@bbox@padding@bottom / \BT@yscale}%
  \BT@bbox@padding@bottom\pgfmathresult pt\relax%
  \pgfmathparse{\BT@bbox@padding@left / \BT@yscale}%
  \BT@bbox@padding@left\pgfmathresult pt\relax%
  \pgfmathparse{\BT@bbox@padding@right / \BT@yscale}%
  \BT@bbox@padding@right\pgfmathresult pt\relax}
```

`\BT@bbox@add@labels`

`\BT@bbox@add@labels` increases the height, depth, right and left width of the bounding box assuming an upward growing tree. It scales the label sizes and adds them to the bounding box, which at this point accommodates only the tree.

```
\def\BT@bbox@add@labels{%
  \BT@bbox@scale@labels%
  \pgfmathparse{\BT@label@distance / \BT@yscale}%
  \ifdim\BT@root@height > \z@\relax%
    \advance\BT@bbox@depth\dimexpr\BT@root@height%
    + \pgfmathresult pt\relax\fi%
  \ifdim\BT@max@final@height > \z@\relax%
    \advance\BT@bbox@height\dimexpr\BT@max@final@height%
    + \pgfmathresult pt\relax\fi%
  \advance\BT@bbox@l@width\dimexpr\BT@l@final@width / 2\relax%
  \advance\BT@bbox@r@width\dimexpr\BT@r@final@width / 2\relax%
```

If the width of the root label is more than the width of the bounding box, use it instead.

```
  \BT@set@dim@to@greater\BT@bbox@l@width{\BT@root@width / 2}%
  \BT@set@dim@to@greater\BT@bbox@r@width{\BT@root@width / 2}}
```

`\BT@bbox@add@padding`

`\BT@bbox@add@padding` simply adds the padding.

```
\def\BT@bbox@add@padding{%
  \BT@bbox@scale@padding%
  \advance\BT@bbox@height\dimexpr\BT@bbox@padding@top\relax%
  \advance\BT@bbox@depth\dimexpr\BT@bbox@padding@bottom\relax%
  \advance\BT@bbox@l@width\dimexpr\BT@bbox@padding@left\relax%
  \advance\BT@bbox@r@width\dimexpr\BT@bbox@padding@right\relax}
```

`/BT/grow`
`\BT@adjust@bbox@sides`
`\BT@grow@direction`

The `grow` key determines the growth direction, sets the default anchors for the labels and defines the macro `\BT@adjust@bbox@sides` which is called right before drawing the tree to compute the coordinates of the bounding box. Since `\BT@bbox@add@labels` calculates the sides of the box as if the growth direction is `up` we need to further transform them according to the actual direction. We call `\BT@bbox@add@padding` at the very end, since these paddings are absolute (independent of direction).

`\BT@grow@direction` is used for generating the file name only.

```
\pgfkeys{/BT/.cd,%
  grow/.is choice,%
  grow/up/.code={\def\BT@adjust@bbox@sides{%
      \BT@bbox@add@labels%
      \BT@bbox@add@padding}%
    \def\BT@grow@direction{up}%
    \pgfkeys{/BT/.cd,%
      /tikz/binary tree/.append style={grow=up},
      root label anchor=below,
      left label anchor=north east,
      right label anchor=north west,
      final label anchor=above}},%
```

Growing down is the same as growing up, but the depth and height of the bounding box need to be swapped.

```
  grow/down/.code={\def\BT@adjust@bbox@sides{%
      \BT@bbox@add@labels
      \BT@swap@dims\BT@bbox@depth\BT@bbox@height%
      \BT@bbox@add@padding}%
    \def\BT@grow@direction{down}%
    \pgfkeys{/BT/.cd,%
      /tikz/binary tree/.append style={grow'=down},
      root label anchor=above,
      left label anchor=south east,
      right label anchor=south west,
      final label anchor=below}},%
```

Growing left means that the x and y scales, as well as the heights and widths of each of the relevant labels, should be swapped *before* calling `\BT@bbox@add@labels`.

```
grow/left/.code={\def\BT@adjust@bbox@sides{%
    \BT@swap@hooks\BT@xscale\BT@yscale%
    \BT@swap@dims\BT@root@width\BT@root@height%
    \BT@swap@dims\BT@max@final@width\BT@max@final@height%
    \BT@swap@dims\BT@l@final@width\BT@l@final@height%
    \BT@swap@dims\BT@r@final@width\BT@r@final@height%
    \BT@bbox@add@labels%
```

Then, cycle the sides of the box according to: depth → right width, right width → height, height → left width, left width → depth.

```
    \@tempdima\BT@bbox@depth\relax%
    \BT@bbox@depth\BT@bbox@l@width\relax%
    \BT@bbox@l@width\BT@bbox@height\relax%
    \BT@bbox@height\BT@bbox@r@width\relax%
    \BT@bbox@r@width\@tempdima\relax%
    \BT@bbox@add@padding}%
  \def\BT@grow@direction{left}%
  \pgfkeys{/BT/.cd,%
    /tikz/binary tree/.append style={grow=left},
    root label anchor=right,
    left label anchor=north west,
    right label anchor=south west,
    final label anchor=left}},%
```

As in the `left` case, swap the x and y scales, widths and heights of labels and cycle the sides in the same order but additionally swap height and depth. This is in fact equivalent to swapping height and right width, depth and left width.

```
grow/right/.code={\def\BT@adjust@bbox@sides{%
    \BT@swap@hooks\BT@xscale\BT@yscale%
    \BT@swap@dims\BT@root@width\BT@root@height%
    \BT@swap@dims\BT@max@final@width\BT@max@final@height%
    \BT@swap@dims\BT@l@final@width\BT@l@final@height%
    \BT@swap@dims\BT@r@final@width\BT@r@final@height%
    \BT@bbox@add@labels%
    \BT@swap@dims\BT@bbox@depth\BT@bbox@l@width%
    \BT@swap@dims\BT@bbox@height\BT@bbox@r@width%
    \BT@bbox@add@padding}%
  \def\BT@grow@direction{right}%
  \pgfkeys{/BT/.cd,%
    /tikz/binary tree/.append style={grow'=right},
    root label anchor=left,
    left label anchor=north east,
    right label anchor=south east,
    final label anchor=right}},%
```

`/BT/root label anchor`
`/BT/left label anchor`
`/BT/right label anchor`
`/BT/final label anchor`

These can be used to override the default anchors set by `grow`. They of course need to be called after this key has been set.

