# The GTK+ Tree View

## 1 Overview

The GTK+ TreeView widget, called `GtkTreeView` in the C language binding, is based on a *Model/View/-Controller* (MVC) design paradigm. In this paradigm, the visual display of information is separate from the object that encapsulates and provides methods for accessing that information. The `GtkTreeView` widget is the widget for presenting the data, and a `GtkTreeModel` encapsulates the data. This is analogous to how the `GtkTextView` displays the data stored in a `GtkTextBuffer`. There is a big difference between the tree view and the text view however. A `GtkTreeModel` is an abstraction; it is purely an interface that must be implemented by some other widget. A program cannot instantiate a `GtkTreeModel`. It can only instantiate a concrete class that implements it. Fortunately, there are two such classes provided by GTK: the `GtkListStore` and the `GtkTreeStore`. As you would expect, the `GtkListStore` implements a list and the `GtkTreeStore` implements a tree.

There are four major components to understand in order to use a `GtkTreeView`:

- The tree view widget itself (`GtkTreeView`)
- The tree view column (`GtkTreeViewColumn`)
- The cell renderer and its subclasses (`GtkCellRenderer`), and
- The tree model interface (`GtkTreeModel`).

The physical appearance of the tree view is governed by the first three objects, while the data contained in it is entirely determined by the model.

Just as multiple text view widgets could display a single text buffer, multiple tree views can be created from a single model. A good example is how file managers/browsers are constructed. Consider a model that contains a mapping of the file system. Many tree view widgets can be created to display various parts of the file system, with only one copy of the model in memory. Any change made in one view would be reflected immediately in the others.

The `GtkTreeViewColumn` object represents a visible column in a `GtkTreeView` widget. It lets you set specific properties of the column header and the column in general (visibility, spacings, width, sortability, etc.), and acts like a container for the cell renderers which determine how the data in the column is displayed.

The `GtkCellRenderer` objects provide extensibility to the tree view widget. They provide a means of rendering a single type of data in different ways. For example, different cell renderers can render a boolean variable as a string of `"True"` or `"False"` or `"On"` or `"Off"`, or as a checkbox. Multiple cell renderers can be packed into a single column.

Figure 1 shows a `GtkTreeView` widget displaying a `GtkListStore` consisting of rows whose members include images, boolean values rendered as check boxes, numeric values rendered as dollar amounts, and strings.

## 2 A Bit of Terminology

Perhaps it is obvious, but to be clear, a *row* is a single horizontal set of data. A *column* is a vertical slice through the tree view, containing all values in that column. The *title* of a column is the label that appears at the top of the column. In Figure 1, the first column's title is "Image". A *cell* is the intersection of a column and a row. Cells are thus parts of rows and columns.

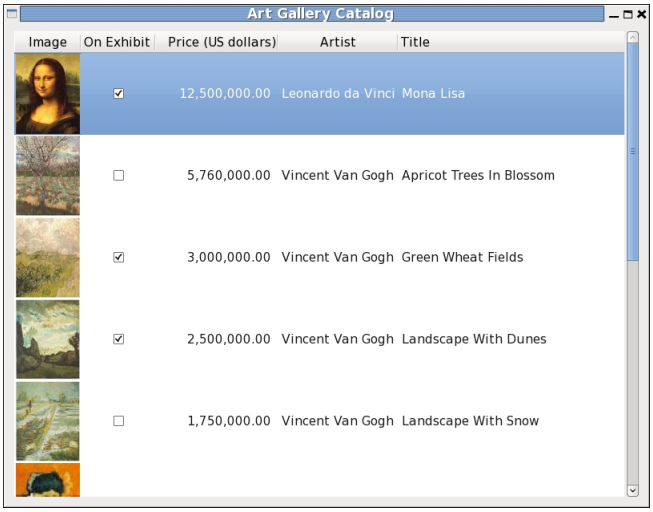


Figure 1: A GtkTreeView displaying a list

# 3 Creating a Tree View

In order to create a tree view, you must:

1. Create the tree model,
2. Create and configure the tree view, and
3. Set the model into the tree view.

We now explain these tasks in more detail.

## 3.1 Creating the Tree Model

GTK+ provides two existing implementations of the `GtkTreeModel`: the `GtkListStore` and the `GtkTreeStore`. The `GtkListStore` is used to model lists of data, whereas the `GtkTreeStore` models trees. These models should be sufficient for most of your purposes.

To create a list store, you should use the function

```
GtkListStore * gtk_list_store_new                ( gint n_columns,
                                                   ...);
```

This function has a variable number of arguments. The first argument is the number of columns that will be in the store. If the first argument is $n$, then there must be $n$ arguments following. Each successive argument must be the name of a `GType`. These are names such as `G_TYPE_INT`, `G_TYPE_BOOLEAN`, `G_TYPE_STRING`, and `GDK_TYPE_PIXBUF`. For example, to create a list store with three columns, one containing a `GdkPixBuf` image, one, a string, and one, an integer, you would write

```
liststore = gtk_list_store_new (3, GDK_TYPE_PIXBUF, G_TYPE_STRING, G_TYPE_INT);
```

As a coding convenience, it is better to create an enumerated type containing names for the columns that will be in the store, followed by a name that will store the total number of values in the enumeration:

```
typedef enum {
    IMAGE_COLUMN,    // 0
    TITLE_COLUMN,    // 1
    QUANTITY_COLUMN, // 2
    NUM_COLUMNS      // 3
} Column_type;
```

This way, you can write, instead,

```
liststore = gtk_list_store_new (NUM_COLUMNS, GDK_TYPE_PIXBUF, G_TYPE_STRING, G_TYPE_INT);
```

and if you need to add a new column, you will not have to change the value of `NUM_COLUMNS`.

Creating a tree store is almost the same; the function is

```
GtkTreeStore * gtk_tree_store_new               ( gint n_columns,
                                                  ...);
```

The parameters have the same meanings as in `gtk_list_store_new()`. This creates a tree store rather than a list store. The differences will be apparent shortly.

## 3.2 Adding Data to the Tree Model

Data is added to an implementation of a `GtkTreeModel` one row at a time. In general, it is a two-step process, in which you first create a new, empty row, setting an iterator to that row, and then add the data to that new row. There are a few shortcut functions that let you do this in a single step, but it is usually easier from a programmer's point of view to use the two-step method. The functions for adding to a list store are different from those that add to a tree store. We will begin with the list store.

### 3.2.1 Adding to a List Store

To add data to a list store, you first need to create a row and acquire an iterator to that row. There are several different ways to do this, depending on whether you want to prepend, append, insert at a specific position, or insert after or before a given, existing row. The functions to prepend or append are

```
void           gtk_list_store_prepend           ( GtkListStore *list_store,
                                                  GtkTreeIter *iter);
void           gtk_list_store_append            ( GtkListStore *list_store,
                                                  GtkTreeIter *iter);
```

These are both given a pointer to the list store and the address of an uninitialized `GtkTreeIter`. They each create a new row and set the iterator to point to that row. The difference is that the prepend function creates the row at the beginning of the list, whereas the append function creates the row after all existing rows. Notice that the iterator is a *tree iterator*, not a list iterator.

The function to insert at a specific position is a generalization of the preceding two functions. Its prototype is

```
void           gtk_list_store_insert            ( GtkListStore *list_store,
                                                  GtkTreeIter *iter,
                                                  gint position);
```

The only way in which it differs from the preceding two functions is that a row is created at the position specified by the integer parameter `position`. It is an error if `position` is less than zero. Otherwise, `position` specifies the index of the row in a zero-based list of rows. If `position` is larger than the number of rows in the list, it is appended to the list. Remember that this is a list and visualize the insertion as the insertion of a list node at that point, effectively increasing the positions of all existing rows from the existing one at that position until that of the last row.

