# HeapPriorityQueue

Describe the characteristic behavior of a priority queue: \_\_\_Removes/adds in order of least to greatest

Look at the cheat sheet. What methods are in priority queues? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A heap, specifically a min-heap, is a good data structure with which to implement a priority queue, because a priority queue needs quickly to find the minimum value. If we are clever, we can add to and remove from a heap in O(log n). This is better performance than the other four ways of implementing priority queues. What were those ways?

\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_

**How HeapPriorityQueue works:**

First of all, recognize that it is a min-heap, meaning that the parent is smaller than each child. As in the lunchroom simulation, a 1 represents a senior, 2 a junior, 3 a sophomore, and 4 a freshman. The heap below models the order in which the students are served their lunches.

1

2

3

4

5

6

8

9

7

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| null | 1 | 1 | 1 | 2 | 4 | 4 | 3 | 2 | 1 |  |  |  |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] |

What is the algorithm to add a student? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the algorithm to remove a student? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Assignment**

Write a HeapPriorityQueue<E> class that acts like java.util.PriorityQueue<E>. Use this header: public class HeapPriorityQueue<E extends Comparable<E>>

* The private field is an ArrayList<E>.
* A default constructor is enough.
* lastIndex() is useful.
* Implement add(E), remove(), peek(), isEmpty()
* add(E) calls heapUp which uses k/2. Modify heapUp to work on a min-heap.
* remove() calls heapDown which uses 2\*k and 2\*k+1. Modify heapDown to work on a min-heap.
* A swap() method will be useful.
* A toString() method will help in debugging

Your HeapPriorityQueue is a resource class which can test with HeapPriorityQueue\_Driver. Then let's get more interesting: instantiate your HeapPriorityQueue in the Lunchroom simulation, either your own or LunchRoom. You should see no change in the results, i.e., seniors are still served first and freshmen last.

**Caution**

Double-check your add and remove methods. Are they O(n) or O(log n)? We want O(log n).

--------------------------------------------------------------------------------------------------------------

Handout from January's unit on Stacks, Queues, and PriorityQueues:

4. You are not expected to know any specific implementation of the PriorityQueue class. You should, however, be able to discuss the effects of common implementations on the Big-O efficiency of the PriorityQueue's methods. Five common implementations are:

|  |  |
| --- | --- |
| insertion |  |
| deletion |  |

i. **An array with elements in random order.** Insertion is done at the end of the list. Deletion requires a linear search. java.util.PriorityQueue actually is an array with elements in random order.

|  |  |
| --- | --- |
| insertion |  |
| deletion |  |

ii. **An array with elements sorted by priority**, smallest elements at the end. Deletion means removing the last element in the array. Insertion requires finding the insertion point and then creating an empty place by shifting array elements.

|  |  |
| --- | --- |
| add() |  |
| peek() |  |

|  |  |
| --- | --- |
| insertion |  |
| deletion |  |

iii. **A linear linked list with elements in random order.** Insertion is done at the front of the list. Deletion requires a linear search.

iv. **A linear linked list with elements sorted by priority,** smallest elements in front. Deletion means removal of the first node. Insertion requires a traverse.

|  |  |
| --- | --- |
| add() |  |
| remove() |  |

v. **A minimum heap**. A heap is a binary tree structure with the property that the value in every node is less than or equal to the value in each of its children. We’ll study min heaps after we study trees. After either insertion or deletion, the binary tree is restored to order (reheaped) in O(log n) time.

|  |  |  |
| --- | --- | --- |
| insertion | ***O(log n)*** |  |
| deletion | ***O(log n)*** |  |