trees.sty – A Package for Typesetting Trees

Manuel Kauers

February 24, 2003

Abstract

This package provides LaTeX-macros for typesetting trees in an optimal way which makes optimal use of horizontal space. Tree nodes may contain arbitrary material, tree edges are composed of vertical and horizontal lines.

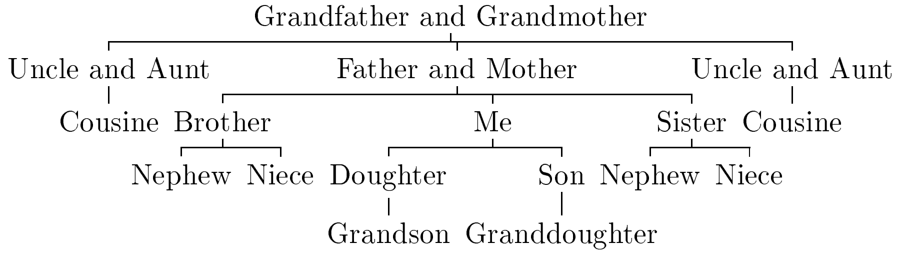
Contents

1 Usage

1.1 What the package can do

Trees?

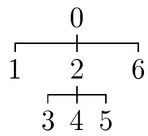
The purpose of this package is to provide the user with macros that allow him/her to typeset trees. A *tree* is recursive structure which is either a *node* or a node together with a set of (sub-)trees. It looks like this:



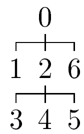
There are already packages for this purpose

Yes, but all of the packages that I was aware of at time of writing either wasted a lot of space, or they were to difficult to use, or they required to use packages which were in conflict with other packages I needed. After all, writing packages is kind of sports...

This package now aims to be both easy to use and smart in procuding nice output. A tree is considered nice if its horizontal expansion is small. For instance, most tree packages produce something like



because their implementation simply puts subtree next to subtree without considering shrinking the space between two neighbours if possible. When our package is asked to procuce the tree above, it will produce



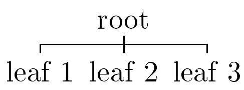
which is quite nice according to the above definition of "nice." In practice, trees tend to become very large, even reasonable trees don't fit on one page, that's why we regard the computation of small output as an important feature of a tree package.

The package does not support edges that go directly from one node to its parent – all tree edges are based on vertical lines which are centered with respect to the nodes they connect and with horizontal parts to connect the vertical parts accordingly.

1.2 How you get it to do it

A tree consists of a node and a list of subtrees where a subtree is a tree or a single node. In the latter case it is called a leaf. That's everything you need to know. Put the `trees.sty` file into a directory where TEX will find it, say `\usepackage{trees}` in the preamble, and you can continue like this:

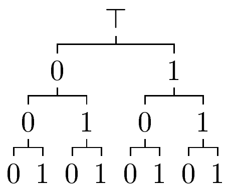
```
\tree{root}{%
 \leaf{leaf 1}
 \leaf{leaf 2}
 \leaf{leaf 3}
}
```



`\tree` `\leaf` The `\tree` command takes two arguments. The first argument contains the root of the tree and the second argument consists of an ordered bunch of subtrees which may themselves be trees or leaves. Here, `\leaf` is simply defined as `\tree{#1}{}` to save one `{}`.

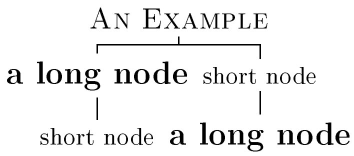
Here is an example with a tree of depth 3:

```
\tree{$\top$}{%
 \tree{0}{%
  \tree{0}{%
   \leaf{0}\leaf{1}}
  \tree{1}{%
   \leaf{0}\leaf{1}}}
 \tree{1}{%
  \tree{0}{%
   \leaf{0}\leaf{1}}
  \tree{1}{%
   \leaf{0}\leaf{1}}}}
```



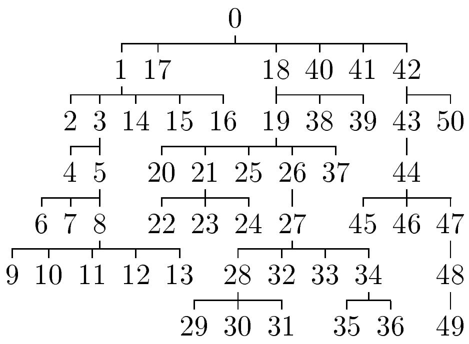
The next example shows how different subtrees come close to each other. The example also shows that different nodes may change font properties like size or shape. In case of different node sizes the package will also take care that all nodes of the same level share the same ground line.

```
\tree{\textsc{An Example}}{
 \tree{\large
     \textbf{a long node}}{
  \leaf{\footnotesize
      short node}}
 \tree{\footnotesize
      short node}{
  \leaf{\large
      \textbf{a long node}}}}
```



You might complain that the package does more than it should in the sense that in the result above the distance between the nodes of the same level are too small. Don't worry, things like this are subject to global objects which will be discussed in the next section.

We conclude this section with a last example



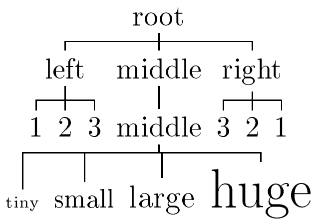
1.3 Customisation

The tree generation depends on certain parameters which can be modified by the user. In the examples of this section, we will assume that `\Tree` is a macro which expands to the following definition:

```
\tree{\fbox{root}}{%
  \tree{left}{%
   \leaf{1}\leaf{\fbox{2}}\leaf{3}}
  \tree{middle}{\tree{middle}{%
    \leaf{\tiny tiny}
    \leaf{\small small}
    \leaf{\large large}
    \leaf{\huge huge}
  }}
```

```
\tree{right}{%
 \leaf{3}\leaf{\fbox{2}}\leaf{1}}}
```

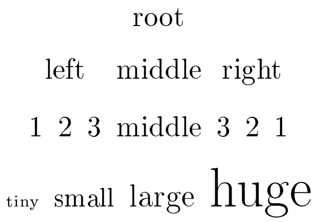
In the normal setting this gives the following tree:

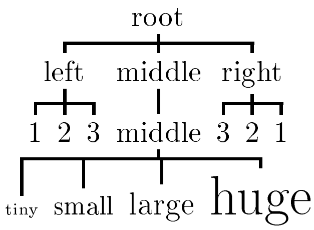


The edge width

`\edgewidth` The TEX length `\edgewidth` denotes the thickness of the lines which are used for drawing the edges. Its default value is `.3pt`.

```
\edgewidth=0pt\Tree
\edgewidth=1pt\Tree
```

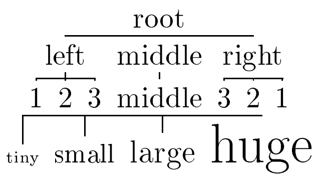


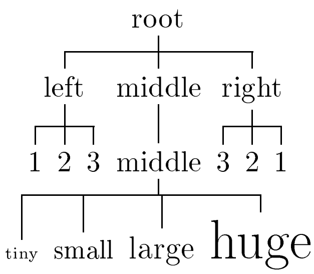


The level skip

`\edgeheight` The TEX length `\edgeheight` denotes half of the distance between two levels, i.e., the length of the vertical part of an edge which connects a node to the horizontal part of an edge. Here the default value is `3pt`.

```
\edgeheight=0pt\Tree
\edgeheight=6pt\Tree
```



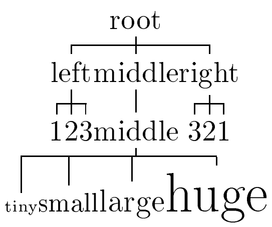


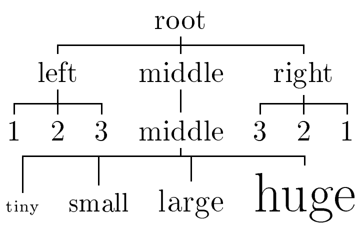
The left example shows that this value only specifies the desired length, in fact the minimum length. It might be stretched by the package in order to get nodes of different size to the same ground line.

The node distance

`\nodeskip` The TEX length `\nodeskip` denotes the desired amount of horizontal space between to consecutive nodes of the same level (regardless whether or not they belong to the same subtree). Again, this is only a minimum value which might be stretched locally by the package, if necessary. The default value is `.5em`.

```
\nodeskip=0pt\Tree
\nodeskip=1em\Tree
```

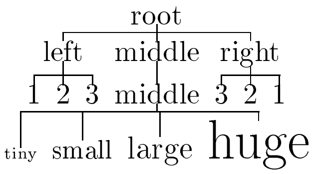


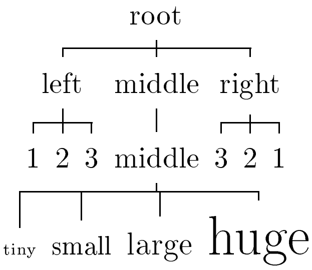


The node separation

`\nodesep` The `\nodesep` TEX length denotes the distance of the box containing a node and the start of the vertical part of the edge which eventually connects this note to the others. The default here is again `3pt`, and the package will not adjust this value locally as it does with others.

```
\nodesep=0pt\Tree
\nodesep=6pt\Tree
```





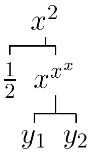
The node fontifier macro

\node The `\node` macro implements the template method design pattern. It is used by the package to typeset individual nodes. By its redefinition, it is for instance possible to define math mode trees

```
\def\mtree#1#2{{\def\node##1{$##1$}\tree{#1}{#2}}}
```

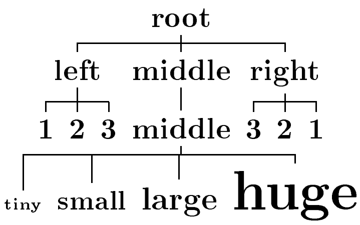
if desired. This allows to omit the math shifts when writing a tree whose nodes contain math material. (Note that only the outermost `\tree` needs to be changed to `\mtree`.)

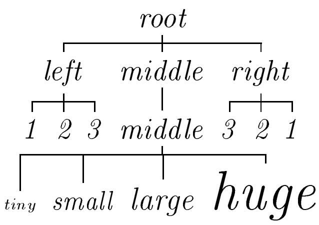
```
\mtree{x^2}{%
 \leaf{\frac{1}{2}}
 \tree{x^{x^x}}{%
  \leaf{y_1}\leaf{y_2}}}
```



We illustrate the usage of `\node` in the context of our `\Tree` example.

```
\def\node#1{\textbf{#1}}\Tree
\def\node#1{\textit{#1}}\Tree
```





The default definition is

```
\def\node#1{#1}
```

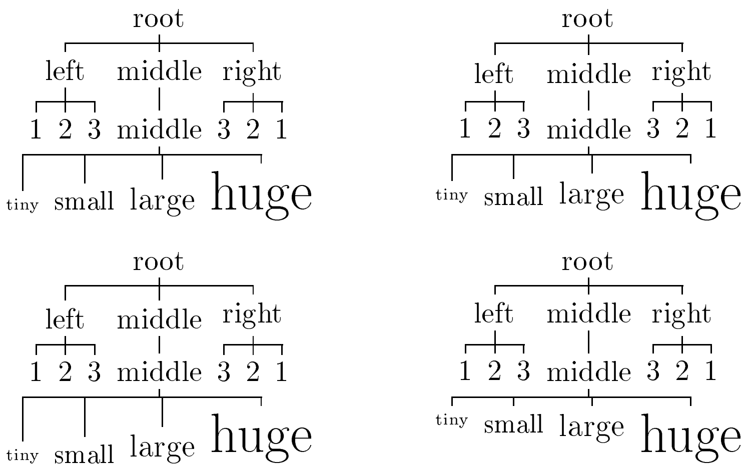
Vertical node orientation

By default, nodes of the same level are arranged such that they share the same base line. This gives by far the nicest result, however, you are free to change to a different alignment policy. You have the choice to align levels with respect to

1. their base line (by `\baselevels`, this is default),
2. their vertical center (by `\centerlevels`),
3. their bottom (by `\floorlevels`), or to
4. their top (by `\ceillevels`).

