/\*

\* Program #3

\* Quang Trinh

\* cssc0759

\*/

**package** data\_structures;

**import** java.util.ConcurrentModificationException;

**import** java.util.Iterator;

**import** java.util.NoSuchElementException;

**public** **class** BinaryHeapPriorityQueue<E **extends** Comparable<E>> **implements** PriorityQueue<E> {

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

**private** **class** Wrapper<T> **implements** Comparable<Wrapper<T>> {

**long** number;

T data;

**public** Wrapper(T d) {

number = entryNumber++; // from parent class

data = d;

}

**public** **int** compareTo(Wrapper<T> o) {

**if**(((Comparable<T>)data).compareTo(o.data) == 0)

**return** (**int**)(number - o.number);

**return** ((Comparable<T>)data).compareTo(o.data);

}

}

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

**private** **long** entryNumber;

**private** **int** currentSize;

**private** **int** maxSize;

**private** Wrapper<E> []array;

**private** **long** modCounter;

// Default constructor

**public** BinaryHeapPriorityQueue() {

**this**(***DEFAULT\_MAX\_CAPACITY***);

}

// Customized constructor

**public** BinaryHeapPriorityQueue(**int** size) {

entryNumber = 0;

maxSize = size;

array = **new** Wrapper[maxSize];

modCounter = 0;

}

**public** **boolean** insert(E object) {

**if**(isFull())

**return** **false**;

Wrapper<E> newNode = **new** Wrapper<E>(object);

// Place item into heap at bottom

array[currentSize++] = newNode;

modCounter++;

// Correcting heap ordering

trickleUp();

**return** **true**;

}

**public** E remove() {

**if**(isEmpty())

**return** **null**;

E result = peek();

// Place last item at root

array[0] = array[currentSize-1];

currentSize--;

// Correcting heap ordering

trickleDown(0); // trickleDown from the root

modCounter++;

**return** result;

}

/\*

\* This method will loop until there is no instance of the object left.

\* By implementing trickleDown(int), the array will be able to delete item

\* and correct heap ordering at the same time before going to the next loop.

\*/

**public** **boolean** delete(E obj) {

**if**(!contains(obj))

**return** **false**;

// Delete until array has no instance of the object

**while**(contains(obj)) {

**for**(**int** i = 0; i < currentSize; i++) {

**if**(array[i].data.compareTo(obj) == 0) { // Index of item

array[i] = array[currentSize-1];

currentSize--;

trickleDown(i); // trickleDown from i

}

}

}

modCounter++;

**return** **true**;

}

**public** E peek() {

**if**(isEmpty())

**return** **null**;

**return** array[0].data;

}

**public** **boolean** contains(E obj) {

**if**(isEmpty())

**return** **false**;

**for**(**int** i = 0; i < currentSize; ++i) {

**if**(obj.compareTo(array[i].data) == 0)

**return** **true**;

}

**return** **false**;

}

**public** **int** size() {

**return** currentSize;

}

**public** **void** clear() {

currentSize = 0;

modCounter = 0;

entryNumber = 0;

}

**public** **boolean** isEmpty() {

**return** currentSize == 0;

}

**public** **boolean** isFull() {

**return** currentSize == maxSize;

}

**public** Iterator<E> iterator() {

**return** **new** IteratorHelper();

}

/\*

\* TrickleUp method for insertion.

\*/

**private** **void** trickleUp() {

**int** newIndex = currentSize - 1;

**int** parentIndex = (newIndex-1) >> 1;

Wrapper<E> newNode = array[newIndex];

**while**(parentIndex >= 0 && newNode.compareTo(array[parentIndex]) < 0) {

array[newIndex] = array[parentIndex];

newIndex = parentIndex;

parentIndex = (parentIndex-1) >> 1;

}

array[newIndex] = newNode;

}

/\*

\* This method trickleDown the heap from index position that passed in.

\*/

**private** **void** trickleDown(**int** index) {

**int** current = index;

**int** child = getNextChild(current);

Wrapper<E> removedNode = array[current];

**while**(child != -1 && array[child].compareTo(array[current]) < 0) {

swap(child, current);

current = child;

child = getNextChild(current);

}

array[current] = removedNode;

}

/\*

\* Private helper for trickleDown(int) method.

\* Return of appropriate child index when called.

\* If a node has 2 children, choose the smaller.

\* If a node has 1 child, choose that child.

\*/

**private** **int** getNextChild(**int** current) {

**int** left = (current << 1) + 1;

**int** right = left+1;

**if**(right < currentSize) { // there are 2 children

**if**(array[left].compareTo(array[right]) < 0)

**return** left; // the left child is smaller

**return** right; // the right child is smaller

}

**if**(left < currentSize) // there is only one child

**return** left;

**return** -1; // no children

}

/\*

\* Private helper for trickleDown(int) method.

\* Swap 2 elements of an array.

\*/

**private** **void** swap(**int** index1, **int** index2) {

Wrapper<E> tmp = array[index1];

array[index1] = array[index2];

array[index2] = tmp;

}

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

**class** IteratorHelper **implements** Iterator<E> {

**int** iterIndex;

**long** stateCheck;

**public** IteratorHelper() {

iterIndex = 0;

stateCheck = modCounter;

}

**public** **boolean** hasNext() {

**if** (stateCheck != modCounter)

**throw** **new** ConcurrentModificationException();

**return** iterIndex < currentSize;

}

**public** E next() {

**if** (!hasNext())

**throw** **new** NoSuchElementException();

**return** array[iterIndex++].data;

}

**public** **void** remove() {

**throw** **new** UnsupportedOperationException();

}

}

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

}

**COMPLEXITY REPORT**

1. **OrderedArrayPriorityQueue.java**

Insertion:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | **1.2** |
| 200 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | **0.3** |
| 400 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | **0.6** |
| 800 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 2 | **0.8** |
| 1600 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | **1.2** |
| 3200 | 4 | 4 | 4 | 4 | 3 | 4 | 5 | 4 | 4 | 4 | **4** |
| 6400 | 12 | 12 | 14 | 14 | 15 | 13 | 14 | 13 | 13 | 16 | **13.6** |
| 12800 | 51 | 51 | 49 | 54 | 48 | 71 | 52 | 54 | 52 | 66 | **54.8** |
| 25600 | 226 | 218 | 230 | 220 | 233 | 219 | 224 | 226 | 212 | 225 | **223.3** |
| 51200 | 979 | 935 | 946 | 1008 | 978 | 946 | 972 | 951 | 948 | 956 | **961.9** |

Figure 1-1: Insertion times

Figure 1-2: Graph of average insertion time (ms)

Removal:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 400 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 800 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 1600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 3200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 6400 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | **0.1** |
| 12800 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | **0.4** |
| 25600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 51200 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **0.1** |

Figure 1-3: Removal times

Figure 1-4: Graph of average removal time (ms)

Expected insertion time complexity for Ordered Array is O(n). The graph of expected insertion time complexity is a linear line. My graph after running the program 10 times yield very similar results with the expected results. The insertion times start to grow fast after n = 1600.

Expected removal time complexity for Ordered Array is O(1). My results after running the program 10 times yield close results with the expected results. That is, no matter how big n grows, the times it takes are similar for every n size.

1. **UnorderedArrayPriorityQueue.java**

Insertion:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 2 | 0 | 1 | **0.7** |
| 200 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0.1** |
| 400 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | **0** |
| 800 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | **0.1** |
| 1600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | **0** |
| 3200 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 4 | **0.3** |
| 6400 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 16 | **0.4** |
| 12800 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 66 | **0.5** |
| 25600 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 225 | **1.2** |
| 51200 | 1 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 956 | **1.5** |

Figure 2-1: Insertion times

Figure 2-2: Graph of insertion times (ms)

Removal:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0.1** |
| 200 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **0.2** |
| 400 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | **0.3** |
| 800 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | **0.6** |
| 1600 | 3 | 4 | 3 | 3 | 4 | 3 | 3 | 3 | 3 | 3 | **3.2** |
| 3200 | 11 | 11 | 11 | 11 | 11 | 11 | 12 | 13 | 11 | 12 | **11.4** |
| 6400 | 45 | 42 | 44 | 42 | 43 | 43 | 44 | 44 | 43 | 43 | **43.3** |
| 12800 | 170 | 177 | 178 | 173 | 173 | 172 | 174 | 182 | 171 | 173 | **174.3** |
| 25600 | 761 | 758 | 826 | 817 | 744 | 826 | 699 | 757 | 750 | 764 | **770.2** |
| 51200 | 2798 | 2623 | 3212 | 3207 | 2898 | 2878 | 3068 | 2630 | 3037 | 3162 | **2951.3** |

Figure 2-3: Removal times

Figure 2-4: Graph of removal times (ms)

Expected insertion time complexity for Unordered Array is O(1). My result after running the program 10 times yield similar results with the expected results. Although the graph does not look like a straight horizontal line (as expected), the values don’t vary a lot.

Expected removal time complexity for Unordered Array is O(n). My results after running the program 10 times yield close results with the expected results. That is, the bigger n grows, the longer the time takes.

