Binary Heap Queue : Heaps « Collections Data Structure « Java

Binary Heap Queue

```
/*
 * Copyright 2005 JBoss Inc
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Comparator;
import java.util.NoSuchElementException;

public class BinaryHeapQueue implements Queue, Externalizable {
  /** The default capacity for a binary heap. */
  private final static int DEFAULT_CAPACITY = 13;

  /** The comparator used to order the elements */
  private Comparator comparator;

  /** The number of elements currently in this heap. */
  private int size;

  /** The elements in this heap. */
  private Queueable[] elements;

  public BinaryHeapQueue() {

  }

  /**
   * Constructs a new <code>BinaryHeap</code> that will use the given
   * comparator to order its elements.
   *
   * @param comparator
   *          the comparator used to order the elements, null means use natural
   *          order
   */
  public BinaryHeapQueue(final Comparator comparator) {
    this(comparator, BinaryHeapQueue.DEFAULT_CAPACITY);
  }

  /**
   * Constructs a new <code>BinaryHeap</code>.
   *
   * @param comparator
   *          the comparator used to order the elements, null means use natural
   *          order
   * @param capacity
   *          the initial capacity for the heap
   * @throws IllegalArgumentException
   *           if <code>capacity</code> is &lt;= <code>0</code>
   */
  public BinaryHeapQueue(final Comparator comparator, final int capacity) {
    if (capacity <= 0) {
      throw new IllegalArgumentException("invalid capacity");
    }

    // +1 as 0 is noop
    this.elements = new Queueable[capacity + 1];
    this.comparator = comparator;
  }

  // ------------------------------------------------------------------------
  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    comparator = (Comparator) in.readObject();
    elements = (Queueable[]) in.readObject();
    size = in.readInt();
```

```
  }

  public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(comparator);
    out.writeObject(elements);
    out.writeInt(size);
  }

  /**
   * Clears all elements from queue.
   */
  public void clear() {
    this.elements = new Queueable[this.elements.length]; // for gc
    this.size = 0;
  }

  /**
   * Tests if queue is empty.
   *
   * @return <code>true</code> if queue is empty; <code>false</code>
   *         otherwise.
   */
  public boolean isEmpty() {
    return this.size == 0;
  }

  /**
   * Tests if queue is full.
   *
   * @return <code>true</code> if queue is full; <code>false</code>
   *         otherwise.
   */
  public boolean isFull() {
    // +1 as Queueable 0 is noop
    return this.elements.length == this.size + 1;
  }

  /**
   * Returns the number of elements in this heap.
   *
   * @return the number of elements in this heap
   */
  public int size() {
    return this.size;
  }

  /**
   * Inserts an Queueable into queue.
   *
   * @param element
   *          the Queueable to be inserted
   */
  public synchronized void enqueue(final Queueable element) {
    if (isFull()) {
      grow();
    }

    percolateUpMinHeap(element);
  }

  /**
   * Returns the Queueable on top of heap and remove it.
   *
   * @return the Queueable at top of heap
   * @throws NoSuchElementException
   *           if <code>isEmpty() == true</code>
   */
  public synchronized Queueable dequeue() throws NoSuchElementException {
    if (isEmpty()) {
      return null;
    }

    final Queueable result = this.elements[1];
    result.dequeue();

    // Code bellow was removed because it is already executed
    // inside result.dequeue()
    //
    // setElement(1, this.elements[this.size--]);
    // this.elements[this.size + 1] = null;
    //
```

```
  // if (this.size != 0) {
  // percolateDownMinHeap(1);
  // }

  return result;
}

/**
 *
 * @param index
 */
public synchronized Queueable dequeue(final int index) {
  if (index < 1 || index > this.size) {
    // throw new NoSuchElementException();
    return null;
  }

  final Queueable result = this.elements[index];
  setElement(index, this.elements[this.size]);
  this.elements[this.size] = null;
  this.size--;
  if (this.size != 0 && index <= this.size) {
    int compareToParent = 0;
    if (index > 1) {
      compareToParent = compare(this.elements[index], this.elements[index / 2]);
    }
    if (index > 1 && compareToParent < 0) {
      percolateUpMinHeap(index);
    } else {
      percolateDownMinHeap(index);
    }
  }

  return result;
}

/**
 * Percolates Queueable down heap from the position given by the index. <p/>
 * Assumes it is a minimum heap.
 *
 * @param index
 *          the index for the Queueable
 */
private void percolateDownMinHeap(final int index) {
  final Queueable element = this.elements[index];
  int hole = index;

  while ((hole * 2) <= this.size) {
    int child = hole * 2;

    // if we have a right child and that child can not be percolated
