AbstractPriorityQueue:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package lab09\_jacob\_huesman;

import java.util.Comparator;

/\*\*

\* Abstract base class for a PriorityQueue that is based off of the class of the same name presented in Data Structures & Algorithms by Goodrich, Tamassia, & Goldwasser

\* @author Jacob Huesman

\* @param <K>

\* @param <V>

\*/

public abstract class AbstractPriorityQueue<K,V> implements PriorityQueue<K,V> {

/\* Priority Queue Entry Class (PQEntry) \*/

protected static class PQEntry<K,V> implements Entry<K,V> {

private K k;

private V v;

/\*\*

\* Constructor

\* @param key desired key

\* @param value desired value

\*/

public PQEntry(K key, V value){

k = key;

v = value;

}

/\*\*

\* Returns the key stored in this entry object.

\* @return the entry's key

\*/

@Override

public K getKey() {

return k;

}

/\*\*

\* Returns the value stored in this entry object.

\* @return the entry's value

\*/

@Override

public V getValue() {

return v;

}

/\*\*

\* Set the key.

\* @param key desired key

\*/

protected void setKey(K key){

k = key;

}

/\*\*

\* Set the value.

\* @param value desired value

\*/

protected void setValue(V value){

v = value;

}

}

/\* End of Priority Queue Entry class \*/

/\*\*

\* Defines the ordering of keys

\*/

private Comparator<K> comp;

/\*\*

\* Creates an empty priority queue using the given comparator to order keys.

\* @param c comparator defining the order of keys in the priority queue

\*/

protected AbstractPriorityQueue(Comparator<K> c){

comp = c;

}

/\*\*

\* Creates an empty priority queue based on the natural ordering of its keys.

\*/

protected AbstractPriorityQueue(){

this(new DefaultComparator<>());

}

/\*\*

\* Method that compares two entries according to their key values

\* @param a Entry 1

\* @param b Entry 2

\* @return the result of the comparison

\*/

protected int compare(Entry<K,V> a, Entry<K,V> b){

return comp.compare(a.getKey(), b.getKey());

}

/\*\*

\* Determines if this given key is valid

\* @param key key to be checked

\* @return true - if valid; false - otherwise

\* @throws IllegalArgumentException -

\*/

protected boolean checkKey(K key) throws IllegalArgumentException{

try {

return (comp.compare(key, key) == 0);

} catch (ClassCastException e) {

throw new IllegalArgumentException("Incompatable key");

}

}

/\*\*

\* Tests whether the priority queue is empty.

\* @return true - if the priority queue is empty, false - otherwise

\*/

@Override

public boolean isEmpty(){

return size() == 0;

}

}

Client:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package lab09\_jacob\_huesman;

/\*\*

\* Client to test code functionality.

\* @author Jacob Huesman

\*/

public class Client {

public static void main(String[] args){

Key[] keys = new Key[25];

Value[] values = new Value[25];

for(int i=0; i<25; i++){

values[i] = new Value(i+1);

keys[i] = new Key(values[i].getPriority());

}

HeapPriorityQueue queue = new HeapPriorityQueue(keys, values);

System.out.println("Insert 25 Entries into the priority queue.");

queue.printBreadthFirst();

queue.printPrioritizedList();

System.out.println("");

for(int i=0; i<10; i++){

Value value = new Value(i+26);

queue.insert(new Key(value.getPriority()), value);

}

System.out.println("Insert 10 additional Entries into the priority queue.");

queue.printBreadthFirst();

queue.printPrioritizedList();

}

}

DefaultComparator:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package lab09\_jacob\_huesman;

import java.util.Comparator;

/\*\*

\* Comparator that is based off of the class of the same name presented in Data Structures & Algorithms by Goodrich, Tamassia, & Goldwasser

\* @author Jacob Huesman

\* @param <E>

\*/

public class DefaultComparator<E> implements Comparator<E> {

/\*\*

\* Compares two elements

\* @param a first element

\* @param b second element

\* @return returns a negative integer if a is less than b, 0 if a is equal to be, or a positive integer if a is greater than b

\* @throws ClassCastException

\*/

@Override

public int compare(E a, E b) throws ClassCastException {

return ((Comparable<E>) a).compareTo(b);

}

}

Entry:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package lab09\_jacob\_huesman;

/\*\*

\* Entry Interface based off of the Entry Interface presented in Data Structures & Algorithms by Goodrich, Tamassia, & Goldwasser

\* @author Jacob Huesman

\* @param <K>

\* @param <V>

\*/

public interface Entry<K,V> {

/\*\*

\* Returns the key stored in this entry object.

\* @return the entry's key

\*/

K getKey();

/\*\*

\* Returns the value stored in this entry object.

\* @return the entry's value

\*/

V getValue();

}

HeapPriorityQueue:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package lab09\_jacob\_huesman;

import java.util.ArrayList;

import java.util.Comparator;

/\*\*

\* Array-based heap implementation of a priority queue based on the class of the same name presented in Data Structures & Algorithms by Goodrich, Tamassia, & Goldwasser

\* @author Jacob Huesman

\* @param <K>

\* @param <V>

\*/

public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {

protected ArrayList<Entry<K,V>> heap = new ArrayList<>(1);

/\*\*

\* Default Constructor

\*/

public HeapPriorityQueue(){

super();

}

/\*\*

\* Constructor that creates a priority queue based on the comparator being passed

\* @param comp comparator to base the queue on

\*/

public HeapPriorityQueue(Comparator<K> comp){

super(comp);

}

/\*\*

\* Constructor that creates a priority queue based on the keys and values passed as parameters

\* @param keys array of keys

\* @param values array of values

\*/

public HeapPriorityQueue(K[] keys, V[] values){

super();

for(int i=0; i < Math.min(keys.length, values.length); i++){

heap.add(new PQEntry<>(keys[i], values[i]));

}

priorityList();

heap.trimToSize();

}

/\* Utility methods \*/

/\*\*

\* Returns the position of the parent of the node at position i

\* @param i position i

\* @return position i

\*/

protected int parent(int i){

return (i-1) / 2;

}

/\*\*

\* Returns the position of the left child of the node at position i

\* @param i position i

\* @return position of left child

\*/

protected int left(int i){

return (2\*i) + 1;

}

/\*\*

\* Returns the position of the right child of the node at position i

\* @param i position i

\* @return position of right child

\*/

protected int right(int i){

return (2\*i) + 2;

}

/\*\*

\* Returns if the node at position i has a left child

\* @param i position i

\* @return true if their is a left child

\*/

protected boolean hasLeft(int i){

return left(i) < heap.size();

}

/\*\*

\* Returns if the node at position i has a right child

\* @param i position i

\* @return true if their is a right child

\*/

protected boolean hasRight(int i){

return right(i) < heap.size();

}

/\*\*

\* Exchanges the entries at index positions i and j of the array.

\* @param i first index position

\* @param j second index position

\*/

protected void swap(int i, int j){

Entry<K,V> temp = heap.get(i);

heap.set(i, heap.get(j));

heap.set(j, temp);

}

/\*\*

\* Moves the entry at index i higher, if necessary, to maintain the properties of a heap.

\* @param i entry to perform the operation on

\*/

protected void upheap(int i){

while (i > 0){

int p = parent(i);

if(compare(heap.get(i), heap.get(p)) >= 0){

break;

}

swap(i, p);

i = p;

}

}

/\*\*

\* Moves the entry at index i lower, if necessary, to maintain the properties of a heap.

\* @param i entry to perform the operation on

\*/

protected void downheap(int i){

while(hasLeft(i)){

int leftIndex = left(i);

int smallChildIndex = leftIndex;

if(hasRight(i)){

int rightIndex = right(i);

if(compare(heap.get(leftIndex), heap.get(rightIndex)) > 0){

smallChildIndex = rightIndex;

}

}

if(compare(heap.get(smallChildIndex), heap.get(i)) >= 0){

break;

}

swap(i, smallChildIndex);

i = smallChildIndex;

}

}

/\*\*

\* Performs a bottom-up construction of the heap in linear time.

\*/

protected void priorityList(){

int startIndex = parent(size()-1);

for(int i=startIndex; i >= 0; i--){

downheap(i);

}

}

/\* Public Methods \*/

/\*\*

\* Returns the size of the priority queue.

\* @return size of queue

\*/

@Override

public int size() {

return heap.size();

}

/\*\*

\* Inserts a key-value pair and returns the entry created.

\* @param key the key of the new entry

\* @param value the associated value of the new entry

\* @return the entry storing the new key-value pair

\* @throws IllegalArgumentException if the key is unacceptable for this queue

\*/

@Override

public Entry<K, V> insert(K key, V value) throws IllegalArgumentException {

checkKey(key);

Entry<K,V> newest = new PQEntry<>(key, value);

heap.add(newest);

upheap(heap.size() - 1);

heap.trimToSize();

return newest;

}

/\*\*

\* Returns without removing an entry with the minimal key.

