C++ AVL Tree Template

Version 1.6

1 Introduction

Note: full formatting cleanup of this document pending.

This document explains how to use the avl_tree template. Adelson-Velskii and Landis Balanced Binary Search Trees (or AVL Trees) are described in the Wikipedia , and in many good textbooks on fundamental data structures. I have also written a C language version of this data structure.

To avoid possible confusion about the terms that I use in this document (and in the source comments), here is a summary description of AVL Trees. An AVL Tree is a set of *nodes* (or *elements*). Each node is associated with a unique *key* value. The key values can be ordered from least to greatest. Each node in the tree may (or may not) have a *less child* node, and it may (or may not) have a *greater child* node. If node A is a child of node B, then B is the *parent* of A. If A is the less child of B, A's key must be less than B's key. Similarly, if A is the greater child of B, A's key must be greater than B's key. (Because of the way that binary search trees are typically diagrammed, the less child is commonly called the left child, and the greater child is commonly called the right child.) All nodes in a tree have exactly one parent, except for the *root* node, which has no parent. Node A is a *descendant* of node C if C is A's parent, or if A's parent is a descendant of C. If a node is not the root of the entire tree, it is the root of a *subtree* consisting of the node and all its descendants. The *less subtree* of a node is the subtree whose root is the less child of the node. The *greater subtree* of a node is the subtree whose root is the greater child of the node. The *depth* of a node is one more than the depth of its parent. The depth of the root node is 1. The depth of a tree is the maximum node depth. The *balance factor* of a node is the depth of its greater subtree minus the depth of its less subtree, with non-existent subtrees being considered to have a depth of 0. In an AVL tree, the balance factor of any node can only be -1, 0 or 1.

There are many open-source C and C++ implementations of AVL Trees available. But as far as I know, this is the only one that manipulates the nodes of the tree using abstract "handles" instead of concrete pointers. If all the nodes are in a single array, you can use node indexes as handles instead of node pointers. This approach makes it possible to compress the size of the nodes if memory is tight. Using indexes as handles (instead of pointers) can make tree persistence as simple as writing the node array out with a single disk write, and reading it back in with a single disk read. The template also allows for a tree to be in secondary storage, with nodes being "paged" in and out of memory.

To achieve the desired level of abstraction, the avl_tree template uses lots of short inline functions. Because of this, function inlining can significantly improve performance when using the template. If the test suite. test_avl.cpp, is compiled with GNU GCC using level 1 optimization (-O option), it executes twice as fast as when the test suite is compiled without optimization (the default).

The template code makes no use of recursion. The implementation is stack-friendly in general, except perhaps for the iter class. Instances of iter contain an array of handles whose dimension is the maximum tree depth minus one.

Since key comparisons can potentially be complex, the code avoids repeated comparisons of the same pair of node key values.

To avoid clutter, default destructor functions are not documented.

2 Source Files

The source code for the template is in the header file avl_tree.h

A test suite for the template is in the file test_avl.cpp.

avl_ex1.cpp shows an example instantiation of the template using pointers as handles.

avl_ex2.cpp shows an example instantiation of the template using array indexes as handles.

All of this code compiles with a contemporary version of GNU GCC, and with Visual C++ .NET.

3 Reference Classes

To help describe the constraints on template class/typename parameters, or on member types of template class parameters, I like to use *reference classes*. This doesn't necessary mean that the type being constrained has to use the reference class as its definition. It is only necessary that every possible usage of the reference class or one of its instances is also a possible usage of the constrained type or one of its instances. When an identifier with the prefix ANY_ is used, this means that all occurrences of that identifier should be substituted with the same type (or with types that implicitly convert to the substituted type). Take for example the function template:

```
template <class A>
void foo(A &a) { a.x(a.y()); }
```

The reference class for the parameter A would be:

```
class A
  {
  public:
    void x(ANY_px p);
    ANY_px y(void);
  };
```

The following class could be passed as the class A parameter to the template:

```
struct someA
  {
  public:
    static double x(int aintp);
    signed char y(bool f = true) const;
  };
```

Since the return type of x() is void in the reference class, it can return any type (or be void) in the actual parameter class. y() can return signed char because signed char implicitly converts to int . Member functions can be made static or const because these make the usage of a function more, not less, flexible.

4 Namespace

The avl_tree template is in the abstract_container namespace. The AVL Tree header file also defines this enumerated type:

```
enum search_type
  {
    EQUAL = 1,
    LESS = 2,
    GREATER = 4,
    LESS_EQUAL = EQUAL | LESS,
    GREATER_EQUAL = EQUAL | GREATER
  };
```

in the abstract_container namespace.