On the `\Tree` example this looks as follows:

```
\baselevels\Tree      \centerlevels\Tree
\floorlevels\Tree     \ceillevels\Tree
```



1.4 Bug parade

No bugs known. If you find one, please let me know: `manuel@kauers.de`.

2 Algorithmic Background

2.1 Basics

The typesetting of a tree proceeds in a divide and conquer approach. In order to typeset a tree t which has the root r and subtrees $t_1, \ldots, t_n$, first compute recursivley boxed for $t_1, \ldots, t_n$ and a box containing r. Then put the boxes of $t_1, \ldots, t_n$ into an `\hbox` s (with appropriate distances between them). From the width of t_1, t_n and s the length of the horizontal edge can be computed easily. Taking the leftmost point of s as a basis, the horizontal part of the edge to typeset starts at width $w(t_1)/2$ and has length $w(s) - (w(t_1) + w(t_2))/2$. The final box containing t will be a vertical box (actually a `\vtop`) containing the following elements:

1. A vertical rule which eventually connects t to its parent (omitted if t doesn't have a parent)
2. The root box r
3. A vertical rule connecting r with the horizontal part of the edge
4. the horizontal edge
5. the subtree array s

All elements except for the horizontal edge will be centered, the horizontal line will be put as described above.

This is the natural way to implement a tree algorithm in TEX, and it is also the idea underlying our implementation. However, quite a bit of work is necessary to tune this algorithm such that it produces nice results. There are essentially two problems:

1. Nodes of the same level need to be aligned. Only nodes of the first level of a tree are aligned properly, but two nodes of the same level but different subtrees might be shifted due to the existence of ancestors with different size.
2. Superfluous horizontal space between subtrees has to be removed.

The next two sections describe how the package deals with these two problems.

2.2 Synchronisation of Levels

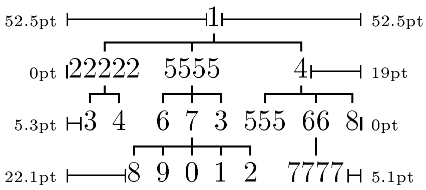
This is actually the easier problem. Before we typeset the tree, we walk through it to do some measurements. We determine the depth d of the tree and for each level l_i $(i = 1, \ldots, d)$ we measure the maximum height and the maximum depth of all the node boxes at level l_i.

In the second walk through the tree, a counter will keep track of the current level, and from the array computed in the first walk, we lengthen the vertical parts above and below a node such as to fill it to the height/depth of the level.

2.3 Removing Superfluous space

We equip each tree t of depth d with two arrays of dimension d called *skyline* — a left one describing the left side of the tree and a right one describing its right side. For the skyline, we assume that the tree is enclosed by a rectangle of minimum size, and specify that the left [right] skyline vector maps an index i to the distance between the leftmot [rightmost] point of the ith level of t to this rectangle. For convenience, we define out-of-bounds values to infinity.

The concept is illustrated in the following example.



In this example, the left skyline is [52.5pt, 0pt, 5.3pt, 22.1pt] and the right skyline is [52.5pt, 19pt, 0pt, 5.1pt].

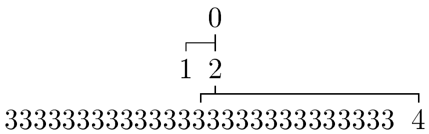
Given two skylines $s = [s_1, \ldots, s_n], s' = [s'_1, \ldots, s'_m]$, their *distance* $d(s, s')$ is defined by

$$d(s, s') = \min_{k=1}^{\min\{m,n\}} (s_k + s'_k).$$

Appending a value v to a skyline s is defined as usual list appending if $v \geq 0$. If $v < 0$, however, it is defined by first *shifting* s by $-v$ and then appending the value 0. Shifting a skyline s by $v \geq 0$ means adding v to each of elements in the array s.

The notion of a skyline gives access to the amount of space which we might delete when typesetting two consecutive subtrees t, t'. If $s_l(t)$ is the left skyline of t and $s_r(t')$ is the right skyline of t', then we consider the value $d = d(s_l(t), s_r(t'))$. If it is less than `\nodeskip`, we put `\nodeskip` horizontal space between them, otherwise we put `\nodeskip`$-d$ space, where a negative result means of course going to the left.

It remains to compute the skylines of a tree, but this is not too difficult. For a leaf there is nothing to do, both skylines are [0pt]. Assume the skylines are computed for the subtrees of a tree t, and we wish to compute the left skyline of t. The skyline of the leftmost subtree is a good starting point. The only nontrivial thing is to be aware of situations like in



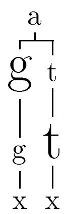
where the left skyline is also influenced by the other subtrees. This is done by looping over all subtrees $t_1, \ldots, t_n$. For each subtree t_i whose depth is higher than the maximum depth among the t_j ($j < i$), shift the left skyline of t_i by $w(t_1, \ldots, t_{i-1})+$`\nodeskip` (where $w(t_1, \ldots, t_{i-1})$ is meant to be the width of the partial subtree array including all spacings and shrinkings) and overlap it with the left skyline of t_1 which means append all overhanging elements. (Note the above definition of append in case they are negative.) Finally we incorporate the root by appending the value $(w(r) - w(s))/2$ where $w(r)$ is the width of the root and $w(s)$ is the width of the subtree array.

It is easy to see that the usage of horizontal space is minimized by this approach.

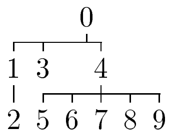
2.4 Wish List

Two more aspects could be dealt with, and might be incorporated into future versions of the package:

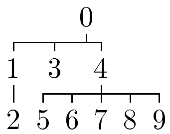
1. The skyline concept might also be used for vertical shrinking between two consecutive levels. In the following example, none of the vertical lines between the g/t levels has the desired minimum length `2\edgeheight`.



2. It might be that the position of the ith subtree in a list of subtrees is not yet uniquely determined by the constraints we have so far. If there is a degree of freedom, such subtrees should be placed centered with respect to their neigbours.



vs.



3. Edge labels would also be nice, although the style of the edges makes it difficult to find an appropriate place for them.

