

```

*****
28038 Sat Aug 1 08:08:32 2015
new/usr/src/common/avl/avl.c
6091 avl_add doesn't assert on non-debug builds
Reviewed by: Andy Stormont <astormont@racktopsystems.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[ ]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22  * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23  * Use is subject to license terms.
24  */

26 /*
27  * Copyright (c) 2014 by Delphix. All rights reserved.
28  * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
29  * #endif /* !codereview */
30  */

32 /*
33  * AVL - generic AVL tree implementation for kernel use
34  *
35  * A complete description of AVL trees can be found in many CS textbooks.
36  *
37  * Here is a very brief overview. An AVL tree is a binary search tree that is
38  * almost perfectly balanced. By "almost" perfectly balanced, we mean that at
39  * any given node, the left and right subtrees are allowed to differ in height
40  * by at most 1 level.
41  *
42  * This relaxation from a perfectly balanced binary tree allows doing
43  * insertion and deletion relatively efficiently. Searching the tree is
44  * still a fast operation, roughly O(log(N)).
45  *
46  * The key to insertion and deletion is a set of tree manipulations called
47  * rotations, which bring unbalanced subtrees back into the semi-balanced state.
48  *
49  * This implementation of AVL trees has the following peculiarities:
50  *
51  * - The AVL specific data structures are physically embedded as fields
52  *   in the "using" data structures. To maintain generality the code
53  *   must constantly translate between "avl_node_t *" and containing
54  *   data structure "void **s" by adding/subtracting the avl_offset.
55  *
56  * - Since the AVL data is always embedded in other structures, there is
57  *   no locking or memory allocation in the AVL routines. This must be
58  *   provided for by the enclosing data structure's semantics. Typically,
59  *   avl_insert()/_add()/_remove()/avl_insert_here() require some kind of
60  *   exclusive write lock. Other operations require a read lock.

```

```

61  *
62  * - The implementation uses iteration instead of explicit recursion,
63  *   since it is intended to run on limited size kernel stacks. Since
64  *   there is no recursion stack present to move "up" in the tree,
65  *   there is an explicit "parent" link in the avl_node_t.
66  *
67  * - The left/right children pointers of a node are in an array.
68  *   In the code, variables (instead of constants) are used to represent
69  *   left and right indices. The implementation is written as if it only
70  *   dealt with left handed manipulations. By changing the value assigned
71  *   to "left", the code also works for right handed trees. The
72  *   following variables/terms are frequently used:
73  *
74  *   int left;           // 0 when dealing with left children,
75  *                       // 1 for dealing with right children
76  *
77  *   int left_heavy;    // -1 when left subtree is taller at some node,
78  *                       // +1 when right subtree is taller
79  *
80  *   int right;         // will be the opposite of left (0 or 1)
81  *   int right_heavy;   // will be the opposite of left_heavy (-1 or 1)
82  *
83  *   int direction;    // 0 for "<" (ie. left child); 1 for ">" (right)
84  *
85  *   Though it is a little more confusing to read the code, the approach
86  *   allows using half as much code (and hence cache footprint) for tree
87  *   manipulations and eliminates many conditional branches.
88  *
89  * - The avl_index_t is an opaque "cookie" used to find nodes at or
90  *   adjacent to where a new value would be inserted in the tree. The value
91  *   is a modified "avl_node_t *". The bottom bit (normally 0 for a
92  *   pointer) is set to indicate if that the new node has a value greater
93  *   than the value of the indicated "avl_node_t *".
94  *
95  * Note - in addition to userland (e.g. libavl and libutil) and the kernel
96  * (e.g. genunix), avl.c is compiled into ld.so and kmd's genunix module,
97  * which each have their own compilation environments and subsequent
98  * requirements. Each of these environments must be considered when adding
99  * dependencies from avl.c.
100 */

102 #include <sys/types.h>
103 #include <sys/param.h>
104 #include <sys/debug.h>
105 #include <sys/avl.h>
106 #include <sys/cmn_err.h>

108 /*
109  * Small arrays to translate between balance (or diff) values and child indices.
110  *
111  * Code that deals with binary tree data structures will randomly use
112  * left and right children when examining a tree. C "if()" statements
113  * which evaluate randomly suffer from very poor hardware branch prediction.
114  * In this code we avoid some of the branch mispredictions by using the
115  * following translation arrays. They replace random branches with an
116  * additional memory reference. Since the translation arrays are both very
117  * small the data should remain efficiently in cache.
118  */
119 static const int avl_child2balance[2] = {-1, 1};
120 static const int avl_balance2child[] = {0, 0, 1};

123 /*
124  * Walk from one node to the previous valued node (ie. an infix walk
125  * towards the left). At any given node we do one of 2 things:
126  *