1. **OrderedLinkedListPriorityQueue.java**

Insertion:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 2 | 1 | 0 | 1 | 0 | 1 | 2 | 0 | 0 | 1 | **0.8** |
| 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 400 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | **0.2** |
| 800 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | **0.8** |
| 1600 | 3 | 3 | 4 | 3 | 3 | 4 | 4 | 3 | 3 | 3 | **3.3** |
| 3200 | 11 | 11 | 12 | 11 | 11 | 11 | 10 | 11 | 12 | 11 | **11.1** |
| 6400 | 56 | 55 | 56 | 56 | 55 | 55 | 58 | 56 | 55 | 56 | **55.8** |
| 12800 | 313 | 310 | 311 | 316 | 313 | 317 | 319 | 310 | 309 | 324 | **314.2** |
| 25600 | 1811 | 1798 | 1814 | 1810 | 1811 | 1823 | 1804 | 1820 | 1813 | 1809 | **1811.3** |
| 51200 | 8173 | 8177 | 8175 | 8170 | 8177 | 8160 | 8176 | 8180 | 8234 | 8150 | **8177.2** |

Figure 3-1: Insertion times

Figure 3-2: Graph of insertion time (ms)

Removal:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 400 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 800 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 1600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 3200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 6400 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | **0.1** |
| 12800 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 25600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 51200 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | **0.1** |

Figure 3-3: Removal times

Figure 3-4: Graph of removal times (ms)

Expected insertion time complexity for Ordered Linked List is O(n) since the worst case is insert at the end. My result after running the program 10 times yield similar results with the expected results. My graph demonstrates that when n grows, the time grows with the same rate every time.

Expected removal time complexity for Ordered Linked List is O(1). My results after running the program 10 times yield close results with the expected results. In fact, values are 0 most of the time.

1. **UnorderedLinkedListPriorityQueue.java**

Insertion:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **0.1** |
| 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 400 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 800 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | **0.1** |
| 1600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 3200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 6400 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | **0.2** |
| 12800 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 25600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 51200 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | **0.3** |

Figure 4-1: Insertion times

Figure 4-2: Graph of insertion time (ms)

Removal:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 0 | 0 | 0 | 1 | 0 | 1 | 2 | 0 | 0 | 1 | **0.5** |
| 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 400 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | **0.2** |
| 800 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | **0.8** |
| 1600 | 9 | 9 | 10 | 9 | 9 | 7 | 8 | 10 | 13 | 7 | **9.1** |
| 3200 | 35 | 35 | 35 | 35 | 39 | 34 | 33 | 35 | 38 | 33 | **35.2** |
| 6400 | 132 | 131 | 142 | 132 | 152 | 142 | 130 | 134 | 141 | 139 | **137.5** |
| 12800 | 467 | 402 | 501 | 479 | 468 | 457 | 471 | 488 | 468 | 472 | **467.3** |
| 25600 | 1640 | 1701 | 1699 | 1650 | 1673 | 1693 | 1690 | 1664 | 1688 | 1679 | **1677.7** |
| 51200 | 6959 | 6177 | 6175 | 6170 | 7177 | 6160 | 6176 | 6180 | 6234 | 6150 | **6355.8** |

Figure 4-3: Removal times

Figure 4-4: Graph of removal times (ms)

Expected insertion time complexity for Unordered Linked List is O(1). My result after running the program 10 times yield similar results with the expected results.

Expected removal time complexity for Unordered Array is O(n). My results after running the program 10 times yield close results with the expected results. In fact, the graph looks very close to graph of O(n).

1. **BinaryHeapPriorityQueue.java**

Insertion:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **0.1** |
| 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 400 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 800 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | **0.1** |
| 1600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 3200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 6400 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | **0.2** |
| 12800 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | **0.6** |
| 25600 | 2 | 2 | 1 | 0 | 1 | 1 | 2 | 1 | 0 | 1 | **1.1** |
| 51200 | 3 | 2 | 2 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | **2.3** |

Figure 5-1: Insertion times

Figure 5-2: Graph of insertion times (ms)

Removal:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (n) | 1st time | 2nd time | 3rd time | 4th time | 5th time | 6th time | 7th time | 8th time | 9th time | 10th time | **Average** |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **0.1** |
| 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 400 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 800 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | **0.1** |
| 1600 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 3200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 6400 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 12800 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 25600 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | **0.2** |
| 51200 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | **0.2** |

Figure 5-3: Removal times

Figure 5-4: Graph of removal times (ms)

Expected insertion time complexity for Binary Heap is O(log n).My result after running the program 10 times yield similar results with the expected results. This in fact is the best implementation for array.

Expected removal time complexity for Binary Heap is O(log n). My results after running the program 10 times yield close results with the expected results. The graph shows that it grows very slow compare to n size.