3 Implementation

The layout algorithm described in on the previous pages is of course simple enough to be implemented quickly in any higher programming language of choice. But to do in in TEX requires some work which to describe will be the content of the rest of this document.

```
\ProvidesPackage{trees}[2003/02/21]
\makeatletter
```

As usual, package private control sequences get a prefix. We choose the prefix `\tree@`.

3.1 Allocation of recources

Allocation of boxes...

```
\newbox\tree@rootbox
\newbox\tree@subtreebox
```

...dimensions...

```
\newdimen\tree@firstnode
\newdimen\tree@lastnode
```

```
\newdimen\tree@arraywidth
\newdimen\tree@width
\newdimen\tree@leftoverhang
\newdimen\tree@rightoverhang
\newdimen\tree@oldfirstnode
\newdimen\tree@oldlastnode
\newdimen\tree@oldarraywidth
\newdimen\tree@oldleftoverhang
\newdimen\tree@oldrightoverhang
```

...counters...

```
\newcount\tree@currentlevel
\newcount\tree@nosubtrees
\newcount\tree@oldnosubtrees
\newcount\tree@depth
```

Allocation and definition of public dimensions

```
\newdimen\edgewidth
\edgewidth=.3pt
\newdimen\edgeheight
\edgeheight=3pt
\newdimen\nodeskip
\nodeskip=.5em
\newdimen\nodesep
\nodesep=3pt
```

3.2 Tools and Environment

We need to prepare some infrastructure.

Remove spaces

The `\tree@removespace` macro serves as a "vacuum cleaner" for spaces which have been put prior to its statement.

```
\def\tree@removespace{%
 \let\next\empty
 \ifhmode
  \ifdim\lastskip>0pt
   \unskip\let\next\tree@removespace
  \fi
  \ifdim\lastkern>0pt
   \unkern\let\next\tree@removespace
  \fi
 \fi
 \next
```

```
}
```

Dimensions in Macros

\tree@set
\tree@value
\tree@reset

We have to deal with quite a number of dimensions when typesetting a tree. Since the number of dimension registers is restricted to 256, we provide a mechanism to store them in macros. Arbitrary text t can be assigned to variables like x, technically there will be a macro `\tree@@`x which expands to t.

The `\tree@set{#1}{#2}` macro sets the value of a variable `#1` to the value `#2`. If the variable has been used before, its value will be overriden, otherwise a new variable will be "allocated." Variable values are global and not affected by grouping.

```
\def\tree@set#1#2{%
 \expandafter\xdef\csname tree@@#1\endcsname{#2}%
}
```

The `\tree@value{#1}` macro expands to the value which is assigned to the variable `#1`. If nothing has been assigned to `#1`, an error will be raised.

```
\def\tree@value#1{%
 \expandafter\if\csname tree@@#1\endcsname\relax
  \PackageError
   {trees}%
   {Requesting value of undefined variable #1}%
   {Defining #1=0pt}%
  \tree@set{#1}{0pt}%
 \fi
 \csname tree@@#1\endcsname
}
```

The `\tree@reset{#1}` macro finally allows to release a variable, i.e., forget its content and regard it as undefined.

```
\def\tree@reset#1{%
 \expandafter\let\csname tree@@#1\endcsname\undefined
}
```

Skyline macros

We now turn to the definition of skyline macros. Skylines will be stored in lists; the dealing with lists implements the ideas outlined in the TEXBook p. 378f. We will assume that a skyline list never contains something else but dimensions.

We need two token registers for temp use.

```
\toksdef\tree@toksa=0
\toksdef\tree@toksb=2
```

\tree@newskyline
\tree@emptyskyline

A constructor for new skyline "objects." The `\tree@newskyline{#1}` macro creates a new skyline and assigns it to `#1` which is assumed to be a control sequence. Actually there is no need to use this macro directly, because an empty skyline is an empty list, and the empty list is unique. We therefore once and for all also provide an empty list in the control sequence `\tree@emptyskyline`.

```
\def\tree@newskyline#1{\def#1{\\}}
\tree@newskyline\tree@emptyskyline
```

What follows are macros to maintain skylines.

\tree@append

The first thing is to append an item to an existing skyline. `\tree@append` takes a control sequence `#1` containing a skyline and a dimension `#2` in the form `\the\counter` or `5pt`. Note that appending a negative length $l < 0$ to a skyline is defined to append 0 to the skyline shifted by $-l$.

```
\def\tree@append#1#2{%
 \expandafter\ifdim#2<0pt%
  \tree@shift#1{-#2}%
  \tree@append#1{0pt}%
 \else
  \edef\@tempa{\noexpand\\#2}%
  \tree@toksa=\expandafter{\@tempa}%
  \tree@toksb=\expandafter{#1}%
  \edef#1{\the\tree@toksa\the\tree@toksb}%
 \fi
}
```

\tree@appendRev

This version of append is needed by the cover macro to be defined below. It behaves appends the new item `#2` to the end of the skyline `#1` and doesn't care if `#2` is negative.

```
\def\tree@appendRev#1#2{%
 \tree@toksa=\expandafter{#2\\}%
 \tree@toksb=\expandafter{#1}%
 \edef#1{\the\tree@toksb\the\tree@toksa}%
}
```

\tree@shift

Given a skyline `#1` and a dimension `#2`, the `\tree@shift` macro shifts all items of `#1` by `#2`.

```
\def\tree@shift#1#2{%
 \let\temp@a\\
```

```
 \let\temp@b\tree@emptyskyline
 \def\\##1\\{%
  \ifx @##1
    \let\next\empty
  \else
    \@tempdima=##1\relax
    \advance\@tempdima by#2\relax
    \edef\temp@c{\the\@tempdima}%
    \tree@toksa=\expandafter{\temp@c\\}%
    \tree@toksb=\expandafter{\temp@b}%
    \edef\temp@b{\the\tree@toksb\the\tree@toksa}%
    \let\next\\
  \fi
  \next
 }%
 #1@\\%
 \let#1\temp@b
 \let\\\temp@a
}
```

\tree@separate

The `\tree@separate{#1}{#2}` macro separates pops the first item of a skyline `#1` to a dimension register `#2`.

```
\def\tree@separate#1#2{%
 \ifx#1\tree@emptyskyline
  #2=5000pt%
 \else
  \expandafter\tree@separate@#1\@#1#2%
 \fi
}
\long\def\tree@separate@\\#1\\#2\@#3#4{%
 #4=#1\def#3{\\#2}%
}
```

\tree@mindist

The `\tree@mindist` macro computes the distance between two skylines `#1` and `#2`. This distance is defined to be the minimum over all sums of corresponding items. The return value of the macro will be put into the dimension register `\@tempdimc`.

