```
**********************************************************
   26903 Fri Apr 11 14:22:19 2014
new/usr/src/common/avl/avl.c
4745 fix AVL code misspellings
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * AVL - generic AVL tree implementation for kernel use
 *
 * A complete description of AVL trees can be found in many CS textbooks.
 *
 * Here is a very brief overview. An AVL tree is a binary search tree that is
 * almost perfectly balanced. By "almost" perfectly balanced, we mean that at
 * any given node, the left and right subtrees are allowed to differ in height
 * by at most 1 level.
 *
 * This relaxation from a perfectly balanced binary tree allows doing
 * insertion and deletion relatively efficiently. Searching the tree is
 * still a fast operation, roughly O(log(N)).
 *
 * The key to insertion and deletion is a set of tree manipulations called
 * The key to insertion and deletion is a set of tree maniuplations called
 * rotations, which bring unbalanced subtrees back into the semi-balanced state.
 *
 * This implementation of AVL trees has the following peculiarities:
 *
 *      - The AVL specific data structures are physically embedded as fields
 *        in the "using" data structures.  To maintain generality the code
 *        must constantly translate between "avl_node_t *" and containing
 *        data structure "void *"s by adding/subtracting the avl_offset.
 *        data structure "void *"s by adding/subracting the avl_offset.
 *
 *      - Since the AVL data is always embedded in other structures, there is
 *        no locking or memory allocation in the AVL routines. This must be
 *        provided for by the enclosing data structure's semantics. Typically,
 *        avl_insert()/_add()/_remove()/avl_insert_here() require some kind of
 *        exclusive write lock. Other operations require a read lock.
 *
 *      - The implementation uses iteration instead of explicit recursion,
 *        since it is intended to run on limited size kernel stacks. Since
 *        there is no recursion stack present to move "up" in the tree,
 *        there is an explicit "parent" link in the avl_node_t.
```



```
 *
 *      - The left/right children pointers of a node are in an array.
 *        In the code, variables (instead of constants) are used to represent
 *        left and right indices.  The implementation is written as if it only
 *        dealt with left handed manipulations.  By changing the value assigned
 *        to "left", the code also works for right handed trees.  The
 *        following variables/terms are frequently used:
 *
 *              int left;       // 0 when dealing with left children,
 *                              // 1 for dealing with right children
 *
 *              int left_heavy; // -1 when left subtree is taller at some node,
 *                              // +1 when right subtree is taller
 *
 *              int right;      // will be the opposite of left (0 or 1)
 *              int right_heavy;// will be the opposite of left_heavy (-1 or 1)
 *
 *              int direction;  // 0 for "<" (ie. left child); 1 for ">" (right)
 *
 *        Though it is a little more confusing to read the code, the approach
 *        allows using half as much code (and hence cache footprint) for tree
 *        manipulations and eliminates many conditional branches.
 *
 *      - The avl_index_t is an opaque "cookie" used to find nodes at or
 *        adjacent to where a new value would be inserted in the tree. The value
 *        is a modified "avl_node_t *".  The bottom bit (normally 0 for a
 *        pointer) is set to indicate if that the new node has a value greater
 *        than the value of the indicated "avl_node_t *".
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/debug.h>
#include <sys/avl.h>
#include <sys/cmn_err.h>

/*
 * Small arrays to translate between balance (or diff) values and child indices.
 * Small arrays to translate between balance (or diff) values and child indeces.
 *
 * Code that deals with binary tree data structures will randomly use
 * left and right children when examining a tree.  C "if()" statements
 * which evaluate randomly suffer from very poor hardware branch prediction.
 * In this code we avoid some of the branch mispredictions by using the
 * following translation arrays. They replace random branches with an
 * additional memory reference. Since the translation arrays are both very
 * small the data should remain efficiently in cache.
 */
static const int  avl_child2balance[2]  = {-1, 1};
static const int  avl_balance2child[]   = {0, 0, 1};


/*
 * Walk from one node to the previous valued node (ie. an infix walk
 * towards the left). At any given node we do one of 2 things:
 *
 * - If there is a left child, go to it, then to it's rightmost descendant.
 *
 * - otherwise we return through parent nodes until we've come from a right
 *   child.
 * - otherwise we return thru parent nodes until we've come from a right child.