5 Template Parameters

The avl_tree template begins with:

template <class abstractor, unsigned max_depth = 32>
class avl_tree …

5.1 Members of Reference Class for `abstractor` Template Parameter

All members of the reference class are public.

5.1.1 Type `handle`

Each node has to be associated with a node handle, which is a unique value of the `handle` type. Here is the reference class for `handle`:

```
class handle
  {
  public:
    // No default value for handles is assumed by the template.
    handle(void);
    handle(handle &h);
    void operator = (handle &h);
    bool operator == (handle &h);
  };
```

5.1.2 Type `key`

Each node has to be associated with a key, which is a unique value of the `key` type. The difference between a key and a handle is that a node can be conveniently "looked up" using its handle, but it can't be conveniently looked up using its key. In fact, the whole point of this template is to make it convenient to look up a node given its key. Here is the reference class for key:

```
class key
  {
  public:

    // Only have to copy it.
    key(key &k);

  };
```

5.1.3 Type `size`

The type `size` is an integral type. It must be large enough the hold the maximum possible number of nodes in the tree.

5.1.4 Functions `get_less`, `get_greater`

```
handle get_less(handle h, bool access = true);
```

```
handle get_greater(handle h, bool access = true);
```

Return the handle of the less/greater child of the node whose handle is h. If access is true, and the child node is in secondary storage, it has to be read into memory. If access is false, the child node does not have to be read into memory. Ignore the access parameter in your implementations of get_less and get_greater if your instantiation makes no use of secondary storage.

5.1.5 Functions `set_less`, `set_greater`

```
void set_less(handle h, handle lh);
void set_greater(handle h, handle gh);
```

Given the handle h of a node, set the handle of the less/greater child of the node.

FORMATTING CLEANUP DONE TO HERE

5.1.6 Function get_balance_factor

int get_balance_factor(handle h);

Return the balance factor of the node whose handle is h.

5.1.7 Function set_balance_factor

void set_balance_factor(handle h, int bf);

Set the balance factor of the node whose handle is h. The only possible balance factor values are -1, 0 and 1.

5.1.8 Function compare_key_node

int compare_key_node(key k, handle h);

Compares a key with the key of a node. Returns a negative value if the key is less than the node’s key. Returns zero if the key is the same as the node’s key. Returns a positive value if the key is greater than the node’s key.

5.1.9 Function compare_node_node

int compare_node_node(handle h1, handle h2);

Compares the keys of two nodes. Returns a negative value if the first node’s key is less than the second node’s key Returns zero if the first node’s key is the same as the second node’s key. Returns a positive value if the first node’s key is greater than the second node’s key.

5.1.10 Function null

handle null(void);

Always returns the same, invalid handle value, which is called the *null value*.

5.1.11 Function read_error

bool read_error(void);

Returns true if there was an error reading secondary storage If your instantiation of the template makes no use of secondary storage, use this definition:

bool read_error(void) { return(false); }

5.1.12 **Parameterless** constructor

abstractor(void);

5.2 max_depth

This is the maximum tree depth for an instance of the instantiated class You almost certainly want to choose the maximum depth based on the maximum number of nodes that could possibly be in the tree instance at any given time. To do this, let the maximum depth be M such that:

MN(M)   <=   maximum number of nodes <   MN(M + 1)

where MN(d) means the minimum number of nodes in an AVL Tree of depth d. Here is a table of MN(d) values for d from 2 to 45.

d	MN(d)
2	2
3	4
4	7
5	12
6	20
7	33
8	54
9	88
10	143
11	232
12	376
13	609
14	986
15	1,596
16	2,583
17	4,180
18	6,764
19	10,945
20	17,710
21	28,656
22	46,367
23	75,024
24	121,392
25	196,417
26	317,810

27	514,228
28	832,039
29	1,346,268
30	2,178,308
31	3,524,577
32	5,702,886
33	9,227,464
34	14,930,351
35	24,157,816
36	39,088,168
37	63,245,985
38	102,334,154
39	165,580,140
40	267,914,295
41	433,494,436
42	701,408,732
43	1,134,903,169
44	1,836,311,902
45	2,971,215,072

(In general, MN(d) = MN(d – 1) + MN(d – 2) + 1 .)

If, in a particular instantiation, the maximum number of nodes in a tree instance is 1,000,000, the maximum depth should be 28. You pick 28 because MN(28) is 832,039, which is less than or equal to 1,000,000, and MN(29) is 1,346,268, which is strictly greater than 1,000,000.

