001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.collections4.trie;  
018  
019import java.io.Serializable;  
020import java.util.Comparator;  
021  
022/\*\*  
023 \* Defines the interface to analyze {@link org.apache.commons.collections4.Trie Trie} keys on a bit level.  
024 \* {@link KeyAnalyzer}'s methods return the length of the key in bits, whether or not a bit is set,  
025 \* and bits per element in the key.  
026 \* <p>  
027 \* Additionally, a method determines if a key is a prefix of another  
028 \* key and returns the bit index where one key is different from another  
029 \* key (if the key and found key are equal than the return value is  
030 \* {@link #EQUAL\_BIT\_KEY}).  
031 \* </p>  
032 \*  
033 \* @param <K> the type of objects that may be compared by this analyzer  
034 \* @since 4.0  
035 \*/  
036public abstract class KeyAnalyzer<K> implements Comparator<K>, Serializable {  
037  
038 /\*\* Serialization version \*/  
039 private static final long serialVersionUID = -20497563720380683L;  
040  
041 /\*\*  
042 \* Returned by {@link #bitIndex(Object, int, int, Object, int, int)}  
043 \* if key's bits are all 0.  
044 \*/  
045 public static final int NULL\_BIT\_KEY = -1;  
046  
047 /\*\*  
048 \* Returned by {@link #bitIndex(Object, int, int, Object, int, int)} if key and found key are equal.  
049 \* This is a very very specific case and shouldn't happen on a regular basis.  
050 \*/  
051 public static final int EQUAL\_BIT\_KEY = -2;  
052  
053 public static final int OUT\_OF\_BOUNDS\_BIT\_KEY = -3;  
054  
055 /\*\*  
056 \* Returns true if bitIndex is a {@link KeyAnalyzer#OUT\_OF\_BOUNDS\_BIT\_KEY}.  
057 \*/  
058 static boolean isOutOfBoundsIndex(final int bitIndex) {  
059 return bitIndex == OUT\_OF\_BOUNDS\_BIT\_KEY;  
060 }  
061  
062 /\*\*  
063 \* Returns true if bitIndex is a {@link KeyAnalyzer#EQUAL\_BIT\_KEY}.  
064 \*/  
065 static boolean isEqualBitKey(final int bitIndex) {  
066 return bitIndex == EQUAL\_BIT\_KEY;  
067 }  
068  
069 /\*\*  
070 \* Returns true if bitIndex is a {@link KeyAnalyzer#NULL\_BIT\_KEY}.  
071 \*/  
072 static boolean isNullBitKey(final int bitIndex) {  
073 return bitIndex == NULL\_BIT\_KEY;  
074 }  
075  
076 /\*\*  
077 \* Returns true if the given bitIndex is valid.  
078 \* Indices are considered valid if they're between 0 and {@link Integer#MAX\_VALUE}  
079 \*/  
080 static boolean isValidBitIndex(final int bitIndex) {  
081 return bitIndex >= 0;  
082 }  
083  
084 /\*\*  
085 \* Returns the number of bits per element in the key.  
086 \* This is only useful for variable-length keys, such as Strings.  
087 \*  
088 \* @return the number of bits per element  
089 \*/  
090 public abstract int bitsPerElement();  
091  
092 /\*\*  
093 \* Returns the length of the Key in bits.  
094 \*  
095 \* @param key the key  
096 \* @return the bit length of the key  
097 \*/  
098 public abstract int lengthInBits(K key);  
099  
100 /\*\*  
101 \* Returns whether or not a bit is set.  
102 \*  
103 \* @param key the key to check, may not be null  
104 \* @param bitIndex the bit index to check  
105 \* @param lengthInBits the maximum key length in bits to check  
106 \* @return {@code true} if the bit is set in the given key and  
107 \* {@code bitIndex} < {@code lengthInBits}, {@code false} otherwise.  
108 \*/  
109 public abstract boolean isBitSet(K key, int bitIndex, int lengthInBits);  
110  
111 /\*\*  
112 \* Returns the n-th different bit between key and other. This starts the comparison in  
113 \* key at 'offsetInBits' and goes for 'lengthInBits' bits, and compares to the other key starting  
114 \* at 'otherOffsetInBits' and going for 'otherLengthInBits' bits.  
115 \*  
116 \* @param key the key to use  
117 \* @param offsetInBits the bit offset in the key  
118 \* @param lengthInBits the maximum key length in bits to use  
119 \* @param other the other key to use  
120 \* @param otherOffsetInBits the bit offset in the other key  
121 \* @param otherLengthInBits the maximum key length in bits for the other key  
122 \* @return the bit index where the key and other first differ  
123 \*/  
124 public abstract int bitIndex(K key, int offsetInBits, int lengthInBits,  
125 K other, int otherOffsetInBits, int otherLengthInBits);  
126  
127 /\*\*  
128 \* Determines whether or not the given prefix (from offset to length) is a prefix of the given key.  
129 \*  
130 \* @param prefix the prefix to check  
131 \* @param offsetInBits the bit offset in the key  
132 \* @param lengthInBits the maximum key length in bits to use  
133 \* @param key the key to check  
134 \* @return {@code true} if this is a valid prefix for the given key  
135 \*/  
136 public abstract boolean isPrefix(K prefix, int offsetInBits, int lengthInBits, K key);  
137  
138 @Override  
139 @SuppressWarnings("unchecked")  
140 public int compare(final K o1, final K o2) {  
141 if (o1 == null) {  
142 return o2 == null ? 0 : -1;  
143 } else if (o2 == null) {  
144 return 1;  
145 }  
146  
147 return ((Comparable<K>) o1).compareTo(o2);  
148 }  
149  
150}