Priority Queue

A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a difference that the logical order of elements in the priority queue depends on the priority of the elements. The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue.

In some implementations, if two elements have the same priority, they are served according to the order in which they were enqueued, while in some implementations, ordering of elements with the same priority is undefined.

A typical priority queue supports following operations (depending on max-heap or min-heap implementation):

- *isEmpty()*: check whether the queue has no elements
- *insert(item, priority)*: Inserts an item with given priority.
- *getMax() / getMin()*: which returns the highest-priority / lowest-priority element but does not modify the queue (peek() in queue).
- *extractMax() / extractMin()*: Removes and returns the highest priority / lowest-priority element.

Implementations of priority queues

- *Implementation using single linked list / double linked list*

> The list is so created so that the highest priority element is always at the head of the list. The list is arranged in descending order of elements based on their priority. This allow us to remove the highest priority element (extractMax() and getMax()) in O(1) time.
> To insert an element we must traverse the list and find the proper position to insert the node so that the overall order of the priority queue is maintained. This makes the push() operation takes O(N) time.

Heap is generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues.

Among *other applications*, we can use max-priority queues to schedule jobs on a shared computer. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling extractMax(). A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program calls EXTRACT-MIN at each step to choose the next event to simulate.

In a *Binary Heap*, getMax() can be implemented in O(1) time, insert() can be implemented in O(Log n) time and extractMax() can also be implemented in O(Log n) time.

Extra info:
With *Fibonacci Heap*, insert() and getMax() can be implemented in O(1) amortized time and extractMax() can be implemented in O(Logn) amortized time.

Binary Heap:
Using Arrays for storing binary heap which is a complete binary trees, there will not be any wastage of locations.

Binary heap is typically represented as an array. The traversal method used to achieve array representation is *Level Order*

```
The root element will be at Arr[0].
Below shows indexes of other nodes for the ith node, i.e., Arr[i]:
Arr[(i - 1) / 2] Returns the parent node
Arr[(2 * i) + 1] Returns the left child node
Arr[(2 * i) + 2] Returns the right child node
```

Implementation of PQ in java language

There is no priority queue interface built into Java, but Java does include a class, java.util.PriorityQueue, which implements the java.util.Queue interface. The "front" of the queue will always be a minimal element, with priorities based either on the natural ordering of the elements, or in accordance with a comparator object sent as a parameter when constructing the priority queue.

The most notable difference between the java.util.PriorityQueue class and our own priority queue ADT is the model for managing keys and values. Whereas our public interface distinguishes between keys and values, the java.util.PriorityQueue class relies on a single element type. That element is effectively treated as a key.

```
PriorityQueueInJava.java //Java language implementation

HeapPriorityQueue.java // ADT based implementation
```

Application of priority queue

An important sorting algorithm known as *heap-sort* also follows naturally from our heap based priority-queue implementations.
Priority queue is used as a basis for a sorting algorithm by inserting a sequence of items, then successively removing the smallest to get them out, in order.

Few of the Priority queues applications are listed below:

- Data compression: Huffman Coding algorithm
- Shortest path algorithms: Dijkstra's algorithm
- Minimum spanning tree algorithms: Prim's algorithm
- Event-driven simulation: customers in a line
- Selection problem: Finding kth- smallest element

Time Complexity of building a heap

The analysis of the running time of the methods is based on the following:

- The heap T has n nodes, each storing a reference to a key-value entry.
- The height of heap T is O(log n), since T is complete.
- The min operation runs in O(1) because the root of the tree contains such an element.
- Locating the last position of a heap, as required for insert and removeMin, can be performed in O(1) time for an array-based representation, or O(log n) time for a linked-tree representation.
- In the worst case, up-heap and down-heap bubbling perform a number of swaps equal to the height of T.

Methods	Running Time
size(), isEmpty()	O(1)
getMin()	O(1)
insert()	O(Log(n))*
extractMin()	O(Log(n))*

> * amortized, if using dynamic array

We let 'n' denote the number of entries in the priority queue at the time an operation is executed. The space requirement is O(n). The running time of operations getMin() and extractMin() are amortized for an array-based representation, due to occasional resizing of a dynamic array; those bounds are worst case with a linked tree structure.