 *
 * Return Value:
 * NULL - if at the end of the nodes
 * otherwise next node
 */
```

```
void *
avl_walk(avl_tree_t *tree, void *oldnode, int left)
{
        size_t off = tree->avl_offset;
        avl_node_t *node = AVL_DATA2NODE(oldnode, off);
        int right = 1 - left;
        int was_child;


        /*
         * nowhere to walk to if tree is empty
         */
        if (node == NULL)
                return (NULL);

        /*
         * Visit the previous valued node. There are two possibilities:
         *
         * If this node has a left child, go down one left, then all
         * the way right.
         */
        if (node->avl_child[left] != NULL) {
                for (node = node->avl_child[left];
                    node->avl_child[right] != NULL;
                    node = node->avl_child[right])
                        ;
        /*
         * Otherwise, return thru left children as far as we can.
         */
        } else {
                for (;;) {
                        was_child = AVL_XCHILD(node);
                        node = AVL_XPARENT(node);
                        if (node == NULL)
                                return (NULL);
                        if (was_child == right)
                                break;
                }
        }

        return (AVL_NODE2DATA(node, off));
}
```

***______unchanged_portion_omitted_***

```
#define CHILDBIT        (1L)

/*
 * Post-order tree walk used to visit all tree nodes and destroy the tree
 * in post order. This is used for destroying a tree without paying any cost
 * in post order. This is used for destroying a tree w/o paying any cost
 * for rebalancing it.
 *
 * example:
 *
 *      void *cookie = NULL;
 *      my_data_t *node;
 *
 *      while ((node = avl_destroy_nodes(tree, &cookie)) != NULL)
 *              free(node);
 *      avl_destroy(tree);
 *
 * The cookie is really an avl_node_t to the current node's parent and
 * an indication of which child you looked at last.
 *
 * On input, a cookie value of CHILDBIT indicates the tree is done.
 */
```



```
void *
avl_destroy_nodes(avl_tree_t *tree, void **cookie)
{
        avl_node_t      *node;
        avl_node_t      *parent;
        int             child;
        void            *first;
        size_t          off = tree->avl_offset;

        /*
         * Initial calls go to the first node or it's right descendant.
         */
        if (*cookie == NULL) {
                first = avl_first(tree);

                /*
                 * deal with an empty tree
                 */
                if (first == NULL) {
                        *cookie = (void *)CHILDBIT;
                        return (NULL);
                }

                node = AVL_DATA2NODE(first, off);
                parent = AVL_XPARENT(node);
                goto check_right_side;
        }

        /*
         * If there is no parent to return to we are done.
         */
        parent = (avl_node_t *)((uintptr_t)(*cookie) & ~CHILDBIT);
        if (parent == NULL) {
                if (tree->avl_root != NULL) {
                        ASSERT(tree->avl_numnodes == 1);
                        tree->avl_root = NULL;
                        tree->avl_numnodes = 0;
                }
                return (NULL);
        }

        /*
         * Remove the child pointer we just visited from the parent and tree.
         */
        child = (uintptr_t)(*cookie) & CHILDBIT;
        parent->avl_child[child] = NULL;
        ASSERT(tree->avl_numnodes > 1);
        --tree->avl_numnodes;

        /*
         * If we just did a right child or there isn't one, go up to parent.
         */
        if (child == 1 || parent->avl_child[1] == NULL) {
                node = parent;
                parent = AVL_XPARENT(parent);
                goto done;
        }

        /*
         * Do parent's right child, then leftmost descendent.
         */
        node = parent->avl_child[1];
        while (node->avl_child[0] != NULL) {
                parent = node;
                node = node->avl_child[0];
        }
```

```
        /*
         * If here, we moved to a left child. It may have one
         * child on the right (when balance == +1).
         */
check_right_side:
        if (node->avl_child[1] != NULL) {
                ASSERT(AVL_XBALANCE(node) == 1);
                parent = node;
                node = node->avl_child[1];
                ASSERT(node->avl_child[0] == NULL &&
                    node->avl_child[1] == NULL);
        } else {
                ASSERT(AVL_XBALANCE(node) <= 0);
        }

done:
        if (parent == NULL) {
                *cookie = (void *)CHILDBIT;
                ASSERT(node == tree->avl_root);
        } else {
                *cookie = (void *)((uintptr_t)parent | AVL_XCHILD(node));
        }

        return (AVL_NODE2DATA(node, off));
}
```
______***unchanged_portion_omitted_***

```
*************************************************************
    8980 Fri Apr 11 14:22:19 2014
new/usr/src/uts/common/sys/avl.h
4745 fix AVL code misspellings
*************************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _AVL_H
#define _AVL_H

/*
 * This is a private header file.  Applications should not directly include
 * this file.