```

```

127 * - If there is a left child, go to it, then to it's rightmost descendant.
128 *
129 * - otherwise we return through parent nodes until we've come from a right
130 *   child.
131 *
132 * Return Value:
133 * NULL - if at the end of the nodes
134 * otherwise next node
135 */
136 void *
137 avl_walk(avl_tree_t *tree, void *oldnode, int left)
138 {
139     size_t off = tree->avl_offset;
140     avl_node_t *node = AVL_DATA2NODE(oldnode, off);
141     int right = 1 - left;
142     int was_child;

145     /*
146      * nowhere to walk to if tree is empty
147      */
148     if (node == NULL)
149         return (NULL);

151     /*
152      * Visit the previous valued node. There are two possibilities:
153      *
154      * If this node has a left child, go down one left, then all
155      * the way right.
156      */
157     if (node->avl_child[left] != NULL) {
158         for (node = node->avl_child[left];
159              node->avl_child[right] != NULL;
160              node = node->avl_child[right])
161             ;
162     }
163     /*
164      * Otherwise, return thru left children as far as we can.
165      */
166     } else {
167         for (;;) {
168             was_child = AVL_XCHILD(node);
169             node = AVL_XPARENT(node);
170             if (node == NULL)
171                 return (NULL);
172             if (was_child == right)
173                 break;
174         }
175     }

176     return (AVL_NODE2DATA(node, off));
177 }

179 /*
180 * Return the lowest valued node in a tree or NULL.
181 * (leftmost child from root of tree)
182 */
183 void *
184 avl_first(avl_tree_t *tree)
185 {
186     avl_node_t *node;
187     avl_node_t *prev = NULL;
188     size_t off = tree->avl_offset;

190     for (node = tree->avl_root; node != NULL; node = node->avl_child[0])
191         prev = node;

```

```

193     if (prev != NULL)
194         return (AVL_NODE2DATA(prev, off));
195     return (NULL);
196 }

198 /*
199 * Return the highest valued node in a tree or NULL.
200 * (rightmost child from root of tree)
201 */
202 void *
203 avl_last(avl_tree_t *tree)
204 {
205     avl_node_t *node;
206     avl_node_t *prev = NULL;
207     size_t off = tree->avl_offset;

209     for (node = tree->avl_root; node != NULL; node = node->avl_child[1])
210         prev = node;

212     if (prev != NULL)
213         return (AVL_NODE2DATA(prev, off));
214     return (NULL);
215 }

217 /*
218 * Access the node immediately before or after an insertion point.
219 *
220 * "avl_index_t" is a (avl_node_t *) with the bottom bit indicating a child
221 *
222 * Return value:
223 * NULL: no node in the given direction
224 * "void *" of the found tree node
225 */
226 void *
227 avl_nearest(avl_tree_t *tree, avl_index_t where, int direction)
228 {
229     int child = AVL_INDEX2CHILD(where);
230     avl_node_t *node = AVL_INDEX2NODE(where);
231     void *data;
232     size_t off = tree->avl_offset;

234     if (node == NULL) {
235         ASSERT(tree->avl_root == NULL);
236         return (NULL);
237     }
238     data = AVL_NODE2DATA(node, off);
239     if (child != direction)
240         return (data);

242     return (avl_walk(tree, data, direction));
243 }

246 /*
247 * Search for the node which contains "value". The algorithm is a
248 * simple binary tree search.
249 *
250 * return value:
251 * NULL: the value is not in the AVL tree
252 * *where (if not NULL) is set to indicate the insertion point
253 * "void *" of the found tree node
254 */
255 void *
256 avl_find(avl_tree_t *tree, const void *value, avl_index_t *where)
257 {
258     avl_node_t *node;

```

```

259     avl_node_t *prev = NULL;
260     int child = 0;
261     int diff;
262     size_t off = tree->avl_offset;

264     for (node = tree->avl_root; node != NULL;
265         node = node->avl_child[child]) {

267         prev = node;

269         diff = tree->avl_compar(value, AVL_NODE2DATA(node, off));
270         ASSERT(-1 <= diff && diff <= 1);
271         if (diff == 0) {
272 #ifdef DEBUG
273             if (where != NULL)
274                 *where = 0;
275 #endif
276             return (AVL_NODE2DATA(node, off));
277         }
278         child = avl_balance2child[1 + diff];

280     }

282     if (where != NULL)
283         *where = AVL_MKINDEX(prev, child);

285     return (NULL);
286 }

289 /*
290 * Perform a rotation to restore balance at the subtree given by depth.
291 *
292 * This routine is used by both insertion and deletion. The return value
293 * indicates:
294 *     0 : subtree did not change height
295 *     !0 : subtree was reduced in height
296 *
297 * The code is written as if handling left rotations, right rotations are
298 * symmetric and handled by swapping values of variables right/left[_heavy]
299 *
300 * On input balance is the "new" balance at "node". This value is either
301 * -2 or +2.
302 */
303 static int
304 avl_rotation(avl_tree_t *tree, avl_node_t *node, int balance)
305 {
306     int left = !(balance < 0);      /* when balance = -2, left will be 0 */
307     int right = 1 - left;
308     int left_heavy = balance >> 1;
309     int right_heavy = -left_heavy;
310     avl_node_t *parent = AVL_XPARENT(node);
311     avl_node_t *child = node->avl_child[left];
312     avl_node_t *cright;
313     avl_node_t *gchild;
314     avl_node_t *gright;
315     avl_node_t *gleft;
316     int which_child = AVL_XCHILD(node);
317     int child_bal = AVL_XBALANCE(child);

319     /* BEGIN CSTYLED */
320     /*
321     * case 1 : node is overly left heavy, the left child is balanced or
322     * also left heavy. This requires the following rotation.
323     *
324     *
325     *
326     *
327     *
328     *
329     *
330     *
331     *
332     *
333     *
334     *
335     *
336     *
337     *
338     *
339     *
340     *
341     *
342     *
343     *
344     *
345     *
346     *
347     *
348     *
349     *
350     *
351     *
352     *
353     *
354     *
355     *
356     *
357     *
358     *
359     *
360     *
361     *
362     *
363     *
364     *
365     *
366     *
367     *
368     *
369     *
370     *
371     *
372     *
373     *
374     *
375     *
376     *
377     *
378     *
379     *
380     *
381     *
382     *
383     *
384     *
385     *
386     *
387     *
388     *
389     *
390     *
391     *
392     *
393     *
394     *
395     *
396     *
397     *
398     *
399     *
400     *
401     *
402     *
403     *
404     *
405     *
406     *
407     *
408     *
409     *
410     *
411     *
412     *
413     *
414     *
415     *
416     *
417     *
418     *
419     *
420     *
421     *
422     *
423     *
424     *
425     *
426     *
427     *
428     *
429     *
430     *
431     *
432     *
433     *
434     *
435     *
436     *
437     *
438     *
439     *
440     *
441     *
442     *
443     *
444     *
445     *
446     *
447     *
448     *
449     *
450     *
451     *
452     *
453     *
454     *
455     *
456     *
457     *
458     *
459     *
460     *
461     *
462     *
463     *
464     *
465     *
466     *
467     *
468     *
469     *
470     *
471     *
472     *
473     *
474     *
475     *
476     *
477     *
478     *
479     *
480     *
481     *
482     *
483     *
484     *
485     *
486     *
487     *
488     *
489     *
490     *
491     *
492     *
493     *
494     *
495     *
496     *
497     *
498     *
499     *
500     *
501     *
502     *
503     *
504     *
505     *
506     *
507     *
508     *
509     *
510     *
511     *
512     *
513     *
514     *
515     *
516     *
517     *
518     *
519     *
520     *
521     *
522     *
523     *
524     *
525     *
526     *
527     *
528     *
529     *
530     *
531     *
532     *
533     *
534     *
535     *
536     *
537     *
538     *
539     *
540     *
541     *
542     *
543     *
544     *
545     *
546     *
547     *
548     *
549     *
550     *
551     *
552     *
553     *
554     *
555     *
556     *
557     *
558     *
559     *
560     *
561     *
562     *
563     *
564     *
565     *
566     *
567     *
568     *
569     *
570     *
571     *
572     *
573     *
574     *
575     *
576     *
577     *
578     *
579     *
580     *
581     *
582     *
583     *
584     *
585     *
586     *
587     *
588     *
589     *
590     *
591     *
592     *
593     *
594     *
595     *
596     *
597     *
598     *
599     *
600     *
601     *
602     *
603     *
604     *
605     *
606     *
607     *
608     *
609     *
610     *
611     *
612     *
613     *
614     *
615     *
616     *
617     *
618     *
619     *
620     *
621     *
622     *
623     *
624     *
625     *
626     *
627     *
628     *
629     *
630     *
631     *
632     *
633     *
634     *
635     *
636     *
637     *
638     *
639     *
640     *
641     *
642     *
643     *
644     *
645     *
646     *
647     *
648     *
649     *
650     *
651     *
652     *
653     *
654     *
655     *
656     *
657     *
658     *
659     *
660     *
661     *
662     *
663     *
664     *
665     *
666     *
667     *
668     *
669     *
670     *
671     *
672     *
673     *
674     *
675     *
676     *
677     *
678     *
679     *
680     *
681     *
682     *
683     *
684     *
685     *
686     *
687     *
688     *
689     *
690     *
691     *
692     *
693     *
694     *
695     *
696     *
697     *
698     *
699     *
700     *
701     *
702     *
703     *
704     *
705     *
706     *
707     *
708     *
709     *
710     *
711     *
712     *
713     *
714     *
715     *
716     *
717     *
718     *
719     *
720     *
721     *
722     *
723     *
724     *
725     *
726     *
727     *
728     *
729     *
730     *
731     *
732     *
733     *
734     *
735     *
736     *
737     *
738     *
739     *
740     *
741     *
742     *
743     *
744     *
745     *
746     *
747     *
748     *
749     *
750     *
751     *
752     *
753     *
754     *
755     *
756     *
757     *
758     *
759     *
760     *
761     *
762     *
763     *
764     *
765     *
766     *
767     *
768     *
769     *
770     *
771     *
772     *
773     *
774     *
775     *
776     *
777     *
778     *
779     *
780     *
781     *
782     *
783     *
784     *
785     *
786     *
787     *
788     *
789     *
790     *
791     *
792     *
793     *
794     *
795     *
796     *
797     *
798     *
799     *
800     *
801     *
802     *
803     *
804     *
805     *
806     *
807     *
808     *
809     *
810     *
811     *
812     *
813     *
814     *
815     *
816     *
817     *
818     *
819     *
820     *
821     *
822     *
823     *
824     *
825     *
826     *
827     *
828     *
829     *
830     *
831     *
832     *
833     *
834     *
835     *
836     *
837     *
838     *
839     *
840     *
841     *
842     *
843     *
844     *
845     *
846     *
847     *
848     *
849     *
850     *
851     *
852     *
853     *
854     *
855     *
856     *
857     *
858     *
859     *
860     *
861     *
862     *
863     *
864     *
865     *
866     *
867     *
868     *
869     *
870     *
871     *
872     *
873     *
874     *
875     *
876     *
877     *
878     *
879     *
880     *
881     *
882     *
883     *
884     *
885     *
886     *
887     *
888     *
889     *
890     *
891     *
892     *
893     *
894     *
895     *
896     *
897     *
898     *
899     *
900     *
901     *
902     *
903     *
904     *
905     *
906     *
907     *
908     *
909     *
910     *
911     *
912     *
913     *
914     *
915     *
916     *
917     *
918     *
919     *
920     *
921     *
922     *
923     *
924     *
925     *
926     *
927     *
928     *
929     *
930     *
931     *
932     *
933     *
934     *
935     *
936     *
937     *
938     *
939     *
940     *
941     *
942     *
943     *
944     *
945     *
946     *
947     *
948     *
949     *
950     *
951     *
952     *
953     *
954     *
955     *
956     *
957     *
958     *
959     *
960     *
961     *
962     *
963     *
964     *
965     *
966     *
967     *
968     *
969     *
970     *
971     *
972     *
973     *
974     *
975     *
976     *
977     *
978     *
979     *
980     *
981     *
982     *
983     *
984     *
985     *
986     *
987     *
988     *
989     *
990     *
991     *
992     *
993     *
994     *
995     *
996     *
997     *
998     *
999     *
1000    */

```

```

325     *
326     *
327     *
328     *
329     *
330     *
331     *
332     *
333     *
334     *
335     *
336     *
337     *
338     *
339     *
340     *
341     *
342     *
343     *
344     *
345     *
346     *
347     *
348     *
349     *
350     *
351     *
352     *
353     *
354     *
355     *
356     *
357     *
358     *
359     *
360     *
361     *
362     *
363     *
364     *
365     *
366     *
367     *
368     *
369     *
370     *
371     *
372     *
373     *
374     *
375     *
376     *
377     *
378     *
379     *
380     *
381     *
382     *
383     *
384     *
385     *
386     *
387     *
388     *
389     *
390     *
391     *
392     *
393     *
394     *
395     *
396     *
397     *
398     *
399     *
400     *
401     *
402     *
403     *
404     *
405     *
406     *
407     *
408     *
409     *
410     *
411     *
412     *
413     *
414     *
415     *
416     *
417     *
418     *
419     *
420     *
421     *
422     *
423     *
424     *
425     *
426     *
427     *
428     *
429     *
430     *
431     *
432     *
433     *
434     *
435     *
436     *
437     *
438     *
439     *
440     *
441     *
442     *
443     *
444     *
445     *
446     *
447     *
448     *
449     *
450     *
451     *
452     *
453     *
454     *
455     *
456     *
457     *
458     *
459     *
460     *
461     *
462     *
463     *
464     *
465     *
466     *
467     *
468     *
469     *
470     *
471     *
472     *
473     *
474     *
475     *
476     *
477     *
478     *
479     *
480     *
481     *
482     *
483     *
484     *
485     *
486     *
487     *
488     *
489     *
490     *
491     *
492     *
493     *
494     *
495     *
496     *
497     *
498     *
499     *
500     *
501     *
502     *
503     *
504     *
505     *
506     *
507     *
508     *
509     *
510     *
511     *
512     *
513     *
514     *
515     *
516     *
517     *
518     *
519     *
520     *
521     *
522     *
523     *
524     *
525     *
526     *
527     *
528     *
529     *
530     *
531     *
532     *
533     *
534     *
535     *
536     *
537     *
538     *
539     *
540     *
541     *
542     *
543     *
544     *
545     *
546     *
547     *
548     *
549     *
550     *
551     *
552     *
553     *
554     *
555     *
556     *
557     *
558     *
559     *
560     *
561     *
562     *
563     *
564     *
565     *
566     *
567     *
568     *
569     *
570     *
571     *
572     *
573     *
574     *
575     *
576     *
577     *
578     *
579     *
580     *
581     *
582     *
583     *
584     *
585     *
586     *
587     *
588     *
589     *
590     *
591     *
592     *
593     *
594     *
595     *
596     *
597     *
598     *
599     *
600     *
601     *
602     *
603     *
604     *
605     *
606     *
607     *
608     *
609     *
610     *
611     *
612     *
613     *
614     *
615     *
616     *
617     *
618     *
619     *
620     *
621     *
622     *
623     *
624     *
625     *
626     *
627     *
628     *
629     *
630     *
631     *
632     *
633     *
634     *
635     *
636     *
637     *
638     *
639     *
640     *
641     *
642     *
643     *
644     *
645     *
646     *
647     *
648     *
649     *
650     *
651     *
652     *
653     *
654     *
655     *
656     *
657     *
658     *
659     *
660     *
661     *
662     *
663     *
664     *
665     *
666     *
667     *
668     *
669     *
670     *
671     *
672     *
673     *
674     *
675     *
676     *
677     *
678     *
679     *
680     *
681     *
682     *
683     *
684     *
685     *
686     *
687     *
688     *
689     *
690     *
691     *
692     *
693     *
694     *
695     *
696     *
697     *
698     *
699     *
700     *
701     *
702     *
703     *
704     *
705     *
706     *
707     *
708     *
709     *
710     *
711     *
712     *
713     *
714     *
715     *
716     *
717     *
718     *
719     *
720     *
721     *
722     *
723     *
724     *
725     *
726     *
727     *
728     *
729     *
730     *
731     *
732     *
733     *
734     *
735     *
736     *
737     *
738     *
739     *
740     *
741     *
742     *
743     *
744     *
745     *
746     *
747     *
748     *
749     *
750     *
751     *
752     *
753     *
754     *
755     *
756     *
757     *
758     *
759     *
760     *
761     *
762     *
763     *
764     *
765     *
766     *
767     *
768     *
769     *
770     *
771     *
772     *
773     *
774     *
775     *
776     *
777     *
778     *
779     *
780     *
781     *
782     *
783     *
784     *
785     *
786     *
787     *
788     *
789     *
790     *
791     *
792     *
793     *
794     *
795     *
796     *
797     *
798     *
799     *
800     *
801     *
802     *
803     *
804     *
805     *
806     *
807     *
808     *
809     *
810     *
811     *
812     *
813     *
814     *
815     *
816     *
817     *
818     *
819     *
820     *
821     *
822     *
823     *
824     *
825     *
826     *
827     *
828     *
829     *
830     *
831     *
832     *
833     *
834     *
835     *
836     *
837     *
838     *
839     *
840     *
841     *
842     *
843     *
844     *
845     *
846     *
847     *
848     *
849     *
850     *
851     *
852     *
853     *
854     *
855     *
856     *
857     *
858     *
859     *
860     *
861     *
862     *
863     *
864     *
865     *
866     *
867     *
868     *
869     *
870     *
871     *
872     *
873     *
874     *
875     *
876     *
877     *
878     *
879     *
880     *
881     *
882     *
883     *
884     *
885     *
886     *
887     *
888     *
889     *
890     *
891     *
892     *
893     *
894     *
895     *
896     *
897     *
898     *
899     *
900     *
901     *
902     *
903     *
904     *
905     *
906     *
907     *
908     *
909     *
910     *
911     *
912     *
913     *
914     *
915     *
916     *
917     *
918     *
919     *
920     *
921     *
922     *
923     *
924     *
925     *
926     *
927     *
928     *
929     *
930     *
931     *
932     *
933     *
934     *
935     *
936     *
937     *
938     *
939     *
940     *
941     *
942     *
943     *
944     *
945     *
946     *
947     *
948     *
949     *
950     *
951     *
952     *
953     *
954     *
955     *
956     *
957     *
958     *
959     *
960     *
961     *
962     *
963     *
964     *
965     *
966     *
967     *
968     *
969     *
970     *
971     *
972     *
973     *
974     *
975     *
976     *
977     *
978     *
979     *
980     *
981     *
982     *
983     *
984     *
985     *
986     *
987     *
988     *
989     *
990     *
991     *
992     *
993     *
994     *
995     *
996     *
997     *
998     *
999     *
1000    */

```

```

391  * a different rotation.
392  *
393  *
394  *          (node b:-2)
395  *         /  \
396  *        /    \
397  *       /      \
398  *      /        \
399  *     /          \
400  *    /            \
401  *   /              \
402  *  /                \
403  * /                  \
404 * /                    \
405 * /                      \
406 * /                        \
407 * /                          \
408 * /                            \
409 * /                              \
410 * /                                \
411 * /                                  \
412 * /                                  \
413 * /                                  \
414 * /                                  \
415 * /                                  \
416 * /                                  \
417 * /                                  \
418 * /                                  \
419 * /                                  \
420 * /                                  \
421 * /                                  \
422 * /                                  \
423 * /                                  \
424 * /                                  \
425 * /                                  \
426 * /                                  \
427 * /                                  \
428 * /                                  \
429 * /                                  \
430 * /                                  \
431 * /                                  \
432 * /                                  \
433 * /                                  \
434 * /                                  \
435 * /                                  \
436 * /                                  \
437 * /                                  \
438 * /                                  \
439 * /                                  \
440 * /                                  \
441 * /                                  \
442 * /                                  \
443 * /                                  \
444 * /                                  \
445 * /                                  \
446 * /                                  \
447 * /                                  \
448 * /                                  \
449 * /                                  \
450 * /                                  \
451 * /                                  \
452 * /                                  \
453 * /                                  \
454 * /                                  \
455 * /                                  \
456 * /                                  \