First, some initialisation is needed. The actual recursion is carried out in the `\tree@mindist@` command.

```
\def\tree@mindist#1#2{%
 \let\@tempa#1%
 \let\@tempb#2%
 \@tempdimc=5000pt%
 \tree@mindist@
```

```
}
\def\tree@mindist@{%
 \let\next\empty
 \ifx\@tempa\tree@emptyskyline\else
  \ifx\@tempb\tree@emptyskyline\else
   \tree@separate\@tempa\@tempdima
   \tree@separate\@tempb\@tempdimb
   \advance\@tempdima\@tempdimb
   \ifdim\@tempdima<\@tempdimc
    \@tempdimc\@tempdima
   \fi
   \let\next\tree@mindist@
  \fi
 \fi
 \next
}
```

`\tree@cover` We now turn to the procedure of covering one skyline by another. Let `#1` and `#2` be two skylines, i.e., skyline control sequences. The macro will redefine `#1` in the following way. Let n be the length of `#1` and m be the length of `#2`. If $n \leq m$ then `#1` will simply be redefined to `#2`. Otherwise, the ith value of the resulting skyline will be equal to `#2`$_i$ if this value exists and to `#1`$_i$ otherwise. If it afterwards turns out that there are negative entries in the resulting skyline, then this skyline will be shifted by the least value which makes all entries nonnegative.

```
\newif\iftree@done
\def\tree@cover#1#2{%
```

Let `\@tempa`, `\@tempb` be working copies of `#1` and `#2`, respectively. The resulting skyline will be put into `\@tempc`.

```
 \let\@tempa#1
 \let\@tempb#2
 \let\@tempc\tree@emptyskyline
```

The `\@tempdimc` dimension is initialized to something "very large." It will during the recursion contain the least value of the resulting skyline, at the end its minimum. Its sign will indicate if a shift is necessary.

```
 \@tempdimc=5000pt
```

As long as neither `\@tempa` nor `\@tempb` are empty…

```
 \tree@donefalse
 \loop
  \ifx\@tempa\tree@emptyskyline
   \ifx\@tempb\tree@emptyskyline
```

Base case: Set the “done” flag to true to exit the loop and assign the computed skyline to `#1`.

```
   \let#1\@tempc
   \tree@donetrue
  \else
```

Append the first item of `\@tempb` to the back of `\@tempc`, update the minimum, if necessary.

```
   \tree@separate\@tempb\@tempdimb
   \ifdim\@tempdimb<\@tempdimc
    \@tempdimc=\@tempdimb
   \fi
   \tree@appendRev\@tempc{\the\@tempdimb}%
  \fi
 \else
  \ifx\@tempb\tree@emptyskyline
```

Now, `\@tempa` has more entries but `\@tempb` is already empty. We have to append the first item of `\@tempa` to the back of `\@tempc` and to eventually update the minimum.

```
   \tree@separate\@tempa\@tempdima
   \ifdim\@tempdima<\@tempdimc
    \@tempdimc=\@tempdima
   \fi
   \tree@appendRev\@tempc{\the\@tempdima}%
  \else
```

At this point, both `\@tempa` and `\@tempb` have more elements. We have to append the first item of `\@tempb` to the back of `\@tempc` and again to update the minimum if necessary.

```
   \tree@separate\@tempb\@tempdimb
   \ifdim\@tempdimb<\@tempdimc
    \@tempdimc=\@tempdimb
   \fi
   \tree@appendRev\@tempc{\the\@tempdimb}%
```

The first item of `\@tempa` has to be skipped to keep synchronized.

```
   \tree@separate\@tempa\@tempdima
  \fi
 \fi
\iftree@done\else\repeat
```

This completes the loop over the elements. Shift the result if its smallest entry is negative.

```
\ifdim\@tempdimc<0pt
 \tree@shift#1{-\@tempdimc}%
```

```
 \fi
}
```

3.3 Public customisation

\node \leaf \baselevels \centerlevels \ceillevels \floorlevels

We define the macros for customisation. There is not much to say about them.

```
\def\node#1{#1}%
\def\leaf#1{\tree{#1}{}}%
\def\baselevels{%
 \let\tree@setrootbox@pre\tree@baserootbox@pre
 \let\tree@setrootbox@post\tree@baserootbox@post
}
\def\centerlevels{%
 \let\tree@setrootbox@pre\tree@centerrootbox@pre
 \let\tree@setrootbox@post\tree@centerrootbox@post
}
\def\ceillevels{%
 \let\tree@setrootbox@pre\tree@ceilrootbox@pre
 \let\tree@setrootbox@post\tree@ceilrootbox@post
}
\def\floorlevels{%
 \let\tree@setrootbox@pre\tree@floorrootbox@pre
 \let\tree@setrootbox@post\tree@floorrootbox@post
}
```

3.4 Tree Typesetting

\tree

It is time to get to the heart of the package.

```
\def\tree#1#2{{%
```

Initialize dimensions and counters

```
 \baselineskip=0pt
 \global\tree@depth=-1
 \global\tree@currentlevel=0
```

We walk through the tree twice. For the first time, we redefine `\tree` to `\tree@measurenodes` (defined below) which measures the level heights and counts the depth of the tree. This measurement is done within an `\hbox` to avoid that something "falls down" and is typeset. The `\hbox` should be empty afterwards, anyway it will be thrown away.

```
 \let\tree\tree@measurenodes
 \setbox\tree@rootbox=\hbox{\tree{#1}{#2}}%
```

Now, typeset the tree. This is done in the second walk through the tree, this time carried out by a `\@tree` macro to be defined below. We have to do some initialisations and to take special care about the different treatment of the root. (Note that this code is only executed on the uppermost level.)

```
 \global\tree@currentlevel=0
 \tree@newskyline\tree@rightskyline
 \tree@newskyline\tree@leftskyline
 \let\tree\@tree
 \let\ifroot\iftrue
 \tree{#1}{#2}%
```

Finally, we release all the allocated dimensions and are done.

```
 \tree@currentlevel=0
 \loop
  \tree@reset{\the\tree@currentlevel}%
  \advance\tree@currentlevel by 1
 \ifnum\tree@currentlevel<\tree@depth\repeat
}} % matches \def\tree
```

`\tree@measurenodes` The purpose of this macro is to set up the data that is needed to typeset the tree, i.e., the number of levels (the depth) and for each depths the height and depth, determined by the highest and deepest node in it. It is assumed that the `\tree@currentlevel` counter is set to `0` the `\tree@depth` is `-1` and the variables `lht0`,..., `lht`$d-1$, `ldp0`,...,`ldp`$d-1$ are undefined or defined to `0pt`. Afterwards, `lht`i will contain the height of level i where levels are counted from top to bottom, and where the root has level 0. The tree depth will reside in the `\tree@depth` counter register, the value will be defined to it globally.

```
\def\tree@measurenodes#1#2{%
```

Typeset the root of *this* tree into a box register.

```
 \setbox\tree@rootbox=\hbox{\node{#1}}%
```

If the current level is more than the tree depth, we have encountered a level on which we have not yet been. We will increase the depth counter accordingly and set the height and depth values of this new level to the respective dimensions of the `#1` box.

```
 \ifnum\tree@currentlevel>\tree@depth
  \global\tree@depth=\tree@currentlevel
  \tree@set{lht\the\tree@currentlevel}{\the\ht\tree@rootbox}%
  \tree@set{ldp\the\tree@currentlevel}{\the\dp\tree@rootbox}%
 \else
```

Otherwise we have been here and we only have to update the level height and level depth if the respective values for `#1` is more than it was for the nodes of this level which we have already seen.

```
 \expandafter\ifdim\tree@value{lht\the\tree@currentlevel}
                   <\ht\tree@rootbox
  \tree@set{lht\the\tree@currentlevel}{\the\ht\tree@rootbox}%
 \fi
 \expandafter\ifdim\tree@value{ldp\the\tree@currentlevel}
                   <\dp\tree@rootbox
  \tree@set{ldp\the\tree@currentlevel}{\the\dp\tree@rootbox}%
 \fi
\fi
```

Recursion step: We increase the current level by 1 and call call recursively the measurement of the subtrees if there are some. We increase the currentlevel aferwards when we come back.

```
\advance\tree@currentlevel by 1
#2
\advance\tree@currentlevel by -1
} % matches \def\tree@measurenodes
```

The alignment of the elements of a level is goverend by macros which insert the vertical pieces of the edges below and above the nodes. There are four `\tree@..rootbox@pre` commands for the above parts and four `\tree@..rootbox@post` commands, namely one where .. is one of `base`, `center`, `ceil`, or `floor`. The tree typesetting macro uses none of them directly, but it uses `\tree@setrootbox@pre` and `\tree@setrootbox@post` which are by default defined to be the alignment at the common baseline. The user can change the alignment policy by use of the commands `\baselevels`, `\centerlevels`, `\floorlevels`, `\ceillevels` which simply redefine `\tree@setrootbox@pre` and `\tree@setrootbox@post` accordingly.

The environment which calls the stretching macros establishes some information in registers. When a `\tree@..rootbox@..` command is called, it can be assumed that `\@tempdima` contains the height of the current level and `\@tempdimb` contains the depth of the current level. Further it can be assumed that no assignments will be done between the call of `..@pre` and `..@post`, also the `\@tempdima` and `\@tempdimb` values remain changed in `..@post` if `..@pre` changes them.

First the base line oriented alignment

```
\def\tree@baserootbox@pre{%
\advance\@tempdima by-\ht\tree@rootbox\relax
\ifdim\@tempdima>0pt
```

```
  \nointerlineskip
  \@tempdimc\@tempdima
  \hbox to\tree@width{\hss\rule\edgewidth\@tempdimc\hss}%
 \fi
 \@tempdima\dp\tree@rootbox\relax
}
\def\tree@baserootbox@post{%
 \advance\@tempdimb by-\@tempdima
 \ifdim\@tempdimb>0pt
  \nointerlineskip
  \@tempdimc\@tempdimb
  \hbox to\tree@width{\hss\rule\edgewidth\@tempdimc\hss}%
 \fi
}
```

Now the floor alignment.

```
\def\tree@floorrootbox@pre{%
 \advance\@tempdima\@tempdimb
 \advance\@tempdima-\ht\tree@rootbox
 \advance\@tempdima-\dp\tree@rootbox
 \ifdim\@tempdima>0pt
  \nointerlineskip
  \@tempdimc\@tempdima
  \hbox to\tree@width{\hss\rule\edgewidth\@tempdimc\hss}%
 \fi
}
\def\tree@floorrootbox@post{}
```

Now the ceiling alignment.

```
\def\tree@ceilrootbox@pre{%
 \advance\@tempdima\@tempdimb
 \advance\@tempdima-\ht\tree@rootbox
 \advance\@tempdima-\dp\tree@rootbox
}
\def\tree@ceilrootbox@post{%
 \ifdim\@tempdima>0pt
  \nointerlineskip
  \@tempdimc\@tempdima
  \hbox to\tree@width{\hss\rule\edgewidth\@tempdimc\hss}%
 \fi
}
```

And finally the centered alignment.

