|  |  |
| --- | --- |
| Implement Queue using Stack |  |
| **Queue Data Structure** |  |
| **Operations on Queue:** |  |
| **Applications of Queue:** | Queue is used when things don’t have to be processed immediatly, but have to be processed in **F**irst **I**n**F**irst **O**ut order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search).  This property of Queue makes it also useful in following kind of scenarios.   1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.   **2)**When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc. |
| **Applications of Priority Queue** | **A Priority Queue is different from a normal queue, because instead of being a “first-in-first-out”, values come out in order by priority.**  It is an abstract data type that captures the idea of a container whose elements have “priorities” attached to them.  An element of highest priority always appears at the front of the queue.  If that element is removed, the next highest priority element advances to the front.  [Heap Sort](https://www.geeksforgeeks.org/heap-sort/) : Heap sort is typically implemented using Heap which is an implementation of Priority Queue.  [Operating systems](https://en.wikipedia.org/wiki/Operating_system): It is also use in Operating System for [load balancing](https://en.wikipedia.org/wiki/Load_balancing_(computing)) ([load balancing on server](https://www.geeksforgeeks.org/load-balancing-on-servers-random-algorithm/)), [interrupt handling](https://practice.geeksforgeeks.org/problems/interrupt-handlers).  [A\* Search Algorithm](https://www.geeksforgeeks.org/a-search-algorithm/) : The A\* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first.  [Data compression](https://en.wikipedia.org/wiki/Data_compression): It is used in [Huffman codes](https://www.geeksforgeeks.org/tag/huffman-coding/) which is used to compresses data. |
|  | int size(): to get the number of elements in the Set.  boolean isEmpty(): to check if Set is empty or not.  boolean contains(Object o): Returns true if this Set contains the specified element.  Iterator iterator(): Returns an iterator over the elements in this set. The elements are returned in no particular order.  boolean removeAll(Collection c): Removes from this set all of its elements that are contained in the specified collection (optional operation).  boolean retainAll(Collection c): Retains only the elements in this set that are contained in the specified collection (optional operation).  void clear(): Removes all the elements from the set.  E remove(): Retrieves and removes the head of this queue.  E poll(): Retrieves and removes the head of this queue, or returns null if this queue is empty.  E peek(): Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.  boolean offer(E e): Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.  ­­E element(): Retrieves, but does not remove, the head of this queue.  boolean add(E e): Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.  Object[] toArray(): Returns an array containing all of the elements in this set. If this set makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order. |
|  | **class** QNode {  **int** value;  QNode next;  **public** QNode(**int** value) {  **this**.value = value;  **this**.next = **null**;  }  }  **class** Queue {  QNode front, rear;  **public** Queue() {  **this**.front = **this**.rear = **null**;  }  **void** enqueue(**int** value) {  QNode temp = **new** QNode(value);  **if** (**this**.rear == **null**) {  **this**.front = **this**.rear = temp;  **return**;  } **else** {  **this**.rear.next = temp;  **this**.rear = temp;  }  }  **void** dequeue() {  **if** (**this**.front == **null**)  **return**;  QNode temp = **this**.front;  **this**.front = **this**.front.next;  **if** (**this**.front == **null**)  **this**.rear = **null**;  }  } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |