A recursive alignment algorithm – pst-tvz

Trees; v.1.02

Timothy Van Zandt
Herbert Voß

January 2, 2026

Contents

Part I.
Using the package

The node and node connections are perfect tools for making trees, but positioning the nodes using `\rput` would be rather tedious, unless you have a computer program that generates the coordinates.

The files `pst-tvz.tex`/`pst-tvz.sty` contains a high-level interface for making trees.

It should be noted that the correct result is not guaranteed with every `dvips` driver. This package was written for Rokicki's `dvips` programme, which is practically part of every TEX distribution. However, with an up-to-date LATEX-distribution you can use `lualatex` to get directly the PDF output.

thanks to: Olivier Guibé;

1. Overview

The tree commands are

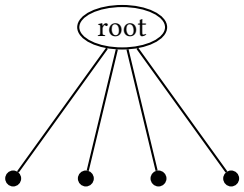
`\pstree{<root>}{<successors>}`

TEX version	LATEX version
`\psTree{<root>}`	`\begin{psTree}{root}`
`<successors>\\`	`<successors> \\`
`<successors>\\`	`<successors> \\`
...	...
`\endpsTree`	`\end{psTree}`

These do the same thing, but just have different syntax. `\psTree` is the "long" version. These macros make a box that encloses all the nodes, and whose baseline passes through the center of the root. Most of the nodes has a variant for use within a tree and are called tree nodes (see Section 2).

Trees and tree nodes are called *tree objects*. The *root* of a tree should be a single tree object, and the *successors* should be one or more tree objects. Here is an example with only nodes:

A simple tree with `\TC*` nodes

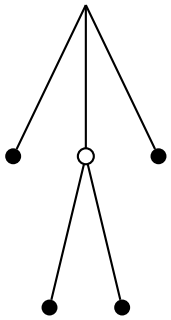



```
\pstree[radius=3pt]{\Toval{root}}{\TC* \TC* \TC*
↪  \TC*}
```

There is no difference between a terminal node and a root node, other than their position in the `\pstree{}` command.

Here is an example where a tree is included in the list of successors, and hence becomes subtree:

A simple tree with a `\Tp` root



```
\pstree[radius=3pt]{\Tp}{%
  \TC*
  \pstree{\TC}{\TC* \TC*}
  \TC*}
```

2. Tree Nodes

In each case, the name of the tree node is formed by omitting "'node"' from the end of the name and adding "T" at the beginning. For example, `\psovalnode` becomes `\Toval`. Here is the list of such tree nodes:

```
\Tp* [Options]
\Tc* [Options] {dim}
\TC* [Options]
\Tf* [Options]
\Tdot* [Options]
\Tr* [Options] {stuff}
\TR* [Options] {stuff}
\Tcircle* [Options] {stuff}
\TCircle* [Options] {stuff}
\Toval* [Options] {stuff}
\Tdia* [Options] {stuff}
\Ttri* [Options] {stuff}
```

The syntax of a tree node is the same as of its corresponding "normal" node, except that:

There is always an optional argument for setting graphics parameters, even if the original node did not have one;

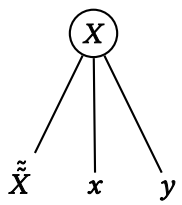
- There is no argument for specifying the name of the node;
- There is never a coordinate argument for positioning the node; and

- To set the reference point with `\Tr`, set the `ref` parameter.

Figure 1 gives a reminder of what the nodes look like.

The difference between `\Tr` and `\TR` (variants of `\rnode` and `\Rnode`, respectively) is important with trees. Usually, you want to use `\TR` with vertical trees because the baselines of the text in the nodes line up horizontally. For example:

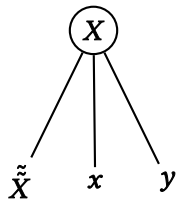
A simple tree




```
$\pstree[nodesepB=3pt]{\Tcircle{X}}{%
    \TR{\tilde{\tilde{X}}}
    \TR{x}
    \TR{y}}$
```

Compare with this example, which uses `\Tr`:

Example with `\Tr` node




```
$ \pstree[nodesepB=3pt]{\Tcircle{X}}{%
    \Tr{\tilde{\tilde{X}}}
    \Tr{x}
    \Tr{y}} $
```

There is also a null tree node:

```
\Tn
```

It is meant to be just a place holder. Look at the tree in Figure page 6. The bottom row has a node missing in the middle. `\Tn{}` was used for this missing node.

There is also a special tree node that doesn't have a "normal" version and that can't be used as the root node of a whole tree:

```
\Tfan* [Options]
```

Available node types

```
\small\psset{armB=1cm, levelsep=3cm, treesep=-3mm,
             angleB=-90, angleA=90, nodesepA=3pt, nodesepB=0}
\def\psedge#1#2{\ncangle{#2}{#1}}

\psTree[treenodesize=2.5cm]{\Toval{Tree nodes}}     \\
  \Tp~{\ttfamily\string\Tp}  \Tc{.5}~{\ttfamily\string\Tc}  \TC~{\ttfamily\string\TC}
  \psTree[levelsep=4cm,armB=2cm]{\Tp[edge=\ncline]} \\
    \Tcircle{\ttfamily\string\Tcircle}              \Tdot~{\ttfamily\string\Tdot}
    \TCircle[radius=1.2]{\ttfamily\string\TCircle}  \Tn[name=Tn]\uput[0](Tn){\ttfamily\string\Tn}
    \Toval{\ttfamily\string\Toval}                  \Ttri{\ttfamily\string\Ttri}
    \Tdia{\ttfamily\string\Tdia}
  \endpsTree%
  \Tf~{\ttfamily\string\Tf}  \Tr{\ttfamily\string\Tr}  \TR{\ttfamily\string\TR}
\endpsTree
```

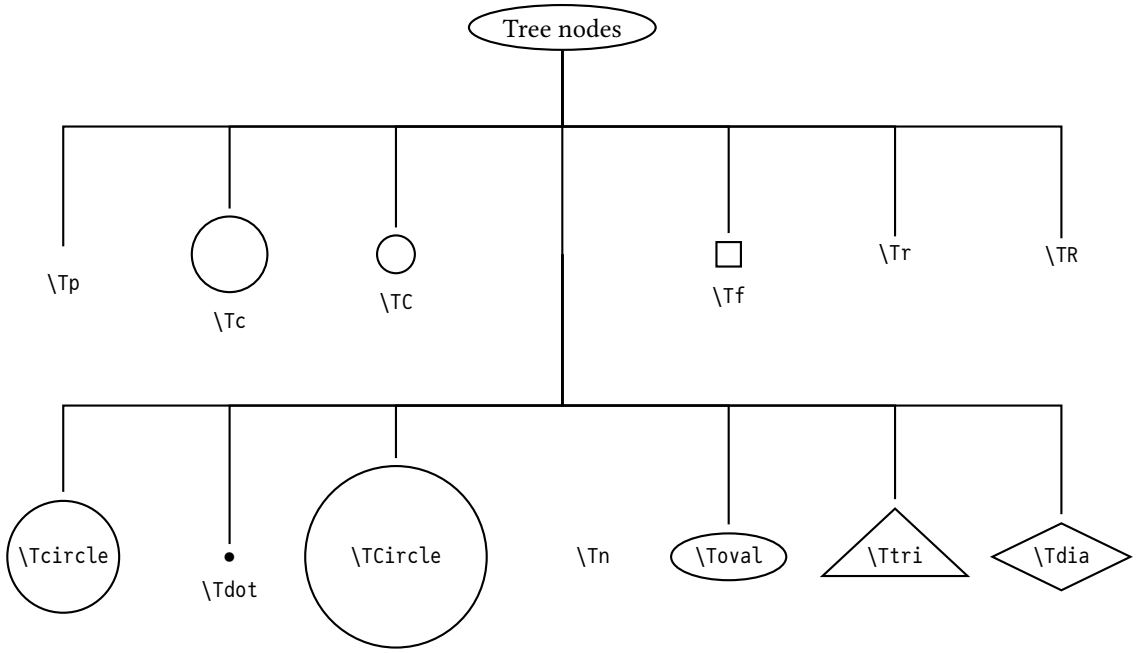
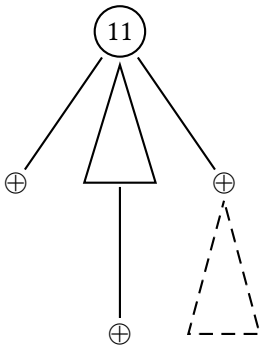


Figure 1: The tree nodes.

This draws a triangle whose base is `fansize` and whose opposite corner is the predecessor node, adjusted by the value of `nodesepA` and `offsetA`. For example:

Using `nodesep`



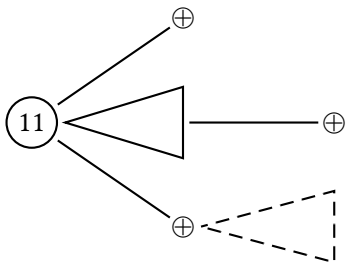

```
\pstree[dotstyle=oplus,dotsize=8pt,nodesep=2pt]{\Tcircle{11}}{%
  \Tdot
  \pstree{\Tfan}{\Tdot}
  \pstree{\Tdot}{\Tfan[linestyle=dashed]}}
```

3. Tree orientation

Trees can grow down, up, right or left, depending on the `treemode=` `D`, `U`, `R`, or `L` parameter.

Here is what the previous example looks like when it grows to the right:

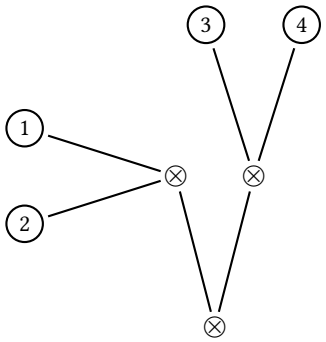
Right treemode




```
\pstree[dotstyle=oplus,dotsize=8pt,
    nodesep=2pt,treemode=R]
  {\Tcircle{11}}{%
    \Tdot
    \pstree{\Tfan}{\Tdot}
    \pstree{\Tdot}{\Tfan[linestyle=dashed]}}
```

You can change the `treemode` in the middle of the tree. For example, here is a tree that grows up, and that has a subtree which grows to the left:

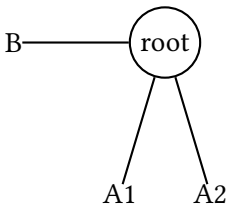
Using `treemode`




```
\footnotesize
\pstree[treemode=U,dotstyle=otimes,dotsize=8pt,nodesep=2pt]
  {\Tdot}{%
    \pstree[treemode=L]{\Tdot}{\Tcircle{1} \Tcircle{2}}
    \pstree{\Tdot}{\Tcircle{3} \Tcircle{4}}}
```

Since you can change a tree's orientation, it can make sense to include a tree (<treeB>) as a root node (of <treeA>). This makes a single logical tree, whose root is the root of <treeB>, and that has successors going off in different directions, depending on whether they appear as a successor to <treeA> or to <treeB>.

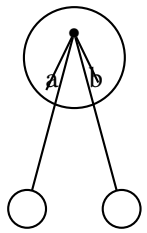
Using `treemode` L




```
\pstree{\pstree[treemode=L]{\Tcircle{root}}{\Tr{B}}}{%
  \Tr{A1}
  \Tr{A2}}
```

On a semi-related theme, note that any node that creates an LR-box can contain a tree. However, nested trees of this kind are not related in any way to the rest of the tree. Here is an example:

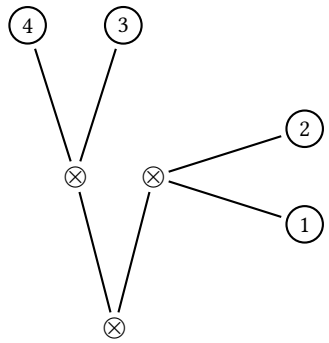
Nested trees




```
\psTree{\Tcircle{\pstree[treesep=0.4,levelsep=0.6,
                  nodesepB=-6pt]{\Tdot}{%
            \TR{a} \TR{b}}}}\\
  \TC
  \TC
\endpsTree
```

When the tree grows up or down, the successors are lined up from left to right in the order they appear in `\pstree`. When the tree grows to the left or right, the successors are lined up from top to bottom. As an afterthought, you might want to flip the order of the nodes. The keyword `treeflip=true/false`let's you do this. For example:

Using `treeflip`




```
\footnotesize
\pstree[treemode=U,dotstyle=otimes,dotsize=8pt,
  nodesep=2pt,treeflip=true]{\Tdot}{%
    \pstree[treemode=R]{\Tdot}{\Tcircle{1} \Tcircle{2}}
    \pstree{\Tdot}{\Tcircle{3} \Tcircle{4}}}
```

Note that I still have to go back and change the `treemode` of the subtree that used to grow to the left.

4. The distance between successors

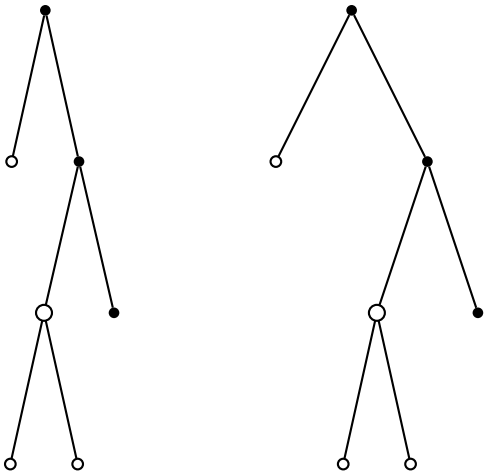
The distance between successors is set by the key `treesep`. The rest of this section describes ways to fine-tune the spacing between successors. You can change the method for calculating the distance between subtrees by setting the `treefit=tight/loose` parameter. Here are the two methods:

tight When `treefit=tight` , which is the default, `treesep` is the minimum distance between each of the levels of the subtrees.

loose When `treefit=loose` , `treesep` is the distance between the subtrees' bounding boxes. Except when you have large intermediate nodes, the effect is that the horizontal distance (or vertical distance, for horizontal trees) between all the terminal nodes is the same (even when they are on different levels).[1]

Compare:

1 When all the terminal nodes are on the same level, and the intermediate nodes are not wider than the base of their corresponding subtrees, then there is no difference between the two methods.

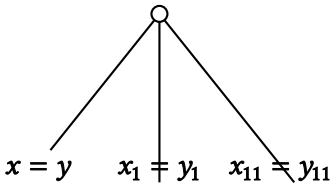


With `treefit=loose` , trees take up more space, but sometimes the structure of the tree is emphasized.

Sometimes you want the spacing between the centers of the nodes to be regular even though the nodes have different sizes. If you set `treenodesize` to a non-negative value, then PSTricks sets the width (or height+depth for vertical trees) to `treenodesize`, *for the purpose of calculating the distance between successors*.

For example, ternary trees look nice when they are symmetric, as in the following example:

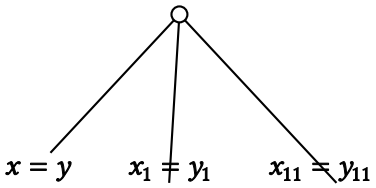
Ternary trees




```
\pstree[nodesepB=-8pt,treenodesize=.85]{\Tc{3pt}}{%
  \TR{$x=y$}
  \TR{$x_1=y_1$}
  \TR{$x_{11}=y_{11}$}}%$
```

Compare with this example, where the spacing varies with the size of the nodes:

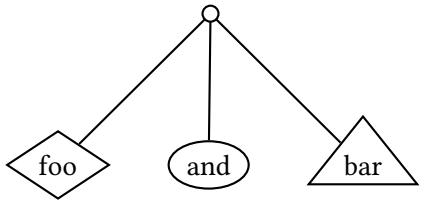
ode size




```
\pstree[nodesepB=-8pt]{\Tc{3pt}}{%
  \TR{$x=y$}
  \TR{$x_1=y_1$}
  \TR{$x_{11}=y_{11}$}}%$
```

Finally, if all else fails, you can adjust the distance between two successors by inserting `\tspace{`*length*`}` between them:

Adjusting with `\space`




```
\psTree{\Tc{3pt}}\\
  \Tdia{foo}
%  \tspace{-0.5}
  \Toval{and}
  \Ttri{bar}
\endpsTree
```

5. Spacing between the root and successors

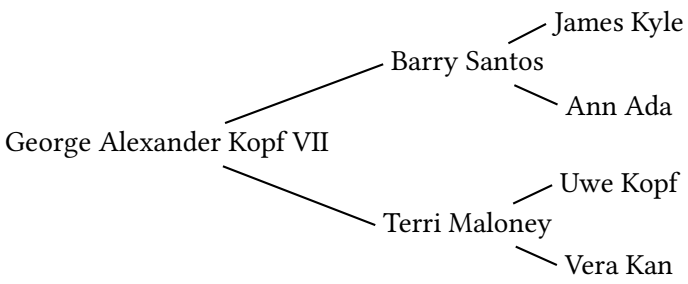
The distance between the center lines of the tree levels is `levelsep`. If you want the spacing between levels to vary with the size of the levels, use the * convention. Then `levelsep` is the distance between the bottom of one level and the top of the next level (or between the sides of the two levels, for horizontal trees).

Note: PSTricks has to write some information to your `.aux` file if using LaTeX, or to `\jobname.pst` otherwise, in order to calculate the spacing. You have to run your input file a few times before PSTricks gets the spacing right.

trees. Compare the following example:

Using `varlevelsep`

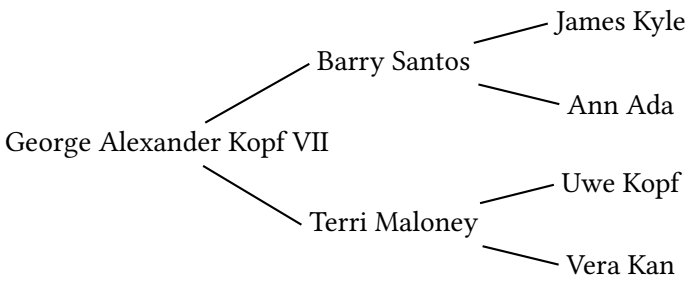
```
\def\psedge#1#2{\ncdiagg[nodesep=3pt,angleA=180,armA=0]{#2}{#1}}
\pstree[treemode=R,varlevelsep]{\Tr{George Alexander Kopf VII}}{%
  \pstree{\Tr{Barry Santos}}{\Tr{James Kyle} \Tr{Ann Ada}}
  \pstree{\Tr{Terri Maloney}}{\Tr{Uwe Kopf} \Tr{Vera Kan}}}
```



with this one, were the spacing between levels is fixed:

Using `levelsep`

```
\def\psedge#1#2{\ncdiagg[nodesep=3pt,angleA=180,armA=0]{#2}{#1}}
\pstree[treemode=R,levelsep=3cm]{\Tr{George Alexander Kopf VII}}{%
    \pstree{\Tr{Barry Santos}}{\Tr{James Kyle} \Tr{Ann Ada}}
    \pstree{\Tr{Terri Maloney}}{\Tr{Uwe Kopf} \Tr{Vera Kan}}}
```



6. Edges

Right after you use a tree node command, `\pssucc` is equal to the name of the node, and `\pspred` is equal to the name of the node's predecessor. Therefore, you can draw a line between the node and its predecessor by inserting, for example,

```
\ncline{\pspred}{\pssucc}
```

To save you the trouble of doing this for every node, each tree node executes

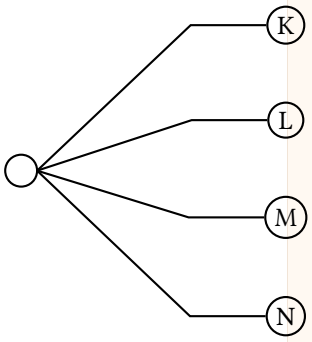
```
\psedge{\pspred}{\pssucc}
```

The default definition of \psedge is \ncline, but you can redefine it as you please with \def or LaTeX's \renewcommand.

For example, here I use \ncdiag, with armA=0, to get all the node connections to emanate from the same point in the predecessor. LaTeX users can instead type:

```
\renewcommand{\psedge}{\ncdiag[armA=0,angleB=180,armB=1cm]}
```

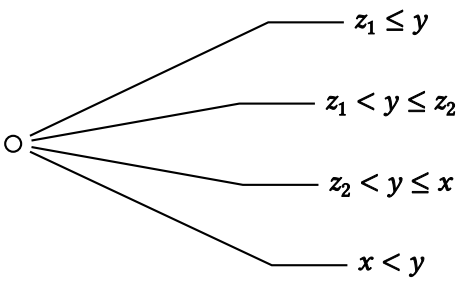
Macro \psedge




```
\def\psedge{\ncdiag[armA=0,angleB=180,armB=1cm]}
\pstree[treemode=R,levelsep=3.5cm,framesep=2pt]{\Tc{6pt}}{%
    \small \Tcircle{K} \Tcircle{L} \Tcircle{M} \Tcircle{N}}
```

Here is an example with \ncdiagg. Note the use of a negative armA value so that the corners of the edges are vertically aligned, even though the nodes have different sizes:

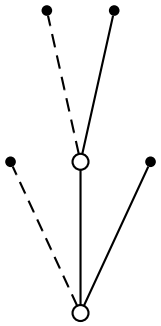
\ncdiagg and armA




```
\def\psedge#1#2{\ncdiagg[angleA=180,
       armA=1cm,nodesep=4pt]{#2}{#1}}
% Or: \renewcommand{\psedge}[2]{ ... }
$\pstree[treemode=R, levelsep=5cm]{\Tc{3pt}}{%
  \Tr{z_1\leq y}
  \Tr{z_1<y\leq z_2}
  \Tr{z_2<y\leq x}
  \Tr{x<y}
}$
```

Another way to define \psedge{} is with the edge parameter. Be sure to enclose the value in braces "" if it contains commas or other parameter delimiters. This gets messy if your command is long, and you can't use arguments like in the preceding example, but for simple changes it is useful. For example, if I want to switch between a few node connections frequently, I might define a command for each node connection, and then use the edge parameter.

edge Parameter

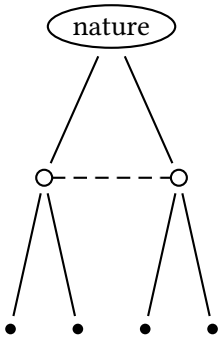


```
\def\dedge{\ncline[linestyle=dashed]}
\pstree[treemode=U,radius=2pt]{\Tc{3pt}}{%
  \TC*[edge=\dedge]
  \pstree{\Tc{3pt}}{\TC*[edge=\dedge]
    \TC*}
    \TC*}
```

You can also set edge=none to suppress the node connection.

If you want to draw a node connection between two nodes that are not direct predecessor and successor, you have to give the nodes a name that you can refer to, using the name parameter. For example, here I connect two nodes on the same level:

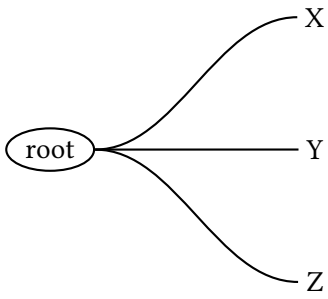
Macro \Tova




```
\pstree[nodesep=3pt,radius=2pt]{\Toval{nature}}{%
  \pstree{\Tc[name=top]{3pt}}{\TC* \TC*}
  \pstree{\Tc[name=bot]{3pt}}{\TC* \TC*}}
 \ncline[linestyle=dashed]{top}{bot}
```

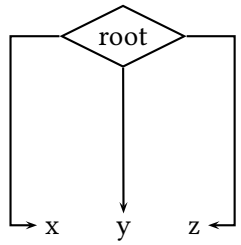
We conclude with the more examples.

Macro \Toval



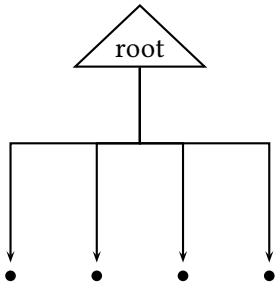

```
\def\psedge{\nccurve[angleB=180, nodesepB=3pt]}
\pstree[treemode=R, treesep=1.5, levelsep=3.5]%
  {\Toval{root}}{\Tr{X} \Tr{Y} \Tr{Z}}
```

Macro \TR




```
\pstree[nodesepB=3pt, arrows=->, xbbl=15pt,
  xbbr=15pt, levelsep=2.5cm]{\Tdia{root}}{%
  \TR[edge={\ncbar[angle=180]}]{x}
  \TR{y}
  \TR[edge=\ncbar]{z}
}
```

Macro \Ttri

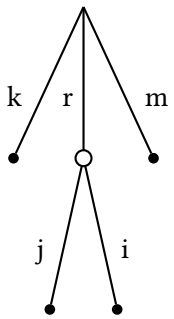



```
\psset{armB=1cm, levelsep=3cm, treesep=1cm,
  angleB=-90, angleA=90, arrows=<-, nodesepA=3pt}
\def\psedge#1#2{\ncangle{#2}{#1}}
\pstree[radius=2pt]{\Ttri{root}}{\TC* \TC* \TC* \TC*}
```

7. Edge and node labels

Right after a node, an edge has typically been drawn, and you can attach labels using `\ncput`, `\tlput`, etc. With `\tlput`, `\trput`, `\taput`, and `\tbput`, you can align the labels vertically or horizontally, just like the nodes. This can look nice, at least if the slopes of the node connections are not too different.

Put macros




```
\pstree[radius=2pt]{\Tp}{%
  \psset{tpos=.6}
  \TC* \tlput{k}
  \pstree{\Tc{3pt} \tlput[labelsep=3pt]{r}}{%
    \TC* \tlput{j}
    \TC* \trput{i}}
  \TC* \trput{m}}
```

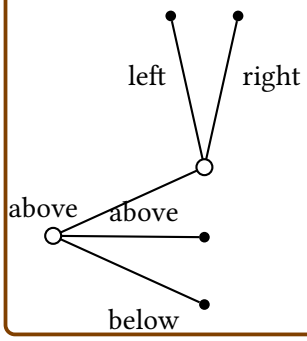
Within trees, the `tpos` parameter measures this distance from the predecessor to the successor, whatever the orientation of the true. (Outside of trees it measures the distance from the top to bottom or left to right nodes.)

PSTricks also sets `shortput=tab` within trees. This is a special `shortput` option that should not be used outside of trees. It implements the following abbreviations, which depend of the orientation of the true:

	Short for:	
Char.	*Vert.*	*Horiz.*
^	\tlput	\taput
_	\trput	\tbput

(The scheme is reversed if `treeflip=true` .)

`thistreesep` and `thislevelsep`



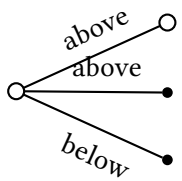

```
\psset{tpos=.6}
\pstree[treemode=R, thistreesep=1cm,
  thislevelsep=3cm,radius=2pt]{\Tc{3pt}}{%
  \pstree[treemode=U, xbbr=20pt]{\Tc{3pt}^{above}}{%
    \TC*^{left}
    \TC*_{right}}
  \TC*^{above}
  \TC*_{below}}
```

You can change the character abbreviations with

`\MakeShortTab{`*<char1>*`}{`*<char2>*`}`

The `\n*put` commands can also give good results:

`\n?put` Macros




```
\psset{npos=.6,nrot=:U}
\pstree[treemode=R, thistreesep=1cm,
  thislevelsep=3cm]{\Tc{3pt}}{%
  \Tc{3pt}\naput{above}
  \Tc*{2pt}\naput{above}
  \Tc*{2pt}\nbput{below}}
```

You can put labels on the nodes using \nput. However, \pstree won't take these labels into account when calculating the bounding boxes.

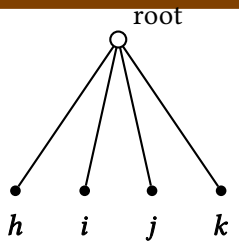
There is a special node label option for trees that does keep track of the bounding boxes:

␣ * [Options] {*stuff*}

Call this a "tree node label".

Put a tree node label right after the node to which it applies, before any node connection labels (but node connection labels, including the short forms, can follow a tree node label). The label is positioned directly below the node in vertical trees, and similarly in other trees. For example:

Macro \pssucc




```
\pstree[radius=2pt]{\Tc{3pt}\nput{45}{\pssucc}{root}}{%
  \TC*~{$h$} \TC*~{$i$} \TC*~{$j$} \TC*~{$k$}}
```

Note that there is no "long form" for this tree node label. However, you can change the single character used to delimit the label with

\MakeShortTnput{<*char1*>}

If you find it confusing to use a single character, you can also use a command sequence. E.g.,

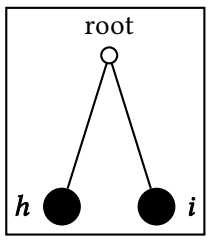
```
\MakeShortTnput{\tnput}
```

You can have multiple labels, but each successive label is positioned relative to the bounding box that includes the previous labels. Thus, the order in which the labels are placed makes a difference, and not all combinations will produce satisfactory results.

You will probably find that the tree node label works well for terminal nodes, without your intervention. However, you can control the tree node labels be setting several parameters.

To position the label on any side of the node ("l"eft, "r"ight, "a"bove or "b"elow), set: tnpos=l/r/a/b

tnpos




```
\psframebox{%
  \pstree{\Tc{3pt}~[tnpos=a,tndepth=0pt,radius=4pt]{root}}{%
    \TC*~[tnpos=l]{$h$}
    \TC*~[tnpos=r]{$i$}}}
```

When you leave the argument empty, which is the default, PSTricks chooses the label position is automatically.

To change the distance between the node and the label, set tnsep to a dimension When you leave the argument empty, which is the default, PSTricks uses the value of labelsep. When the value is negative, the distance is measured from the center of the node.

When labels are positioned below a node, the label is given a minimum height of tnheight. Thus, if you add labels to several nodes that are horizontally aligned, and if either these nodes have the same depth or tnsep is negative, and if the height of each of the labels is no more than tnheight, then the labels will also be aligned by their baselines. The default is \ht\strutbox, which in most TEX formats is the height of a typical line of text in the current font. Note that the value of tnheight is not evaluated until it is used.

The positioning is similar for labels that go below a node. The label is given a minimum *depth* of `tndepth`. For labels positioned above or below, the horizontal reference point of the label, i.e., the point in the label directly above or below the center of the node, is set by the `href` parameter.

When labels are positioned on the left or right, the right or left edge of the label is positioned distance `tnsep` from the node. The vertical point that is aligned with the center of the node is set by `tnyref`. When you leave this empty, `vref` is used instead. Recall that `vref` gives the vertical *distance* from the baseline. Otherwise, the `tnyref` parameter works like the `yref` parameter, giving the fraction of the distance from the bottom to the top of the label.

8. Details

PSTricks does a pretty good job of positioning the nodes and creating a box whose size is close to the true bounding box of the tree. However, PSTricks does not take into account the node connections or labels when calculating the bounding boxes, except the tree node labels.

If, for this or other reasons, you want to fine tune the bounding box of the nodes, you can set the following parameters to a dimension:

name	*default*
`bbl`	0pt
`bbr`	0pt
`bbh`	0pt
`bbd`	0pt
`xbbl`	0pt
`xbbr`	0pt
`xbbh`	0pt
`xbbd`	0pt

The "'x"' versions increase the bounding box by <dim>, and the others set the bounding box to the dimension. There is one parameter for each direction from the center of the node, **l**eft, **r**ight, **h**eight, and **d**epth.

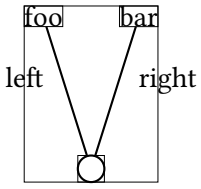
These parameters affect trees and nodes, and subtrees that switch directions, but not subtrees that go in the same direction as their parent tree (such subtrees have a profile rather than a bounding box, and should be adjusted by changing the bounding boxes of the constituent nodes).

Save any fiddling with the bounding box until you are otherwise finished with the tree.

You can see the bounding boxes by setting the `showbbox=true/false`parameter to `true`. To see the bounding boxes of all the nodes in a tree, you have to set this parameter before the tree.

In the following example, the labels stick out of the bounding box:

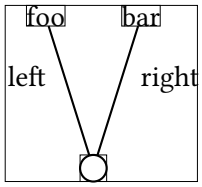
showbox




```
\psset{tpos=.6,showbbox=true}
\pstree[treemode=U]{\Tc{5pt}}{%
    \TR{foo}^{left}
    \TR{bar}_{right}}
```

Here is how we fix it:

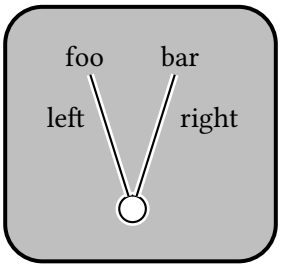
showbbox




```
\psset{tpos=.6,showbbox=true}
\pstree[treemode=U,xbbl=8pt,xbbr=14pt]{\Tc{5pt}}{%
    \TR{foo}^{left}
    \TR{bar}_{right}}
```

Now we can frame the tree:

\psframebox




```
\psframebox[fillstyle=solid,fillcolor=lightgray,framesep=14pt,
  linearc=14pt,cornersize=absolute,linewidth=1.5pt]{%
  \psset{tpos=.6,border=1pt,nodesepB=3pt}
  \pstree[treemode=U,xbbl=8pt,xbbr=14pt]{%
    \Tc[fillcolor=white,fillstyle=solid]{5pt}}{%
    \TR*{foo}^{left}
    \TR*{bar}_{right}}}
```

We would have gotten the same result by changing the bounding box of the two terminal nodes.

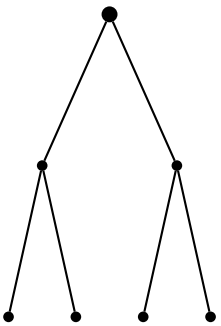
9. The scope of parameter changes

`edge` is the only parameter which, when set in a tree node's parameter argument, affects the drawing of the node connection (e.g., if you want to change the `nodesep`, your edge has to include the parameter change, or you have to set it before the node).

As noted at the beginning of this section, parameter changes made with `\pstree` affect all subtrees. However, there are variants of some of these parameters for making local changes, i.e, changes that affects only the current level: `thistreesep`, `thistreenodesize`, `thistreefit=tight/loose`, and `thislevelsep`.

For example:

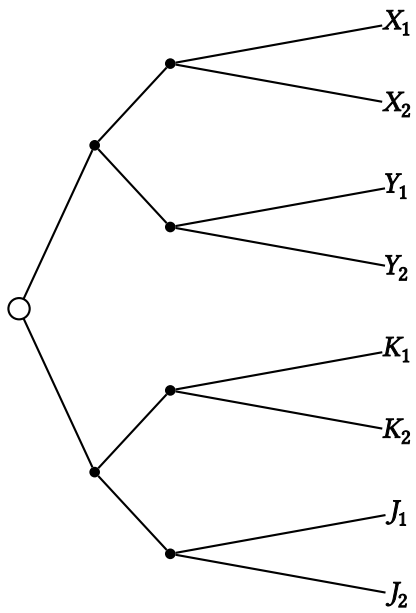
Macro \TC*



```
\pstree[thislevelsep=.5cm,thistreesep=2cm,
  radius=2pt]{\Tc*{3pt}}{%
  \pstree{\TC*}{\TC* \TC*}
  \pstree{\TC*}{\TC* \TC*}}
```

There are some things you may want set uniformly across a level in the tree, such as the `levelsep`. At level `<n>`, the command `\pstreehook<roman(n)>` (e. g., `\pstreehookii`) is executed, if it is defined (the root node of the whole tree is level 0, the successor tree objects and the node connections from the root node to these successors is level 1, etc.). In the following example, the `levelsep` is changed for level 2, without having to set the `thislevelsep` parameter for each of the three subtrees that make of level 2:

Keyword thislevelseo




```
\[
\def\pstreehookiii{\psset{thislevelsep=3cm}}
\pstree[treemode=R,levelsep=1cm,radius=2pt]{\Tc{4pt}}{%
  \pstree{\TC*}{%
    \pstree{\TC*}{\Tr{X_1} \Tr{X_2}}
    \pstree{\TC*}{\Tr{Y_1} \Tr{Y_2}}}
  \pstree{\TC*}{%
    \pstree{\TC*}{\Tr{K_1} \Tr{K_2}}
    \pstree{\TC*}{\Tr{J_1} \Tr{J_2}}}}
\]
```

Part II.
Theory

This is a description of a recursive alignment algorithm that is useful for drawing trees and tree-like graphs. It is a generalization of the algorithm in [5]. The purpose of the algorithm is to recursively construct a description of a *tree* in a high-level graphics language with the capabilities of PostScript. Thus, the algorithm is a preprocessor, and the graphics interpreter is a postprocessor. This division makes the algorithm simpler and more modular. The postprocessing could be implemented internally, if a low-level graphics description is required.

Thanks to: Ed Reingold

10. Introduction

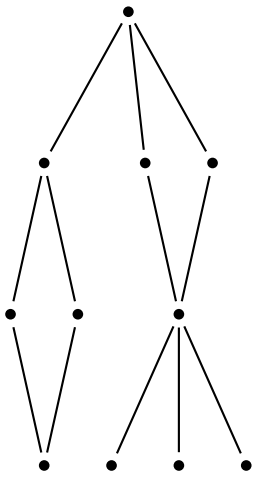
A tree is a collection of nodes, organized into levels, with each node's center assigned a coordinate position. The center of a node is where edges should point to. Trees have ragged left and right profiles, because the widths of the levels vary. In *horizontal mode*, the algorithm joins trees side by side, aligned by their top levels and fitted together tightly. In *vertical mode*, the algorithm stacks trees so that the nodes at the bottom level of the each tree are centered above the nodes at the top level of the next tree.

The algorithm is implemented in `pst-tvz`, which is part of the PSTricks package. PSTricks is a collection of PostScript extensions to TEX. The examples in this paper use the PSTricks implementation. The syntax of the input file is:

```
\psTree
  ~tree objects~ \\
  ~tree objects~ \\
  ...
  ~tree objects~
\endpsTree
```

Each row except for the last ends in \\. Each row is processed in horizontal mode, and then the rows are stacked in vertical mode. See next example.

Stacked nodes



```
\psTree[radius=2pt,nodesep=3pt]
  \TC* \\
  \psTree
    \TC* \\
    \TC* \TC* \\
    \TC*
  \endpsTree
  \psTree
    \TC* \TC* \\
    \TC* \\
    \TC* \TC* \TC*
  \endpsTree
\endpsTree
```

11. The graphics description

The graphics language should have whatever features one needs to draw the nodes, edges and labels, plus the ability to define procedures and variables for later reference. Furthermore, the graphics state should keep track of

a current point, which can be manipulated as follows:

1. Operators `gsave` and `grestore`, respectively, push the current point onto a stack and pop the top current point from that stack.

2. The operator ***x y*** `RMOVETO` shifts the current point ***x*** units to the right and ***y*** units down.

Note the convention that the ***y***-direction is *down*.

The tree graphics description should place (the center of) the top-left node at the current point, and should not change the current point.

The graphics description consists of these operators plus nodes, node labels, edges and edge labels. Here is what these objects do:

Node Draws the node, without changing the current point, and defines a procedure, identified by the node's name, that can answer queries about where to draw edges. For example, in PSTricks the nodes can report the coordinate of the center of the node, and the coordinate of the boundary of the node in any direction from the center.

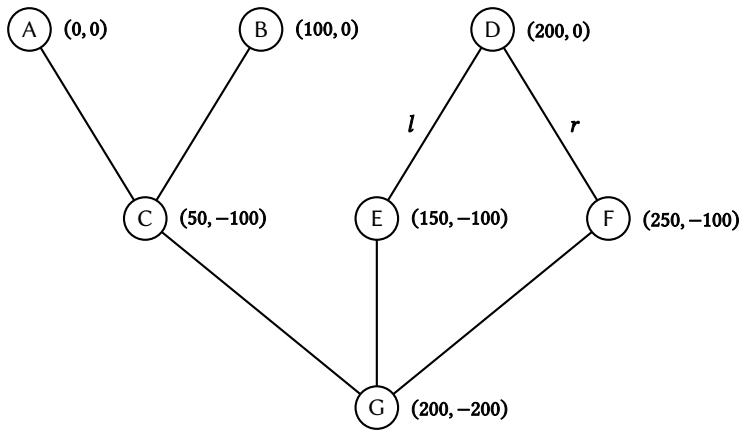
Node label Draws a label at a node, by querying the node to find out where to position the label.

Edge Draws a line between two nodes, querying the nodes to find out where to connect the lines, and then defines a procedure for finding the coordinate and slope at any point on the line.

Edge label Puts a label on an edge, using the procedure for finding the coordinate and slope of a point on the last edge that was drawn.

Labels

```
\def\sm{\rmfamily\scriptsize}
\footnotesize\sffamily
\psTree[radius=8pt,treesep=2.5cm,levelsep=2.5cm]
  \psTree
    \TCircle{A}\nput{r}{\pssucc}{\sm $(0,0)$}\TCircle{B}\nput{r}{\pssucc}{\sm $(100,0)$}
      \\
    \TCircle{C}\nput{r}{\pssucc}{\sm $(50,-100)$}
  \endpsTree
  \psTree
    \TCircle{D}~[tnpos=r]{\sm $(200,0)$}
      \\
    \TCircle{E}^{$l$}\nput{r}{\pssucc}{\sm $(150,-100)$}\TCircle{F}~[tnpos=r]{\sm $(250,-100)$}_{$r$}
  \endpsTree
  \\
  \TCircle{G}~[tnpos=r]{\sm $(200,-200)$}
\endpsTree
```



Suppose we want to draw the graph in the above example. We start by constructing the code for the subgraph containing nodes A, B and C. The first row (nodes A and B) is:

```
gsave
```

```
  ~Node A~
  100 0 rmoveto
  ~Node B~
grestore
```

and the second row is:

```
gsave
  ~Node C~
  ~Line from Node A to Node C~
  ~Line from Node B to Node C~
grestore
```

Then we calculate that the top-left node (node C) of the second row is positioned at $(50, 100)$ from the top-left node (node A) of the top row. The subgraph is thus:

```
gsave
  gsave
    ~Node A~
    100 0 rmoveto
    ~Node B~
  grestore
  50 100 rmoveto
  gsave
    ~Node C~
    ~Edge from Node A to Node C~
    ~Edge from Node B to Node C~
  grestore
grestore
```

Similary, the subgraph for nodes D, E and F is:

```
gsave
  gsave
    ~Node D~
  grestore
  -50 100 rmoveto
  gsave
    ~Node E~
    ~Edge from Node A to Node C~
    ~Edge label~
    ~Node F~
    ~Edge from Node B to Node C~
    ~Edge label~
  grestore
grestore
```

To join these two subgraphs, we calculate that the distance from the top-left node of $\{A, B, C\}$ to the top-left node of $\{D, E, F\}$ is $(200, 0)$. Thus, the subgraph $\{A, B, C, D, E, F\}$ is

```
gsave
  ~Subgraph A,B,C~
  200 0 rmoveto
  ~Subgraph D,E,F~
grestore
```

The code for the the bottom row (node G) is:

```
gsave
  ~Node G~
  ~Edge from Node C to Node G~
  ~Edge from Node E to Node G~
  ~Edge from Node F to Node G~
grestore
```

This node is positioned distance $(150, 200)$ from the top-left node of subgraph $\{A, B, C, D, E, F\}$, and so the code for the whole graph is

```
gsave
  gsave
    ~Subgraph A,B,C~
    rmoveto(200,0)
    ~Subgraph D,E,F~
  grestore
  150 200 rmoveto
  gsave
    ~Node G~
    ~Edge from Node C to Node G~
    ~Edge from Node E to Node G~
    ~Edge from Node F to Node G~
  grestore
grestore
```

12. Language requirements

I assume that the preprocessing language has operators `BEGINGROUP` and `ENDGROUP` that keep changes to variables local to the group, and `GLOBAL` which make the next change global.

There must be enough memory to hold the entire description of the tree in memory, because the algorithm constructs the description recursively rather than linearly.

I use the following data types:

integer	INT
boolean	BOOL
string	STRING
dimension	DIM
list of strings	LOS
list of dimensions	LOD

Dimensions might be integers or reals, depending on the implementation. The algorithm only uses integer arithmetic.

13. Accounting

As seen in Section 11, joining subtrees is mainly a problem of finding the distance between them. If we simply joined them by inserting a fixed amount of space between their bounding boxes (the way TEX builds boxes from boxes) then we would only need to know each subtree's bounding box. Instead, for horizontal mode we need to keep track of the different sizes of the levels (the profiles). For alignment in vertical mode, we also need to know the positions of the extreme nodes in the top and bottom levels. For automatic drawing of edges, we need to keep track of the names of the nodes at the bottom level (which the top nodes of the next level draw edges to). We keep track of a few more items that are used by some of the special features described in Section 16.

Here is the list of the tree data. (The distance between nodes refers to the distance between the centers of the nodes.) There is some redundancy, because it can be faster to keep track of information in the form it is needed rather than extracting it from other information.

treecode The graphics description of the tree. (DIM)

width The distance from the top-left node to the top-right node. (DIM)

leftprofile The horizontal distance from the left edge of the bounding box of each level to the top-left node. (LOD)

rightprofile The horizontal distance from the top-right node to the right edge of the bounding box of each level. (LOD)

leftbase The horizontal distance from the bottom-left node to the top-left node. (DIM)
rightbase The horizontal distance from the top-right node to the bottom-right node. (DIM)
center The distance from the top-left node to the center of the top level (for alignment in vertical mode), or NULL if the center should be the midpoint between the top-left and the top-right nodes. (DIM)
centerbase The distance from the top-left node to the center of the bottom level (for alignment in vertical mode), or NULL if the center should be the midpoint between the bottom-left and bottom-right nodes. (DIM)
height The vertical distance from the top of the bounding box to the top level. (DIM)
depth The vertical distance from the top level to the bottom of the bounding box. (DIM)
leftsize The horizontal distance from the left side of the bounding box to the top-left node. (DIM)
rightsize The horizontal distance from the top-right node to the right side of the bounding box. (DIM)
rootnodes A list of the names of the top-level nodes. (LOS)
basenodes A list of the names of the bottom-level nodes. (LOS)
cumlevelsep The distance between the first and last levels. (DIM)
numlevels The number of levels in the tree. (INT)
levelsizes The list of the height and depth of the bounding box of each level, plus, for every level except the last, the vertical distance to the next level.

See Figure 2.

14. Horizontal mode

In horizontal mode, the trees are aligned by their toplevels (i. e., a tree's baseline is the center of its top level). We add trees to the row one-by-one, updating the description of the row each time.

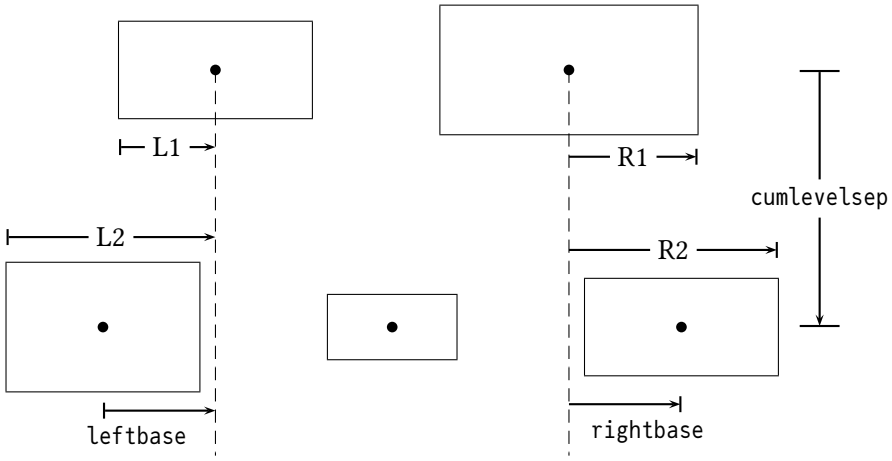
A row, while under construction, is itself a tree, and each time we add a tree we update the data for the row. As we construct the graphics description for the row, the current point is left at the top-left node of the last tree. We keep track of the width of the last tree (Lastwidth). Each time we add a tree to the row, we face the canonical problem of determining how much space to leave between the top-right node of the row and the top-left node of the next tree.

To distinguish the tree data variables of the row from those of the next tree to be added to the row, we begin the variable names for the row with capital letters. E. g., Leftprofile is the leftprofile of the row, and leftprofile is the leftprofile of the next tree.