If you insert a node that would cause the tree to grow to a depth greater than the maximum you gave, the results are undefined.

Each increase of 1 in the value of max_depth increases the size of an instance of the iter class (see definition below) by sizeof(handle). The only other use of max_depth is as the size of bit arrays used at various places in the code. Generally, the number of bytes in a bit array is the size rounded up to a multiple of the number of bits in an int, and divided by the number of bits in a byte. All this is a roundabout way of saying that, if you don't use iter instances, you can guiltlessly add a big safety margin to the value of max_depth.

6 **Public** Members

6.1 Type handle

Same as handle type member of the abstractor parameter class.

6.2 Type key

Same as key type member of the abstractor parameter class.

6.3 Type size

Same as size type member of the abstractor parameter class.

6.4 *Function* insert

handle insert(handle h);

Insert the node with the given handle into the tree. The node must be associated with a key value. The initial values of the node's less/greater child handles and its balance factor are don't-cares. If successful, this function returns the handle of the inserted node. If the node to insert has the same key value as a node that's already in the tree, the insertion is not performed, and the handle of the node already in the tree is returned. Returns the null value if there is an error reading secondary storage. Calling this function invalidates all currently-existing instances of the iter class (that are iterating over this tree).

6.5 *Function* search

handle search(key k, search_type st = EQUAL);

Searches for a particular node in the tree, returning its handle if the node is found, and the null value if the node is not found. The node to search for depends on the value of the st parameter.

Value of st	Node to search for
EQUAL	Node whose key is equal to the key k.
LESS	Node whose key is the maximum of the keys of all the nodes with keys less than the key k.
GREATER	Node whose key is the minimum of the keys of all the nodes with keys greater than the key k.
LESS_EQUAL	Node whose key is the maximum of the keys of all the nodes with keys less than or equal to the key k.
GREATER_EQUAL	Node whose key is the minimum of the keys of all the nodes with keys greater than or equal to the key k.

6.6 *Function* search_least

handle search_least(void);

Returns the handle of the node whose key is the minimum of the keys of all the nodes in the tree. Returns the null value if the tree is empty or an error occurs reading from secondary storage.

6.7 *Function* search_greatest

handle search_greatest(void);

Returns the handle of the node whose key is the maximum of the keys of all the nodes in the tree. Returns the null value if the tree is empty or an error occurs reading from secondary storage.

6.8 *Function* remove

handle remove(key k);

Removes the node with the given k from the tree. Returns the handle of the node removed. Returns the null value if there is no node in the tree with the given key, or an error occurs reading from secondary storage. Calling this function invalidates all currently-existing instances of the iter class (that are iterating over this tree).

6.9 Function purge

void purge(void);

Removes all nodes from the tree, making it empty.

6.10 Function is_empty

bool is_empty(void);

Returns true if the tree is empty.

6.11 Function read_error

void read_error(void);

Returns true if an error occurred while reading a node of the tree from secondary storage. When a read error has occurred, the tree is in an undefined state.

6.12 Parameterless Constructor

avl_tree(void);

Initializes the tree to the empty state.

6.13 Function Template build

```
template<typename fwd_iter>
bool build(fwd_iter p, size num_nodes);
```

Builds a tree from an sequence of nodes that are sorted ascendingly by their key values. The number of nodes in the sequence is given by num_nodes. p is a forward iterator that initially refers to the first node in the sequence. Here is the reference class for the fwd_iter:

```
class fwd_iter
  {
  public:
    fwd_iter(fwd_iter &);
    handle operator * (void);
    void operator ++ (int);
  };
```

Any nodes in the tree (prior to calling this function) are purged. The iterator will be incremented one last time when it refers to the last node in the sequence. build() returns false if a read error occurs while trying to build the tree. The time complexity of this function is O(n x log n), but it is more efficient than inserting the nodes in the sequence one at a time, and the resulting tree will generally have better balance.

6.14 Function subst

handle subst(handle new_node);

If the node whose handle is passed as new_node has the same key as a node that is already in the tree, then the node already in the tree is removed from the tree and replaced by the new node. If there is no node in the tree with the same key as the new node, no substitution is made, and the null value is returned. If a substitution is made, the handle of the node that was removed from the tree is returned. The null value is returned if a read error occurs. Calling this function invalidates all currently-existing iter instances (that are iterating over this tree).

6.15 *Copy* Constructor and Assignment Operator?

If the abstractor class has a public copy constructor and public assignment operator, the avl_tree instantiation will have a (default) copy constructor and assignment operator.

