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.list;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.Collection;  
024  
025/\*\*  
026 \* A <code>List</code> implementation that stores a cache of internal Node objects  
027 \* in an effort to reduce wasteful object creation.  
028 \* <p>  
029 \* A linked list creates one Node for each item of data added. This can result in  
030 \* a lot of object creation and garbage collection. This implementation seeks to  
031 \* avoid that by maintaining a store of cached nodes.  
032 \* </p>  
033 \* <p>  
034 \* This implementation is suitable for long-lived lists where both add and remove  
035 \* are used. Short-lived lists, or lists which only grow will have worse performance  
036 \* using this class.  
037 \* </p>  
038 \* <p>  
039 \* <b>Note that this implementation is not synchronized.</b>  
040 \* </p>  
041 \*  
042 \* @since 3.0  
043 \*/  
044public class NodeCachingLinkedList<E> extends AbstractLinkedList<E> implements Serializable {  
045  
046 /\*\* Serialization version \*/  
047 private static final long serialVersionUID = 6897789178562232073L;  
048  
049 /\*\*  
050 \* The default value for {@link #maximumCacheSize}.  
051 \*/  
052 private static final int DEFAULT\_MAXIMUM\_CACHE\_SIZE = 20;  
053  
054 /\*\*  
055 \* The first cached node, or <code>null</code> if no nodes are cached.  
056 \* Cached nodes are stored in a singly-linked list with  
057 \* <code>next</code> pointing to the next element.  
058 \*/  
059 private transient Node<E> firstCachedNode;  
060  
061 /\*\*  
062 \* The size of the cache.  
063 \*/  
064 private transient int cacheSize;  
065  
066 /\*\*  
067 \* The maximum size of the cache.  
068 \*/  
069 private int maximumCacheSize;  
070  
071 //-----------------------------------------------------------------------  
072 /\*\*  
073 \* Constructor that creates.  
074 \*/  
075 public NodeCachingLinkedList() {  
076 this(DEFAULT\_MAXIMUM\_CACHE\_SIZE);  
077 }  
078  
079 /\*\*  
080 \* Constructor that copies the specified collection  
081 \*  
082 \* @param coll the collection to copy  
083 \*/  
084 public NodeCachingLinkedList(final Collection<? extends E> coll) {  
085 super(coll);  
086 this.maximumCacheSize = DEFAULT\_MAXIMUM\_CACHE\_SIZE;  
087 }  
088  
089 /\*\*  
090 \* Constructor that species the maximum cache size.  
091 \*  
092 \* @param maximumCacheSize the maximum cache size  
093 \*/  
094 public NodeCachingLinkedList(final int maximumCacheSize) {  
095 super();  
096 this.maximumCacheSize = maximumCacheSize;  
097 init(); // must call init() as use super();  
098 }  
099  
100 //-----------------------------------------------------------------------  
101 /\*\*  
102 \* Gets the maximum size of the cache.  
103 \*  
104 \* @return the maximum cache size  
105 \*/  
106 protected int getMaximumCacheSize() {  
107 return maximumCacheSize;  
108 }  
109  
110 /\*\*  
111 \* Sets the maximum size of the cache.  
112 \*  
113 \* @param maximumCacheSize the new maximum cache size  
114 \*/  
115 protected void setMaximumCacheSize(final int maximumCacheSize) {  
116 this.maximumCacheSize = maximumCacheSize;  
117 shrinkCacheToMaximumSize();  
118 }  
119  
120 /\*\*  
121 \* Reduce the size of the cache to the maximum, if necessary.  
122 \*/  
123 protected void shrinkCacheToMaximumSize() {  
124 // Rich Dougherty: This could be more efficient.  
125 while (cacheSize > maximumCacheSize) {  
126 getNodeFromCache();  
127 }  
128 }  
129  
130 /\*\*  
131 \* Gets a node from the cache. If a node is returned, then the value of  
132 \* {@link #cacheSize} is decreased accordingly. The node that is returned  
133 \* will have <code>null</code> values for next, previous and element.  