```
root label anchor/.initial=,%
left label anchor/.initial=,%
right label anchor/.initial=,%
```

```
final label anchor/.initial=,%
```

`/BT/root edge`
`\BT@effective@level`

The way the root edge is drawn is a single child is added to it, which offsets the level numbering by one. Since we don't want the overall size of the tree to decrease, we define `\BT@effective@level` to subtract 1 (for levels > 1).

```
root edge/.is if={BT@root@edge},%
root edge/.append code={%
  \ifBT@root@edge%
    \def\BT@effective@level##1{%
      \ifnum\numexpr##1\relax < \numexpr 3\relax\expandafter\@firstoftwo%
      \else\expandafter\@secondoftwo\fi{1}{(##1-1)}}%
  \else\def\BT@effective@level##1{##1}\fi},%
```

`\BT@set@min@bbox@size`
`/BT/draw missing`
`/tikz/binary tree/default`

`draw missing` key chooses between `/tikz/missing` and unlabelled edge (in the default color) as the default style for children which have not been visited. If set to `true`, then we need to set the bounding box size to accommodate all of the children. Then we may define the `\BT@[lr]@xshift`, `\BT@update@bbox@[lr]@width` macros to empty, we don't need them anymore. `\BT@set@min@bbox@size` will eitherway be called before setting the styles.

```
draw missing/.is if={BT@draw@missing},%
draw missing/.append code={%
  \ifBT@draw@missing%
    \def\BT@set@min@bbox@size{\BT@set@max@l@or@r@bbox@size{l}%
      \BT@set@max@l@or@r@bbox@size{r}%
      \let\BT@l@xshift\@gobble%
      \let\BT@r@xshift\@gobble%
      \let\BT@update@bbox@l@width\relax%
      \let\BT@update@bbox@r@width\relax}%
    \tikzset{binary tree/default/.style={binary tree/empty={BT@default}}}%
  \else\let\BT@set@min@bbox@size\relax%
    \tikzset{binary tree/default/.style={missing}}\fi},%
```

`/BT/label on every edge`

If this key is `true`, then the label specified for a subpath will be applied to every edge.

```
label on every edge/.is if={BT@label@every@edge},%
```

`/BT/math labels`
`\BT@wrap@label`

`\BT@wrap@label` wraps the label in math mode if `math labels` is `true`.

```
math labels/.is if={BT@math@labels},%
math labels/.append code={%
  \ifBT@math@labels\def\BT@wrap@label##1{\(##1\)}%
  \else\def\BT@wrap@label##1{##1}\fi},%
```

`/BT/continue at path end`

This determines if we stop or continue the last move when the end of a subpath is reached.

```
continue at path end/.is if={BT@continue@at@end},%
```

`/BT/continue after turn`

This determines if we stop or continue after a turn is made.

```
continue after turn/.is if={BT@continue@after@turn},%
```

`/BT/default color`

This key sets `BT@default`—the default color for the tree.

```
default color/.code={\colorlet{BT@default}{#1}},%
default color/.value required,%
```

`/BT/default color after turn`

This key determines if the default color will be used when continuing a path after a turn.

```
default color after turn/.is if={BT@default@color@after@turn},%
```

`/BT/xscale`
`/BT/yscale`
`/BT/scale`
`\BT@xscale`
`\BT@yscale`

The x and y scale are simply appended to the `/tikz/binary tree` style. `\BT@xscale` and `\BT@yscale` are used in calculating the bounding box size.

```
xscale/.store in=\BT@xscale,%
xscale/.append style={/tikz/binary tree/.append style={xscale=#1}},%
xscale/.value required,%
yscale/.store in=\BT@yscale,%
yscale/.append style={/tikz/binary tree/.append style={yscale=#1}},%
yscale/.value required,%
scale/.forward to=/BT/xscale,%
scale/.forward to=/BT/yscale,%
```

`/BT/label distance`
`/BT/sibling distance`
`/BT/level distance`

These keys simply set the corresponding lengths

```
label distance/.code={\BT@label@distance=\dimexpr#1\relax},%
label distance/.value required,%
sibling distance/.code={\BT@sibling@distance=\dimexpr#1\relax},%
sibling distance/.value required,%
level distance/.code={\BT@level@distance=\dimexpr#1\relax},%
level distance/.value required,%
```

`/BT/sibling distance scales`
`/BT/level distance scales`
`\BT@current@sibling@distance`
`\BT@current@level@distance`

If set to `true` we scale the corresponding distance by the current level number. Note that `\BT@current@sibling@distance` and `\BT@current@level@distance` expand either to a length register, or to an expression understood by `\pdfmathparse`, so they can only be used inside `\pgfmathparse`.

```
sibling distance scales/.is if={BT@sibling@distance@scales},%
sibling distance scales/.append code={%
  \ifBT@sibling@distance@scales%
    \def\BT@current@sibling@distance##1{%
      \BT@sibling@distance/(\BT@effective@level{##1})}%
  \else\def\BT@current@sibling@distance##1{\BT@sibling@distance}\fi},%
level distance scales/.is if={BT@level@distance@scales},%
level distance scales/.append code={%
  \ifBT@level@distance@scales%
    \def\BT@current@level@distance##1{%
      \BT@level@distance/(\BT@effective@level{##1})}%
  \else\def\BT@current@level@distance##1{\BT@level@distance}\fi},%
```

`/BT/top padding`
`/BT/bottom padding`
`/BT/left padding`
`/BT/right padding`

These keys set the padding on each side of the bounding box.

```
top padding/.code={\BT@bbox@padding@top=#1\relax},%
top padding/.value required,%
bottom padding/.code={\BT@bbox@padding@bottom=#1\relax},%
bottom padding/.value required,%
left padding/.code={\BT@bbox@padding@left=#1\relax},%
left padding/.value required,%
right padding/.code={\BT@bbox@padding@right=#1\relax},%
right padding/.value required,%
```

`/BT/framed` If set to `true` the bounding box will be drawn.