    // up then move onto other child
    if (child != this.size && compare(this.elements[child + 1], this.elements[child]) < 0) {
      child++;
    }

    // if we found resting place of bubble then terminate search
    if (compare(this.elements[child], element) >= 0) {
      break;
    }

    setElement(hole, this.elements[child]);
    hole = child;
  }

  setElement(hole, element);
}

/**
 * Percolates Queueable up heap from the position given by the index. <p/>
 * Assumes it is a minimum heap.
 *
 * @param index
 *          the index of the Queueable to be percolated up
 */
private void percolateUpMinHeap(final int index) {
  int hole = index;
  final Queueable element = this.elements[hole];
  while (hole > 1 && compare(element, this.elements[hole / 2]) < 0) {
    // save Queueable that is being pushed down
    // as the Queueable "bubble" is percolated up
```

```
      final int next = hole / 2;
      setElement(hole, this.elements[next]);
      hole = next;
    }
    setElement(hole, element);
  }

  /**
   * Percolates a new Queueable up heap from the bottom. <p/> Assumes it is a
   * minimum heap.
   *
   * @param element
   *          the Queueable
   */
  private void percolateUpMinHeap(final Queueable element) {
    setElement(++this.size, element);
    percolateUpMinHeap(this.size);
  }

  /**
   * Compares two objects using the comparator if specified, or the natural
   * order otherwise.
   *
   * @param a
   *          the first object
   * @param b
   *          the second object
   * @return -ve if a less than b, 0 if they are equal, +ve if a greater than b
   */
  private int compare(final Queueable a, final Queueable b) {
    return this.comparator.compare(a, b);
  }

  /**
   * Increases the size of the heap to support additional elements
   */
  private void grow() {
    final Queueable[] elements = new Queueable[this.elements.length * 2];
    System.arraycopy(this.elements, 0, elements, 0, this.elements.length);
    this.elements = elements;
  }

  /**
   *
   * @param index
   * @param element
   */
  private void setElement(final int index, final Queueable element) {
    this.elements[index] = element;
    element.enqueued(this, index);
  }

  public Queueable[] getQueueable() {
    return this.elements;
  }

  public Object[] toArray() {
    final Object[] result = new Object[this.size];
    System.arraycopy(this.elements, 1, result, 0, this.size);
    return result;
  }

  public Object[] toArray(Object a[]) {
    if (a.length < this.size) {
      a = (Object[]) java.lang.reflect.Array
          .newInstance(a.getClass().getComponentType(), this.size);
    }

    System.arraycopy(this.elements, 1, a, 0, this.size);

    if (a.length > this.size) {
      a[this.size] = null;
    }

    return a;
  }
}

/*
 * Copyright 2005 JBoss Inc
 *
```

```
 * Licensed under the Apache License, Version 2.0 (the "License"); you may not
 * use this file except in compliance with the License. You may obtain a copy of
 * the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 * License for the specific language governing permissions and limitations under
 * the License.
 */
interface Queue {
  public void enqueue(Queueable queueable);

  public Queueable dequeue();

  public Queueable dequeue(int index);

  public boolean isEmpty();
}

/*
 * Copyright 2005 JBoss Inc
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you may not
 * use this file except in compliance with the License. You may obtain a copy of
 * the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 * License for the specific language governing permissions and limitations under
 * the License.
 */

interface Queueable {
  public void enqueued(Queue queue, int index);

  public void dequeue();
}
```

Related examples in the same category

1. Demonstrates heaps (/Code/Java/Collections-Data-Structure/Demonstratesheaps.htm)

2. Fibonacci heap data structure (/Code/Java/Collections-Data-Structure/Fibonacciheapdatastructure.htm)
3. Tree Heap (/Code/Java/Collections-Data-Structure/TreeHeap.htm)
4. This class implements the heap interface using a java.util.List as the underlying data structure. (/Code/Java/Collections-Data-Structure/ThisclassimplementstheheapinterfaceusingajavautilListastheunderlyingdatastructure.htm)
5. A heap-based priority queue, without any concurrency control (/Code/Java/Collections-Data-Structure/Aheapbasedpriorityqueuewithoutanyconcurrencycontrol.htm)

Structure/Aheapbased

6. Minimum heap implementation. (/Code/Java/Collections-Data-Structure/Minimumheapimplementation.htm)
7. A MinMaxHeap provides a heap-like data structure that provides fast access to both the minimum and maximum elements of the heap. (/Code/Java/Collections-Data-Structure/AMinMaxHeapprovidesaheaplikedatastructurethatprovidesfastaccesstoboththeminimumandmaximumelementsoftheheap.htm)