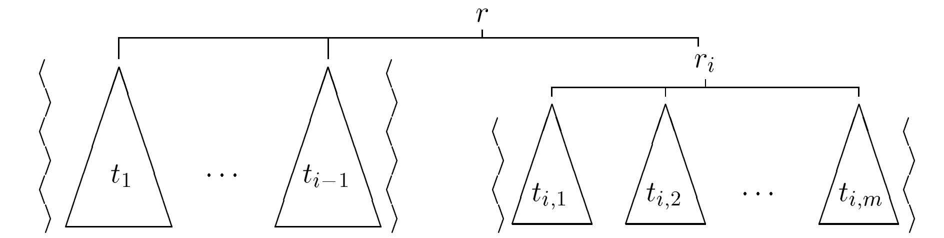
```
\def\tree@centerrootbox@pre{%
 \advance\@tempdima\@tempdimb
 \advance\@tempdima-\ht\tree@rootbox
```

```
  \advance\@tempdima-\dp\tree@rootbox
  \@tempdimc.5\@tempdima
  \ifdim\@tempdimc>0pt
    \nointerlineskip
    \hbox to\tree@width{\hss\rule\edgewidth\@tempdimc\hss}%
  \fi
}
\def\tree@centerrootbox@post{%
  \ifdim\@tempdimc>0pt
    \nointerlineskip
    \hbox to\tree@width{\hss\rule\edgewidth\@tempdimc\hss}%
  \fi
}
```

We define the default to base line oriented alignment.

```
\let\tree@setrootbox@pre\tree@baserootbox@pre
\let\tree@setrootbox@post\tree@baserootbox@post
```

Now we will turn to the definition of `\@tree` which does the actual typesetting after the tree cells have been measured. Skylines are computed on the fly, and the horizontal shrinking is done accordingly. In the general situation, we have already arranged subtrees $t_1, \ldots, t_{i-1}$ of a list of subtrees and got by a recursive call a box containing the subtrees $t_{i,1}, \ldots, t_{i,m}$ of the subtree t_i. When computing the amount of horizontal space which might be removed the situation of the variables is according to the following drawing.



In a first step, the root of t_i is incorporated into `\tree@leftskyline` and `\tree@rightskyline`. This is nothing more than appending the appropriate length which depends on the root width and the subtree array width to each of them.

Next, the distance between `\tree@oldrightskyline`, i.e., the left shape of the already put $t_1, \ldots, t_{i+1}$, and `\tree@leftskyline`, i.e., the shape of the just obtained subtree t_i, yields the minimum distance between corresponding nodes of $t_1, \ldots, t_{i-1}$ and t_i. If this distance is more than desired, i.e., more than `\nodeskip`, t_i can be moved to the left by an according value.

The last step is to update the two "old" skylines `\tree@oldleftskyline` and `\tree@oldrightskyline` such that they are fit to incorporate the next subtree t_{i+1}.

```
\def\@tree#1#2{%
```

The whole thing is intended to work recursively, so we have to save global variables in private ones in order to be able to restore them afterwards.

```
  \tree@oldnosubtrees=\tree@nosubtrees
  \tree@oldfirstnode=\tree@firstnode
  \tree@oldlastnode=\tree@lastnode
  \tree@oldarraywidth=\tree@arraywidth
  \tree@oldleftoverhang=\tree@leftoverhang
  \tree@oldrightoverhang=\tree@rightoverhang
  \let\tree@oldleftskyline\tree@leftskyline
  \let\tree@oldrightskyline\tree@rightskyline
```

We typeset the node and put it into the root box register.

```
  \setbox\tree@rootbox=\hbox{\node{#1}}%
```

After the appropriate initialisation, we typeset the subtrees recursively and put the result into the `\tree@subtreebox` box register.

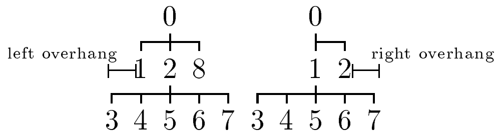
```
  \global\tree@nosubtrees=0%
  \global\tree@firstnode=-1pt%
  \global\tree@arraywidth=0pt%
  \global\tree@leftoverhang=0pt%
  \global\tree@rightoverhang=0pt%
  \let\tree@leftskyline\tree@emptyskyline
  \let\tree@rightskyline\tree@emptyskyline
```

Here we are typesetting the subtree array. First we go some distance to the left to make up for the corresponding going to the right of the first subtree in the list. The `\tree@arraywidth` will always contain the "current" length of the subtree `\hbox`. The level is increased by one, we are not any more in the root, and call the subtree generation recursively. The recursive call is hidden in `#2` which in general consists of `\tree` calls. At the end, superfluous horizontal space of the list will be removed.

```
  \setbox\tree@subtreebox=\hbox{{%
      \global\tree@arraywidth=-\nodeskip
      \kern-\nodeskip
      \advance\tree@currentlevel by 1
      \let\ifroot\iffalse
      #2\tree@removespace}}%
```

In the recursion, the left and the right skylines of the subtree have been constructed. Also, there are two lengths `\tree@leftoverhang` and

\tree@rightoverhang which denote how much the subtree box overhangs to the left/right side, i.e., if parts of the subtree array range outside the \hbox, how far do they do this to the left and to the right. Left and right overhang are illustrated in the following examples for the subtrees with root 2.



According to this information, we have to fill the respective amount of space into the box. After that the subtrees box will have the correct dimensions.

```
\setbox\tree@subtreebox=\hbox{%
  \null\kern\tree@leftoverhang
  \unhbox\tree@subtreebox
  }%
```

We need to know the width of the first and the last subtree in the procuded array for determining the start and end point of the horizontal part of the edge. This information is provided after the recursive descend in the registers \tree@firstnode and \tree@lastnode. In the case of overhangs, these lengths have to be adjusted according to the overhanging parts.

```
\advance\tree@firstnode2\tree@leftoverhang
\advance\tree@lastnode2\tree@rightoverhang
```

The old overhang values are restored for the recursive ascend.

```
\global\tree@leftoverhang=\tree@oldleftoverhang
\global\tree@rightoverhang=\tree@oldrightoverhang
```

Determine the overall width of this tree which is the maximum of the node's and the subtree's widths.

```
\tree@width=\wd\tree@rootbox
\ifdim\tree@width<\wd\tree@subtreebox
  \tree@width=\wd\tree@subtreebox
\fi
```

\tree@leftskyline and \tree@rightskyline are the left and right skylines of the subtreearray just produced. Extend them by the value for this root which is

$$\frac{1}{2}(\texttt{\textbackslash wd\textbackslash tree@subtreebox} - \texttt{\textbackslash wd\textbackslash tree@rootbox})$$

```
\@tempdimc=.5\wd\tree@subtreebox
```

```
\advance\@tempdimc-.5\wd\tree@rootbox
\tree@append\tree@leftskyline{\the\@tempdimc}%
\tree@append\tree@rightskyline{\the\@tempdimc}%
```

Insert space between the previous subtree and *this* subtree. The space to insert now is the desired amount of space minus the distance between the two skylines and minus the space which might be already there. The `\null` box prevents the kerning from being removed.

```
\tree@removespace
\kern\nodeskip
\null
```

The left and right skyline need to be updated in order to take deeper subtrees of the array into account as well. For some reason, `\tree@shift` introduces space which we have to remove afterwards: insert `\null` not to remove the intended space, and a `\tree@removespace` behind all the space introducing commands. . .

```
\tree@mindist\tree@oldrightskyline\tree@leftskyline
\tree@removespace
\ifdim\@tempdimc<5000pt
 \kern-\@tempdimc
 \advance\tree@oldarraywidth-\@tempdimc
\else
 \@tempdimc=0pt
\fi
```

If the old array width is negative, borrow some space from `\leftoverhang`

```
\ifdim\tree@oldarraywidth<-\nodeskip
 \global\advance\tree@leftoverhang-\tree@oldarraywidth
 \global\advance\tree@leftoverhang-\nodeskip
 \tree@oldarraywidth=-\nodeskip
\fi
```

If the move to the left is more than the subtree is wide, add their difference to `\rightoverhang`.

```
\global\tree@rightoverhang0pt
\ifdim\@tempdimc>\tree@width
 \global\tree@rightoverhang\@tempdimc
 \global\advance\tree@rightoverhang-\tree@width
 \global\advance\tree@rightoverhang-\nodeskip
\else
\fi
\null
\@tempdimb=-\@tempdimc
\advance\@tempdimb\nodeskip
```

```
\advance\@tempdimb\tree@width
\tree@shift\tree@oldrightskyline\@tempdimb
\tree@cover\tree@oldrightskyline\tree@rightskyline
\let\tree@rightskyline\tree@oldrightskyline
```

same with the left skyline

```
\tree@shift\tree@leftskyline\tree@oldarraywidth
\tree@cover\tree@leftskyline\tree@oldleftskyline
\let\tree@oldleftskyline\tree@leftskyline
\tree@removespace
```

Typeset node, subtrees and the edges between them. distinguish between leaf and no-leaf.

```
\ifdim\wd\tree@subtreebox>0pt%
  \vtop{%
```

Typeset the vertical edge above the node, if the node is not the root of the entire tree.

```
    \ifroot\else
      \nointerlineskip
      \hbox to\tree@width{\hss\rule\edgewidth\edgeheight\hss}%
    \fi
```

Typeset the node itself, surrounded by `..@pre` and `..@post` commands to align them. Also put space where needed.

```
    \expandafter\@tempdima
                \tree@value{lht\the\tree@currentlevel}%
    \expandafter\@tempdimb
                \tree@value{ldp\the\tree@currentlevel}%
    \tree@setrootbox@pre
    \ifroot\else
     \kern\nodesep
    \fi
    \nointerlineskip
    \hbox to\tree@width{\hss\fbox{\unhbox\tree@rootbox}\hss}%
    \nointerlineskip
    \kern\nodesep
    \tree@setrootbox@post
```

If there is more than one subtree, typeset the horizontal part of the edge, otherwise, insert some vertical line which is by `\edgewidth` higher to keep consistence.

```
    \ifnum\tree@nosubtrees>1
      \nointerlineskip
      \hbox to\tree@width{\hss\rule\edgewidth\edgeheight\hss}%
      \@tempdimc=\wd\tree@subtreebox
```

```
        \advance\@tempdimc-.5\tree@firstnode
        \advance\@tempdimc-.5\tree@lastnode
        \advance\@tempdimc\edgewidth
        \nointerlineskip
        \hbox{%
          \kern.5\tree@width
          \kern-.5\wd\tree@subtreebox
          \kern.5\tree@firstnode
          \kern-.5\edgewidth
          \rule\@tempdimc\edgewidth}%
      \else
        \@tempdimc=\edgeheight
        \advance\@tempdimc\edgewidth
        \nointerlineskip
        \hbox to\tree@width{\hss\rule\edgewidth\@tempdimc\hss}%
      \fi
```

Now put the subtrees.

```
      \nointerlineskip
      \hbox to\tree@width{\hss\fbox{\unhbox\tree@subtreebox}\hss}%
    }%
  \else
```

Now we are in the case of a leaf. This is similar and easier.

```
    \vtop{%
      \nointerlineskip
      \hbox to\tree@width{\hss\rule\edgewidth\edgeheight\hss}%
      \expandafter\@tempdima
                  \tree@value{lht\the\tree@currentlevel}%
      \expandafter\@tempdimb
                  \tree@value{ldp\the\tree@currentlevel}%
      \tree@setrootbox@pre
      \ifroot\else
       \kern\nodesep
      \fi
      \nointerlineskip
      \hbox to\tree@width{\hss\fbox{\unhbox\tree@rootbox}\hss}%
```

Put some space below the tree

```
      \nointerlineskip
      \hbox to\tree@width{\null\mathstrut\hfill\null}%
    }%
  \fi
```

Essentially, we are done. It remains to restore variables, and to put some horizontal space.

```
  \global\tree@nosubtrees=\tree@oldnosubtrees
  \global\advance\tree@nosubtrees by 1
  \global\tree@firstnode=\tree@oldfirstnode
  \global\tree@arraywidth\tree@oldarraywidth
  \global\advance\tree@arraywidth\nodeskip
  \global\advance\tree@arraywidth\tree@width
  \global\let\tree@leftskyline\tree@oldleftskyline
  \global\let\tree@rightskyline\tree@oldrightskyline
  \ifdim\tree@firstnode<0pt
   \global\tree@firstnode\tree@width
  \fi
  \global\tree@lastnode\tree@width
  \tree@removespace
  \kern\tree@rightoverhang
  \null
} % matches \def\@tree
```

That's it.