```
framed/.is if={BT@framed},%
```

`/BT/defaults` We define a single key which sets the default values for all keys related to the appearance. We do not set defaults for keys related to externalization since the user will likely want to set those globally for the entire document. This is meant to be a shortcut to simply reset the styles for the current group or tree.

```
defaults/.style={/BT/.cd,%
  default color=black,%
  default color after turn=true,%
  grow=up,%
  root edge=false,%
  draw missing=false,
  label on every edge=false,%
  math labels=false,%
  continue at path end=true,%
  continue after turn=true,%
  scale=1,%
  sibling distance=40mm,%
  level distance=20mm,%
  sibling distance scales=true,%
  level distance scales=true,%
  label distance=10pt,%
  top padding=3pt,%
  bottom padding=3pt,%
  left padding=3pt,%
  right padding=3pt,%
  framed=false},%
defaults/.value forbidden,%
```

6.2.4 Keys related to externalization

`/BT/separate` If set to `true` the code for the tree will be put in a `tikzpicture` environment, otherwise—in a `scope` environment.

```
separate/.is if={BT@separate},%
separate/.append code={%
  \ifBT@separate\else\pgfkeys{/BT/external=false}\fi},%
```

`/BT/external` If set to `true` `separate` will also be enabled. Additionally, right before drawing the tree, `\tikzexternalenable` will be run and all the `/tikz/external` keys that the user has passed will be set.

```
external/.is if={BT@external},%
external/.append code={%
```

```
    \ifBT@external\pgfkeys{/BT/separate=true}\fi}}
```

`/BT/use automatic file name`
`/BT/file name`

If `external/file name` is set, this will be used as the file name for the figure. Otherwise, if `external/use automatic file name` is `true` the automatically generated file name will be used (a figure with the same file name may already be present, this may or may not be what the user intended); if it is `false` the figure name (the default one or the one set by the user) will be used with a unique number automatically appended.

```
\pgfkeys{/BT/external/.cd,%
  use automatic file name/.is if={BT@auto@file@name},%
  file name/.style={/tikz/binary tree/externalize/.append code={%
      \tikzsetnextfilename{#1}}},%
  file name/.belongs to family=/BT/local options,%
  file name/.value required}
```

6.2.5 Setting default values

`/tikz/binary tree`
`/tikz/binary tree/externalize`

The `/tikz/binary tree` style is applied to the scope of the tree. `/tikz/binary tree/externalize` is run right before entering the scope and it sets the given `/tikz/external` keys. It also sets the default figure name to `binary-tree_`.

```
\tikzset{binary tree/.style={%
    level/.style={level distance=\BT@current@level@distance{##1},%
      sibling distance=\BT@current@sibling@distance{##1}},%
    parent anchor=center,child anchor=center,%
    label distance=\BT@label@distance,every node/.style={outer sep=0pt,%
      inner sep=0pt}},%
  binary tree/externalize/.code={%
    \tikzexternalenable\tikzsetfigurename{binary-tree_}}}
```

Now we set the defaults.

```
\pgfqkeys{/BT}{%
  defaults,%
  separate=false,%
  external=false,%
  external/use automatic file name=false}
```

6.2.6 TikZ styles for child nodes

The `\BT@edge@label` and `\BT@edge@no@label` macros are used by the TikZ child styles as `edge from parent macro`.

`\BT@edge@label`

#1: ⟨*label*⟩, #2: ⟨*anchor*⟩, #3: ⟨*color*⟩, #4–5: ⟨*options and node specifications*⟩ (ignored)

```
\def\BT@edge@label#1#2#3#4#5{%
  [style=edge from parent, color=#3]%
    (\tikzparentnode\tikzparentanchor) --%
      node[anchor=#2,inner sep=\BT@label@distance/2] {#1}%
        (\tikzchildnode\tikzchildanchor)}
```

`\BT@edge@no@label` #1: ⟨*color*⟩, #2–3: ⟨*options and node specifications*⟩ (ignored)

```
\def\BT@edge@no@label#1#2#3{%
  [style=edge from parent, color=#1]%
    (\tikzparentnode\tikzparentanchor) --%
      (\tikzchildnode\tikzchildanchor)}
```

`/tikz/binary tree/root`
`/tikz/binary tree/child`
`/tikz/binary tree/final child`

The `root`, `child` and `final child` styles draw the edges for the root and *labelled* child nodes. The `root` style additionally sets the empty style which has effect only when it is passed to a `child` operation (where the `label` key has no effect); this is how the root label and the edge from it are drawn in the same color when `root edge` is `true`—the root style is passed to both the root node and its single immediate child.
#1: ⟨*label*⟩, #2: ⟨*label anchor*⟩, #3: ⟨*color*⟩

```
\tikzset{binary tree/.cd,%
  root/.style n args={3}{%
    label={[text=#3]#2:#1},binary tree/empty={#3}},%
  root/.value required,%
  child/.style n args={3}{%
    edge from parent macro=\BT@edge@label{#1}{#2}{#3},%
    every child node/.style={}},%
  child/.value required,%
  final child/.style n args={3}{%
    edge from parent macro=\BT@edge@no@label{#3},%
    every child node/.style={label={[text=#3]#2:#1}}},%
  final child/.value required,%
```

`/tikz/binary tree/empty` The `empty` style is for unlabelled child nodes.
#1: ⟨*color*⟩

```
  empty/.style={%
    edge from parent macro=\BT@edge@no@label{#1},%
    every child node/.style={}},%
  empty/.value required}
```

6.3 General helper macros

`\BT@message` When modifying the code, `\BT@message` is used to printing debugging information (comment out the second line).

```
\def\BT@message#1{\pgfinterruptpicture #1\par\endpgfinterruptpicture}
\def\BT@message#1{}
```

`\BT@gobble@till@nil` Gobble all tokens until the next `\@nil` including.

```
\def\BT@gobble@till@nil#1\@nil{}
```

`\BT@endgroup@let` End the current group and define token `#1` to be the expansion of token `#2`.

```
\def\BT@endgroup@let#1#2{%
  \expandafter\endgroup\expandafter\def\expandafter#1\expandafter{#2}}
```

`\BT@app@to@hook`
`\BT@eapp@to@hook`

`\BT@app@to@hook` appends `#2`, as is, to an already defined (parameterless) macro `#1`, while `\BT@eapp@to@hook` fully expands `#2`.