134 \*  
135 \* @return a node, or <code>null</code> if there are no nodes in the cache.  
136 \*/  
137 protected Node<E> getNodeFromCache() {  
138 if (cacheSize == 0) {  
139 return null;  
140 }  
141 final Node<E> cachedNode = firstCachedNode;  
142 firstCachedNode = cachedNode.next;  
143 cachedNode.next = null; // This should be changed anyway, but defensively  
144 // set it to null.  
145 cacheSize--;  
146 return cachedNode;  
147 }  
148  
149 /\*\*  
150 \* Checks whether the cache is full.  
151 \*  
152 \* @return true if the cache is full  
153 \*/  
154 protected boolean isCacheFull() {  
155 return cacheSize >= maximumCacheSize;  
156 }  
157  
158 /\*\*  
159 \* Adds a node to the cache, if the cache isn't full.  
160 \* The node's contents are cleared to so they can be garbage collected.  
161 \*  
162 \* @param node the node to add to the cache  
163 \*/  
164 protected void addNodeToCache(final Node<E> node) {  
165 if (isCacheFull()) {  
166 // don't cache the node.  
167 return;  
168 }  
169 // clear the node's contents and add it to the cache.  
170 final Node<E> nextCachedNode = firstCachedNode;  
171 node.previous = null;  
172 node.next = nextCachedNode;  
173 node.setValue(null);  
174 firstCachedNode = node;  
175 cacheSize++;  
176 }  
177  
178 //-----------------------------------------------------------------------  
179 /\*\*  
180 \* Creates a new node, either by reusing one from the cache or creating  
181 \* a new one.  
182 \*  
183 \* @param value value of the new node  
184 \* @return the newly created node  
185 \*/  
186 @Override  
187 protected Node<E> createNode(final E value) {  
188 final Node<E> cachedNode = getNodeFromCache();  
189 if (cachedNode == null) {  
190 return super.createNode(value);  
191 }  
192 cachedNode.setValue(value);  
193 return cachedNode;  
194 }  
195  
196 /\*\*  
197 \* Removes the node from the list, storing it in the cache for reuse  
198 \* if the cache is not yet full.  
199 \*  
200 \* @param node the node to remove  
201 \*/  
202 @Override  
203 protected void removeNode(final Node<E> node) {  
204 super.removeNode(node);  
205 addNodeToCache(node);  
206 }  
207  
208 /\*\*  
209 \* Removes all the nodes from the list, storing as many as required in the  
210 \* cache for reuse.  
211 \*  
212 \*/  
213 @Override  
214 protected void removeAllNodes() {  
215 // Add the removed nodes to the cache, then remove the rest.  
216 // We can add them to the cache before removing them, since  
217 // {@link AbstractLinkedList.removeAllNodes()} removes the  
218 // nodes by removing references directly from {@link #header}.  
219 final int numberOfNodesToCache = Math.min(size, maximumCacheSize - cacheSize);  
220 Node<E> node = header.next;  
221 for (int currentIndex = 0; currentIndex < numberOfNodesToCache; currentIndex++) {  
222 final Node<E> oldNode = node;  
223 node = node.next;  
224 addNodeToCache(oldNode);  
225 }  
226 super.removeAllNodes();  
227 }  
228  
229 //-----------------------------------------------------------------------  
230 /\*\*  
231 \* Serializes the data held in this object to the stream specified.  
232 \*  
233 \* @param out the output stream  
234 \* @throws IOException if an error occurs while writing to the stream  
235 \*/  
236 private void writeObject(final ObjectOutputStream out) throws IOException {  
237 out.defaultWriteObject();  
238 doWriteObject(out);  
239 }  
240  
241 /\*\*  
242 \* Deserializes the data held in this object to the stream specified.  
243 \*  
244 \* @param in the input stream  
245 \* @throws IOException if an error occurs while reading from the stream  
246 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
247 \*/  
248 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
249 in.defaultReadObject();  
250 doReadObject(in);  
251 }  
252  
253}