DATA STRUCTURES

Priority Queue

In normal queue data structure, insertion is performed at the end of the queue and deletion is performed based on the FIFO principle. This queue implementation may not be suitable for all situations.

Consider a networking application where server has to respond for requests from multiple clients using queue data structure. Assume four requests arrived to the queue in the order of R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. Queue is as follows...

Now, check waiting time for each request to be complete.

1. **R1 : 20 units of time**
2. **R2 : 22 units of time (R2 must wait till R1 complete - 20 units and R2 itself requeres 2 units. Total 22 units)**
3. **R3 : 32 units of time (R3 must wait till R2 complete - 22 units and R3 itself requeres 10 units. Total 32 units)**

4. **R4 : 37 units of time (R4 must wait till R3 complete - 35 units and R4 itself requeres 5 units. Total 37 units)**

Here, average waiting time for all requests (R1, R2, R3 and R4) is (20+22+32+37)/4 ≈ 27 units of time.

That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is 27 units of time.

Now, consider another way of serving these requests. If we serve according to their required amount of time. That means, first we serve R2 which has minimum time required (2) then serve R4 which has second minimum time required (5) then serve R3 which has third minimum time required (10) and finnaly R1 which has maximum time required (20).

Now, check waiting time for each request to be complete.

1. **R2 : 2 units of time**
2. **R4 : 7 units of time (R4 must wait till R2 complete 2 units and R4 itself requeres 5 units. Total 7 units)**
3. **R3 : 17 units of time (R3 must wait till R4 complete 7 units and R3 itself requeres 10 units. Total 17 units)**
4. **R1 : 37 units of time (R1 must wait till R3 complete 17 units and R1 itself requeres 20 units. Total 37 units)**

Here, average waiting time for all requests (R1, R2, R3 and R4) is (2+7+17+37)/4 ≈ 15 units of time.

From above two situations, it is very clear that, by using second method server can complete all four requests with very less time compared to the first method. This is what exactly done by the priority queue.

Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

There are two types of priority queues they are as follows...

1. **Max Priority Queue**
2. **Min Priority Queue**

1. Max Priority Queue

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

The following are the operations performed in a Max priority queue...

1. **isEmpty() - Check whether queue is Empty.**
2. **insert() - Inserts a new value into the queue.**
3. **findMax() - Find maximum value in the queue.**
4. **remove() - Delete maximum value from the queue.**

Max Priority Queue Representations

There are 6 representations of max priority queue.

1. **Using an Unordered Array (Dynamic Array)**
2. **Using an Unordered Array (Dynamic Array) with the index of the maximum value**
3. **Using an Array (Dynamic Array) in Decreasing Order**
4. **Using an Array (Dynamic Array) in Increasing Order**
5. **Using Linked List in Increasing Order**
6. **Using Unordered Linked List with reference to node with the maximum value**

#1. Using an Unordered Array (Dynamic Array)

In this representation elements are inserted according to their arrival order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.

Now, let us analyse each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at the end of the queue. This operation requires **O(1)** time complexity that means constant time.

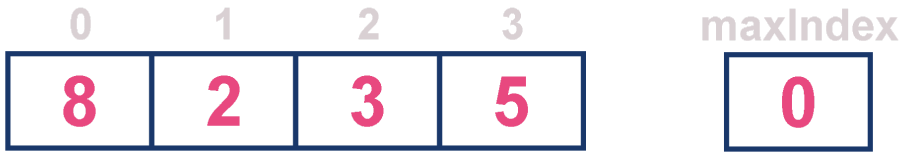
findMax() - To find maximum element in the queue, we need to compare with all the elements in the queue. This operation requires **O(n)** time complexity.

remove() - To remove an element from the queue first we need to perform **findMax()** which requires **O(n)** and removal of particular element requires constant time **O(1)**. This operation requires **O(n)** time complexity.

#2. Using an Unordered Array (Dynamic Array) with the index of the maximum value

In this representation elements are inserted according to their arrival order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at the end of the queue with **O(1)** and for each insertion we need to update maxIndex with **O(1)**. This operation requires **O(1)** time complexity that means constant time.

