The Wayback Machine - https://web.archive.org/web/20070824055931/http://archive.eso.org:8...

# The Tree Widget

**Author:** ***Allan Brighton, abrighto@eso.org***

***You can get the latest version (tree-8.0.3) here.***

Much of the information that computers deal with is hierarchical, or can be viewed as a hierarchy. A UNIX file system is one of the most common examples. The Tk widget hierarchy is another. One of the best ways to display an overview of this kind of data is in the form of a tree.

This chapter describes a widget for displaying trees in Tk. The tree widget really has two interfaces. The first is to the basic Tk widget, which is implemented in C++. This is the most flexible and is the one we will look at first. The second interface is implemented as an [incr Tcl] class built on top of the tree widget. The [incr Tcl] class offers a simpler, but less flexible interface that displays trees where each node is made up of a label and an optional bitmap. We will come to that later on.

The tree widget differs from most other Tk widgets in that it doesn't display its own window and does not need to be *packed* in the window hierarchy. What the tree widget does is manage items that you create in a Tk canvas. You create the tree as a *child* of a canvas widget and then tell it which canvas items make up each tree node and line. The tree widget calculates the layout and repositions the canvas items to make the tree. Using canvas items for the tree nodes and lines has some great advantages. The tree nodes can be made up of any combination of canvas items, including text, bitmaps, images and other graphics. You design the lines connecting the tree nodes, so you can make them in any color, style or thickness you like.

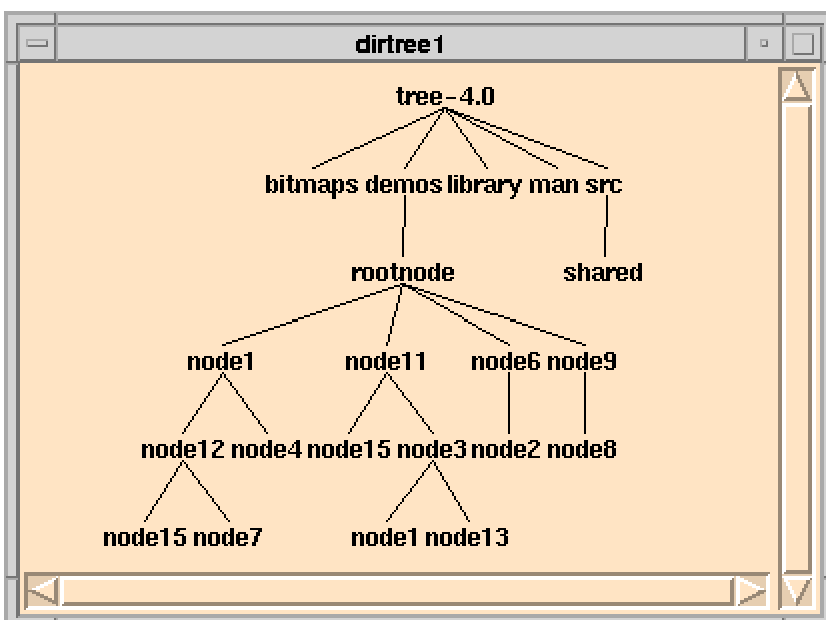
## A Simple Tree

Before we get into any more detail, let's look at an example of a simple tree application: a directory tree. The program below uses the Unix ls(1) command to recursively build up a tree of directories. Its only a simple demonstration, so there are no menus and no bindings yet. (Note that all of the examples shown here can be found in the `demos` directory of the tree widget extension.)

```
# create a canvas with horizontal and vertical scrollbars in the
# given frame with the given name

proc MakeCanvas {frame canvas} {
    set vscroll [scrollbar $frame.vscroll -command "$canvas yview"]
    set hscroll [scrollbar $frame.hscroll -orient horiz -command "$canvas xview"]
    set canvas [canvas $canvas -xscroll "$hscroll set" -yscroll "$vscroll set"]
    pack $vscroll -side right -fill y
    pack $hscroll -side bottom -fill x
    pack $canvas -fill both -expand 1
    bind $canvas  "$canvas scan mark %x %y"
    bind $canvas  "$canvas scan dragto %x %y"
    return $canvas
}
```

```
# add the directories under $dir to the tree (recursively)

proc ListDirsRec {canvas tree dir} {
    foreach i [exec ls $dir] {
        if {[file isdir $dir/$i]} {
            AddDir $canvas $tree $dir $dir/$i $i
            ListDirsRec $canvas $tree $dir/$i
        }
    }
}

# add the given directory to the tree
#
# Args:
#  canvas  - tree's canvas
#  tree    - the tree
#  parent  - pathname of parent dir
#  dir     - pathname of new dir being added
#  text    - text for tree node label (last component of dir)

proc AddDir {canvas tree parent dir text} {
    $canvas create text 0 0 -text $text -tags $dir
    set line [$canvas create line 0 0 0 0 -tag "line"]
    $tree addlink $parent $dir $line
}


# main

wm geometry . 400x275
set canvas [MakeCanvas . .c]
set tree [tree $canvas.t -layout vertical]

cd ..
set dir [pwd]
AddDir $canvas $tree {} $dir [file tail $dir]
ListDirsRec $canvas $tree $dir
```

The figure below shows the window produced by the above code when run from the *demos* directory in the tree-4.0 distribution (the *rootnode* directory is a dummy directory created for test purposes).




## Creating the Tree

The first thing you need to do to use the tree widget is to create a canvas widget. The `MakeCanvas` procedure above creates a canvas with two scrollbars and the usual Tk bindings, so you can scroll by dragging the middle mouse button. As we will see later on, the [incr Tcl] tree interface takes care of this part for us.

The tree widget itself is created toward the end of the example with the following command:

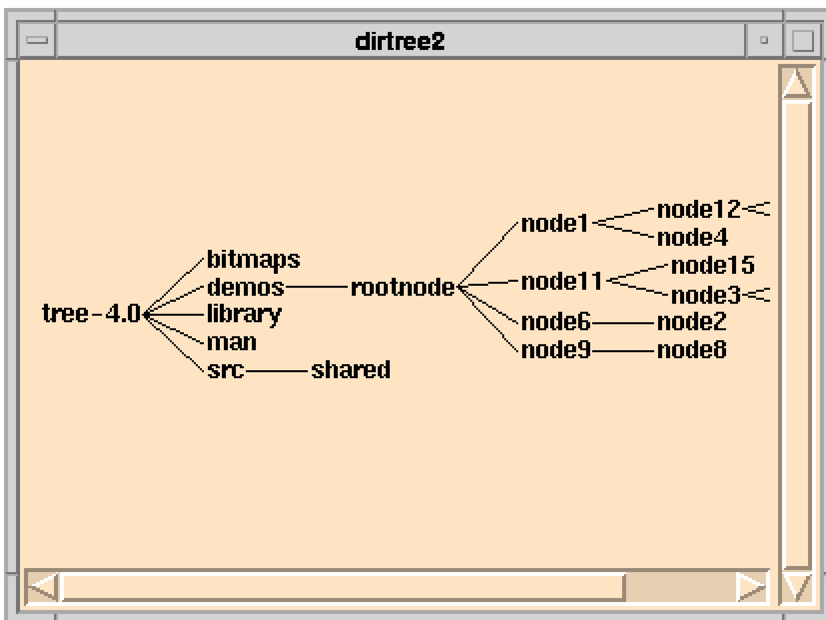
```
set tree [tree $canvas.t -layout vertical]
```

The tree widget, like all Tk widgets, returns its path name, in this case `.c.t`, when created. For clarity, I prefer to use variable names such as `$canvas` or `$tree` to refer to widgets rather than the path names. In this case the path name is quite short, but that is not usually the case.

The tree widget must be created as a child of the canvas widget it will manage (in fact the tree widget derives the name of the canvas widget from its own path name). The default layout of the tree is `horizontal` or left to right. In this example the layout was set to `vertical`. You can change the layout at any time by using the `configure` subcommand:

```
$tree configure -layout horizontal
```

This produces a tree as shown below.




## Adding Nodes to the Tree

Tree nodes are made up of one or mode canvas items, such as text, bitmap, image or other graphics. Canvas items in Tk are referred to by their tags or Ids. Each canvas item has an id and an optional list of tags, where a tag is any (non-integer) string. You can refer to a group of canvas items by assigning them all a common tag. This feature is used by the tree widget to refer to the tree nodes as well.

When the tree widget is created, a single, invisible root node is also created. This node is given an *empty* tag and can be referred to as `""` or `{}`. To start adding nodes to the tree, you use the tree subcommand `addlink`:

```
$canvas create text 0 0 -text $text -tags $dir
set line [$canvas create line 0 0 0 0 -tag "line"]
$tree addlink $parent $dir $line
```

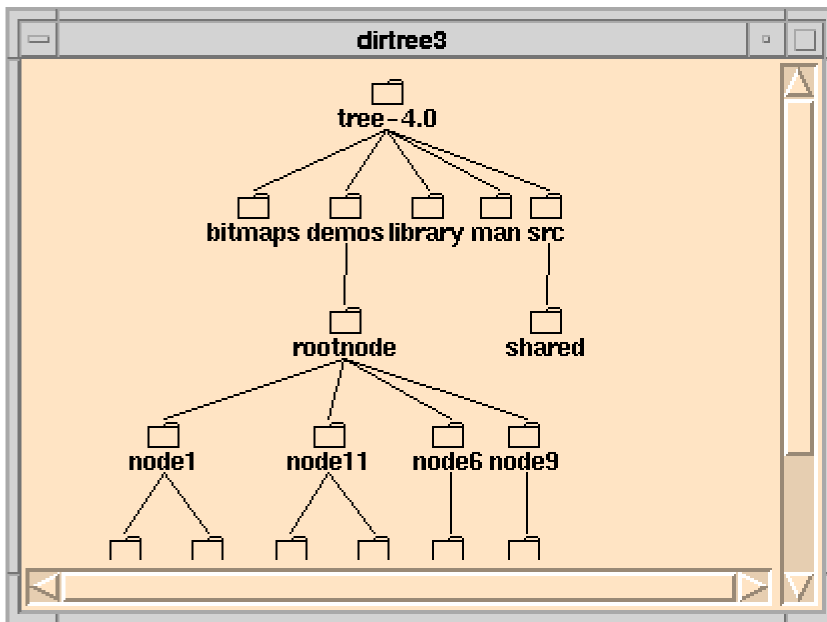
The `addlink` subcommand takes three arguments, which are the canvas tags or ids of the *parent* node, the *node* you are adding to the tree and the *line* connecting them. If the new node should be the root of the tree, then *parent* is specified as the empty tag `{}`. In the example above, the tree node consists only of a canvas text item displaying the last component of the directory name `$text`. We use the full path name `$dir` as the canvas tag, since it uniquely describes the node. We could also have used some other unique value, such as the the directory's i-node. The label is created at the arbitrary position (0, 0). The tree widget will move it to its final position, so we can place it anywhere we like. Likewise the line connecting the parent and child nodes is created as a canvas line with length 0 at position (0, 0), since the tree widget will position the line as needed. The advantage of creating the line yourself rather than having the tree widget do it for you is that you can specify the color, thickness and other properties of the line when you create it. The line does not need to (and should not) have the same tag as the node. It is enough to specify the line's id, as returned from the canvas `create` command. In fact, in the example above, we could also have specified the canvas id for the node's label rather than using the `$dir` tag, but that would only work as long as the node is made up of a single canvas item.

## More Complex Nodes

To illustrate the point, let's modify the example above to add a bitmap image to each node. To do that we modify the `AddDir` procedure from the example above and add a new procedure `LayoutNode` to arrange the items in the node:

```
# add the given directory to the tree
#
# Args:
#  canvas  - tree's canvas
#  tree    - the tree
#  parent  - pathname of parent dir
#  dir     - pathname of new dir being added
#  text    - text for tree node label (last component of dir)

proc AddDir {canvas tree parent dir text} {
    set bitmap @bitmaps/dir.xbm
    $canvas create text 0 0 -text $text -tags [list $dir text $dir:text]
    $canvas create bitmap 0 0 -bitmap $bitmap -tags [list $dir bitmap $dir:bitmap]
    set line [$canvas create line 0 0 0 0 -tag "line"]
    LayoutNode $canvas $tree $dir
    $tree addlink $parent $dir $line
}


# layout the components of the given node depending on whether
# the tree is vertical or horizontal

proc LayoutNode {canvas tree dir} {
    set text $dir:text
    set bitmap $dir:bitmap

    if {[$tree cget -layout] == "horizontal"} {
        scan [$canvas bbox $text] "%d %d %d %d" x1 y1 x2 y2
        $canvas itemconfig $bitmap -anchor se
        $canvas coords $bitmap $x1 $y2
    } else {
        scan [$canvas bbox $bitmap] "%d %d %d %d" x1 y1 x2 y2
        $canvas itemconfig $text -anchor n
        $canvas coords $text [expr "$x1+($x2-$x1)/2"] $y2
    }
}
```

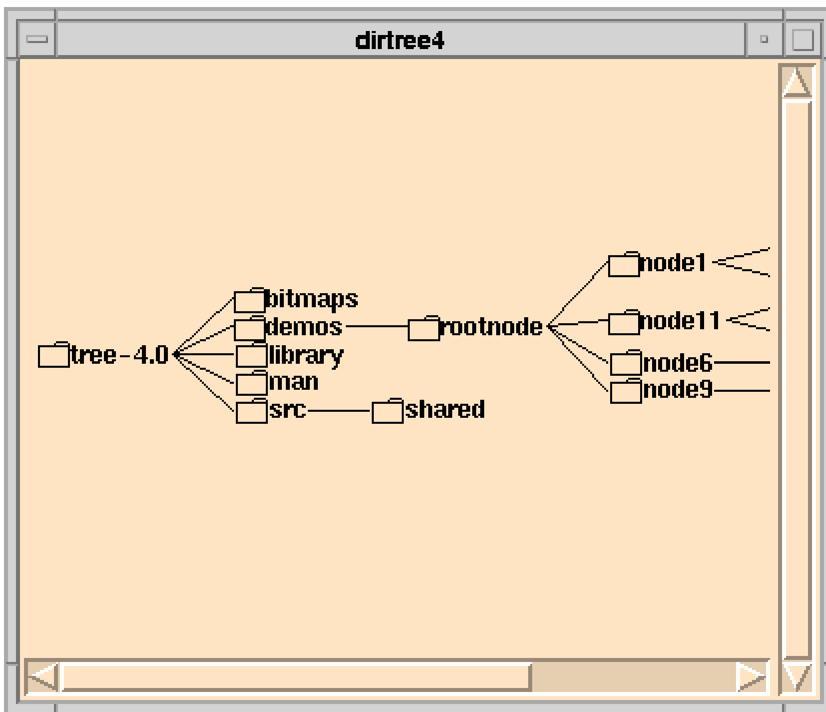
The figure below shows what the new tree looks like.




In the example above, each node's bitmap and label are initially created at the position (0, 0) and then the `LayoutNode` procedure is called to center the bitmap above the label for vertical trees or to the left of the label for horizontal trees. All of the components that make up a node are later moved as a unit to their final position in the tree, so the relative positions of the items stay the same. The `LayoutNode` procedure was written so that it could be called again for each node if the tree's layout is changed. For example, we could now toggle the tree's layout between vertical and horizontal with the following procedure:

```
# Toggle the layout of the tree between vertical and horizontal

proc ToggleLayout {canvas tree} {
    if {[$tree cget -layout] == "horizontal"} {
        $tree config -layout vertical
    } else {
        $tree config -layout horizontal
    }

    # change the layout of the nodes so that the bitmap is on top for
    # vertical trees and at left for horizontal trees
    foreach i [$canvas find withtag text] {
        set dir [lindex [$canvas gettags $i] 0]
        LayoutNode $canvas $tree $dir
        $tree nodeconfig $dir
    }
}
```

What this procedure does is loop through all of the tree nodes and change the layout of the items in each node depending on the layout of the tree (`horizontal` or `vertical`). The `nodeconfig` tree subcommand is then called for each node to tell the tree widget to recalculate the node's size and position. The figure below shows the same tree now with the *layout* set to `horizontal`.




## Using Canvas Tags to Reference Nodes

In the previous example the canvas items that make up a tree node were each given a list of three tags:

```
$canvas create text 0 0 -text $text -tags [list $dir text $dir:text]
$canvas create bitmap 0 0 -bitmap $bitmap -tags [list $dir bitmap $dir:bitmap]
```

The first tag `$dir` is shared by all of the items that make up a single node and is used to reference that node. the `ToggleLayout` procedure depends on this tag being the first in the list. The second tag identifies all of the node labels (`text`) or all of the bitmaps (`bitmap`) in the tree. The third tag is a combination of the first two and identifies the label or bitmap of a particular tree node. This is used in the `LayoutNode` procedure to change the position of the label and bitmap of each node.

The command:

```
$canvas find withtag text
```

returns a list of all the canvas items having the tag `text`. Since each of the tree nodes we created above also has a label with this tag, the return value is a list of all of the labels in the tree. Since we know that the `$dir` tag (the directory path name for the node) is the first in the list of tags, it can be retrieved with the command:

```
set dir [lindex [$canvas gettags $i] 0]
```

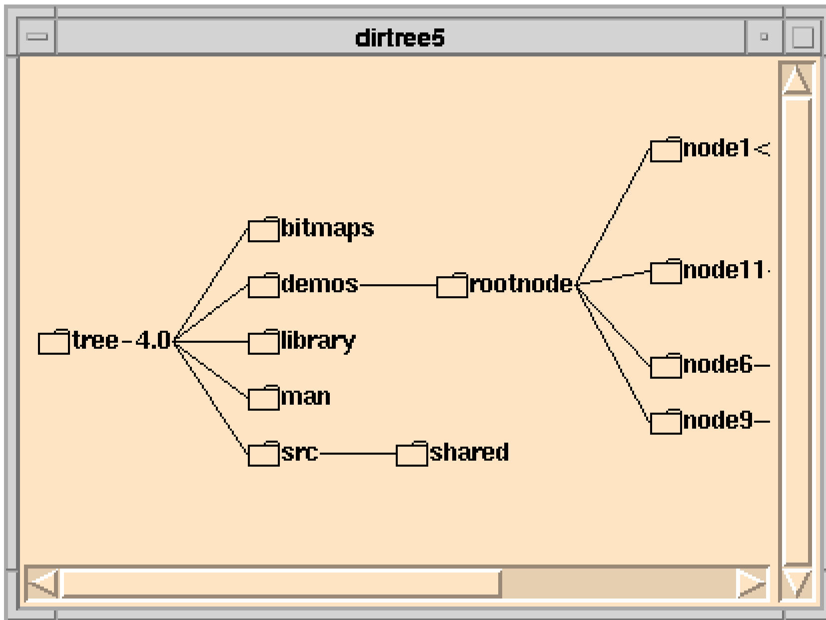
where `$i` is the id for the node's label or bitmap. We also use this feature later on to get the directory name associated with a node when the user clicks on it with the mouse.

## Changing the Distance between Sibling Nodes

Depending on the layout of the tree nodes, you may want to add some extra space between the nodes. For example, in the horizontal tree shown above, the layout looks a bit crowded. You can change this by specifying a border space around the individual nodes with the `-border:` option to `addlink`

```
proc AddDir {canvas tree parent dir text} {
    ...
    $tree addlink $parent $dir $line -border 2m
}
```

Although this option is set separately on each node, it usually looks best when all of the nodes have the same border size, as shown below. By default there is no border around a node:

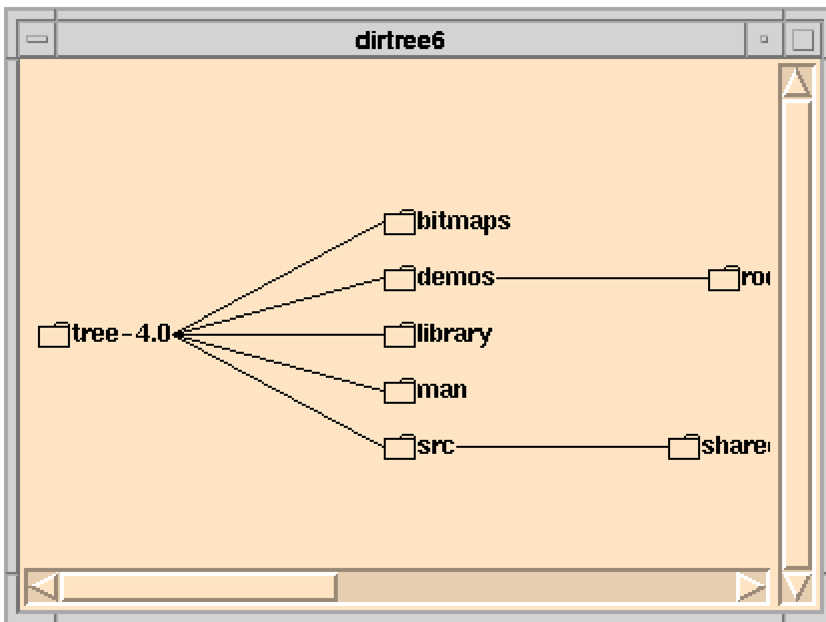



## Changing the Distance between Parent and Child Nodes

You can also change the distance between parent and child nodes. Unlike the `-border` option above, which is applied to each node, the `-parentdistance` option is set once for the entire tree, either as an option when creating the tree or through the `configure` widget subcommand. The default `parentdistance` is 30 pixels, however you can use any of the standard Tk units, such as `c` for centimeter as shown below:

```
proc AddDir {canvas tree parent dir text} {
    ...
    $tree addlink $parent $dir $line -border 2m
}
...
set tree [tree $canvas.t -layout horizontal -parentdistance 3c]
...
```

The resulting tree is shown below:




# Manipulating Trees

In the previous examples we have seen how to create and display a tree in a Tk canvas. Most applications that display trees also need to modify them dynamically and possibly allow the user to manipulate them directly. In this section we will discuss ways to examine and modify the tree structure and handle user input.

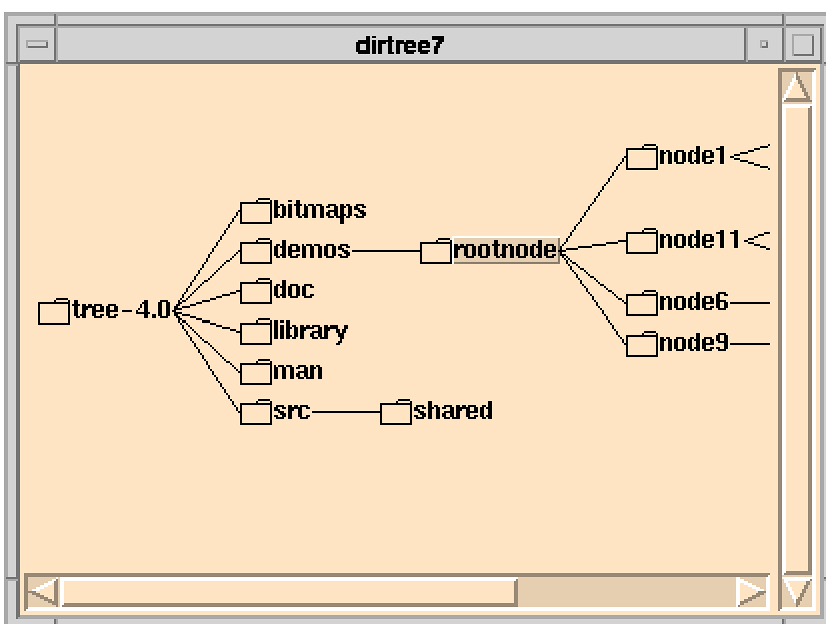
## Selecting Tree Nodes

Let's extend the example directory tree application to allow the user to select tree nodes with the mouse and perform operations on them. Since the tree nodes are made up of canvas items, we use the canvas widget's commands to handle the user input. First we add some bindings so that clicking with mouse button 1 on a tree node highlights the label or bitmap and prints out the pathname of the selected directory. The bindings and the procedures that implement them are listed below.

```
# set the bindings for the tree canvas

proc SetBindings {canvas tree} {
    # bind mouse button <1> to select the label or bitmap
    $canvas bind text <1> "focus %W; SelectNode $canvas"
    $canvas bind bitmap <1> "SelectBitmap $canvas"
}


# select the current node's label

proc SelectNode {canvas} {
    $canvas select from current 0
    $canvas select to current [string length [$canvas itemcget current -text]]
    DeSelectBitmap $canvas
    puts "selected label: [GetPath $canvas]"
}


# de-select all node labels

proc DeSelectNode {canvas} {
    $canvas select clear
}
```

```
# highlight the node's bitmap

proc SelectBitmap {canvas} {
    catch {focus {}}
    set path [lindex [$canvas gettags current] 0]
    DeSelectNode $canvas
    DeSelectBitmap $canvas
    $canvas itemconfig current -background [$canvas cget -selectbackground] ₩
        -tags "[$canvas gettags current] selected"
    puts "selected bitmap: [GetPath $canvas]"
}


# stop highlighting the node's bitmap

proc DeSelectBitmap {canvas} {
    $canvas itemconfig selected -background [$canvas cget -background]
    $canvas dtag selected
}



# return the pathname (dir) for the item currently selected (bitmap or text)

proc GetPath {canvas} {
    set id [$canvas select item]
    if {"$id" == ""} {
        return [lindex [$canvas gettags selected] 0]
    }
    return [lindex [$canvas gettags $id] 0]
}
```

The code to select a node's label is fairly straight forward. You use the canvas features to select the text item under the mouse, given by the special canvas tag `current`. Since there is no predefined way of selecting a bitmap, we simply set the bitmap's background color to the same color used to display selected text and give it the tag `selected`, which otherwise has no special meaning in this context. The `GetPath` routine checks for either a selected label or a selected bitmap and returns the first tag in the list of tags for the item, which is in this case the directory path name.




On my machine, clicking on the node highlighted above produced the following output:

```
selected label: /home/tcl/new/dist/tree-4.0/demos/rootnode
```

You could of course use other methods to select a node, such as drawing a border around it or changing the bitmap or image used to display it.

## Editing Trees Interactively

Suppose we want to allow the user to navigate the tree by double-clicking on the tree nodes. We could define the following behavior:

- A double-click on a node's bitmap either makes the node the new root of the tree, or if the node is already the root node, then the node's parent and siblings are added to the tree.
- A double-click on a node's label will add any subnodes (subdirectories here) to the tree, or if the subnodes are already in the tree, remove them.

To implement this behavior we add two new bindings that call some new Tcl procedures. These procedures make use of a number of `tree` subcommands that we haven't discussed yet, but I will explain them shortly.

```
$canvas bind text  "ToggleChildren $canvas $tree"
$canvas bind bitmap  "ToggleParent $canvas $tree"


# If the current selection is a leaf, add its subnodes, otherwise
# remove them

proc ToggleChildren {canvas tree} {
    set path [GetPath $canvas]
    if [$tree isleaf $path] {
        ListDirs $canvas $tree $path
        $tree draw
    } else {
        $tree prune $path
    }
}


# If the selected node is the root of the tree, add its parent and siblings
# to the tree, otherwise make the selected node the new root of the tree.

proc ToggleParent {canvas tree} {
    global dirtree
    set path [GetPath $canvas]
    if [$tree isroot $path] {
        set dir [file dirname $path]
        if {$dir != $path} {
            AddDir $canvas $tree "" $dir [dir_tail $dir]
            set tail [file tail $path]
            foreach i [exec ls $dir] {
                if {[file isdirectory $dir/$i]} {
                    if {$i == $tail} {
                        $tree movelink $path $dir
                    } else {
                        AddDir $canvas $tree $dir $dir/$i "$i"
                    }
                }
            }
            $tree draw
        }
    } else {
        $tree root $path
    }
}
```

```
# return the last component of the directory name
# (/ is a special case)

proc dir_tail {dir} {
    if {$dir == "/"} {return $dir}
   return [file tail $dir]
}
```

The `ToggleChildren` procedure adds child nodes to a node if none are present or removes them otherwise. First, we use the `GetPath` procedure defined earlier to get the tag for the selected node, which is, in our case, the directory path name used to uniquely identify the node.

To determine if the node has any subnodes, we use the `isleaf` tree subcommand. Given the canvas tag or id of a tree node, `isleaf` will return 1 if the node is a leaf node and 0 otherwise. In this case, if the node is a leaf node, we add its subnodes by calling `ListDirs` to add the directories under the selected directory.

The call to `$tree draw` causes the tree to be drawn on the canvas. This call is not actually necessary, since the tree would be drawn anyway, but it makes sure that the nodes (canvas items) are in the right places before they become visible.

If the node is not a leaf, then it has subnodes and we want to remove them from the tree. This is exactly what the `prune` tree subcommand does. It removes any subnodes of the given node and the corresponding canvas items, but otherwise leaves the node intact.

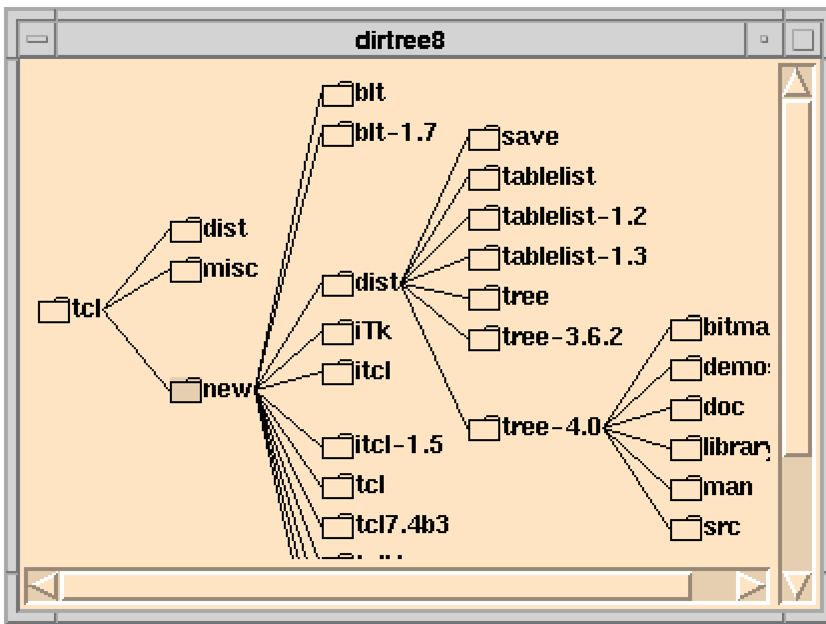
The other procedure, `ToggleParent`, is somewhat more complicated. It makes the selected node the root of the tree, unless it already is the root, in which case a new root and its subnodes is inserted before the selected node.