\* @return entry having the minimal key (or null if empty)

\*/

@Override

public Entry<K, V> min() {

if(heap.isEmpty()){

return null;

}

return heap.get(0);

}

/\*\*

\* Removes and returns an entry with the minimal key.

\* @return the removed entry (or null if empty)

\*/

@Override

public Entry<K, V> removeMin() {

if(heap.isEmpty()){

return null;

}

Entry<K,V> answer = heap.get(0);

swap(0, heap.size() -1);

heap.remove(heap.size() - 1);

downheap(0);

return answer;

}

public void printBreadthFirst(){

heap.trimToSize();

System.out.println("Printing BreadthFirst (Format (key, value)): ");

printBreadthRecursively(0, 1);

System.out.println("");

}

private void printBreadthRecursively(int i, int d){

if(i>=heap.size()){

return;

}

for(int j=0; ((j<d) && ((j+d-1)<heap.size())); j++){

System.out.print("(" + heap.get(j+d-1).getKey().toString() + ", " + heap.get(j+d-1).getValue().toString() + ") ");

}

System.out.println("");

printBreadthRecursively(i+d, d\*2);

}

private ArrayList<Entry<K,V>> getPriorityList(){

heap.trimToSize();

int n = heap.size();

ArrayList<Entry<K,V>> list = new ArrayList<Entry<K,V>>(n);

HeapPriorityQueue clone = new HeapPriorityQueue();

clone.heap = (ArrayList<Entry<K,V>>) heap.clone();

for(int i=0; i<n; i++){

list.add(clone.removeMin());

}

return list;

}

public void printPrioritizedList(){

System.out.println("Printing PrioritizedList (Format (key, value)): ");

ArrayList<Entry<K,V>> list = getPriorityList();

for(Entry<K,V> entry : list){

System.out.print("(" + entry.getKey().toString() + ", " + entry.getValue().toString() + ") ");

}

System.out.println("");

}

}

Key:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package lab09\_jacob\_huesman;

/\*\*

\* Key class designed to satisfy the requirements of the lab09 assignment.

\* @author Jacob Huesman

\*/

public class Key implements Comparable {

private final int key;

/\*\*

\* Constructor for a key object

\* @param key desired key value

\*/

public Key(int key){

this.key = key;

}

/\*\*

\* Accessor for the key object value

\* @return integer key

\*/

public int getKey(){

return key;

}

public String toString(){

return String.valueOf(key);

}

@Override

public int compareTo(Object o) {

return key - ((Key) o).getKey();

}

}

PriorityQueue:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package lab09\_jacob\_huesman;

/\*\*

\* PriorityQueue Interface ADT based on the the one presented in Data Structures and Algorithms by Goodrich, Tamassia, & Goldwasser.

\* @author Jacob Huesman

\* @param <K>

\* @param <V>

\*/

public interface PriorityQueue<K,V> {

/\*\*

\* Returns the size of the priority queue.

\* @return size of queue

\*/

int size();

/\*\*

\* Returns if the priority queue is empty.

\* @return true - if empty; false - otherwise

\*/

boolean isEmpty();

/\*\*

\* Inserts a key-value pair and returns the entry created.

\* @param key the key of the new entry

\* @param value the associated value of the new entry

\* @return the entry storing the new key-value pair

\* @throws IllegalArgumentException if the key is unacceptable for this queue

\*/

Entry<K,V> insert(K key, V value) throws IllegalArgumentException;

/\*\*

\* Returns without removing an entry with the minimal key.

\* @return entry having the minimal key (or null if empty)

\*/

Entry<K,V> min();

/\*\*

\* Removes and returns an entry with the minimal key.

\* @return the removed entry (or null if empty)

\*/

Entry<K,V> removeMin();

}

Value:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package lab09\_jacob\_huesman;

import java.util.Random;

/\*\*

\* Value object designed to satisfy the requirements of lab09.

\* @author Jacob Huesman

\*/

public class Value {

private final int order, priority;

private final Random random = new Random();

/\*\*

\* Constructor that allows for the specification of the order of the object.

\* @param order order of the object

\*/

public Value(int order){

this.order = order;

priority = random.nextInt(11);

}

/\*\*

\* Returns the insertion order of this object

\* @return order

\*/

public int getOrder(){

return order;

}

/\*\*

\* Returns the priority of this object

\* @return priority

\*/

public int getPriority(){

return priority;

}

public String toString(){

return String.valueOf(order);

}

}

Screenshot:

