September 27, 2009

`tree.hh` documentation

Kasper Peeters

`kasper.peeters@gmail.com`

Abstract

The `tree.hh` library for C++ provides an STL-like container class for n-ary trees, templated over the data stored at the nodes. Various types of iterators are provided (post-order, pre-order, and others). Where possible the access methods are compatible with the STL or alternative algorithms are available. The library is available under the terms of the GNU General Public License.

Code and examples available at: `http://tree.phi-sci.com/`

This documentation is not yet entirely complete. Refer to the `tree.hh` header file for a full list of member functions.

Table of Contents

1 Overview

1.1 The container class

The tree class of `tree.hh` is a templated container class in the spirit of the STL. It organises data in the form of a so-called n-ary tree. This is a tree in which every node is connected to an arbitrary number of child nodes. Nodes at the same level of the tree are called "siblings", while nodes that are below a given node are called its "children". At the top of the tree, there is a set of nodes which are characterised by the fact that they do not have any parents. The collection of these nodes is called the "head" of the tree. See figure 1 for a pictorial illustration of this structure (90 degrees rotated for convenience).

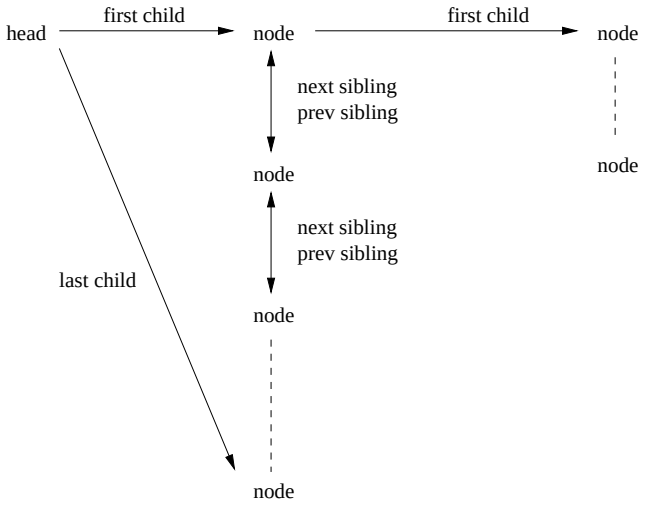
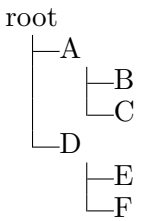


Figure 1: Overview of the tree structure. The elements at the top of the tree (here displayed at the left for convenience) are in the "head" (there can be more than one such element). Every node is linked to its children using the "first child" and "last child" links. In addition, all nodes on a given level are doubly-linked using the "previous sibling" and "next sibling" links. The "depth" of a given node refers to the horizontal distance from the head nodes.

The tree class is templated over the data objects stored at the nodes; just like you can have a `vector<string>` you can now have a `tree<string>`. Many STL algorithms work on this data structure, and where necessary alternatives have been provided.

1.2 Iterators

The essential difference between a container with the structure of a tree and the STL containers is that the latter are "linear". While the STL containers thus only have essentially one way in which one can iterate over their elements, this is not true for trees. The `tree.hh` library provides (at present) four different iteration schemes. To describe them, consider the following tree:



The three iteration types and the resulting order in which nodes are visited are tabulated below:

pre-order (default)	"element before children"	`pre_order_iterator`	root A B C D E F
post-order	"element after children"	`post_order_iterator`	B C A E F D root
breadth-first		`breadth_first_iterator`	root A D B C E F
sibling	"only siblings"	`sibling_iterator`	(for ex.) A D
fixed-depth		`fixed_depth_iterator`	(for ex.) A D
leaf		`leaf_iterator`	B C E F

The pre-order ones are the default iterators, and therefore also known under the name of `iterator`. Sibling iterators and fixed-depth iterators iterate only over the nodes at a given depth of the tree. The former restrict themselves to the child nodes of one given node, while the latter iterates over all child nodes at the given depth. Finally, leaf iterators iterate over all leafs (bottom-most) nodes of the tree.

There are copy constructors that will convert iterators of the various types into each other. The post- and pre-order iterators are both also known as "depth-first", in contrast to the "breadth-first" iterator.

The begin and end iterators of a tree can be obtained using `begin()` and `end()` (for pre-order iterators) or alternatively `begin_post()` and `end_post()` (for post-order iterators) and `begin_leaf()` and `end_leaf()` (for leaf iterators). Similarly, the begin and end sibling iterators can be obtained by calling `begin(iterator)` and `end(iterator)`. The range of children of a given node can also be obtained directly from an iterator, by using the `iterator::begin()` and `iterator::end()` member functions.

If you want to (temporarily) make an iterator not go into the child subtree, call the member function `skip_children`. This will only keep effect for a single increment or decrement of the iterator. Finally, whether or not an iterator is actually pointing at a node (i.e. is not an "end" iterator) can be tested using the `is_valid(iterator)` member of the tree class.

2 Basic operations

Initialising There are two nontrivial constructors. One which takes a single node element as argument. It constructs a tree with this node begin the sole node in the head (in other words, it is a combination of a trivial constructor together with a `set_head` call). The other non-trivial constructor takes an iterator, and copies the subtree starting at that node into the newly created tree (useful for constructing new tree objects given by subtrees of existing trees).

Tree traversal Besides the `operator++` and `operator--` members for step-wise traversal through the tree, it is also possible to use the `operator+=` and `operator-=` member functions to make more than one step at the same time (though these are linear time, not amortized constant). The result of stepping beyond the end of the tree or stepping beyond the end of a sibling range (for sibling iterators) is undefined.

The parent of a given node can be reached by calling the `parent` member of the tree object, giving it an iterator pointing to the node.

If you know the number of children of a given node, you can get direct access to the nth child by using the `child` member function. Note that the value of the index is not checked and should therefore always be valid.

Appending child nodes Nodes can be added as children of a given node using the `append_child` member function.

Inserting nodes Nodes can be inserted at the same depth as a given other node using the `insert` and `insert_after` members functions. This is also how you insert the first node into a tree.

3 Other algorithms

3.1 Non-mutating algorithms

Counting nodes The total number of nodes of a tree can be obtained using the `size` member function, while the number of children of a given node can be obtained with a call to `number_of_children(iterator)`. Similarly, the number of nodes at a given depth (the number of siblings of a given node) can be obtained using the `number_of_siblings` member function.

Determining depth The `depth()` member function returns the distance of a node to the root.

Accessing siblings by their index See the next item.

Determining index in a sibling range In order to determine the index of a node in the range of siblings to which it belongs, use the `index(sibling_iterator)` member function. The first sibling node has index 0. The reverse of this function (obtaining a sibling node given its index in the range of siblings) is called `child(const iterator_base&, unsigned int)`.

Comparing trees While the STL `equal` algorithm can be used to compare the values of the nodes in two different trees, it does not know about the structure of the tree. If you want the comparison to take this into account, use the `equal(iterator, iterator, iterator, BinaryPredicate)` call of the tree class. As an addition to the STL algorithm, the length of the first range does not have to be equal to the length of the range pointed to by the second iterator.

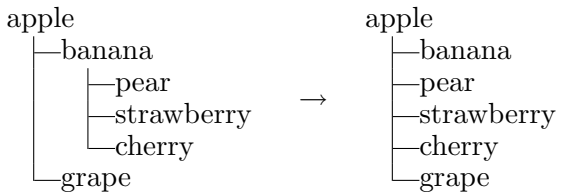
There is also an `equal_subtree` algorithm which takes only two iterators, pointing to the (single-node) heads of two subtrees.

3.2 Mutating algorithms

Erasing nodes and subtrees In order to remove a node including its children from the tree, use the `erase(iterator)` call. If you just want to erase the children, but not the node itself, use the `erase_children(iterator)` call.

Replacing individual nodes or subtrees

Flattening subtrees The procedure of moving all children of a given node to be siblings of that node is called "flattening"; it acts as



when the tree is flattened at the "banana" node.

Moving or exchanging subtrees Simple exchange of one sibling node with the next one is done through the member function `swap(sibling_iterator)`. The iterator remains valid and remains pointing to the moved subtree.

More complicated move operations are the `move_ontop`, `move_before` and `move_after` ones. These all take two iterators, a source and a target. The member `move_ontop(target, source)` removes the 'target' node and all its children, and replaces it with the 'source' node and its children. The 'source' subtree is removed from its original location. The other two move members do a similar thing, differing only in the node which is to be replaced.

Extracting subtrees You can create a new tree object filled with the data of a subtree of the original tree. This is analogous to the extraction of a substring of a string. The relevant member function is `subtree(sibling_iterator, sibling_iterator)` which takes a range of siblings as argument. There is also a slight variation of this member, which does not return a tree object but instead populates one that is passed as an argument (useful if you want to call this on a tree object subclassed from `tree<T>`.

Sorting The standard STL sort algorithm is not very useful for trees, because it only exchanges values, not nodes. Applying it to a tree would mean that the structure of the tree remains unmodified, only node values get moved around (not their subtrees).

Therefore, the `tree` class has its own sort member. It comes in two forms, just like the STL sort, namely

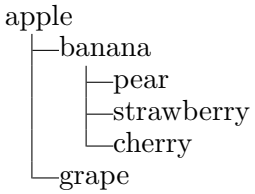
```
void     sort(sibling_iterator from, sibling_iterator to, bool deep=false);

template<class StrictWeakOrdering>
void     sort(sibling_iterator from, sibling_iterator to,
              StrictWeakOrdering comp, bool deep=false);
```

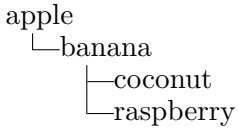
The result of a call to either of these is that the nodes in the range described by the two iterators get sorted. If the boolean `deep` is true, the subtrees of all these nodes will get sorted as well (and so one can sort the entire tree in one call). As in the STL, you can use the second form of this function to pass your own comparison class.

If the nodes to which the two iterators point are not in the same sibling range (i.e. not at the same depth in the tree), the result is undefined.

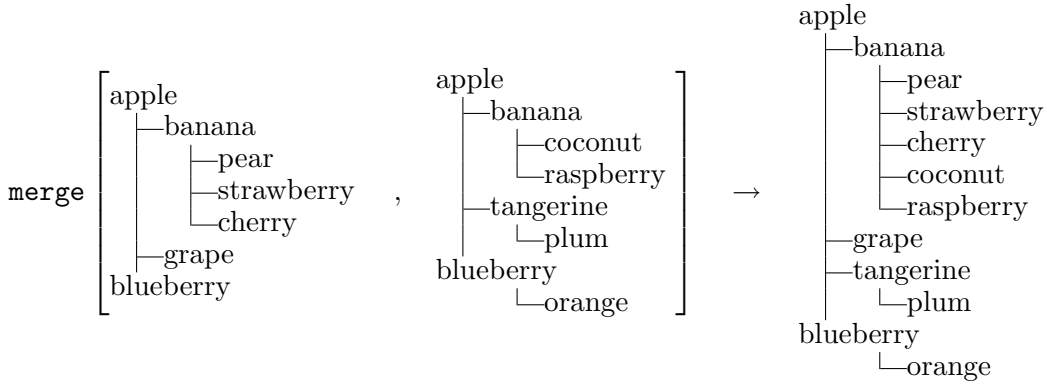
Merging One way in which one might think of indicating the position where new nodes are to be inserted, is to give the path that leads to the insertion point. For instance, given the tree



one could imagine using the sub-tree



to indicate that the nodes "coconut" and "raspberry" are to be inserted as new children of the "banana" node. In `tree.hh` this process is called *tree merging*. It can do the simple addition of children as above, but actually handles the generic case too: as an example consider the merge



As is clear from the above, the arguments to `merge` are two sibling ranges.

Index