The `isroot` tree subcommand is similar to `isleaf` used above. It returns 1 if the given node is the root of the tree and 0 otherwise.

If the selected node is the root of the tree (but the directory is not `/`), then we add a new subtree for the parent directory, initially under the *pseudo* root node (tag = `""`). We then add the subdirectories of the new root node to the tree using `AddDir` as before, except that when we come to the node that was originally selected, instead of re-creating it, we move it to its new position under the new root node using the `movelink` tree subcommand. `movelink` works much like the UNIX `mv` command when both arguments are directories. It unhooks the node named by the first argument and inserts it as a child of the node named by the second argument.

In the case where the selected node was not the root of the tree, we simply use the tree `root` subcommand to make the given node the new root of the tree. The canvas items for any nodes that are no longer accessible from the root node are automatically deleted.

The figure below shows the tree after navigating up the directory hierarchy a few levels.




## Pruning Trees, Moving and Removing Nodes

In the previous example we used the `prune` and `movelink` tree subcommands. The `prune` subcommand is used to remove the subnodes from the given node. The `movelink` command moves the given node to a new position in the tree. One other command that we did not discuss yet is used to remove the named node and all of its subnodes from the tree. The `rmlink` tree subcommand is similar to the `prune` subcommand, except that the given node is also removed from the tree. The example below adds key bindings to the previous example so that Control-d removes the selected node and its subtree.

```
bind $canvas  "HideNode $canvas $tree"


# remove the selected node and its subnodes from the display

proc HideNode {canvas tree} {
    set path [GetPath $canvas]
    if {"$path" != "" && ![$tree isroot $path]} {
        $tree rmlink $path
    }
}
```

## Setting a New Root Node

As we saw in the previous example, any tree node that is not already the root of the tree can be made the root of the tree by using the `root` subcommand. Any nodes that are not subnodes of the given node will be deleted from the tree and the corresponding canvas items deleted.

## Displaying A Forest

When we talk about the *root* of the tree, we usually mean the visible root. As mentioned earlier, there is actually a *pseudo* root node that is not visible and has the empty tag `{}`. Since you add the visible root by specifying this pseudo root as the parent, it is also possible to display multiple trees in one canvas by adding multiple visible root nodes under this node. This kind of thing might be useful for displaying a class hierarchy where there might not be a single root.

## Traversing the Tree

There are a number of tree subcommands available for querying and traversing the tree that is being displayed. Each of the following subcommands takes as an argument the canvas tag or id of a tree node and returns the tags of one or more tree nodes as a result:

- The `subnodes` command returns a Tcl list of the given node's subnodes.
- The `parent` subcommand returns the name of the parent of the given node. If the node has no parent (i.e.: is a root node), an empty string is returned.
- The `child` subcommand returns the tag of the first child node of the given node. If the node has no children, an empty string is returned. The `sibling` subcommand can be used to get the names of the other child nodes in sequence.
- The `sibling` subcommand returns the name of the next sibling of the given node. If the node has no siblings, an empty string is returned.
- The `ancestors` subcommand returns a Tcl list of the given nodes ancestors, for example: {parent grandparent ...}.

Let's use the above commands to implement key bindings for our example tree so that you can use the cursor (arrow) keys to navigate the tree.

```
# add bindings for the arrow keys

bind $canvas  "SelectNext $canvas $tree %K"
bind $canvas  "SelectNext $canvas $tree %K"
bind $canvas  "SelectNext $canvas $tree %K"
bind $canvas  "SelectNext $canvas $tree %K"


# select the current node's parent, child or sibling
# depending on the value of direction (Left, Right, Up or Down)

proc SelectNext {canvas tree direction} {
    set id [$canvas select item]
    set path [lindex [$canvas gettags $id] 0]

    # for vertical trees its different...
    if {[$tree cget -layout] == "vertical"} {
        case $direction in {
            Left        {set direction Up}
            Right       {set direction Down}
            Up          {set direction Left}
            Down        {set direction Right}
        }
    }

    case $direction in {
        Left {
            set node [$tree parent $path]
            if {"$node" == ""} {
                ToggleParent $canvas $tree
                set node [$tree parent $path]
            }
        }
        Right {
            set node [$tree child $path]
            if {"$node" == ""} {
                ListDirs $canvas $tree $path
                set node [$tree child $path]
                $tree draw
            }
        }
```

```
        Down {
            set node [$tree sibling $path]
            if {"$node" == ""} {
                set node [$tree child [$tree parent $path]]
            }
        }
        Up {
            set next [$tree child [$tree parent $path]]
            while {"$next" != ""} {
                set node $next
                set next [$tree sibling $next]
                if {"$next" == "$path"} {
                    break;
                }
            }
        }
        default {return}
    }

    if {"$node" != ""} {
        set next [$canvas find withtag $node:text]
        $canvas select from $next 0
        $canvas select to $next [string length [$canvas itemcget $next -text]]
    }
}
```

The `SelectNext` procedure above takes as arguments the name of the tree and canvas widgets and a *direction*, which is one of the arrow key names `Up`, `Down`, `Right` or `Left`. The action taken depends on the key was pressed, the selected node and the layout of the tree. For example, if the tree layout is horizontal, pressing `Left` will traverse the tree selecting the current node's parent, adding it to the tree first if necessary. Pressing `Right` will traverse the tree in the other direction adding subtrees as needed. The `Up` and `Down` arrow keys don't add any nodes, they only traverse the selected node's siblings.

## Reconfiguring Tree Nodes

When you make any changes to the canvas items that make up a tree node, such as rearranging the layout, adding or removing items, the tree widget needs to be told to recalculate the node's size. The `nodeconfigure` subcommand does this and also allows you to change the node's options. `nodeconfig` accepts the same options as the `addlink` command:

`-border` *width*
: Specifies the width of the nodes border. This determines the distance between the individual nodes in the tree. The default is 0.

`-remove` *command*
: Specifies a Tcl command to be evaluated when the node is removed from the tree. This option is necessary when a node contains a window such as a listbox or button, so that these will also be properly deleted. The default action, when a node is to be removed, is to use the canvas `delete` command to delete the canvas items with the given tag or id.

The `-remove` option will be used in the next example, where we add Tk windows to tree nodes.

## Using Tk Windows in Tree Nodes

In the examples so far the trees have only displayed directories. Normally you would also want to see a list of the files in a selected directory. We could display files in the tree along with the directories, say with a different bitmap, but that might clutter up the tree too much, since there

tend to be many more files than directories. We could display a list of files in the selected directory in a separate listbox. In fact, we could make the listbox a part of a tree node and display it there when needed. Whether this is a good idea or not probably depends on the application, but for the sake of example, let's add bindings to the example directory tree application so that clicking on a tree node with the middle mouse button opens or closes a listbox under the node listing the files in the directory. In order to be useful, the example also has to allow resizing of the listbox and add code to get the name of a selected file.

```
# add a binding to tree node labels to open/close a listbox listing files in
# the selected directory
$canvas bind text <2> "focus %W; SelectNode $canvas; ShowFiles $canvas $tree"


# Display the file names in the selected directory in
# a scrolling list beneath the node

proc ShowFiles {canvas tree} {
    set dir [GetPath $canvas]
    set frame [UniqueName $canvas $dir]

    # create the frame and list if not already there
    if {![llength [$canvas find withtag $dir:list]]} {
        MakeListFrame $canvas $tree $frame $dir
    } else {
        RemoveListFrame $canvas $tree $frame $dir
    }
}


# generates a unique name for a widget from the given
# directory name by replacing the '.'s with '_'s

proc UniqueName {canvas dir} {
    global dirtree
    if {![catch {set path $dirtree($canvas.$dir)}]} {
        return $path
    }
    if {[catch {incr dirtree(listboxcnt)}]} {
        set dirtree(listboxcnt) 0
    }
    set dirtree($canvas.$dir) $canvas.list$dirtree(listboxcnt)
    return $dirtree($canvas.$dir)
}


# make the list frame for displaying the files in dir

proc MakeListFrame {canvas tree frame dir} {
    global dirtree
    set bg [$canvas cget -bg]

    # make the list frame and set up resizing
    frame $frame -borderwidth 3 -cursor bottom_right_corner
    scrollbar $frame.scroll -relief sunken -command "$frame.list yview" -width 10
    listbox $frame.list -yscrollcommand "$frame.scroll set" -relief sunken      ₩
        -bg $bg -selectmode single
    pack $frame.scroll -side right -fill y
    pack $frame.list -side left -expand yes -fill both

    # fill the list with the files in the selected dir
    # (with the uninteresting files filtered out)
    set n 0
    foreach i [exec ls $dir] {
        if {[file isfile $dir/$i]} {
            incr n
            $frame.list insert end $i
        }
```

```
    }

    # insert the list frame in the canvas
    $canvas create window 0 0 -tags "$dir $dir:list list" ₩
        -window $frame -width 3c -height 2c

    # arrange the items in the tree node and change the bitmap
    # to show an "open" directory
    LayoutNode $canvas $tree $dir
    $canvas itemconfig $dir:bitmap -bitmap "@bitmaps/open_dir.xbm"
    $tree nodeconfig $dir -remove "destroy $frame"

    # add bindings for resizing the listbox in the canvas
    SetFileListBindings $canvas $tree $frame $frame.list $frame.scroll $dir
}


# remove the list frame for displaying the files in dir

proc RemoveListFrame {canvas tree frame dir} {
    global dirtree
    $canvas delete $dir:list
    destroy $frame

    $canvas itemconfig $dir:bitmap -bitmap @bitmaps/dir.xbm
    $tree nodeconfig $dir -remove "" -border 2
    set dirtree(list) ""
    set dirtree(dir) ""
}


# event proc called to resize the file listbox in the canvas
# while the user is dragging its corner

proc ResizeList {canvas tree frame list x y dir {when any}} {
    global dirtree

    case "$when" {
        "first" {
            set dirtree(tlx) $x
            set dirtree(tly) $y
        }
        "last" {
            $tree nodeconfig $dir
        }
        default {
            set w  [$canvas itemcget $dir:list -width]
            set h  [$canvas itemcget $dir:list -height]
            set nw [expr $w+($x-$dirtree(tlx))]
            set nh [expr $h+($y-$dirtree(tly))]
            $canvas itemconfig $dir:list -width $nw -height $nh
            set dirtree(tlx) $x
            set dirtree(tly) $y
        }
    }
}


# set the bindings for a file listbox in the tree canvas

proc SetFileListBindings {canvas tree frame list scroll dir} {
    global dirtree
    bind $frame  "ResizeList $canvas $tree $frame $list %x %y $dir first"
    bind $frame  "ResizeList $canvas $tree $frame $list %x %y $dir"
    bind $frame  "ResizeList $canvas $tree $frame $list %x %y $dir last"

    bind $list <1>  "set dirtree(list) %W; set dirtree(dir) $dir; [bind Listbox <1>]"
    bind $list  ChooseFile
```

```
    # set/clear the resize cursor
    bind $list  "$frame config -cursor {}"
    bind $list  "$frame config -cursor bottom_right_corner"
    bind $scroll  "$frame config -cursor {}"
    bind $scroll  "$frame config -cursor bottom_right_corner"
}


# This proc is called when the user double-clicks on a filename
# in one of the listboxes listing the files in a directory.

proc ChooseFile {} {
    set file [GetFilename]
    if {"$file" == ""} {return}

    # for test purposes, just print the file name here
    puts "got $file"
}


# Return the path name of the file currently selected in the
# current node's listbox, or the empty string if none are selected.

proc GetFilename {} {
    global dirtree
    set dir $dirtree(dir)
    set list $dirtree(list)
    if {"$dir" == "" || "$list" == ""} {return ""}

    set sel [$list curselection]
    if {![llength $sel]} {return ""}
    return $dir/[$list get [lindex $sel 0]]
}


# layout the components of the given node depending on whether
# the tree is vertical or horizontal
# (Also take the file listbox into consideration if it is open)

proc LayoutNode {canvas tree dir} {
    set text $dir:text
    set bitmap $dir:bitmap
    set list [$canvas find withtag $dir:list]

    if {[$tree cget -layout] == "horizontal"} {
        scan [$canvas bbox $text] "%d %d %d %d" x1 y1 x2 y2
        $canvas itemconfig $bitmap -anchor se
        $canvas coords $bitmap $x1 $y2
        if {"$list" != ""} {
            scan  [$canvas bbox $text $bitmap] "%d %d %d %d" x1 y1 x2 y2
            $canvas itemconfig $list -anchor nw
            $canvas coords $list $x1 $y2
        }
    } else {
        scan [$canvas bbox $bitmap] "%d %d %d %d" x1 y1 x2 y2
        $canvas itemconfig $text -anchor n
        $canvas coords $text [expr "$x1+($x2-$x1)/2"] $y2
        if {"$list" != ""} {
            scan  [$canvas bbox $text $bitmap] "%d %d %d %d" x1 y1 x2 y2
            $canvas itemconfig $list -anchor n
            $canvas coords $list [expr "$x1+($x2-$x1)/2"] $y2
        }
    }
}
```

The figure below shows the example application now after some file lists have been opened and resized.

As usual when working with the tree widget, most of the operations have more to do with the canvas widget than with the tree. The tree widget only arranges the nodes in the canvas.

The `ShowFiles` procedure in the previous example is called to open or close the file list for the selected node in the tree. If the file list is present in the tree (the canvas tag `$dir:list` is found), it is removed and deleted, otherwise a new frame with a unique name is created as a child of the canvas widget. The listbox and scrollbar will be packed into this frame.

The `MakeListFrame` procedure takes care of filling out the contents of the new frame in the tree. It creates the listbox and scrollbar and packs them in the frame and then fills the listbox with the list of files in the selected directory. The `LayoutNode` procedure from earlier examples has been extended here to handle the layout when a listbox frame is part of the node. We also add some bindings to the listbox frames here so that you can resize them by dragging on the corner of the frame with mouse button 1.

The important thing to remember when working with windows in tree nodes or as canvas items in general is that, since they are not managed by a geometry manager, such as the *packer*, you have to set and get the window size with canvas commands and not with the `-width` or `-height` widget options. We set the default size of the listbox frame when we create the canvas window item:

```
$canvas create window 0 0 -tags "$dir $dir:list list" ₩
    -window $frame -width 3c -height 2c
```

And later, when we want to resize the listbox frame interactively, we use the canvas `itemconfig` subcommand:

```
set w  [$canvas itemcget $dir:list -width]
set h  [$canvas itemcget $dir:list -height]
    ...
$canvas itemconfig $dir:list -width $nw -height $nh
```

It is important that the canvas knows the size of the window, so that the tree can get the correct information from the canvas about the size of the the tree nodes.

# An [incr Tk] Interface to the Tree Widget

The example directory tree application developed in the previous section only made use of standard Tcl, Tk and the tree extension. In practice, I almost always use a *wish* with at least the *[incr Tcl]*, *TclX* and *BLT* extensions also added. In particular, I prefer to organize Tcl code in an object oriented way using [incr Tcl] classes. The [incr Tcl] extension, also called *Itcl*, adds classes and inheritance to Tcl and also includes features that make it easy to create *mega-widgets* or combinations of widgets that behave like the standard Tk widgets.

The tree widget release also contains an [incr Tk] version of the Tree widget. In order to use the version of the tree widget, you use the capitalized widget name `Tree` instead of `tree`. The Itcl tree widget accepts all of the standard tree options and subcommands and passes them on to the underlying tree widget. In addition, the Itcl widget takes care of creating the canvas, tree and scrollbars, defines a standard layout for tree nodes, handles node selection and offers methods for common operations on trees. Below is an example of a simple tree application using the Itcl tree class.

```
class Simple {
    inherit iwidgets::Dialog
```

```
    # constructor: create a toplevel window for the demo

    constructor {args} {

        wm withdraw .

        # create an instance of the Itk "Tree" widget
        # (based on the C++ "tree" widget)
        itk_component add tree {
            Tree $itk_interior.tree
        }
        pack $itk_component(tree) -fill both -expand 1

        eval itk_initialize $args
        wm title $itk_component(hull) {Simple [incr Tcl] Tree Demo}
    }


    # draw a tree in the canvas for demonstration purposes

    method draw_tree {} {
        # add some nodes to the tree
        add_nodes {} root
        add_nodes root node1 node2 node3 node4
        add_nodes node1 node1.1 node1.2 node1.3
        add_nodes node2 node2.1 node2.2 node2.3
        add_nodes node3 node3.1 node3.2
    }


    # Add one or more subnodes to the given parent node.
    # In this case, the label is just the same as the node's tag

    method add_nodes {parent args} {
        foreach node $args {
            $itk_component(tree) addnode $parent $node ₩
                    -bitmap @$bitmap ₩
                    -text $node
        }
    }

    private variable bitmap "../bitmaps/node.xbm"
}


# Start the demo:
Simple .s -modality application
.s center
.s draw_tree
.s activate
exit
```

The figure below shows the window created by the simple application above:

In the example above, we create a class named `Simple` that inherits its top level window from class `Dialog`, which is part of the Itcl2.x [incr Widgets] distribution.