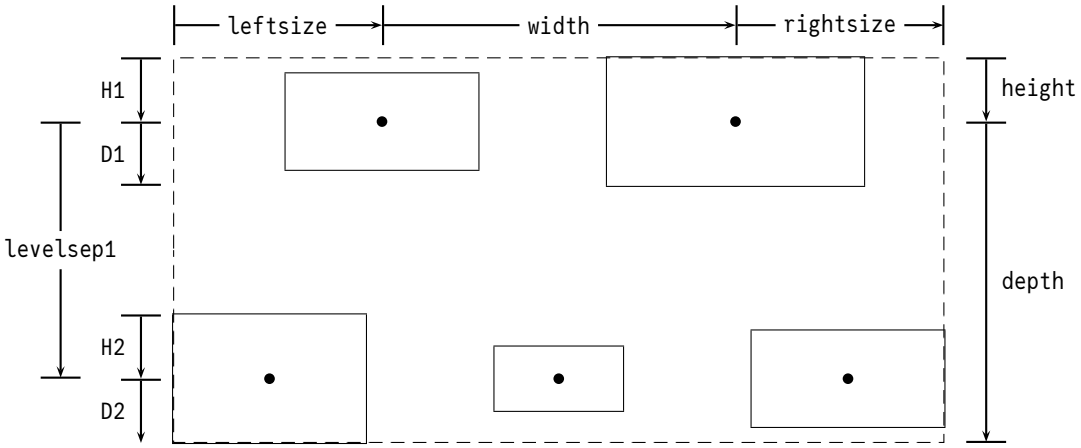
When adding the first tree object, we have to simply initialize the row's variables:

```
Treecode  = treecode
Width     = width
Lastwidth = width
Leftprofile = leftprofile
Rightprofile = rightprofile
Leftbase  = leftbase
Rightbase = rightbase
Center    = center
Centerbase = centerbase
Height    = height
Depth     = depth
Leftsize  = leftsize
Rightsize = rightsize
Rootnodes = rootnodes
Basenodes = basenodes
Cumlevelsep = cumlevelsep
Numlevels = numlevels
Levelsizes = levelsizes
```

For subsequent tree object's, we first find the distance between the top-right node of the current row and the top-left node of the next object, and we assign the result to sep. We want the minimum distance between the objects, level-by-level, to be treesep (a parameter):




```
leftprofile  = { L1 , L2 }
rightprofile = { R1 , R2 }
```




```
levelsizes = { H1 , D1 , levelsep1 , H2 , D2 }
```

Figure 2: Tree data

```
sep = MAX { Rightprofile ++ leftprofile } + treesep
```

where ++ makes a list by adding two lists item-by-item, up to the length of the shortest list.
Now we can add the new tree's code to the row's code:

```
Treecode = CONCAT
        {
            Treecode
            sep + Lastwidth 0 rmoveto
            treecode
        }
```

Then we update the row description. First we set `Width` to the distance from the top-left node of the row to the top-left node of the next tree (`Width+sep`) and we set `sep` to the distance from the top-right node of the previous tree to the top-right node of the next tree (`sep+width`), because these quantities are used in the calculations of the other row variables. At the end, we set `Width` to the actual width of the row (`Width+width`).

```
Width     = Width + sep
sep       = sep + width
Lastwidth = width
Leftprofile = BIMAX { Leftprofile , leftprofile - Width }
Rightprofile = BIMAX { Rightprofile - sep , rightprofile }
IF   Numlevels < numlevels
THEN Leftbase = Leftbase - Width
FI
Rightbase = IF  Numlevels > numlevels
            THEN Rightbase - sep - width
            ELSE rightbase
            FI
Height    = MAX { Height , height }
Depth     = MAX { Depth , depth }
Leftsize  = MAX { Leftsize , leftsize - Width }
Rightsize = MAX { Rightsize - sep , rightsize }
Rootnodes = CONCAT { Rootnodes , rootnodes }
IF   center = NULL
ELSE Center = center + Width
FI
IF   Numlevels < numlevels OR ( Numlevels = numlevels
         AND NOT centerbase = NULL )
THEN Centerbase = centerbase + Width
FI
IF   Numlevels < numlevels
THEN Basenodes = basenodes
ELSE IF  Numlevels = numlevels
     THEN Basenodes = CONCAT { Basenodes , basenodes }
     FI
FI
Numlevels = MAX { Numlevels , numlevels }
Levelsizes = BIMAX { Levelsizes, levelsizes }
Width     = Width + width
```

The updating that depends on `Numlevels` and `numlevels` can be summarized:

```
IF   Numlevels < numlevels
THEN Leftbase = leftbase - Width
     Centerbase = centerbase + Width
     Rightbase = rightbase
     Basenodes = basenodes
     Cumlevelsep = cumlevelsep
ELSE IF  Numlevels = numlevels
     THEN Basenodes = CONCAT { Basenodes , basenodes }
          Rightbase = rightbase
```

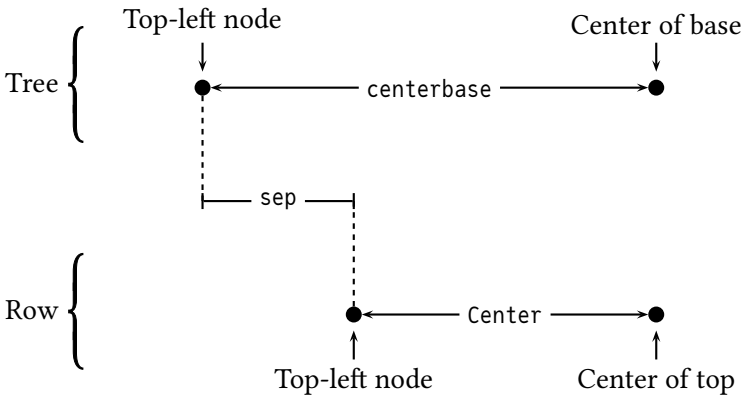


Figure 3: Aligning rows in vertical mode.

```
        IF   centerbase = NULL
        ELSE Centerbase = centerbase + Width
        FI
    ELSE Rightbase = Rightbase - sep
    FI
FI
```

Nodes are treated in the same way. A node is a trivial tree. It is completely described by its nodeleftsize (distance from the center to the left side of the bounding box), noderightsize, nodeheight, nodedepth and name. Here is the value of all the tree object variables in terms of the leftsize, rightsize, height, depth and name:

```
treecode  = {~node~}
width     = 0
leftprofile = {nodeleftsize}
rightprofile = {noderightsize}
leftbase  = 0
rightbase = 0
center    = NULL
centerbase = NULL
height    = nodeheight
depth     = nodedepth
leftsize  = nodeleftsize
rightsize = noderightsize
rootnodes = {name}
basenodes = {name}
cumlevelsep = 0
numlevels = 1
levelsizes = {height,depth}
```

15. Vertical mode

Here is the description of vertical mode. We also add rows one at a time, updating the description of the tree each time. Each row is just a tree object, and a partially completed tree is just a tree object. Therefore, the problem of joining rows is just the canonical problem of stacking two tree objects.

The description variables of the row begin with capital letters, and so we revert to uncapitalized names for the description variables of the tree.

When adding the first row, we simply have to initialize the tree's variables, setting `treecode=Treecode`, etc.

To add the subsequent rows, we first have to find the horizontal displacement of the top-left node of the next row from the top-left node of the tree. We chose this displacement so that the `centerbase` of the tree is aligned with the `Center` of the row, as shown in Figure 3.

First, calulate `centerbase` and `Center` if these are `NULL`:

```
IF   centerbase = NULL
THEN centerbase = ( width + rightbase - leftbase ) / 2
FI
IF   Center = NULL
THEN Center = width / 2
FI
```

Then set sep to the horizontal distance between the top-left nodes of the tree and row:

```
sep = centerbase - Centerbase
```

Next we calculate the vertical displacement. Each time we add a row, the current point ends up at the top-left node of the last row. We save the Cumlevelsep of the last row as lastcumlevelsep. The distance from the bottom level of the tree and the top level of the next row is the canonical distance between levels, levelsep, which is a parameter. Hence, the total displacement is

```
lastcumlevelsep + levelsep
```

Thus, we add the new row's code to the tree's code with:

```
treecode = CONCAT {
             treecode
             sep lastcumlevelsep+levelsep rmoveto
             Treecode
        }
```

Now we have to update the description. At first, cumlevelsep is set to the distance from the top level of the tree to the top level of the next row (cumlevelsep + levelsep) and rightsep is set to the horizontal distance from the top-right node of the tree to the top-right node of the next row (sep + Width - width), because these are used when updating the other variables. At the end, cumlevelsep is set to the actual cumlevelsep (cumlevelsep + Cumlevelsep).

```
cumlevelsep = cumlevelsep + levelsep
lastcumlevelsep = Cumlevelsep
rightsep    = sep + Width - width
leftprofile = CONCAT { leftprofile , Leftprofile - sep }
rightprofile = CONCAT {
                   rightprofile ,
                   Rightprofile + rightsep )
             }
leftbase    = Leftbase - sep
rightbase   = Rightbase + rightsep
centerbase  = IF  Centerbase=NULL
              THEN NULL
              ELSE Centerbase - sep
              FI
height      = MAX { height , Height - cumlevelsep }
depth       = MAX { depth , Depth + cumlevelsep }
leftsize    = MAX { leftsize , Leftsize - sep }
rightsize   = MAX { rightsize , Rightsize + rightsep }
rootnodes   = rootnodes
numlevels   = numlevels + Numlevels
levelsizes  = CONCAT { levelsizes , levelsep , Levelsizes }
cumlevelsep = cumlevelsep + Cumlevelsep
```

16. Bells and whistles

e also need to keep track of the list of nodes in the tree object, and the coordinates of the nodes. We can measure the coordinates relative to the top-left node. Then when we join two tree objects, we find the top-left node of the new object, join the lists of nodes, and update the coordinates with respect to the top-left node. This is simplified

by the fact that once a tree object has been formed, the relative position of the nodes within that object does not change when the object is nested inside another tree object.

I have so far described the algorithm assuming that the objects in a row are joined from left to right, and then the rows are stacked from top to bottom, and I will continue to use this convention throughout. However, the algorithm is the same when the tree objects grow in different directions; all that differs in pst-tvz is how one joins tree objects. For example, after calculating the distance between the top-left nodes of two tree objects, do we position the second object below, to the right, above or to the left of the first object?

pst-tvz uses a key=value system for controlling the algorithm. Keys are called *parameters*. Here are the parameters that control the direction in which the tree is constructed:

treemode The treemode is the direction in which trees grow (in which rows are stacked). The value is stored as an integer:

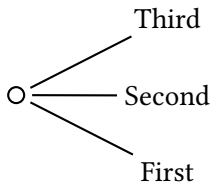
```
down -> 0
right -> 1
up   -> 2
left -> 3
```

In vertical trees (treemode is even), the rows are horizontal. In horizontal trees (treemode is odd), the rows are vertical.

treeflip treeflip is a boolean that sets the direction in which rows are constructed. When false, the horizontal rows of vertical trees are constructed from left to right (in the order in which objects appear in the input file), and the vertical rows of horizontal trees are constructed from top to bottom. When true, the rows of vertical trees are constructed from right to left, and the rows of horizontal trees are constructed from bottom to top.

For example:

treemode




```
\psTree[treemode=R,treeflip=true,nodesep=3pt]
  \Tc{3pt} \\
  \Tr{First} \Tr{Second} \Tr{Third}
\endpsTree
```

There are several methods for setting this distance.

If the ”treesep*” parameter has been set, then

```
sep = treesep*
```

That is, the spacing between the centers of the nodes (and hence between edges) is fixed.

Otherwise, if the treefit parameter equals tight,

If instead treefit=loose , the distance between the tree objects’ bounding boxes be treesep. I. e.,

```
sep = MAX { Rightprofile } + MAX { leftprofile } + treesep
```

In summary:

```
sep = IF treesep* = NULL
    THEN IF  treefit = tight
        THEN MAX { Rightprofile ++ leftprofile } + treesep'
        ELSE MAX { Rightprofile } + MAX { leftprofile } + treesep
        FI
    ELSE treesep*
    FI
```

If both objects have more than one level, then increase sep by xtreesep:

```
IF   Numlevels > 1
THEN IF  numlevels > 1
```

```
    THEN ADVANCE sep BY xtreesep
    FI
FI
```

Positive values of xtreesep can be used to highlight the structure of the trees.
Finally, if the user inserts

```
\addtreesep{~dim~}
```

before a tree object, then dim is saved in the addtreesep variable, and we add this to sep:

```
IF  addtreesep = NULL
ELSE ADVANCE sep BY addtreesep
FI
```

```
Treecode = CONCAT {
           Treecode ,
           IFODD Treemode
           THEN IF  Treeflip=TRUE
                THEN 0 sep rmoveto
                ELSE 0 -sep rmoveto
                FI
           ELSE IF  Treeflip=TRUE
                THEN -sep 0 rmoveto
                ELSE sep 0 rmoveto
                FI
           FI
           ,
           treecode
       }
```

A node calculates its leftsize, rightsize, height, depth and name, and then invokes \node@makecanonical@tree, which does the assignment given above.

The assignment actually depends on the orientation of the row, because the node calculates its dimensions for an upright orientation. That is, the assignment given above is correct if the row is part of a horizontal tree that grows down and if the row adds objects from left to right.

Here is the general assignment of leftprofile, rightprofile, height and depth:

```
height    = IFCASE Treemode
                nodeheight
            OR  leftsize
            OR  nodedepth
            OR  rightsize
            FI
depth     = IFCASE Treemode
                nodedepth
            OR  rightsize
            OR  nodeheight
            OR  leftsize
            FI
leftsize  = IFODD Treemode
            THEN IF Treeflip=TRUE
                THEN nodedepth
                ELSE nodeheight
                FI
            ELSE IF Treeflip=TRUE
                THEN rightsize
                ELSE leftsize
                FI
            FI
rightsize = IFODD Treemode
```

```
            THEN IF Treeflip=TRUE
                THEN nodeheight
                ELSE nodedepth
                FI
            ELSE IF Treeflip=TRUE
                THEN leftsize
                ELSE rightsize
                FI
            FI
leftprofile = { leftsize, }
rightprofile = { rightsize, }
```

However, if the `treenodesize` is set, then the profile are set using this value as half the "width" of the node. That is:

```
IF   treenodesize = NULL
THEN leftprofile = { leftsize, }
     rightprofile = { rightsize, }
ELSE leftprofile = { treenodesize, }
     rightprofile = leftprofile
FI
```

Tree objects whose orientation is different from the row are given special treatment. If the object has the same direction, but a different flip, then we simply swap the left and right profiles, and related items:

```
IF   Treemode = treemode
THEN IF  Treeflip = treeflip
     ELSE temp     = leftprofile
          leftprofile = rightprofile
          rightprofile = temp
          temp      = leftbase
          leftbase  = rightbase
          rightbase = temp
          temp      = leftsize
          leftsize  = rightsize
          rightsize = temp
          center    = IF  center = NULL
                      THEN NULL
                      ELSE width - center
                      FI
          centerbase = IF centerbase = NULL
                      THEN NULL
                      ELSE width - centerbase
                      FI
     FI
FI
```

If the tree objects has a different direction, then we treat the object like a node, centered at the center of its top level

```
IF   Treemode = treemode
ELSE tree@makecanonical@node
     node@makecanonical@tree
FI
```

Here is the definition of `tree@makecanonical@node`:

```
IF   center = NULL
THEN center = width / 2
FI
IF   center = 0
ELSE SETBOX box =
     IFODD treemode
```

```
    THEN VBOX TO 0 BEGINBOX VSS
    ELSE HBOX TO 0 BEGINBOX HSS
    FI
        BOX box
        KERN IF Treeflip = TRUE THEN - FI center
    ENDBOX
FI
IF   treeflip = TRUE
THEN tempa = rightsize + width - center
    tempb = leftsize + center
ELSE tempa = leftsize + center
    tempb = rightsize + width - center
FI
IFODD treemode
THEN nodeheight = tempa
    nodedepth = tempb
ELSE leftsize = tempa
    rightsize = tempb
FI
IFCASE treemode
     nodeheight = height
     nodedepth = depth
OR   leftsize = height
     rightsize = depth
OR   nodeheight = depth
     nodedepth = height
OR   leftsize = depth
     rightsize = height
FI
```

We increase ”sep” by `treeshift`:

```
ADVANCE sep BY treeshift
```

Now we insert `levelsep` between the trees, move the row by `sep`, and add an extra space of `Cumlevelsep` so that the total size of the the tree is the same as `cumlevelsep`. With vertical trees we do this in a `\vtop` and in horizontal trees we do this in an `\hbox`. I. e.,

```
IFODD treemode
THEN HBOX
    {
        UNHBOX box
        IF   treeflip = TRUE
        THEN KERN - levelsep
            LOWER sep BOX hbox
            KERN - Cumlevelsep
        ELSE KERN levelsep
            RAISE sep BOX hbox
            KERN Cumlevelsep
        FI
    }
ELSE VTOP
    {
        UNVBOX box
        IF   treeflip = TRUE
        THEN KERN - levelsep
            MOVELEFT sep BOX hbox
            KERN - Cumlevelsep
        ELSE KERN levelsep
            MOVERIGHT sep BOX hbox
            KERN Cumlevelsep
        FI
    }
```

```
FI
```

17. The PSTricks implementation

In pst-tvz, we can let TEX keep track of nodes and node coordinates internally. We store each tree object in a TEX box with zero size, such that the current point is at the center of the top left node. We create a new tree object in a TEX box, inserting space between the component objects.

With TEX, we construct the row for both vertical and horizontal trees using an \hbox. In an \hbox, we can insert horizontal space to separate tree objects in a horizontal row, and we can lower or raise objects below or above to baseline to separate tree objects in a vertical row. For horizontal rows (vertical trees), the current insertion point is thus at the top-left node of the last object, and so we also need to know the width of this object. This value is stored in wsep after adding a tree object, and then wsep is set to the total distance between the top-left node of the last object and the top-left node of the current object. In vertical rows (horizontal trees). the current insertion point is at the top-left node of the row, and so we also need to know the width of the current row, but this is already stored in Width. We set wsep to the total distance between the top-left node of the row and the top-left node of the new object.

```
IFODD treemode
THEN wsep = Width + sep
ELSE wsep = wsep + sep
FI
```

We add the space and the tree object

```
IFODD treemode
THEN LOWER wsep
ELSE KERN wsep
FI
BOX box
```

and update the row description:

18. Examples

Here is how this information is used to position the successors. First, all terminal nodes are treated as single-level trees. Thus, the canonical successor is a subtree (that has the same orientation as the parent tree). The successors are positioned so that their centers line up horizontally. How the distance between successors is calculate depends on the values of two parameters:

- When ”treefit=tight”, the subtrees are positioned so that the minimum distance between levels is ”treesep”. This is calculated by adding the right profile of the current group of successors (this profile is with respect to the center of the right-most successor) to the left profile of the new successor item by item, finding the maximum of the resulting list, and then adding ”treesep”.
- When treefit=loose , the subtrees are positioned so that the distance between their bounding boxes is treesep.

- When also treenodesize is non-negative, the top level of each subtrees is given a width of <dim>, *for the purpose of fitting the subtrees together.*

After the row of successors is constructed, and its profiles, height and depth are calculated, the root object is positioned above the row of successors so that the center of the root object is centered between the centers of the first and last successors (although this can be modified). The distance between the root object and the row of successors is levelsep. The profiles, height and depth of the resulting tree are calculated from the dimensions of the root object and the dimensions of the row of successors.