 */

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/types.h>
#include <sys/avl_impl.h>

/*
 * This is a generic implementation of AVL trees for use in the Solaris kernel.
 * This is a generic implemenatation of AVL trees for use in the Solaris kernel.
 * The interfaces provide an efficient way of implementing an ordered set of
 * data structures.
 *
 * AVL trees provide an alternative to using an ordered linked list. Using AVL
 * trees will usually be faster, however they requires more storage. An ordered
 * linked list in general requires 2 pointers in each data structure. The
 * AVL tree implementation uses 3 pointers. The following chart gives the
 * approximate performance of operations with the different approaches:
 *
 *      Operation        Link List      AVL tree
 *      ---------        --------       --------
 *      lookup             O(n)         O(log(n))
 *
 *      insert 1 node    constant       constant
 *
 *      delete 1 node    constant       between constant and O(log(n))
 *
 *      delete all nodes   O(n)         O(n)
```



```
 *
 *      visit the next
 *      or prev node     constant       between constant and O(log(n))
 *
 *
 * The data structure nodes are anchored at an "avl_tree_t" (the equivalent
 * of a list header) and the individual nodes will have a field of
 * type "avl_node_t" (corresponding to list pointers).
 *
 * The type "avl_index_t" is used to indicate a position in the list for
 * certain calls.
 *
 * The usage scenario is generally:
 *
 * 1. Create the list/tree with: avl_create()
 *
 * followed by any mixture of:
 *
 * 2a. Insert nodes with: avl_add(), or avl_find() and avl_insert()
 *
 * 2b. Visited elements with:
 *       avl_first() - returns the lowest valued node
 *       avl_last() - returns the highest valued node
 *       AVL_NEXT() - given a node go to next higher one
 *       AVL_PREV() - given a node go to previous lower one
 *
 * 2c.  Find the node with the closest value either less than or greater
 *      than a given value with avl_nearest().
 *
 * 2d. Remove individual nodes from the list/tree with avl_remove().
 *
 * and finally when the list is being destroyed
 *
 * 3. Use avl_destroy_nodes() to quickly process/free up any remaining nodes.
 *    Note that once you use avl_destroy_nodes(), you can no longer
 *    use any routine except avl_destroy_nodes() and avl_destoy().
 *
 * 4. Use avl_destroy() to destroy the AVL tree itself.
 *
 * Any locking for multiple thread access is up to the user to provide, just
 * as is needed for any linked list implementation.
 */


/*
 * Type used for the root of the AVL tree.
 */
typedef struct avl_tree avl_tree_t;

/*
 * The data nodes in the AVL tree must have a field of this type.
 */
typedef struct avl_node avl_node_t;

/*
 * An opaque type used to locate a position in the tree where a node
 * would be inserted.
 */
typedef uintptr_t avl_index_t;


/*
 * Direction constants used for avl_nearest().
 */
#define AVL_BEFORE      (0)
#define AVL_AFTER       (1)
```

```
/*
 * Prototypes
 *
 * Where not otherwise mentioned, "void *" arguments are a pointer to the
 * user data structure which must contain a field of type avl_node_t.
 *
 * Also assume the user data structures looks like:
 *      stuct my_type {
 *              ...
 *              avl_node_t      my_link;
 *              ...
 *      };
 */

/*
 * Initialize an AVL tree. Arguments are:
 *
 * tree   - the tree to be initialized
 * compar - function to compare two nodes, it must return exactly: -1, 0, or +1
 *          -1 for <, 0 for ==, and +1 for >
 * size   - the value of sizeof(struct my_type)
 * offset - the value of OFFSETOF(struct my_type, my_link)
 */
extern void avl_create(avl_tree_t *tree,
        int (*compar) (const void *, const void *), size_t size, size_t offset);


/*
 * Find a node with a matching value in the tree. Returns the matching node
 * found. If not found, it returns NULL and then if "where" is not NULL it sets
 * "where" for use with avl_insert() or avl_nearest().
 *
 * node   - node that has the value being looked for
 * where  - position for use with avl_nearest() or avl_insert(), may be NULL
 */
extern void *avl_find(avl_tree_t *tree, const void *node, avl_index_t *where);

/*
 * Insert a node into the tree.
 *
 * node   - the node to insert
 * where  - position as returned from avl_find()
 */
extern void avl_insert(avl_tree_t *tree, void *node, avl_index_t where);

/*
 * Insert "new_data" in "tree" in the given "direction" either after
 * or before the data "here".
 *
 * This might be useful for avl clients caching recently accessed
 * This might be usefull for avl clients caching recently accessed
 * data to avoid doing avl_find() again for insertion.
 *
 * new_data     - new data to insert
 * here         - existing node in "tree"
 * direction    - either AVL_AFTER or AVL_BEFORE the data "here".
 */
extern void avl_insert_here(avl_tree_t *tree, void *new_data, void *here,
    int direction);


/*
 * Return the first or last valued node in the tree. Will return NULL
 * if the tree is empty.
```



```
 *
 */
extern void *avl_first(avl_tree_t *tree);
extern void *avl_last(avl_tree_t *tree);


/*
 * Return the next or previous valued node in the tree.
 * AVL_NEXT() will return NULL if at the last node.
 * AVL_PREV() will return NULL if at the first node.
 *
 * node   - the node from which the next or previous node is found
 */
#define AVL_NEXT(tree, node)    avl_walk(tree, node, AVL_AFTER)
#define AVL_PREV(tree, node)    avl_walk(tree, node, AVL_BEFORE)


/*
 * Find the node with the nearest value either greater or less than
 * the value from a previous avl_find(). Returns the node or NULL if
 * there isn't a matching one.
 *
 * where     - position as returned from avl_find()
 * direction - either AVL_BEFORE or AVL_AFTER
 *
 * EXAMPLE get the greatest node that is less than a given value:
 *
 *      avl_tree_t *tree;
 *      struct my_data look_for_value = {....};
 *      struct my_data *node;
 *      struct my_data *less;
 *      avl_index_t where;
 *
 *      node = avl_find(tree, &look_for_value, &where);
 *      if (node != NULL)
 *              less = AVL_PREV(tree, node);
 *      else
 *              less = avl_nearest(tree, where, AVL_BEFORE);
 */
extern void *avl_nearest(avl_tree_t *tree, avl_index_t where, int direction);


/*
 * Add a single node to the tree.
 * The node must not be in the tree, and it must not
 * compare equal to any other node already in the tree.
 *
 * node   - the node to add
 */
extern void avl_add(avl_tree_t *tree, void *node);


/*
 * Remove a single node from the tree.  The node must be in the tree.
 *
 * node   - the node to remove
 */
extern void avl_remove(avl_tree_t *tree, void *node);

/*
 * Reinsert a node only if its order has changed relative to its nearest
 * neighbors. To optimize performance avl_update_lt() checks only the previous
 * node and avl_update_gt() checks only the next node. Use avl_update_lt() and
 * avl_update_gt() only if you know the direction in which the order of the
 * node may change.
 */
```

```
extern boolean_t avl_update(avl_tree_t *, void *);
extern boolean_t avl_update_lt(avl_tree_t *, void *);
extern boolean_t avl_update_gt(avl_tree_t *, void *);

/*
 * Return the number of nodes in the tree
 */
extern ulong_t avl_numnodes(avl_tree_t *tree);

/*
 * Return B_TRUE if there are zero nodes in the tree, B_FALSE otherwise.
 */
extern boolean_t avl_is_empty(avl_tree_t *tree);

/*
 * Used to destroy any remaining nodes in a tree. The cookie argument should
 * be initialized to NULL before the first call. Returns a node that has been
 * removed from the tree and may be free()'d. Returns NULL when the tree is
 * empty.
 *
 * Once you call avl_destroy_nodes(), you can only continuing calling it and
 * finally avl_destroy(). No other AVL routines will be valid.
 *
 * cookie - a "void *" used to save state between calls to avl_destroy_nodes()
 *
 * EXAMPLE:
 *      avl_tree_t *tree;
 *      struct my_data *node;
 *      void *cookie;
 *
 *      cookie = NULL;
 *      while ((node = avl_destroy_nodes(tree, &cookie)) != NULL)
 *              free(node);
 *      avl_destroy(tree);
 */
extern void *avl_destroy_nodes(avl_tree_t *tree, void **cookie);


/*
 * Final destroy of an AVL tree. Arguments are:
 *
 * tree   - the empty tree to destroy
 */
extern void avl_destroy(avl_tree_t *tree);



#ifdef  __cplusplus
}
```

______***unchanged_portion_omitted_***