The two other functions can be useful when you have a specific row and need to do an insertion before or after it, but do not have its position, as when a user selects a row and clicks a proxy to insert a new row there (or when you want to drag a row on top of the treeview and have it drop at a given place.) Their prototypes are

```
void            gtk_list_store_insert_before      ( GtkListStore *list_store,
                                                    GtkTreeIter *iter,
                                                    GtkTreeIter *sibling);
void            gtk_list_store_insert_after       ( GtkListStore *list_store,
                                                    GtkTreeIter *iter,
                                                    GtkTreeIter *sibling);
```

Their meanings are perhaps obvious. They work the same way as the others, except that they are given in their third parameter the address of a tree iterator, `sibling`, that has been set to an existing row. If `sibling` is not `NULL`, then the new row will be created before or after `sibling` accordingly. If sibling is `NULL`, then `gtk_list_store_insert_before()` will append the new row and `gtk_list_store_insert_after()` will prepend it.

Once you have an iterator to an empty row, you can set data into it. You again have a few choices. You can insert data into multiple cells of the row in one, fell swoop, or insert data into a single cell at a time. To add to multiple cells, use

```
void            gtk_list_store_set                ( GtkListStore *list_store,
                                                    GtkTreeIter *iter,
                                                    ...);
```

The first two arguments are the list in which to insert and the tree iterator that points to the row whose cells are to be set. The remaining arguments are a sequence of pairs terminated by a -1. Each pair consists of an integer column number followed by the value to assign to the cell in that column. For example, if column 0 has data of type `G_TYPE_STRING`, column 1 has data of type `G_TYPE_INT`, and column 2, of type `G_TYPE_BOOLEAN`, then we can call the function with

```
gtk_list_store_set( liststore, &iter, 0, "A string", 1, 64, 2, TRUE, -1);
```

You need to know when data is copied versus when the store acquires a pointer to it and increments its reference count. These are summarized as follows:

1. If the data is a `GObject` (i.e., an object derived directly from `GObject`) , then the store takes ownership of it by calling `g_value_dup_object()` on the object and relinquishing ownership of the previously held object in that cell, if it is replacing an old value. Pixbufs fall into this category.
2. If the data is a simple scalar data type such as a numeric, Boolean, or enumerated type or a pointer, then the store makes a copy of the data. In particular, if the data is a pointer, the pointer is copied, not the data to which it points.
3. If the data is a string or a boxed structure, then the store duplicates the string or the boxed structure and stores a pointer to it. If the data is replacing existing data, then in this case the old string or boxed structure is freed first using `g_free()` or `g_boxed_free()` respectively. ( `GBoxed` is a wrapper mechanism for arbitrary C structures. They are treated as opaque chunks of memory.)

Usually when you call this function, instead of passing column numbers as numeric literals, you will use the values of an enumerated type. Also, the data will be in variables, not literals. The following code fragment illustrates these steps in creating a list store. Some details are omitted.

Listing 1: Creating a list store from data in a file.

```
// Assume the following enumeration has been declared:
typedef enum
{
  ON_EXHIBIT,
  PRICE,
  ARTIST,
  TITLE,
  NUM_COLUMNS
} Column_type;

// Assume a store has been created with the columns specified here:
GtkListStore *store = gtk_list_store_new ( NUM_COLUMNS,
                                   G_TYPE_BOOLEAN,
                                   G_TYPE_FLOAT,
                                   G_TYPE_STRING,
                                   G_TYPE_STRING
                                  );

/* Open a file, and get the data out of the file, record by record.
   Assume that each record contains the data for a single row. Also
   assume that:
      on_exhibit is a boolean
      price is a float
      artist is a string
      title is a string
*/

// Within a loop, these steps would be taken:
      gtk_list_store_append (*store, &iter);
      gtk_list_store_set (*store,   &iter,
                          ON_EXHIBIT,   on_exhibit,
                          PRICE,        price,
                          ARTIST,       artist,
                          TITLE,        title,
                          -1);
```

The function to add data to a single cell in the store is

```
void          gtk_list_store_set_value         ( GtkListStore *list_store,
                                                 GtkTreeIter *iter,
                                                 gint column,
                                                 GValue *value);
```

This will set the given value into the cell specified by column in the row pointed to by iter. The value parameter is a pointer to a `GValue`. A `GValue` is defined in the GObject library. It is an opaque structure, which means that you do not have access to its members. The GObject library has methods that act on `GValue` objects. When you use this function, you must supply a pointer to a value that can be cast to the type of the given column. The GObject type system will cast it accordingly.

### 3.2.2 Adding Data to a Tree Store

A tree store is hierarchical. Every row is the child of some parent, and if the parent row is not visible in the view, its children will be invisible as well. Adding data to a tree store is a bit more complex because of this

hierarchical structure. A new row must be made the child of an existing row. Generally speaking, each of the functions defined for inserting into a list store has an analog in the tree store with an extra parameter that specifies the parent whose child that row is to become. The complications arise because of the possibilities of that parameter being `NULL`, or the sibling, if it is supplied, being `NULL`.

It will be easy to sort things out if you remember that *the root of the tree store is not any of the rows you put into it, but an invisible, abstract row that has no data.* We call this invisible row the *root* of the tree. When a row is added to the tree with a `NULL` parent, it means that this row is a top-level row, an immediate child of the root. Another way to see this is that your tree is not really a tree, but a forest whose root is not part of your tree, but is a part of the library.

The simplest case is appending a new row to the children of a given parent, which is done with

```
void            gtk_tree_store_append           ( GtkTreeStore *tree_store,
                                                  GtkTreeIter *iter,
                                                  GtkTreeIter *parent);
```

If `parent` is not `NULL`, then the row is appended after the last child of this `parent`. Otherwise, it is created as a top-level row and appended after the last top-level row.

There is a similar function to prepend a row before all children of a given parent:

```
void            gtk_tree_store_prepend          ( GtkTreeStore *tree_store,
                                                  GtkTreeIter *iter,
                                                  GtkTreeIter *parent);
```

The semantics are analogous to the appending version.

There are functions to insert at a specific position in a child list, or to insert before or after a sibling within a child list. To insert a row at a specific position within the list of children of a given parent row, use

```
void            gtk_tree_store_insert           ( GtkTreeStore *tree_store,
                                                  GtkTreeIter *iter,
                                                  GtkTreeIter *parent,
                                                  gint position);
```

This is like the corresponding function for the list store, except that it is relative to the list of children of the given parent. That is, if `parent` is not `NULL`, it inserts a new row within the parent's child list at the given `position`. If `position` is larger than the number of children, it is appended after the last child. If `position` is negative it is an error. If `parent` is `NULL`, the row is inserted as a top-level row using the same rules for `position`.

The two functions for inserting relative to a sibling are

```
void            gtk_tree_store_insert_before    ( GtkTreeStore *tree_store,
                                                  GtkTreeIter *iter,
                                                  GtkTreeIter *parent,
                                                  GtkTreeIter *sibling);
```

and

```
void            gtk_tree_store_insert_after     ( GtkTreeStore *tree_store,
                                                  GtkTreeIter *iter,
                                                  GtkTreeIter *parent,
                                                  GtkTreeIter *sibling);
```

The semantics are a bit complex. There are four cases to consider for each.

**Case 1: `parent` is not `NULL` and `sibling` is not `NULL`.** In this case, both of these functions first check whether `sibling` is a child of `parent`. If not, it is an error. If it is, then `gtk_tree_store_insert_before()` inserts a new row before the row pointed to by `sibling`, and `gtk_tree_store_insert_after()` inserts a new row after it.

**Case 2: `parent` is `NULL` and `sibling` is not `NULL`.** This is the same as Case 1 except that no check is performed; a row is inserted before or after `sibling` as specified.

**Case 3: `parent` is not `NULL` and `sibling` is `NULL`.** In this case, `gtk_tree_store_insert_before()` will append a new row after its last child, and `gtk_tree_store_insert_after()` will prepend a new row before its first child.

**Case 4: `parent` is `NULL` and `sibling` is `NULL`.** `gtk_tree_store_insert_before()` will append a new row to the top-level, and `gtk_tree_store_insert_after()` will prepend a new row to the top-level.

The functions to set data in a tree store row have the exact same semantics and form as those of the list store. They differ only in name:

```
void          gtk_tree_store_set              ( GtkTreeStore *tree_store,
                                                GtkTreeIter *iter,
                                                ...);
void          gtk_tree_store_set_value        ( GtkTreeStore *tree_store,
                                                GtkTreeIter *iter,
                                                gint column,
                                                GValue *value);
```

## 3.3 Creating and Configuring the Tree View Component

### 3.3.1 Creating the Tree View

There are two functions to create a tree view widget, one with an existing model, and one without a model. To create a tree view with an existing model, use

```
GtkWidget *    gtk_tree_view_new_with_model    ( GtkTreeModel *model);
```

This initializes the tree view's model to the model passed to it. To create a tree view with no model, use

```
GtkWidget *    gtk_tree_view_new               ( void);
```

If you create a tree view without a model you then have to set the model into it with

```
void          gtk_tree_view_set_model         ( GtkTreeView *tree_view,
                                                GtkTreeModel *model);
```

This will set the given model into the tree view. If the tree view already has a model, it will be removed. If `model` is passed as a `NULL` value, then the old model is unset.

Once the tree view has been created, the next step is to configure how it will display the data in its model. Display is controlled by (1) the global properties of the `GtkTreeView` itself, (2) the `GtkTreeColumn` object, and (3) the `GtkCellRenderer` objects. Global properties include things such as making headers visible or hidden.

### 3.3.2 Creating and Adding Columns

In simple cases, creating and adding columns is a relatively easy task, but the `GtkTreeViewColumn` object has been designed with a great deal of flexibility in mind.

A `GtkTreeViewColumn` is the object that `GtkTreeView` uses to organize the vertical columns in the tree view. To use a tree view column, you have to give it the title that will be displayed in the column header and a set of one or more cell renderers that will render the data in the column. The column object does not do any drawing itself; it is just a container for the cell renderers. In the simplest case, a column has a single cell renderer, but there are times when you may want to give it more than one.

In a sense, describing how to add and customize columns is like putting the cart before the horse, because unless you understand what a cell renderer does, you will not understand how to create the columns. Therefore, we discuss the cell renderers briefly before continuing.

**Overview of GTK Cell Renderers** The `GtkCellRenderer` is derived directly from `GObject`. It is not a widget and has no window. It is the base class of a set of objects that can render a cell onto a `GdkDrawable`, i.e., the underlying window of the treeview widget. Although these objects exist independently of the `GtkTreeView` widget and could be used outside of it, they exist in order to render the cells in `GtkTreeView` widgets. There are several subclasses of the `GtkCellRenderer`:

- `GtkCellRendererText` for rendering text in a cell.
- `GtkCellRendererPixbuf` for rendering images in a cell.
- `GtkCellRendererProgress` for rendering numeric values as progress bars
- `GtkCellRendererSpinner` for rendering a spinning animation in a cell
- `GtkCellRendererToggle` for rendering a toggle button in a cell.

The base class has a set of properties such as "width" , "height", "cell-background", "visible", and more, and the child classes add to these properties their own specific ones. For example, a `GtkCellRendererText` object has a "foreground" property and a "foreground-set" property. A cell renderer applies its properties to every cell in the column that it renders. So if the "foreground" is set to "Blue" and the "foreground-set" property is set to `TRUE` (meaning that the renderer should use the foreground property), then every cell will have blue text. Cell renderer properties are set with the underlying GObject `g_object_set()` function.

To repeat, the properties of a cell renderer are applied to every cell in the column. One way to render different cells differently, is to use a method of the `GtkTreeViewColumn` class to specify this. This may seem counterintuitive at first, but you will see why it makes sense soon. A second method is to use a special function called a `GtkTreeCellDataFunc` function. This is a function that can be attached to a renderer and that will be called for each data cell in the column. The cell renderer function can use the data in the cell to customize the appearance of that cell.

Returning to the tree view columns, the simplest way to create a column is with a function that creates it and also sets the attributes of a single cell renderer to use in that column. This function is

```
GtkTreeViewColumn * gtk_tree_view_column_new_with_attributes
                                                ( const gchar *title,
                                                  GtkCellRenderer *cell,
                                                  ...);
```

This function has a variable number of arguments. The first argument is the title of the column. This will be displayed in the column's header. The second argument is a `GtkCellRenderer`, which will draw each cell's contents on the underlying GDK window. Following the cell renderer is a `NULL`-terminated list of

attribute-value pairs. Each pair consists of a cell renderer property name, such as "text", and the number of a `GtkTreeModel` column from the store that is set in the tree view. The cell renderer will take the data from that column of the model and apply its value to the property that is associated with it. For example, the lines

```
renderer = gtk_cell_renderer_text_new ();
column = gtk_tree_view_column_new_with_attributes
                          ("Artist", renderer, "text", ARTIST, NULL);
```

will create a text renderer, storing a pointer to it in `renderer`, and create a column whose title is "Artist" and pack `renderer` into it. A `GtkCellRendererText` has a property called "text", and the above call tells the column object that every time the renderer renders a cell in this column, it should set its "text" property to have the value of the cell from the `ARTIST` column in the model. If the model's `ARTIST` column's first row has the string "Renoir", then the cell renderer will print the string "Renoir" in that cell.

This function will remove all existing attributes from the column. Sometimes, you may want to preserve a column's existing attributes but add a new one. This would be the case if you want a column to have two or more renderers. You might want to do this to render the data from a single column in different ways in the same column. An example of this is when you have a boolean value in a model column, and you want to put both a checkbox and a string of some kind into the column. You could pack a `GtkCellRendererToggle` into the column as well as a `GtkCellRendererText` into the same column. The former will display the boolean as a checkbox, and the latter, as the word `TRUE` or `FALSE`. (This word can be customized, as you will see later.)

To put two renderers into a single column, you have to add a second renderer to the column after putting the first one into it. The function to do this is

```
void          gtk_tree_view_column_add_attribute ( GtkTreeViewColumn *tree_column,
                                                   GtkCellRenderer *cell_renderer,
                                                   const gchar *attribute,
                                                   gint column);
```

This adds a single attribute pair to a cell renderer, packing it into the column.