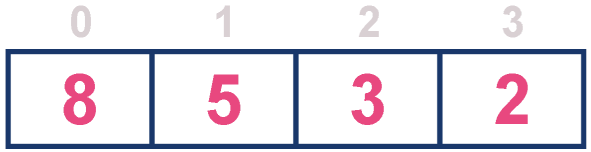
findMax() - To find maximum element in the queue is very simple as maxIndex has maximum element index. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue first we need to perform **findMax()** which requires **O(1)** , removal of particular element requires constant time **O(1)** and update maxIndex value which requires **O(n)**. This operation requires **O(n)** time complexity.

#3. Using an Array (Dynamic Array) in Decreasing Order

In this representation elements are inserted according to their value in decreasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 5, 3 and 2. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at a particular position in the decreasing order into the queue with **O(n)**, because we need to shift existing elements inorder to insert new element in decreasing order. This operation requires **O(n)** time complexity.

findMax() - To find maximum element in the queue is very simple as maximum element is at the beginning of the queue. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue first we need to perform **findMax()** which requires **O(1)**, removal of particular element requires constant time **O(1)** and rearrange remaining elements which requires **O(n)**. This operation requires **O(n)** time complexity.

#4. Using an Array (Dynamic Array) in Increasing Order

In this representation elements are inserted according to their value in increasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.

Now, let us analyse each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at a particular position in the increasing order into the queue with **O(n)**, because we need to shift existing elements inorder to insert new element in increasing order. This operation requires **O(n)** time complexity.

findMax() - To find maximum element in the queue is very simple as maximum element is at the end of the queue. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue first we need to perform **findMax()** which requires **O(1)**, removal of particular element requires constant time **O(1)** and rearrange remaining elements which requires **O(n)**. This operation requires **O(n)** time complexity.

#5. Using Linked List in Increasing Order

In this representation, we use a single linked list to represent max priority queue. In this representation elements are inserted according to their value in increasing order and node with maximum value is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.

Now, let us analyse each operation according to this representation...

isEmpty() - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at a particular position in the increasing order into the queue with **O(n)**, because we need to the position where new element has to be inserted. This operation requires **O(n)** time complexity.

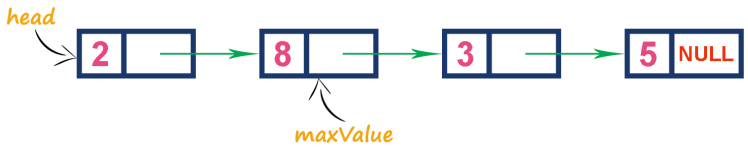
findMax() - To find maximum element in the queue is very simple as maximum element is at the end of the queue. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue is simply removing the last node in the queue which requires **O(1)**. This operation requires **O(1)** time complexity.

#6. Using Unordered Linked List with reference to node with the maximum value

In this representation, we use a single linked list to represent max priority queue. Always we maitain a reference (maxValue) to the node with maximum value. In this representation elements are inserted according to their arrival and node with maximum value is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 8, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity that means constant time.

insert() - New element is added at end the queue with **O(1)** and update maxValue reference with **O(1)**. This operation requires **O(1)** time complexity.

findMax() - To find maximum element in the queue is very simple as maxValue is referenced to the node with maximum value in the queue. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue is deleting the node which referenced by maxValue which requires **O(1)** and update maxValue reference to new node with maximum value in the queue which requires **O(n) time complexity**. This operation requires **O(n)** time complexity.

2. Min Priority Queue Representations

Min Priority Queue is similar to max priority queue except removing maximum element first, we remove minimum element first in min priority queue.

The following operations are performed in Min Priority Queue...

1. **isEmpty()** - Check whether queue is Empty.
2. **insert()** - Inserts a new value into the queue.
3. **findMin()** - Find minimum value in the queue.
4. **remove()** - Delete minimum value from the queue.

Min priority queue is also has same representations as Max priority queue with minimum value removal.

← Previous

Next →

About Us ¦ Contact Us

Website designed by RAJINIKANTH B