```
\def\BT@app@to@hook#1#2{\expandafter\def\expandafter#1\expandafter{#1#2}}
\def\BT@eapp@to@hook#1#2{\edef#1{\unexpanded\expandafter{#1}#2}}
```

`\BT@if`

If the TEX boolean named `#1` is `true`, expands to ⟨*true*⟩, otherwise to ⟨*false*⟩.

```
\def\BT@if#1{\csname if#1\endcsname\expandafter\@firstoftwo%
  \else\expandafter\@secondoftwo\fi}
```

`\BT@if@blank`
`\BT@if@blank@i`

If `#1` is empty or consists of spaces only, expands to ⟨*true*⟩, otherwise to ⟨*false*⟩. No expansion of the first argument is performed.

`#1`: ⟨*tokens*⟩, `#2`: ⟨*true*⟩, `#3`: ⟨*false*⟩

```
\def\BT@if@blank#1{\BT@if@blank@i#1\@nil}
\def\BT@if@blank@i#1{\ifx\@nil#1\expandafter\@firstoftwo%
  \else\expandafter\expandafter\expandafter\@secondoftwo\expandafter%
    \BT@gobble@till@nil\fi}
```

`\BT@strip@prefix`

Remove `#1` from beginning of replacement text of macro `#2`.

```
\def\BT@strip@prefix#1#2{%
  \begingroup%
  \def\@@suffix#1##1\@nil{##1}%
  \edef#2{\unexpanded\expandafter\expandafter\expandafter{%
      \expandafter\@@suffix#2\@nil}}%
  \BT@endgroup@let#2#2}
```

`\BT@set@to@dim`
`\BT@set@to@width`
`\BT@set@to@height`
`\BT@set@to@total@height`

"`settoheight`" and friends in plain TEX. It does not work for the scratch registers (`\dimen`0–9).

`#1`: ⟨`\ht` | `\wd` | `\dp`⟩, `#2`: ⟨*TEX length register*⟩, `#3`: ⟨*content*⟩

```
\def\BT@set@to@dim#1#2#3{%
  \begingroup%
  \setbox0\hbox{#3}%
  \expandafter\endgroup\expandafter#2\the#10\relax}
\def\BT@set@to@width{\BT@set@to@dim\wd}
\def\BT@set@to@height{\BT@set@to@dim\ht}
\def\BT@set@to@depth{\BT@set@to@dim\dp}
\def\BT@set@to@total@height#1#2{%
  \BT@set@to@height#1{#2}%
  \begingroup%
  \BT@set@to@depth#1{#2}%
  \expandafter\endgroup\expandafter\advance\expandafter#1\the#1\relax}
```

`\BT@swap@hooks`
`\BT@swap@dims`

Swap two (parameterless) macros or lengths.

```
\def\BT@swap@hooks#1#2{%
  \expandafter\let\expandafter#1\expandafter#2%
    \expandafter\def\expandafter#2\expandafter{#1}}
\def\BT@swap@dims#1#2{%
  \expandafter#1\expandafter\the#2\expandafter\relax%
    \expandafter#2\the#1\relax}
```

`\BT@set@dim@to@greater`
`\BT@set@dim@to@smaller`

Set length `#1` to the greater or smaller of the two lengths.

```
\def\BT@set@dim@to@greater#1#2{%
  #1\dimexpr\ifdim\dimexpr#1\relax < \dimexpr#2\relax#2\else#1\fi\relax}
\def\BT@set@dim@to@smaller#1#2{%
  #1\dimexpr\ifdim\dimexpr#1\relax < \dimexpr#2\relax#1\else#2\fi\relax}
```

`\BT@extract@rgb@color@specs`

Extract color specifications and convert them to `RGB`.
`#1`: ⟨*color*⟩, `#2`: ⟨*macro to save in*⟩

```
\def\BT@extract@rgb@color@specs#1#2{%
  \begingroup%
  \extractcolorspecs{#1}{\tmp@mod}{\tmp@col}%
  \convertcolorspec{\tmp@mod}{\tmp@col}{rgb}{\tmp@col}%
  \BT@endgroup@let#2\tmp@col}
```

6.4 Macros for setting the styles

6.4.1 Helper macros

`\BT@at@to@dash`
`\BT@at@to@dash@i`

Convert `@` to `-` (except trailing `@`, which is removed) for root children. For example `@r@l@` → `-r-l`. Used in naming the child nodes. For root, `@`, it expands to empty.

```
\def\BT@at@to@dash#1{\BT@at@to@dash@i#1\@nil}
\def\BT@at@to@dash@i @#1{\ifx\@nil#1\else-#1\expandafter\BT@at@to@dash@i\fi}
```

`\BT@anchor@or@default`

Use the given anchor, or if empty—the default one.
`#1`: ⟨*anchor*⟩, `#2`: ⟨`root` | `left` | `right` | `final`⟩

```
\def\BT@anchor@or@default#1#2{%
  \BT@if@blank{#1}{\pgfkeysvalueof{/BT/#2 label anchor}}{#1}}
```

`\BT@color@or@default`

Use the given color, or if empty—the default one.

```
\def\BT@color@or@default#1{\BT@if@blank{#1}{BT@default}{#1}}
```

`\BT@if@child@is@outer`
`\BT@if@child@is@outer@i`
`\BT@if@child@is@outer@ii`

Expands to ⟨*true*⟩ if the child node named `#1` lies on an outer edge (reached by all right or all left moves from the root), and to ⟨*false*⟩ otherwise.
`#1`: ⟨*child name*⟩, `#2`: ⟨*true*⟩, `#3`: ⟨*false*⟩

```
\def\BT@if@child@is@outer#1{\BT@if@child@is@outer@i#1\@nil}
\def\BT@if@child@is@outer@i @#1@#2{%
```

If `#1 == #2`, compare `#2` to its child;

```
  \ifx#1#2\expandafter\BT@if@child@is@outer@i\expandafter @\expandafter#2%
```

else, see whether we have reached the end, or its child is on the other side.

```
  \else\expandafter\BT@if@child@is@outer@ii\expandafter#2\fi}
\def\BT@if@child@is@outer@ii#1{%
  \ifx\@nil#1\expandafter\@firstoftwo%
  \else\expandafter\expandafter\expandafter\@secondoftwo%
    \expandafter\BT@gobble@till@nil\fi}
```

`\BT@check@if@new@level`
`\BT@new@level`

If this is the first time we visit a child on level `#1`, increase the height of the box.