## 3.4 Specific Cell Renderers

### 3.4.1 GtkCellRendererToggle

A `GtkCellRendererToggle` renders a toggle button in a cell. The button is drawn as a radio- or check-button, depending on the radio property. When activated, it emits the toggled signal. You create one with

```
GtkCellRenderer *gtk_cell_renderer_toggle_new      ( void);
```

The default is to create a checkbutton. To render it as a radio button, use

```
void          gtk_cell_renderer_toggle_set_radio  ( GtkCellRendererToggle *toggle,
                                                    gboolean radio);
```

with the second argument `TRUE`. In order to allow the user to change the state of the toggle, it needs to have its "activatable" property set to `TRUE`, which you can use the `g_object_set()` function to do, or call

```
void        gtk_cell_renderer_toggle_set_activatable ( GtkCellRendererToggle *toggle,
                                                       gboolean setting);
```

When the user clicks on the button, the renderer will emit a "toggled" signal. Since you must change the model data to reflect the change in the view, you must connect a callback to the signal that will update the model. The callback for this signal has the prototype

```
void          user_function                   ( GtkCellRendererToggle *cell_renderer,
                                                gchar                 *path,
                                                gpointer               user_data);
```

This function is passed the pointer to the emitting rendered and a string representing a tree path. Tree paths are explained below. In order to change the model, the user data should have a pointer through which you can access the model. You can pass in the tree view and get the model from it. The following callback illustrates. Assume that `ON_EXHIBIT` is the column number in the model whose data is rendered by the toggle button.

```
void on_exhibit_toggled    ( GtkCellRendererToggle *renderer,
                             gchar                 *path,
                             GtkTreeView           *treeview )
{
    GtkTreeModel  *model;
    GtkTreeIter    iter;
    gboolean       state;

    model = gtk_tree_view_get_model(treeview);
    if ( gtk_tree_model_get_iter_from_string ( model, &iter, path) ) {
        gtk_tree_model_get ( model, &iter, ON_EXHIBIT, &state, -1);
        gtk_list_store_set ( GTK_LIST_STORE (model), &iter, ON_EXHIBIT,
                             !state, -1 );
    }
}
```

The path is delivered to the callback by the signaling system. It is a reference to the specific row whose toggle button was clicked. In order to get the data in the cell, we need to pass an iterator to that row to `gtk_tree_model_get()`, as well as the number of the column in the model whose data is being rendered by the toggle. This callback hard-codes that number because it is only called for this particular toggle. If the row had several toggles, and we wanted to use a single callback, then we would have to either set the column number as data against the renderer, or pass it in the user data also.

### 3.4.2 GtkCellRendererSpinner

### 3.4.3 GtkCellRendererPixbuf

# 4 Referencing Rows

The `GtkTreeModel` provides a few different ways to access the nodes (rows) of the tree, and within a row, to access a particular column. The approach is to get a reference to the row, and having that, to access

the particular column on that node. The structures that reference a particular node in a model are the `GtkTreePath`, the `GtkTreeIter`, and the `GtkTreeRowReference`.

A tree path represents a potential node. Paths define locations within a generic model. A given model may not actually have a node at the specified path. This will be clear momentarily.

The `GtkTreeModel` provides methods that can convert a `GtkTreePath` into either an array of unsigned integers or a string. The string representation is a list of numbers separated by a colon. Each number refers to the offset at that level. for example

"`0`" refers to the first top-level row

"`0:0`" refers to the first child of the first top-level row

"`1:2:3`" refers to the fourth child of the third child of the second top-level row

"`1:0:0:4`" refers to the fifth child of the first child of the first child of the second top-level row

After a tree changes as a result of a row insertion or deletion, a path may no longer correspond to an actual row. One can define a path arbitrarily as well, using the function

```
GtkTreePath *  gtk_tree_path_new_from_string     ( const gchar *path);
```

Given a string in the above form, this creates a `GtkTreePath` and returns a pointer to it. The fact that such a path structure exists has no bearing on whether or not a node exists at that location! Many callback functions are supplied paths by the run-time signaling system. These will be valid paths pointing to rows at within which some event took place.

In contrast, a `GtkTreeIter` is a reference to a specific node on a specific model. These iterators are the primary way of accessing a model and are similar to the iterators used by `GtkTextBuffer`. They are generally statically allocated on the stack and only used for a short time. Like the iterators used with text buffers, tree iterators can become invalidated as a result of various changes to the tree. In the `GtkTreeModel` in general, when the model emits a signal, the iterators become invalid. Signals are emitted when rows are inserted or removed, so generally speaking, such changes invalidate iterators. In the tree and list store, this is not true though – they preserve iterators as long as the rows are valid, and when a row becomes invalid any iterator to it is invalidated as well. A specific implementation of a `GtkTreeModel` may or may not persist the iterators when the tree is changed. In recent version of GTK+, the `GTK_TREE_MODEL_ITERS_PERSIST` flag indicates whether that implementation preserves the validity of iterators when the rows they reference exist. This flag can be checked with `gtk_tree_model_get_flags()`.

The `GtkTreeRowReference` is like a `GtkTextMark` in the sense that it is a permanent reference to a row in the tree. This reference will keep pointing to the node to which it was first associated, so long as it exists. It listens to all signals emitted by model, and updates its path appropriately. Therefore, these row references are useful if you need a permanent reference to a row. The penalty is a hit on performance, since every signal emission must update the references.

The `GtkTreeModel` provides an assortment of methods to acquire iterators to specific rows and navigate around with them. In addition, it has methods to convert between iterators and paths.

To create the string representation of a path, use

```
gchar *        gtk_tree_path_to_string           ( GtkTreePath *path);
```

To obtain an iterator to a given path:

```
gboolean       gtk_tree_model_get_iter           ( GtkTreeModel *tree_model,
                                                   GtkTreeIter *iter,
                                                   GtkTreePath *path);
```

To get a path at the row of a given iterator, use

```
GtkTreePath *  gtk_tree_model_get_path           ( GtkTreeModel *tree_model,
                                                   GtkTreeIter *iter);
```

The returned path must be freed with

```
void           gtk_tree_path_free                ( GtkTreePath *path);
```

when it is no longer needed. If you have a string representing a path, and you want an iterator at that row, you can use

```
gboolean       gtk_tree_model_get_iter_from_string ( GtkTreeModel *tree_model,
                                                     GtkTreeIter *iter,
                                                     const gchar *path_string);
```

and the inverse

```
gchar *        gtk_tree_model_get_string_from_iter ( GtkTreeModel *tree_model,
                                                     GtkTreeIter *iter);
```

These are just a few of the conversion functions. In addition there are functions back and forth from `GtkTreeRowReference` to paths. Consult the API documentation.

## 4.1 Navigating the Tree

The `GtkTreeModel` provides many functions for moving paths and iterators around in the tree. Functions to move a path include

```
void           gtk_tree_path_next                ( GtkTreePath *path);
gboolean       gtk_tree_path_prev                ( GtkTreePath *path);
void           gtk_tree_path_down                ( GtkTreePath *path);
gboolean       gtk_tree_path_up                  ( GtkTreePath *path);
```

The next() function moves the path to the next row in the same level; the prev() function to the preceding row, returning `FALSE` if it was the first in its level. The down() function moves the path to the first child of the current path, and up() moves it to the parent of the current path, returning `FALSE` if it is a top-level row.

