/\*

Alexander Giang

masc2169

\*/

package data\_structures;

import java.util.Iterator;

import java.util.NoSuchElementException;

import java.util.ConcurrentModificationException;

public class OrderedListPriorityQueue<E> implements PriorityQueue<E> {

public static final int DEFAULT\_MAX\_CAPACITY = 1000;

private long changeCounter;

private int currentSize;

Node<E> head;

public OrderedListPriorityQueue() {

head = null;

currentSize = 0;

changeCounter = 0;

}

// Inserts a new object into the priority queue. Returns true if

// the insertion is successful. If the PQ is full, the insertion

// is aborted, and the method returns false.

public boolean insert(E object) {

Node<E> newNode = new Node<E>(object);

Node<E> current = head, previous = null;

if (isFull())

return false;

while (current != null &&

((Comparable<E>)object).compareTo(current.data) >= 0) {

previous = current;

current = current.next;

}

if (previous == null) {

newNode.next = head;

head = newNode;

currentSize++;

changeCounter++;

return true;

}

else {

previous.next = newNode;

newNode.next = current;

changeCounter++;

currentSize++;

return true;

}

}

// Removes the object of highest priority that has been in the

// PQ the longest, and returns it. Returns null if the PQ is empty.

public E remove() {

if (isEmpty())

return null;

Node<E> current = head;

head = head.next;

currentSize--;

changeCounter++;

return current.data;

}

// Returns the object of highest priority that has been in the

// PQ the longest, but does NOT remove it.

// Returns null if the PQ is empty.

public E peek() {

if (isEmpty())

return null;

return head.data;

}

// Returns true if the priority queue contains the specified element

public boolean contains(E obj) {

Node<E> current = head;

while(current != null) {

if (((Comparable<E>)obj).compareTo(current.data) == 0)

return true;

current = current.next;

}

return false;

}

// Returns the number of objects currently in the PQ.

public int size() {

return currentSize;

}

// Returns the PQ to an empty state.

public void clear() {

currentSize = 0;

}

// Returns true if the PQ is empty, otherwise false

public boolean isEmpty() {

return currentSize == 0;

}

// Returns true if the PQ is full, otherwise false. List based

// implementations should always return false.

public boolean isFull() {

return currentSize == DEFAULT\_MAX\_CAPACITY;

}

// Returns an iterator of the objects in the PQ, in no particular

// order.

public Iterator<E> iterator() {

return new IteratorHelper();

}

class IteratorHelper implements Iterator<E> {

private Node<E> iterPointer;

private long modCounter;

//constructor

public IteratorHelper() {

iterPointer = head;

changeCounter = modCounter;

}

public boolean hasNext() {

if(changeCounter != modCounter)

throw new ConcurrentModificationException();

return iterPointer != null;

}

public E next() {

if(!hasNext())

throw new NoSuchElementException();

E temp = iterPointer.data;

iterPointer = iterPointer.next;

return temp;

}

public void remove() {

throw new UnsupportedOperationException();

}

}

/////////////////////////

private class Node<E> {

E data;

Node<E> next;

public Node(E d) {

data = d;

}

}

////////////////////////

}

/\*

Alexander Giang

masc2169

\*/

package data\_structures;

import java.util.Iterator;

import java.util.NoSuchElementException;

public class OrderedArrayPriorityQueue<E> implements PriorityQueue<E> {

public static final int DEFAULT\_MAX\_CAPACITY = 1000;

private OrderedArrayList<E> list;

// default constructor w/ no arguments that uses the DEFAULT\_MAX\_CAPACITY

// constant from the PriorityQueue.java interface

public OrderedArrayPriorityQueue() {

list = new OrderedArrayList<E>(DEFAULT\_MAX\_CAPACITY);

}

// constructor that takes a single integer parameter that

// represents the maximum capacity of the PQ

public OrderedArrayPriorityQueue(int listSize) {

list = new OrderedArrayList<E>(listSize);

}

// Inserts a new object into the priority queue. Returns true if

// the insertion is successful. If the PQ is full, the insertion

// is aborted, and the method returns false.

public boolean insert(E object) {

if(isFull())

return false;

list.insert(object);

return true;

}

// Removes the object of highest priority that has been in the

// PQ the longest, and returns it. Returns null if the PQ is empty.

public E remove() {

if(list.isEmpty())

return null;

return list.removeMin();

}

// Returns the object of highest priority that has been in the

// PQ the longest, but does NOT remove it.

// Returns null if the PQ is empty.

public E peek() {

return list.get(0);

}

// Returns true if the priority queue contains the specified element

public boolean contains(E obj) {

return list.contains(obj);

}

// Returns the number of objects currently in the PQ.

public int size() {

return list.size();

}

// Returns the PQ to an empty state.

public void clear() {

list.clear();

}

// Returns true if the PQ is empty, otherwise false

public boolean isEmpty() {

return list.isEmpty();

}

// Returns true if the PQ is full, otherwise false. List based

// implementations should always return false.

public boolean isFull() {

return list.isFull();

}

// Returns an iterator of the objects in the PQ, in no particular

// order.

public Iterator<E> iterator() {

return list.iterator();

}

}

/\* Alexander Giang

masc2169

\*/

package data\_structures;

import java.util.Iterator;

import java.util.NoSuchElementException;

public class OrderedArrayList<E> implements OrderedListADT<E> {

private int maxSize, currentSize;

private E[] storage;

//constructor

public OrderedArrayList(int size) {

maxSize = size;

currentSize = 0;

storage = (E[]) new Object[maxSize];

}

//binary search for insert()

public int findInsertionPoint(E key, int lo, int hi) {

if( hi < lo )

return lo;

int mid = (lo+hi)/2;

if(((Comparable<E>)key).compareTo(storage[mid]) < 0)

return findInsertionPoint(key,lo,mid-1);

return findInsertionPoint(key,mid+1,hi);

}

// binary search for find()

public int findBinSearch(E key, int lo, int hi) {

if(hi == lo) {

if(((Comparable<E>)key).compareTo(storage[lo]) == 0)

return lo;

return -1;

}

int mid = (lo+hi)/2;

if(((Comparable<E>)key).compareTo(storage[mid]) <= 0)

return findBinSearch(key,lo,mid);

return findBinSearch(key,mid+1,hi);

}

// Adds the Object obj to the list in the correct position as determined by the Comparable interface.

public void insert(E obj) {

if(isFull())

throw new RuntimeException();

int where = findInsertionPoint(obj,0,currentSize-1);

for(int i = currentSize ; i > where ; i--) //shift

storage[i] = storage[i-1];

storage[where] = obj;

currentSize++;

}

// Removes and returns the object located at the parameter index position (zero based).

// Throws IndexOutOfBoundsException if the index does not map to a valid position within the list.

public E remove(int index) {

E temp = (E) new Object();

temp = storage[index];

for(int i = index ; i < currentSize-1 ; i++) //shift

storage[i] = storage[i+1];

if(index < 0 || index > currentSize)

throw new IndexOutOfBoundsException();

currentSize--;

return temp;

}

// Removes and returns the parameter object obj from the list if the list contains it, null otherwise.

public E remove(E obj) {

int where = find(obj);

if(!contains(obj))

return null;

return remove(where);

}

// Removes and returns the smallest element in the list and null if the it is empty.

public E removeMin() {

if(isEmpty())

return null;

return remove(0);

}

// Removes and returns the largest element in the list and null if the it is empty.

public E removeMax() {

if(isEmpty())

return null;

return remove(currentSize-1);

}

// Returns the parameter object located at the parameter index position (zero based).

// Throws IndexOutOfBoundsException if the index does not map to a valid position within the underlying array

public E get(int index) {

if(index < 0 || index > currentSize)

throw new IndexOutOfBoundsException();

return storage[index];

}

// Returns the list object that matches the parameter, and null if the list is empty.

// Also returns null if the obj is NOT in the list.

// This method is stable, if obj matches more than one element, the element that

// has been in the list longest is returned.

public E get(E obj) {

if(isEmpty() )

return null;

if(!contains(obj))

return null;

int where = find(obj);

return storage[where];

}

// Returns the index of the first element that matches the parameter obj

// and -1 if the item is not in the list.

public int find(E obj) {

return findBinSearch(obj,0,currentSize-1);

}

// Returns true if the parameter object obj is in the list, false otherwise.

public boolean contains(E obj) {

if(find(obj) == -1)

return false;

return true;

}

// The list is returned to an empty state.

public void clear() {

currentSize = 0;

}

// Returns true if the list is empty, otherwise false

public boolean isEmpty() {

return currentSize == 0;

}

// Returns true if the list is full, otherwise false

public boolean isFull() {

return currentSize == maxSize;

}

// Returns the number of Objects currently in the list.

public int size() {

return currentSize;

}

// Returns an Iterator of the values in the list, presented in

// the same order as the list.

public Iterator<E> iterator() {

return new IteratorHelper();

}

class IteratorHelper implements Iterator<E> {

private int iterIndex;

//constructor

public IteratorHelper() {

iterIndex = 0;

}

public boolean hasNext() {

return iterIndex < currentSize;

}

public E next() {

if(!hasNext())

throw new NoSuchElementException();

return storage[iterIndex++];

}

public void remove() {

throw new UnsupportedOperationException();

}

}

}

/\*

Alexander Giang

masc2169

\*/

package data\_structures;

import java.util.Iterator;

import java.util.NoSuchElementException;

public class OrderedArrayList<E> implements OrderedListADT<E> {

private int maxSize, currentSize;

private E[] storage;

//constructor

public OrderedArrayList(int size) {

maxSize = size;

currentSize = 0;

storage = (E[]) new Object[maxSize];

}

//binary search for insert()

public int findInsertionPoint(E key, int lo, int hi) {

if( hi < lo )

return lo;

int mid = (lo+hi)/2;

if(((Comparable<E>)key).compareTo(storage[mid]) < 0)

return findInsertionPoint(key,lo,mid-1);

return findInsertionPoint(key,mid+1,hi);

}

// binary search for find()

public int findBinSearch(E key, int lo, int hi) {

if(hi == lo) {

if(((Comparable<E>)key).compareTo(storage[lo]) == 0)

return lo;

return -1;

}

int mid = (lo+hi)/2;

if(((Comparable<E>)key).compareTo(storage[mid]) <= 0)

return findBinSearch(key,lo,mid);

return findBinSearch(key,mid+1,hi);

}

// Adds the Object obj to the list in the correct position as determined by the Comparable interface.

public void insert(E obj) {

if(isFull())

throw new RuntimeException();

int where = findInsertionPoint(obj,0,currentSize-1);

for(int i = currentSize ; i > where ; i--) //shift

storage[i] = storage[i-1];

storage[where] = obj;

currentSize++;

}

// Removes and returns the object located at the parameter index position (zero based).

// Throws IndexOutOfBoundsException if the index does not map to a valid position within the list.

public E remove(int index) {

E temp = (E) new Object();

temp = storage[index];

for(int i = index ; i < currentSize-1 ; i++) //shift

storage[i] = storage[i+1];

if(index < 0 || index > currentSize)

throw new IndexOutOfBoundsException();

currentSize--;

return temp;

}

// Removes and returns the parameter object obj from the list if the list contains it, null otherwise.

public E remove(E obj) {

int where = find(obj);

if(!contains(obj))

return null;

return remove(where);

}

// Removes and returns the smallest element in the list and null if the it is empty.

public E removeMin() {

if(isEmpty())

return null;

return remove(0);

}

// Removes and returns the largest element in the list and null if the it is empty.

public E removeMax() {

if(isEmpty())

return null;

return remove(currentSize-1);

}

// Returns the parameter object located at the parameter index position (zero based).

// Throws IndexOutOfBoundsException if the index does not map to a valid position within the underlying array

public E get(int index) {

if(index < 0 || index > currentSize)

throw new IndexOutOfBoundsException();

return storage[index];

}

// Returns the list object that matches the parameter, and null if the list is empty.

// Also returns null if the obj is NOT in the list.

// This method is stable, if obj matches more than one element, the element that

// has been in the list longest is returned.

public E get(E obj) {

if(isEmpty() )

return null;

if(!contains(obj))

return null;

int where = find(obj);

return storage[where];

}

// Returns the index of the first element that matches the parameter obj

// and -1 if the item is not in the list.

public int find(E obj) {

return findBinSearch(obj,0,currentSize-1);

}

// Returns true if the parameter object obj is in the list, false otherwise.

public boolean contains(E obj) {

if(find(obj) == -1)

return false;

return true;

}

// The list is returned to an empty state.

public void clear() {

currentSize = 0;

}

// Returns true if the list is empty, otherwise false

public boolean isEmpty() {

return currentSize == 0;

}

// Returns true if the list is full, otherwise false

public boolean isFull() {

return currentSize == maxSize;

}

// Returns the number of Objects currently in the list.

public int size() {

return currentSize;

}

// Returns an Iterator of the values in the list, presented in

// the same order as the list.

public Iterator<E> iterator() {

return new IteratorHelper();

}

class IteratorHelper implements Iterator<E> {

private int iterIndex;

//constructor

public IteratorHelper() {

iterIndex = 0;

}

public boolean hasNext() {

return iterIndex < currentSize;

}

public E next() {

if(!hasNext())

throw new NoSuchElementException();

return storage[iterIndex++];

}

public void remove() {

throw new UnsupportedOperationException();

}

}

}

**Priority Queue Report**

|  |  |
| --- | --- |
| **Size** | **Time(ms)** |
| 100 | 87 |
| 2100 | 38 |
| 4100 | 66 |
| 6100 | 110 |
| 8100 | 207 |
| 10100 | 229 |
| 12100 | 462 |
| 14100 | 549 |
| 16100 | 580 |
| 18100 | 721 |
| 20100 | 782 |
| 22100 | 1073 |
| 24100 | 1324 |
| 26100 | 1518 |
| 28100 | 1454 |

n + log n + 1 = O(n)

**Analysis of the insert()**

In the OrderedArrayList implemented by the OrderedArrayPriorityQueue, the insert() method requires a binary search to find insertion point and a shift for an average of *n* elements. Even though the insert itself is of O(1) complexity, and the binary search is O(log n), the shift is O(n) complexity with an average case of n/2, which makes the overall complexity O(n). We know it is O(n) because when the size doubles, the time roughly doubles. For example, from 8100 to 16100 elements, it raised from 207ms to 580ms. With equal intervals between the doubling sizes, we can see the graph is roughly linear which also proves it is O(n) complexity.

|  |  |
| --- | --- |
| **Size** | **Time(ms)** |
| 100 | 87 |
| 2100 | 38 |
| 4100 | 66 |
| 6100 | 110 |
| 8100 | 207 |
| 10100 | 229 |
| 12100 | 462 |
| 14100 | 549 |
| 16100 | 580 |
| 18100 | 721 |
| 20100 | 782 |
| 22100 | 1073 |
| 24100 | 1324 |
| 26100 | 1518 |
| 28100 | 1454 |

n = O(n)

**Analysis of the remove()**

The remove() method uses the removeMin() from the OrderedArrayList, which is O(n) complexity because in a priority queue, you should remove from the head since it is a FIFO method. This can be done by shifting the elements over the to-be-removed-element, therefore it is O(n) complexity with an average case of n/2 (looking through half of the array). The data seems to look like a complexity slower than O(n) since it appears to curve, yet some of the times roughly double as expected, for example 8100 to 16100 is 207ms to 580ms.

|  |  |
| --- | --- |
| **Size** | **Time(ms)** |
| 200000 | 528 |
| 400000 | 456 |
| 600000 | 444 |
| 800000 | 445 |
| 1000000 | 423 |
| 1200000 | 545 |
| 1400000 | 444 |
| 1600000 | 617 |
| 1800000 | 678 |
| 2000000 | 599 |
| 2200000 | 575 |
| 2400000 | 649 |
| 2600000 | 693 |
| 2800000 | 711 |
| 3000000 | 662 |

n + 1 = O(n)

**Analysis of the insert()**

In the OrderedListPriorityQueue, the insert() is O(n) because we need to traverse through the list one element at a time to find the insertion point with an average case of n/2, and the insert itself is O(1). Since the insert itself is practically insignificant, the overall complexity is O(n). The data seems to have a complexity of O(n) as well, since the graph appears to have a slight linear increase.

|  |  |
| --- | --- |
| **Size** | **Time(ms)** |
| 2000000 | 138 |
| 4000000 | 37 |
| 6000000 | 56 |
| 8000000 | 36 |
| 10000000 | 56 |
| 12000000 | 26 |
| 14000000 | 38 |
| 16000000 | 44 |
| 18000000 | 57 |
| 20000000 | 34 |
| 22000000 | 36 |
| 24000000 | 165 |
| 26000000 | 36 |
| 28000000 | 39 |
| 30000000 | 33 |

1 = O(1)

**Analysis of the remove()**

The remove() is O(1) because we only pop off the head of the priority queue. We don’t need to shift like in an array, since we can overwrite the head node in a linked list to be the element after the head, which makes the previous head node garbage. The data seems to also be O(1) complexity with data spikes (probably from other computer applications that affected the runtime data). The range stays between a constant range of 30ms and 60ms, aside from the jumps.