svn.phex.phex.trunk.src.main.java.phex.common.collections.IntSet.java - svn.acqlite.AcqliteCore.components.collection.src.main.java.org.limewire.collection.IntSet.java

```java
/*
 *  PHEX - The pure-java Gnutella-servent.
 *  Copyright (C) 2001 - 2007 Phex Development Group
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 *  --- CVS Information ---
 *  $Id: IntSet.java 4229 2008-07-13 21:42:10Z gregork $
 */
package phex.common.collections;

//Modified version (1.3 2007-06-11 15:37:09) of
  /java/org/limewire/collection/IntSet.java
//Copyright by Limewire

import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Set;

/**
 * Represents a set of distinct integers.
 * Like {@link Set}, <code>IntSet</code> is <b>not synchronized</b>.
 * <p>
 * Optimized to have an extremely compact representation
 * when the set is "dense", i.e., has many sequential elements. For example {1,
 * 2} and {1, 2, ..., 1000} require the same amount of space. All retrieval
 * operations run in O(log n) time, where n is the size of the set. Insertion
 * operations may be slower.
 * <p>
 * All methods have the same specification as the <code>Set</code> class, except
 * that values are restricted to int' for the reason described  above. For
 * this reason, methods are not specified below.
 * <p>
 * This class is not thread-safe.
 * <pre>
    IntSet s = new IntSet(10);
    s.add(1); s.add(1);
    s.add(3); s.add(4); s.add(5);
    s.add(7);
    System.out.println("Set is " + s);
    s.add(2);
    System.out.println("Set is " + s);
    s.remove(3);
```

```java
package org.limewire.collection;

import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Set;

/**
 * Represents a set of distinct integers.
 * Like {@link Set}, <code>IntSet</code> is <b>not synchronized</b>.
 * <p>
 * Optimized to have an extremely compact representation
 * when the set is "dense", i.e., has many sequential elements. For example {1,
 * 2} and {1, 2, ..., 1000} require the same amount of space. All retrieval
 * operations run in O(log n) time, where n is the size of the set. Insertion
 * operations may be slower.
 * <p>
 * All methods have the same specification as the <code>Set</code> class, except
 * that values are restricted to int' for the reason described  above. For
 * this reason, methods are not specified below.
 * <p>
 * This class is not thread-safe.
 * <pre>
    IntSet s = new IntSet(10);
    s.add(1); s.add(1);
    s.add(3); s.add(4); s.add(5);
    s.add(7);
    System.out.println("Set is " + s);
    s.add(2);
    System.out.println("Set is " + s);
    s.remove(3);
```

svn.phex.phex.trunk.src.main.java.phex.common.collections.IntSet.java - svn.acqlite.AcqliteCore.components.collection.src.main.java.org.limewire.collection.IntSet.java

```
56      System.out.println("Set is " + s);
57
58      Output:
59          Set is [1, 3-5, 7]
60          Set is [1-5, 7]
61          Set is [1-2, 4-5, 7]
62
63  * </pre>
64  */
65
66  public class IntSet {
67      /**
68       * Our current implementation consists of a list of disjoint intervals,
69       * sorted by starting location. As an example, the set {1, 3, 4, 5, 7} is
70       * represented by
71       *     [1, 3-5, 7]
72       * Adding 2 turns the representation into
73       *     [1-5, 7]
74       * Note that [1-2 ,3-5, 7] is not allowed by the invariant below.
75       * Continuing with the example, removing 3 turns the rep into
76       *     [1-2, 4-5, 7]
77       *
78       * We use a sorted List instead of a TreeSet because it has a more compact
79       * memory footprint, and memory is at a premium here. It also makes
80       * implementation much easier. Unfortunately it means that insertion
81       * and some set operations are more expensive because memory must be
82       * allocated and copied.
83       *
84       * INVARIANT: for all i<j, list[i].high < (list[j].low-1)
85       */
86      private ArrayList<Interval> list;
87
88      /**
89       * The size of this.
90       *
91       * INVARIANT: size==sum over all i of (get(i).high-get(i).low+1)
92       */
93      private int size=0;
94
95
96      /** The interval from low to high, inclusive on both ends. */
97      private static class Interval {
98          /** INVARIANT: low<=high */
99          int low;
100         int high;
101         /** @requires that low<=high */
102         Interval(int low, int high) {
103             this.low=low;
104             this.high=high;
105         }
106         Interval(int singleton) {
107             this.low=singleton;
108             this.high=singleton;
109         }
110
```

```
32      System.out.println("Set is " + s);
33
34      Output:
35          Set is [1, 3-5, 7]
36          Set is [1-5, 7]
37          Set is [1-2, 4-5, 7]
38
39  * </pre>
40  */
41
42  public class IntSet {
43      /**
44       * Our current implementation consists of a list of disjoint intervals,
45       * sorted by starting location. As an example, the set {1, 3, 4, 5, 7} is
46       * represented by
47       *     [1, 3-5, 7]
48       * Adding 2 turns the representation into
49       *     [1-5, 7]
50       * Note that [1-2 ,3-5, 7] is not allowed by the invariant below.
51       * Continuing with the example, removing 3 turns the rep into
52       *     [1-2, 4-5, 7]
53       *
54       * We use a sorted List instead of a TreeSet because it has a more compact
55       * memory footprint, and memory is at a premium here. It also makes
56       * implementation much easier. Unfortunately it means that insertion
57       * and some set operations are more expensive because memory must be
58       * allocated and copied.
59       *
60       * INVARIANT: for all i<j, list[i].high < (list[j].low-1)
61       */
62      private ArrayList<Interval> list;
63
64      /**
65       * The size of this.
66       *
67       * INVARIANT: size==sum over all i of (get(i).high-get(i).low+1)
68       */
69      private int size=0;
70
71
72      /** The interval from low to high, inclusive on both ends. */
73      private static class Interval {
74          /** INVARIANT: low<=high */
75          int low;
76          int high;
77          /** @requires that low<=high */
78          Interval(int low, int high) {
79              this.low=low;
80              this.high=high;
81          }
82          Interval(int singleton) {
83              this.low=singleton;
84              this.high=singleton;
85          }
86
87          @Override
```

svn.phex.phex.trunk.src.main.java.phex.common.collections.IntSet.java - svn.acqlite.AcqliteCore.components.collection.src.main.java.org.limewire.collection.IntSet.java

```
111         public String toString() {
112             if (low==high)
113                 return String.valueOf(low);
114             else
115                 return String.valueOf(low)+"-"+String.valueOf(high);
116         }
117     }
118 
119 
120     /** Checks rep invariant. */
121     protected void repOk() {
122         if (list.size()<2)
123             return;
124 
125         int countedSize=0;
126         for (int i=0; i<(list.size()-1); i++) {
127             Interval lower=get(i);
128             countedSize+=(lower.high-lower.low+1);
129             Interval higher=get(i+1);
130             assert lower.low<=lower.high :
131                         "Backwards interval: "+toString();
132             assert lower.high<(higher.low-1) :
133                         "Touching intervals: "+toString();
134         }
135 
136         //Don't forget to check last interval.
137         Interval last=get(list.size()-1);
138         assert last.low<=last.high :
139                     "Backwards interval: "+toString();
140         countedSize+=(last.high-last.low+1);
141 
142         assert countedSize==size :
143                     "Bad size.  Should be "+countedSize+" not "+size;
144     }
145 
146     /** Returns the i'th Interval in this. */
147     private final Interval get(int i) {
148         return list.get(i);
149     }
150 
151 
152     /**
153      * Returns the largest i s.t. list[i].low<=x, or -1 if no such i.
154      * Note that x may or may not overlap the interval list[i].<p>
155      *
156      * This method uses binary search and runs in O(log N) time, where
157      * N=list.size().  The standard Java binary search methods could not
158      * be used because they only return exact matches.  Also, they require
159      * allocating a dummy Interval to represent x.
160      */
161     private final int search(int x) {
162         int low=0;
163         int high=list.size()-1;
164 
165         while (low<=high) {
166             int i=(low+high)/2;
```

```
88          public String toString() {
89              if (low==high)
90                  return String.valueOf(low);
91              else
92                  return String.valueOf(low)+"-"+String.valueOf(high);
93          }
94      }
95 
96 
97      /** Checks rep invariant. */
98      protected void repOk() {
99          if (list.size()<2)
100             return;
101 
102         int countedSize=0;
103         for (int i=0; i<(list.size()-1); i++) {
104             Interval lower=get(i);
105             countedSize+=(lower.high-lower.low+1);
106             Interval higher=get(i+1);
107             assert lower.low<=lower.high :
108                         "Backwards interval: "+toString();
109             assert lower.high<(higher.low-1) :
110                         "Touching intervals: "+toString();
111         }
112 
113         //Don't forget to check last interval.
114         Interval last=get(list.size()-1);
115         assert last.low<=last.high :
116                     "Backwards interval: "+toString();
117         countedSize+=(last.high-last.low+1);
118 
119         assert countedSize==size :
120                     "Bad size.  Should be "+countedSize+" not "+size;
121     }
122 
123     /** Returns the i'th Interval in this. */
124     private final Interval get(int i) {
125         return list.get(i);
126     }
127 
128 
129     /**
130      * Returns the largest i s.t. list[i].low<=x, or -1 if no such i.
131      * Note that x may or may not overlap the interval list[i].<p>
132      *
133      * This method uses binary search and runs in O(log N) time, where
134      * N=list.size().  The standard Java binary search methods could not
135      * be used because they only return exact matches.  Also, they require
136      * allocating a dummy Interval to represent x.
137      */
138     private final int search(int x) {
139         int low=0;
140         int high=list.size()-1;
141 
142         while (low<=high) {
143             int i=(low+high)/2;
```

svn.phex.phex.trunk.src.main.java.phex.common.collections.IntSet.java - svn.acqlite.AcqliteCore.components.collection.src.main.java.org.limewire.collection.IntSet.java

```
            int li=get(i).low;

            if (li<x)
                low=i+1;
            else if (x<li)
                high=i-1;
            else
                return i;
        }

        //Remarkably, this does the right thing.
        return high;
    }


    ////////////////////// Set-like Public Methods //////////////////////////

    public IntSet() {
        this.list=new ArrayList<Interval>();
    }

    public IntSet(int expectedSize) {
        this.list=new ArrayList<Interval>(expectedSize);
    }

    public int size() {
        return this.size;
    }

    public boolean contains(int x) {
        int i=search(x);
        if (i==-1)
            return false;

        Interval li=get(i);
        assert  li.low<=x : "Bad return value from search.";
        if (x<=li.high)
            return true;
        else
            return false;
    }


    public boolean add(int x) {
        //This code is a pain--nine different return cases.  It could be
        //factored somewhat, but I believe the following is the easiest to
        //understand.  The cases are illustrated to the right.
        int i=search(x);

        //Optimistically increment size.  Decrement it later if needed.
        size++;

        //Add x to beginning of list
        if (i==-1) {
            if ( list.size()==0 || x<(get(0).low-1) ) {
                //1. Add [x, x] to beginning of list.       x ---
```

```
            int li=get(i).low;

            if (li<x)
                low=i+1;
            else if (x<li)
                high=i-1;
            else
                return i;
        }

        //Remarkably, this does the right thing.
        return high;
    }


    ////////////////////// Set-like Public Methods //////////////////////////

    public IntSet() {
        this.list=new ArrayList<Interval>();
    }

    public IntSet(int expectedSize) {
        this.list=new ArrayList<Interval>(expectedSize);
    }

    public int size() {
        return this.size;
    }

    public boolean contains(int x) {
        int i=search(x);
        if (i==-1)
            return false;

        Interval li=get(i);
        assert  li.low<=x : "Bad return value from search.";
        if (x<=li.high)
            return true;
        else
            return false;
    }


    public boolean add(int x) {
        //This code is a pain--nine different return cases.  It could be
        //factored somewhat, but I believe the following is the easiest to
        //understand.  The cases are illustrated to the right.