```
\def\BT@check@if@new@level#1{%
  \ifcsname level \the\numexpr#1\relax exists\endcsname\else%
    \BT@new@level{#1}%
    \csname level \the\numexpr#1\relax exists\expandafter\endcsname\fi}
\def\BT@new@level#1{%
  \BT@message{adding new level: \the\numexpr#1\relax}%
  \pgfmathparse{\BT@current@level@distance{#1}}%
  \advance\BT@bbox@height\pgfmathresult pt\relax%
  \BT@message{added \pgfmathresult pt to bounding box height}}
```

`\BT@l@xshift`
`\BT@r@xshift`
`\BT@xshift`

Update the current x-coordinate (relative to root) in the bounding box by shifting it by half ±sibling distance on level `#1`. + for right moves, - for left.

```
\def\BT@l@xshift{\BT@xshift-}
\def\BT@r@xshift{\BT@xshift+}
\def\BT@xshift#1#2{%
```

`#1`: ⟨`+` | `-`⟩, `#2`: ⟨`r` | `l`⟩

```
  \pgfmathparse{\BT@current@sibling@distance{#2} / 2}%
  \advance\BT@bbox@current@x #1\pgfmathresult pt\relax%
  \BT@message{current x coordinate is \the\BT@bbox@current@x}}
```

`\BT@update@bbox@l@width`
`\BT@update@bbox@r@width`

Compare the current x-coordinate with the size of the bounding box and update it if necessary. For the left side, negate the coordinate first.

```
\def\BT@update@bbox@l@width{%
  \@tempdima=-\BT@bbox@current@x\relax%
  \BT@set@dim@to@greater\BT@bbox@l@width\@tempdima}
\def\BT@update@bbox@r@width{%
  \BT@set@dim@to@greater\BT@bbox@r@width\BT@bbox@current@x}
```

`\BT@save@final@child@size`

Set the relevant lengths to the size of the final child's label—if it's an outer one, save its width and height; eitherway, check if this label is wider (taller) than the current maximum final child width (height), and update the width if so.
`#1`–`3`: ⟨*side, name and depth of child*⟩

```
\def\BT@save@final@child@size#1#2#3{%
  \BT@set@to@width\@tempdima{%
    \pgfinterruptpicture#3\endpgfinterruptpicture}%
  \BT@set@to@total@height\@tempdimb{%
    \pgfinterruptpicture#3\endpgfinterruptpicture}%
  \BT@set@dim@to@greater\BT@max@final@width\@tempdima%
  \BT@set@dim@to@greater\BT@max@final@height\@tempdimb%
  \BT@if@child@is@outer{#2}{%
    \csname BT@#1@final@width\endcsname\@tempdima\relax%
    \csname BT@#1@final@height\endcsname\@tempdimb\relax}{}}
```

6.4.2 Macros for processing subpath specifications

`\BT@process@next@path`

Take the next path (delimited by `,`) and split it into subpaths (delimited by `!`). `\BT@file@name@new@path` inserts a string to the file name currently being generated indicating the start of a new path at the root.

```
\def\BT@process@next@path#1,{%
  \ifx\@nil#1\relax\expandafter\@firstoftwo%
  \else\expandafter\@secondoftwo\fi{}{%
    \BT@file@name@new@path%
    \BT@bbox@current@x=\z@\relax%
    \def\BT@current@child{@}%
    \def\BT@current@side{}%
    \def\BT@current@level{\BT@root@depth}%
    \BT@if@blank{#1}{}{\BT@process@next@subpath#1!\@nil!}%
    \BT@process@next@path}}
```

`\BT@process@next@subpath` Take the next subpath and split it into ⟨*l/r moves*⟩, ⟨*label*⟩, ⟨*color*⟩, ⟨*anchor*⟩. If we've reached the end (token `#1` is `\@nil`), then continue the last move if `continue at path end` is `true`.

```
\def\BT@process@next@subpath#1!{%
  \ifx\@nil#1\relax\expandafter\@firstoftwo%
  \else\expandafter\@secondoftwo\fi{%
    \BT@if{BT@continue@at@end}{%
      \BT@message{continuing last path, starting at \BT@current@child}%
      \BT@set@subpath@style c::::\@nil}{}}{%
    \BT@if@blank{#1}{}{%
      \BT@set@subpath@style#1::::\@nil}%
    \BT@process@next@subpath}}
```

`\BT@set@subpath@style` If the path is empty (the subpath starts with `:`), then set the style for the last visited child (insert an empty token for the "next move").
`#1`: ⟨*path*⟩, `#2`: ⟨*label*⟩, `#3`: ⟨*color*⟩, `#4`: ⟨*anchor*⟩, `#5`: ignored

```
\def\BT@set@subpath@style#1:#2:#3:#4:#5\@nil{%
  \BT@message{setting style (#2, #3, #4) for subpath #1,
    starting at \BT@current@child}%
```

Append the style to the file name currently being generated.

```
  \BT@file@name@append@style{#1}{#2}{#3}{#4}%
  \edef\tmp@hook{\noexpand\BT@set@next@style{\BT@current@child}{%
      \BT@current@level}\unexpanded{{#2}}{#3}{#4}{\BT@current@side}%
    \BT@if@blank{#1}{{}}{#1}s}%
```

stop here (append an `s` move), we'll continue when the last subpath is done

```
  \tmp@hook\@nil}%
```

In what follows (unless stated otherwise) the signature for the macros is as follows:
`#1–2`: ⟨*name and depth of* ***parent***⟩, `#2`: ⟨*label*⟩, `#3`: ⟨*edge color*⟩, `#4`: ⟨*label anchor*⟩

`\BT@set@next@style` `\BT@set@next@style` expects the previous and next moves (either of them can be empty) and calls macros for each valid combination that do what needs to be done. These macros will call `\BT@set@next@style` again, to set the next move. If `label on every edge` is `false`, they will pass an empty label. If the current depth is the tree depth, `\BT@max@depth`, then call `\BT@terminate@path@style` which saves the last visited child's name, depth and side, so we can continue from here next time.

, #6–7: ⟨*previous and next moves*⟩