The treemode parameter determines the direction in which the tree grows, and the treeflip parameter determines the direction in which successors are added. The values of these parameters together are called the tree’s

orientation. The terminology used above is for trees with the default orientation: `treemode=D` and `treeflip=false` (the tree grows down and successors are added from left to right). However, the logical structure of a tree does not depend on its orientation, and so we can use the same terminology and accounting system for all trees. Here is the correspondence between

- For a vertical tree that grows up, successors are also added from left to right, and so the profiles are as described above (although "top" levels are physically at the bottom of the tree). The "height" and "depth" of tree is the distance from the center to the physical bottom and top, respectively, of the bounding box.
- For a horizontal tree, successors are added from top to bottom. The "left" profiles are the physical top profiles, and the "right" profiles are the physical bottom profiles. The "height" and "depth" of a horizontal tree that grows to the right are the distances from the center of the tree to the left and right sides, respectively, of its bounding box. The opposite holds if the tree grows to the left.

- If `treeflip=true`, then successors are added in the opposite direction, and so the left and right profiles are switched. "Right" always refers to the direction in which new successors are added.

A root node can actually be a subtree, and a subtree can have a different orientation from its parent. Here is how we deal with these special cases:

The canonical root object is a node. Trees are converted to this canonical root object by calculating their bounding box, and thereby determining their height, width, left and right sizes.

The canonical successor is a subtree that has the same orientation as the parent tree. Nodes and trees that grow in other directions are converted to this canonical succcessor by treating them as single level trees. That is, the left profile is just the left size of the node or tree (with respect to the orientation of the parent tree), and the right profile is the right size of the node or tree.

In addition to being a root or successor of a tree, a tree object can be unnested, or "outer". The canonical outer object is a node, and it is made into a box whose dimensions are the size of the node. Trees are converted to this canonical outer. Hence, when a tree is outer, we only need to remember store its bounding box, and can forget about its profile.

A subtree that grows in the same direction (which is weaker than having the same orientation) is called a *proper* subtree. All other trees—outer, root, or subtrees that change directions—are not proper.

The three places a tree object can be found—root, successor or outer—are called *modes*. By an unfortunate historical accident, the directions trees grow—down, up, right and left—are also called modes. However, this should not cause confusion in the code.

The programming implementation of this algorithm is modular. Tree objects, which are either nodes or (sub)trees, save their dimensions in designated registers and commands. Then they invoke

```
\ps<object_type>@makecanonical
```

<object_type> is `node` or `tree`. This translates the object's dimensions into dimensions of a canonical object for the current mode. Then the object invokes

```
\ptr@build
```

which positions the box (`\ptr@box`) containing the object, also depending on the current mode.

Here are the registers and commands that a tree object must set before invoking `\ps<object_type>@makecanonical` and `\ps<object_type>@build`:

Nodes These are all dimension registers, and should not be set globally.

`\pst@dima`	left size
`\pst@dimb`	right size
`\pst@dimc`	height
`\pst@dimd`	depth

Trees `\PTR@height` and `\PTR@depth` are count registers, measuring sp units. The others are lists. These are set globally, so that a tree can use these commands and registers while it is being constructed. However, changes are actually kept local with respect to the structure of trees because he values in effect when the tree is started are restored at the end of the tree.

\PTR@height	height	
\PTR@depth	depth	
\PTR@leftprofile	left profile	ll
\PTR@rightprofile	right profile	
\PTR@levelsizes	level size	

The \ps<object_type>@makecanonical commands translate the values stored in the commands and registers listed above, and assign the results to the following commands and registers, for use by \ps<object_type>@build:

Outer mode Outer objects use \pst@dima, \pst@dimb, \pst@dimc and \pst@dimd, like nodes. These dimensions refer to the physical dimensions. I.e., they do not depend on the orientation of the object.

Root mode These are all commands.

\psroot@leftsize	left size
\psroot@rightsize	right size
\psroot@height	height
\psroot@depth	depth

Successor objects \ptr@height and \ptr@depth are counters, and the remaining are lists.

\ptr@height	height	
\ptr@depth	depth	
\ptr@leftprofile	left profile	ll
\ptr@rightprofile	right profile	
\ptr@levelsizes	level size	

Except for \pst@dima, etc., used for the dimensions of nodes, root and outer objects, all values are stored as integers, giving the distance in sp units. Much of the accounting is done using counters and sp units, because this is more efficient and because counters are not quite as scarce as dimension registers.

When a subtree is made canonical, we need to know the orientation of both the subtree and the parent tree. We use \psk@Treemode and \if@Treeflip to keep track of the orientation of the parent tree. These are set by the parent tree when it begins to process the row of successors. This information is not needed by root objects.

When the value of the levelsep parameter is preceded by *, the size of the levels is taken into account when setting the distance between levels. This information is only known after the tree has been constructed, because levels extend beyond the recursive structure of the trees. That is, the distance between levels of one subtree will depend on the distance be levels of other (disjoint) subtrees. Therefore, we write this information to an auxilary file, to be read the next time the main input file is processed. Each level in a tree must have a unique identifier, so that a subtree can find the distance between levels by looking up the value for its tree and level. The count register \ptr@ID is used to indentify trees, and the count register \ptr@levelID is used to identify the levels in a tree.

The logical compenents of a tree do not coincide with the physical components when subtree appears as a root tree, or as a successor to a parent with a distinct orientation. E. g., there is no point in trying to synchronize the distance between levels of two subtrees that grow in different directions. Hence, in each of these cases, the \ptr@ID is advanced so that for the purpose of determining the distance between levels, the subtree has a different indentifier from its parents. For a subtree that appears as a successor, the identifier should be changed when \psk@treemode does not equal \psk@Treemode. So that this test works in outer and root mode, the \psk@treemode is set to **−1** in these modes (the treemode is saved as 0, 1, 2 or 3).

Because the value of \ptr@ID can change globally within a tree, a tree's identifier is saved as \ptr@id immediately after \ptr@ID is incremented. \ptr@id is an ordinary command.

In addition to not relying on TEX boxes to do all the accounting, we cannot rely on TEX grouping to do keep values of certain commands and registers local. This is because the successors, which are in their own TEX group, must communicate information to the parent tree as it constructs the row of successors (e. g., by modifying the parent tree's \PTR@height). We get around this by saving and restoring the values of certain commands and paramters just before and after processing the row of sucessors.

There are some special features whose implementation is incorporated into the makecanonical and build commands, for efficiency:

Adjust bounding boxes Bounding boxes are only adjusted for nodes or for a tree that is an outer or root object or that changes directions. Such trees are first made into nodes, via \ptr@makecanonical@outer, and it is at the end of this command that bounding box adjustment is invoked. The bounding box adjustment is invoked

by nodes just before \psnode@makecanonical.

Show bounding boxes The show bounding box commands are invoked as follows:

- For nodes, just after the bounding box adjustment.
- For trees that invoke \ptr@makecanonical@outer, just after the bounding box adjustment.
- For subtrees that have the same treemode as their parent, at the beginning of the \ptr@makecanonical@succ command.

Skip levels The commands for finding the amount of space to be skipped and the profiles of the skipped levels are invoked at the beginning of the tree macros, or in \psnode@makecanonical@succ. The adjustment of the box and profiles takes place in \ptr@build@succ.

19. List of all optional arguments for pst-thick

Key	Type	Default
treemode	ordinary	D
treeflip	boolean	true
root	ordinary	\TC
treesep	ordinary	0.75cm
thistreesep	ordinary	[none]
xtreesep	ordinary	0.75cm
thisxtreesep	ordinary	[none]
treenodesize	ordinary	-1pt
thistreenodesize	ordinary	-1pt
treefit	ordinary	tight
thistreefit	ordinary	tight
treerep	ordinary	1
bbl	ordinary	[none]
bbr	ordinary	[none]
bbh	ordinary	[none]
bbd	ordinary	[none]
xbbl	ordinary	[none]
xbbr	ordinary	[none]
xbbh	ordinary	[none]
xbbd	ordinary	[none]
showbbox	boolean	true
levelsep	ordinary	2cm
thislevelsep	ordinary	[none]
varlevelsep	boolean	true
treeshift	ordinary	0
skiplevels	ordinary	0
unary	ordinary	middle
thisunary	ordinary	middle
leafalign	ordinary	true
edge	ordinary	\ncline
skipedge	ordinary	
fansize	ordinary	1cm
tnsep	ordinary	
tnyref	ordinary	
tnheight	ordinary	\ht \strutbox
tndepth	ordinary	\dp \strutbox
tnpos	ordinary	

References

[1] Denis Girou. “Présentation de PSTricks”. In: *Cahier GUTenberg* 16 (Apr. 1994), pp. 21–70.

[2] Michel Goosens et al. *The LaTeX Graphics Companion*. second. Boston, Mass.: Addison-Wesley Publishing Company, 2007.

[3] Michel Goossens et al. *The LaTeX Graphics Companion: Illustrating Documents with TeX and PostScript*. Tools and Techniques for Computer Typesetting. Reprint of the second edition from Addison Wesley. Heidelberg and Berlin: Lehmanns Media, 2022, pp. xxi + 975.

[4] Nikolai G. Kollock. *PostScript richtig eingesetzt: vom Konzept zum praktischen Einsatz*. Vaterstetten: IWT, 1989.

[5] Edward Reingold and John Tilford. “Tidier Drawings of Trees”. In: *IEEE Transactions on Software Engineering* SE-7.2 (1981).

[6] Herbert Voß. *PSTricks – Grafik für TEX und LATEX*. fifth. Heidelberg/Hamburg: DANTE – lehmanns media, 2010.

[7] Herbert Voß. *PSTricks – Graphics for LATEX*. 1. Cambridge: UIT, 2011.

[8] Timothy Van Zandt. *multido.tex - a loop macro, that supports fixed-point addition*. CTAN:/macros/generic/multido.tex, 1997.

[9] Timothy Van Zandt and Denis Girou. “Inside PSTricks”. In: *TUGboat* 15 (Sept. 1994), pp. 239–246.

Index

Symbols

A

B

D

E

F

G

H

I

J

K