```

becomes:

```

406  *          (gchild b:0)
407  *         /  \
408  *        /    \
409  *       /      \
410  *      /        \
411  *     /          \
412  *    /            \
413  *   /              \
414  *  /                \
415  * /                  \
416 * /                    \
417 * /                      \
418 * /                        \
419 * /                          \
420 * /                            \
421 * /                              \
422 * /                                \
423 * /                                  \
424 * /                                  \
425 * /                                  \
426 * /                                  \
427 * /                                  \
428 * /                                  \
429 * /                                  \
430 * /                                  \
431 * /                                  \
432 * /                                  \
433 * /                                  \
434 * /                                  \
435 * /                                  \
436 * /                                  \
437 * /                                  \
438 * /                                  \
439 * /                                  \
440 * /                                  \
441 * /                                  \
442 * /                                  \
443 * /                                  \
444 * /                                  \
445 * /                                  \
446 * /                                  \
447 * /                                  \
448 * /                                  \
449 * /                                  \
450 * /                                  \
451 * /                                  \
452 * /                                  \
453 * /                                  \
454 * /                                  \
455 * /                                  \
456 * /                                  \

```

computing the new balances is more complicated. As an example:  
if gchild was right\_heavy, then child is now left heavy  
else it is balanced

```

420 /* END CSTYLED */
421 gchild = child->avl_child[right];
422 gleft = gchild->avl_child[left];
423 gright = gchild->avl_child[right];

```

```

425 /*
426  * move gright to left child of node and
427  *
428  * move gleft to right child of node
429  */
430 node->avl_child[left] = gright;
431 if (gright != NULL) {
432     AVL_SETPARENT(gright, node);
433     AVL_SETCHILD(gright, left);
434 }