```
\def\BT@set@next@style#1#2#3#4#5#6#7{%
  \BT@message{previous: #1, went: #6, going #7, depth:
    \the\numexpr#2\relax/\BT@max@depth}%
  \ifnum\numexpr#2\relax = \numexpr\BT@max@depth\relax%
    \expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi%
    {\BT@terminate@path@style{#1}{#2}{#6}}%
    {\ifcsname BT@go@#6@#7\endcsname%
      \expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi%
      {\csname BT@go@#6@#7\endcsname{#1}{#2}{#3}{#4}{#5}}%
      {\PackageError{binarytree}{Invalid path: only l, r, c or s allowed}{}}}}
```

`\BT@terminate@path@style`

#1–3: ⟨*name, depth, and side of last visited child*⟩

```
\def\BT@terminate@path@style#1#2#3{%
  \edef\BT@current@child{#1}%
  \edef\BT@current@level{\the\numexpr#2\relax}%
  \def\BT@current@side{#3}%
  \BT@gobble@till@nil}
```

`\BT@go@@c`
`\BT@go@l@c`
`\BT@go@r@c`

Continuing—keep inserting the previous move (unless we are at the root).

```
\def\BT@go@@c{%
  \BT@terminate@path@style{@}{\BT@root@depth}{}}
\def\BT@go@l@c{\BT@continue{l}}
\def\BT@go@r@c{\BT@continue{r}}
```

`\BT@go@@s`
`\BT@go@l@s`
`\BT@go@r@s`

Stopping.

```
\let\BT@go@@s\BT@go@@c
\def\BT@go@l@s#1#2{\BT@terminate@path@style{#1}{#2}{l}}
\def\BT@go@r@s#1#2{\BT@terminate@path@style{#1}{#2}{r}}
```

`\BT@go@@`

Previous and current moves are empty—we're setting the root style.

```
\def\BT@go@@#1#2#3#4#5{%
  \BT@set@root@style{#3}{#4}{#5}%
  \BT@if{BT@label@every@edge}{%
    \BT@set@next@style{@}{\BT@root@depth}{#3}{#4}{#5}{}}{%
    \BT@set@next@style{@}{\BT@root@depth}{}{#4}{}{}}}
```

`\BT@go@@l`
`\BT@go@@r`

Previous move was empty—we are setting one of root's children.

```
\def\BT@go@@l{\BT@go@l@or@r{l}}
\def\BT@go@@r{\BT@go@l@or@r{r}}
```

`\BT@go@l@l`
`\BT@go@r@r`

Keep moving left or right.

```
\let\BT@go@l@l\BT@go@@l
\let\BT@go@r@r\BT@go@@r
```

`\BT@go@l@` `\BT@go@r@` Current move is empty—set the style for the current child.

```
\def\BT@go@l@{\BT@stay@here{l}}
\def\BT@go@r@{\BT@stay@here{r}}
```

`\BT@go@r@l` `\BT@go@l@r` Changed direction—continue previous path until the top of the tree (if configured this way) and then do requested move.

```
\def\BT@go@r@l{\BT@turn{r}{l}}
\def\BT@go@l@r{\BT@turn{l}{r}}
```

`\BT@continue` #1: ⟨*previous move*⟩, #2–6: same as #1–5 from above

```
\def\BT@continue#1#2#3#4#5#6{%
  \BT@if{BT@label@every@edge}{%
    \BT@set@next@style{#2}{#3}{#4}{#5}{#6}#1#1c}{%
    \BT@set@next@style{#2}{#3}{}{#5}{}#1#1c}}
```

`\BT@turn` Save `\BT@bbox@current@x` and restore it later. `\@tempdimb` is only used when setting a labelled final child, so it will not be used here
#1–2: ⟨*previous and next moves*⟩, #3–7: same as #1–5 from above

```
\def\BT@turn#1#2#3#4#5#6#7{%
  \BT@if{BT@continue@after@turn}{%
    \@tempdimb=\BT@bbox@current@x\relax%
    \BT@if{BT@default@color@after@turn}{%
      \BT@set@next@style{#3}{#4}{}{}{#7}#1c\@nil}{%
      \BT@set@next@style{#3}{#4}{}{#6}{#7}#1c\@nil}%
    \BT@bbox@current@x=\@tempdimb\relax}{}%
  \BT@go@l@or@r{#2}{#3}{#4}{#5}{#6}{#7}}
```

`\BT@stay@here` #1: ⟨*previous move*⟩, #2–6: same as #1–5 from above

```
\def\BT@stay@here#1#2#3#4#5#6{%
  \BT@set@child@style{#1}{#2}{#3}{#4}{#5}{#6}%
  \BT@if{BT@label@every@edge}{%
    \BT@set@next@style{#2}{#3}{#4}{#5}{#6}#1}{%
    \BT@set@next@style{#2}{#3}{}{#5}{}#1}}
```

`\BT@go@l@or@r` #1: ⟨*current move*⟩, #2–6: same as #1–5 from above

```
\def\BT@go@l@or@r#1#2#3#4#5#6{%
  \BT@set@child@style{#1}{#2#1@}{#3+1}{#4}{#5}{#6}%
  \BT@if{BT@label@every@edge}{%
    \BT@set@next@style{#2#1@}{#3+1}{#4}{#5}{#6}#1}{%
    \BT@set@next@style{#2#1@}{#3+1}{}{#5}{}#1}}
```

`\BT@set@root@style` Set the `\BT@root@width` and `\BT@root@height` lengths as well as a style named `@` under (`/tikz/binary tree`).
#1: ⟨*label*⟩, #2: ⟨*edge color*⟩, #3: ⟨*label anchor*⟩

```
\def\BT@set@root@style#1#2#3{%
  \BT@set@to@width\BT@root@width{\pgfinterruptpicture%
```

```
    \BT@wrap@label{#1}\endpgfinterruptpicture}%
  \BT@set@to@total@height\BT@root@height{\pgfinterruptpicture%
    \BT@wrap@label{#1}\endpgfinterruptpicture}%
  \BT@set@root@style@i{@}{\BT@wrap@label{#1}}{\BT@color@or@default{#2}}{#3}}
```

The signature of the following macros is:
#1–3: ⟨*side, name and depth of child*⟩, #3: ⟨*label*⟩, #4: ⟨*edge color*⟩, #5: ⟨*label anchor*⟩

`\BT@set@child@style`

Set the style for a child (either intermediate or final). Check if it is the first visited child on this level and if so—increase the height of the bounding box. Also check if it is an outer child and if it is the first time we are setting the style, increase the width of the box on its side. Call the relevant macro to actually set the style.