        int i=search(x);

        //Optimistically increment size.  Decrement it later if needed.
        size++;

        //Add x to beginning of list
        if (i==-1) {
            if ( list.size()==0 || x<(get(0).low-1) ) {
                //1. Add [x, x] to beginning of list.       x ---
```

svn.phex.phex.trunk.src.main.java.phex.common.collections.IntSet.java - svn.acqlite.AcqliteCore.components.collection.src.main.java.org.limewire.collection.IntSet.java

```
                list.add(0, new Interval(x));                                   
                return true;                                                    
            } else {                                                            
                //2. Merge x with beginning of list.        x----               
                get(0).low=x;                                                   
                return true;                                                    
            }                                                                   
        }                                                                       
                                                                                
        Interval lower=get(i);                                                  
        assert(lower.low<=x);                                                   
        if (x<=lower.high) {                                                    
            //3. x already in this.                       --x--               
            size--;    //Undo previous increment.                               
            return false;                                                       
        }                                                                       
                                                                                
        //Adding x to end of the list.                                          
        if (i==(list.size()-1)) {                                               
            if (lower.high < (x-1)) {                                           
                //4. Add x to end of list                   --- x               
                list.add(new Interval(x));                                      
                return true;                                                    
            } else {                                                            
                //5. Merge x with end of list               ----x               
                lower.high=x;                                                   
                return true;                                                    
            }                                                                   
        }                                                                       
                                                                                
        //Adding x to middle of the list                                        
        Interval higher=get(i+1);                                               
        boolean touchesLower=(lower.high==(x-1));                               
        boolean touchesHigher=(x==(higher.low-1));                              
        if (touchesLower) {                                                     
            if (touchesHigher) {                                                
                //6. Merge lower and higher intervals       --x--               
                lower.high=higher.high;                                         
                list.remove(i+1);                                               
                return true;                                                    
            } else {                                                            
                //7. Merge with lower interval              --x --              
                lower.high=x;                                                   
                return true;                                                    
            }                                                                   
        } else {                                                                
            if (touchesHigher) {                                                
                //8. Merge with higher interval             -- x--              
                higher.low=x;                                                   
                return true;                                                    
            } else {                                                            
                //9. Insert as new element                  -- x --             
                list.add(i+1, new Interval(x));                                 
                return true;                                                    
            }                                                                   
        }                                                                       
```

```
                list.add(0, new Interval(x));                                   
                return true;                                                    
            } else {                                                            
                //2. Merge x with beginning of list.        x----               
                get(0).low=x;                                                   
                return true;                                                    
            }                                                                   
        }                                                                       
                                                                                
        Interval lower=get(i);                                                  
        assert(lower.low<=x);                                                   
        if (x<=lower.high) {                                                    
            //3. x already in this.                       --x--               
            size--;    //Undo previous increment.                               
            return false;                                                       
        }                                                                       
                                                                                
        //Adding x to end of the list.                                          
        if (i==(list.size()-1)) {                                               
            if (lower.high < (x-1)) {                                           
                //4. Add x to end of list                   --- x               
                list.add(new Interval(x));                                      
                return true;                                                    
            } else {                                                            
                //5. Merge x with end of list               ----x               
                lower.high=x;                                                   
                return true;                                                    
            }                                                                   
        }                                                                       
                                                                                
        //Adding x to middle of the list                                        
        Interval higher=get(i+1);                                               
        boolean touchesLower=(lower.high==(x-1));                               
        boolean touchesHigher=(x==(higher.low-1));                              
        if (touchesLower) {                                                     
            if (touchesHigher) {                                                
                //6. Merge lower and higher intervals       --x--               
                lower.high=higher.high;                                         
                list.remove(i+1);                                               
                return true;                                                    
            } else {                                                            
                //7. Merge with lower interval              --x --              
                lower.high=x;                                                   
                return true;                                                    
            }                                                                   
        } else {                                                                
            if (touchesHigher) {                                                
                //8. Merge with higher interval             -- x--              
                higher.low=x;                                                   
                return true;                                                    
            } else {                                                            
                //9. Insert as new element                  -- x --             
                list.add(i+1, new Interval(x));                                 
                return true;                                                    
            }                                                                   
        }                                                                       
```

svn.phex.phex.trunk.src.main.java.phex.common.collections.IntSet.java - svn.acqlite.AcqliteCore.components.collection.src.main.java.org.limewire.collection.IntSet.java

```
279     }
280
281
282     public boolean remove(int x) {
283         //Find the interval overlapping x.
284         int i=search(x);
285         if (i==-1 || x>get(i).high)
286             //1. x not in this.                          ----
287             return false;
288
289         Interval interval=get(i);
290         boolean touchesLow=(interval.low==x);
291         boolean touchesHigh=(interval.high==x);
292         if (touchesLow) {
293             if (touchesHigh) {
294                 //2. Singleton interval.  Remove.        -- x --
295                 list.remove(i);
296             }
297             else {
298                 //3. Modify low end.                     x---
299                 interval.low++;
300             }
301         } else {
302             if (touchesHigh) {
303                 //4. Modify high end.                    ---x
304                 interval.high--;
305             } else {
306                 //5. Split entire interval.              --x--
307                 Interval newInterval=new Interval(x+1, interval.high);
308                 interval.high=x-1;
309                 list.add(i+1, newInterval);
310             }
311         }
312         size--;
313         return true;
314     }
315
316
317     public boolean addAll(IntSet s) {
318         //TODO2: implement more efficiently!
319         boolean ret=false;
320         for (IntSetIterator iter=s.iterator(); iter.hasNext(); ) {
321             ret=(ret | this.add(iter.next()));
322         }
323         return ret;
324     }
325
326
327     public boolean retainAll(IntSet s) {
328         //We can't modify this while iterating over it, so we need to
329         //maintain an external list of items that must go.
330         //TODO2: implement more efficiently!
331         List<Integer> removeList = new ArrayList<Integer>();
332         for (IntSetIterator iter = this.iterator(); iter.hasNext(); ) {
333             int x = iter.next();
334             if (! s.contains(x))
```

```
256     }
257
258
259     public boolean remove(int x) {
260         //Find the interval overlapping x.
261         int i=search(x);
262         if (i==-1 || x>get(i).high)
263             //1. x not in this.                          ----
264             return false;
265
266         Interval interval=get(i);
267         boolean touchesLow=(interval.low==x);
268         boolean touchesHigh=(interval.high==x);
269         if (touchesLow) {
270             if (touchesHigh) {
271                 //2. Singleton interval.  Remove.        -- x --
272                 list.remove(i);
273             }
274             else {
275                 //3. Modify low end.                     x---
276                 interval.low++;
277             }
278         } else {
279             if (touchesHigh) {
280                 //4. Modify high end.                    ---x
281                 interval.high--;
282             } else {
283                 //5. Split entire interval.              --x--
284                 Interval newInterval=new Interval(x+1, interval.high);
285                 interval.high=x-1;
286                 list.add(i+1, newInterval);
287             }
288         }
289         size--;
290         return true;
291     }
292
293
294     public boolean addAll(IntSet s) {
295         //TODO2: implement more efficiently!
296         boolean ret=false;
297         for (IntSetIterator iter=s.iterator(); iter.hasNext(); ) {
298             ret=(ret | this.add(iter.next()));
299         }
300         return ret;
301     }
302
303
304     public boolean retainAll(IntSet s) {
305         //We can't modify this while iterating over it, so we need to
306         //maintain an external list of items that must go.
307         //TODO2: implement more efficiently!
308         List<Integer> removeList = new ArrayList<Integer>();
309         for (IntSetIterator iter = this.iterator(); iter.hasNext(); ) {
310             int x = iter.next();
311             if (! s.contains(x))
```

svn.phex.phex.trunk.src.main.java.phex.common.collections.IntSet.java - svn.acqlite.AcqliteCore.components.collection.src.main.java.org.limewire.collection.IntSet.java

```
335                 removeList.add(Integer.valueOf(x));
336         }
337         //It's marginally more efficient to remove items from end to beginning.
338         for (int i=removeList.size()-1; i>=0; i--) {
339             int x = (removeList.get(i)).intValue();
340             this.remove(x);
341         }
342         //Did we remove any items?
343         return removeList.size()>0;
344     }
345
346
347     /** Ensures that this consumes the minimum amount of memory. This method
348      *  should typically be called after the last call to add(..). Insertions
349      *  can still be done after the call, but they might be slower.
350      *
351      *  Because this method only affects the performance of this, there
352      *  is no modifies clause listed.  */
353     public void trim() {
354         list.trimToSize();
355     }
356
357
358     /**
359      * Returns the values of this in order from lowest to highest, as int.
360      *     @requires this not modified while iterator in use
361      */
362     public IntSetIterator iterator() {
363         return new IntSetIterator();
364     }
365
366     /** Yields a sequence of int's (not Object's) in order, without removal
367      *  support.  Otherwise exactly like an Iterator. */
368     public class IntSetIterator {
369         /** The next interval to yield */
370         private int i;
371         /** The next element to yield, from the i'th interval, or undefined
372          *  if there are no more intervals to yield.
373          *  INVARIANT: i<list.size() ==> get(i).low<=next<=get(i).high */
374         private int next;
375
376         private IntSetIterator() {
377             i=0;
378             if (i<list.size())
379                 next=get(i).low;
380         }
381
382         public boolean hasNext() {
383             return i<list.size();
384         }
385
386         public int next() throws NoSuchElementException {
387             if (! hasNext())
388                 throw new NoSuchElementException();
389
390             int ret=next;
```

```
312                 removeList.add(new Integer(x));
313         }
314         //It's marginally more efficient to remove items from end to beginning.
315         for (int i=removeList.size()-1; i>=0; i--) {
316             int x = (removeList.get(i)).intValue();
317             this.remove(x);
318         }
319         //Did we remove any items?
320         return removeList.size()>0;
321     }
322
323
324     /** Ensures that this consumes the minimum amount of memory. This method
325      *  should typically be called after the last call to add(..). Insertions
326      *  can still be done after the call, but they might be slower.
327      *
328      *  Because this method only affects the performance of this, there
329      *  is no modifies clause listed.  */
330     public void trim() {
331         list.trimToSize();
332     }
333
334
335     /**
336      * Returns the values of this in order from lowest to highest, as int.
337      *     @requires this not modified while iterator in use
338      */
339     public IntSetIterator iterator() {
340         return new IntSetIterator();
341     }
342
343     /** Yields a sequence of int's (not Object's) in order, without removal
344      *  support.  Otherwise exactly like an Iterator. */
345     public class IntSetIterator {
346         /** The next interval to yield */
347         private int i;
348         /** The next element to yield, from the i'th interval, or undefined
349          *  if there are no more intervals to yield.
350          *  INVARIANT: i<list.size() ==> get(i).low<=next<=get(i).high */
351         private int next;
352
353         private IntSetIterator() {
354             i=0;
355             if (i<list.size())
356                 next=get(i).low;
357         }
358
359         public boolean hasNext() {
360             return i<list.size();
361         }
362
363         public int next() throws NoSuchElementException {
364             if (! hasNext())
365                 throw new NoSuchElementException();
366
367             int ret=next;
```

svn.phex.phex.trunk.src.main.java.phex.common.collections.IntSet.java - svn.acqlite.AcqliteCore.components.collection.src.main.java.org.limewire.collection.IntSet.java

```
391             next++;
392             if (next>get(i).high) {
393                 //Advance to next interval.
394                 i++;
395                 if (i<list.size())
396                     next=get(i).low;
397             }
398             return ret;
399         }
400     }
401
402
403     public String toString() {
404         return list.toString();
405     }
406}
```

```
368             next++;
369             if (next>get(i).high) {
370                 //Advance to next interval.
371                 i++;
372                 if (i<list.size())
373                     next=get(i).low;
374             }
375             return ret;
376         }
377     }
378
379
380     @Override
381     public String toString() {
382         return list.toString();
383     }
384}
385
```