6.16 *Class* iter

Instances of this member class are bi-directional iterators over the ascendingly sorted (by key) sequence of nodes in a tree. The subsections of this section describe the public members of iter.

A useful property of this data structure is the ability to do a final, destroying iteration over the nodes. Heap memory or any other resources allocated to a node can be freed after the iterator has stepped past the node. See the clear function in example 1 to see how this is done.

6.16.1 Parameterless constructor

iter(void);

Initializes the iterator to the null state.

6.16.2 Function start_iter

void start_iter(avl_tree &tree, key k, search_type st = EQUAL);

Causes the iterator to refer to a particular node in the tree that is specified as the first parameter. If the particular node cannot be found in the tree, or if a read error occurs, the iterator is put into the null state. The particular node to refer to is determined by the st parameter.

Value of st	Node to refer to
EQUAL	Node whose key is equal to the key k.
LESS	Node whose key is the maximum of the keys of all the nodes with keys less than the key k.
GREATER	Node whose key is the minimum of the keys of all the nodes with keys greater than the key k.
LESS_EQUAL	Node whose key is the maximum of the keys of all the nodes with keys less than or equal to the key k.
GREATER_EQUAL	Node whose key is the minimum of the keys of all the nodes with keys greater than or equal to the key k.

6.16.3 Function start_iter_least

void start_iter_least(avl_tree &tree);

Cause the iterator to refer to the node with the minimum key in the given tree. Puts the iterator into the null state if the tree is empty or a read error occurs.

6.16.4 Function start_iter_greatest

void start_iter_greatest(avl_tree &tree);

Cause the iterator to refer to the node with the maximum key in the given tree. Puts the iterator into the null state if the tree is empty or a read error occurs.

6.16.5 Operator *

handle operator * (void);

Returns the handle of the node that the iterator refers to. Returns the null value if the iterator is in the null state.

6.16.6 Prefix and Postfix Operator ++

void operator ++ (void);
void operator ++ (int);

Causes the iterator to refer to the node whose key is the next highest after the key of the node the iterator currently refers to. Puts the iterator into the null state if the key of the node currently referred to is the maximum of the keys of all the nodes in the tree, or if a read error occurs. Has no effect if the iterator is already in the null state.

6.16.7 Prefix and Postfix Operator --

void operator -- (void);
void operator -- (int);

Causes the iterator to refer to the node whose key is the next lowest after the key of the node the iterator currently refers to. Puts the iterator into the null state if the key of the node currently referred to is the minimum of the keys of all the nodes in the tree, or if a read error occurs. Has no effect if the iterator is already in the null state.

6.16.8 Function read_error

Returns true if a read error occurred.

6.16.9 Default Copy Constructor and Assignment Operator

These member functions exist and can be safely used.

7 Protected Members

7.1 Variable abs

```
struct abs_plus_root : public abstractor
  {
    handle root;
  };
```

abs_plus_root abs;

abs is an instance of the abstractor template class parameter, and also contains the handle of the root node of the AVL tree (or the null value if the tree is empty).

The sole purpose of combining the abstractor instance and the root handle into a single structure is to take advantage of the so-called “empty member” optimization. If a class has no data members or base classes, the ISO C++ Standard allows instances of the class to have zero size if and only if they are sub-instances of an instance of a (derived) class. Since there are many situation where the abstractor class would not need to have data members, it’s desirable for the template to take advantage of this (potential) optimization.

It’s interesting (well, to me, anyway) to note that in the C AVL tree implementation, it’s much more straight-forward to avoid “padding” tree instances when the instantiation requires no per-tree-instance data (other than the root handle). (More straight-forward in the sense that you don’t need to use some weird special clause in the language definition.) But I think this is the only case where the C implementation seems to have an advantage over the C++ implementation.

7.2 Other Protected Members

The other protected members are most easily understood by reading the source code.

8 Version History

Version 1.0
- Initial version.

Version 1.1
- Fixed bugs in the handling of read errors.
- Minor changes to documentation.

Version 1.2
- Changes to document final, destructing iteration.

Version 1.3
- Improvements to code in example 1 showing final destructing iteration.
- Minor corrections and improvements to external documentation.

Version 1.4
- Added missing usage of *typename* reserve word at various places.
- Added usage of the “empty member” optimization.
- Cosmetic changes.

Version 1.5
- Added “subst” member function.

Version 1.6
- Fixed various warnings.
- Changed documentation to ODT format, fixes some URIs.
- Added condition-free MIT license.