```
\def\BT@set@child@style#1#2#3#4#5#6{%
  \BT@check@if@new@level{#3}%
  \csname BT@#1@xshift\endcsname{#3}%
  \csname BT@update@bbox@#1@width\endcsname%
  \BT@if@blank{#4}{\BT@if@blank{#5}{%
      \BT@set@default@if@missing{#2}}{%
      \BT@set@empty@style{#2}{\BT@color@or@default{#5}}}}{%
    \ifnum\numexpr#3\relax = \numexpr\BT@max@depth\relax%
      \BT@set@final@style{#1}{#2}{#3}{#4}{#5}{#6}%
    \else\BT@set@inter@style{#1}{#2}{#3}{#4}{#5}{#6}\fi}}
```

`\BT@set@inter@style`

Set the child for a left or right intermediate child—simply call the relevant macro.

```
\def\BT@set@inter@style#1#2#3#4#5#6{%
  \csname BT@set@inter@style@#1\endcsname{#2}{%
    \BT@wrap@label{#4}}{\BT@color@or@default{#5}}{#6}}
```

`\BT@set@final@style`

Set the style for a final child—save the size of its label (if needed) first.

```
\def\BT@set@final@style#1#2#3#4#5#6{%
  \BT@save@final@child@size{#1}{#2}{\BT@wrap@label{#4}}%
  \BT@set@final@style@i{#2}{%
    \BT@wrap@label{#4}}{\BT@color@or@default{#5}}{#6}}
```

`\BT@set@root@style@i`
`\BT@set@inter@style@l`
`\BT@set@inter@style@r`
`\BT@set@final@style@i`

The following macros set the TikZ style and define the macro \⟨*child name*⟩ `styled`.
#1: ⟨*name*⟩, #2: ⟨*label*⟩, #3: ⟨*color*⟩, #4: ⟨*anchor*⟩

```
\def\BT@set@root@style@i#1#2#3#4{%
  \BT@message{setting style (#2, #3,
    \BT@anchor@or@default{#4}{root}) for root}%
  \ifcsname#1 styled\endcsname\else\csname#1 styled\expandafter\endcsname\fi%
  \tikzset{binary tree/#1/.style={binary tree/root={#2}{%
        \BT@anchor@or@default{#4}{root}}{#3}}}}
\def\BT@set@inter@style@l#1#2#3#4{%
  \BT@message{setting style (#2, #3,
    \BT@anchor@or@default{#4}{left}) for left child #1}%
  \ifcsname#1 styled\endcsname\else\csname#1 styled\expandafter\endcsname\fi%
  \tikzset{binary tree/#1/.style={binary tree/child={#2}{%
        \BT@anchor@or@default{#4}{left}}{#3}}}}
```

```
\def\BT@set@inter@style@r#1#2#3#4{%
  \BT@message{setting style (#2, #3,
    \BT@anchor@or@default{#4}{right}) for right child #1}%
  \ifcsname#1 styled\endcsname\else\csname#1 styled\expandafter\endcsname\fi%
  \tikzset{binary tree/#1/.style={binary tree/child={#2}{%
        \BT@anchor@or@default{#4}{right}}{#3}}}}
\def\BT@set@final@style@i#1#2#3#4{%
  \BT@message{setting style (#2, #3,
    \BT@anchor@or@default{#4}{final}) for final child #1}%
  \ifcsname#1 styled\endcsname\else\csname#1 styled\expandafter\endcsname\fi%
  \tikzset{binary tree/#1/.style={binary tree/final child={#2}{%
        \BT@anchor@or@default{#4}{final}}{#3}}}}
```

`\BT@set@empty@style` An unlabelled child—simply color the edge.
#1: ⟨*name*⟩, #2: ⟨*color*⟩

```
\def\BT@set@empty@style#1#2{%
  \BT@message{setting empty style (#2) for child #1}%
  \ifcsname#1 styled\endcsname\else\csname#1 styled\expandafter\endcsname\fi%
  \tikzset{binary tree/#1/.style={binary tree/empty={#2}}}}
```

`\BT@set@default@if@missing` Set a default style (used if the neither label nor color is given and we have not set this style before).
#1: ⟨*name*⟩

```
\def\BT@set@default@if@missing#1{%
  \BT@message{setting default style for child #1}%
  \ifcsname#1 styled\endcsname\else%
    \tikzset{binary tree/#1/.style={binary tree/empty={BT@default}}}%
    \csname#1 styled\expandafter\endcsname\fi}
```

6.4.3 Macros for generating the file name

`\BT@file@name@init` Set the first part of the file name according to the global style.

```
\def\BT@file@name@init{%
  \BT@extract@rgb@color@specs{BT@default}{\@@tmp}%
  \edef\BT@file@name{%
    btree-\the\numexpr\BT@max@depth\relax% prefix and depth
    _\BT@grow@direction%                   grow direction
    \ifBT@root@edge _edge\fi%              root edge?
    _\@@tmp%                               default color
    _\number\BT@level@distance%            level distance in sp
    \ifBT@level@distance@scales%
      -scaled\fi%                          level distance scales
    _\number\BT@sibling@distance%          sibling distance in sp
    \ifBT@sibling@distance@scales%
      -scaled\fi%                          sibling distance scales
    _\number\BT@label@distance%            label distance in sp
    _\BT@xscale%                           x-scale
    _\BT@yscale}}%                         y-scale
```

`\BT@file@name@new@path` When starting a new path, append a `_` to the filename.