There are even more functions for moving iterators around. Each returns `TRUE` on success.

```
gboolean       gtk_tree_model_get_iter_first     ( GtkTreeModel *tree_model,
                                                   GtkTreeIter *iter);
```

This gets an iterator to the first top-level node, returning `FALSE` for an empty tree. The iterator will not be changed in this case.

```
gboolean       gtk_tree_model_iter_next          ( GtkTreeModel *tree_model,
                                                   GtkTreeIter *iter);
```

This advances the iterator to the next node at the same level if it exists, returning `FALSE` if it is the last node in its level and invalidating the iterator.

```
gboolean        gtk_tree_model_iter_children      ( GtkTreeModel *tree_model,
                                                    GtkTreeIter *iter,
                                                    GtkTreeIter *parent);
```

If `parent` is not `NULL`, this makes the iterator point to the first of its child nodes if it has any, and if not, returning `FALSE` and invalidating the iterator. If `parent` is `NULL`, this is equivalent to `gtk_tree_model_get_iter_first()`.

```
gboolean        gtk_tree_model_iter_nth_child     ( GtkTreeModel *tree_model,
                                                    GtkTreeIter *iter,
                                                    GtkTreeIter *parent,
                                                    gint n);
```

This sets `iter` to be the child of `parent`, using the given index. The first index is 0. If `n` is too big, or parent has no children, `iter` is set to an invalid iterator and `FALSE` is returned. If `parent` is `NULL`, then `iter` is set to point to the nth top-level node.

```
gboolean        gtk_tree_model_iter_parent        ( GtkTreeModel *tree_model,
                                                    GtkTreeIter *iter,
                                                    GtkTreeIter *child);
```

Lastly, this makes `iter` point to its parent. If it is a top-level row, it returns `FALSE` and invalidates `iter`.

## 4.2 Using Iterators and Paths

You need to get the hang of using iterators and paths to do any serious programming with the `GtkTreeView` and its cohort of objects. In this section, we will look at a few different applications.

One simple application of these methods is to traverse all of the rows in a model, printing the data in each row to a file. For simplicity, we begin with a list store. The idea is simply to set an iterator to the first row and advance it until all rows have been printed. We will assume the same set of model columns as in Listing 1.

Listing 2: Saving a list store to a file.

```
gint save_store_to_file       ( GtkListStore* store,
                                gchar* filename)
{
    FILE*          file_ptr;
    gchar          *artist;
    gchar          *title;
    gint           on_exhibit;
    gfloat         price;
    GtkTreeIter iter;
    gboolean valid;
    gint row_count = 0;

    // Validate file access and existence
    file_ptr = fopen(filename, "w");
    if ( file_ptr == NULL ) {
        // check errno
```

```
        return -1;
    }

    valid = gtk_tree_model_get_iter_first (GTK_TREE_MODEL(store),
                                           &iter);
    while (valid) {
        gtk_tree_model_get ( GTK_TREE_MODEL(store), &iter,
                             ON_EXHIBIT,   &on_exhibit,
                             PRICE,        &price,
                             ARTIST,       &artist,
                             TITLE,        &title,
                             -1);
       fprintf(file_ptr, "%d:%f:%s:%s:",
           on_exhibit, price, artist, title);

      row_count++;
      valid = gtk_tree_model_iter_next (GTK_TREE_MODEL(store), &iter);
    }
    fclose(file_ptr);
    return row_count;
}
```

If we had to use the same technique to traverse all of the rows in a tree store, we would be forced to use a stack or write a recursive traversal function. Fortunately, the `GtkTreeModel` has a "*foreach*" method that will traverse the entire model, executing a user-supplied function at each model node. As both the `GtkListStore` and the `GtkTreeStore` implement the model, we can use the *foreach* function for either of these. Its prototype is

```
void          gtk_tree_model_foreach          ( GtkTreeModel *model,
                                                GtkTreeModelForeachFunc func,
                                                gpointer user_data);
```

The first argument is the model to be traversed, and the last argument is user-supplied data. The second argument is a `GtkTreeModelForeachFunc` function. Such a function is defined by

```
gboolean      (*GtkTreeModelForeachFunc)      ( GtkTreeModel *model,
                                                GtkTreePath *path,
                                                GtkTreeIter *iter,
                                                gpointer data);
```

When the function is called, the model will supply the path to the current row, an iterator to the row, and the user data provided to `gtk_tree_model_foreach()`. The function must return a boolean value, `TRUE` to stop successive iteration, `FALSE` to continue it. You might be using this function to search the tree, for example, and having found what you were looking for, you could stop by returning `TRUE`.

To illustrate, we will work with a tree store based on the list store above, except that there is now a new model column with number `PERIOD`. The `PERIOD` column holds string data and will contain the name of a stylistic period in the history of painting, such as "Renaissance" or "Post-Impressionist". The top-level rows will be periods, and their children will be rows with specific paintings.

Figure 2 shows our example tree store model on view. Notice that the top-level rows have no data other than the name of the period. Cell renderer functions ensure that the only displayed data in a top-level row is the period. The images are created from thumbnails, but for simplicity, the example code here omits the logic related to thumbnails. The data for this model comes from a plain text file in CSV format, with colons ':' separating the fields. A fragment of such a file might look like

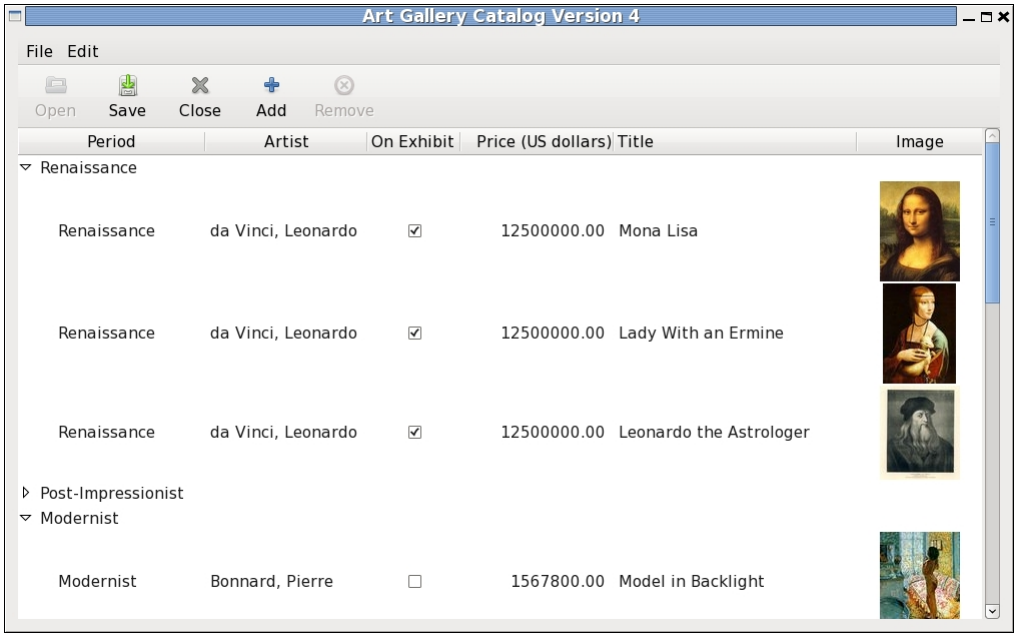


Figure 2: Treeview with a tree store in view.

```
0:0.00:Renaissance: : : :
1:12500000.00:Renaissance:da Vinci, Leonardo:Mona Lisa:
1:12500000.00:Renaissance:da Vinci, Leonardo:Lady With an Ermine:
1:12500000.00:Renaissance:da Vinci, Leonardo:Leonardo the Astrologer:
0:0.00:Post-Impressionist: : : :
0:5760000.00:Post-Impressionist:Van Gogh, Vincent:Apricot Trees In Blossom:
```

The top-level rows have a single blank character in the string fields other than the period. To create such a tree store from the data in the CSV file, we use the function create_store_from_file() in Listing 3.

Listing 3: Creating a tree store from file data.

```
gint create_store_from_file  (GtkTreeStore** store,
                              gchar* filename)
{
    FILE*          file_ptr;
    GtkTreeIter    iter, child;
    gchar          artist[MAX_ARTIST_NAME];
    gchar          title [MAX_TITLE_LENGTH];
    gchar          period[MAX_PERIOD_NAME];
    gint           on_exhibit;
    gfloat         price;

    // Validate file access and existence
    file_ptr = fopen(filename, "r");
    if ( file_ptr == NULL ) {
        return -1;
    }

    *store = gtk_tree_store_new ( NUM_COLUMNS,
                                  G_TYPE_BOOLEAN,
                                  G_TYPE_FLOAT,
```

```
                                    G_TYPE_STRING,
                                    G_TYPE_STRING,
                                    G_TYPE_STRING
                                   );

    while (fscanf(file_ptr, "%d:%f:%[^:]:%[^:]:%[^:]:",
           &on_exhibit, &price, period, artist, title) != EOF ) {


        if ( g_ascii_strcasecmp (artist, " ") == 0 &&
             g_ascii_strcasecmp (title,  " ") == 0 &&
             g_ascii_strcasecmp (period, " ") != 0 )  {

            // It is a period entry to be put at root level
            gtk_tree_store_append( *store, &iter, NULL);
            gtk_tree_store_set( *store, &iter,
                                PERIOD,       period,
                                ARTIST,       artist,
                                TITLE,        title,
                                THUMB_PATH,   full_thumb_path,
                                -1);
        }
        else {
            // It is an actual painting row
            gtk_tree_store_append (*store, &child, &iter);
            gtk_tree_store_set (*store,   &child,
                                ON_EXHIBIT,   on_exhibit,
                                PRICE,        price,
                                ARTIST,       artist,
                                TITLE,        title,
                                PERIOD,       period,
                                -1);
        }
    }
    return 0;
}
```

Notice that we create the top-level rows with

```
gtk_tree_store_append( *store, &iter, NULL);
```

but that the child rows are created with

```
gtk_tree_store_append (*store, &child, &iter);
```

The value of the iterator `iter` when it is used to create the child row is the last value that it was set to when a top-level row was created, so all rows found in the file until the next top-level row will be appended to the list of children of the preceding one.

To write the currently displayed model to a file requires that its data be written back to the file in the same format in which it was read. To do this, we define a `GtkTreeModelForeachFunc` named `print_row()` that will get the model data in the row pointed to by the iterator, and print it to the file whose `FILE*` pointer is passed to it in the user_data argument. `print_row()` is shown in Listing 4, as well as the function that would traverse the tree with the foreach method.

Listing 4: Functions to save tree to a file

```
gboolean  print_row          ( GtkTreeModel *model,
                               GtkTreePath *path,
                               GtkTreeIter *iter,
                               gpointer    user_data)
{
    gchar          *artist;
    gchar          *title;
    gchar          *period;
    gint           on_exhibit;
    gfloat         price;
    FILE           *file_ptr = (FILE*) user_data;

    gtk_tree_model_get ( model, iter,
                         ON_EXHIBIT,   &on_exhibit,
                         PRICE,        &price,
                         PERIOD,       &period,
                         ARTIST,       &artist,
                         TITLE,        &title,
                         -1);

    fprintf(file_ptr, "%d:%.2f:%s:%s:%s:\n",
            on_exhibit, price, period, artist, title);
    g_free(artist);
    g_free(title);
    g_free(period);
    return FALSE;
}

gint save_store_to_file      ( GtkTreeStore* store,
                               gchar* filename)
{
    // Validate file access and existence
    FILE*  file_ptr = fopen(filename, "w");
    if ( file_ptr == NULL ) {
        return -1;
    }
    gtk_tree_model_foreach( GTK_TREE_MODEL(store), print_row, (gpointer) file_ptr);
    fclose(file_ptr);
    return 0;
}
```

As a final example that uses iterators, suppose that we want to create a function that, given the name of a period, and the data for a new row, appends a new row to the end of the list of child rows of that period. To do this requires that we

1. find the top-level row in the model whose `PERIOD` data matches the period name we are given,
2. get a new iterator at the end of the list of children of that row, and
3. sets the data into that new row.

The following code snippet does this. Assume that `period` is the string containing the period we are looking for in the model.

```
    gtk_tree_model_get_iter_from_string (model, &iter, "0");

    /* Retrieve an iterator pointing to the selected period. */
    do {
        gtk_tree_model_get (model, &iter, PERIOD, &name, -1);
        if (g_ascii_strcasecmp (name, period) == 0)  {
            g_free (name);
            break;
        }
        g_free (name);
    } while (gtk_tree_model_iter_next (model, &iter));

    gtk_tree_store_append (GTK_TREE_STORE (model), &new_row_iter, &iter);
    gtk_tree_store_set (GTK_TREE_STORE (model), &new_row_iter,
                        ON_EXHIBIT,   on_exhibit,
                        PRICE,        price,
                        ARTIST,       artist,
                        TITLE,        title,
                        PERIOD,       period,
                        -1);
```

This sets the iterator `iter` to point to the first top-level row, and successively searches only top-level rows for those whose `PERIOD` column data matches `period`, breaking out of the loop on success. This code assumes the period is present; it does not contain code to handle the case that it is not.

# 5 GtkTreeView Signals

When a user double-clicks on a row in a tree view widget, or when a non-editable row is selected and one of the keys: Space, Shift+Space, Return or Enter is pressed, the tree view emits a "row-activated" signal. If you want the application to respond to this signal in some way, then you need to connect a callback with the prototype below to the tree view widget:

```
    void          user_function                  ( GtkTreeView       *tree_view,
                                                   GtkTreePath       *path,
                                                   GtkTreeViewColumn *column,
                                                   gpointer           user_data);
```

The second argument is a `GtkTreePath`. The third is the address of the column within which the click event occurred. The last is whatever user data you want to pass to the callback. You may or may not wish to use the column argument; it is there in case you want to do something in particular with the cell that was clicked. A simple function to illustrate the use of this callback is shown in Listing 5 below. It prints the data in the activated row to the standard output stream.

Listing 5: Handling a row-activated signal.

```
void on_row_activated       ( GtkTreeView       *treeview,
                              GtkTreePath       *path,
                              GtkTreeViewColumn *column,
                              gpointer           data)
{
    GtkTreeModel  *model;
    GtkTreeIter    iter;
    gchar         *artist;
```

```
    gchar          *title;
    gint           on_exhibit;
    gfloat         price;

    model = gtk_tree_view_get_model(treeview);
    if ( gtk_tree_model_get_iter ( model, &iter, path) ) {
        gtk_tree_model_get ( model, &iter,
                             ON_EXHIBIT,   &on_exhibit,
                             PRICE,        &price,
                             ARTIST,       &artist,
                             TITLE,        &title,
                             -1);
        g_print("Artist:\t%s\n", artist);
        g_print("Title:\t%s\n", title);
        g_print("Price:\t$%'.2f\n", price);
        if ( on_exhibit )
            g_print("Currently on exhibit.\n");
        else
            g_print("Currently not on exhibit.\n");
        g_free(artist);
        g_free(title);
    }
}
```

The `on_row_activated()` callback would be connected to the "row-activated" signal with

```
g_signal_connect (G_OBJECT (*treeview), "row-activated",
                  G_CALLBACK(on_row_activated), NULL);
```

The tree view widget emits several other signals.

# 6 Handling Selections

to be continued...

# 7 Sorting Rows

There are two different ways in which sortability can be added to a `GtkTreeView` application, one of which is much more complex than the other to implement. The simpler of the two methods is sufficient provided that there is only one view of the model at any time. If, on the other hand, the model data can be viewed by two or more views, then the simpler method will not work. In either case, the basic idea is that the user can click on the column headings to have the rows of the model sorted by the data in that column. The comparison function that will be used to decide the sorting order can be customized by the programmer.

The simpler of the two methods is to use just the `GtkTreeSortable` interface. This is an interface that tree models can implement to support sorting. Both the `GtkListStore` and the `GtkTreeStore` implement `GtkTreeSortable`. The `GtkTreeView` uses the methods defined in the `GtkTreeSortable` interface to sort the model, actually reordering the model data. Therefore, it is enough to cast a list or tree store to a

`GtkTreeSortable` and use its methods for sorting the rows. The programmer can configure the treeview so that each column has a different sort comparison function.

The problem with this simple approach is that when the view sorts the data, the model data itself is re-sorted. If the model data is to be held by two or more views, then sorting in one view would cause the other view's display to be sorted as well, which is usually antithetical to the purpose of allowing multiple views. For this reason, GTK+ provides a third tree model that implements the `GtkTreeSortable` interface, called a `GtkTreeModelSort`, which does not hold any data itself. When the treeview is sorted, the underlying model data does not get re-sorted; instead, the `GtkTreeModelSort` re-sorts its representation of the underlying data. The `GtkTreeModelSort` is created with a child model, such as a list store or a tree store, and proxies its data. It has identical column types to the child model, and the changes in the child's data are propagated. The difficulty in using this approach lies in the fact that the iterators held by the model are inconsistent with those presented by the `GtkTreeModelSort` interface, and therefore, getting at the actual data takes a bit more work.

We will begin by showing how to use the `GtkTreeSortable` interface directly, without an interposed `GtkTreeModelSort` model. Then we will introduce the `GtkTreeModelSort`.

## 7.1 Using the `GtkTreeSortable` Interface

Using the `GtkTreeSortable` interface for a simple application is pretty straightforward. Basically, you first need to decide on which columns the user should be able to sort. For each such column, you have to tell the model how to sort based on that particular column,. Then you have to tell the view that these columns can be used for sorting. The view takes care of making each such column header clickable and connecting the "clicked" signal to the comparison function that you write and attach to the model.

Let us call a column that is to be used for sorting, a *sort-column*. The procedure for enabling sorting with this interface is as follows:

1. For each sort-column in the model, create a comparison function that, for a pair of values of the sort-column's data type, returns -1, 0, or 1[1] depending on which is greater. This is called a `GtkTreeIterCompareFunc` function.
2. Assign a sort column-id to each *tree view column* that will hold a model's sort-column, using `gtk_tree_view_column_set`
3. Using the `gtk_tree_sortable_set_sort_func()` in the tree sortable interface, add the comparison function to the *model's* column.
4. Tell GTK+ what the initial sorting method will be.

These steps are now described in detail.

**1.** A `GtkTreeIterCompareFunc` function is a function with prototype

```
gint            (*GtkTreeIterCompareFunc)         ( GtkTreeModel *model,
                                                    GtkTreeIter *a,
                                                    GtkTreeIter *b,
                                                    gpointer user_data);
```

It should return a negative integer, zero, or a positive integer. Specifically, within the column in which it is run, if the cell pointed to by iterator `a` "sorts before" that of `b`, then it should return a negative number, say -1. If the data pointed to by `a` and `b` sorts equally, then it returns 0, and if the data pointed to by `a` "sorts after" `b`, then it should return a positive number, usually 1.

For example, if the data being compared are two numbers, such as the price in our catalog example, the function could be

[1] Technically it can return any negative number, 0, or any positive number.

Listing 6: Example sort comparison function

```
gint sort_by_price          ( GtkTreeModel *model,
                              GtkTreeIter  *a,
                              GtkTreeIter  *b,
                              gpointer      userdata)
{
    gfloat  price1, price2;

    gtk_tree_model_get(model, a, PRICE, &price1, -1);
    gtk_tree_model_get(model, b, PRICE, &price2, -1);
    return_value = price1 - price2);
}
```

It is required that the function defines a partial order so that the `GtkTreeSortable` behaves as expected. A partial order is reflexive, antisymmetric and transitive. A relation $R$ has these three properties if for any $x$, $xRx$, and any $x$ and $y$, if $xRy$ and $yRx$ then $x = y$, and for any $x$, $y$, and $z$, if $xRy$ and $yRz$ then $xRz$. An example of an antisymmetric relation is the $\leq$relation.

If the data are strings, such as titles, and you want to compare them using the user's current locale settings, you could use a function such as

```
gint sort_by_title          ( GtkTreeModel *model,
                              GtkTreeIter  *a,
                              GtkTreeIter  *b,
                              gpointer      userdata)
{
    gchar *first, *second;
    gtk_tree_model_get(model, a, ARTIST, &first, -1);
    gtk_tree_model_get(model, b, ARTIST, &second, -1);

    gint  return_value = g_utf8_collate(first,second);
    g_free(first);
    g_free(second);
    return return_value;
}
```

**2.** Next, add to each column a logical sort_id to be used for sorting that column. If you want, you can create a separate set of logical sort-ids for the columns, using an enumeration such as

```
typedef enum
{
    SORT_BY_PRICE,
    SORT_BY_ARTIST,
    SORT_BY_TITLE,
    SORT_BY_PERIOD,
    NUM_SORT_IDS
} sort_id_type;
```

This might make the code more readable. Then, when adding the columns to the `GtkTreeView` widget, you can assign the logical sort ids to each column as the column is created and appended to the view, using

```
void        gtk_tree_view_column_set_sort_column_id ( GtkTreeViewColumn *tree_column,
                                                      gint sort_column_id);
```

This is given the column and the logical sort id to be assigned to that column. Not only does this associate the logical sort id to the column, but it makes that column clickable as well, so that the user can sort by clicking in the column header. For example, to make the `TITLE` column sortable by id `SORT_BY_TITLE`, you would write

```
gtk_tree_view_column_set_sort_column_id( column, SORT_BY_TITLE);
```

You can also just use the actual model column numbers, which will simplify the number of different symbols to keep track of. In this case, you would just issue the instruction

```
gtk_tree_view_column_set_sort_column_id( column, TITLE);
```

Using the latter approach, we would set up the `TITLE` column as follows:

```
renderer = gtk_cell_renderer_text_new ();
g_object_set (renderer,"xalign", 0.0, NULL);
column = gtk_tree_view_column_new_with_attributes
                         ("Title", renderer, "text", TITLE, NULL);
gtk_tree_view_column_set_alignment(column, 0.0);
// Make the logical sort id for this column the one defined by TITLE id
gtk_tree_view_column_set_sort_column_id( column, TITLE);
gtk_tree_view_append_column (GTK_TREE_VIEW (*treeview), column);
```

**3.** For each column that you enable sorting on, you must then set the sorting function in the corresponding column in the model to be the one you want called when that column's header is clicked. The function that associates the `GtkTreeIterCompareFunc` function to the model column is

```
void           gtk_tree_sortable_set_sort_func   ( GtkTreeSortable *sortable,
                                                   gint sort_column_id,
                                                   GtkTreeIterCompareFunc sort_func,
                                                   gpointer user_data,
                                                   GDestroyNotify destroy);
```

This is given the model, cast to a `GtkTreeSortable`, the logical sort id of the column to be sorted, the name of the function, user data, or `NULL`, and an optional function to be called to destroy memory allocated to the user data passed to the function, or `NULL`.