```

```

436 child->avl_child[right] = gleft;
437 if (gleft != NULL) {
438     AVL_SETPARENT(gleft, child);
439     AVL_SETCHILD(gleft, right);
440 }

```

```

442 /*
443  * move child to left child of gchild and
444  *
445  * move node to right child of gchild and
446  *
447  * fixup parent of all this to point to gchild
448  */
449 balance = AVL_XBALANCE(gchild);
450 gchild->avl_child[left] = child;
451 AVL_SETBALANCE(child, (balance == right_heavy ? left_heavy : 0));
452 AVL_SETPARENT(child, gchild);
453 AVL_SETCHILD(child, left);

```

```

455 gchild->avl_child[right] = node;
456 AVL_SETBALANCE(node, (balance == left_heavy ? right_heavy : 0));

```

```

457     AVL_SETPARENT(node, gchild);
458     AVL_SETCHILD(node, right);

```

```

460     AVL_SETBALANCE(gchild, 0);
461     AVL_SETPARENT(gchild, parent);
462     AVL_SETCHILD(gchild, which_child);
463     if (parent != NULL)
464         parent->avl_child[which_child] = gchild;
465     else
466         tree->avl_root = gchild;

```

```

468     return (1);    /* the new tree is always shorter */
469 }

```

```

472 /*
473  * Insert a new node into an AVL tree at the specified (from avl_find()) place.
474  *
475  * Newly inserted nodes are always leaf nodes in the tree, since avl_find()
476  * searches out to the leaf positions. The avl_index_t indicates the node
477  * which will be the parent of the new node.
478  *
479  * After the node is inserted, a single rotation further up the tree may
480  * be necessary to maintain an acceptable AVL balance.
481  */
482 void
483 avl_insert(avl_tree_t *tree, void *new_data, avl_index_t where)
484 {
485     avl_node_t *node;
486     avl_node_t *parent = AVL_INDEX2NODE(where);
487     int old_balance;
488     int new_balance;
489     int which_child = AVL_INDEX2CHILD(where);
490     size_t off = tree->avl_offset;

```

```

492     ASSERT(tree);
493     #ifdef _LP64
494     ASSERT(((uintptr_t)new_data & 0x7) == 0);
495     #endif

```

```

497     node = AVL_DATA2NODE(new_data, off);

```

```

499     /*
500      * First, add the node to the tree at the indicated position.
501      */
502     ++tree->avl_numnodes;

```

```

504     node->avl_child[0] = NULL;
505     node->avl_child[1] = NULL;

```

```

507     AVL_SETCHILD(node, which_child);
508     AVL_SETBALANCE(node, 0);
509     AVL_SETPARENT(node, parent);
510     if (parent != NULL) {
511         ASSERT(parent->avl_child[which_child] == NULL);
512         parent->avl_child[which_child] = node;
513     } else {
514         ASSERT(tree->avl_root == NULL);
515         tree->avl_root = node;
516     }
517     /*
518      * Now, back up the tree modifying the balance of all nodes above the
519      * insertion point. If we get to a highly unbalanced ancestor, we
520      * need to do a rotation. If we back out of the tree we are done.
521      * If we brought any subtree into perfect balance (0), we are also done.
522      */

```

```

523     for (;;) {
524         node = parent;
525         if (node == NULL)
526             return;

528         /*
529          * Compute the new balance
530          */
531         old_balance = AVL_XBALANCE(node);
532         new_balance = old_balance + avl_child2balance[which_child];

534         /*
535          * If we introduced equal balance, then we are done immediately
536          */
537         if (new_balance == 0) {
538             AVL_SETBALANCE(node, 0);
539             return;
540         }

542         /*
543          * If both old and new are not zero we went
544          * from -1 to -2 balance, do a rotation.
545          */
546         if (old_balance != 0)
547             break;

549         AVL_SETBALANCE(node, new_balance);
550         parent = AVL_XPARENT(node);
551         which_child = AVL_XCHILD(node);
552     }

554     /*
555     * perform a rotation to fix the tree and return
556     */
557     (void) avl_rotation(tree, node, new_balance);
558 }

560 /*
561 * Insert "new_data" in "tree" in the given "direction" either after or
562 * before (AVL_AFTER, AVL_BEFORE) the data "here".
563 *
564 * Insertions can only be done at empty leaf points in the tree, therefore
565 * if the given child of the node is already present we move to either
566 * the AVL_PREV or AVL_NEXT and reverse the insertion direction. Since
567 * every other node in the tree is a leaf, this always works.
568 *
569 * To help developers using this interface, we assert that the new node
570 * is correctly ordered at every step of the way in DEBUG kernels.
571 */
572 void
573 avl_insert_here(
574     avl_tree_t *tree,
575     void *new_data,
576     void *here,
577     int direction)
578 {
579     avl_node_t *node;
580     int child = direction; /* rely on AVL_BEFORE == 0, AVL_AFTER == 1 */
581 #ifdef DEBUG
582     int diff;
583 #endif

585     ASSERT(tree != NULL);
586     ASSERT(new_data != NULL);
587     ASSERT(here != NULL);
588     ASSERT(direction == AVL_BEFORE || direction == AVL_AFTER);

```

```

590     /*
591     * If corresponding child of node is not NULL, go to the neighboring
592     * node and reverse the insertion direction.
593     */
594     node = AVL_DATA2NODE(here, tree->avl_offset);

596 #ifdef DEBUG
597     diff = tree->avl_compar(new_data, here);
598     ASSERT(-1 <= diff && diff <= 1);
599     ASSERT(diff != 0);
600     ASSERT(diff > 0 ? child == 1 : child == 0);
601 #endif

603     if (node->avl_child[child] != NULL) {
604         node = node->avl_child[child];
605         child = 1 - child;
606         while (node->avl_child[child] != NULL) {
607 #ifdef DEBUG
608             diff = tree->avl_compar(new_data,
609                 AVL_NODE2DATA(node, tree->avl_offset));
610             ASSERT(-1 <= diff && diff <= 1);
611             ASSERT(diff != 0);
612             ASSERT(diff > 0 ? child == 1 : child == 0);
613 #endif
614             node = node->avl_child[child];
615         }
616 #ifdef DEBUG
617         diff = tree->avl_compar(new_data,
618             AVL_NODE2DATA(node, tree->avl_offset));
619         ASSERT(-1 <= diff && diff <= 1);
620         ASSERT(diff != 0);
621         ASSERT(diff > 0 ? child == 1 : child == 0);
622 #endif
623     }
624     ASSERT(node->avl_child[child] == NULL);

626     avl_insert(tree, new_data, AVL_MKINDEX(node, child));
627 }

629 /*
630 * Add a new node to an AVL tree.
631 */
632 void
633 avl_add(avl_tree_t *tree, void *new_node)
634 {
635     avl_index_t where;

637     /*
638     * This is unfortunate. We want to call panic() here, even for
639     * non-DEBUG kernels. In userland, however, we can't depend on anything
640     * in libc or else the rtld build process gets confused.
641     * Thankfully, rtld provides us with its own assfail() so we can use
642     * that here. We use assfail() directly to get a nice error message
643     * in the core - much like what panic() does for crashdumps.
644     * in libc or else the rtld build process gets confused. So, all we can
645     * do in userland is resort to a normal ASSERT().
646     */
647     if (avl_find(tree, new_node, &where) != NULL)
648 #ifdef _KERNEL
649         panic("avl_find() succeeded inside avl_add()");
650 #else
651         (void) assfail("avl_find() succeeded inside avl_add()",
652             _FILE_, _LINE_);
653 #endif
654     ASSERT(0);
655 #endif

```

new/usr/src/common/avl/avl.c

11

```
652     avl_insert(tree, new_node, where);  
653 }  
_____unchanged_portion_omitted_____
```

\*\*\*\*\*

1341 Sat Aug 1 08:08:32 2015

new/usr/src/lib/libavl/Makefile.com

6091 avl\_add doesn't assert on non-debug builds

Reviewed by: Andy Stormont <astormont@racktopsystems.com>

\*\*\*\*\*

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright 2008 Sun Microsystems, Inc. All rights reserved.
23 # Use is subject to license terms.
24 #
25 # ident "%Z%M% %I% %E% SMI"
26 #

28 LIBRARY=      libavl.a
29 VERS=         .1
30 OBJECTS=      avl.o

32 include ../Makefile.lib
33 include ../Makefile.rootfs

35 LIBS =        $(DYNLIB) $(LINTLIB)
36 SRCS =        $(COMDIR)/avl.c

38 $(LINTLIB) := SRCS = $(SRCDIR)/$(LINTSRC)

40 COMDIR =      $(SRC)/common/avl

42 LDLIBS +=     -lc
43 #endif /* ! codereview */
44 CFLAGS +=     $(CCVERBOSE)

46 .KEEP_STATE:

48 all : $(LIBS)

50 lint : lintcheck

52 pics%.o:      $(COMDIR)/%.c
53               $(COMPILE.c) -o $@ $<
54               $(POST_PROCESS_O)

56 include ../Makefile.targ
```