```
\def\BT@file@name@new@path{\BT@eapp@to@hook\BT@file@name{_}}
```

`\BT@file@name@append@style` When setting the style for a new subpath, append the style for it.

`#1`: ⟨*path*⟩, `#2`: ⟨*label*⟩, `#3`: ⟨*color*⟩, `#4`: ⟨*anchor*⟩

```
\def\BT@file@name@append@style#1#2#3#4{%
  \BT@if@blank{#3}{\def\@@tmp{}}{%
    \BT@extract@rgb@color@specs{#3}{\@@tmp}}%
  \BT@eapp@to@hook\BT@file@name{%
    -#1%                                        subpath
    -\BT@if@blank{#2}{}{%                       label
        \ifBT@math@labels MATH\else#2\fi}%
    -#4%                                        anchor
    -\@@tmp}}%                                  color
```

`\BT@file@name@set` If a file name is already set (by the `external/file name` or via direct call to `\tikzsetnextfilename`), do not do anything, otherwise—set the file name for the next figure if `external/use automatic file name` is `true`.

```
\def\BT@file@name@set{%
  \ifBT@auto@file@name%
    \if\relax\detokenize\expandafter{\tikzexternal@nextfile}\relax%
      \expandafter\tikzsetnextfilename\expandafter{\BT@file@name}\fi\fi}
```

6.5 Drawing the tree

`\BT@draw@tree@children` Enter a recursive for loop for `\BT@max@depth` levels. On the first call `#2 == 0`; the first `\BT@max@depth` children to be drawn are the rightmost ones. Then, the top of the tree is reached and we do not recurse another time here. The macro exits and is back into the previous call, where `#2 == \BT@max@depth - 1`. It then calls itself again to draw the left sibling of the rightmost final child; `##2 == \BT@max@depth` again, so it only sets the style and exits. It is back into the parent call (`##2 == \BT@max@depth - 1`) with nothing else to do—exit. Back into the parent call (`##2 == \BT@max@depth - 2`), it draws the left sibling of the rightmost child on level `\BT@max@depth - 1`, then its right child, *its* left child and so on…

`#1`–`2`: ⟨*name and depth of **parent***⟩

```
\def\BT@draw@tree@children#1#2{%
  \ifnum\numexpr#2\relax = \numexpr\BT@max@depth\relax\else%
    child[\ifcsname#1r@ styled\endcsname binary tree/#1r@\else%
      binary tree/default\fi] {node (btree\BT@at@to@dash{#1r@}) {}% right
        \BT@draw@tree@children{#1r@}{#2+1} }%
    child[\ifcsname#1l@ styled\endcsname binary tree/#1l@\else%
      binary tree/default\fi] {node (btree\BT@at@to@dash{#1l@}) {}% left
        \BT@draw@tree@children{#1l@}{#2+1} }\fi}
```

`\BT@draw@tree` Draw the whole path—before we do that, adjust the sizes of the bounding box. If we are adding a single edge to the root, we need to add a level distance to the height before adjustment. Expansion of `\BT@draw@tree@children` after `\node` does not work, so we expand the whole path before inserting it. Clip, draw the frame (if requested) and insert the path.

```
\def\BT@draw@tree{%
  \ifBT@root@edge%
    \advance\BT@bbox@height\BT@level@distance\relax%
    \BT@adjust@bbox@sides%
    \edef\BT@tree{%
      \noexpand\node[\ifcsname @ styled\endcsname binary tree/@\fi] (btree-root) {}%
      child[\ifcsname @ styled\endcsname binary tree/@\fi] {%
        node {} \BT@draw@tree@children{@}{\BT@root@depth}}}%
    \clip (-\BT@bbox@l@width,-\BT@bbox@depth)%
      rectangle (\BT@bbox@r@width,\BT@bbox@height);%
    \ifBT@framed\draw (current bounding box.south west)%
      rectangle (current bounding box.north east);\fi%
    \BT@tree;%
  \else\BT@adjust@bbox@sides%
    \edef\BT@tree{%
      \noexpand\node[\ifcsname @ styled\endcsname binary tree/@\fi] (btree-root) {}%
      \BT@draw@tree@children{@}{\BT@root@depth}}%
    \clip (-\BT@bbox@l@width,-\BT@bbox@depth)%
      rectangle (\BT@bbox@r@width,\BT@bbox@height);%
    \ifBT@framed\draw (current bounding box.south west)%
      rectangle (current bounding box.north east);\fi%
    \BT@tree;%
  \fi}
```

`\BinaryTree`
`\@BinaryTree`
`\BT@max@depth`
`\BT@root@depth`

`\BinaryTree` is the user interface. It resets the lengths, sets the keys, defines `\BT@max@depth` and `\BT@root@depth`, initializes the file name, sets the styles; if needed, enables externalization and sets the `/tikz/binary tree/externalize` style; draws the tree in either a `tikzpicture` or a `scope` environment, applying the `/tikz/binary tree` style. Everything is done inside a group.

#1: ⟨*local options*⟩, #2: ⟨*path specification*⟩, #3: ⟨*tree depth*⟩

```
\def\BinaryTree{\@ifnextchar[\@BinaryTree{\@BinaryTree[]}}
\def\@BinaryTree[#1]#2#3{%
  \begingroup%
  \BT@bbox@r@width=\z@\relax%
  \BT@bbox@l@width=\z@\relax%
  \BT@bbox@height=\z@\relax%
  \BT@bbox@depth=\z@\relax%
  \BT@root@width=\z@\relax%
  \BT@root@height=\z@\relax%
  \BT@max@final@width=\z@\relax%
  \BT@max@final@height=\z@\relax%
  \BT@l@final@width=\z@\relax%
  \BT@l@final@height=\z@\relax%
  \BT@r@final@width=\z@\relax%
  \BT@r@final@height=\z@\relax%
  \pgfqkeys{/BT}{#1}%
\ifBT@root@edge\def\BT@max@depth{#3+1}\def\BT@root@depth{1}%
\else\def\BT@max@depth{#3}\def\BT@root@depth{0}\fi%
\BT@set@min@bbox@size%
\BT@file@name@init%
\BT@process@next@path#2,\@nil,%
\ifBT@separate\ifBT@external%
  \tikzset{binary tree/externalize}\BT@file@name@set\fi%
  \begin{tikzpicture}[binary tree]%
    \BT@draw@tree%
  \end{tikzpicture}%
\else\begin{scope}[binary tree]%
    \BT@draw@tree%
  \end{scope}\fi%
\endgroup}
```

Change History

1.01

`\BinaryTree`: Each node is now uniquely named. 31

v1.0

General: First published version . . 1

Index