To illustrate, suppose that `store` is the name of our `GtkTreeStore` model. If we want the `sort_by_price()` function to be called when the `PRICE` column is clicked, and the `sort_by_title()` function to be called when the `TITLE` column is clicked, we would cast store to a `GtkTreeSortable` and call the function as follows. We assume that the model column value is the same as the logical sort id that the treeview column uses for sorting:

```
sortable = GTK_TREE_SORTABLE(*store);
gtk_tree_sortable_set_sort_func(sortable, PRICE, sort_iter_compare_func,
                                NULL, NULL);
gtk_tree_sortable_set_sort_func(sortable, TITLE, sort_iter_compare_func,
                                NULL, NULL);
```

assuming that we do not need to pass any user data to the functions.

**4.** Having set up the columns, we need to pick a column by which to sort the data initially. The `GtkTreeSortable` function

```
void          gtk_tree_sortable_set_sort_column_id  ( GtkTreeSortable *sortable,
                                                      gint sort_column_id,
                                                      GtkSortType order);
```

sets the current sort column to be `sort_column_id`. The sortable will sort itself automatically when this call is made. The `order` argument can be one of `GTK_SORT_ASCENDING` or `GTK_SORT_DESCENDING`. Also, if you do not want the rows to be sorted at all, then instead of a sort column id, pass `GTK_TREE_SORTABLE_UNSORTED_SORT_COLUMN_ID` as the second argument.

If you pass `GTK_TREE_SORTABLE_DEFAULT_SORT_COLUMN_ID` and you have set a default sort function with `gtk_tree_sortable_set_default_sort_func()`, that function will be used instead.

To make the `TITLE` column the initial sort criterion, we would call

```
gtk_tree_sortable_set_sort_column_id(sortable, TITLE, GTK_SORT_ASCENDING);
```

These are the basics, and all you need to do to set up a fairly simple application. The `GtkTreeSortable` interface takes care of the details, emitting the required signals when headers are clicked, making indicators visible and invisible when the sorting column changes, reversing their directions and toggling the sort direction, and sorting everything for you as required. It is a pretty powerful interface.

## 7.2 Using the GtkTreeModelSort Interface

The problem with the preceding method of sorting is that the data itself gets rearranged each time that the column headers are clicked. This may be fine in most cases, but it will not work when multiple views need to share a model. The `GtkTreeModelSort` model implements `GtkTreeSortable` but does not hold any data. It acts like a proxy.

It may help to think of the model as a set of pointers to the rows of the underlying model. If you have ever written an indirect sort of an array, in which you do not sort the array, but instead sort an array of pointers or indices into the original array, then your understanding of the `GtkTreeModelSort` will be made clearer if you think of it as that array of pointers. of course it is far more complex than this, but it is the basic idea.

Indirect sorting of an array works as follows. Suppose A is an array of 6 unequal strings:

| pear | apple | kiwi | mango | banana | lemon |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

Let P be an array of 6 non-negative integers, such that initially, P[i] == i for all i:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

In an indirect sort, the array P is permuted so that, for all k, $0 \leq k < 5$, A[P[k]] < A[P[k+1]]:

| 1 | 4 | 2 | 5 | 3 | 0 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

The data in the array A is unsorted, but using the array P, it can be presented in sorted order. This is a useful sort when the data should not be rearranged, or when doing so is a very costly operation because of its size.

`GtkTreeModelSort` uses this principle. The actual model containing the data to be sorted is embedded in the `GtkTreeModelSort`, which we will now call the *sort model*. The embedded model is called the *child*

*model*. The sort model maintains a set of iterators that point to its own rows, called the *parent*, or *sort iterators*. The child model has its own iterators, which we call the *child iterators*.

To use this method, the sort model is added to the tree view, not the child model. The child model is added to the sort model. When the treeview column is clicked, the treeview uses the corresponding column of the sort model to re-sort the data. The sort model's rows are "sorted", not the child model's rows. The `GtkTreeModelSort` contains methods to convert between child and parent iterators and child and parent paths.

To summarize, given that you have already made the model sortable by using the functions in the GtkTreeSortable interface, to interpose the `GtkTreeModelSort` model between the view and the actual data store, you need to make the following changes:

### Setting Up the Models

Instead of adding the store that contains the actual data to the treeview, you add it to the sort model and add the sort model to the treeview. The function

```
GtkTreeModel * gtk_tree_model_sort_new_with_model( GtkTreeModel *child_model);
```

is given the child model, i.e., the store to be added to the sort model, and adds it to a newly created sort model, returning a pointer to that new model. A sample code sequence could be:

```
// Create the GtkTreeView, passing in application state data
setup_tree_view (treeview, appState);

if ( create_store_from_file(store, fileName ) == -1 ) {
    g_print("Could not create tree store from input file\n");
    exit(1);
}
sort_model = gtk_tree_model_sort_new_with_model( GTK_TREE_MODEL (store));
gtk_tree_view_set_model (GTK_TREE_VIEW (treeview),GTK_TREE_MODEL (sort_model));
```

assuming the the functions `setup_treeview()` and `create_store_from_file()` do what their names suggest.

### Modifying Accesses to Data

In every part of the code where the store's data would be accessed, the code must be modified so that the child model is first obtained from the sort model, and the sort model iterators are converted to child iterators.

### Examples

To remove the row that is currently selected.

```
sort_model = gtk_tree_view_get_model (GTK_TREE_VIEW(appState->treeview));

/* Get the underlying child model out of the sort model */
child_model = gtk_tree_model_sort_get_model(  GTK_TREE_MODEL_SORT (sort_model));

tree_selection = gtk_tree_view_get_selection( GTK_TREE_VIEW (treeview));

if ( gtk_tree_selection_get_selected (tree_selection, &sort_model, &sort_iter) ) {
```

```
    /* In order to remove any real data, we have to get the iterator to
       the row in the underlying child model. The nect function does this.
    */
    gtk_tree_model_sort_convert_iter_to_child_iter(
                            GTK_TREE_MODEL_SORT (sort_model),
                            &child_iter, &sort_iter);
    gtk_tree_store_remove( GTK_TREE_STORE (child_model), &child_iter);
}
```

To insert a new row of data at the end of the list of children of the first row of the store, given the data already exists:

```
sort_model  = gtk_tree_view_get_model  ( GTK_TREE_VIEW(treeview));
child_model = gtk_tree_model_sort_get_model( GTK_TREE_MODEL_SORT (sort_model));

gtk_tree_model_get_iter_from_string (child_model, &iter, "0");
gtk_tree_store_append (GTK_TREE_STORE (child_model), &new_row_iter, &iter);
        gtk_tree_store_set (GTK_TREE_STORE (child_model), &new_row_iter,
                            PRICE,          price,
                            ARTIST,         artist,
                            TITLE,          title,
                           -1);
```

To illustrate how to modify the sort comparison functions to access the data in the child model, we modify the one from Listing 6:

```
gint sort_by_price          ( GtkTreeModel *sort_model,
                              GtkTreeIter  *a,
                              GtkTreeIter  *b,
                              gpointer      userdata)
{
    gfloat           price1, price2;
    GtkTreeModel    *child_model;
    GtkTreeIter      child_a;
    GtkTreeIter      child_b;

    /* Get the underlying child model out of the sort model */
    child_model = gtk_tree_model_sort_get_model(
                            GTK_TREE_MODEL_SORT (sort_model));

    gtk_tree_model_sort_convert_iter_to_child_iter(
                            GTK_TREE_MODEL_SORT (sort_model),
                            &child_a, a);
    gtk_tree_model_sort_convert_iter_to_child_iter(
                            GTK_TREE_MODEL_SORT (sort_model),
                            &child_b, b);
    gtk_tree_model_get(child_model, &child_a, PRICE, &price1, -1);
    gtk_tree_model_get(child_model, &child_b, PRICE, &price2, -1);
    return_value = price1 - price2);
}
```